DESIGN FOR EMBEDDED IMAGE PROCESSING ON FPGAS

DESIGN FOR EMBEDDED IMAGE PROCESSING ON FPGAS

Donald G. Bailey
Massey University, New Zealand

IEEE

John Wiley & Sons (Asia) Pte Ltd

Library of Congress Cataloging-in-Publication Data

Bailey, Donald G. (Donald Graeme), 1962-
  Design for embedded image processing on FPGAs / Donald G. Bailey.
      p. cm.
    Includes bibliographical references and index.
  ISBN 978-0-470-82849-6 (hardback)
 1. Embedded computer systems. 2. Field programmable gate arrays. I. Title.
  TK7895.E42B3264 2011
  621.39'9–dc22   2011002991

Print ISBN: 978-0-470-82849-6
ePDF ISBN: 978-0-470-82850-2
oBook ISBN: 978-0-470-82851-9
ePub ISBN: 978-0-470-82852-6
Mobi ISBN: 978-1-118-07331-5

Set in 9/11 pt Times New Roman by Thomson Digital, Noida, India

Contents

Preface

I think it is useful to provide a little background as to why and how this book came into being. This will perhaps provide some insight into the way the material is structured, and why it is presented in the way that it is.

Background

Firstly, a little bit of history. I have an extensive background in image processing, particularly in the areas of image analysis, machine vision and robot vision, all strongly application-orientated areas. With over 25 years of applying image processing techniques to a wide range of problems, I have gained considerable experience in algorithm development. This is not only at the image processing application level but also at the image processing operation level. My approach to an application has usually been more pragmatic than theoretical – I have focussed on developing image processing algorithms that solved the problem at hand. Often this involved assembling sequences of existing image processing operations, but occasionally it required developing new algorithms and techniques to solve particular aspects of the problem. Through work on machine vision and robotics applications, I have become aware of some of the limitations of software-based solutions, particularly in terms of speed and algorithm efficiency.

This led naturally to considering FPGAs as an implementation platform for embedded imaging applications. Many image processing operations are inherently parallel and FPGAs provide programmable hardware, also inherently parallel. Therefore, it should be as simple as mapping one onto the other, right? Well, when I started implementing image processing algorithms on FPGAs, I had lots of ideas, but very little knowledge. I very soon found that there were a lot of tricks that were needed to create an efficient design. Consequently, my students and I learned many of these the hard way, through trial and error.

With my basic training as an electronics engineer, I was readily able to adapt to the hardware mindset. I have since discovered through observing my students, both at the undergraduate and postgraduate level, that this is perhaps the biggest hurdle to an efficient implementation. Image processing is traditionally thought of as a software domain task, whereas FPGA-based design is firmly in the hardware domain. To bridge the gap, it is necessary to think of algorithms not on their own but more in terms of their underlying computational architecture.

Implementing an image processing algorithm (or indeed any algorithm) on an FPGA, therefore, consists of determining the underlying architecture of an algorithm, mapping that architecture onto the resources available within an FPGA and finally mapping the algorithm onto the hardware architecture. Unfortunately, there is very little material available to help those new to the area to get started. Even this insight into the process is not actually stated anywhere, although it is implicitly followed (whether consciously or not) by most people working in this area.

Available Literature

While there are many research papers published in conference proceedings and journals, there are only a few that focus specifically on how to map image processing algorithms onto FPGAs. The research papers found in the literature can be classified into several broad groups.

The first focuses on the FPGA architecture itself. Most of these provide an analysis of a range of techniques relating to the structure and granularity of logic blocks, the routing networks and embedded memories. As well as the FPGA structure, a wide range of topics is covered, including underlying technology, power issues, the effects of process variability and dynamic reconfigurability. Many of these papers are purely proposals or relate to prototype FPGAs rather than commercially available chips. Although such papers are interesting in their own right and represent perfectly legitimate research topics, very few of these papers are directly useful from an applications point of view. While they provide insights into some of the features which might be available in the next generation of devices, most of the topics within this group are at too low a level.

A second group of papers investigates the topic of reconfigurable computing. Here the focus is on how an FPGA can be used to accelerate some computationally intensive task or range of tasks. While image processing is one such task considered, most of the research relates more to high performance (and high power) computing rather than low power embedded systems. Topics within this group include hardware and software partitioning, hardware and software co-design, dynamic reconfigurability, communication between an FPGA and CPU, comparisons between the performance of FPGAs, GPUs and CPUs, and the design of operating systems and specific platforms for both reconfigurable computing applications and research. Important principles and techniques can be gleaned from many of these papers, even though this may not be their primary focus.

The next group of papers is closely related to the previous group and considers tools for programming FPGAs and applications. The focus here is more on improving the productivity of the development process. A wide range of hardware description languages have been proposed, with many modelled after software languages such as C, Java and even Prolog. Many of these are developed as research tools, with very few making it out of the laboratory to commercial availability. There has also been considerable research on compilation techniques for mapping standard software languages to hardware. Such compilers attempt to exploit techniques such as loop unrolling, strip mining and pipelining to produce parallel hardware. Again, many of these papers describe important principles and techniques that can result in more efficient hardware designs. However, current compilers are still relatively immature in the level and kinds of parallelism that they can automatically exploit. They are also limited in that they can only perform relatively simple transformations to the algorithm provided; they cannot redesign the underlying algorithm.

The final group of papers focuses on a range of applications, including image processing and the implementation of both image processing operations and systems. Unfortunately, as a result of page limits and space constraints, many of these papers give the results of the implementation of various systems, but present relatively few design details. Often the final product is described, without describing many of the reasons or decisions that led to that design. Many of these designs cannot be recreated without acquiring the specific platform and tools that were used, or inferring a lot of the missing details. While some of these details may appear obvious in hindsight, without this knowledge many were far from obvious just from reading the papers. The better papers in this group tended to have a tighter focus, considering the implementation of a single image processing operation.

So while there may be a reasonable amount of material available, it is quite diffuse. In many cases, it is necessary to know exactly what you are looking for, or just be lucky to find it.

Shortly after beginning in this area, my research students and I wrote down a list of topics and techniques that we would have liked to have known when we started. As we progressed, our list grew. Our intention from the start was to compile this material into a book to help others who, like us, were having to learn things the hard way by themselves. Essentially, this book reflects our distilled experiences in this field, combined with techniques (both FPGA design and image processing) that have been gleaned from the literature.

Intended Audience

This book is written primarily for those who are familiar with the basics of image processing and want to consider implementing image processing using FPGAs. It accomplishes this by presenting the techniques and approaches that we wished we knew when we were starting in this area. Perhaps the biggest hurdle is switching from a software mindset to a hardware way of thinking. Very often, when programming software, we do so without great consideration of the underlying architecture. Perhaps this is because the architecture of most software processors is sufficiently similar that any differences are really only a second order effect, regardless of how significant they may appear to a computer engineer. A good compiler is able to map the algorithm in the programming language onto the architecture relatively efficiently, so we can get away without thinking too much about such things. When programming hardware though, architecture is everything. It is not simply a matter of porting the software onto hardware. The underlying hardware architecture needs to be designed as well. In particular, programming hardware usually requires transforming the algorithm into an appropriate parallel architecture, often with significant changes to the algorithm itself. This is not something that the current generation of compilers is able to do because it requires significant design rather than just decomposition of the dataflow. This book addresses this issue by providing not only algorithms for image processing operations, but also underlying architectures that can be used to implement them efficiently.

This book would also be useful to those who are familiar with programming and applying FPGAs to other problems and are considering image processing applications. While many of the techniques are relevant and applicable to a wide range of application areas, most of the focus and examples are taken from image processing applications. Sufficient detail is given to make many of the algorithms and their implementation clear. However, I would argue that learning image processing is more than just collecting a set of algorithms, and there are any number of excellent image processing texts that provide these. Imaging is a practical discipline that can be learned most effectively by doing, and a software environment provides a significantly greater flexibility and interactivity than learning image processing via FPGAs.

That said, it is in the domain of embedded image processing where FPGAs come into their own. An efficient, low power design requires that the techniques of both the hardware engineer and the software engineer be integrated tightly within the final solution.

Outline of the Contents

This book aims to provide a comprehensive overview of algorithms and techniques for implementing image processing algorithms on FPGAs, particularly for low and intermediate level vision. However, as with design in any field, there is more than one way of achieving a particular task. Much of the emphasis has been placed on stream-based approaches to implementing image processing, as these can efficiently exploit parallelism when they can be used. This emphasis reflects my background and experience in the area, and is not intended to be the last word on the topic.

A broad overview of image processing is presented in Chapter 1, with a brief historical context. Many of the basic image processing terms are defined and the different stages of an image processing algorithm are identified and illustrated with an example algorithm. The problem of real-time embedded image processing is introduced, and the limitations of conventional serial processors for tackling this problem are identified. High speed image processing must exploit the parallelism inherent in the processing of images. A brief history of parallel image processing systems is reviewed to provide the context of using FPGAs for image processing.

FPGAs combine the advantages of both hardware and software systems, by providing reprogrammable (hence flexible) hardware. Chapter 2 provides an introduction to FPGA technology. While some of this will be more detailed than is necessary to implement algorithms, a basic knowledge of the building blocks and underlying architecture is important to developing resource efficient solutions. The key features of currently available FPGAs are reviewed in the context of implementing image processing algorithms.

FPGA-based design is hardware design, and this hardware needs to be represented using some form of hardware description language. Some of the main languages are reviewed in Chapter 3, with particular emphasis on the design flow for implementing algorithms. Traditional hardware description languages such as VHDL and Verilog are quite low level in that all of the control has to be explicitly programmed. The last 15 years has seen considerable research into more algorithm approaches to programming hardware, based primarily on C. An overview of some of this research is presented, finishing with a brief description of a number of commercial offerings.

The process of designing and implementing an image processing application on an FPGA is described in detail in Chapter 4. Particular emphasis is given to the differences between designing for an FPGA-based implementation and a standard software implementation. The critical initial step is to clearly define the image processing problem that is being tackled. This must be in sufficient detail to provide a specification that may be used to evaluate the solution. The procedure for developing the image processing algorithm is described in detail, outlining the common stages within many image processing algorithms. The resulting algorithm must then be used to define the system and computational architectures. The mapping from an algorithm is more than simply porting the algorithm to a hardware description language. It is necessary to transform the algorithm to make efficient use of the resources available on the FPGA. The final stage is to implement the algorithm by mapping it onto the computational architecture.

Three types of constraints on the mapping process are: limited processing time, limited access to data and limited system resources. Chapter 5 describes several techniques for overcoming or alleviating these constraints. Possible FPGA implementations are described of several data structures commonly found in computer vision algorithms. These help to bridge the gap between a software and hardware implementation. Number representation and number systems are described within the context of image processing. A range of efficient hardware computational techniques is discussed. Some of these techniques could be considered the hardware equivalent of software libraries for efficiently implementing common functions.

The next section of this book describes the implementation of many common image processing operations. Some of the design decisions and alternative ways of mapping the operations onto FPGAs are considered. While reasonably comprehensive, particularly for low level image-to-image transformations, it is impossible to cover every possible design. The examples discussed are intended to provide the foundation for many other related operations.

Chapter 6 considers point operations, where the output depends only on the corresponding input pixel in the input image(s). Both direct computation and lookup table approaches are described. With multiple input images, techniques such as image averaging and background subtraction are discussed in detail. The final section in this chapter extends the earlier discussion to the processing of colour images. Particular topics given emphasis are colour space conversion, colour segmentation and colour balancing.

The implementation of histograms and histogram-based processing are discussed in Chapter 7. Techniques of accumulating a histogram and then extracting data from the histogram are described in some detail. Particular tasks are histogram equalisation, threshold selection and using histograms for image matching. The concepts of standard one-dimensional histograms are extended to multidimensional histograms. The use of clustering for colour segmentation and classification is discussed in some detail. The chapter concludes with the use of features extracted from multidimensional histograms for texture analysis.

Chapter 8 focuses considers a wide range of local filters, both linear and nonlinear. Particular emphasis is given to caching techniques for a stream-based implementation and methods for efficiently handling the processing around the image borders. Rank filters are described and a selection of associated sorting network architectures reviewed. Morphological filters are another important class of filters. State machine implementations of morphological filtering provide an alternative to the classic filter implementation. Separability and both serial and parallel decomposition techniques are described that enable more efficient implementations.

Image warping and related techniques are covered in Chapter 9. The forward and reverse mapping approaches to geometric transformation are compared in some detail, with particular emphasis on techniques for stream processing implementations. Interpolation is frequently associated with geometric transformation. Hardware-based algorithms for bilinear, bicubic and spline based interpolation are described. Related techniques of image registration are also described at the end of this chapter, including a discussion of the scale invariant feature transform and super-resolution.

Chapter 10 introduces linear transforms, with a particular focus on the fast Fourier transform, the discrete cosine transform and the wavelet transform. Both parallel and pipelined implementations of the FFT and DCT are described. Filtering and inverse filtering in the frequency domain are discussed in some detail. Lifting-based filtering is developed for the wavelet transform. This can reduce the logic requirements by up to a factor of four over a direct finite impulse response implementation. The final section in this chapter discusses the stages within image and video coding, and outlines some of the techniques that can be used at each stage.

A selection of intermediate level operations relating to region detection and labelling is presented in Chapter 11. Standard software algorithms for chain coding and connected component labelling are adapted to give efficient streamed implementation. These can significantly reduce both the latency and memory requirements of an application. Hardware implementaions of the distance transform, the watershed transform and the Hough transform are also described.

Any embedded application must interface with the real world. A range of common peripherals is described in Chapter 12, with suggestions on how they may be interfaced to an FPGA. Particular attention is given to interfacing cameras and video output devices, although several other user interface and memory devices are described. Image processing techniques for deinterlacing and Bayer pattern demosaicing are reviewed.

The next chapter expands some of the issues with regard to testing and tuning that were introduced earlier. Four areas are identified where an implementation might not behave in the intended manner. These are faults in the design, bugs in the implementation, incorrect parameter selection and not meeting timing constraints. Several checklists provide a guide and hints for testing and debugging an algorithm on an FPGA.

Finally, a selection of case studies shows how the material and techniques described in the previous chapters can be integrated within a complete application. These applications briefly show the design steps and illustrate the mapping process at the whole algorithm level rather than purely at the operation level. Many gains can be made by combining operations together within a compatible overall architecture. The applications described are coloured region tracking for a gesture-based user interface, calibrating and correcting barrel distortion in lenses, development of a foveal image sensor inspired by some of the attributes of the human visual system, the processing to extract the range from a time of flight range imaging system, and a machine vision system for real-time produce grading.

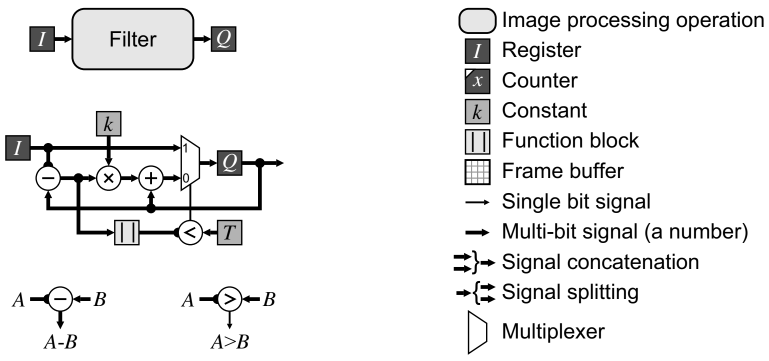


Figure P.1 Conventions used in this book. Top left: representation of an image processing operation; middle left: a block schematic representation of the function given by Equation P.1; bottom left: representation of operators where the order of operands is important. Right: symbols used for various blocks within block schematics.

Conventions Used

The contents of this book are independent of any particular FPGA or FPGA vendor, or any particular hardware description language. The topic is already sufficiently specialised without narrowing the audience further! As a result, many of the functions and operations are represented in block schematic form. This enables a language independent representation, and places emphasis on a particular hardware implementation of the algorithm in a way that is portable. The basic elements of these schematics are illustrated in Figure P.1. I is generally used as the input of an image processing operation, with the output image represented by Q.

With some mathematical operations, such as subtraction and comparison, the order of the operands is important. In such cases, the first operand is indicated with a blob rather than an arrow, as shown on the bottom in Figure P.1.

Consider a recursive filter operating on streamed data:

$$Q_n = \begin{cases} I_n, & |I_n - Q_{n-1}| < T \\ Q_{n-1} + k(I_n - Q_{n-1}), & \text{otherwise} \end{cases} \tag{P.1}$$

where the subscript in this instance refers to the nth pixel in the streamed image. At a high level, this can be considered as an image processing operation and represented by a single block, as shown in the top left of Figure P.1. The low level implementation is given in the middle left panel. The input and output, I and Q, are represented by registers – dark blocks, with optional register names in white; the subscripts have been dropped because they are implicit with streamed operation. In some instances additional control inputs may be shown: CE for clock enable, RST for reset, and so on. Constants are represented as mid-grey blocks and other function blocks with light grey background.

When representing logic functions in equations, $\vee$ is used for logical OR and $\wedge$ for logical AND. This is to avoid confusion with addition and multiplication.

Acknowledgements

I would like to acknowledge all those who have helped me to get me where I currently am in my understanding of FPGA-based design. In particular, I would like to thank my research students (David Johnson, Kim Gribbon, Chris Johnston, Aaron Bishell, Andreas Buhler and Ni Ma) who helped to shape my thinking and approach to FPGA development as we struggled together to work out efficient ways of implementing image processing algorithms. This book is as much a reflection of their work as it is of mine.

Most of our algorithms were programmed for FPGAs using Handel-C and were tested on boards provided by Celoxica Ltd. I would like to acknowledge the support provided by Roger Gook and his team, originally with the Celoxica University Programme, and later with Agility Design Solutions. Roger provided heavily discounted licences for the DK development suite, without which many of the ideas presented in this book would not have been as fully explored.

Massey University has provided a supportive environment and the freedom for me to explore this field. In particular, Serge Demidenko gave me the encouragement and the push to begin playing with FPGAs. Since that time, he has been a source of both inspiration and challenging questions. Other colleagues who have been of particular encouragement are Gourab Sen Gupta and Richard Harris. I would also like to acknowledge Paul Lyons, who co-supervised a number of my students.

Early versions of some of the material in this book were presented as half-day tutorials at the IEEE Region 10 Conference (TenCon) in 2005 in Melbourne, Australia, the IEEE International Conference on Image Processing (ICIP) in 2007 in San Antonio, Texas, USA, and the 2010 Asian Conference on Computer Vision (ACCV) in Queenstown, New Zealand. I would like to thank attendees at these workshops for providing valuable feedback and stimulating discussion.

During 2008, I spent a sabbatical with the Circuits and Systems Group at Imperial College London, UK. I am grateful to Peter Cheung, who hosted my visit, and provided a quiet office, free from distractions and interruptions. It was here that I actually began writing, and got most of the text outlined at least. I would particularly like to thank Peter Cheung, Christos Bouganis, Peter Sedcole and George Constantinides for discussions and opportunities to bounce ideas off.

My wife, Robyn, has given me the freedom of many evenings and weekends over the two years since then to complete this manuscript. I am grateful for both her patience and her support. She now knows that field programmable gate arrays are not alligators with ray guns stalking the swamp. This book is dedicated to her.

Donald Bailey

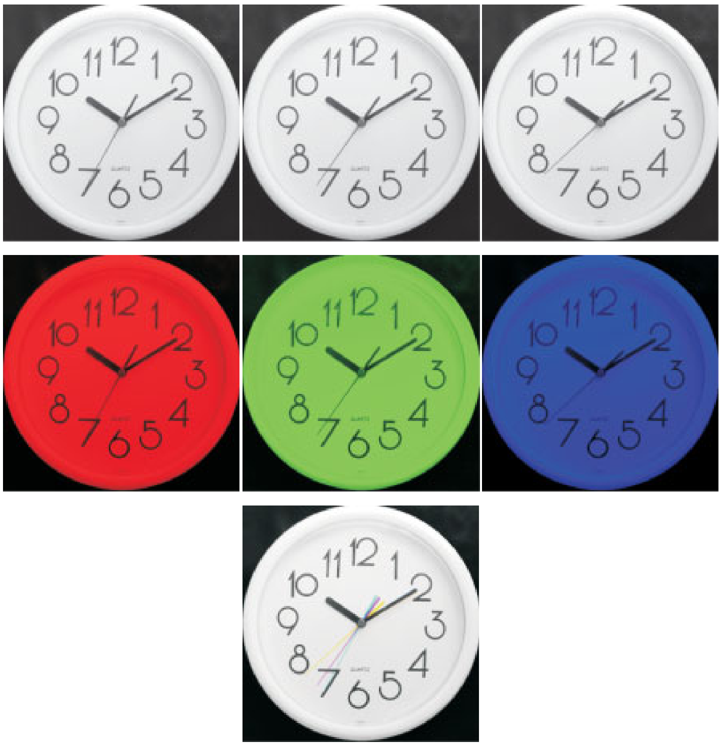


Figure 6.28 Temporal false colouring. Images taken at different times are assigned to different channels, with the resultant output showing coloured regions where there are temporal differences.

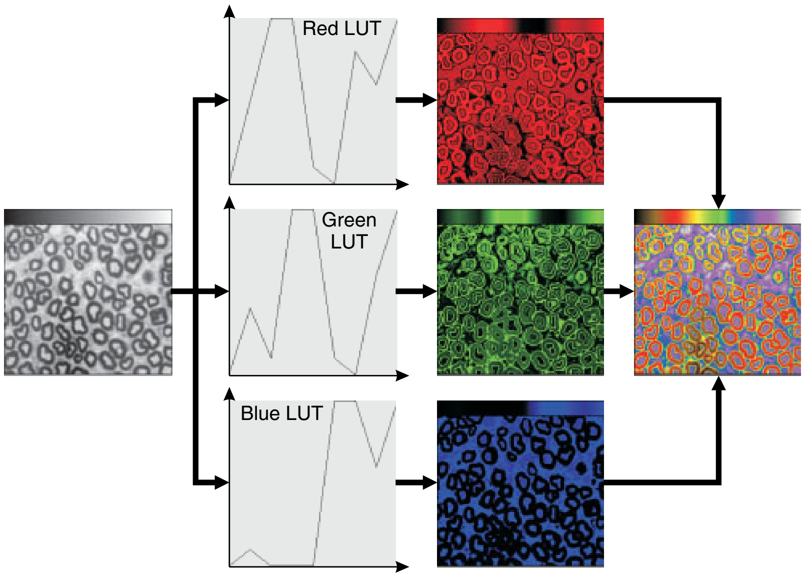


Figure 6.29 Pseudocolour or false colour mapping using lookup tables.

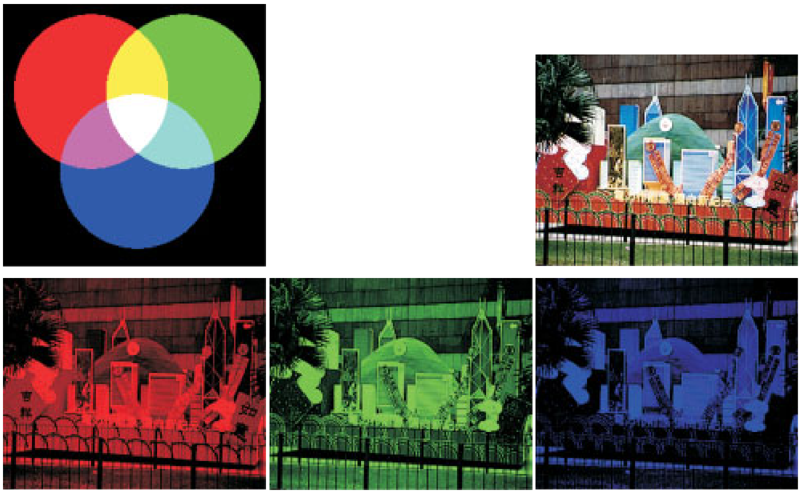


Figure 6.30 RGB colour space. Top left: combining red, green and blue primary colours; bottom: the red, green and blue components of the colour image on the top right.

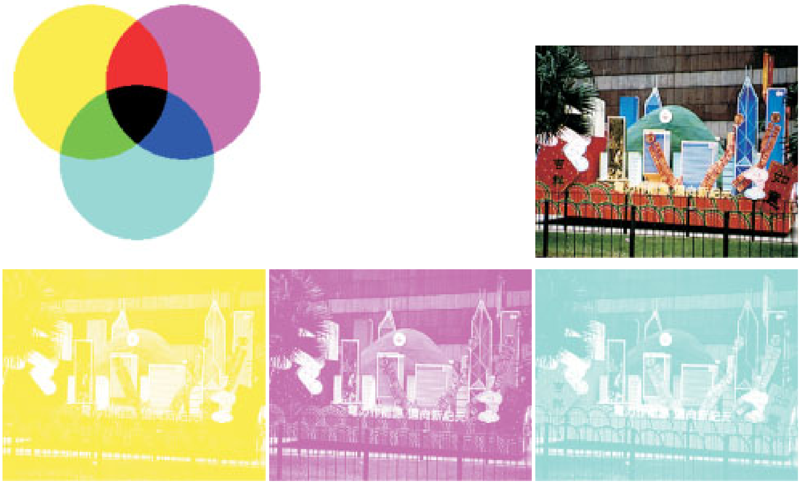


Figure 6.32 CMY colour space. Top left: combining yellow, magenta and cyan secondary colours; bottom: the yellow, magenta and cyan components of the colour image on the top right.

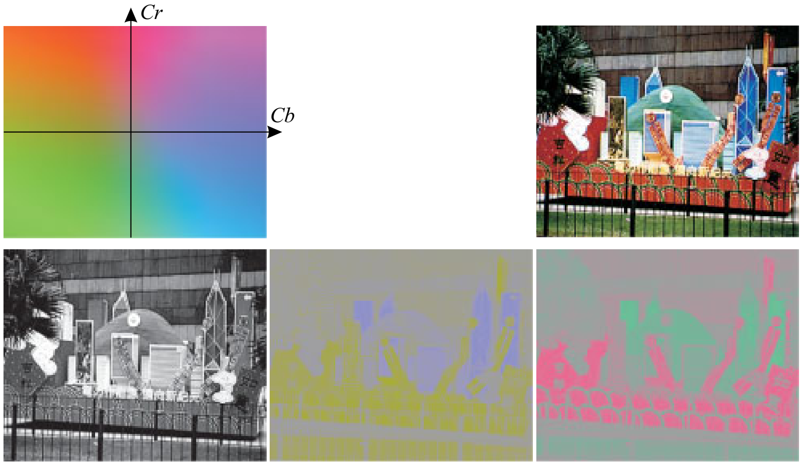


Figure 6.34 YCbCr colour space. Top left: the *Cb*−*Cr* colour plane at mid luminance; bottom: the luminance and chrominance components of the colour image on the top right.

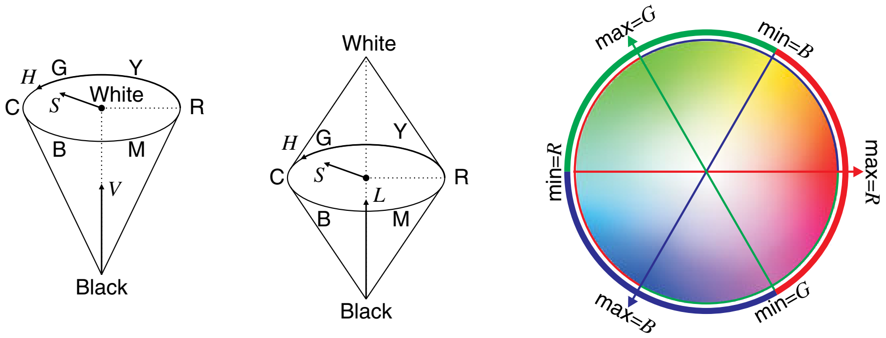


Figure 6.36 HSV and HLS colour spaces. Left: the HSV cone; centre: the HLS bi-cone; right: the hue colour wheel.

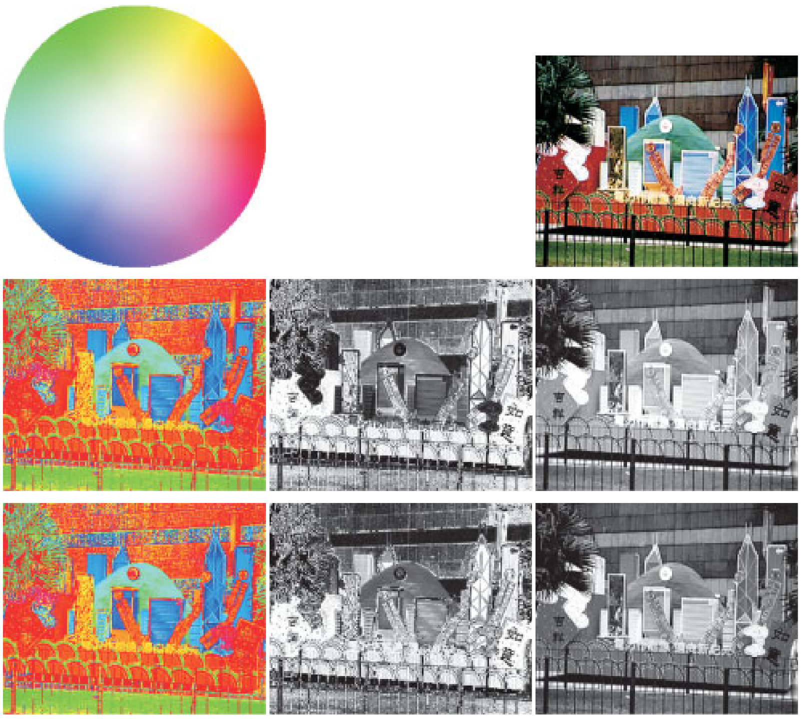


Figure 6.37 HSV and HLS colour spaces. Top left: HSV hue colour wheel, with saturation increasing with radius; middle row: the HSV hue, saturation and value components of the colour image on the top right; bottom row: the HLS hue, saturation and lightness components.

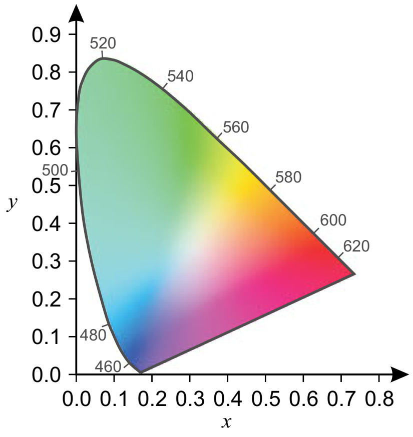


Figure 6.40 Chromaticity diagram. The numbers are wavelengths of monochromatic light in nanometres.

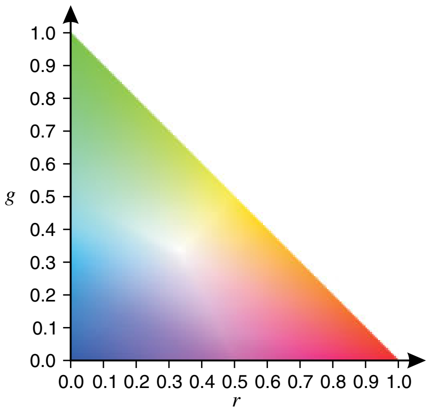


Figure 6.41 Device dependent $r-g$ chromaticity.

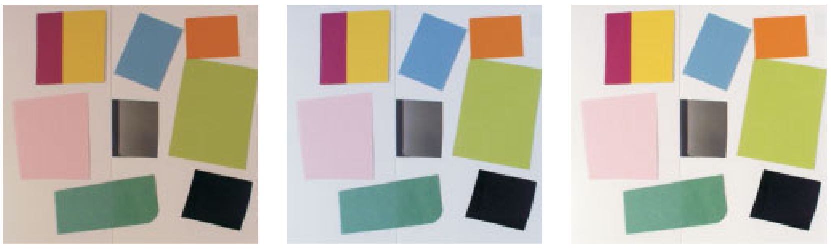


Figure 6.43 Simple colour correction. Left: original image captured under incandescent lights, resulting in a yellowish-red cast; centre: correcting assuming the average is grey, using Equation 6.86; right: correcting assuming the brightest pixel is white, using Equation 6.88.

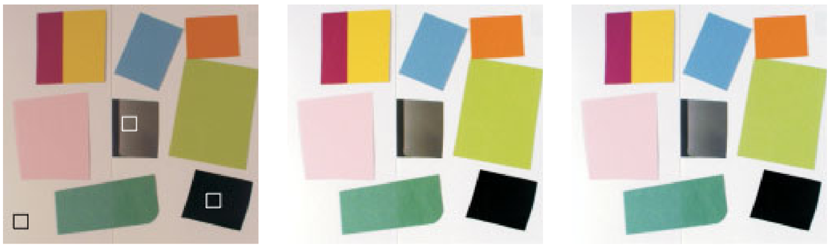


Figure 6.44 Correcting using black, white and grey patches. Left: original image with the patches marked; centre: stretching each channel to correct for black and white, using Equation 6.90; right: adjusting the gamma of the red and blue channels using Equation 6.91 to make the grey patch grey.

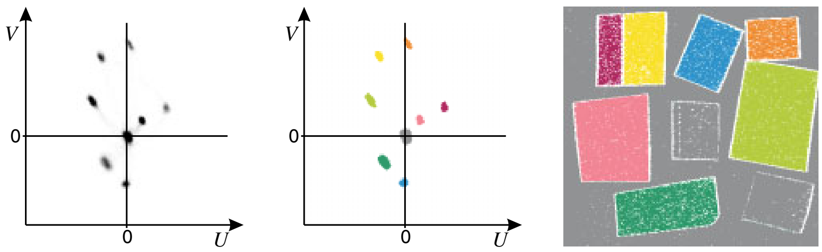


Figure 7.24 Using a two-dimensional histogram for colour segmentation. Left: $U-V$ histogram using Equation 6.61; centre: after thresholding and labelling, used as a two-dimensional lookup table; right: segmented image.

1

Image Processing

Vision is arguably the most important human sense. The processing and recording of visual data therefore has significant importance. The earliest images are from prehistoric drawings on cave walls or carved on stone monuments commonly associated with burial tombs. (It is not so much the medium that is important here – anything else would not have survived to today). Such images consist of a mixture of both pictorial and abstract representations. Improvements in technology enabled images to be recorded with more realism, such as paintings by the masters. Images recorded in this manner are indirect in the sense that the light intensity pattern is not used directly to produce the image. The development of chemical photography in the early 1800s enabled direct image recording. This trend has continued with electronic recording, first with analogue sensors, and subsequently with digital sensors, which include the analogue to digital (A/D) conversion on the sensor chip.

Imaging sensors have not been restricted to the portion of the electromagnetic spectrum visible to the human eye. Sensors have been developed to cover much of the electromagnetic spectrum from radio waves through to X-rays and gamma rays. Other imaging modalities have been developed, including ultrasound, and magnetic resonance imaging. In principle, any quantity that can be sensed can be used for imaging – even dust rays (Auer, 1982).

Since vision is such an important sense, the processing of images has become important too, to augment or enhance human vision. Images can be processed to enhance their subjective content, or to extract useful information. While it is possible to process the optical signals that produce the images directly by using lenses and optical filters, it is digital image processing – the processing of images by computer – that is the focus of this book.

One of the earliest applications of digital image processing was for transmitting digitised newspaper pictures across the Atlantic Ocean in the early 1920s (McFarlane, 1972). However, it was only with the advent of digital computers with sufficient memory and processing power that digital image processing became more widespread. The earliest recorded computer-based image processing was from 1957, when a scanner was added to a computer at the National Bureau of Standards in the USA (Kirsch, 1998). It was used for some of the early research on edge enhancement and pattern recognition. In the 1960s, the need for processing large numbers of large images obtained from satellites and space exploration stimulated image processing research at NASA's Jet Propulsion Laboratory (Castleman, 1979). In parallel with this, research in high energy particle physics led to a large number of cloud chamber photographs that had to be interpreted to detect interesting events (Duff, 2000). As computers grew in power and reduced in cost, there was an explosion in the range of applications for digital image processing, from industrial inspection, to medical imaging.

1.1 Basic Definitions

More formally, an *image* is a spatial representation of an object, scene or other phenomenon (Haralick and Shapiro, 1991). Examples of images include: a photograph, which is a pictorial record formed from the light intensity pattern on an optical sensor; a radiograph, which is a representation of density formed through exposure to X-rays transmitted through an object; a map, which is a spatial representation of physical or cultural features; a video, which is a sequence of two-dimensional images through time. More rigorously, an image is any continuous function of two or more variables defined on some bounded region of a plane.

Such a definition is not particularly useful in terms of computer manipulation. A *digital image* is an image in digital format, so that it is suitable for processing by computer. There are two important characteristics of digital images. The first is spatial quantisation. Computers are unable to easily represent arbitrary continuous functions, so the continuous function is sampled. The result is a series of discrete picture elements, or *pixels*, for two-dimensional images, or volume elements, *voxels*, for three-dimensional images. Sampling can result in an exact representation (in the sense that the underlying continuous function may be recovered exactly) given a band-limited image and a sufficiently high sample rate. The second characteristic of digital images is sample quantisation. This results in discrete values for each pixel, enabling an integer representation. Common bit widths per pixel are 1 (binary images), 8 (greyscale images), and 24 (3×8 bits for colour images). Unlike sampling, value quantisation will always result in an error between the representation and true value. In many circumstances, however, this *quantisation error* or *quantisation noise* may be made smaller than the uncertainty in the true value resulting from inevitable measurement noise.

In its basic form, a digital image is simply a two (or higher) dimensional array of numbers (usually integers) which represents an object, or scene. Once in this form, an image may be readily manipulated by a digital computer. It does not matter what the numbers represent, whether light intensity, reflectance, distance to a point (or range), temperature, population density, elevation, rainfall, or any other numerical quantity.

Image processing can therefore be defined as subjecting such an image to a series of mathematical operations in order to obtain a desired result. This may be an enhanced image; the detection of some critical feature or event; a measurement of an object or key feature within the image; a classification or grading of objects within the image into one of two or more categories; or a description of the scene.

Image processing techniques are used in a number of related fields. While the principle focus of the fields often differs, at the fundamental level many of the techniques remain the same. Some of the distinctive characteristics are briefly outlined here.

Digital image processing is the general term used for the processing of images by computer in some way or another.

Image enhancement involves improving the subjective quality of an image, or the detectability of objects within the image (Haralick and Shapiro, 1991). The information that is enhanced is usually apparent in the original image, but may not be clear. Examples of image enhancement include noise reduction, contrast enhancement, edge sharpening and colour correction.

Image restoration goes one step further than image enhancement. It uses knowledge of the causes of the degradation present in an image to create a model of the degradation process. This model is then used to derive an inverse process that is used to restore the image. In many cases, the information in the image has been degraded to the extent of being unrecognisable, for example severe blurring.

Image reconstruction involves restructuring the data that a available into a more useful form. Examples are image super-resolution (reconstructing a high resolution image from a series of low resolution images) and tomography (reconstructing a cross-section of an object from a series of projections).

Image analysis refers specifically to using computers to extract data from images. The result is usually some form of measurement. In the past, this was almost exclusively two-dimensional imaging, although with the advent of confocal microscopy and other advanced imaging techniques, this has extended to three dimensions.

Pattern recognition is concerned with the identification of objects based on patterns in the measurements (Haralick and Shapiro, 1991). There is a strong focus on statistical approaches, although syntactic and structural methods are also used.

Computer vision tends to use a model-based approach to image processing. Mathematical models of both the scene and the imaging process are used to derive a three-dimensional representation based on one or more two-dimensional images of a scene. The use of models implicitly provides an interpretation of the contents of the images obtained.

The fields are sometimes distinguished based on application:

Machine vision is using image processing as part of the control system for a machine (Schaffer, 1984). Images are captured and analysed, and the results are used directly for controlling the machine while performing a specific task. Real-time processing is often emphasised.

Remote sensing usually refers to the use of image analysis for obtaining geographical information, either using satellite images or aerial photography.

Medical imaging encompasses a wide range of imaging modalities (X-ray, ultrasound, magnetic resonance, etc.) concerned primarily with medical diagnosis and other medical applications. It involves both image reconstruction to create meaningful images from the raw data gathered from the sensors, and image analysis to extract useful information from the images.

Image and video coding focuses on the compression of an image or image sequence so that it occupies less storage space or takes less time to transmit from one location to another. Compression is possible because many images contain significant redundant information. In the reverse step, image decoding, the full image or video is reconstructed from the compressed data.

1.2 Image Formation

While there are many possible sensors that can be used for imaging, the focus in this section is on optical images, within the visible region of the electromagnetic spectrum. While the sensing technology may differ significantly for other types of imaging, many of the imaging principles will be similar.

The first requirement to obtaining an image is some form of sensor to detect and quantify the incoming light. In most applications, it is also necessary for the sensor to be directional, so that it responds primarily to light arriving at the sensor from a particular direction. Without this direction sensitivity, the sensor will effectively integrate the light arriving at the sensor from all directions. While such sensors do have their applications, the directionality of a sensor enables a spatial distribution to be captured more easily.

The classic approach to obtain directionality is through a pinhole as shown in Figure 1.1, where light coming through the pinhole at some angle maps to a position on the sensor. If the sensor is an array then a particular sensing element (a pixel) will collect light coming from a particular direction. The biggest

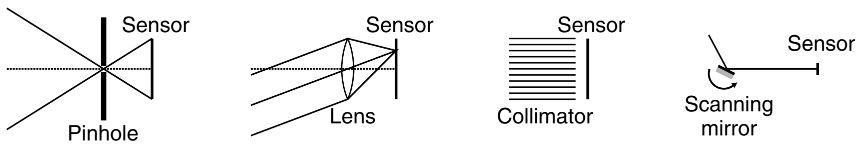


Figure 1.1 Different image formation mechanisms: pinhole, lens, collimator, scanning mirror.

limitation of a pinhole is that only limited light can pass through. This may be overcome using a lens, which focuses the light coming from a particular direction to a point on the sensor. A similar focussing effect may also be obtained by using an appropriately shaped concave mirror.

Two other approaches for constraining the directionality are also shown in Figure 1.1. A collimator allows light from only one direction to pass through. Each channel through the collimator is effectively two pinholes, one at the entrance, and one at the exit. Only light aligned with both the entrance and the exit will pass through to the sensor. The collimator can be constructed mechanically, as illustrated in Figure 1.1, or through a combination of lenses. Mechanical collimation is particularly useful for imaging modalities such as X-rays, where diffraction through a lens is difficult or impossible. Another sensing arrangement is to have a single sensing element, rather than an array. To form a two-dimensional image, the single element must be mechanically scanned in the focal plane, or alternatively have light from a different direction reflected towards the sensor using a scanning mirror. This latter approach is commonly used with time-of-flight laser range scanners (Jarvis, 1983).

The sensor, or sensor array, converts the light intensity pattern into an electrical signal. The two most common solid state sensor technologies are charge coupled device (CCD) and complementary metal oxide semiconductor (CMOS) active pixel sensors (Fossum, 1993). The basic light sensing principle is the same: an incoming photon liberates an electron within the silicon semiconductor through the photoelectric effect. These photoelectrons are then accumulated during the exposure time before being converted into a voltage for reading out.

Within the CCD sensor (Figure 1.2) a bias voltage is applied to one of the three phases of gate, creating a potential well in the silicon substrate beneath the biased gates (MOS capacitors). These attract and store the photoelectrons until they are read out. By biasing the next phase and reducing the bias on the current phase, the charge is transferred to the next cell. This process is repeated, successively transferring the charge from each pixel to a readout amplifier where it is converted to a voltage signal. The nature of the readout process means that the pixels must be read out sequentially.

A CMOS sensor detects the light using a photodiode. However, rather than transferring the charge all the way to the output, each pixel has a built in amplifier that amplifies the signal locally. This means that the charge remains local to the sensing element, requiring a reset transistor to reset the accumulated charge at the start of each integration cycle. The amplified signal is connected to the output via a row select transistor and column lines. These make the pixels individually addressable, making it easier to read sections of the array, or even accessing the pixels randomly.

Although the active pixel sensor technology was developed before CCDs, the need for local transistors made early CMOS sensors impractical because the transistors consumed most of the area of the device (Fossum, 1993). Therefore, CCD sensors gained early dominance in the market. However, the continual reduction of feature sizes has meant that CMOS sensors became practical from the early 1990s. The early CMOS sensors had lower sensitivity and higher noise than a similar format CCD sensor, although with recent technological improvements there is now very little difference between the two families (Litwiller, 2005). Since they use the same process technology as standard CMOS devices, CMOS sensors also enable other functions, such as A/D conversion, to be directly integrated on the same chip.

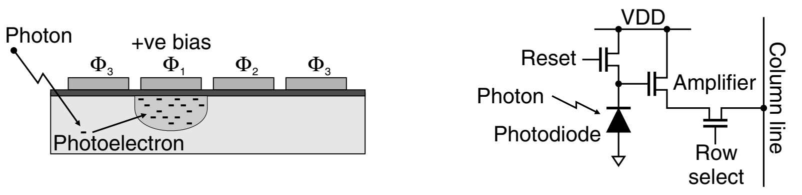


Figure 1.2 Sensor types. Left: the MOS capacitor sensor used by CCD cameras; right: a three-transistor active pixel sensor used by CMOS cameras.

Humans are able to see in colour. There are three different types of colour receptors (cones) in the human eye that respond differently to different wavelengths of light. If the wavelength dependence of a receptor is $S_k(\lambda)$, and the light falling on the receptor contains a mix of light of different wavelengths, $C(\lambda)$, then the response of that receptor will be given by the combination of the responses of all different wavelengths:

$$R_k = \int C(\lambda)S_k(\lambda)d\lambda \tag{1.1}$$

What is perceived as colour is the particular combination of responses from the three different types of receptor. Many different wavelength distributions produce the same responses from the receptors, and therefore are perceived as the same colour. In practise, it is a little more complicated than this. The incoming light depends on the wavelength distribution of the source of the illumination, $I(\lambda)$, and the wavelength dependent reflectivity, $\rho(\lambda)$, of the object being looked at. So Equation 1.1 becomes:

$$R_k = \int I(\lambda)\rho(\lambda)S_k(\lambda)d\lambda \tag{1.2}$$

To reproduce a colour, it is only necessary to reproduce the associated combination of receptor responses. This is accomplished in a television monitor by producing the corresponding mixture of red, green and blue light.

To capture a colour image for human viewing, it is necessary to have three different colour receptors in the sensor. Ideally, these should correspond to the spectral responses of the cones. However, since most of what is seen has broad spectral characteristics, a precise match is not critical except when the illumination source has a narrow spectral content (for example sodium vapour lamps or LED-based illumination sources). Silicon has a broad spectral response, with a peak in the near infrared. Therefore, to obtain a colour image, this response must be modified through appropriate colour filters.

Two approaches are commonly used to obtain a full colour image. The first is to separate the red, green and blue components of the image using a prism and with dichroic filters. These components are then captured using three separate sensor chips, one for each component. The need for three chips with precise alignment makes such cameras relatively expensive. An alternative approach is to use a single chip, with small filters integrated with each pixel. Since each pixel senses only one colour, the effective resolution of the sensor is decreased. A full colour image is created by interpolating between the pixels of a component. The most commonly used colour filter array is the Bayer pattern (Bayer, 1976), with filters to select the red, green and blue primary colours. Other patterns are also possible, for example filtering the yellow, magenta and cyan secondary colours (Parulski, 1985). Cameras using the secondary colours have better low light sensitivity because the filters absorb less of the incoming light. However, the processing required to produce the output gives a better signal-to-noise ratio from the primary filters than from secondary filters (Parulski, 1985; Baer *et al.*, 1999).

One further possibility that has been considered is to stack the red, green and blue sensors one on top of the other (Lyon and Hubel, 2002; Gilblom *et al.*, 2003). This relies on the fact that longer wavelengths of light penetrate further into the silicon before being absorbed. Thus, by using photodiodes at different depths, a full colour pixel may be obtained without explicit filtering.

Since most cameras produce video output signals, the most common format is the same as that of television signals. A two-dimensional representation of the timing is illustrated in Figure 1.3. The image is read out in raster format, with a horizontal blanking period at the end of each line. This was to turn off the cathode ray tube (CRT) electron beam while it retraced to the start of the next line. During the horizontal blanking, a horizontal synchronisation pulse controlled the timing. Similarly, after the last line is displayed there is a vertical blanking period to turn off the CRT beam while it is retraced vertically. Again, during the vertical blanking there is a vertical synchronisation pulse to control the vertical timing.

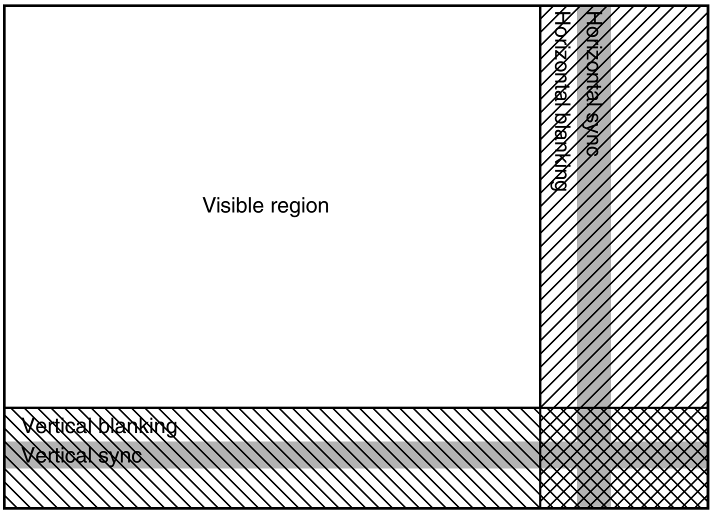


Figure 1.3 Regions within a scanned video image.

While such timing is not strictly necessary for digital cameras, at the sensor level there can also be blanking periods at the end of each line and frame.

Television signals are interlaced. The scan lines for a frame are split into two fields, with the odd and even lines produced and displayed in alternate fields. This effectively reduces the bandwidth required by halving the frame rate without producing the associated annoying flicker. From an image processing perspective, if every field is processed separately, this doubles the frame rate, albeit with reduced vertical resolution. To process the whole frame, it is necessary to re-interlace the fields. This can produce artefacts; when objects are moving within the scene, their location will be different in each of the two fields. Cameras designed for imaging purposes (rather than consumer video) avoid this by producing a non-interlaced or progressive scan output.

Cameras producing an analogue video signal require a *frame grabber* to capture digital images. A frame grabber preprocesses the signal from the camera, amplifying it, separating the synchronisation signals, and if necessary decoding the composite colour signal into its components. The analogue video signal is digitised by an A/D converter (or three A/D converters for colour) and stored in memory for later processing.

Digital cameras do much of this processing internally, directly producing a digital output. A digital sensor chip will usually provide the video data and synchronisation signals in parallel. Cameras with a low level interface will often serialise these signals (for example the Camera Link interface; AIA, 2004). Higher level interfaces will sometimes compress and will usually split the data into packets for transmission from the camera. The *raw* format is simply the digitised pixels; the colour filter array processing is not performed for single chip cameras. Another common format is RGB; for single chip sensors with a colour filter array, the pixels are interpolated to give a full colour value for each pixel. As the human eye has a higher spatial resolution to brightness than to colour, it is common to convert the image to YCbCr (Brown and Shepherd, 1995):

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.0 \\ 112.0 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \tag{1.3}$$

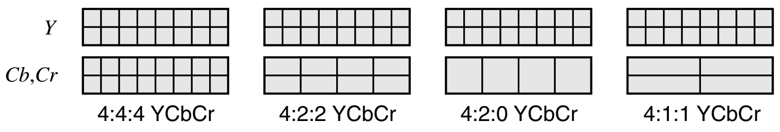


Figure 1.4 Common YCbCr subsampling formats.

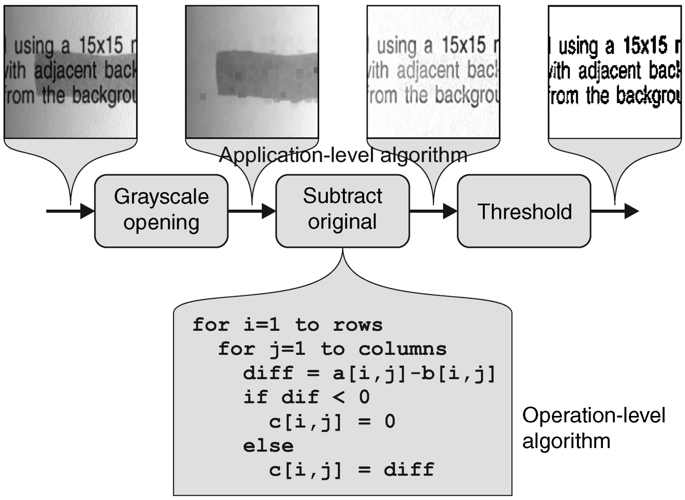
where the RGB components are normalised between 0 and 1, and the YCbCr components are 8-bit integer values. The luminance component, *Y*, is always provided at the full sample rate, with the colour difference signals, *Cb* and *Cr* provided at a reduced resolution. Several common subsampling formats are shown in Figure 1.4. These show the size or resolution of each *Cb* and *Cr* pixel in terms of the *Y* pixels. For example, in the 4:2:0 format, there is a single *Cb* and *Cr* value for each 2×2 block of *Y* values. To reproduce the colour signal, the lower resolution *Cb* and *Cr* values are combined with multiple luminance values. For image processing, it is usual to convert these back to RGB values using the inverse of Equation 1.3.

1.3 Image Processing Operations

After capturing the image, the next step is to process the image to achieve the desired result.

The sequence of image processing operations used to process an image from one state to another is called an *image processing algorithm*. The term algorithm is sometimes a cause of confusion, since each operation is also implemented through an algorithm in some programming language. This distinction between application level algorithms and operation level algorithms is illustrated in Figure 1.5. Usually the context will indicate in which sense the term is used.

As the input image is processed by the application level algorithm, it undergoes a series of transformations. These are illustrated for a simple example in Figure 1.6. The input image consists of an array of pixels. Each pixel, on its own, carries very little information, but there are a large number of pixels. The individual pixel values represent high volume, but low value data. As the image is processed, collections of pixels are grouped together. At this intermediate level, the data may still be represented in



```
for i=1 to rows
  for j=1 to columns
    diff = a[i,j]-b[i,j]
    if dif < 0
      c[i,j] = 0
    else
      c[i,j] = diff
```

Figure 1.5 Algorithms at two levels: application level and operation level.

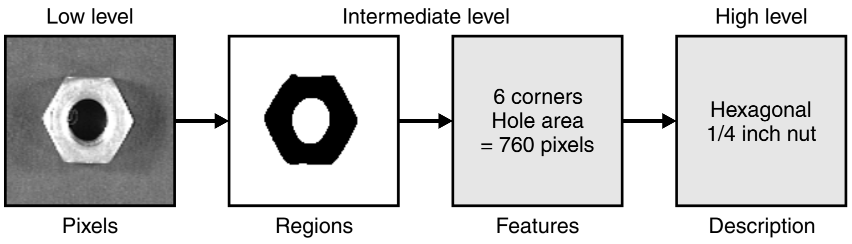


Figure 1.6 Image transformations.

terms of pixels, but the pixel values usually have more meaning; for example, they may represent a label associated with a region. At this intermediate level, the representation may depart from an explicit image representation, with the regions represented by their boundaries (for example as a chain code or a set of line segments) or their structure (for example as a quad tree). From each region, a set of features may be extracted that characterise the region. Generally, there are a small number of features (relative to the number of pixels within the region) and each feature contains significant information that can be used to distinguish it from other regions or other objects that may be encountered. At the intermediate level, the data becomes significantly lower in volume but higher in quality. Finally, at the high level, the feature data is used to classify the object or scene into one of several categories, or to derive a description of the object or scene.

Rather than focussing on the data, it is also possible to focus on the image processing operations. The operations can be grouped according to the type of data that they process (Weems, 1991). This grouping is sometimes referred to as an image processing pyramid (Downton and Crookes, 1998; Ratha and Jain, 1999), as represented in Figure 1.7. At the lowest level of the pyramid are preprocessing operations. These are image-to-image transformations, with the purpose of enhancing the relevant information within the image, while suppressing any irrelevant information. Examples of preprocessing operations are distortion correction, contrast enhancement and filtering for noise reduction or edge detection. Segmentation operations such as thresholding, colour detection, region growing and connected components labelling occur at the boundary between the low and intermediate levels. The purpose of segmentation is to detect objects or regions in an image, which have some common property. Segmentation is therefore an image to region transformation. After segmentation comes classification. Features of each region are used to identify objects or parts of objects, or to classify an object into one of several predefined categories. Classification transforms the data from regions to features, and then to labels. The data is no longer image based, but position information may be contained within the features, or be associated with the labels. At the highest level is recognition, which derives a description or some other interpretation of the scene.

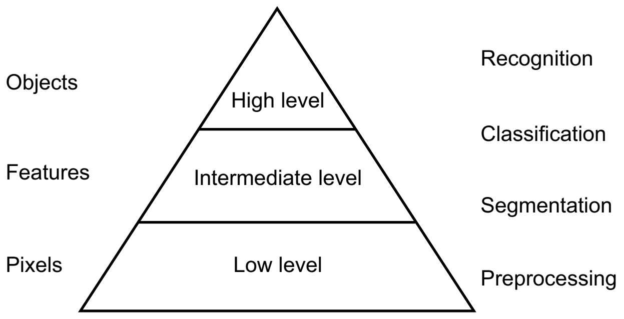


Figure 1.7 Image processing pyramid.

1.4 Example Application

To illustrate the different stages of the processing, and how the different stages transform the image, the problem of detecting blemishes on the surface of kiwifruit will be considered (this example application is taken from Bailey, 1985). One of the problems frequently encountered with 100% grading is obtaining and training sufficient graders. This is made worse for the kiwifruit industry because the grading season is very short (typically only four to six weeks). Therefore, a pilot study was initiated to investigate the potential of using image processing techniques to grade kiwifruit.

There are two main types of kiwifruit defects. The first are shape defects, where the shape of the kiwifruit is distorted. With the exception of the Hayward protuberance, the fruit in this category are rejected purely for cosmetic reasons. They will not be considered further in this example. The second class of defects is that which involves surface blemishes or skin damage. With surface blemishes, there is a one square centimetre allowance, provided the blemish is not excessively dark. However, if the skin is cracked or broken or is infested with scale or mould, there is no size allowance and the fruit is reject. Since almost all of the surface defects appear as darker regions on the surface of the fruit, a single algorithm was used to detect both blemishes and damage.

In the pilot study, only a single view of the fruit was considered; a practical application would have to inspect the complete surface of the fruit rather than just one view. Figure 1.8 shows the results of processing a typical kiwifruit with a water stain blemish. The image was captured with the fruit against a dark background to simplify segmentation, and diffuse lighting was used to reduce the visual texture caused by the hairs on the kiwifruit. Diffuse light from the direction of the camera also makes the centre of the fruit brighter than around the edges; this property is exploited in the algorithm.

The dataflow for the image processing algorithm is represented in Figure 1.9. The first three operations preprocess the image to enhance the required information while suppressing the information that is irrelevant for the grading problem. In practise, it is not necessary to suppress all irrelevant information, but sufficient preprocessing is performed to ensure that subsequent operations perform reliably. Firstly, a constant is subtracted from the image to segment the fruit from the background; any pixel less than the constant is considered to be background and is set to zero. This simple segmentation is made possible by using a dark background. The next step is to normalise the intensity range to make the blemish measurements independent of fluctuations in illumination and the exact shade of a particular piece of fruit. This is accomplished by expanding the pixel values linearly to set the largest pixel value to 255. This effectively makes the blemish detection relative to the particular shade of the individual fruit. The third preprocessing step is to filter the image to remove the fine texture caused by the hairs on the surface of the fruit. A 3×3 median filter was used because it removes the local intensity variations without affecting the larger features that must be detected. It also makes the modelling stage more robust by significantly reducing the effects of noise on the model.

The next stage of the algorithm is to compare the preprocessed image with an ideal model of an unblemished fruit. The use of a standard fixed model is precluded by the normal variation in both the size and shape of kiwifruit. It would be relatively expensive to normalise the image of the fruit to conform to a

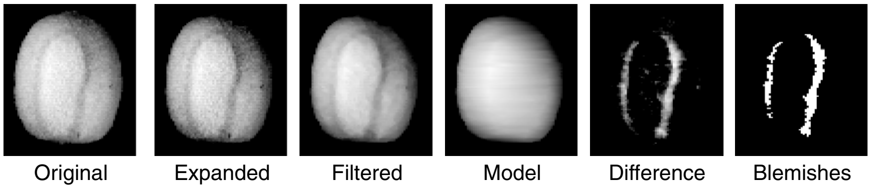


Figure 1.8 Steps within the processing of kiwifruit images (Bailey, 1985).

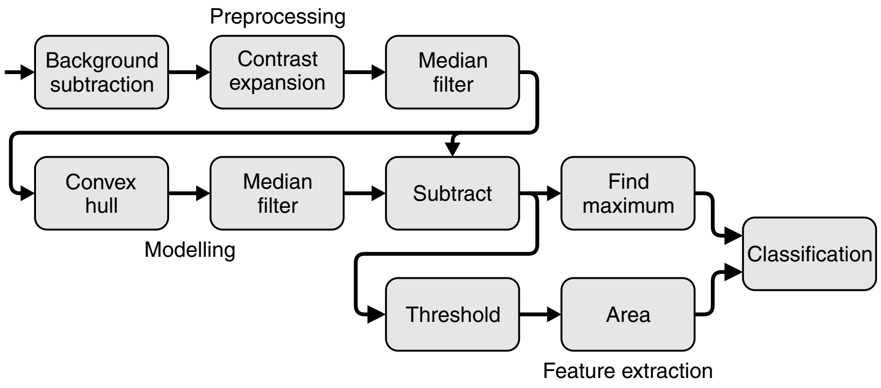


Figure 1.9 Image processing algorithm for detecting surface defects on kiwifruit (Bailey, 1985).

standard model, or to transform a model to match the fruit. Instead, a dynamic model is created from the image of the fruit. This works on the principle of removing the defects, followed by subtraction to see what has been removed (Batchelor, 1979). Such a dynamic model has the advantage of always being perfectly aligned with the fruit being examined both in position and in shape and size. A simple model can be created efficiently by taking the convex hull of the non-zero pixels along each row within the image. It relies on the lighting arrangement that causes the pixel values to decrease towards the edges of the fruit because of the changing angle of the surface normal. Therefore, the expected profile of an unblemished kiwifruit is convex, apart from noise caused by the hairs on the fruit. The convex hull fills in the pixels associated with defects by setting their values based on the surrounding unblemished regions of the fruit. The accuracy of the model may be improved either by applying a median filter after the convex hull to smooth the minor discrepancies between the rows, or by applying a second pass of the convex hull to each column in the image (Bailey, 1985).

The preprocessed image is subtracted from the model to obtain the surface defects. (The contrast of this difference image has been expanded in Figure 1.8 to make the subtle variations clearer.) A pixel value in this defect image represents how much that pixel had to be filled to obtain the model, and is therefore a measure of how dark that pixel is compared with the surrounding area. Some minor variations can be seen resulting from noise remaining after the prefiltering. These minor variations are insignificant, and should be ignored. The defect image is thresholded to detect the significant changes resulting from any blemish.

Two features are extracted from the processed images. The first is the maximum value of the difference image. This is used to detect damage on the fruit, or to determine if a blemish is excessively dark. If the maximum difference is larger than a preset threshold, then the fruit is rejected. The second feature is the area of the blemish image after thresholding. For blemishes, an area less than one square centimetre is acceptable; larger than this, the fruit is rejected because the blemish is outside the allowance.

Three threshold levels must be determined within this algorithm. The first is the background level subtracted during preprocessing. This is set from looking at the maximum background level when processing several images. The remaining two thresholds relate directly to the classification. The point defect threshold determines how large the difference needs to be to reject the fruit. The second is the area defect threshold for detecting blemished pixels. These thresholds were determined by examining several fruit that spanned the range from acceptable, through marginal, to reject fruit. The thresholds were then set to minimise the classification errors (Bailey, 1985).

This example is typical of a real-time inspection application. It illustrates the different stages of the processing, and demonstrates how the image and information are transformed as the image is processed.

This application has mainly low and intermediate level operations; the high level classification step is relatively trivial. As will be seen later, this makes such an algorithm a good candidate for hardware implementation on a FPGA (field programmable gate array).

1.5 Real-Time Image Processing

A *real-time system* is one in which the response to an event must occur within a specific time, otherwise the system is considered to have failed (Dougherty and Laplante, 1985). From an image processing perspective, a real-time imaging system is one that regularly captures images, analyses those images to obtain some data, and then uses that data to control some activity. All of the processing must occur within a predefined time (often, but not always, the image capture frame rate). Examples of real-time image processing systems abound. In machine vision systems, the image processing algorithm is used for either inspection or process control. Some robot vision systems use vision for path planning or to control the robot in some way where the timing is critical. Autonomous vehicle control requires vision or some other form of sensing for vehicle navigation or collision avoidance in a dynamic environment. In video transmission systems, successive frames must be transmitted and displayed in the right sequence and with minimum jitter to avoid a loss of quality of the resultant video.

Real-time systems are categorised into two types: hard and soft real time. A *hard real-time system* is one in which the complete system is considered to have failed if the output is not produced within the required time. An example is using vision for grading items on a conveyor belt. The grading decision must be made before the item reaches the actuation point, where the item is directed one way or another depending on the grading result. If the result is not available by this time, the system has failed. On the other hand, a *soft real-time system* is one in which the complete system does not fail if the deadline is not met, but the performance deteriorates. An example is video transmission via the internet. If the next frame is delayed or cannot be decoded in time, the quality of the resultant video deteriorates. Such a system is soft real time, because although the deadline was not met, an output could still be produced, and the complete system did not fail.

From a signal processing perspective, real time can mean that the processing of a sample must be completed before the next sample arrives (Kehtarnavaz and Gamadia, 2006). For video processing, this means that the total processing per pixel must be completed within a pixel sample time. Of course, not all of the processing for a single pixel can be completed before the next pixel arrives, because many image processing operations require data from many pixels for each output pixel. However, this provides a limit on the average processing rate, including any overhead associated with temporarily storing pixel values that will be used later (Kehtarnavaz and Gamadia, 2006).

A system that is not real time may have components that are real time. For instance, in interfacing to a camera, an imaging system must do something with each pixel as it is produced by the camera – either process it in some way, or store it into a frame buffer – before the next pixel arrives. If not, then the data for that pixel is lost. Whether this is a hard or soft real-time process depends on the context. While missing pixels would cause the quality of an image to deteriorate (implying soft real time), the loss of quality may have a significant negative impact on the performance of the imaging application (implying that image capture is a hard real-time task). Similarly, when providing pixels for display, if the required pixel data is not provided in time, that region of the display would appear blank. In the case of image capture and display, the deadlines are in the order of tens of nanoseconds, requiring such components to be implemented in hardware.

The requirement for the whole image processing algorithm to have a bounded execution time implies that each operation must also have a bounded execution time. This characteristic rules out certain classes of operation level algorithms from real-time processing. In particular, operations that are based on iterative or recursive algorithms can only be used if they can be guaranteed to converge satisfactorily within a predefined number of iterations, for the whole range of inputs that may be encountered.

One approach to guaranteeing a fixed response time is to make the imaging system synchronous. Such a system schedules each operation or step to execute at a particular time. This is suitable if the inputs occur at regular (or fixed) intervals, for example the successive frames from a video camera. However, synchronous systems cannot be used reliably when events occur randomly, especially when the minimum time between events is less than the processing time for each event.

The time between events may be significantly less than the required response time. A common example of this is a conveyor-based inspection system. The time between items on the conveyor may be significantly less than the time between the inspection point and the actuator. There are two ways of handling this situation. The first is to constrain all of the processing to take place during the time between successive items arriving at the inspection point, effectively providing a much tighter real-time constraint. If this new time constraint cannot be achieved then the alternative is to use distributed or parallel processing to maintain the original time constraint, but spread the execution over several processors. This can enable the time constraint to be met, by increasing the throughput to meet the desired event rate.

A common misconception of real-time systems, and real-time imaging systems in particular, is that they require high speed or high performance (Dougherty and Laplante, 1985). The required response time depends on the application and is primarily dependent on the underlying process to which the image processing is being applied. For example, a real-time coastal monitoring system looking at the movement of sand bars may require analysis times in the order of days or even weeks. Such an application would probably not require high speed! Whether or not real-time imaging requires high performance computing depends on the complexity of the algorithm. Conversely, an imaging application that requires high performance computing may not necessarily be real time (for example complex iterative reconstruction algorithms).

1.6 Embedded Image Processing

An *embedded system* is a computer system that is embedded within a product or component. Consequently, an embedded system is usually designed to perform one specific task, or a small range of specific tasks (Catsoulis, 2005), often with real-time constraints. An obvious example of an embedded image processing system is a digital camera. There the imaging functions include exposure and focus control, displaying a preview, and managing image compression and decompression.

Embedded vision is also useful for smart cameras, where the camera not only captures the image, but also processes it to extract information as required by the application. Examples of where this would be useful are "intelligent" surveillance systems, industrial inspection or control, robot vision, and so on.

A requirement of many embedded systems is that they need to be of small size, and light weight. Many run off batteries, and are therefore required to operate with low power. Even those that are not battery operated usually have limited power available.

1.7 Serial Processing

Traditional image processing platforms are based on a serial computer architecture. In its basic form, such an architecture performs all of the computation serially by breaking the operation level algorithm down to a sequence of arithmetic or logic operations that are performed by the ALU (arithmetic logic unit). The rest of the CPU (central processing unit) is then designed to feed the ALU with the required data. The algorithm is compiled into a sequence of instructions, which are used to control the specific operation performed by the CPU and ALU during each clock cycle. The basic operation of the CPU is therefore to fetch an instruction from memory, decode the instruction to determine the operation to perform, and execute the instruction.

All of the advances in mainstream computer architecture have been developed to improve performance by squeezing more data through the narrow bottleneck between the memory and the ALU (the so-called von Neumann bottleneck; Backus 1978).

The obvious approach is to increase the clock speed, and hence the rate at which instructions are executed. Tremendous gains have been made in this area, with top clock speeds of several GHz being the norm for computing platforms. Such increases have come primarily as the result of reductions in propagation delay brought about through advances in semiconductor technology reducing both transistor feature sizes and the supply voltage. This is not without its problems, however, and one consequence of a higher clock speed is significantly increased power consumption by the CPU.

While the speeds of bulk memory have also increased, they have not kept pace with increasing CPU speeds. This has caused problems in reading both instructions and data from the system memory at the required rate. Caching techniques have been developed to buffer both instructions and data in a smaller high speed memory that can keep pace with the CPU. The problem then is how to maximise the likelihood that the required data will be in the cache memory rather than the slower main memory. Cache misses (where the data is not in cache memory) can result in a significant degradation in processor performance because the CPU is sitting idle waiting for the data to be retrieved.

Both instructions and data need to be accessed from the main memory. One obvious improvement is to use a Harvard architecture, which doubles the effective memory bandwidth by using separate memories for instructions and data. This can give a speed improvement by up to a factor of two. A similar improvement may be obtained with a single main memory, using separate caches for instructions and data.

Another approach is to increase the width of the ALU, so that more data is processed in each clock cycle. This gives obvious improvements for wider data word lengths (for example double precision floating-point numbers) because each number may be loaded and processed in a fewer clock cycles. However, when the natural word length is narrower, such is typically encountered in image processing, the performance improvement is not as great unless data memory bandwidth is the limiting factor. When the data path is wider than the word length, it is also possible to design the ALU to operate as a vector processor. Multiple data words are packed into a single processor word allowing the ALU to perform the same operation simultaneously on several data items (for example using the Intel MMX instructions; Peleg *et al.*, 1997). This can be particularly effective for low level image processing operations where the same operation is performed on each pixel.

The speed of the fetch/decode/execute cycle may be improved by pipelining the separate phases. Thus, while one instruction is being executed, the next instruction is being decoded, and the instruction after that is being fetched. Depending on the complexity of the instructions, there may also be phases (and pipeline stages) for reading data from memory and writing the results to memory. Such pipelining is effective when executing long sequences of instructions without branches. However, instruction pipelining becomes more complex with loops or branches that break the regular sequence. This necessitates flushing the instruction pipeline to remove the successive instructions that have been incorrectly loaded and then loading the correct instructions. Techniques have been developed to attempt to minimise time lost through pipeline flushing. Branch prediction tries to anticipate which path will be taken at a branch, and increases the likelihood that the correct instructions will be loaded. Speculative execution, executes the instructions that have been loaded in the pipeline already, so that the time is not lost if the branch is predicted correctly. If the branch is incorrectly predicted, the results of the speculative execution are discarded.

Other instruction set architectures have been designed to increase the CPU throughput. RISC (reduced instruction set computer) architectures do this by simplifying the instructions, enabling a higher clock speed. VLIW (very large instruction word) architectures enable multiple independent instructions to execute in parallel, in an effort to maximise the utilisation of all parts of the CPU.

More recently, multiple core architectures have become mainstream. These enable multiple threads within an application to execute in parallel on separate processor cores (Geer, 2005). These can give some improvement for image processing applications, provided that the application has been developed carefully to support multiple threads. If care is not taken, the memory bandwidth in accessing the image can still be a bottleneck.

While impressive performance gains have been achieved, with serial processors, time is usually the critical resource. This is because serial processors can only do one thing at a time. VLIW and multicore processors are beginning to overcome this limitation, but, even so, they are still serially bound. Some problems do not fit well onto serial processors, for example, those for which the processing does not scale linearly with the number of inputs, or where the complexity of the problem is such that it remains intractable (or impractical) given current processor speeds.

Another recent development is the GPU (graphics processing unit), a processor customised primarily for graphics rendering, and initially driven by the high-end video game market. The primary function performed by a GPU is to take vertex data representing triangular patches within the scene and produce the corresponding output pixels, which are stored in a frame buffer. Native operations include texture mapping, pixel shading, z-buffering and blending, and anti-aliasing. Early GPUs had dedicated pipelines for each of these stages, which restricted their use for wider application. More recent devices are programmable, enabling them to be used for image processing (Cope *et al.*, 2005) or other computationally intensive tasks (Manocha, 2005). The speed gain is achieved through a combination of data pipelining and lightweight multithreading (NVIDIA, 2006). Data pipelining reduces the need to write temporary results to memory only to read them in again to perform further processing. Instead, results are written to sets of local registers where they may be accessed in parallel. Multithreading splits the task into several small steps that can be implemented in parallel, which is easy to achieve for low level image processing operations where the independent processing of pixels maps well to the GPU architecture. Lightweight threads reduce the overhead of switching the context from one thread to the next. Therefore, when one thread stalls waiting for data, the context can be rapidly switched to another thread to keep the processing units fully used. GPUs can give significant improvements over CPUs for algorithms that are easily parallelised, especially where data access is not the bottleneck (Cope *et al.*, 2005). However, power considerations rule them out for many embedded vision applications.

For low power, or embedded vision applications, the size and power requirements of a standard serial processor are impractical in many cases. Lowering the clock speed can reduce the power significantly, but on a serial processor will also limit the algorithms that can be implemented in real time. To retain the computing power while lowering the clock speed requires multiple parallel processors.

1.8 Parallelism

In principle, every step within any algorithm may be implemented on a separate processor, resulting in a fully parallel implementation. However, if the algorithm is predominantly sequential, with every step within the algorithm dependent on the data from the previous step, then little can be gained in terms of reducing the response time. To be practical for parallel implementation, an algorithm has to have a significant number of steps that may be implemented in parallel. This is referred to as Amdahl's law (Amdahl, 1967). Let s be the proportion of the algorithm that is constrained to run serially (the housekeeping and other sequential components) and p the proportion of the algorithm that may be implemented in parallel over N processors. The best possible speedup that can be obtained is then:

$$Speedup \leq \frac{s+p}{s+\frac{p}{N}} = \frac{N}{1+(N-1)\,s} \tag{1.4}$$

The equality will only be achieved if there is no additional overhead (for example communication or other housekeeping) introduced as a result of parallelisation. This speedup is always less than the number of parallel processors, and will be limited by the proportion of the algorithm that must be serially executed:

$$\underset{N\to\infty}{\text{Limit}}\ Speedup = \frac{1}{s} \tag{1.5}$$

Therefore, to achieve any significant speedup, the proportion of the algorithm that can be implemented in parallel must also be significant. Fortunately, image processing is inherently parallel,

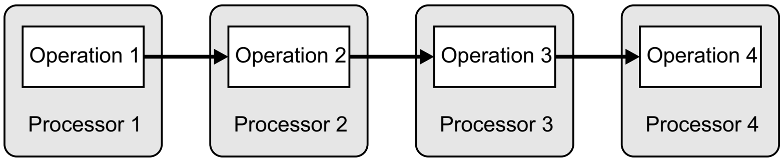


Figure 1.10 Temporal parallelism exploited using a processor pipeline.

especially at the low and intermediate levels of the processing pyramid. This parallelism shows in a number of ways.

Virtually all image processing algorithms consist of a sequence of image processing operations. This is a form of *temporal parallelism*. Such a structure suggests using a separate processor for each operation, as shown in Figure 1.10. This is a *pipelined architecture*. It works a little like a production line in that the data passes through each of the stages as it is processed. Each processor applies its operation and passes the result to the next stage. If each successive processor has to wait until the previous processor completes its processing, this arrangement will not reduce the total processing time, or the response time. However, the throughput can increase, because while the second processor is working on the output of operation 1, processor 1 can begin processing the next image. When processing images, data can usually begin to be output from an operation long before the complete image has been processed by that operation. The time between when data is first input to an operation and the corresponding output is available is the *latency* of that operation. The latency is lowest when each operation only uses input pixel values from a small, local neighbourhood because each output only requires data from a few input pixel values. Operations that require the whole image to calculate an output pixel value will have a higher latency. Operation pipelining can give significant performance improvements when all of the operations have low latency, because a downstream processor may begin performing its operation before the upstream processors have completed. This can give benefits, not only for multiprocessor systems, but also for software-based systems using a separate thread for each process (McLaughlin, 2000) because the user can begin to see the results before the whole image has been processed. Of course, in a software system using a single core, the total response time will not normally be decreased, although there may be some improvement resulting from switching to another thread while waiting for data from slower external memory when it is not available in the cache. In a hardware system, however, the total response time is given by the sum of the latencies of each of the stages plus the time to input in the whole image. If the individual latencies are small compared to the time required to load the image, then the speedup factor can be significant, approaching the number of processors in the pipeline.

Two difficulties can be encountered with using pipelining for more complex image processing algorithms (Duff, 2000). The first is with multiple parallel paths. If, for example, Processor 4 in Figure 1.10 also takes as input the results output from processor 1 then the data must be properly synchronised to allow for the latencies of the other processors in the parallel path. For synchronous processing, this is simply a delay line, but synchronisation becomes considerably more complex when the latencies of the parallel operations are variable. A greater difficulty is handling feedback, either with explicitly iterative algorithms, or implicitly where the parameters of the earlier processors adapt to the results produced in later stages.

A lot of the parallelism within an operation level algorithm is in the form of loops. The outermost loop within each operation usually iterates over the pixels within the image, because many operations perform the same function independently on many pixels. This is *spatial parallelism*, which may be exploited by partitioning the image and using a separate processor to perform the operation on each partition. Common partitioning schemes split the image into blocks of rows, blocks of columns, or rectangular blocks, as illustrated in Figure 1.11. For video processing, the image sequence may also be partitioned in time, by assigning successive frames to separate processors (Downton and Crookes, 1998).

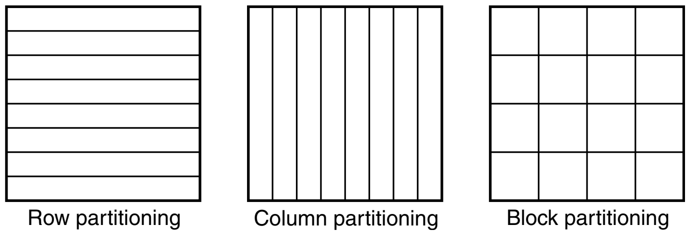


Figure 1.11 Spatial parallelism exploited by partitioning the image.

In the extreme case, a separate processor may be allocated to each pixel within the image (for example the MPP; Batcher, 1980). Dedicating a processor to each pixel can facilitate some very efficient algorithms. For example, image filtering may be performed in only a few clock cycles. However, one problem with such parallelism is the overhead required to distribute the image data to and from each of the processors. Many algorithms for massively parallel processors assume that the data is already available at the start.

The important consideration when partitioning the image is to minimise the communication between processors, which corresponds to minimising the communication between the different partitions. For low level image processing operations, the performance improvement approaches the number of processors. However, the performance will degrade as a result of any communication overheads or contention when accessing shared resources. Consequently, each processor must have some local memory to reduce any delays associated with contention for global memory. Partitioning is therefore most beneficial when the operations only require data from within a local region, where local is defined by the partition boundaries. If the operations performed within each region are identical, this leads to a SIMD (single instruction, multiple data) parallel processing architecture according to Flynn's taxonomy (Flynn, 1972).

With some intermediate level operations, the processing time for each partition may vary significantly depending on the content of the image within that region. A simple static, regular, partitioning strategy will be less efficient in this instance because the worst-case performance must be allowed for when allocating partitions to processors. As a result, many of the processors may be sitting idle for much of the time. In such cases, better performance may be achieved by having more partitions than processors, and using a processor farm approach (Downton and Crookes, 1998). Each partition is then allocated dynamically to the next available processor. Again, it is important to minimise the communications between processors.

For high level image processing operations, the data is no longer image based. However, data partitioning methods can still be exploited by assigning a separate data structure, region, or object to a separate processor. Such assignment generally requires the dynamic partitioning approach of a processor farm.

Logical parallelism reuses the same functional block many times within an operation. This often corresponds to inner loops within the algorithm implementing the operation. Logical parallelism is exploited by executing each instance of the function block in parallel, effectively unrolling the inner loops. Often the functions can be arranged in a regular structure, as illustrated in the examples in Figure 1.12. The odd–even transposition network within a bubble sort (for example to sort the pixel values within a rank window filter; Heygster, 1982; Hodgson *et al.*, 1985) consists of a series of compare and swap function blocks. Implementing the blocks in parallel gives a considerable speed improvement over iterating with a single compare and swap block. A linear filter or convolution multiplies the pixel values within a window by a set of weights, or filter coefficients. The multiply and accumulate block is repeated many times, and is a classic example of logic parallelism.

Often the structure of the group of function blocks is quite regular, which, when combined with pipelining, results in a synchronous systolic architecture (Kung, 1985). When properly implemented, such

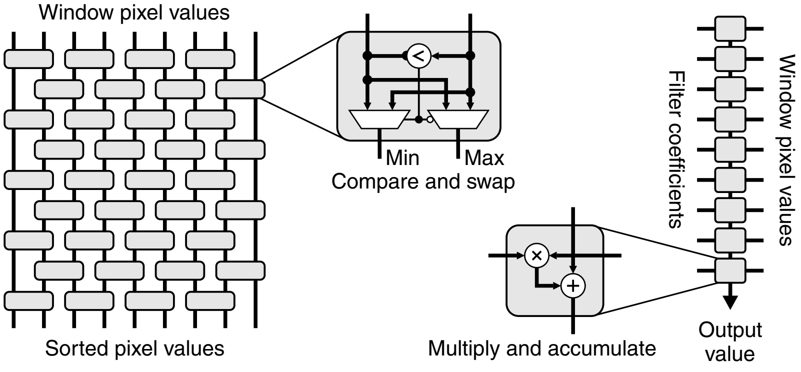


Figure 1.12 Examples of logical parallelism. Left: compare and swap within a bubble sorting odd–even transposition network; right: multiply and accumulate within a linear filter.

architectures have a very low communication overhead, enabling significant speed improvements resulting from multiple copies of the function block.

One of the common bottlenecks within image processing is the time, and bandwidth, required to read the image from memory and write the resultant image to memory. *Stream processing* exploits this to convert spatial parallelism into temporal parallelism. The image is read (and written) sequentially using a raster scan often at a rate of one pixel per clock cycle (Figure 1.13). It then performs all of its processing on the pixels on-the-fly as they are being read or written. The individual operations with the operation level algorithm are pipelined as necessary to maintain the required throughput. Data is cached locally to avoid the need for parallel external data accesses. This is most effective if each operation uses input from a small local neighbourhood, otherwise the caching requirements can be excessive. Stream processing also works best when the time taken to process each pixel is constant. If the processing time is data dependent, it may be necessary to insert FIFO (first-in, first-out) buffers on the input or output (or both) to maintain a constant data rate. With stream processing, the response time is given by the sum of the time required to read or write the image and the processing latency (the time between reading a pixel and producing its corresponding output). For most operations the latency is small compared with loading the whole image,

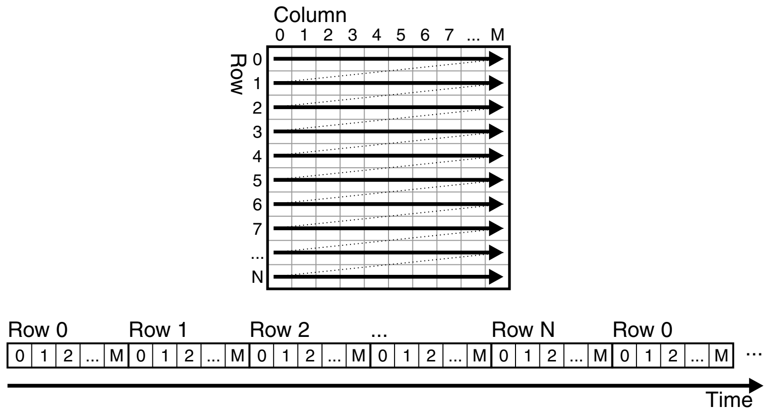


Figure 1.13 Stream processing converts spatial parallelism into temporal parallelism.

so if the whole application level algorithm can be implemented using stream processing the response time is dominated by frame rate.

1.9 Hardware Image Processing Systems

In the previous section, it was not made explicit whether the processors that made up the parallel systems were a set of standard serial processors, or dedicated hardware implementing the parallel functions. In theory, it makes little difference, as long as the parallelism is implemented efficiently, and the communication overheads are kept low.

Early serial computer systems were too slow for processing images of any size or in any volume. The regular structure of images led to designs for hardware systems to exploit the parallelism because hardware systems are inherently parallel. Many of the earliest systems were based on having a relatively simple processing element (PE) at each pixel. Unger (Unger, 1958) proposed a system based on a 4-connected square grid for processing binary images. Although the hardware for each PE was relatively simple, he demonstrated that many useful low level image processing operations could be performed. Although this early system was not built (Duff, 2000), it inspired other similar architectures. Golay proposed a system based on a hexagonal grid (Landsman *et al.*, 1965; Golay, 1969) with the image cycled through a single PE using stream processing. Such a system was ultimately built by Preston (Preston *et al.*, 1979). The CLIP series of image processors used a parallel set of PEs. The CLIP2 (Duff *et al.*, 1973) used a 16×12 hexagonal array, and although it was able to perform some basic processing it was not practical for real applications. The successor, CLIP4 (Duff, 2000) used a 96×96 array, and was able to operate on greyscale images in a bit serial manner. A similar 128×128 array (the massively parallel processor) was built by Batcher (Batcher, 1980); it also operated in a bit serial manner.

For capturing and display of video or images, even serial computers required a hardware system to interface with the camera or display. These frame grabbers interfaced with the host computer system either through direct memory access (DMA), or shared memory that was mapped within the address space of the host. A basic frame grabber consisted of an A/D converter, a bank of memory capable of holding at least one image frame, and a digital to analogue converter for image display. Many allowed basic point operations to be applied through lookup tables on the image being captured or displayed. This simple pipelining was extended to more sophisticated operations and completely pipelined image processing systems, the most notable of which was the Datacube. Pipelining is less suitable for high level image processing; hybrid systems were developed consisting of pipelined hardware for the preprocessing and an array of serial processors (often transputers) for high level processing, such as in the Kiwivision system (Clist and Valkenburg, 1994).

The early hardware systems were implemented with small and medium scale integrated circuits. The advent of VLSI (very large scale integration) lowered the cost of hardware and dramatically increased the speed and performance of the resulting systems. This led to a wider range of hardware architectures being used and image processing operations being implemented (Offen, 1985). It is interesting to note that, as technology has improved, the reducing feature sizes has meant that the cost of producing the masks has increased, significantly increasing the proportion of the one-off costs relative to the cost of individual devices. Consequently, the economics of VLSI production meant that only a limited range of circuits (those general enough to warrant large production runs) saw widespread commercial use.

Hardware-based image processing systems are very fast, but their biggest problem is their relative inflexibility. Once configured they perform their task very well, but it is difficult, if not impossible, to reconfigure these systems if the nature of the task changes. In the 1980s, the introduction of FPGA (field programmable gate array) technology opened new possibilities for digital logic design. FPGAs combine the inherent parallel nature of hardware with the flexibility of software in that their functionality can be reprogrammed or reconfigured. Early FPGAs were quite small in terms of the equivalent number of gates, so they tended to be used primarily for providing flexible interconnect and interface logic (sometimes called *glue logic*), and for implementing finite state machine-based controllers. Multiple FPGAs were required to implement any significant image processing system. As FPGAs grew in power, hybrid systems were developed where FPGAs were used as vision coprocessors or accelerators within a host computer (for example the Splash-2 system; Athanas and Abbot, 1995). Modern FPGAs now have sufficient resources to enable whole applications to be implemented on a single FPGA, making them an ideal choice for embedded real-time vision systems.

2

Field Programmable Gate Arrays

In this chapter, the basic architectural features of FPGAs are examined. While a detailed understanding of the internal architecture of an FPGA is not essential to programme them (the vendor-specific place and route tools manage most of these details), a basic knowledge of the internal structure and how a design maps onto that architecture can be used to develop a more efficient implementation.

2.1 Programmable Logic

The basic idea behind programmable hardware is to have a generic circuit where the functionality can be programmed for a particular application. Conventional computers are based on this idea, where the ALU can perform one of several operations based on a set of control signals. The limitation of an ALU is that it can only perform one operation at a time. Therefore, a particular application must be broken into the sequence of control signals for controlling the function of the ALU, along with the logic required to provide the appropriate data to the inputs of the ALU. The sequence of controls is provided by a sequence of programme instructions stored in the memory.

Programmable logic, on the other hand, represents the functionality as a circuit, where the particular circuit can be programmed to meet the requirements of an application. The key distinction is that the functionality is implemented as a parallel system, rather than sequential. Any logic function may be implemented using a two-level minterm (OR of AND) representation. Therefore, it made sense that early programmable logic devices consisted of a two-level programmable array architecture, such as that shown on the left in Figure 2.1. Each of the inputs (and its inverse) can potentially be connected to the input of each of a set of AND gates. Each circle represents a programmable connection between the corresponding vertical and horizontal line, as shown in the bottom of Figure 2.1. A similar arrangement is used to connect the output of each of the AND gates to the inputs of the OR gates. In practice, both input and output sections use the same circuit since, from De Morgan's theorem, the OR of ANDs can be implemented as a NAND of NANDs. Sequential logic, such as counters and finite state machines can be implemented with the addition of latches or flip-flops on the output, and appropriate feedback from the output back to the input.

Simple programmable logic devices began appearing in the early 1970s, and were mainly used to simplify circuits containing many discrete logic devices. The early devices were either mask programmable or fuse programmable. Mask-programmable devices were programmed at manufacture, effectively hard wiring the programming using a metal mask. Fuse-programmable devices can be programmed by the user, making such circuits *field programmable*. The programmable sections can consist either of fuses

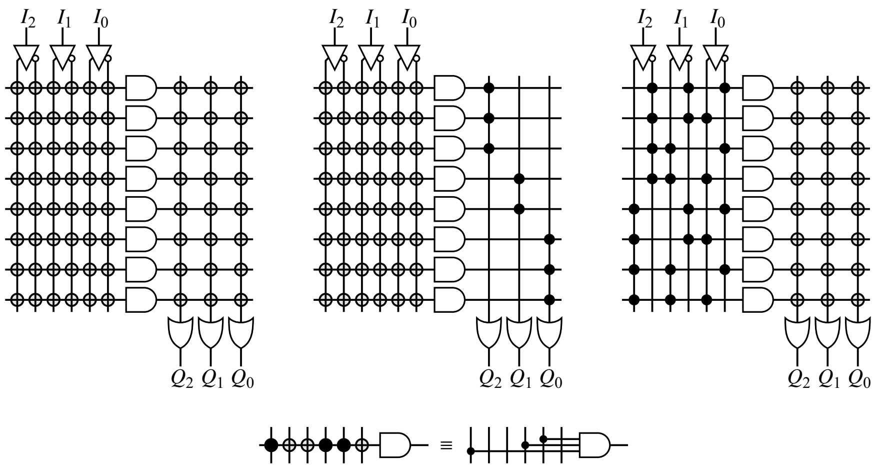


Figure 2.1 Types of programmable logic array. Left: both the input AND and output OR sections are programmable. Centre: PAL, with programmable AND section and fixed OR section; right: PROM, with fixed AND section and programmable output OR section; bottom: interpretation of the programmable sections.

where a higher than normal current is used to break a connection (blow the fuse in the conventional sense), or more usually antifuses, where a thin insulating layer is broken down by applying a high voltage across it, and it becomes conducting. Since blowing a fuse (or antifuse) is destructive, once programmed the devices cannot be reprogrammed.

Such a programmable array provides full programming flexibility. However, it was soon realised that advantages could be gained from having a less flexible array. If the output (OR) array is replaced by fixed connections, so that each OR gate has a fixed number of AND gates connected to it, the result is *programmable array logic* (PAL) and is illustrated in the centre panel of Figure 2.1. While this may appear to reduce the number of connections, in practice the number of AND gates must be increased because the AND terms cannot be shared between different outputs. However, restricting the programmability to just one section does reduce the area, and the speed of the OR section is increased. The only limitation this imposes on the logic function is the maximum number of terms that may be ORed together to give a particular output. For all but the more complex circuits, this limitation is not significant, and even when this limit is reached, multiple outputs may be used and combined to implement more complex logic functions.

Another advance, in the mid 1980s, was to control each programmable connection with an EEPROM (electrically erasable programmable read only memory) cell rather than a fuse. This enables the devices to be cleared and reprogrammed. A key benefit is that the logic may be altered simply by reprogramming the device. This opens possibilities for remote upgrading, without necessarily having the product returned to the factory or service centre.

An alternative simplification to the PAL is to fix the AND section and make the OR section programmable, as shown in the right panel of Figure 2.1. This structure effectively results in a programmable read only memory (PROM) or lookup table (LUT). Each AND gate will correspond to one combination of the input signals, corresponding to exactly one line of the truth table representation of the function. So for three inputs, there will be 2^3 or eight AND gates. The OR section is programmed to select the combination of inputs required to give a high output. A lookup table may implement any arbitrary function of its inputs. This gives PROMs considerable flexibility, although the circuit size grows exponentially with the number

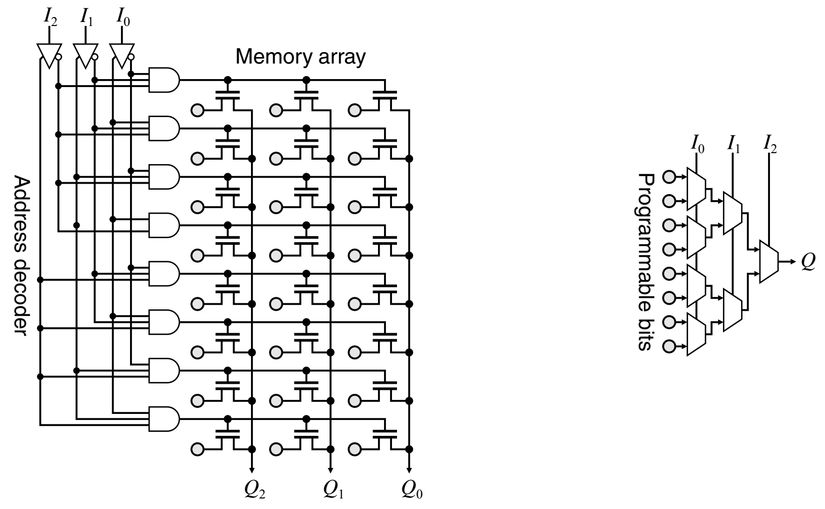


Figure 2.2 PROM architectures. Left: implemented as memory; right: multiplexer tree.

of inputs, which restricts the usefulness of PROMs to where only a few inputs are required. Larger PROMs are also much slower than dedicated logic circuits, and consume more power. For these reasons, large PROMs are not often used for general programmable logic. However, the flexibility of the lookup table architecture can be effective with a small number of inputs, making it the most commonly used building block within FPGAs. In practise, a PROM is not implemented as shown in Figure 2.1. The programmable OR connection is usually replaced by a programmed memory cell, which is connected to the output via a pass transistor as shown in Figure 2.2. Alternatively, for a small number of inputs the programmed memory cell may be connected to the output via a multiplexer tree.

The programmable logic described so far only has a limited number of inputs and outputs. The next step in the development of programmable logic was to incorporate several blocks of logic on a single chip, with programmable interconnections to enable more complex logic functions to be created. Two classes of device resulted – complex programmable logic devices (CPLDs) based on the PAL architecture, and FPGAs based on the LUT architecture. These differences mean that CPLDs have a higher density of logic compared to interconnect, whereas the logic blocks within FPGAs are smaller and the architecture is more dominated by interconnects. The other main difference between CPLDs and FPGAs is that CPLDs tend to be flash-programmable, whereas the programme or configuration for most FPGAs is usually held in static memory cells. This makes the configuration volatile, requiring it to be reloaded each time the device is powered on. The configuration data is contained in a *configuration file* which is used to programme the system.

Both CPLDs and FPGAs were introduced in the mid 1980s, and were initially used primarily as glue logic on printed circuit boards to flexibly interconnect a range of other components. However, as device densities increased, FPGAs began to dominate because of the increased flexibility that came primarily through the more flexible interconnect structure. Current FPGAs have sufficient logic resources (and associated interconnect routing) to implement even complex applications on a single chip. More recent trends have led to the integration of specific functional blocks within the FPGA, including multipliers, memories, high speed input–output interfaces, and even serial processor cores (Leong, 2008). Implementing such commonly used features as dedicated blocks frees up the programmable logic resources that would otherwise be required. It also increases the speed and reduces the power consumption compared with implementing those functions out of general purpose programmable logic.

2.1.1 FPGAs vs. ASICs

Programmability comes at a cost, however. If an FPGA-based implementation is compared with the same implementation in dedicated hardware as an application specific integrated circuit (ASIC), the FPGA requires more silicon, is slower, and requires more power. This should not come as a surprise – a custom made circuit will always be smaller than a generic circuit for a number of reasons. Firstly, the custom circuit only needs to consist of the gates that are required for the application, whereas the programmable circuit will inevitably have components that are not used, or not used in the best possible way. Secondly, a programmable interconnect is not required in a custom circuit; the logic is just wired where it is needed. The interconnect logic must also be sufficiently flexible to allow a wide range of designs, so again it will only be partially used in any particular configuration. Thirdly, the programmable hardware requires extra configuration logic, which also consumes a significant proportion of the chip real estate. Taking all these factors into consideration, an FPGA requires 20–40 times the silicon area of an equivalent ASIC (Kuon and Rose, 2006). It is at the higher end of this range for designs dominated by logic, and at the lower end of the range for those applications that make significant use multiplier blocks and memories, where the programmable blocks are bigger.

In terms of speed, the flexibility of programmable logic means that it will always be slower than a custom circuit. Also, since programmable circuits are larger, they will have more capacitance, reducing the maximum clock speed. On custom hardware, the wiring delays can be minimised by placing connected components as close as possible to each other. The need for programmable interconnect on FPGAs means the logic blocks are spaced further apart, so it takes longer for the signals to travel from one block to another. This is made worse by the fact that designs tend to be more scattered over the resources available by the automated place and route tools. Finally, every interconnection switch that a signal passes through also introduces a delay. Therefore, an ASIC will typically be three to four times faster than an FPGA (Kuon and Rose, 2006) for the same level of technology.

An FPGA will also consume about 10–15 times more dynamic power than a comparable ASIC (Kuon and Rose, 2006). The main causes of the large disparity in power dissipation are the increased capacitance and the larger number of transistors that must be switched.

In spite of these disadvantages, FPGAs do have significant advantages over ASICs, particularly with digital systems. The mask and design costs of ASICs are significantly higher, although the cost per chip is significantly lower (Figure 2.3). This makes ASICs only economical where high volumes are required, or where the speed cannot be matched by an FPGA. With each new technology generation the capabilities of FPGAs increase, moving the crossover point to higher and higher volumes. FPGAs also enable a shorter time to market, both because of the shorter design period and because the reconfigurability allows the design to be modified relatively late into the design cycle. The fact that the design may also be reconfigured in the field can also extend the useful lifetime of a product based on FPGAs. The future is likely to see an increased use of reconfigurability within ASICs to address these issues (Rutenbar *et al.*, 2001).

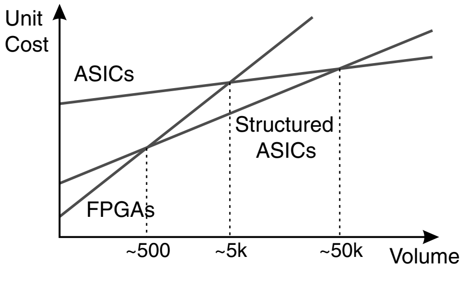


Figure 2.3 The relative costs of FPGAs, ASICs and structured ASICs.

Structured ASIC approaches, which effectively take an FPGA design and replace it with the same logic (hardwired) and fixed routing, can also overcome some of these problems. A structured ASIC consists of a predefined sea of gates, with the interconnections added later. This effectively makes them mask-programmed FPGAs. Fabrication of the user's device consists of adding from two to six metal layers forming the interconnection wiring between the gates. The initial cost is lower because the design and mask costs for the underlying silicon can be shared over a large number of devices, and only the metal interconnection layers need to be added to make a functioning device.

Both Altera (with HardCopy; Altera, 2009a) and Xilinx (with EasyPath; Krishnan, 2005) provide services based on their high end FPGA families for converting an FPGA implementation into a structured ASIC. The underlying sea of gates for these reflects the underlying architecture of the FPGA on which they are based. Such devices provide significant cost reductions for volume production, while achieving a 50% improvement in performance at less than half the power of the FPGA design they are based on (Leong, 2008). NEC Electronics Corporation also provides an embedded gate array service (NEC, 2009). This is based in a generic sea of gates, with the logic designs mapped onto these gates.

For prototyping, and designs where a relatively small number of units is required, the programmability and low cost of FPGAs makes them the device of choice.

2.2 FPGAs and Image Processing

Since an FPGA implements the logic required by an application by building separate hardware for each function, FPGAs are inherently parallel. This gives them the speed that results from a hardware design while retaining the reprogrammable flexibility of software at a relatively low cost. This makes FPGAs well suited to image processing, particularly at the low and intermediate levels where they are able to exploit the parallelism inherent in images. FPGAs can readily implement most of the forms of parallelism discussed in Section 1.8.

For a pipelined architecture, separate hardware is built for each image processing operation in the pipeline. In a data synchronous system, data is simply passed from the output of one operation to the input of the next. If the data is not synchronous, appropriate buffers may be incorporated between the operations to manage variations in dataflow or access patterns.

Spatial parallelism may be exploited by building multiple copies of the processing hardware and assigning different image partitions to each of the copies. The extreme case of spatial parallelism consists of building a separate processor for each pixel. This is not really practical for all but very small images; for realistic sized images, such massive parallelism does not map well onto current FPGAs, simply because the number of pixels within images exceeds the resources available within an FPGA.

Logical parallelism within an image processing operation is well suited to FPGA implementation, and it is here where many image processing algorithms may be accelerated significantly. This is accomplished by unrolling the inner loops, so that rather than performing the operations sequentially, parallel hardware is used.

Streaming feeds image data serially through a single function block. This maps well to a hardware implementation, especially when interfacing directly to a camera or display where the images are naturally streamed. If all of the operations are able to be implemented using stream processing, then the implementation of the whole algorithm as a single streamed pipeline results in a very efficient implementation. With stream processing, pipelining is usually required to achieve the required throughput.

The ability to make significant use of parallelism has significant implications when building embedded vision systems. Performing multiple operations in parallel enables the clock speed to be lowered significantly. A VGA resolution video streamed from a camera at 30 frames per second produces approximately 10 million pixels per second (although the clock speed is usually higher to account for the blanking periods). Any significant processing requires many operations to be performed for each pixel, requiring a conventional serial processor to have a much higher clock frequency. A streamed pipelined system implemented on an FPGA can often be operated at the native pixel input (or output) clock frequency. This corresponds to a reduction in clock speed over a serial processor of two orders of magnitude or more. The dynamic power consumption of a system is directly related to the clock frequency, so a slower clock results in a significantly lower power design.

If the whole algorithm can be implemented on a single FPGA, the resulting system has a small form factor. Designs with only two or three chips are possible, enabling the whole image processing system to be embedded with the sensor. This enables smart sensors and smart cameras to be built, where the intelligence of the system is built within the camera (Leeser *et al.*, 2004; Mosqueron *et al.*, 2007). The result is that vision can then be embedded within many applications as a versatile sensor.

2.3 Inside an FPGA

So, what exactly is inside an FPGA? Figure 2.4 shows the basic structure and essential components of a generic FPGA. The programmable logic consists of a set of fine-grained blocks that are used to implement the logic of the application. This is sometimes called the *fabric* of the FPGA. The logic blocks are usually based on a lookup table architecture, enabling them to implement any arbitrary function of the inputs. The logic blocks are typically tiled in a grid structure and interconnected via a programmable routing matrix that enables the blocks to be connected in arbitrary configurations. The input and output (I/O) blocks interface between the internals or core of the FPGA and external devices. The routing means that virtually any signal can be routed to any I/O pin of the device.

In addition to these basic features, most FPGAs provide some form of clock synchronisation to control the timing of a clock signal relative to an external source. A clock distribution network provides clock signals to all parts of the FPGA while limiting the clock skew between different sections of a design. There is also some dedicated logic for loading the configuration into the FPGA. This logic does not directly form part of the user's design, but is the overhead required for FPGAs to be programmable and provides the mechanisms for loading the user's design onto an FPGA.

Each of these components is examined in a little more detail in the following sections.

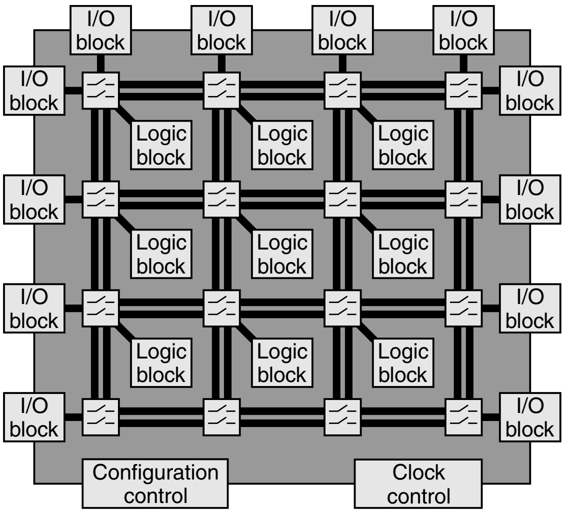


Figure 2.4 The basic architecture of an FPGA.

2.3.1 Logic

The smallest unit of logic within the FPGA is a *logic cell*. This is the finest grain logic unit producing a single bit output. It typically consists of a small LUT and a latch on the output, as represented in Figure 2.5. A common terminology is to refer to a three-input LUT as a 3-LUT, and similarly for other LUT sizes. The latch is used to register the output. On some FPGAs, both outputs may be provided, as in Figure 2.5; on others, a multiplexer may be used to select whether the registered or unregistered signal is output. The registers can be used for building finite state machines, counters, and for registering data within pipelined systems.

The LUT can implement any arbitrary function of its inputs. More complex functions can be implemented by cascading multiple levels of LUTs by connecting the output, X, to the input of the next layer. FPGA manufacturers face a trade-off in choosing the size of the LUTs. Consider implementing a simple two-input AND gate. With a large LUT, a significant fraction of the logic would be unused. Smaller LUTs would achieve a higher usage of resources. However, when implementing complex functions with small LUTs, multiple logic cells would have to be cascaded, which can significantly increase the propagation delay. For complex functions, using a larger LUT would be preferable. To balance this, the silicon area of the table grows exponentially with the number of inputs, so at some stage it is more efficient to split a complex function over multiple LUTs. A larger LUT also has a longer propagation delay than a smaller table because of the extra levels in the multiplexer tree. Early studies showed that the optimum size is for the LUT to have four to six inputs (Kouloheris and El Gamal, 1991; Singh *et al.*, 1992). Early FPGAs used 3-LUTs and 4-LUTs. However, as device scales have shrunk, the proportion of the total propagation delay used by the routing matrix has increased. This has led to larger LUTs in more recent devices, for example recent Xilinx devices use 6-LUTs. Another approach taken is to combine two basic lookup tables, sharing some of the inputs, but allow the logic to be partitioned between the two outputs according to the application (Hutton *et al.*, 2004).

It is also usual to combine multiple logic cells into a logic block or tile, as in Figure 2.4. All of the logic cells within a block share common control signals, for example, common clock and clock enable signals. In a logic block, the outputs are directly available as inputs to other logic cells within the same block. The resulting direct connections reduce the number of signals that need to be routed through the interconnect matrix, reducing the propagation delay on the critical path. A small number of additional dedicated multiplexers allow more complex functions to be synthesised by combining adjacent LUTs to effectively create a larger LUT. Studies have shown that the optimal logic block size is four to ten logic cells (Ahmed and Rose, 2000).

Many of the early FPGAs were homogenous in that there was only one type of logic resource. However, as FPGAs developed and moved from being used primarily as glue logic to compute resources in their own

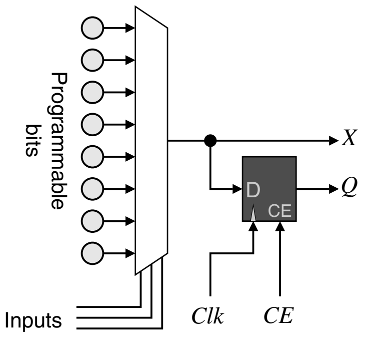


Figure 2.5 A logic cell is the basic building block of an FPGA.

right, implementing many functions using fine-grained logic was expensive. This has led to a more heterogeneous architecture, with a range of more complex coarser-grained building blocks.

Addition is a commonly performed operation. On an FPGA, a full adder would require two 3-LUTs, one to produce the sum and one to produce the carry. With a ripple adder, the propagation of the carry signal through the routing matrix is more likely to form the critical path. Since addition is such a common operation, many manufacturers provide additional circuitry within logic cells (or logic blocks) to reduce the resources required and significantly reduce the critical path for carry propagation. This usually consists of dedicated logic to detect the carry (so only one LUT is needed per bit of addition) and a dedicated direct connection for the carry signal between adjacent logic cells.

Multiplication is also a common operation in digital signal processing (DSP). A multiplier can be implemented using logic blocks to perform a series of additions, either sequentially or in parallel. Sequentially, multiplication is time consuming, requiring one clock cycle for each bit of the multiplier to perform the associated addition. Implementing the multiplication in parallel improves the timing, but consumes significant resources. Therefore, it is common in FPGAs targeting DSP and other compute intensive applications to have dedicated multiply or multiply and accumulate blocks.

On a fine-grained FPGA, all of the storage is within the flip-flops in the logic cells. While this is suitable for data that is accessed frequently, the storage space is limited, making it expensive for storing larger volumes of data. Rather than require the use of off-chip memories, many FPGAs have had small blocks of memory added on-chip for intermediate storage and buffers. This can be implemented in two ways – either by adding dedicated memory blocks (*block RAM*) or by adapting the structure of the logic cells to enable them to be used as a random access memory rather than as LUTs; such memory is called *fabric RAM* because it is made from the "fabric" of the FPGA. A high bandwidth is maintained by keeping the memory blocks small, enabling independent access to each block. Many of the memory blocks are also dual-port, simplifying the construction of FIFO buffers and custom caches. *True dual-port* allows independent reading and writing on each of the ports. Some memories are only *simple dual-port*, where one port is write only and one port is read only.

Often, some parts of an application benefit from the parallelism offered by an FPGA, and other parts are more efficiently implemented using a serial processor. In these latter cases, a serial processor may be constructed from the fabric of the FPGA. The delays associated with programmable logic and interconnect will limit the clock speed of such a processor. Therefore several high end FPGA families implement a high performance serial processor as a hardware block on the FPGA.

To summarise, fine-grained homogenous FPGAs consist of a sea of identical logic blocks. However, most current FPGAs have moved away from this homogenous structure, with a heterogeneous mixture of both fine-grained logic blocks and larger coarse-grained blocks targeted for accelerating commonly used operations. This has both advantages and disadvantages in terms of design. Heterogeneous systems will generally be faster because the coarse-grained blocks will not have the same level of overhead that comes with fine-grained flexibility. However, heterogeneous devices are also harder to programme, and if the mix of specialised blocks is not right much of the functionality may remain underused.

2.3.2 Interconnect

The purpose of the programmable interconnect is to flexibly enable the different logic blocks to be connected to implement the desired functionality. Obviously it is impractical to provide a dedicated direct connection from every possible output to every possible input. In any given application, only a very small fraction of these connections would be required; however, any of the connections potentially may be required. The solution is to have a smaller set of connection resources (routing lines) that can be shared flexibly to create the connections required for a given application, a little like the cables in a telephone network.

Since the interconnect takes up room on the silicon, there is a compromise. If only a few routing lines are provided, more space is available for logic, but complex circuits cannot be constructed because of insufficient interconnection resources. Alternatively, if too many routing lines are provided, many will not be used, and this reduces the area available for logic. There is an optimum somewhere in between these extremes, where sufficient routing lines are available for most applications, without using too much space.

The connection network for most FPGAs is based on a grid structure, similar to that shown in Figure 2.4. At each grid intersection some form of crossbar switch enables programmable connection between the horizontal and vertical routing lines.

Another factor that needs to be considered is the propagation delay through the interconnection network. Each switch that the signal passes through will add to the delay. This has led to a segmented structure, where not every routing line is switched at every junction. Also, dedicated direct connections can be provided between adjacent logic blocks to reduce the number of signals that need to go via the interconnect matrix. While having a range of different interconnection resources improves the flexibility and speed, it can make the process of finding good routes significantly more complex, especially for designs which use most of the logic available on the FPGA.

It is important that only a single output is driving a routing line. If two outputs are connected together, with one high and the other low, the power supply will effectively be shorted, causing damage to the chip. One of the roles of the FPGA programming tools is to ensure that this never happens. Devices that use bus structures for routing must limit the current of driving transistors, or use open drain and passive pull-up. Both approaches affect the speed and power dissipation.

2.3.3 Input and Output

Input and output blocks enable the connection of signals between the FPGA and external devices. The flexibility of FPGAs requires them to be able to connect to a wide range of other components. Most of the I/O pins can be programmed as inputs, outputs, or both, and be connected directly to the internal logic of the FPGA, or pass via registers. This range of functionality may be provided by a generic I/O block such as that shown in Figure 2.6, with each multiplexer controlled by a programmable configuration bit. The main complexity of I/O arises from the large number of different signalling standards used by different devices and ensuring appropriate synchronisation between the FPGA and the off-chip components.

Perhaps the most common signalling standard is Low Voltage Transistor–Transistor Logic (LVTTL) or Low Voltage CMOS (LVCMOS), which are suitable for general purpose 3.3V signals. FPGAs will almost certainly require a separate power supply for the I/O pins because the I/O voltages are often larger than the core supply voltage. Some standards may also require an external voltage reference to determine the threshold level between a logical 0 and 1. The voltage supply and reference levels will usually apply to a group or bank of I/O pins.

It is important when interfacing high speed signals to consider transmission line effects. A general rule of thumb is that any connection longer than about one-eighth of the rise time should have

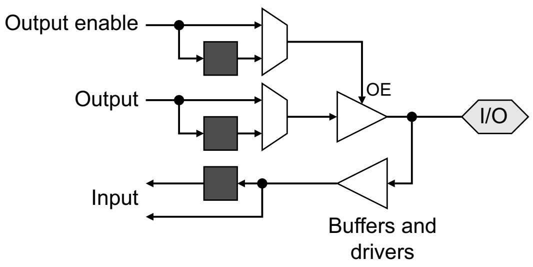


Figure 2.6 Generic input/output block.

appropriate termination to minimise signal reflections. As FPGAs have become larger and faster, it has become common practise to provide programmable on-chip termination. If not provided, it is necessary to ensure that appropriate termination is built off-chip for fast switching signals.

Just as dedicated logic blocks are provided in some FPGAs for commonly used functions, many newer FPGAs may also include dedicated I/O hardware to manage some commonly used connections and protocols. One example is the logic required to interface to *double data rate* (DDR) memories. Two data bits are transferred per clock cycle on each I/O pin, one on clock high and one on clock low. Multiplexing and de-multiplexing are built into the I/O block, enabling the internal circuitry to operate at the clock rate rather than twice the clock rate. Differential signalling is commonly used for high speed communication because it is less sensitive to noise and cross-talk. This improved noise immunity enables a lower signalling voltage to be used, reducing the power requirements. Most of the recent FPGAs include the logic for managing *low voltage differential signalling* (LVDS), which requires two I/O pins for each bit.

High speed serial standards have signalling rates higher than the maximum clock speed supported by the core of the FPGA. Support for such standards requires incorporation of parallel-to-serial converters on outputs and serial-to-parallel converters on inputs. The *serialisation and deserialisation* (SERDES) logic enables the FPGA to run at a lower speed than required by the communication data rate.

Many high level communication protocols would require significant FPGA resources simply to manage the protocol. While this can be implemented in the fabric of the FPGA, the design is often quite complex and is better implemented using a dedicated interface chip. Such chips typically manage the media access control (MAC) and physical signalling (PHY) requiring only standard logic signals in their interface with the FPGA. Some FPGAs incorporate dedicated MAC and PHY circuitry for commonly used communications protocols, such as PCI express and Gigabit Ethernet.

Most FPGAs provide a range of both general purpose and dedicated I/O connections.

2.3.4 Clocking

FPGAs are designed primarily as synchronous devices, so require a clock signal. Although multiple clocks may be used in a design, registers, synchronous memories and I/O blocks can only be controlled by one clock. A *clock domain* consists of all the circuitry that is controlled by a particular clock.

Since a clock drives a lot of inputs potentially over the whole of the FPGA, there are two special characteristics of clock lines that distinguish them from other interconnections on an FPGA. Firstly, it is important that all registers in a clock domain are clocked simultaneously. *Clock skew* is the difference in arrival time of the clock signal at successive registers in a design (Friedman, 2001). Any skew will reduce the timing margin and impact the maximum clock speed of a design. Although skew can be minimised by using H-tree or X-tree clock distribution networks, these do not fit well with the grid based structure of FPGAs (Lamoureux and Wilton, 2006). Instead, a spine and ribs structure is typically used (Figure 2.7),

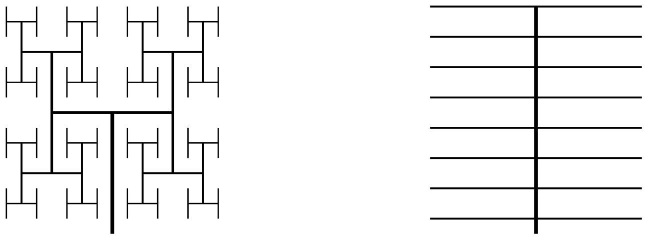


Figure 2.7 Clock distribution networks. Left: H-tree structure; right: spine and ribs structure.

where the clock is distributed to rows using a spine and then to the elements on the row using the ribs (Lamoureux and Wilton, 2006). A series of buffers is used within the distribution network to manage the high fan-out. In practise, it is a little more complex than this, as the network must cater for multiple clock domains, and many FPGAs also have a mixture of local clocks and global clocks.

*Delay locked loop*s (DLLs) are used to minimise the skew and to synchronise clocks with external sources. These dynamically adjust a programmable delay to align the clock edge with an external reference. A side effect is making the output duty cycle 50% and providing quadrature phase clock signals. While the delay locked loop will match the input frequency, *phase locked loops* (PLLs) can also be used to synthesise different clock frequencies by multiplying a reference clock by an integer ratio.

2.3.5 Configuration

All of the switches, structures and options within the FPGA are controlled by configuration bits. Most commonly, the configuration data is contained within the FPGA in static memory (SRAM) cells, although flash-programmed devices are available from some manufacturers. Flash-programmed FPGAs are non-volatile, retaining their configuration after power-off, so they can immediately begin working after powering-on. Static memory based FPGAs lose their configuration on power-off, requiring the configuration file to be reloaded every time the system is powered on. Static memory had the advantage of being infinitely reprogrammable; flash memory typically has a limited number of write cycles.

An FPGA is configured by loading the configuration file into the internal memory. This is usually accomplished by streaming the configuration file onto the FPGA either serially, or in parallel 8, 16 or in some cases 32 bits at a time. In a stand-alone system, the configuration file is usually contained within an external flash memory, such as that shown in Figure 2.8, with the board set up to automatically configure the FPGA on power-on. FPGAs can also be programmed serially via a JTAG (Joint Test Action Group) interface, and this interface is commonly used when programming an FPGA from a host computer system. The need to transfer the configuration into the FPGA each time the system is powered exposes the configuration file making the intellectual property contained within vulnerable to piracy or reverse engineering. The higher end FPGAs overcome this limitation by encrypting the configuration file and decrypting the configuration once it is within the FPGA.

FPGAs can also be reconfigured at any time after power-on. Since the configuration controls all aspects of the operation of the FPGA, it is necessary to suspend operation and restart it after the new configuration is loaded. No state information can be retained from one configuration to the next – all of the flip-flops are reset and all memories are usually initialised according to the new configuration. Therefore, any state information must be saved off-chip and reloaded after reconfiguration.

Some FPGAs can be partially reconfigured. The internal static memory controlling the configuration is addressed via internal registers, allowing the configuration for part of the FPGA to be accessed and changed. As before, this cannot be done while that part of the FPGA is operating; however, the rest of the

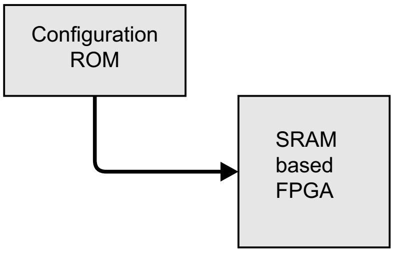


Figure 2.8 Configuration of static RAM-based FPGAs.

FPGA can continue executing normally. Partial reconfiguration requires the layout of the design to be structured in such a way that the blocks being reconfigured are within a small area on the FPGA to facilitate loading at run-time.

Some FPGAs also support read-back of the configuration from the FPGA. This feature can be useful for debugging a design, given appropriate tools, but is generally not used in normal operation.

2.3.6 Power Consumption

In embedded applications, the power consumption of the system, and in particular of the FPGA, is of importance. The power dissipated by an FPGA can be split into two components: static power and dynamic power. *Dynamic power* is the power required for switching signals from 0 to 1 and vice versa. Conversely, *static power* is that consumed when not switching, simply by virtue of the device being switched on. Most FPGAs are based on CMOS technology. A CMOS inverter, the most basic CMOS circuit, is shown in Figure 2.9. It consists of a P-channel pull-up transistor and an N-channel pull-down transistor. Only one of the transistors is on at any one time, and since the MOS transistors have a very high off-resistance, CMOS circuits typically have very low static power dissipation.

Any significant power is only dissipated when the output changes state. This results from two main sources (Sarwar, 1997). The first is from charging the capacitance on the output line. The capacitance results from the inputs of any gates connected to the output, combined with the stray capacitance associated with the wiring between the output and any downstream inputs. The power required is given by

$$P = NCV_{DD}^2 f \tag{2.1}$$

where N is the average number of outputs that are changing in each clock cycle, C is the average capacitance on each output and f is the clock frequency. For an embedded system, the power consumption may be reduced by considering each of the terms of Equation 2.1:

- by limiting the number of outputs that change with each clock cycle;
- by minimising the fan-out from each gate and minimising the use of long wires (to keep the capacitance lower);
- by reducing the power supply voltage (although the developer has little control over this, it has been reducing with each technology generation); and
- by reducing the clock frequency.

In terms of a given design, reducing the clock frequency probably has the most significant effect on reducing the power consumption, as it is often the clock signal itself that dissipates the most power. It is toggling with every cycle, and because the clock distribution network extends over the whole FPGA it also has a high capacitance. Minimising the number of clock domains and keeping the logic associated with each clock domain together can also help to limit the power used by the clock circuitry.

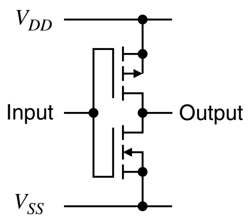


Figure 2.9 CMOS inverter.

The second component of dynamic power dissipation results from the fact that as one transistor is switching off, the other is switching on. During the transition there is a brief interval when both transistors are partially on and there is a conduction path between V_{DD} and V_{SS}. For fast transition rates, power dissipation from this source is typically much lower than that from charging the load capacitance. The flow-through current can be minimised by keeping the input rise time short, which can be managed by limiting the fan-out from the driving source. This flow-through current is becoming more important with reducing V_{DD} because the threshold voltages of the P- and N-channel transistors are brought closer together and both transistors are partially on for a wider proportion of the input voltage swing.

The static power dissipation arises from leakage currents from five different sources (Unsal and Koren, 2003) of which the subthreshold leakage is dominant. This is the leakage current through the transistor when the gate voltage is below the switching threshold and the transistor should be switched off. As the feature size and supply voltage of devices is scaled down, the threshold voltage must also be scaled down to maintain device speed. Unfortunately, the subthreshold leakage current scales exponentially with decreasing threshold voltage, so the leakage current contributes to an increasing proportion of the total power dissipation with each technology generation (Kao *et al.*, 2002; Unsal and Koren, 2003).

Several techniques can mitigate against this increase in subthreshold leakage current (Kao *et al.*, 2002). One is to use a higher threshold voltage in the sections of the circuit where the speed is not critical, and only use low threshold devices on the critical path. Another technique is to switch off sections of the circuit which are not being used, or at least place them on standby. Both of these techniques are used by FPGA manufacturers.

The user's design can significantly affect the power dissipated by an FPGA. In general, minimising the clock frequency and minimising the number of signals that transition every clock cycle can significantly reduce power dissipation. There may be limited control over the length of the interconnections, as much of this is managed automatically by the vendor's place and route tools. However, these tools are also programmed to minimise the length of the interconnections.

Another factor that can influence the design of embedded systems is the large current burst drawn from the power supply when the device is switched on. This results from the activity that takes place as the FPGA is configured. It is essential that the power supplies be able to handle this peak current, and not just the average required for normal operation. This is of particular concern in low power, battery operated devices where the peak current may be limited.

2.4 FPGA Families and Features

In the previous section the architectural features of FPGAs were reviewed in a general way. In this section, the particular characteristics of devices from several manufacturers are considered in more detail. The two main FPGA manufacturers in terms of market share are Xilinx and Altera, although there are several others that provide FPGAs. The current offerings from each of these are described in turn. Of particular interest from an image processing perspective is the size of the device in terms of logic resources, embedded memories, embedded multipliers or DSP blocks, and whether or not the device includes a processor core. (Note, where a processor core is not available, it is still possible to implement a soft processor using the logic resources.) For each family the range of resources across the different family members is listed. In the number of logic cells, 1 k represents 1000 cells. The RAM size is given in bits, where 1 K = 1024 bits. The total RAM figures represent the dedicated block RAM available on the FPGA, and do not include fabric RAM. The descriptions given are by necessity brief; more details may be found in the data sheets or product guides referred to; these are available from the manufacturer's web sites.

2.4.1 Xilinx

Xilinx was one of the first developers of field programmable gate array technology. It has had a number of families of devices, with the two current families being the Spartan series and the Virtex series.

The main difference between the two families is that the Spartan devices are designed primarily for low cost, and the Virtex devices are designed primarily for high performance. Both families are discussed here chronologically, outlining the key features that have been added with each succeeding generation. In the newest generation, the 7 Series devices, the Spartan family is replaced by the Artix and Kintex families. Within each generation, a range of device sizes is available. The key characteristics are summarised in Table 2.1 with more details to be found from the data sheets (the shaded lines in the table represent “mature” devices that are no longer recommended for new designs).

2.4.1.1 Virtex and Spartan-II

Although the Virtex and Spartan-II are now obsolete, it is instructive to look at their architecture to see the evolution with successive generations. Both the Virtex and Spartan-II are structurally similar, so will be described together.

The basic logic cell is a 4-LUT. Each logic cell also has a flip-flop on the output. However, a bypass input is also provided allowing the LUT to be bypassed, enabling the LUT and flip-flop to be used independently. A dedicated multiplexer combines the outputs of two adjacent logic cells to implement a 5-LUT or a 4:1 multiplexer. (Xilinx calls this pair of logic cells a *slice*.) Another multiplexer enables two slices to be combined, to implement a 6-LUT or 8:1 multiplexer. Additional logic is provided to accelerate arithmetic operations. The carry chain includes multiplexers for carry generation and propagation, along with an XOR gate to enable a full adder to be implemented within a single logic cell, as demonstrated in Figure 2.10. The carry chain propagates vertically up the columns within the FPGA. Each logic cell also provides a dedicated AND gate to reduce the logic required to implement multiplication.

Each logic cell can also be configured as a 16×1 bit fabric RAM or as a 16-bit shift register. The two logic cells within a slice may be combined to implement a dual-port memory. This fabric RAM is useful for storing coefficients or other small data structures where the memory depth is shallow. In addition to the fine-grained fabric RAM, both the Virtex and Spartan-II contain blocks of dual-port memory, where each block contains 4096 bits. The aspect ratio may be configured between 4096×1 to 256×16, making it useful for implementing row buffers and larger FIFO buffers.

There is a range of interconnection resources between the logic blocks. There are direct connections within a logic block and to adjacent logic blocks horizontally, enabling the logic cells to be chained efficiently. In addition to the direct connections, the routing matrix provides four types of connections:

- single length lines connecting adjacent tiles, both horizontally and vertically;
- hex lines that span six tiles, with a connection after three tiles;
- buffered long lines that span the whole chip, both horizontally and vertically; and
- horizontal bus lines. These can be driven by tri-state buffers within the logic blocks.

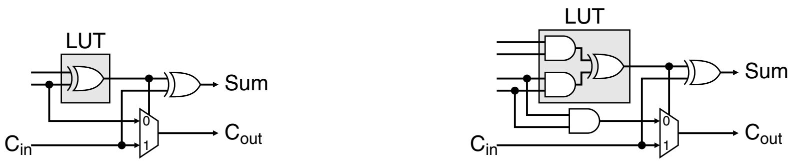


Figure 2.10 Left: logic required to implement a full adder with a single logic cell; right: a two-bit multiplication in a single logic cell. (Adapted from Elzinga *et al.* (2000); reproduced by permission of Xilinx, Inc.)

Table 2.1 Characteristics of the Xilinx FPGA families

Family and Generation	Process	LUT Size	Block Size	Logic Cells	RAM Size	Total RAM	DSP Size	DSP Blocks	Processor
Spartan-II (Xilinx, 2008b)	0.18 μm	4	4	432–5.3 k	Fabric: 64 256×16	16 K–56 K	—	—	—
Spartan-III (Xilinx, 2008c)	90 nm	4	8	1.5 k–66 k	Fabric: 64 512×36	72 K–1.8 M	18×18	4–104	—
Spartan-III DSP (Xilinx, 2008d)	90 nm	4	8	33 k–47 k	Fabric: 64 512×36	1.5 M–2.2 M	$18 \times 18 + 48$	84–126	—
Spartan-6 (Xilinx, 2009c)	45 nm	6 or 5×2	8	2 k–92 k	Fabric: 256 512×36	144 K–4.7 M	$18 \times 18 + 48$	4–182	—
Artix-7 (Xilinx, 2010a)	28 nm	6	8	11 k–220 k	Fabric: 256 512×72	720 K–12 M	$25 \times 18 + 48$	40–700	—
Kintex-7 (Xilinx, 2010a)	28 nm	6	8	19 k–254 k	Fabric: 256 512×72	2.3 M–29 M	$25 \times 18 + 48$	120–1540	—
Virtex (Xilinx, 2001)	0.22 μm	4	4	1.7 k–27 k	Fabric: 64 256×16	32 K–128 K	—	—	—
Virtex-II (Xilinx, 2007a)	0.15 μm/0.12 μm	4	8	512–93 k	Fabric: 128 512×36	72 K–128 K	18×18	4–168	—
Virtex-II Pro (Xilinx, 2007b)	0.13 μm/90 nm	4	8	2.8 k–88 k	Fabric: 128 512×36	216 K–7.8 M	18×18	12–444	PPC405
Virtex-4 (Xilinx, 2008e)	90 nm	4	8	12 k–200 k	Fabric: 64 512×36	648 K–9.7 M	$18 \times 18 + 48$	32–96	PPC405 (FX only)
Virtex-5 (Xilinx, 2008f)	65 nm	6 or 5×2	8	12 k–415 k	Fabric: 256 512×72	936 K–18 M	$25 \times 18 + 48$	32–1056	PPC440 (FXT only)
Virtex-6 (Xilinx, 2010d)	40 nm	6 or 5×2	8	46 k–474 k	Fabric: 256 512×72	5.5 M–37 M	$25 \times 18 + 48$	288–2016	—
Virtex-7 (Xilinx, 2010a)	28 nm	6	8	179 k–1222 k	Fabric: 256 512×72	14 M–63 M	$25 \times 18 + 48$	700–3960	—

In addition to these, there are global routing resources to distribute several clocks and other global signals (such as reset) with low skew. The clock drivers use a delay locked loop to synchronise skew free to either an external or internal source. The drivers can also divide down the signal or provide multiple phases if necessary.

Configuration of the FPGA is based on a frame structure, where each frame is associated with a column (or part of a column) within the FPGA. This enables partial reconfiguration of the FPGA, on a frame by frame basis. The configuration can also be read back from the FPGA for verification or debugging. Readback will also read the contents of any user registers (the flip-flops within I/O blocks or logic cells), and RAM (both fabric RAM and block RAM).

2.4.1.2 Virtex-II

Many of the changes in moving to the Virtex-II were relatively minor. The structure of each logic cell was basically unchanged, although a logic block is now made up of eight logic cells grouped in four slices of two logic cells each. As before, the logic cells may also be configured as fabric RAM or shift registers. An additional OR gate was added per slice to enable implementation of large sum of products chains (Xilinx, 2007b). Each logic cell can be configured as a 4-input AND gate, with the carry chain used to parallel adjacent logic cells to give wider input AND gates. The OR is placed at the top of each slice and can be linked horizontally to arbitrary widths. Dedicated multiplier blocks (18×18) were introduced with the Virtex-II, accelerating DSP and other compute applications.

The size of the block RAMs was increased to 18 Kbits, with configurable aspect ratio from $16\,\text{K} \times 1$ to 512×36. The extra bits can be can be used as additional storage, although they are intended as parity bits. However, the user is required to implement any parity generation and checking logic.

There was a small change to the hierarchical routing resources. The single lines were replaced with double lines which connect over one or two tiles, both horizontally and vertically. The direct connections were also extended to allow direct connection to diagonally adjacent logic blocks.

The clock network was also augmented with local clocks associated primarily with I/Os. The clock drivers were augmented to include phase locked loops to enable frequency synthesis by a ratio of integers. The I/O blocks were enhanced to provide digitally controlled termination and direct support for differential I/O and double data rate standards.

The Virtex-II Pro added a number of additional coarse-grained blocks (Xilinx, 2007b). A key new feature was the introduction of a PowerPC processor core to accelerate high performance computing applications. Another new feature to facilitate high speed communication was the multigigabit transceiver cores (called RocketIO by Xilinx). These cores include SERDES logic to enable the communication to take place at a much higher clock speed than is used internally for the logic.

2.4.1.3 Virtex-4 and Spartan-III

The previous generation of FPGAs saw an increase in the use of coarser grained block for implementing specific functions. However, different applications require a different mix of these blocks. For example DSP applications make extensive use of hardware multiplication; communication applications require the high speed connectivity; image processing makes extensive use of the internal RAM for buffering. Therefore, to provide a good mix of features targeted at specific application areas, Xilinx provides several FPGAs within each family that have a different mix of coarse-grained blocks. From an image processing perspective, of particular interest are the devices that provide a focus on logic, block memories, and in some imaging applications, DSP resources.

The biggest change within the logic blocks is that only half of the slices may now be used as fabric RAM or as a shift register. All of the slices within a logic block still have fast carry chains. All of the devices have multiplier blocks, although the Virtex-4 devices also include a 48-bit post-adder and accumulator, and the Spartan-III add to this an 18-bit pre-adder on one of the multiplier inputs. Both families use 18 Kbit block

RAMs, although the Virtex 4 also provides built-in logic to operate the block RAM as a FIFO without using additional logic resources.

The interconnect resources are the similar to those provided with the previous generation devices. Two notable exceptions are that the long lines only connect to every sixth tile, and the horizontal busses and associated tri-state buffers are no longer provided.

The clock managers introduced with the Virtex-II have been extended to the Spartan family as well. Similarly, the programmable I/O termination, support for differential signalling and double data rate interface have been extended to both families.

The Virtex-4 FX branch also includes the PowerPC core, high speed transceiver blocks and an Ethernet MAC. The Virtex-4 also introduces an internal configuration access port that makes the configuration memory of the clock managers and high speed I/O ports accessible from internal to the FPGA. This enables these devices to be reconfigured directly from within the fabric of the FPGA, or through the embedded processor. Also introduced with the Virtex-4 is encryption of the configuration file to prevent the design from being copied or reverse engineered. The decryption key is stored in volatile memory, which requires a battery backup to retain when the system is powered off.

The Spartan-3AN includes a serial flash memory on chip that can be used to configure the FPGA (Xilinx, 2009b). This has sufficient capacity for two complete configuration files, plus additional capacity for user applications.

2.4.1.4 Virtex-5

In the Virtex-5, the size of the logic cell increased to a 6-LUT, which may be fragmented to two 5-LUTs (sharing common inputs). The logic block architecture has four LUTs per slice, and two slices per logic block. As with the Virtex-4, only one of the slices can be configured as fabric RAM or a shift register. However, the increase to four LUTs per slice allows the fabric RAM to be configured as quad-port (using all four logic cells as separate interfaces), although only one port has read and write capability; the other ports are read only. A logic cell can be configured as a programmable 32-bit or dual 16-bit shift register.

The size of the block RAMs has been increased to 36 Kbits per block. The aspect ratio may be configured from $32\,\text{K} \times 1$ to 1024×36 as full dual-port, or as 512×72 simple dual-port. The block can also be partitioned as two independent 18 Kbit memories. Optional 64-bit error correction circuitry is built in, enabling detection of 2-bit errors and correction of single-bit errors.

New to the Virtex-5 are PCI express endpoint blocks in some branches of the family. This can simplify connection with other devices, or a host computer through the PCI express interface.

2.4.1.5 Virtex-6 and Spartan-6

The latest members of the Xilinx FPGA families available at the time of writing are the Spartan-6 and Virtex-6. These extend the storage per logic cell to two flip-flops. While all of the logic cells within the Virtex-6 provide arithmetic logic and carry chains, only half of those within the Spartan-6 do so. Similarly, approximately half of the Virtex-6 logic cells can be used as fabric RAM or shift registers, while only approximately one quarter of the Spartan-6 logic cells can be configured to do so.

The interconnect model within the Spartan-6 has also been changed to give more focus on local routing. The horizontal and vertical long-lines have been removed, with the following resources now available (Xilinx, 2010b):

- direct connections back to the inputs within the same logic block;
- single hop, to adjacent tiles horizontally and vertically;
- double hop, to two tiles away horizontally and vertically, or one tile diagonally;
- quad hop, to four tiles away horizontally and vertically, or two tiles diagonally.

While the Virtex-6 block RAMs have integrated FIFO logic and support for error correction, this has not been migrated to the Spartan family. However, new with the Spartan-6 is an integrated memory control block that simplifies the interface to common memory types. This transparently hides the buffering and control with high speed high bandwidth DDR connections (Xilinx, 2010c). The PCI express endpoint blocks and high speed transceivers previously only present in the Virtex family have now been migrated to some of the Spartan-6 FPGAs. It is interesting that Xilinx has not continued the FX series with the PowerPC processor into the Virtex-6 generation.

2.4.1.6 Virtex-7, Artix-7 and Kintex-7

Xilinx has recently announced its 7 Series of devices. This latest generation of Xilinx FPGAs all share a common underlying architecture, designed to simplify the portability of a design from one family to the other. At the logic level, the Xilinx-7 Series is similar to the Virtex-6, but implemented in an advanced 28 nm process technology. This gives improved performance while at the same time giving significant power savings over previous generations. The main differences between the families are the size of the devices and the target market. The Artix-7 family is optimised for low cost and power and is designed with a small footprint targeting high volume embedded applications. As before, the Virtex-7 family is targeted for highest performance and capacity. The Kintex-7 family sits between the other two families, giving a balance between cost and performance.

A new feature added with the 7 Series is a general purpose analogue to digital converter that gives 12 bits resolution at up to one mega-samples per second. While too slow for video, these A/D converters would be useful in many embedded control applications.

2.4.2 Altera

Altera currently provides three families of FPGAs: the low cost Cyclone series, the mid range Arria series and the high performance Stratix series (Table 2.2). The original Stratix used the same logic architecture as the Cyclone, but with the Stratix II Altera introduced a new adaptive logic module (Hutton *et al.*, 2004). None of the current families incorporate a processor core within the logic; the last to do so was the Excalibur FPGA, based on the now obsolete Apex20K family. In terms of embedded processors, Altera has focussed its efforts on its soft-core NIOS processor. However, the Excalibur is included in Table 2.2 for comparison.

2.4.2.1 Cyclone

The Cyclone series was designed for low cost applications, making it well suited for commodity embedded devices. This includes embedded image processing applications, especially distributed vision systems where a large number of sensors is required.

The basic logic cell is a 4-LUT with a register on the output. However, for arithmetic operations, the 4-LUT is partitioned into four 2-LUTs, enabling a full adder to be implemented using a single logic cell. Both the sum and the carry are implemented efficiently as a carry-select adder, with two 2-LUTs calculating each the sum and carry on the input signals, both without and with carry. These outputs are multiplexed by the carry-in, as demonstrated in Figure 2.11. This speeds up the addition or subtraction of wide operands because the LUTs are not in the critical path and the carry signals are generated in parallel for each section. A logic block consists of ten logic cells arranged vertically in a column.

In addition to the logic blocks, there are also blocks of dual-port memory, with 4608 bits per block. The aspect ratio of each port can be configured from 4096×1 up to 128×36, although the largest width is only

Table 2.2 Characteristics of the Altera FPGA families

Family and Generation	Process	LUT Size	Block Size	Logic Cells	RAM Size	Total RAM	DSP Size	DSP Blocks	Processor
Excalibur (Altera, 2002)	0.18 µm	4	10	4 k–34 k	128×16	32 K–256 K	—	—	ARM922T
Cyclone (Altera, 2008b)	130 nm	4	10	2.9 k–20 k	128×36	58 K–288 K	—	—	—
Cyclone II (Altera, 2008c)	90 nm	4	16	4.6 k–64 k	128×36	117 K–1.1 M	$2 \times 9 \times 9$ 18×18	13–150	—
Cyclone III (Altera, 2008d)	65 nm	4	16	5.2 k–119 k	256×36	414 K–3.8 M	$2 \times 9 \times 9$ 18×18	23–288	—
Cyclone IV (Altera, 2010b)	60 nm	4	16	6.3 k–150 k	256×36	270 K–6.3 M	$2 \times 9 \times 9$ 18×18	15–266	—
Arria GX (Altera, 2008a)	90 nm	8 Input ALM	8	8 k–36 k	32×18 128×36 $4\,K \times 144$	1.2 M–4.3 M	$8 \times 9 \times 9$ $4 \times 18 \times 18 + 52$ 36×36	10–44	—
Arria II GX (Altera, 2010a)	40 nm	8 Input ALM	10	6 k–102 k	Fabric: 640 256×36	783 K–8.3 M	$8 \times 9 \times 9$ – $2 \times 36 \times 36$	29–92	—
Stratix (Altera, 2006)	130 nm	4	10	10 k–79 k	32×18 128×36 $4\,K \times 144$	899 K–7 M	$8 \times 9 \times 9$ $4 \times 18 \times 18 + 52$ 36×36	6–22	—
Stratix II (Altera, 2007)	90 nm	8 Input ALM	8	6 k–72 k	32×18 128×36 $4\,K \times 144$	410 K–8.9 M	$8 \times 9 \times 9$ $4 \times 18 \times 18 + 52$ 36×36	12–96	—
Stratix III (Altera, 2008f)	65 nm	8 Input ALM	10	19 k–135 k	Fabric: 320 256×36 $2\,K \times 72$	1.8 M–14 M	$8 \times 9 \times 9$ – $2 \times 36 \times 36$	27–112	—
Stratix IV (Altera, 2010d)	40 nm	8 Input ALM	10	29 k–325 k	Fabric: 640 256×36 $2\,K \times 72$	6.3 M–22 M	$8 \times 9 \times 9$ – $2 \times 36 \times 36$	48–161	—
Stratix V (Altera, 2010e)	28 nm	8 Input ALM	10	239 k–1087 k	Fabric: 640 512×40	29 M–53 M	9×9 – $54 \times 54 + 64$	200–1840	—

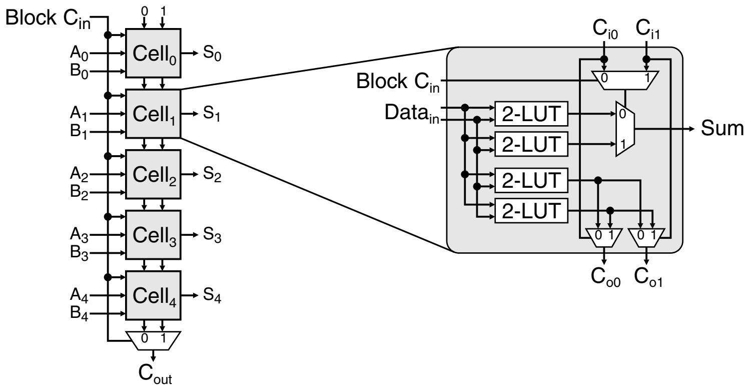

Block C_{in}
A_0 B_0 A_1 B_1 A_2 B_2 A_3 B_3 A_4 B_4
$Cell_0$ $Cell_1$ $Cell_2$ $Cell_3$ $Cell_4$
S_0 S_1 S_2 S_3 S_4
C_{out}
C_{i0} C_{i1}
$Data_{in}$
2-LUT
Sum
C_{o0} C_{o1}

Figure 2.11 Carry select adder as implemented by Altera. (Adapted from Altera, 2008b; reproduced by permission of Altera Corporation. Altera is a trademark and service mark of Altera Corporation in the United States and other countries. Altera products are the intellectual property of Altera Corporation and are protected by copyright laws and one or more U.S. and foreign patents and patent applications.)

available as simple dual-port. Each memory block can also be configured as a bank of tapped shift registers, with up to 36 outputs from a combination of taps and parallel registers.

Associated with each logic block is a local interconnect, which connects between logic cells within a block, or between adjacent blocks in the same row. More distant connections are made by driving horizontal and vertical lines which extend to the local interconnects of other logic blocks up to four tiles away.

In addition to the local data interconnects, there is a global clock network, which distributes up to eight clocks throughout the FPGA. The Cyclone has two phase locked loops, which can be used to synthesise clock signals by multiplying a signal by a ratio of integers. The PLL can also provide a programmable phase shift.

The basic I/O structure of the Cyclone is similar to that in Figure 2.6. A group of I/O blocks has a local interconnect in much the same way as the logic blocks, and it is primarily through this that the I/Os are connected to the logic. Selected I/Os have integrated circuitry to interface to DDR memories. Differential signalling support is also provided, although with an external resistor network for providing appropriate termination.

Configuration is based on a SRAM cell, which is programmed by serially loading the configuration data, either through dedicated pins or through the JTAG interface. The Cyclone FPGAs are not partially reconfigurable; the complete configuration must be loaded.

2.4.2.2 Stratix

The basic structure of the Stratix FPGA is similar to that of the Cyclone. The logic block also consists of ten 4-LUTs, with local interconnect. The biggest difference is the inclusion of DSP blocks, and a wider range of memory block sizes.

Each DSP block consists of a 36×36 bit multiplier. One of the unique features of this multiplier is that it can be segmented, enabling four parallel 18×18 multiplications, or eight parallel 9×9 multiplications. This latter feature is particularly useful for image processing, where 9-bit multiplications are suitable in many image processing operations. The 18×18 and 9×9 multiplications can operate with an associated 52-bit accumulator, making them suitable for DSP applications.

In addition to the 4608-bit memory block provided by the Cyclone, the Stratix also provides 576 bit and 72 Kbyte blocks. A 576-bit memory may be configured with aspect ratios from 512×1 to 32×18, with separate independent read and write ports. The two ports can be of different widths and in different clock domains, enabling them to be used as small FIFO buffers, or converting between parallel and serial data for lower speed serial I/O. The 72 Kbyte sized dual-port blocks can be configured with aspect ratios from 64 K × 9 up to 4 K × 144 (although the largest data width is only simple dual-port). This makes them suitable for storing larger data tables, or even buffering part of an image. Having a range of memory sizes enables the best size to be selected for an application without wasting too many resources.

The interconnect matrix of the Stratix extends that of the Cyclone by adding additional higher speed lines that span eight rows or columns from a logic block, and high speed lines that span 24 tiles horizontally or 16 columns vertically. The clock network is also significantly enhanced, with 16 global clocks and a number of regional clocks which drive each quarter of the FPGA. These enable up to 48 independent clock domains to be implemented on the FPGA. The phase locked loops of the Stratix can be reconfigured on-the-fly, enabling the clock frequencies and delays to be dynamically adjusted by an application.

The Stratix provides direct support for double date rate (DDR) memories and quad data rate (QDR) memories with two reads and two writes per clock cycle. This simplifies interfacing to external high speed memories. On-chip termination is provided for differential signalling, with dedicated high speed SERDES blocks for high speed serial signalling.

In addition to the serial configuration modes of the Cyclone, the Stratix can also be reconfigured in parallel, significantly reducing the configuration time, especially for the larger devices. It also has a register which can be used to control which of up to eight configurations is loaded, supporting dynamic update and reconfiguration of the FPGA with a small number of components. Partial reconfigurability is limited to reprogramming the phase locked loops.

2.4.2.3 Cyclone II, III and IV

The successive generations of the Cyclone family are based on much the same architecture as the original Cyclone. The basic logic cell is still a 4-LUT with a register on the output, although the logic block is extended to 16 logic cells. However, the use of the logic cell for performing arithmetic has been changed. Rather than use the carry select adder of the original Cyclone, a simpler ripple carry adder is used. The logic cell is split into two 3-LUTs; one implements the sum and the other the carry. The carry-out connects directly to the carry-in of the next logic cell in the chain.

The new generations incorporate dedicated hardware multiplication blocks. These are not as fully featured as the DSP blocks within the Stratix series, but accelerate designs where small multipliers are used. Each multiplier block can perform a single multiplication of two 18-bit numbers, or two multiplications of 9-bit numbers.

In the Cyclone II, the embedded memory blocks are the same size as the Cyclone. However, the Cyclone III and IV double the size of each block RAM to 9 Kbits. This can be configured with aspect ratios from 8 K × 1 to 256 × 36, although only simple dual-port is supported with the widest memory.

The interconnect with the original Cyclone was restricted to short local distances. These have been extended in the later generations to include higher speed horizontal lines which span 24 tiles, and vertical lines which span 16 tiles.

With each generation, the number of PLLs and global clock lines increases. The Cyclone III extends the size of the PLL multiplier and divider from 32 to 256, and the Cyclone IV extends this further to 512. This enables finer resolution clock frequencies to be synthesised with each succeeding generation. Both the Cyclone III and IV allow the PLLs to be reconfigured at run-time without having to reconfigure the entire FPGA.

The performance of the I/O blocks improves with each succeeding generation. The Cyclone III introduces programmable on-chip termination of outputs, although inputs still require external termination. From the Cyclone II onwards, support for memory interfaces is extended to QDR memories.

PCI express is also supported, although this requires an external PHY and an internal soft intellectual property core (available from Altera) to implement the required endpoint logic in the FPGA. The Cyclone IV GX introduces up to eight high speed transceivers, with SERDES logic built into the I/O block. Also included within the Cyclone IV GX is a hard-wired endpoint for PCI express, including the buffering and control logic and PHY-MAC layer.

From the Cyclone II, the configuration data can be compressed to reduce the configuration file size by up to 55%, depending on the complexity of the design and device use. The FPGA then decompresses the data as it is streamed into the FPGA. This can reduce reconfiguration times by up to a factor of two. Configuration compression is only supported with serial reconfiguration. The Cyclone III and IV also provide the option of parallel configuration, up to 16 bits per clock cycle.

2.4.2.4 Stratix II, III and IV

With the Stratix II, Altera replaced the standard 4-LUT logic cell with an adaptive logic module (ALM) (Hutton *et al.*, 2004). The key idea behind the adaptive logic module is that when a large LUT is used, much of the logic is often unutilised. By being able to adaptively adjust the effective size of the LUTs, the logic resources within the logic cell may be better used. This increases the effective logic density without a significant penalty in terms of chip area and routing overhead. It also has the advantage that larger functions which share common inputs may be packed within a single adaptive structure. In the Stratix II ALM, up to eight inputs may be combined with two adaptive LUTs to produce two outputs. The various combinations are (Altera, 2007):

- two independent 4-LUTs;
- a 5-LUT and 3-LUT within independent inputs;
- a single 6-LUT;
- two independent 5-LUTs, with two inputs shared between the LUTs;
- two 6-LUTs with the same function, with four inputs shared between the LUTs.

As well as the adaptive LUTs, each logic cell also contains two full-adders, enabling two result bits to be generated by each logic cell. Both the LUTs and adders may be used in a design, for example to apply logic to the inputs before adding, or even to use the result of the addition to select one of the inputs. This enables, for example, three input adders to be constructed, or very efficient implementation of functions such as (Altera, 2007):

$$\begin{aligned} Q &= \min(X, Y) \\ &= (X < Y) ? X : Y \end{aligned} \tag{2.2}$$

The outputs from the logic cell (from either the LUTs or the adders) can be optionally registered, with two registers per logic cell. The Stratix II has eight ALMs per logic block; this is increased to ten for the Stratix III and IV.

In the Stratix II, the ALMs can only implement logic. From the Stratix III on, some of the logic blocks can be configured as simple dual-port fabric RAMs. In the Stratix III, a logic cell may be configured as a 16×2 RAM, whereas in the Stratix IV it can be configured as a 64×1 RAM. The fabric RAM replaces the small 576-bit memory blocks that were available in the Stratix and Stratix II. The size of the other memory blocks was also changed with the Stratix III and IV. The size of the intermediate blocks was doubled, implementing RAMs from $8\,\mathrm{K} \times 1$ to 256×36. The size of the large blocks was reduced significantly, with each block implementing sizes from $16\,\mathrm{K} \times 8$ up to $2\,\mathrm{K} \times 72$. Again, the block memories are true dual-port except for the widest data words, where only simple dual-port is supported. Although parity bits are provided, if they are to be used for error detection and correction the user must provide the appropriate logic. The Stratix IV provides a hardware block for setting the parity bits and correction of single bit errors when the block RAM is 64 bits wide.

The architecture of the multipliers in the Stratix II is much the same as those in the original Stratix, although rounding and saturation units were introduced. These can significantly reduce the logic required to manage overflow in particular. In the Stratix III the multipliers were redesigned to natively support 9×9, 12×12, 18×18 and 36×36 multiplications. The DSP blocks can also be configured to provide a 54×54 bit multiplier to support double precision floating point multiplication. However, rounding and saturation logic is only provided on 18-bit multiplications.

The interconnect structure was simplified a little from the Stratix, by removing the intermediate length lines. The routing resources are made up of:

- connections within a logic block through the local interconnect;
- direct connections between horizontally adjacent logic blocks;
- carry and register chains directly from one block to another vertically;
- horizontal and vertical connections from a logic block that span up to four tiles in each of the four directions horizontally and vertically;
- fast horizontal connections that span 24 tiles left and right. This was reduced to 20 tiles left and right for the Stratix III and IV;
- fast vertical connections that span 16 tiles up and down. This was reduced to 12 tiles up and down for the Stratix III and IV.

In terms of I/O resources, each generation saw a general improvement in speeds available. Full on-chip termination of both input and outputs was introduced with the Stratix III. In the Stratix IV, dedicated circuitry was introduced to support a range of high-speed communication standards, including PCI express and gigabit Ethernet.

With the Stratix II, Altera introduced both compression and encryption of the configuration file. The decryption key is stored internally in non-volatile memory. Stratix devices do not support configuration read-back or partial reconfiguration (other than of the clock circuitry).

2.4.2.5 Arria and Arria II

The Arria and Arria II families are basically the same as the Stratix II and Stratix IV respectively, but with enhanced high speed transceiver blocks designed primarily for serial communication applications. Since these features are less relevant for image processing, they are not described further here.

2.4.2.6 Stratix V

With the recently announced Stratix V, Altera has made a number of minor changes over previous generations. The ALM has been redesigned to improve logic utilisation. It is now able to implement many seven input LUT functions. The number of registers per ALM has been doubled to four. The increased availability of registers will improve the speed of heavily pipelined designs and this is likely to be of benefit in high speed image processing applications. The DSP block has been redesigned to natively support variable precision from 9×9 (three of which can be built with a single DSP block) up to 54×54 (which requires four DSP blocks). The small sizes will be of value for image filtering, while the large size is sufficient to implement double precision floating point for numeric intensive applications.

Another area where there is significant change is the size of memory blocks. The ability to use some ALMs as fabric memory has been retained, but the block RAMs are now only available in

20 Kbit blocks rather than the 9 K and 144 K of the Stratix IV. The 20 Kbit blocks support error correction, and both fabric RAM and block RAM can be configured as dual-port RAM, FIFO or shift registers.

An increased range of hard intellectual property blocks is available. Most of these provide direct hardware implementation for high speed communication protocols, in particular PCI Express. This speeds the design, and reduces logic resources and system power requirements.

Partial reconfigurability is not currently available (at the time of writing) but full support for partial reconfigurability is planned in future releases of Altera's Quartus software tools.

2.4.3 *Lattice Semiconductor*

Lattice Semiconductor produces a number of FPGA families (Table 2.3). Its current families are the ECP series designed for low cost applications, the corresponding non-volatile XP series and the high performance SC/M family. The Lattice Semiconductor documentation does not list the process technology, so that is not listed in the table.

Table 2.3 Characteristics of the Lattice FPGA families

Family and Generation	LUT Size	Block Size	Logic Cells	RAM Size	Total RAM	DSP Size	DSP Blocks
LatticeECP (Lattice, 2008a)	4	8	1.5 k–32 k	Fabric: 128 256 × 36	18 K–498 K	8 × 9 × 9 4 × 18 × 18 36 × 36	0–8
LatticeECP2 (Lattice, 2008b)	4	8	6 k–95 k	Fabric: 64 512 × 36	55 K–5.2 M	8 × 9 × 9 4 × 18 × 18 36 × 36	6–42
LatticeECP3 (Lattice, 2010)	4	8	17 k–149 k	Fabric: 64 512 × 36	700 K–6.7 M	8 × 9 × 9 4 × 18 × 18 18 × 36	12–160
LatticeXP (Lattice, 2007)	4	8	3 k–20 k	Fabric: 128 256 × 36	54 K–396 K	—	—
LatticeXP2 (Lattice, 2008d)	4	8	5 k–40 k	Fabric: 64 512 × 36	166 K–885 K	8 × 9 × 9 4 × 18 × 18 36 × 36	3–8
LatticeSC/M (Lattice, 2008c)	4	8	15 k–115 k	Fabric: 128 512 × 36	1 M–7.8 M	—	—

2.4.3.1 ECP

The LatticeECP is based on a 4-LUT with a carry chain to speed arithmetic operations and a flip-flop on the output of each logic cell. Adjacent cells form a slice, with four slices combining to make a logic block. Additional multiplexers associated with each cell enable adjacent cells to be combined to create 5-LUTs, 6-LUTs or a 7-LUT within a single logic block. The carry logic enables one bit of addition or subtraction per logic cell. Some of the logic blocks can be configured as fabric RAM, with the LUTs within each slice providing a 16×2 single-port memory. Adjacent slices may be combined to create either a wider memory or add a second port, giving a simple dual-port fabric RAM.

In addition to the fabric RAM, there are 9 Kbit blocks of RAM with configurable aspect ratio from $8\,\mathrm{K} \times 1$ up to 256×36. The block RAMs are true dual-port, except for the widest data width ($\times 36$) which is only simple dual-port.

DSP blocks enable multiplication at a range of data sizes from 9–36 bits wide. The smaller sizes can combine the multiplication with an accumulator which has an extra 16 guard bits (giving a total of 34 bits for 9×9, and 52 bits for 18×18).

The system has four global clocks, plus an additional four local clocks in each quadrant of the FPGA. On the FPGA are up to four phase locked loops which can be used to provide the global clock signals. Dynamic clock multiplexers enable the application to switch between different clock sources at run-time without glitches.

As with most FPGAs, a wide range of I/O standards is supported. Adjacent I/O blocks can be combined to provide support differential signalling. Multiplexing is provided on inputs and outputs to enable interface with DDR memories. A delay locked loop is used to provide the required clock alignment for the memory strobe signals.

Lattice Semiconductor provides an internal logic analyser which is accessible through the JTAG interface. The necessary logic to support the logic analyser must be added to the user design at compile time.

2.4.3.2 ECP2 and ECP3

With the next generations of the ECP, the logic blocks were restructured. In the ECP2, carry chains and output flip-flops are only available in three of the four slices within each logic block. The size of the fabric RAM from those blocks that supported it was also reduced. However, the size of the block RAMs was increased to 18 Kbits with configurable aspect ratio. As with the ECP, this is true dual-port except for the widest data widths.

With the ECP2, Lattice Semiconductor introduced high speed SERDES blocks to support high speed serial connections. The logic also includes the corresponding physical coding sublayer including PCI express and Gigabit Ethernet. Designs use a combination of dedicated hardware and user logic to manage the protocol.

The DSP block was been completely redesigned for the ECP3 to optimise its speed. The input registers are connected as a shift register to simplify the moving of coefficients or samples when filtering. This is followed by a set of multipliers which can be configured to manage the range of word sizes from 9×9 to 18×36. A set of pipeline registers separates the multipliers and an adder network and accumulators.

The clock network of the ECP2 consists of eight global clocks, and an additional four local clocks are available in each quadrant of the FPGA. This was extended in the ECP3 by splitting the FPGA into up to 36 local clock regions (depending on the size of the FPGA) with eight local clocks available in each region.

Configuration file encryption was introduced with the ECP2, as was dual boot support. This enables two configuration files to be present in the configuration ROM. If one configuration file fails to load for some reason, the FPGA will automatically switch to the other configuration file. Therefore, if an error occurs during a remote configuration update (into the configuration ROM), there is always a working configuration file available.

2.4.3.3 XP and XP2

The LatticeXP has basically the same architecture as the LatticeECP, with the exception that there are no multiplication blocks. The XP also contains on-chip a non-volatile flash memory that may be used for configuring the FPGA on power-up. This saves the need for an external flash memory for configuration, reducing the chip count. It is also faster because the transfer from the flash memory to the SRAM-based configuration memory has a significantly higher bandwidth. A further benefit is that it provides some protection of the intellectual property because the configuration file is not visible every time the FPGA is powered on.

The LatticeXP also has a sleep mode, where the logic is disabled, and the outputs are tri-stated. In sleep mode, the current is reduced by up to three orders of magnitude over the normal running mode. The advantage of sleep mode over powering off is that normal functionality may be resumed very quickly. Note though, that the block RAM and register contents are not maintained during sleep.

Similar to the XP, the LatticeXP2 is a version of the LatticeECP2 with built-in flash. It has been optimised further over the LatticeXP so that configuration from the internal flash only takes a few microseconds, making the FPGA virtually instant-on. It is also possible to transfer the contents of the block RAMs back into the flash memory without changing the rest of the configuration. The internal flash memory may be updated while the device is running normally, enabling live update of the device configuration. The XP2 also includes a configuration decryption block to provide additional design security. This enables the configuration file to be encrypted before transferring it to the FPGA, whether to the on-chip flash memory or direct to the SRAM configuration memory.

2.4.3.4 SCP/M

The LatticeSCP/M family is designed primarily for high speed communications applications. The internal architecture is much the same as the ECP series. The main difference is in the advanced high speed I/O blocks incorporated into the SC/M family. These are implemented as hard ASIC blocks integrated onto the FPGA. A flexible hardware PHY supports a number of high speed communications standards including PCI express and gigabit Ethernet. Programmable on-chip termination reduces the need for external termination resistances on both inputs and outputs.

A novel feature added by the LatticeSCP/M is a microprocessor interface. This enables the FPGA to be placed on the microprocessor's bus, enabling high speed configuration or direct processing of data as a hardware coprocessor. This is a viable alternative to providing a series of FPGAs with a built-in processor.

2.4.4 Achronix

A relative newcomer in FPGA terms is the Achronix Speedster. This claims to be the world's fastest FPGA (Achronix, 2009), with clock speeds up to 1.5 GHz. These speeds are obtained through extensive pipelining – even the routing is broken into short pipelined segments to reduce the capacitance and enable high clock speeds. Internally, the FPGA is self-synchronous, by combining clock and data together to create a data token. The development tools map a conventional synchronous design onto the fine-grained pipelined architecture. The characteristics of the Speedster are summarised in Table 2.4, with key features described in more detail below.

Table 2.4 Characteristics of the Achronix Speedster family

Family	Process Technology	LUT Size	Block Size	Logic Cells	RAM Size	Total RAM	DSP Size	DSP Blocks
Speedster (Achronix, 2009)	65 nm	4	8	25 k–164 k	Fabric: 128 512 × 36	1.2 M–9.8 M	2 × 9 × 9 18 × 18	50–270

The logic cell consists of a conventional 4-LUT. These can be combined in pairs to create a 5-LUT. Half of the logic cells also have carry chain logic for implementing arithmetic operations. The output of each cell has a flip-flop that can be used either for storage or pipelining. To achieve high speed, there is a maximum of one level of logic per pipeline stage. A logic block consists of eight logic cells. This may be configured either as logic, or as a 16×8 simple dual-port RAM. Each logic block also has a clock domain converter which facilitates transfer of data tokens from one clock domain to another.

Two other block types are included in the Speedster FPGA. The first is an 18 Kbit dual-port block RAM with configurable aspect ratio from $16\,\text{K} \times 1$ to 512×36. The extra bits may be used for error correction, although this must be implemented through user logic. The second block type is a hardware 18×18 multiplication, which can be split into two 9×9 multiplies. Both blocks operate at up to 1.5 GHz.

The Speedster supports a wide range of I/O standards, including both differential and DDR signalling. For high speed I/Os, both input and output termination is provided on-chip, with the actual impedance value being set by external resistances on dedicated control pins. High speed serialisation and deserialisation enable serial data rates of up to 10.3 Gbps, with standards such as PCI express, gigabit Ethernet, SATA and so on supported directly in hardware.

Device configuration is either serial or parallel (8 bits wide). For serial configuration up to four serial flash devices may connected simultaneously, reducing the configuration time by a factor of four. Configuration encryption is supported, with the key stored in non-volatile memory.

2.4.5 SiliconBlue

Another recent newcomer to FPGAs is SiliconBlue Technologies. It has a series of low power FPGAs suitable for embedded applications (Table 2.5). The logic cell consists of a 4-LUT combined with carry logic and a flip-flop. A logic block is made up of eight logic cells. In addition to the logic, the FPGA also contains a number of simple dual-port block RAMs. The I/O block is similar to the basic block shown in Figure 2.6, although DDR and differential signalling are supported on some I/Os. Although the configuration is static RAM based, the iCE65 FPGAs have on chip a non-volatile memory that can optionally be used to reconfigure the device.

The low power of the iCE65 makes it well suited to battery powered embedded applications. The low power is achieved by keeping the clock rate as low as possible, or putting the FPGA into sleep mode when not processing data. Typical power dissipation when operating at 32 MHz (suitable for pipeline-based image processing from a camera) is 20 mW.

Table 2.5 Characteristics of the SiliconBlue iCE65 family

Family	Process Technology	LUT Size	Block Size	Logic Cells	RAM size	Total RAM
iCE65 (SiliconBlue, 2009)	65 nm	4	8	3.5 k–7.8 k	256×16	80 K–128 K

2.4.6 Tabula

Tabula is a new entrant in the FPGA area, announcing its ABAX family of programmable logic early in 2010. Tabula has taken a completely new approach to solving the problem of device densities and speed. The ABAX has a three-dimensional architecture, enabling it to stack more logic into a given area. However, since three-dimensional logic devices are still not yet practical, it is using a novel approach with time as the third dimension, which Tabula has called a SpaceTime architecture (Tabula, 2010b). The basic principle is to dynamically reconfigure the whole device (logic, memory and even interconnect) with

Table 2.6 Characteristics of the Tabula ABAX family

Family	Process Technology	Logic Cell	Block Size	Logic Cells	RAM Size	Total RAM	DSP Size	DSP Blocks
ABAX (Tabula, 2009, 2010a)	40 nm	4:1 mux	16	27 k–79 k	64×9 $2\,\text{K} \times 18$ $4\,\text{K} \times 18$	44 M	$18 \times 18 + 44$	0–160

every clock cycle (up to 1.6 GHz) enabling each block to be used for multiple different functions. This reuse reduces the number of logic blocks actually required and also shortens interconnect paths, resulting in a faster and more compact design. The key characteristics are summarised in Table 2.6.

The rapid reconfiguration is achieved by having a stack of up to eight active configurations available. The user clock is multiplied up by the number of active configurations to give the internal system clock. Then with each internal system clock cycle the active configuration is cycled between those available. Thus each component is effectively being time-multiplexed up to eight times each user clock cycle.

At the time of writing, full data was not available on the internal components of the ABAX FPGA. The logic cell appears to be based on a 4:1 multiplexer, but may be more general than this. Four of these are grouped into a slice, with associated arithmetic logic. A logic block appears to consist of four slices, giving a total of 16 logic cells per tile. However, since each block is used for different functions up to eight times per user clock cycle, the effective logic densities are up to eight times the physical densities listed in Table 2.6. On the output of each logic cell is a number of flip-flops to enable communication between the different time slices.

On the FPGA, there are three different sizes of block memory. The small 64×9 blocks can be used either as a register file or as a 6-LUT. The medium sized dual-port block can be configured from $16\,\text{K} \times 2$ to $2\,\text{K} \times 18$. It also has built-in FIFO control, reducing the logic needed to implement FIFO buffers. The large size blocks are single port with aspect ratio from $36\,\text{K} \times 2$ to $4\,\text{K} \times 18$. Dividing the user clock cycle into up to eight internal time slices also means that a standard single-port memory can be accessed up to eight times every user clock cycle, effectively creating a multiport memory with a large number of ports. The DSP block performs a pre-addition, an 18×18 multiplication, followed by a 44 bit sum. Like the rest of the FPGA, it is capable of running at up to 1.6 GHz with the internal system clock.

A wide range of I/O standards is supported, including DDR and LVDS signalling. High speed serial links have built in SERDES logic, although the protocols must be managed with user logic.

Although the primary target market is DSP for wireless networking, the ABAX devices should also perform well in an image or video processing context.

2.4.7 Actel

Actel provides a range of low power FPGAs, making them ideally suited for embedded applications. There are three main families: the Axcelerator, the ProASIC3 and the IGLOO. Also included in Table 2.7 for comparison is the SmartFusion family, which includes an ARM processor and programmable analogue circuitry in addition to a ProASIC3 based FPGA block.

The Axcelerator family is one-time programmable using anti-fuse technology. Once programmed, it cannot be reprogrammed. A big advantage of one-time configuration is that it can begin operating immediately on power-on. The basic logic cell is a 4:1 multiplexer, which makes it different from the other cells looked at so far. Associated with each multiplexer is additional logic, including a carry chain, to make a full adder. A logic block consists of four such cells combined with two flip-flops. Although the logic is

Table 2.7 Characteristics of the Actel FPGA families

Family and Generation	Process Technology	Logic Cell	Block Size	Logic Cells	RAM Size	Total RAM	Processor
Axcelerator (Actel, 2009)	0.15 μm	4:1 multiplexer	4	1.3 k–22 k	128 × 36	18 K–288 K	—
ProASIC3 (Actel, 2008b)	130 nm	3-LUT	1	384–75 k	256 × 18	0–504 K	—
IGLOO (Actel, 2008a)	130 nm	3-LUT	1	384–75 k	256 × 18	0–504 K	—
SmartFusion (Actel, 2010)	130 nm	3-LUT	1	1.5 k–12 k	256 × 18	36 K–108 K	ARM Cortex-M3

quite fine grained, it is relatively rich in memory resources, with a number of 4608-bit memory blocks. These memories are simple dual-port with configurable aspect ratio from 4 K × 1 to 128 × 36. The routing within the Axcelerator is relatively fast because the fuses do not introduce delay in the same way that pass transistors do. The lack of reprogrammability makes the Axcelerator best suited in embedded designs only where the functionality is not going to change.

The other families are based on flash memory programmed FPGAs. These also have the advantage that they are immediately ready on power-on, but with the ability to be reprogrammed if necessary. The basic logic element consists of a 3-LUT, which can also be configured as a D flip-flop. This makes them very fine grained and quite homogenous logically. Within the FPGA is a single block of 128 × 8 of user flash and a number of 4608-bit block RAMs. These can be configured from 4 K × 1 to 256 × 18 and are true dual-port within the ProASIC3 or simple dual-port within the IGLOO. The IGLOO also has hardware logic to use the RAMs as FIFO buffers. The larger devices are able to implement a 32-bit ARM processor as a soft-core block. While both families are low power, the IGLOO is designed particularly for ultra low power applications, making it ideal for embedded vision sensors. Both families are available in very small footprint packages (as small as 3 mm × 3 mm) making for an extremely compact design.

2.4.8 Atmel

The Atmel devices are relatively old technology, although they are still available. The main features are summarised in Table 2.8. The relatively small size of the FPGA (in terms of logic resources) limits what they can be used for in terms of image processing applications.

Table 2.8 Characteristics of the Atmel FPGA families

Family and Generation	Process Technology	LUT Size	Block Size	Logic Cells	RAM Size	Total RAM	DSP Size	Processor
AT40 K (Atmel, 2006)	0.35 μm	3	2	256–2.3 k	32 × 4	2 K–18 K	—	—
FPLIC (Atmel, 2008)	0.35 μm	3	2	256–2.3 k	32 × 4	2 K–18 K 36 Kbytes	8 × 8 In AVR	AVR

The logic cell consists of a 3-LUT, which are grouped two to a logic block. This enables a 4-LUT or full-adder to be implemented from each logic block. Each logic block also contains a single flip-flop. Small dual-port RAMs are distributed through the logic giving a relatively homogenous architecture. The devices are partially reconfigurable, enabling adaptive systems to be constructed where blocks of the design can be overwritten after the initial programming.

The FPLIC combines an AT40K FPGA with an 8-bit AVR microcontroller and 36 Kbytes of static RAM for the microcontroller instruction and data memory.

2.4.9 QuickLogic

QuickLogic used to make a range of FPGAs; its most recent family was the Eclipse (Table 2.9). Because of their age, the devices were relatively small, making them less suitable for many image processing applications. Their distinctive feature was that the logic was not LUT based, but consisted of a mixture of difference size AND gates and small multiplexers.

More recently, QuickLogic has moved from the FPGA market to providing much higher level application blocks that can be combined and customised for specific applications. A radiation tolerant version of the Eclipse is provided from Aeroflex that would be suitable for space-based applications.

Table 2.9 Characteristics of the QuickLogic FPGA families

Family and Generation	Process Technology	Logic Cell	Block Size	Logic Cells	RAM Size	Total RAM	Multiplier
Eclipse (QuickLogic, 2007b)	0.25 μm	Other	1	960–4032	128×18	45 K–81 K	—
Eclipse II (QuickLogic, 2007a)	0.18 μm	Other	1	128–1536	128×18	9 K–54 K	$8 \times 8 + 16$
Aeroflex UT6325 (Aeroflex, 2008)	0.25 μm	Other	1	1536	128×18	54 K	—

2.4.10 MathStar

An alternative approach to programmable logic is taken by MathStar with its Arrix Field Programmable Object Array (MathStar, 2007). It is like an FPGA, but rather than the fine-grained logic blocks, it consists of a set of much coarser-grained building blocks. The advantage of this is that the coarse-grained blocks may be implemented more efficiently than building complex logic out of LUTs. However, unless the mix of blocks matches that required by an application, the resulting design may make poor use of the available resources. MathStar targets applications where there is significant mathematical computation (including image and signal processing) that can effectively make use of the coarser-grained blocks.

The Arrix is a 16-bit device in that all data paths are 16 bits wide rather than working with each bit individually as is the case with FPGAs. The available building blocks are:

- An arithmetic and logic unit (ALU). These have a 16-bit data path and can perform addition, subtraction and a wide range of logic functions. The ALUs are controlled by a reconfigurable state machine. The control bits which specify the operation come with the data.
- A multiply and accumulate block, that performs a 16×16 multiply, with a 40-bit accumulator.
- A register file that contains a 64×20 simple dual-port memory. The register file can also be configured as a 6-LUT (producing 20 outputs) or as a FIFO buffer.

- A block RAM. Each block is a single-port 2048 × 76 memory.
- An external RAM interface that facilitates connection to external DDR memories.
- A general I/O interface.
- A high-speed LVDS block with separate transmitters and receivers. These have built-in FIFO buffers to manage the flow of data onto and off the chip.

Two forms of interconnect are used: nearest neighbour and shared longer lines. The longer lines must be registered, giving them a latency of at least one clock cycle (longer for longer distances). Such pipelining enables a high clock speed to be maintained regardless of the length of interconnection. Each interconnect path is 21 bits wide: 16 bits for the data, one bit to indicate valid data and four control bits are used to control the ALU function.

While not strictly an FPGA, it is appropriate to use the Arrix for image processing applications.

2.4.11 Cypress

Cypress Semiconductor provides a series of three programmable system-on-chip systems. They combine an 8-bit, 16-bit or 32-bit microcontroller with a range of relatively high level analogue and digital building blocks or peripherals. Again, the blocks are too high level to classify them as FPGAs, and the underlying system is based primarily on the microcontroller rather than programmable logic. While these devices are less suited to real-time image processing applications, they may be appropriate when combining serial image processing (performed by the microcontroller) with a range of other sensors in a low cost sensor network.

Cypress Semiconductor also makes a wide range of peripherals (USB, wireless nodes, etc.) that provide a relatively simple way of integrating these functions within an FPGA system. Some of these are discussed in more detail in Chapter 12.

2.5 Choosing an FPGA or Development Board

Many FPGA manufacturers also sell development boards or evaluation boards. These boards often consist of an FPGA with a set of peripherals to demonstrate the capabilities of the device within a range of targeted applications. In particular, most boards provide some form of external memory and some means of interfacing with a host computer (even if it is just for the downloading of configuration files). Other than this, the specific set of peripherals can vary widely depending on the target application and market. Boards aimed for education use tend to be lower cost, often with a smaller FPGA and a wide range of peripherals, enabling them to be used for laboratory instruction.

With the large range of both FPGAs and development boards available, how does one choose the right system? Unfortunately, there is no easy answer to this, because the requirements vary widely depending on the target application, and whether or not a development system or product is being built.

For image processing, the development system should have the largest FPGA that can be afforded, with sufficient peripherals to enable a wide range of applications to be considered and programmed. Some essentials for any image processing application:

- There needs to be some method for getting images into the FPGA system. Depending on the intended input source, there are a number of possibilities here. To come from an analogue video camera, some form of codec is necessary to decode the composite video signal into its colour components and to digitise those components. Some systems provide a codec to digitise VGA signals, allowing the capture of images produced by another computer system. Common interface standards to connect to digital cameras include USB, Firewire and CameraLink. Most development boards have a USB interface (although actually getting to the video stream is more complex than just the physical interface), but few will have other camera inputs.

- The development board must have some method of displaying the results of image processing. While the final implementation may not require image display, it is essential to be able to see the output image while debugging. Either a VGA output or digital video interface (DVI) is suitable.
- The system must have sufficient memory to buffer one or more frames of video data. While the final application may not require a frame buffer, just capturing and displaying an image will require buffering the frame unless the camera and display are synchronised. Only the largest FPGAs have sufficient on-chip memory to buffer a whole image, and this memory is probably better used for other purposes in most applications. Therefore it is usually necessary for the frame buffer to be in off-chip memory. Dynamic memory has a variable latency, making it less suitable for high speed random access, although is suited for stream processing where the access pattern is known in advance. Static memory has faster access, although is generally available in smaller blocks. Working with two or more banks of static memory can often simplify an algorithm by increasing the memory bandwidth.

For systems where the FPGA is installed as a card within a standard PC, the host computer usually manages the image capture and display. The interface between the FPGA and host is usually through PCI express. Managing the interface with the host computer can use considerable resources on the FPGA. Depending on the application, the interface may also be a bandwidth bottleneck within the system.

When implementing a target embedded product, power and size are often the critical constraints. This will involve eliminating any peripherals that are unnecessary in the final application. However, for debugging it may be useful to have additional peripherals accessible through an expansion port. To minimise the cost of a product, it is desirable to reduce the size of the FPGA used. However, it is important to keep in mind upgrades that may require a larger FPGA. Many families have several different footprints for each size FPGA. Choosing a footprint that allows a larger FPGA to be substituted will reduce the retooling costs if a larger FPGA is required in later models.

3

Languages

In Chapter 2, a low level view of the architecture of FPGAs was presented. To programme an FPGA at this level, it is necessary to break down the design into the fine-grained logic blocks available on the FPGA and build up the logic required by the application from these blocks. There are two main problems with representing designs at this level. Firstly, designing at such a low level is both tedious and error prone. Secondly, the portability would be limited to FPGAs that had the same basic architecture and logic block granularity. While it is possible to programme FPGAs at this level, it is akin to programming a microprocessor in assembly language. It may be appropriate for the parts of the design where the timing and resource use are critical. However, for most applications, this is too low level, and this is also the case for most image processing designs based on FPGAs.

Fortunately, the tools for implementing algorithms and systems on FPGAs enable working at a higher level. Implementing a working design requires several key steps, as illustrated by the generic design flow in Figure 3.1. Firstly, the design must be coded in some human readable form. This represents the design using a *hardware description language* (HDL) and captures the essential aspects of the design that enables it to be both simulated and synthesised onto an FPGA. Design representation is the primary focus of this chapter. Although many HDLs are based on software languages, they are not software but instead describe the hardware required to implement the design. For an FPGA implementation of an algorithm it is necessary to describe not only the algorithm, but also the hardware used to implement it. Most development environments enable the hardware description to be 'compiled' and simulated at the logical level to verify that it behaves in the intended manner.

Synthesis takes the logical representation describing the hardware and converts it to a device or gate level net-list representing the actual hardware that is to be constructed on the FPGA. This form defines the circuits in terms of basic building blocks, both at the logic gate level and in terms of the primitives available on the FPGA if they have been used within the description. It also defines the interconnectivity between the building blocks. For interchange between different tools, a net-list is often represented using *EDIF* (electronic design interchange format). Synthesis constraints control aspects of the synthesis process, such as whether to: optimise for speed or area; automatically extract and optimise finite state machines; force set and reset to operate synchronously; and so on. Gate level functional simulation verifies that the design has been synthesised correctly, and provides more information on the specific low level characteristics of the circuit than the higher level simulation.

The next stage is to map the net-list onto the target FGPA. There are two phases to this process. *Mapping* determines how the logic maps onto the specific components available on FPGA. In particular, this phase partitions the logic of the design into the LUTs, splitting complex logic over several LUTs, or merging logic to fit within a single LUT where appropriate. The *place and route* phase then associates these

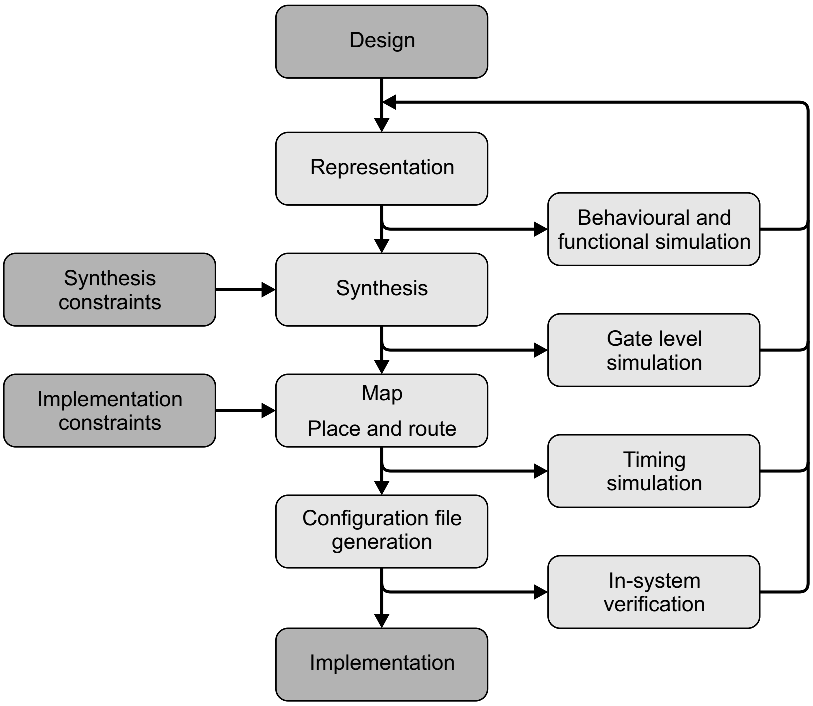


Figure 3.1 Steps to implementing a design on an FPGA.

mapped components with particular logic blocks on the FPGA and determines the routing required to connect the logic blocks, memories and I/Os. Implementation constraints direct the placement of logic to specific blocks, in particular associating inputs and outputs with specified pins. They also specify the required timing, which is used to guide the placement and routing of critical nets. Once the design has been mapped onto the FPGA, more accurate estimates of the circuit timing are available based on the lengths of the connections, fan-outs and particular resources that are used to implement the logic function.

The final stage is to generate the configuration file required to programme the FPGA. During development, the configuration file can be loaded onto the target system to verify that the design works as intended and that it correctly interacts with the other components within the system. In the deployed system, the FPGA is programmed by automatically loading the configuration file from flash memory on power-on.

As can be seen from Figure 3.1, implementing a design on an FPGA is quite different from implementing a design in software, even on a processor within an embedded system. It requires a hardware mindset, even during the initial design. However, image processing is usually considered to be a software development task. Successful implementation of algorithms on an FPGA therefore requires a mix of both software (algorithmic) and hardware (logic circuit design) skills.

Most FPGA-based design is carried out at an intermediate level that is a mixture of both algorithmic and circuit level design. The implementation of synchronous systems is usually performed at the *register transfer level* (RTL). As illustrated in Figure 3.2, this structures the application as a series of alternating blocks of logic and registers. The structure is not necessarily linear, as implied by Figure 3.2; from a computational perspective, the primary purpose of each block of logic is to determine the next value that will be stored into the registers on its outputs. It can take its inputs from anywhere: other registers, I/O pins or internal memory blocks. As demonstrated in Figure 3.2, inputs, outputs and memory accesses may be

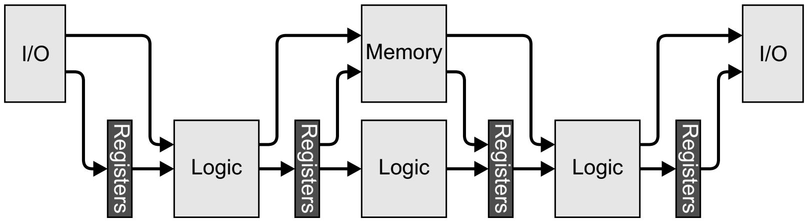


Figure 3.2 Basic structure of any application – logic interspersed with registers.

either direct or via registers. The blocks of logic are not directly clocked; all of the clocking is through the registers (and also the memory if it is synchronous). Therefore, the signals must propagate through the logic before they can be clocked into the registers. This limits the clock speed of the design to the propagation delay through the most complex block.

The art of implementing an algorithm on an FPGA therefore falls to decomposing the algorithm into a sequence of register transfers, and determining the appropriate logic between each register. Note that, unlike implementing algorithms in software, each logic block operates in parallel. This usually requires adapting the algorithm from a conventional sequential form into one that can exploit the parallelism available from hardware.

Implicit within Figure 3.2 is the control logic required to sequence the steps within the algorithm. Depending on the algorithm and its implementation, not all of the registers may necessarily be updated at every clock cycle. This can be achieved by using the clock enable for the registers to control when they are updated. Also, the computation to produce a value for a register may not necessarily be the same every cycle. This requires a multiplexer on the inputs of the registers to select the appropriate computation depending on the current stage within the algorithm. These are made more explicit in Figure 3.3, with control logic providing the clock enable and multiplexer selection signals. Implicit within the compute logic are any memory elements that may be used to perform part of the computation, or provide cached input or output data.

The distinction between control logic and compute logic can help with the transition from an algorithm to hardware. The control logic determines what is computed (or more precisely which computation gets saved in which register – in hardware, the logic is always adjusting its output to reflect changes on its inputs), while the compute logic actually performs the computation.

Since the computation is often context dependent, a common implementation of the control logic is as a finite state machine, with the context encoded in the states. Figure 3.4 gives two examples of simple finite state machines. The one on the left illustrates how a sequential algorithm may be implemented on

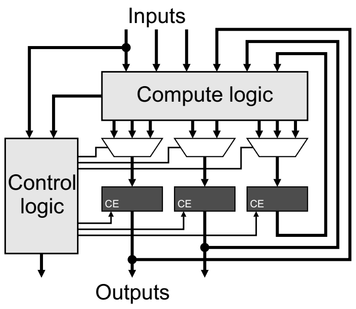


Figure 3.3 A rearrangement of Figure 3.2 to explicitly separate the control and compute logic.

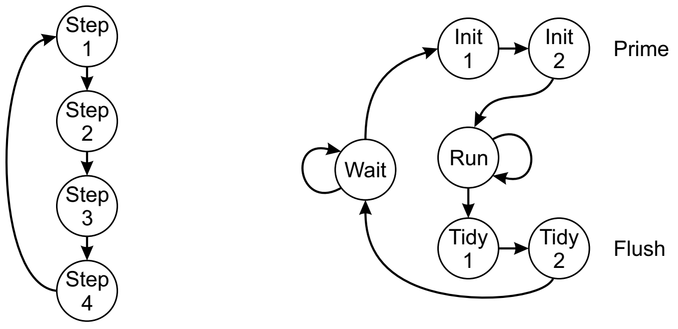


Figure 3.4 Example finite state machines. Left: a repeating sequential process; right: a pipelined stream process.

parallel hardware. The algorithm consists of four steps, each of which take exactly one clock cycle, and when the algorithm has completed the fourth step, it returns to the start (perhaps operating on the next data item). The finite state machine on the right represents the typical structure for the control for a three-stage pipelined stream process with one clock cycle per pixel. The algorithm begins in the *Wait* state, waiting for the first pixel. When the first pixel arrives, the next two states are initialisation, priming the processing pipeline. Once primed, the pipeline stays within the *Run* state while pixel data is arriving. At the end of each block of data, the state machine transitions to the *Tidy* states where the pipeline is flushed, completing the calculations for the last pixels to arrive. It then returns to the *Wait* state, waiting for the first pixel on the next block.

The transitions from one state to the next may be determined by input (control) signals or, for more complex algorithms, determined by the data being processed, as reflected in Figure 3.3. In practise, an algorithm is decomposed into a series of modules. Rather than control everything with a single large finite state machine, the control logic is also modularised, with a separate state machine for each module.

Having looked at the basic structure of an algorithm when implemented on an FPGA, the next step is to look at the representation of the algorithm being programmed onto the FPGA.

3.1 Hardware Description Languages

The traditional approach to describing electronic circuits is through schematic diagrams. These provide a graphical representation of the components and connectivity. They are best used for representing smaller designs at the transistor and gate level. For higher level designs, they represent modules and packages as block diagrams. Modern schematic capture tools integrate circuit simulation and printed circuit board layout, making them invaluable for the production of physical systems. High end tools also integrate hardware description languages allowing the complete development and modelling of systems containing FPGAs and other programmable hardware. However, the level of abstraction of schematic diagrams is typically too low for representing complex algorithmic designs. At the lowest level, they can represent logic circuits as transistors and logic gates. At the register transfer level, a schematic representation is structural, as a collection of components or block diagram, rather than clearly representing the behaviour or algorithm. This makes them difficult to navigate for algorithmic designs, because the behaviour is not particularly transparent from a structural representation.

From these roots, hardware description languages evolved in the 1980s to address some of the limitations and shortcomings of schematic representations, and to provide a standard representation of the behaviour of a circuit. Since hardware is inherently parallel, it is necessary for an HDL to be able to specify and model concurrent behaviour down to the logic gate level. This makes HDLs quite different from conventional software languages, which are primarily sequential.

The two main HDLs are VHDL (very high speed integrated circuit HDL) and Verilog, both of which have since been made into IEEE standards (the current standards are IEEE, 2009, 2006a, respectively). Their original purpose was for documenting and modelling electronic systems. Both are hierarchical structural languages in that they describe the hardware structurally in terms of logic blocks and their interconnections. They also allow a behavioural description of the functionality of the circuit, rather than necessarily having to specifically identify the logic circuits. A behavioural description, as its name implies, describes how the circuit is to behave, or respond to changes on its inputs. It is, therefore, the role of the synthesis tools to determine the actual logic required to implement the specified behaviour.

The primary focus of VHDL and Verilog is for verification and simulation of logic circuits. Therefore, modules can readily be written in either language for testing the functionality of the circuits described. Through the design of appropriate test benches, this allows the design to be thoroughly tested for correctness before it is synthesised. The complexity of synthesising a circuit from its behavioural description limits what can be synthesised. Structural designs can be readily synthesised, as can behavioural descriptions that conform to an RTL coding style. However, these restrictions mean that many of the constructs available within both languages are not synthesisable.

3.1.1.1 VHDL

VHDL has its roots in the US Department of Defence and was built on the Ada language. VHDL therefore inherits many of the characteristics of Ada, including being strongly typed and having a tendency to be somewhat verbose. The synthesisable subset of VHDL is specified in (IEEE, 2004).

Perhaps the best way to briefly describe the language is to demonstrate its functionality through a simple example. This will implement the recursive streamed filter with input *I* and output *Q*:

$$Q_n = \begin{cases} I_n, & |I_n - Q_{n-1}| < t \\ Q_{n-1} + k(I_n - Q_{n-1}), & \text{otherwise} \end{cases} \tag{3.1}$$

where *k* and *t* are constants. The corresponding schematic for this is shown in Figure 3.5. One VHDL representation of this is given in Listing 3.1.

Listing 3.1 VHDL implementation of the filter illustrated in Figure 3.5

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

package globals is
   constant PIXEL_WID : integer := 8;
   subtype PIXEL_VALUE is STD_LOGIC_VECTOR (PIXEL_WID-1 downto 0);
   subtype SIGNED_PIXEL is STD_LOGIC_VECTOR (PIXEL_WID downto 0);
   subtype SIGNED_PROD is STD_LOGIC_VECTOR (PIXEL_WID*2 downto 0);
end package globals;

----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use globals.ALL;
----------------------------------------------------------------
entity filter is
   generic (k : SIGNED_PIXEL;  -- Fixed point fraction -1 to 1
20              t : PIXEL_VALUE := "00001010"); -- Threshold level
21     port (clk, ce : in STD_LOGIC;
22          data_in : in PIXEL_VALUE;
23          data_out : out PIXEL_VALUE);
24  end entity filter;
25
26  ------------------------------------------------------------------
27  architecture filter_implement of filter is
28      component absv
29          port (s_in : in SIGNED_PIXEL;
30                u_out : out PIXEL_VALUE);
31      end component;
32
33      signal diff : SIGNED_PIXEL;
34      signal q, av : PIXEL_VALUE;
35
36  begin
37     abs_1 : absv port map(diff, av);                     -- Structural style
38
39     diff <= ('0' & data_in) - ('0' & q);                 -- Concurrent style
40     data_out <= q;
41
42     process (clk)                                        -- Behavioural style
43         variable prod : SIGNED_PRODUCT;
44     begin
45         if rising_edge(clk) and ce = '1' then
46             if av < t then
47                 q <= data_in;
48             else
49                 prod:= diff * k;
50                 q <= q + prod(PIXEL_WID*2 downto PIXEL_WID+1);
51             end if;
52         end if;
53     end process;
54  end architecture filter_implement;
```

The main features of VHDL will be demonstrated through a line-by-line explanation of the code in Listing 3.1. The first two lines indicate that the following code will make use of definitions from within the `STD_LOGIC_1164` package within the `IEEE` library. The basic representation of a single bit wire is `STD_LOGIC`, which can represent not only logic 0 and 1, but a range of other levels as well including a weak high or low, high impedance, and undefined. When synthesised, this will be either 0 or 1 (or high impedance if the output of a tristate buffer), but the other states are useful for detecting errors during simulation. A multi-bit number is represented by the type `STD_LOGIC_VECTOR`.

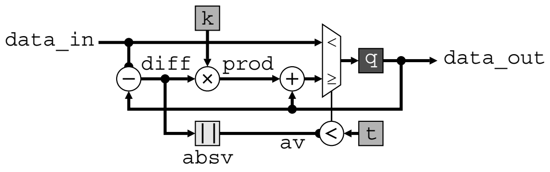


Figure 3.5 Schematic representation of Equation 3.1.

Since VHDL is strongly typed, the types of every operand and the result to which it is assigned must match exactly; any mismatch will result in a compiler error. This helps to ensure the correctness and internal consistency of the specification, but does mean that the developer has to do extra work to make everything match. To simplify this, a `package` is defined in lines 4–9, which defines the width of a pixel, and a number of types used within the project. Normally the package will be in a separate file and be used by all of the files within the project, effectively centralising the definitions. In this case it is included within the same file for compactness.

Lines 12–16 list the packages that will be used in the following definitions. The `STD_LOGIC_ARITH` overloads the definitions of arithmetic operations for the `STD_LOGIC_VECTOR` types, and these are extended in `STD_LOGIC_SIGNED` for signed arithmetic. Line 16 includes the package containing our type definitions.

Every block within VHDL consists of two parts: an `entity` (lines 18–24), which defines the block and its interfaces, and an `architecture` (lines 27–54), which describes the implementation. Within the entity, `generic` defines parameters of the `entity` – in this case the filter gain and threshold value. The threshold `t` is given a default value of 10. Rather than enter the value as a bit string, the conversion function `CONV_TO_STD_LOGIC_VECTOR()` defined in `STD_LOGIC_ARITH` can be used. A value needs to be given for `k` when the filter is instantiated and, if a value is given for `t`, it will override the default. Making them `generic` generalises the code. The `port` defines all of the data connections for the block. Here, a clock and clock enable signals are required to control the register internal to the filter. The data input and output are defined as `PIXEL_VALUE`, making use of the previously defined type. Note that the `generic` items must be constants; if they are to be controlled by other logic they need to be specified within the `port` list to enable them to be connected with whatever is controlling their value. The `entity` definition contains all that is required to connect the filter to other components within the design.

The `architecture` block defines the implementation of the `entity`. An `entity` may have several `architecture`s associated with it, each describing a different implementation. The head of the `architecture` block defines the `component`s and `signal`s used within the `architecture`. A `component` is any other external `entity` that will be instantiated within the body of the `architecture`. In the example here, lines 28–31 define the absolute value block in Figure 3.5, which is going to be instantiated by another VHDL entity called `absv`. Lines 33 and 34 define the `signal`s used within this implementation; these are labels for any internal wires or registers. The names here correspond to the labels within Figure 3.5.

There are three distinct coding styles that can be used within the body of an `architecture`. The first style is a structural style. This builds up a design by instantiating and connecting a number of `component` blocks. On line 37, an instance of the `absv component` called `abs_1` is incorporated into the design. The `port map` defines the connections to the `component` instance; here `diff` is connected to the `s_in port` and `av` to the `u_out` output. A purely structural implementation would describe everything in this manner, using a hierarchy of connected components in much the same way that the schematic diagram would represent the circuit. More complex structural implementations can use `generate` statements and loops to instantiate multiple copies of a `component` for example to build multi-bit circuits from single bit `components`.

The second programming style is illustrated in lines 39 and 40. It places more focus on the flow of data through the block, and describes the implementation in terms of concurrent `signal` assignments. Whenever any of the `signal`s on the right hand side changes, the corresponding statement is evaluated with the result immediately assigned to the left hand side. Line 39 subtracts `q` from `data_in` to produce the `signal` labelled `diff`. It is necessary to concatenate a 0 bit to the front of each of the terms to be able to represent the sign of the result. Line 40 connects the output `port` of the filter to the output of the register.

The third programming style defines the implementation in terms of its behaviour through one or more `process` statements. Each `process` block is executed whenever any of the `signal`s within its sensitivity list changes. The `process` on lines 42 to 53 will be executed whenever `clk` changes. The statements within a `process` block are executed sequentially, making it a little more like a conventional programming language. If written in an RTL style, such a behavioural description can be converted to the corresponding logic by the synthesis tools. Within the header, any `variable`s used within the `process` are defined. A `variable` is a little like a `signal`, with the difference being that an assignment to a `variable` takes place immediately within the process, whereas any assignment to `signal`s is made whenever a `wait` statement or the end of the `process` is reached. At that point, all `signal` assignments within the process are made concurrently; up to that point, all `signal`s will have the value when execution of the `process` started. That is `variable` assignments are made sequentially, whereas `signal` assignments are concurrent.

The `if` statement on line 45 specifies that the enclosed statements are only executed on the rising edge of the clock when the clock enable is set (the `rising_edge` function is defined in `STD_LOGIC_1164`). Consequently, any `signal` assignments within this block will be synchronous, implying that a register is required. The `if` on lines 46 to 51 imply a multiplexer, with each branch providing an appropriate input for the `signal` (register) `q`. Line 49 performs the multiplication and assigns the result to the variable `prod`. The multiplication operation is defined in `STD_LOGIC_SIGNED` to perform a signed multiplication, producing a full precision output. The least significant bits are truncated and the result accumulated in `q` in line 50.

As can be seen, the different programming styles may be mixed within any given design. All statements within the `architecture` are executed concurrently, in the sense that whenever any of the inputs change, the corresponding statement is evaluated.

While this example illustrates many of the features and characteristics of VHDL, it is by no means exhaustive. The interested reader should consult the standard (IEEE, 2004; 2009), the documentation for their synthesis tool or other references on VHDL (for example Bhasker, 1999; Ashenden, 2008).

3.1.1.2 Verilog

Verilog has many of the same features as VHDL. However, in contrast with VHDL which has its roots in Ada, Verilog is loosely modelled after C. Consequently, Verilog is more compact that VHDL. Again, as primarily a modelling language, not all language constructs are synthesisable (IEEE, 2005).

Each block within a design is specified as a `module`, with associated input and output ports. A structural design defines its operation by referencing lower level modules, and describes the interconnections between the module instances.

Within the module, there are two types of variables: `wire` and `reg`. A `wire` is as it sounds, a wire, while a `reg` is a variable that holds its value until it is changed (this is often, but not necessarily, associated with a flip-flop). Wider variables are created as arrays of bits. An `assign` statement performs continuous assignment. Whenever any variable on the right hand side of such a statement changes, the assignment is evaluated to determine the new value for the left hand side. The assignments are effectively all concurrent; the order that they appear within the module is immaterial.

An `initial` block contains sequential statements, with the block executing once on initialisation. An `always` block similarly executes a block of sequential statements whenever one of the variables in its sensitivity list changes. Within such blocks, there are two types of assignment: sequential, where the left hand side is updated immediately; and concurrent which waits until the next time unit or clock cycle, when all such assignments are made in parallel.

As with VHDL, a range of programming styles may be used. A full description of Verilog is beyond the scope of this book. The interested reader is referred to the standards (IEEE, 2005; 2006a) or other language references (for example Bhasker, 2005).

3.1.1.3 Summary

Standard HDLs are very good at describing the structure of a hardware design. Structural descriptions generally give the best performance, using the minimum of resources (if optimised appropriately by the designer). This makes them good for developing libraries of intellectual property (IP) blocks that may be instantiated many times or in many designs. However, design at the structural level is quite low level and can be very tedious. At a higher level, HDLs can model functionality through a behavioural specification. This describes what the circuit does, without necessarily specifying how it is to be implemented. Consequently, the designer generally has less control, so although the design is at a higher level, it is relying on the synthesis tools to derive an efficient implementation. Only the subset of behavioural descriptions that conform to an RTL style is synthesisable.

The low level of HDLs makes exploration of the *design space* (the trade-offs between area, speed, latency and throughput) more difficult. It is time consuming enough to programme one design, let alone several different designs to investigate the effects of different levels of loop unrolling and pipelining.

HDLs, by necessity, are general purpose and allow considerable control over how the circuit is implemented. This is both an advantage and a disadvantage. The advantage is that the developer can target specific structures available on the FPGA, or optimise the design for speed or resources required by programming in particular ways. By giving the designer control over every aspect of the design, fast and efficient designs are possible. The disadvantage is that the designer must control everything in quite fine detail, including both the data and control flows. The flexibility comes from the relatively low-level constructs, but programming at this level requires a lot of basic bookkeeping. This makes complex algorithmic programming more difficult.

The low level of programming is also not always particularly transparent. Very similar behavioural constructs map to latches or multiplexers. While a structural design can specify registers explicitly, they tend to be implicit in the behavioural programming style. These characteristics require close attention be paid at the low level, and has a strong focus on the hardware rather than the algorithm. Compared to software-based languages, HDLs are a lot like programming in assembly language.

3.2 Software-Based Languages

In an attempt to overcome this problem, over the last fifteen years considerable research has gone into the synthesis of hardware from high level languages, in particular from C. The rationale behind this is twofold. Firstly, it enables existing software to be easily ported or recompiled to a hardware implementation. Many algorithmic designs are first developed and tested using C or MATLAB®. Once working, the design is then often re-entered manually into an HDL after which it is necessary to verify and refine the design manually. This process is both time consuming and error-prone. Having a single source that can be compiled to both hardware and software would speed the development process considerably. It would also enable the partitioning of the final implementation between hardware and software to be explored more easily. The time critical and easily parallelisable components can be compiled to hardware, with the more sequential components compiled as software. A second motivation is to raise the level of abstraction to make hardware design more like software design, making it more accessible to software engineers (Alston and Madahar, 2002).

Unfortunately, there are significant differences between hardware and software that pose significant challenges to describing a hardware implementation using a software language. Five areas of difference identified by Edwards (2005; 2006) are concurrency, having a model of time, data types, memory model and communication patterns.

Hardware is inherently concurrent, with each step or computation performed by separate hardware. In contrast, software performs its computation by reusing a central processor, making software inherently serial. As indicated in Chapter 1, software processors introduce parallelism at a range of levels: at the instruction level through wide instruction words (effectively issuing several instructions in parallel), temporally, through instruction pipelining (with successive instructions overlapping by being in different phases of the fetch/decode/execute cycle), through to coarse grained parallelism introduced by splitting the application into several relatively independent processes and using a separate thread for each. The fine-grained parallelism at the instruction level is closer to that of hardware, but is controlled by the particular architecture of the processor. The compiler just needs to map the high level statements to the given processing architecture. At this level the algorithm is primarily sequential, although compilers can (and do) exploit dependencies between instructions to maximise parallelism. At the thread or process level, the coarse-grained model does not match the fine-grained concurrency of hardware, but maps closer to having separate blocks of hardware for each thread or process.

Time and timing is absent from most software programming models. The sequential processing model of software ensures causality within an instruction sequence (Edwards, 2006), although this becomes more complex and less deterministic with multithreaded execution and task switching. However, the time taken for each task is not considered by the language. In contrast, hardware, and especially synchronous hardware, is often governed by strict timing constraints. Many hardware interactions must be controlled at the clock level and representing these with a high level software language can be difficult.

The data word width of a hardware system is usually optimised to the specific task being performed. In contrast, software languages have fixed word widths (8, 16, 32, 64 bits) related more to the underlying architecture than the computation being performed. Two approaches to this are to modify the language to allow the word width to be explicitly specified, or to make use of the object orientated features (Java or C++) to define new types and associated operators. Virtually all software languages support floating-point computation, and floating-point numbers have been the mainstay of scientific computing, including image and signal processing, for some time. However, until recently floating-point number representations have been avoided on FPGAs because of the cost of implementing the more complex hardware. It is only with higher capacity FPGAs and higher level languages that floating-point calculations have become practical in hardware for other than very specialised tasks.

The memory model of software is very different to that used within hardware systems. Software treats memory as a single large block. The compiler may map some local variables to registers, but the software model has everything stored in memory. This allows memory to be allocated to a process dynamically, with the associated memory block referenced by its address. C even allows efficient memory traversal algorithms through pointer arithmetic. In hardware, many local variables are not implemented in memory, but directly as registers. Memory is often fragmented into a large number of small, independent blocks. In such an environment, pointers have very little meaning, and dynamic allocation of memory resources is difficult, if not impossible. This has a direct impact on the algorithms that can readily be used or compiled to hardware.

The differences in memory structure also directly impact communication mechanisms between parallel processes. Software generally uses shared memory (including mailboxes) to communicate between processes. Such a model is sensible where there is only a single CPU (or small number of CPUs), and the processes cannot communicate directly. In hardware systems, parallel processes are truly concurrent and can therefore communicate directly. In data-synchronous systems, such as streamed pipelining, communication is often implicit, or through simple token passing. More complex systems can use FIFO buffers or other synchronising and communication mechanisms. In many hardware systems, the communication relies on dedicated hardware. This is particularly the case when communicating between clock domains.

Another major difference between hardware and software relates to the representation and execution of an algorithm. In software, an algorithm is represented as a sequence of instructions, stored in memory. A program counter points to the current instruction in memory, and a memory-based stack can be used to maintain state information, including any local variables. This allows algorithms to be implemented using recursion where within a procedure the state can be saved and the procedure called again. In hardware, the algorithm is represented by physical hardware connections. Saving the current algorithm state is difficult, making recursive algorithms impractical in hardware. Any such recursion within an algorithm must be remapped to iteration to allow a practical implementation.

In spite of these significant differences, three approaches have been used to develop synthesis tools from high level languages. One is to use the host language to create and manipulate a structural representation of the circuit constructed. This is effectively using the high level language to simply model the hardware, rather than directly specifying it. The second is to modify or extend the syntax of a high level language to provide constructs that enable it to be used to describe hardware (Todman *et al.*, 2005; Edwards, 2006). The third approach is to use a standard high level language and requires the compiler to automatically extract parallelism from the design and appropriately map it to hardware. These three approaches are described in more detail with some examples of each of these approaches. Much of the literature reports experimental work, although there are now several commercial offerings.

3.2.1 Structural Approaches

Structural approaches work at describing the design using a structural model, rather than algorithmically. These are primarily tools that mimic the structural designs of conventional HDLs but within a standard programming language making the language and design more accessible to software engineers. They are able to use the programming constructs of the host language to automate some design tasks, or to facilitate parameterisation more naturally than standard HDLs. The other advantage of using a high level language is that it more readily enables targeting the result to either software or hardware, facilitating the partitioning of the design.

3.2.1.1 JHDL

JHDL (Bellows and Hutchings, 1998; Hutchings *et al.*, 1999) is a Java-based hardware description language developed by the Configurable Computing Laboratory at the Brigham Young University. It began as a tool to experiment with run-time reconfigurability (Bellows and Hutchings, 2001) and was extended to synthesise the designs, targeting Xilinx FPGAs. It is based on a structural design, which is described programmatically using Java, with sets of classes representing objects, and wires. The integrated environment then allows the circuits to be simulated to verify and debug operation. Tools within the environment allow the manipulation of the design, including the placing of components on the FPGA (floor planning). The design is then compiled to an EDIF net-list which the FPGA vendor's tools then place and route to produce the bitstream.

The main limitation of the original tool was that it used purely a structural design model. While this enabled smaller and faster designs to be produced more quickly than conventional tools, design of complex control circuits and finite state machines was awkward. This led to the incorporation of behavioural synthesis (Wirthlin *et al.*, 2001). Here the behavioural code is written in two key Java methods of the class associated with the module, which are then compiled by the Java tools. The Java byte codes are analysed to extract both the data flow and control flow from which the circuit is synthesised.

Although the documentation and papers talk about a technology mapping phase (where the platform independent representation is mapped to the resources of a specific FPGA technology), the environment currently supports only Xilinx FPGAs up to the Virtex II.

3.2.1.2 Quartz

Quartz (Pell and Luk, 2005) is a structural hardware description language based on block composition. This allows concise descriptions which can be manipulated by formal reasoning to prove the correctness of the implementation. One of the focuses of this work was on parameterised pipelines for exploring power consumption. The motivation for the work was to create efficient parameterised libraries, so the representation was compiled to structural VHDL.

3.2.1.3 HIDE

Crookes *et al.* (1998; 2000) extended their earlier work on implementing image algebra-based systems on transputers (Crookes *et al.*, 1989) and other parallel processing systems to target FPGAs. The approach taken was to have a library of primitive operations, with corresponding hardware implementation, and use a Prolog-based system of rules to construct the final system automatically from the image algebra representation. The work was later extended to provide a range of low level implementations (bit-serial, parallel, signed-digit) to enable low level architectural exploration without having to change the high level program representation (Benkrid *et al.*, 2000).

A later incarnation of the language, HIDE4K (Benkrid, 2000; Benkrid *et al.*, 2002b) extended this rule-based approach to hardware skeletons, effectively a set of high level library constructs for building more complex operations. The image processing problem is represented as a directed acyclic graph of basic image processing operations, which is then matched to skeletons to get the framework for the implementation. The components within the skeleton are then instantiated from a library of optimised basic building blocks (line buffers, adders, multipliers, absolute value units, and so on) depending on the particular image processing operations being used. Although the HIDE language can produce platform independent VHDL with the vender tools implementing the design, it was found that higher clock speeds and better packing density could be achieved with libraries that target a particular FPGA (Benkrid *et al.*, 2006). The platform specific libraries perform an initial placement, producing an EDIF net-list which is mapped to the FPGA.

The use of a skeleton library allows the user to specify a high level description of the operations, which are then internally translated to a structural representation, with the detailed implementation complexity hidden from the user (Benkrid and Benkrid, 2008). While more flexible than a pure library-based approach, it still requires that the design be able to be mapped to the available skeletons.

3.2.2 Augmented Languages

The second approach is to modify the syntax of the high level language to effectively convert it into a hardware description language (Todman *et al.*, 2005). Particular extensions are the ability to specify concurrency and associated communication mechanisms, and to include hardware specific constructs such as RAMs, ROMs and variable data bit widths. With this approach the responsibility of mapping the algorithm onto hardware still rests primarily with the designer. Fully exploiting parallelism often requires significant changes to the algorithm to handle hardware constraints. This, of course, requires the developer to be familiar with hardware design. However, a wider range of parallelisation techniques can be exploited, enabling the design to be optimised for real-time operation. The advantage of using a high level language rather than a more conventional HDL is that much of the low level bookkeeping is abstracted from the developer. However, since this still allows some control at the lower levels, it really provides an intermediate level of abstraction and control. While some of the changes to the language may appear relatively superficial, the introduction of parallel constructs significantly modifies the language and the way that it is programmed. This requires the programmer to modify their thinking – it is still important to think in terms of hardware.

3.2.2.1 SystemC

SystemC (IEEE, 2006b) is a modelling language based on C++. System languages can be used for modelling a complete system, including both hardware and software components. SystemC decomposes the system into a set of modules, which are connected using ports. In this way it represents the structure of the system. A class library is defined which provides the components used to build up the system. Derived classes can then be used to implement the system directly using C++. The expressive functionality is similar to VHDL or Verilog, with each module containing processes that run concurrently, with the code inside each process executing sequentially. Like processes in VHDL, SystemC processes are triggered through events (changes on particular signals), perform some action, and then suspend execution. In this way, the language can also operate at the behavioural level. Although the system is structurally based, the behavioural description is written using standard C++.

One of the features of SystemC is the separation of the computation from the communication. This allows a move to a higher level of communication between processes through transaction level modelling (Swan, 2006). At the register transfer level, a communication takes place using a set of signals between the communicating modules, and a series of signal assignments and read operations on the wires of the bus. A transaction level model would represent the complete transaction with a single function call containing the request and response. Modelling at a higher level both simplifies the design and increases the speed of simulation by up to two orders of magnitude.

SystemC contains hardware orientated data types and directly supports fixed-point arithmetic. This raises the level of abstraction from conventional HDLs, because appropriate rounding and overflow operations can much more easily and transparently be specified. Although designed primarily as a modelling language, a subset of SystemC (including at the behavioural level) can be synthesised for FPGA implementation. Although early synthesis work was experimental (Economakos *et al.*, 2001), most electronic design automation (EDA) vendors now provide tools for synthesis from SystemC.

3.2.2.2 ASC

ASC (A Stream Compiler) (Mencer *et al.*, 2003; Mencer, 2006) also uses C++ classes to represent hardware data types and provides a layered approach to the design at the algorithm, architecture, arithmetic and gate level. The focus is primarily for the development and design space exploration of hardware accelerators to augment a standard serial processor.

The approach taken is to convert iteration through an array into a hardware stream process. At the algorithm analysis level, the developer is responsible for performing the associated loop transformations and selecting the appropriate architecture and data structures. The architecture generation is defined by the resulting ASC code. A set of C++ classes define the variable types (integer, fixed-point and floating-point), the data width, with the specific hardware number representation and storage type (register, stream, on-chip memory or off-chip memory) specified through an attribute. These attributes allow the design space to be explored simply by changing the appropriate attribute at the higher level, rather than having to specifically redesign everything at the low level. A library approach is used for the generation of arithmetic modules, with the specific format and storage defined by an extensive library. Compilation of the code produces either an executable that simulates the design at either the word level or bit level, or produces the circuit in the form of an EDIF net-list. Each module can be constrained on the basis of area, latency or throughput, enabling exploration of the design space from within the one framework.

Processing as streams simplifies the interface between the host processor and the accelerator. Data from the host is streamed into input FIFOs at the head of the stream and streamed from output FIFOs at the output.

3.2.2.3 SA-C

Single assignment C (Draper *et al.*, 2000; Bohm *et al.*, 2001; 2002; Draper *et al.*, 2002; Najjar *et al.*, 2003) is a high level variant of C specifically designed for image processing applications. It was originally developed as an accelerator for within the Khoros image processing framework (Hammes *et al.*, 1999) and fits well with the low level image processing operations for which it is targeted.

There are three main differences between regular C and SA-C (Draper *et al.*, 2000). Firstly, hardware data types are introduced. In particular, variable width integer and fixed-point numbers enable the data width to be tailored to the application. Standard floating-point and Boolean types are also supported. Secondly, arrays are natively supported, with associated control constructs for simply looping through multidimensional arrays and for scanning windows through arrays. Such constructs not only provide a higher level representation but also makes analysis of array accessing easier for the compiler. Looping through an array also maps naturally to a streamed style of architecture. Thirdly, aspects of C that do not fit well with FPGA hardware are eliminated. Pointers are not allowed (they are less necessary with the powerful new array iterators) and recursion is prohibited. The language also enforces single assignment semantics (each variable may only be written to once), which facilitates analysis of the dataflow, and enables a one-to-one correspondence between each variable and wires on the FPGA (registers are not required to hold a variable, except for pipelining to improve throughput). All of these features enable the dataflow to be analysed and optimised by the compiler to result in an efficient mapping to the FPGA architecture.

The computational architecture is split into three levels. The first is the data generator, which is responsible for streaming the data in from memory as required by the loop. This includes buffering data from one window position to the next. SA-C does not, however, infer or build row buffers, but reduces the number of reads by processing multiple rows across the image simultaneously (Draper *et al.*, 2001). The second level is the loop body, which performs the processing on the data as the image is scanned. The single assignment restriction enables producer and consumer relationships to be identified at the image level, enabling two or more loops to be combined, minimising the number of times that the data needs to be transferred between the host (or memory) and the FPGA. This optimisation is effectively inferring a streamed pipeline from the dataflow. The third level is the data collector, which stores the results back to external memory.

The compactness of a SA-C representation is illustrated in Listing 3.2 for the implementation of a Prewitt edge detection filter (Bohm *et al.*, 2002). Line 1 declares a procedure that takes as input a two dimensional image of 8-bit unsigned integers, and produces a 16-bit unsigned image result. Lines 2 and 3 declare and initialise the Prewitt masks. An image is declared on line 4, which is assigned the result of scanning a 3×3 window through the input image (lines–10). The body of the loop (lines 6–8) is executed for each window position. An inner loop (lines 7 and 8) iterates in parallel (the `dot` operator) through the two masks and the pixels within the window. The sum functions on line 8 reduce the pixel by pixel products to two numbers, one for each of the masks. These are assigned to the 16-bit integer variables `dfdx` and `dfdy`. Line 9 combines these values together to create the `magnitude`, which is then used to build up the array `M`.

Listing 3.2 SA-C code for a Prewitt edge detector (With kind permission from Springer Science + Business Media: *Journal of Supercomputing*, "Mapping a single assignment programming language to reconfigurable systems," **21**, © 2002, 119, W. Bohm, J. Hammes, B. Draper *et al.*, Figure 2.)

```
 1  int16 [:,:] main (uint8 Image[:,:]) {
 2     int16 H[3,3] = {{-1,-1,-1}, { 0, 0, 0}, { 1, 1, 1}};
 3     int16 V[3,3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
 4     int16 M[:,:] =
 5         for window W[3,3] in Image {
 6            int16 dfdy, int16 dfdx =
 7               for h in H dot w in W dot v in V
 8                  return(sum(h*w), sum(v*w));
 9            int16 magnitude = sqrt(dfdy*dfdy + dfdx*dfdx);
10        } return(array (magnitude));
11  } return(M);
```

The language can be compiled for serial processing, for simulation and debugging on a conventional processor, or via VHDL to target hardware. The mapping to hardware is facilitated by a library of constructs for implementing each of the components.

The lack of a while statement makes other types of operations more difficult. However, the array access mechanisms are efficient for implementing operations that require a scan through the image and provide a higher level representation for those tasks than regular C.

3.2.2.4 A|RT

An early commercial C to HDL compiler was A|RT (for Algorithm to Register Transfer), marketed by Frontier Design. It extends C with fixed-point data types (Johnson and Defossez, 1999). A|RT creates a processor like architecture and uses a VLIW type processor to direct and schedule the operations (Johnson, 2000). The tools allow interactive exploration of different processor architectures during the design. One of the features is the ability to import predefined components (intellectual property cores) into the C programme as function calls. These map to a structural representation in VHDL, allowing optimised code to be directly incorporated into the design. However, as an early generation language, it has not survived (Martin and Smith, 2009).

3.2.2.5 Mitrion-C

Mitrion-C (Mohl, 2006) is a parallel C dialect marketed by Mitrionics specifically targeted towards high performance computing (including image processing). Like SA-C, it is based on the single assignment semantic to simplify extraction of parallelism from the code. Every statement returns a value and performs an assignment, including loops and conditionals. The only effect that a loop or block has on the rest of the programme is through the return value. This enables loops to be executed independently of other parts of the programme. Within a block, all assignments are parallel. The language supports a wide range of types, including variable sized integers and floating-point numbers. Collections of types include lists, vectors and streams. Memory accesses are synchronised through the use of instance tokens, which guarantee the sequential order of accesses.

The basic concept of Mitrion-C is that everything is parallel. Using this concept, a Mitrion virtual processor is created and the language is compiled to that. For an FPGA, the algorithm is accelerated to the maximum extent by unrolling loops (effectively creating multiple parallel processors) until the FPGA is filled. The latest version of their compiler will take the same source code and target an FPGA, GPU or multithreaded CPU exploiting the parallel capabilities of the target platform.

3.2.2.6 Handel-C

One of the difficulties with multiple parallel processes is ensuring reliable communication between them. Simply using shared registers or shared memory only works if the processes are synchronised. One approach is to use the *communicating sequential processes* (CSP) model described by Hoare (1985). This model requires all communication channels between processes to be unidirectional, point-to-point and blocking. That is the processes at both the sending and receiving end of the channel will wait until the communication successfully takes place before continuing. Using CSP primitives, it is possible to build more complex communication methods, including buffered communication.

The Occam programming language was the first to implement CSP for communication between parallel processes, and as a parallel language this was readily adapted to programming FPGAs (Page and Luk, 1991). Converting the Occam language (and the CSP model) to use a C-like syntax led to Handel-C (Page, 1996).

The Handel-C language was commercialised by Celoxica through the DK Design Suite (an integrated compiler and clock accurate simulator) and quickly established a strong following in the academic community. In 2006, the Handel-C development tools were passed to Agility Design Solutions. In 2009 Agility ceased trading and Mentor Graphics acquired Handel-C and the DK Design Suite.

Handel-C (Mentor, 2010b) extends a subset of C with variable word length integer variables, bit manipulation operations, hardware architectural components (ports, interfaces, RAMs, ROMs), constructs for specifying concurrency through parallel blocks and channel communication between parallel processes. While pointers are allowed, pointer arithmetic is not supported by the language.

The language also provides powerful recursive macros for constructing repetitive complex circuits (for example variable width multiplication and division, or CORDIC calculations). A distinctive feature of Handel-C is that every assignment takes exactly one clock cycle. The resulting strongly timed synchronous design is therefore implicitly at the register transfer level, although using a higher level algorithmic syntax. The compiler internally builds a state machine using a token passing scheme to control the execution and sequencing of commands. The compiler therefore reduces the burden of designing the control logic, as it is implicit in the language constructs. In other respects, though, it is still the responsibility of the developer to identify and exploit parallelism and create pipelines where appropriate. This makes Handel-C an intermediate-level rather than a high-level language. However, even this is sufficient to give significant design productivity over programming in VHDL.

Listing 3.3 A Handel-C code fragment to implement the filter shown in Figure 3.5

```
 1  #include <stdlib.hch>
 2
 3  macro expr t = 10;
 4  macro expr k = 128;      //fixed point number/256
 5
 6  macro expr absv(si) =
 7     (unsigned)(((si < 0) ? -si: si) <- (width(si) - 1));
 8
 9  macro proc filter(data_in, data_out) {
10     macro expr dw = width(data_in);
11     unsigned int (dw) i, q;
12     signal <int (dw+1)> diff;
13     do par {
14        data_in ? i;             //Stream input through data_in
15        diff = (signed)((0@i) - (0@q));
16        if (absv(diff) < t)
17           q = i;
18        else
19           q += (unsigned)(adjs(diff, dw*2+1) * k)\\(dw+1);
20        data_out ! q;           //Stream output through data_out
21     } while(1);
22  }
```

A Handel-C implementation of the circuit in Figure 3.5 is given in Listing 3.3. Lines 3 and 4 define constants for the threshold level and filter gain. The absolute value block is implemented by the macro expression on lines 6 and 7. Since Handel-C is strongly typed, it is necessary to cast the result to unsigned. `width(si)` is a compile time constant that gives the bit width of `si`; the expression here is taking the least significant bits, reducing the width by one. The actual filter is represented by the macro procedure on lines 9–22. The arguments of a macro procedure are passed by reference, so any operations on the parameter within the procedure will be directly accessing the caller's arguments. The parameters of macro procedures are untyped, deriving their type from the arguments. This allows macro procedures to be more easily parameterised, whereas the parameters of standard functions must be explicitly typed. Line 10 illustrates one mechanism for such parameterisation by defining `dw` to be the width of the argument.

This is used in the following lines to declare the variables `i` and `q`, and the signal `diff`. Note that signals in Handel-C have a different meaning to those in VHDL. Here a `signal` corresponds to an unregistered wire. Lines 13–21 contain an infinite loop. The `par` specifies that all of the statements within the loop are executed in parallel. The data is streamed into the filter through the channel read from `data_in` on line 14, with the result streamed out through the channel `data_out` on line 20. Since these are all executing in parallel, if input data is not available or the downstream process is not ready, then the corresponding channel operation will block, stalling the process until the input data is available and the output transferred. In this way, the filter is automatically self-synchronising with the upstream and downstream processes using the CSP model. Lines 15–19 implement the actual filter. Line 15 takes the difference between the input and output. To prevent losing the sign of the result, the widths of the variables are increased by concatenating (`@`) a 0 on the front. Since `diff` is a signal, the result is not registered, so this does not take any clock cycles. A signal is used here to reuse the result in multiple places (in lines 16 and 19). Line 19 is effectively executing `q += diff * k;` however the output of the multiplication in Handel-C is the same size as the inputs, so `diff` needs to be sign extended (`adjs` is defined in `stdlib.hch`) to the desired width. Since `k` is a constant, it will automatically be made the right width. The result is cast to unsigned and the least significant bits dropped (`\\`) to make it the right width and type for adding to `q`. While the strong typing is good for detecting errors, the need for casting when the variable widths change can reduce the readability of the code.

The Handel-C source can either be compiled to VHDL, or directly to an EDIF net-list. Vendor tools are then used to place and route the design for the target FPGA. These can be integrated within the DK Design Suite as custom build commands, producing the configuration file as the output.

3.2.2.7 Handel-C Derivatives

One of the limitations of Handel-C is that once a program has been developed with the strict timing model, it can be very time consuming to rearrange the algorithm. This makes exploration of the design space more difficult. Haydn-C (Coutinho and Luk, 2003) overcomes this problem by having two distinct formats. One is a strict timing model and structure that closely follows Handel-C, and indeed is compiled to hardware by using Handel-C as an intermediate language. The other format is an untimed dataflow graph that represents the algorithm itself. From this form, the algorithm may be transformed into a range of different structures, enabling exploration of pipelining, loop unrolling, resource allocation and scheduling. A single design can then be used to create a range of implementations guided by a set of annotations in the code. This enables a very flexible exploration of the design space without sacrificing the strict timing model.

Bach-C (Yamada *et al.*, 1999; Kambe *et al.*, 2001) is very similar to Handel-C, and with similar design goals. The main difference is that Bach-C does not have timed semantics. This allows for greater optimisation within each thread. Synchronisation is performed using either synchronous channels (like Handel-C) or asynchronous channels (effectively shared variables or memory).

3.2.3 Native Compilation Techniques

An alternative approach to those described in the previous section is to use standard software source code, and require the compiler to automatically extract parallelism from the design and map the design to hardware. The compilation process may be guided, either with annotations within the source or constraint files that control and optimise the mapping process (Todman *et al.*, 2005). Common optimisations include loop unrolling and automatic pipelining based on an analysis of the algorithm dataflow. By hiding the hardware constraints from the designer, the lower level hardware design is abstracted away, enabling the developer to focus primarily on the algorithm development. Note that the significant differences between the way in which software and hardware are implemented may place some limitations on how the algorithm is coded or restrictions on the language to enable the compiler to identify constructs that may be parallelised. However, most of the work in making the design suitable for FPGA implementation is moved from the designer to the compiler. One of the limitations of this approach is that the compiler is effectively optimising a sequential algorithm, and only the obvious forms of parallelism may be extracted and exploited. The underlying algorithm is still sequential. However, for low level image processing, this approach has been shown to be effective (for example, Streams-C; Gokhale *et al.*, 2000).

3.2.3.1 Transmogrifier C

One of the early efforts at a C to FPGA compiler was Transmogrifier C (Galloway, 1995). It supported a very limited subset of C (no multiplication, division, pointers, arrays or structures). One extension to regular C was the use of pragmas to specify the number of bits used by integers. It provided a simple behavioural method of describing a hardware circuit. The timing semantic was also very simple – there was one clock cycle for each function call and one for each loop iteration. Transmogrifier C was more recently reincarnated as FPGA C on Sourceforge.

3.2.3.2 SUIF-Based Compilers

The SUIF (Stanford University Intermediate Format) compiler system (Wilson *et al.*, 1994; Hall *et al.*, 1996) is a framework designed within Stanford University for collaborative research on parallelising compilers. The framework has been used as the basis for experimental work on compiling from C to HDLs.

The NAPA C compiler (Gokhale and Stone, 1998) works on standard C code and targets the NAPA platform, which consists of a RISC processor coupled with an adaptive logic array. Sections of code can be marked with pragmas to target execution on either the RISC processor or the reconfigurable logic. The compiler pipelines loops targeted for hardware to accelerate their execution.

Streams-C (Gokhale *et al.*, 2000) is an extension of NAPA C to efficiently handle pipelined stream computing. As such, it is able to exploit the parallelism associated with processing streams, with pipeline optimisations performed automatically by the compiler. Directives are added as comment-based annotations that declare processes, streams and signals, and to assign resources to these on the FPGA. A process is a computational block that operates on streamed data. The compiler builds a state machine to control the flow of data through the process pipeline. A library of functions is provided for basic stream manipulations (opening, reading, writing and closing). These link successive processes, including those running on a host serial processor, with inter-process communication based on the CSP model. The same source can be compiled for sequential execution (using threads for each process) or to synthesizable VHDL for FPGA implementation.

One fruitful area of parallelisation is loop unrolling and pipelining. SPC (Weinhardt and Luk, 2001b) takes standard C input and performs loop transformation compiling to a net-list, with the vendor tools used for place and route. ROCCC (Buyukkurt *et al.*, 2006) similarly takes standard C and compiles to VHDL. ROCCC is designed to manage and pipeline two-dimensional arrays and the access patterns required for image processing filters. Smart buffers are used to cache data required in subsequent rows to increase data reuse and reduce the number of memory accesses.

3.2.3.3 Impulse C

Impulse C (Pellerin and Thibault, 2005) is a commercial offering, from Impulse Accelerated Technologies, that follows along the lines of Streams-C. Following the CSP model, the application is divided by the programmer into a set of processes which communicate with one another through streams. This allows the processors with high computation density to be implemented on an FPGA, with the remainder of the application implemented on a serial processor (Pellerin *et al.*, 2005). The development tools (called

CoDeveloper) compile modules either to software or to hardware, using a library of streaming functions for software–hardware and hardware–hardware communication. The compiler automatically extracts parallelism, pipelines the design and builds the controlling state machine for each hardware process. However, the developer must still structure the code in a way that maximises the use of hardware resources.

3.2.3.4 Dime-C and DimeTalk

Dime-C (Genest *et al.*, 2007) is a C to VHDL compiler provided by Nallatech. It aims to be ANSI compliant, but does not allow pointers, structures, and some types of loop or switch statements that can be expressed in other ways. The code can be compiled using standard software compilers for testing and debugging the algorithm. When compiled for hardware, pragmas are used to annotate the code to indicate to the compiler sections that contain significant parallelism. The compiler automatically parallelises and pipelines the code when possible.

The compiler can be run on its own or as part of DimeTalk IDE. DimeTalk (Sanderson, 2004) enables the communication between processes both within and between FPGAs to be designed at the conceptual level and it generates the required synthesisable code. Communication is based on a lightweight network architecture for connecting the various modules, with data transferred using a custom, low overhead packet-based protocol.

3.2.3.5 Catapault C

Catapault C (Bollaert, 2008) is a high level synthesis tool provided by Mentor Graphics. It uses standard, untimed C++ code allowing the design to be directly specified at the algorithmic level. It has no specific concurrency constructs – any concurrency is automatically extracted at the compilation and optimisation stage. The only restriction on the language is that everything must be determinable at compile time. This means dynamic memory allocation and recursion cannot be used. The language is extended with a class library that supports variable bit-width integers and fixed-point real numbers, as needed for an efficient hardware design. As standard C++ code, the algorithm can be tested and validated within the standard software environment before synthesising to hardware. Of all the languages reviewed here, Catapault C is the most compliant with C (or C++) in terms of what it is able to synthesise.

The compilation to hardware is controlled by the user by specifying timing and resource constraints. These, combined with characterisations of the target technology, are used to optimise the mapping to hardware through loop unrolling, loop merging, and pipelining. The tools provide several views of the algorithm and its performance, allowing the user to interactively adjust the constraints to efficiently explore the design space without having to rewrite the source code. During synthesis, the communication mechanisms between the different modules are automatically chosen based on the access patterns (stream FIFO, bank-switched memories, round robin memories and other interleaving constructs). The control logic required to schedule and sequence all of the modules is also built automatically. Synthesis then converts the C++ code to RTL in VHDL, Verilog or SystemC, which can then be used to directly generate hardware either for FPGA or ASIC implementation.

3.2.3.6 MATLAB® Based

Many image and signal processing algorithms are first developed and tested in MATLAB®. Therefore, considerable development time would be saved if the MATLAB® code could be synthesised directly to an FPGA. The MATCH project aimed to do exactly this (Banerjee *et al.*, 2000).

Firstly, the MATLAB® code is analysed to determine the operations performed and the data types. This is complicated by the fact that MATLAB® is dynamically typed, with the type changing depending on the results of any assignments. For many operations there are optimised cores (Banerjee *et al.*, 2000), otherwise the code is mapped to sequential code on the FPGA. Matrix operations are expanded out into loops, with complex expressions broken down to simpler expressions. The loops are then pipelined to accelerate the computation. Since most processing in MATLAB® is on large arrays, the input and output arrays are assumed to be in external memory. Speed, therefore, is dominated by memory references (Haldar *et al.*, 2001a; 2001b). The floating-point calculations used by MATLAB® are converted to fixed-point for computation on the FPGA (Nayak *et al.*, 2001; Banerjee *et al.*, 2003). This significantly reduces the hardware and, if the bit width can also be reduced, multiple array elements may be packed into a single memory location, improving the bandwidth. The algorithm is compiled to the target FPGA via VHDL.

Since MATCH was developed, its technology was transferred to a start-up company, AccelChip. In 2006, Xilinx acquired AccelChip and the technology is now available as AccelDSP (Xilinx, 2009a). The design flow still follows the principles outlined above. The MATLAB® code has to follow a particular coding style that assumes streamed input and output data. The floating-point design is converted to fixed-point and verified for fidelity with the original floating-point model. The fixed-point model is then converted to RTL and a test bench created for validation. Acceleration optimisations performed by AccelDSP include partial or full loop unrolling, mapping of variables (arrays) to memory and pipelining. The final design can also be compiled to a block for use with Xilinx System Generator (described in the next section).

Another MATLAB®-based tool is Compaan/Laura (Harriss *et al.*, 2002; Stefanov *et al.*, 2004). This toolset converts a sequential MATLAB® design into a network of independent processes, which is more dataflow orientated. The revised processing model is easier to map to hardware. The data flows from one process to another using blocking FIFOs to ensure correct computation order. Within each process, optimisations such as loop unrolling and skewing are used to improve the implementation.

3.3 Visual Languages

An alternative to representing an algorithm textually is to use a graphical programming environment and its associated visual language. A visual programming language is one which uses a visual representation to augment what is normally represented purely with text. Textual programming languages are a one-dimensional sequence of commands or statements, which matches the needs of a serial processing environment with a linear program memory architecture. However, programming FPGAs efficiently requires parallel and concurrent programming. A visual language offers multidimensionality (Chang *et al.*, 1999), enabling both spatial and temporal semantic relationships to be expressed more clearly.

At the low level, a schematic diagram is a form of visual language. It enables a network of components to be created with connections between the components indicated by wires. In a textual language, each wire becomes a net name, with the circuit represented by a net-list. Connections are made by listing the net names associated with each port of the component (for example structural VHDL).

Many of the control structures of complex logic systems are based on finite state machines. These can readily be represented graphically using state transition diagrams, such as those shown in Figure 3.4. Therefore, a natural tool for creating and editing finite state machines could be based around the direct editing of the associated state transition diagrams. One example of such a tool is StateCAD, which is part of the Xilinx toolset. While such tools are good for creating finite state machines, this only forms part of the control structure for the complete design.

A range of higher level techniques are reviewed in the following sections.

3.3.1 Behavioural

3.3.1.1 vVHDL

Visual VHDL (vVHDL) (Miller-Karlow and Golin, 1992; Golin *et al.*, 1993) provides a visual representation of VHDL, with different shaped icons and boxes representing concurrent and sequential processes. Otherwise, it provides much the same features as VHDL. Advantages of the graphical representation are that it overcomes the syntax problems associated with VHDL and provides better insight into the function of a design by distinguishing between sequential and concurrent processes (Miller-Karlow and Golin, 1992).

3.3.2 Dataflow

At a higher level of abstraction, both parallelism and pipelining are well represented using a dataflow graph. The parallel nature of FPGAs results in a better fit with dataflow languages, both visual and text based. Indeed, most image and signal processing algorithms are often represented graphically by block diagrams with blocks representing operations on the data, and connections between the blocks indicating the flow of data. Dataflow graphs expose the parallelism of an algorithm, while imposing minimal constraints on the execution order (Buck and Lee, 1993). This allows an appropriate set of hardware processors to be defined for the dataflow. All of the compilers of text-based serial languages (described in the previous section) that automatically extract and exploit the parallelism use some form of dataflow analysis of the algorithm to identify dependencies and parallelism.

This has led to the obvious approach of directly representing image processing algorithms graphically using some form of dataflow language. This can be at two levels: building the algorithm from image processing operations, and at the lower level of dataflow within the individual operations.

3.3.2.1 Khoros Based

Khoros is a software-based image processing environment (Williams and Rasure, 1990), of which Cantata is the dataflow-based graphical programming tool. Within Cantata, each image processing operation is represented by a block, with the blocks wired together according to the dataflow. There have been several research projects using Khoros diagrams as a front-end for targeting FPGA implementations (Hoisko *et al.*, 1996; Levine *et al.*, 1999; Ong *et al.*, 2001).

The basic approach is to have a library of HDL primitives for each Khoros function (Levine *et al.*, 1999). The application can then be compiled by piecing together these primitives based on the dataflow. Since the latency of each primitive is known, synchronising delays can automatically be inserted in parallel paths. Adding a new operation requires providing the underlying code in both C/C++ for the software environment and VHDL for the hardware implementation. These must be functionally equivalent to ensure that the hardware produced will gave the same results as the software algorithm (Ong *et al.*, 2001). This is achieved by using the same fixed-point C/C++ code and compiling it to VHDL to create the library primitives using the A|RT compiler.

3.3.2.2 Simulink Based

Simulink is a graphical dataflow-based programming environment within MATLAB$^{®}$. It is widely used for DSP-based design and is also applicable for stream-based image processing designs. A number of systems leverage off the environment by providing Simulink block sets for targeting FPGA-based designs. Examples are System Generator from Xilinx (Hwang *et al.*, 2001), DSP builder from Altera (2010c), Synplify DSP from Synplicity (Balakrishnan and Eddington, 2007), and Simulink's own HDL Coder directly from MathWorks (2010). A brief comparison of some of these tools is performed by (Zoss *et al.*, 2011)—there can be quite significant differences in both resources used and processing time between the different tools for the same design. Using the Simulink environment allows application developers without a detailed knowledge and experience of the hardware to produce efficient implementations relatively quickly and effectively. The modelling environment offered by Simulink (and MATLAB®) provides accurate simulation of the design that is considerably faster than that provided by conventional HDLs.

The design flow involves developing the algorithm within MATLAB® and then representing the algorithm graphically within Simulink. At this stage, the specific block sets are used for targeting the design to an FPGA. The design can then be thoroughly tested with bit-level accuracy within the Simulink environment, before being compiled directly to an FPGA. Many of the blocks are parameterisable, allowing the data width and other parameters to be tuned for the design. The block sets from Xilinx and Altera specifically target the advanced DSP features within their respective FPGAs, resulting in an efficient design and use of resources. For example, Altera provides a block set specifically targeted for video and image processing (Altera, 2010f).

The library-based approach hides many of the low level implementation details within the block sets. If all of the blocks required by an application are in the library, then this can be an efficient design approach. However, if a high level block is not available, it is necessary to use quite low level blocks to implement the design, which significantly increases the design complexity and reduces the transparency.

3.3.2.3 Others

LabVIEW is another dataflow-based programming environment. While targeted more for control applications, it can also be used for image processing. LabVIEW provides a set of blocks which have both a software and hardware implementation (National Instruments, 2005). The software implementation is used within the LabVIEW environment for simulating the design, while the hardware implementation is used when targeting an FPGA. When producing an FPGA implementation, the tools effectively use a library-based approach. Again, this has the limitation that only functions available within the library may be used for the design.

Another visual language is the PixelStreams graphical interface to Handel-C (Mentor, 2010c). It uses a library approach to provide a range of image processing operations in a stream based graphical environment. Each operation is written in Handel-C, so is readily extensible. The operations are connected using a consistent stream based interface which consists of a number of control flags and a range of data variables: pixel values, coordinates, and synchronisation pulses. The common interface makes the operations plug and play, with the resulting system compiled and implemented by the underlying Handel-C compiler.

3.3.3 Hybrid

While dataflow languages provide a good match for a parallel hardware implementation, unless the control can be embedded with the data, the control tends to be hidden. Directly expressing complex control within a dataflow language can be clumsy. Therefore, a hybrid approach can give improvement, where a dataflow representation is used for streaming data and a finite state machine can be used for designing the more complex control.

3.3.3.1 VERTIPH

The Visual Environment for Real Time Image Processing in Hardware (VERTIPH) (Johnston *et al.*, 2004; 2006a; Johnston, 2009) is the approach that Massey University is investigating to address this issue. The representation of a design is divided into three main views. At the top, the architecture view provides a high level dataflow representation of the image processing algorithm in a manner similar to that used by Khoros or PixelStreams. The computational view is a timed dataflow representation of the computations required to implement each operation; this is at a similar level to that of the inner blocks within a Simulink representation. Finally, the scheduling view uses augmented state transition diagrams for providing the complex control, for scheduling each of the computational blocks and for managing shared resources.

The architecture view comprises a hierarchical block and wire representation of the algorithm (Johnston, 2009). The top level gives an overall view of the complete algorithm. Since the top view must be constructed first, VERTIPH enforces a top-down design. If necessary, each architectural block can be decomposed into a networked collection of either other architectural blocks or computational blocks as illustrated in Figure 3.6. The hierarchical nature of architectural blocks enables resources to be encapsulated with their controlling processes (for example a bank-switched frame buffer). Although such encapsulation is not enforced by the language, it follows good design principles for hiding complexity from where it is not needed, and helps to improve the reusability and maintainability of components. The wires between blocks can carry both data and control signals. The connections also use a hierarchical data structure (similar to structs in C) to hide unnecessary details from the diagram. An example architectural view is shown in Figure 3.7. Note that the control flow (shown in grey) flows in a different direction to the dataflow in many cases. Image processing specialists who never develop their own operations can use the architectural view to assemble predefined library modules into their design.

The computational view specifies the arithmetic and logic operations required to implement each block within the design. To encourage functional decomposition, it is also hierarchical, allowing computational blocks to be used in place of operation blocks. Like the architectural view, the computational view follows a dataflow style, but also introduces control constructs (loops and conditional executions) using a graphical representation modified from Nassi–Shneiderman diagrams (Nassi and Shneiderman, 1973). The control structures enclose the operations that they control, making the structure more evident.

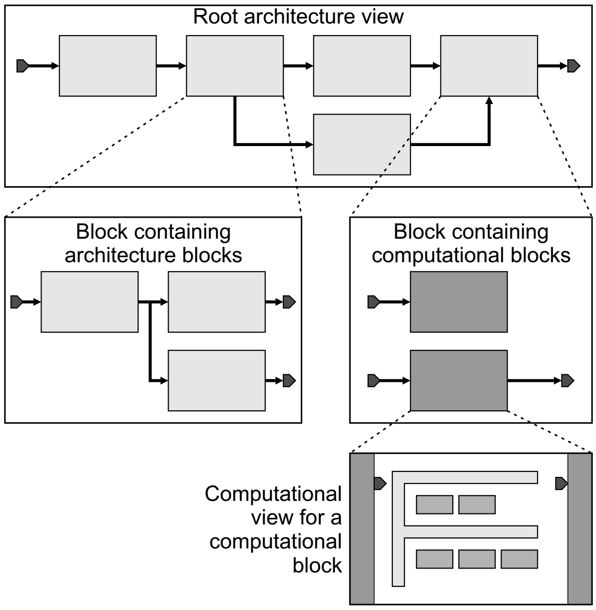

Root architecture view
Block containing architecture blocks
Block containing computational blocks
Computational view for a computational block

Figure 3.6 Hierarchical structure of a VERTIPH design. (Johnston, 2009; Reproduced by permission of C. Johnston.)

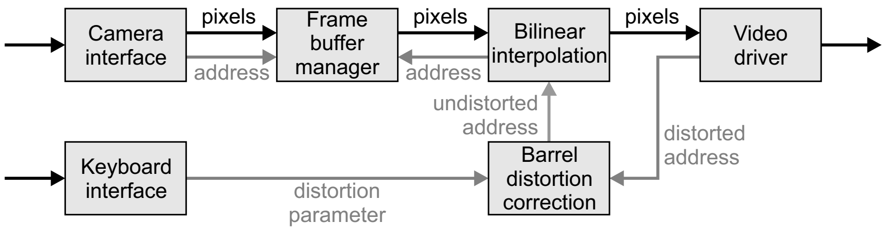


Figure 3.7 VERTIPH architectural view for lens distortion correction. (Johnston *et al.*, 2004; Reproduced by permission of C. Johnston.)

The syntax also introduces a strict timing semantic, with each operation assigned to a particular clock cycle, with the horizontal axis used to indicate time whereas the vertical axis indicates concurrency. The visual structure more clearly conveys the intention of the code, with a clear distinction between sequential, parallel and pipelined operations. The VERTIPH computational view for Figure 3.5 is illustrated in Figure 3.8.

VERTIPH is unique in its representation of pipelines. All other languages consider pipelines as a special case of concurrent operations, reflecting the fact that each stage of the pipeline operates concurrently. However, from a dataflow perspective a pipeline is really a sequential construct. The fact that other data is also being processed in the other stages is immaterial to the dataflow (Johnston *et al.*, 2010). Separating the sequential semantic aspects from the concurrent syntactic aspects significantly improves the transparency of the design. The representation used clearly shows the timing, throughput and latency of the design. Exploration of the design at this level is facilitated by giving the designer to direct control over the pipeline phasing. Having an explicit representation of the pipeline can also aid with pipeline control, enabling priming, flushing and stalling circuitry to be instantiated automatically in many cases (Johnston *et al.*, 2010).

The third view within VERTIPH is the scheduling view. The primary purpose of this view is to control when each of the computational blocks is executed, and to manage resource contentions. For simple systems, the resource and scheduling view is not necessary; all of the controls may be specified by using conditional structures directly within the individual computational views. However, as the system grows in complexity, this can become unwieldy, especially as one change may require consequent changes in other computational blocks. Centralising the high level control in a single view makes the process more manageable. The scheduling view has two parts (Johnston *et al.*, 2008). The first part consists of one or more state transition diagrams. These may be controlled by either external events or internal counters.

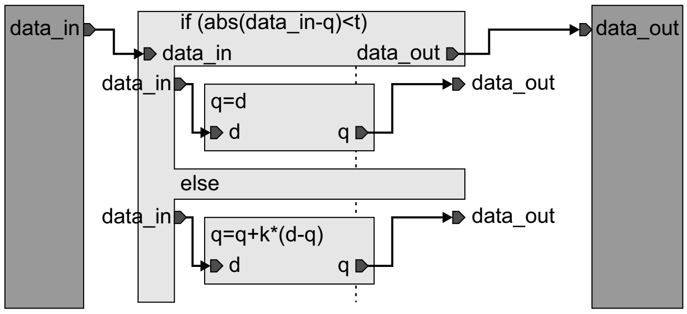


Figure 3.8 VERTIPH computational view for the filter shown in Figure 3.5.

In each state, one or more processes (computational blocks) may be scheduled to execute. The second part of the view indicates which processes are associated with each state. When multiple processes are associated with a state, these may be scheduled sequentially, concurrently, or as a pipeline. When multiple processes share a common resource, potential conflicts may arise. Many resource conflicts are simplified by encapsulating the resource and its conflict resolution processes within a single architectural block.

At the time of writing, the VERTIPH language does not include a simulator or compiler, but the language is described here to give a flavour of the approaches that can be taken for implementing image processing systems.

3.4 Summary

Four quite different approaches to programming an FPGA have been presented in this chapter. Hardware description languages (VHDL and Verilog) are good at describing hardware, and are inherently concurrent. Their main limitation is that they require programming at the structural level or register transfer level, which is quite different from the algorithmic approach to representing image processing algorithms. Consequently, the algorithm is usually developed within a software environment, leaving the laborious and error-prone task of translating it from the software into the HDL.

Conventional software languages, such as C and MATLAB®, are much better for representing algorithms, so if the translation step from software to HDL can be avoided, significant gains in productivity can be achieved. Two approaches are taken for using these languages to programme FPGAs. One is to extend the language to enable it to describe hardware, by adding concurrent constructs. The result is a software-based hardware description language – it retains the algorithmic structure, but is able to describe concurrent hardware. The conversion from software to hardware is then one of refinement, rather than rewriting the algorithm from scratch. The other approach is to leave the language unchanged, but to enhance the compiler to automatically identify and extract any parallelism from within the algorithm. The limitation of this approach is than many algorithms are represented serially, and are optimised for implementation on serial processors. While there may be parallelism that can be exploited, unless the underlying algorithm is essentially parallel it is not going to give the most efficient solution. The danger here is that the wrong algorithm is being parallelised and parallelising serial algorithms is not trivial.

The fourth approach is to use some form of visual language to represent the operation of the hardware. For many image processing algorithms, a dataflow representation exposes the natural parallelism inherent within the algorithm. One limitation of visual languages is that the representation is significantly more expansive (requires more screen real-estate), making it harder to maintain an overall view of the whole algorithm. In exposing the flow of data through the system, the control flow can also become obscured.

None of the current offerings is perfect (Martin and Smith, 2009). Which is best? The answer depends on the goals. A solution hand crafted in an HDL using a structural coding style will generally give the smallest and fastest design, although the development process is very time consuming. Using a C-based language gives higher productivity, but this often comes at the expense of a larger and slower design. The compiler orientated approaches will usually enable a faster exploration of different architectures, as these can often be set up as compiler constraints. Library-based approaches (whether using a textual or visual language) can be very efficient if all of the required functions are available within the library. The design then becomes one of piecing together library components, which usually have implementations that have been optimised for a particular family of FPGA.

In the Massey University laboratory, a combination of VHDL and Handel-C is used. VHDL is used primarily for developing interfaces to external components where precise control is required. All of the algorithmic design is in Handel-C, which is an intermediate level C-based hardware description language.

This gives significant productivity gains over VHDL through its higher level algorithmic representation, but also enables control at the lower levels where necessary.

It is important to keep in mind that FPGA-based development is hardware design. Design for hardware and for software requires different skills. Even if a C-based language is used to represent the design, it is important to keep in mind the architecture that is being built or is implied by the algorithm. Not all software algorithms map well to hardware. For example, recursion is not available unless hardware is explicitly built for saving the context and recalling it again later (Skliarova and Sklyarov, 2009). Under real-time constraints, there is limited access to memory. These and many other aspects of hardware design usually require that the algorithm be transformed rather than simply mapped to hardware. Again, this requires a hardware orientated mindset.

Image processing algorithm development, however, is usually considered software design. Therefore, the development of efficient FPGA-based image processing systems requires a mix of both hardware and software engineering skills. The differences between these is explored in more detail in the next chapter.

4

Design Process

The process of developing an embedded image processing application involves four steps or stages (Gribbon *et al.*, 2007). The relationship between these is illustrated in Figure 4.1. The problem specification clearly defines the problem in such a way that the success of the proposed solution can be measured. The algorithm development step determines the sequence of image processing operations required to transform the expected input image or images into the desired result. Architecture selection involves determining the computational structure of the processors that are required to execute the algorithm at both the application and operation levels. Finally, system implementation is the process of mapping the algorithm onto the selected architecture, including construction and testing of the final system.

It has been observed that the development of complex algorithms on FPGAs is sensitive to the quality of the implementation (Herbordt *et al.*, 2007). To gain a significant speed improvement over a software implementation, it is necessary that a significant fraction of the algorithm can be parallelised. Often the speedup obtained from implementing an application on an FPGA is disappointing. One of the main factors is that it is not merely sufficient to port an algorithm from software onto an FPGA implementation. To obtain an effective and efficient solution, it is necessary for the computation and the computational architecture to be well matched. While algorithm development is usually considered part of the software engineering realm, the design of the processing architecture is firmly in the domain of hardware engineering. Consequently, FPGA-based image processing requires a mix of both skill sets. To fully exploit the resources available on an FPGA, the system implementation stage often requires changes to both the algorithm and the architecture. As a result, the development process is usually iterative, rather than performed as discrete distinct steps as implied by Figure 4.1.

In the description of each of the stages below, the important tasks within each step are identified and explained. In particular, the differences between designing for implementation on an FPGA and on a standard software-based system are highlighted.

4.1 Problem Specification

Deriving a detailed specification of the problem or application is arguably the most important step, regardless of whether the resultant system is software or hardware based. Without an adequate specification of the problem, it is impossible to measure how well the problem has been solved. It is impossible to know when a project has been completed when ‘completed’ has not been defined!

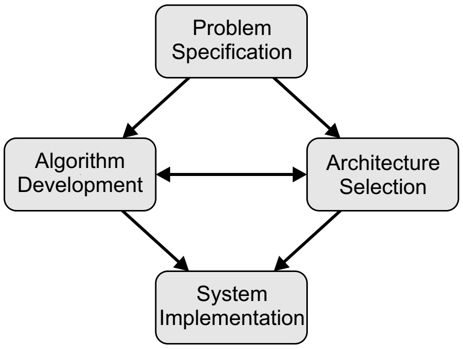


Figure 4.1 The relationship between the four main steps of the design process.

Within image processing, it is common to have a relatively vague description of the problem. For example, a produce inspection problem may be described as inspecting the produce in order to remove blemished and damaged produce. While this may be the aim, such a description is of limited value from an engineering design perspective. In an engineering context, determining the problem specification is usually called *requirements analysis*. To be useful, the problem specification needs to be specific, complete, achievable and measurable (DAU, 2001). These will be defined more fully in the following paragraphs.

A problem specification should clearly describe the problem rather than the solution. It should cover what the system needs to do and why it needs to do it, but should not cover how the system should be implemented; that is the result of the whole design process. For the specification to be specific, it will need to address at least three areas (DAU, 2001). The first is the system functionality – what the system needs to be able to do. In an image processing application, it needs to specify the desired result of image processing. Secondly, the performance of the system must be addressed – how well must it perform the functions. For real-time image processing, important aspects of this are the maximum allowable latency and the number of images or frames that must be processed every second. If the problem involves classification, then for non-trivial problems it will be inevitable that the system will make misclassifications. If the decision is binary, the allowable failure rate should be specified in terms of both false acceptance and false rejection rates. The third area of consideration is the environment in which the system will operate. Applied image processing consists of more than just the image processing algorithm; it is a systems engineering problem that requires consideration and specification of the complete system (Bailey, 1988). Other important aspects to consider include (Batchelor and Whelan, 1994): lighting, optics and interfacing with supporting hardware and machinery. The relationships between the image processing system and the rest of the system need to be carefully elucidated and defined.

To illustrate some of these points, consider the kiwi fruit grading application introduced in Section 1.4. The system had to detect both blemished and damaged fruit based on their visible surface characteristics. The output was a binary decision: whether the fruit was acceptable or not. The minimum processing rate, to work in with the grading lines at the time, was four kiwifruit per second, although the latency could be several seconds if necessary. The initial pilot investigated prescreening the fruit (Bailey, 1985), rather than performing a full grading. To enable the manual grading line to operate at close to full capacity, the proportion of reject fruit at the output needed to be less than 3% of the total. Profitability required minimising the number of false rejects, so when in doubt the system needed to err on the side of retaining fruit with the false acceptances removed by the manual grading stage. It was also desirable to integrate the system at the front of the existing grading line rather than as a separate stand-alone grading system. The system therefore required a mechanism for removing rejected fruit from the grading line.

Note that almost as much of this specification relates to the working environment as to the image processing requirements.

The problem specification must be complete. It should consider not only the normal operation, but should also consider how the system should behave in exceptional circumstances. If the system is interactive, then the interface between the user and the system needs to be clearly defined, at least in terms of functionality. The specification should also consider not only the operation of the system, but also address the desired reliability and the required maintenance over the expected life of the system. Again, all aspects of the system should be addressed, not just the image processing components.

The system as defined by the problem specification must be achievable. In a research context, it is essential that the research questions can be answered by the research. For a development project, the final system must be technically achievable at an affordable cost.

Finally, the problem specification must be measurable. This enables the resultant system to be evaluated objectively to verify that it satisfies the specification. Consequently, the requirements should be quantitative, and should avoid vague words such as excessive, sufficient, resistant and so on (DAU, 2001). It is important to determine the constraints on the system. For embedded real-time systems, this includes the frame rate, system latency, size, weight, power and cost constraints. The resulting set of requirements must be mutually consistent. A distinction should be made between hard constraints (those that are essential for successful operation or completion of the project) and soft constraints (those that are considered desirable, but may be relaxed if necessary in the final system). Inevitably, there will be conflicts between the different constraints, particularly with real-time processing. A common trade-off occurs between speed and accuracy (Kehtarnavaz and Gamadia, 2006). These conflicts must be resolved before the development begins.

Successful specification of the problem requires comprehensive knowledge of the problem or task to which image processing is being applied. This knowledge is important, since it increases the likelihood that the application level algorithm will be robust. It is used to select the representative set of sample images that will be used to develop and test the imaging algorithm. During the algorithm development process, it also guides the selection of the image features that are measured. Without such *problem knowledge*, it is easy to make invalid assumptions about the nature of the task. There is always a danger that a resultant algorithm will work well for the specific cases that were used to develop it, but not be sufficiently general or robust to be of practical use. Often the system developer has limited knowledge of the problem. In such cases, it is essential to have regular feedback and verification from the client or other problem domain expert, particularly during the algorithm development phase.

4.2 Algorithm Development

The task of image processing algorithm development is to find a sequence of image processing operations that transform the input image into the desired result. Algorithm development is a form of problem solving activity, and therefore usually follows heuristic development principles (Ngan, 1992). Like all problem solving tasks, there is not a unique solution. The usual way of progressing is to find an initial solution and then refine it until it meets the functionality required by the problem specification.

Before the algorithm can be developed, it is necessary to capture a set of sample images. These should be representative of the imaging task that is to be performed and should be captured under conditions that match as closely as possible the conditions under which the system is expected to operate.

In scenarios where it is possible to control the lighting (machine vision for example) then choosing appropriate lighting is essential to simplify the task (Uber, 1986; Batchelor, 1994). In some applications, much of the problem can be solved using structured lighting techniques (such as projecting a grid onto the scene; Will and Pennington, 1972). Conversely, inadequate lighting can make segmentation more difficult by reducing contrast, illuminating the background or object unevenly, saturating part of the scene, or introducing shadows or specular reflections that may be mistaken for other objects (Bailey, 1988). If the lighting is likely to be variable (for example mobile robotics or video surveillance using natural lighting) then it is essential that the set of sample images include images captured under the full range of lighting conditions that the system is expected to work under.

4.2.1 Algorithm Development Process

Once the sample images have been captured, one or two of these are used to develop the initial sequence of image processing operations. Developing an image processing algorithm is not as straightforward as it sounds. There are many imaging tasks that humans can perform with ease, that are difficult for computer-based image processing. One classic example is face recognition, something humans can do naturally, without thinking, whereas it is a difficult computer vision problem. This dichotomy is sometimes called the 'trap of the two-legged existence theorem' (Hunt, 1983). Because humans can perform the task, it is known that a solution to the problem exists. However, finding that solution (or indeed any solution) in the context of computer vision can be very difficult. While significant advances have been made in understanding how the human brain and visual system works, this does not necessarily lead to algorithmic solutions any more than understanding how a computer works can lead to programming it for a specific task.

One of the difficulties is that there is still little underlying theory for developing image processing algorithms. While attempts have been made to formulate the process within a more theoretical framework, for example in terms of mathematical morphology (Vogt, 1986) or statistics (Therrien *et al.*, 1986), these are incomplete. Even within such frameworks, algorithm development remains largely a heuristic (trial and error) process (Bailey and Hodgson, 1988). An operation is chosen from those available and applied to the image. If it performs satisfactorily, it is kept and an operation is found for the next step. However, if the operation does not achieve the desired result, another is tried in its place. Algorithm development must rely heavily on using the human visual system to evaluate the results of applying each operation. The experience of the developer plays a significant role in this process. The brute force search for a solution to the algorithm development problem can be significantly streamlined by the application of appropriate heuristics (Ngan, 1992).

This process requires an interactive image processing system that supports such experimentation (Brumfitt, 1984). The algorithm development environment needs to have available a wide range of different operations, because it is generally not known in advance which operations will be required in any particular application. It also needs to support the development and integration of new operations should a particular operation not be supplied with the development environment. It requires an appropriate image display for viewing and evaluating the effects of an operation on an image. The processing time of each operation must be sufficiently short so as not to distract the developer from their task. It has been suggested that within the development environment operations should take no longer than about fifteen seconds (Cady *et al.*, 1981). It is also desirable to have some form of scripting capability, to avoid the need for re-entering complex sequences of commands when testing the algorithm on different images.

The sequence of operations within an algorithm generally follows that outlined in Section 1.3. However, the particular operations required depend strongly on the image processing task. The order in which the operations are added to the algorithm may not necessarily correspond to the order that they appear in the final algorithm. For example in the kiwifruit grading application described in Section 1.4, the key step that forms the basis of the algorithm is using the convex hull to create the dynamic model. After this was tested, it was necessary to introduce the background subtraction and noise filtering as preprocessing steps to make the modelling work reliably. Later, while testing the algorithm on a range of images, the contrast normalisation step was introduced into the algorithm to compensate for the fact that some fruit were naturally darker or lighter than others were.

4.2.2 Algorithm Structure

The remainder of this section briefly reviews the operations that may be used at each stage within the algorithm, with particular emphasis on how the operations work together to give a robust algorithm. There are many good references that review image processing operations and their functionality (Pratt, 1978; Jain, 1989; Castleman, 1996; Russ, 2002; Gonzalez and Woods, 2004; 2008; Solomon and Breckon, 2011). The FPGA implementation of many of these operations is considered in more detail in Chapters 6–11. Here, the focus is on where the operations fit within the application level algorithm.

Appropriate preprocessing is essential to make the algorithm robust to the variations encountered within the range of images. The primary goal of preprocessing is to enhance the information or features of interest in the scene while suppressing irrelevant information. However, the necessity of an operation in a particular application depends strongly on the detailed properties and characteristics of the operations that follow in the algorithm.

Physical or environmental constraints may mean that the input images are not ideal. To compensate for deficiencies in the image capture process, preprocessing can be used: to remove background or dark current noise; to correct for nonlinearity in the image transfer function; to correct for distortion caused by optics or camera angle; and to compensate for deficiencies in illumination. Image filtering may be used to smooth noise or to enhance and detect edges, lines and other image features. Examples of some of these preprocessing operations are illustrated in Figure 4.2.

One way of suppressing irrelevant information is to normalise the image content. Intensity normalisation, such as contrast expansion or histogram equalisation can be used to compensate for variations in lighting, reflectivity or contrast. Position and size normalisation may be used to centre, orientate and scale an image or object to match a template or model. This may require an initial segmentation to first locate the object.

Segmentation is the processes of splitting the image into its meaningful components. This can vary from separating objects from the background to separating an image into its constituent parts. Segmentation can be based on intensity, colour, texture or any other local property of the pixel values. There are two broad approaches to segmentation, as shown in Table 4.1. Edge-based methods focus on detecting the boundaries between regions, with the regions defined implicitly by the enclosing boundaries. They usually use edge detection or gradient filters to detect edges and then link the significant edges to form the boundaries. Region-based methods work the other way around. They effectively classify pixels as belonging to one class of objects or another based on some local property, with the edges determined implicitly by changes in property at the region boundary. They are characterised by filtering (to enhance or detect the property used for segmentation; Randen and Husoy, 1999) followed by thresholding or statistical classification methods to assign each pixel to a region. Both edge and region based methods can be performed either globally or incrementally (Bailey, 1991). Global methods perform the segmentation by processing every pixel in the image. Incremental methods only operate locally on the part of the image that is being segmented. They are usually initialised with a seed pixel and work by extending the associated edge or region by adding neighbouring pixels until a termination condition is met. Segmentation may also involve using techniques for separating touching objects such as concavity analysis, watershed segmentation and Hough transform.

The next stage within the algorithm is to extract any required information from the image. In image analysis applications, the measurements are the desired output from the image processing procedure. The measurements may be tangible (physical size, intensity) or abstract (for example, the fractal dimension to quantify roughness (Brown, 1987) or the average response to a particular filter to characterise texture (Randen and Husoy, 1999)). Most tangible measurements require the imaging system to be calibrated to determine the relationship between pixel measurements and real world measurements.

In other applications, a set of image or object feature measurements is used to characterise or classify the image or object. Boundary features are properties of the edges of the object (such as contrast) while region features are properties of the pixels within the region (colour, texture, shading). Shape features

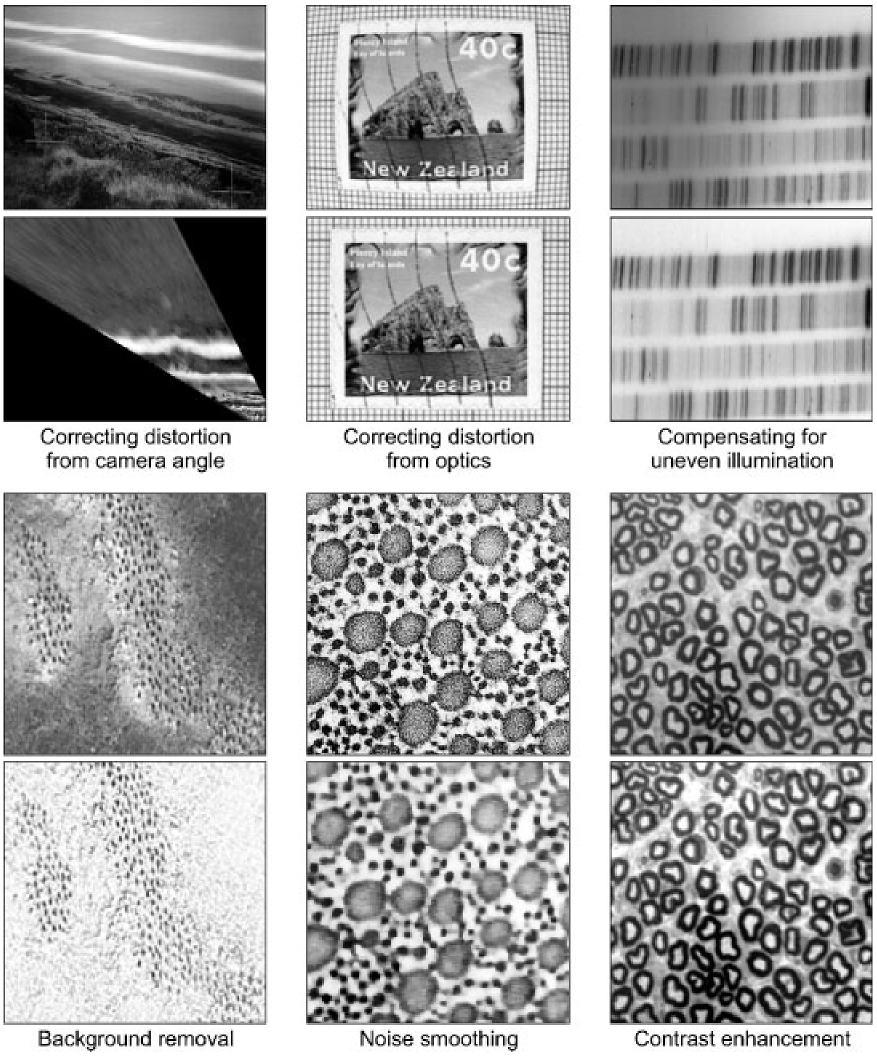

Correcting distortion from camera angle
Correcting distortion from optics
Compensating for uneven illumination
Background removal
Noise smoothing
Contrast enhancement

Figure 4.2 Example preprocessing operations.

consider the whole object (area, height, width, perimeter, enclosing rectangle, number of corners, convex hull area and perimeter, moments, best-fit ellipse, Fourier descriptors etc.). Virtually anything that can be measured, or can be derived from multiple measurements, can be used as a feature. The set of measured features makes up a multidimensional feature vector, with each feature corresponding to a dimension.

Table 4.1 Image segmentation methods

	Edge-Based Segmentation	Region-Based Segmentation
Global processing	Edge detection and linking	Thresholding
Incremental processing	Boundary tracking	Region growing

Classification is the process of recognising or assigning meaning to an object based on the measured feature vector. Each point in the feature space represents a particular combination of feature values. When an appropriate set of features is chosen, the feature vectors for objects within a class should be similar and fall in the same region of feature space. The classification process therefore involves clustering or segmentation within the feature space. With such a large number of possible features to choose, the feature space potentially has a high dimensionality. However, there is often a strong correlation between many of the features. Choosing the best features to use can be a bit of an art. The goal in selecting the feature set is to choose only as many features as are necessary to distinguish the objects as unambiguously as possible. Obviously, a starting point is to look at the features used to distinguish the objects of interest. An analysis of the covariance can help to identify significant correlations between the features. Where there are strong correlations, consider replacing one or more of the features by a derived feature. For example, the length and width of an object are often correlated, especially as the scale changes. In this case more information can often be obtained from the aspect ratio (length divided by width) than either the length or width. Unless the size or some other feature is used directly to distinguish between classes, it is often better to use dimensionless derived features such as aspect ratio, compactness and so on. Principal components analysis and related techniques can help to identify the features that account for most of the variance in the population.

Given a set of feature vectors (for example those derived from the set of sample images) there is a wide range of techniques used to perform the classification: decision trees, feature template-based classification, Bayesian classification, neural networks, and support vector machines, amongst many others. In most applications, each object within the set of sample images has an associated desired classification. These associations form the training data for the classification. Training is the process of determining the classification parameters that minimise the classification errors. This may be as simple as determining the average feature vector for the objects belonging to each class for feature template classification, or as complex as determining the optimum set of weights for a neural network.

Some applications have no training data but a large number of samples. Unsupervised classification is the process of determining the classes purely from an analysis of the resultant feature vectors. It derives the classification by looking for groups or clusters of samples within the data provided. One example of an unsupervised classifier is K-means clustering (Leeser *et al.*, 2000).

Finally, the data that is extracted may require further processing depending on the application. For example, a Kalman filter may be used for object tracking (Kalman, 1960), or in mobile robotics the data may be used to update a map or to plan a route through the scene. In many computer vision applications, the data derived from the image is used to update the parameters of an underlying model representing the scene.

Once the initial algorithm has been developed, often on only a small set of images, the next step is to refine the algorithm using a wider range of images. The algorithm is tested on the remaining images within the representative sample to ensure that it is able to handle all of the expected conditions in the correct manner. Algorithm refinement is most likely to affect the preprocessing operations to make the complete algorithm more robust. Particular care should be taken to check the algorithm over the full range of lighting conditions that it is expected to operate under.

Very few algorithms are perfect. There are two key points where the algorithm is most likely to fail. The first is segmentation failure, where the object is not correctly segmented from the background. Often a partial segmentation is worse than a complete failure, because the features obtained will generally not reflect those of the complete object. With any segmentation failure, it is important to re-evaluate any assumptions that are made about the task or scene to check that they are not the cause of the failure, and correct these if necessary. If the problem is caused by poor contrast or noise, this may require further preprocessing to correct, or even redesign of the image capture system to obtain better starting images. If possible, the algorithm should detect whether or not the segmentation was successful, or at least act appropriately when it encounters errors. Remember the ‘garbage in, garbage out’ principle – if the image quality is very low, the algorithm is not likely to work reliably.

The second point of failure is classification failure, which can have several causes. The first, and probably most common, is when the feature vector distributions for two or more classes are not completely separated. In the region of overlap in feature space, it is impossible to distinguish to which class an unknown feature vector belongs. One solution is to find an additional feature (or combination of features) that is able to better discriminate between the overlapping classes. Misclassification can also result from under-training, where insufficient samples are used to represent each of the classes adequately. As a result, the region of feature space associated with one or more classes in inaccurately represented. A particular danger occurs if the samples in the training set are not representative and the set of feature vectors obtained is biased in some way. The opposite of under-training is over-training. This can be a particular problem with neural network-based classifiers. An over-trained classifier gives excellent results on the set of images used for training, but gives very poor results on previously unseen images. The classifier is able to recognise the training examples but does not generalise well to other examples of the same class. A fourth cause of classification failure results from using an inappropriate classifier, which does not have the power to make the required distinctions. An example is a neural network with insufficient nodes in the hidden layer, or using linear discriminant analysis when the classes cannot be separated using hyperplanes. Finally, faulty or inaccurate segmentation may result in feature vectors that are outliers of the correct distribution. Such outliers can have an undue influence over the training of some classifiers (for example, support vector machines) and should be eliminated before training.

4.2.3 FPGA Development Issues

The image processing algorithm cannot be developed directly on the FPGA. The main reason is that the development cycle times are far too long to allow interactive design. Assuming that an implementation of the desired operation is available in a hardware description language, the compilation and place and route times required to map it onto an FPGA are prohibitive.

Therefore, the algorithm first needs to be developed in a software environment. For this, any image processing algorithm development environment that satisfies the requirements described in the previous section is suitable. Such an environment will usually be distinct from the FPGA development environment.

One of the advantages of separating the algorithm development from the FPGA implementation is that the application level algorithm can be thoroughly tested before the complex task of mapping the algorithm onto the target hardware. Such testing is often much easier to perform within the software-based image processing environment, which has been designed to facilitate this purpose. Testing the hardware implementation then becomes a matter of verifying that each of the image processing operations has been implemented correctly. If the operations behave correctly, then the whole image processing algorithm will also function in the same way as the software version.

The algorithm development process does not stand alone. The implementation process, as described shortly, is often iterative as the algorithm is modified to make all of the operations architecturally compatible. Whenever any changes are made to the image processing algorithm, it is necessary to retest the algorithm to ensure that it still performs satisfactorily.

Some algorithm testing must be deferred to the final implementation. Evaluating the effectiveness of an image processing algorithm on a continuous, noisy, real-time video stream is difficult in any environment.

The output from the algorithm development step is the sequence of image processing operations required to transform the input image or images into the desired form. It has been tested (within the software-based image processing environment) and shown to meet the relevant accuracy and robustness specifications for the application.

4.3 Architecture Selection

Once an initial algorithm has been developed, it is necessary to define the implementation architecture. It is important to obtain a good match between the architecture and both the application level and operation level algorithms. A poor match makes inefficient use of the available resources on the FPGA and can limit the extent to which the algorithm may be accelerated. Consequently, the system may not meet the targeted speed requirements or the design may require a larger FPGA than is really needed.

The architecture can be considered at two distinct levels. The system level architecture is concerned primarily with the relationship and connections between the components of the system. Three aspects of the system architecture are of primary importance: the relationship between the FPGA and any other processing elements (whether they be a standard computer, or DSP); the nature of any peripherals and how they are connected; and the nature and structure of external memory within the system. The computational architecture is concerned primarily with how the computation of the algorithm is performed. This includes the forms of parallelism that are being exploited and the split between using hardware and software for implementing the algorithm.

On a software-based image processing system, the computational architecture is fixed and the system architecture is generally predefined. For an FPGA based system, however, nothing is provided and all aspects of the architecture must be developed. Once exception to this is perhaps the use of development boards for prototyping, where much of the system level architecture is fixed by the board vendor.

4.3.1 System Level Architecture

Perhaps the biggest factor of the system architecture is whether the system has a conventional serial computer as a host processor or whether the FPGA stands alone. Compton and Hauck (Compton and Hauck, 2002) identified four distinct ways in which programmable logic could be coupled with a host computer system within this spectrum, as illustrated in Figure 4.3. Todman *et al.* (2005) added a fifth coupling architecture: that of a CPU embedded within the FPGA (either as an embedded hard-wired core or as a soft core built from the FPGA fabric).

In the most closely coupled arrangement, the programmable logic provides reconfigurable functions units integrated within the host CPU. This requires that the programmable logic be built inside the processor and allows instructions to be customised according to the application that is being executed (Hauck *et al.*, 2004). The purpose is to accelerate key instructions within the host processor, while retaining the simplicity of the standard software processing model. Data is usually passed between the reconfigurable logic and CPU through shared registers (Compton and Hauck, 2002).

With modern FPGAs, this approach may be readily taken by building the processor from the logic within the FPGA (the fifth architecture identified by Todman *et al.,* 2005). Such a processor may be augmented with whatever custom instructions the developer requires. However, one limitation of building a custom processor is that the developer also needs to build all of the associated tools (optimising

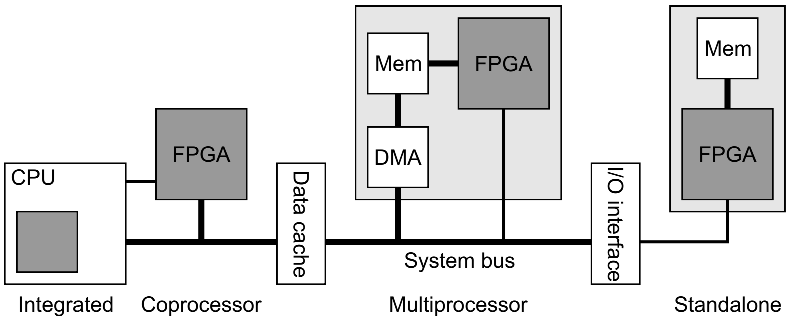


Figure 4.3 Coupling between a host processor and an FPGA. (K. Compton and S. Hauck. "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, Vol. 34: 171–210, © 2002 Association for Computing Machinery, Inc. Reprinted by permission.)

compilers, etc.). This may be overcome by using vendor provided soft-core processors, such as the NIOS II with Altera FPGAs, or MicroBlaze with Xilinx FPGAs. The NIOS allows the provided processor to be augmented with custom instructions. One limitation of custom instructions is the restriction in the amount of data that may be passed to or from the function unit with each instruction, and the limited time that is available to complete the processing. It is usually necessary to stall the main processor if the custom instruction takes a long time to execute.

In the next more loosely coupled configuration, an FPGA can be used as a coprocessor (Miyamori and Olukotun, 1998), to accelerate particular types of operations in much the same way that a floating-point coprocessor accelerates floating-point operations within a standard CPU. In this configuration, the FPGA-based processing unit is usually larger than an ALU or similar functional unit within the host. The coprocessor operates on its data relatively independently of the main processor. The main processor provides the data to be processed (either directly or indirectly) and then triggers the coprocessor to begin executing. The results are then returned after completion (Compton and Hauck, 2002). The operations performed can run for many clock cycles without intervention from the host processor. This allows the host to continue with other tasks, while waiting for the coprocessor to complete. Note that the FPGA-based coprocessor has access to the data from the host's cache.

Using dedicated hardware as a coprocessor is most commonly used when the main application is implemented within software, but time critical or computationally dense sections of an algorithm are accelerated through direct hardware implementation. There may also be multiple hardware processors, each developed to accelerate a critical part of the algorithm.

In the next loosely coupled arrangement, the FPGA system effectively behaves as an additional processor in a multiprocessor system (Compton and Hauck, 2002). This requires the FPGA system to process large sections of an application independently of the host to be the most effective. The FPGA system sits on the system bus, with DMA used to transfer data from the host system memory into local memory. Depending on the processing performed, the results can either be transferred back to the host through DMA, or via registers within the FPGA that are mapped into the address space of the host. The host system configures the FPGA, indicates where the data to be processed is located, and triggers the FPGA system to begin its execution. When processing is complete, this is signalled back to the host processor.

For image processing, the performance is strongly influenced by the I/O bandwidth, the amount of local memory available and its arrangement (Benitez, 2002). The independent processor arrangement also maps well to the image processing pyramid introduced in Figure 1.7, with the lower level operations implemented directly in hardware, and with software used to implement the higher level operations.

In the final configuration, the FPGA operates independently of the host system, which may not even be connected. In this arrangement, the communication with the host is generally sporadic and of relatively low bandwidth. In the stand-alone configuration, usually all of the processing is performed by the FPGA, including any embedded software processor it may contain.

Within image processing, where a large degree of parallelism can be exploited by the operation level algorithms, either multiprocessor or stand-alone configurations are usually the most appropriate. The integrated and coprocessor configurations are usually too tightly coupled to fully exploit parallelism in low level image processing operations.

For high performance computing applications, the hosted multiprocessor configuration is the most common. For image processing, the host is usually responsible for image capture and display, and any user interaction. The host operating system needs to be augmented to enable it to reconfigure the FPGA (Andrews *et al.*, 2004). This is usually dynamic reconfiguration, allowing the host to change the FPGA configuration depending on the application (Wigley and Kearney, 2001; Sedcole *et al.*, 2007). The host operating system is also responsible for distributing tasks between the serial processor and the FPGA system (Steiger *et al.*, 2004; Lubbers and Platzner, 2007), and synchronising communications between all of the processes (both hardware and software) (Mignolet *et al.*, 2003; So, 2007).

The stand-alone configuration performs all or most of the work on the FPGA. This does not necessarily mean that there is no serial processor. Some modern FPGAs have a processor built into them, or it is always possible to include a soft-core processor within the fabric of the FPGA. However, the FPGA is responsible for any image capture and display. Any user interaction must also be built into the FPGA, either as hardware modules, or using the serial processor. Unlike on a conventional serial processor, an FPGA has no operating system (Bailey *et al.*, 2006). If there is an embedded serial processor, this may have an operating system running on it, but the drivers will be limited to common peripherals. Much of the low level interfacing must be built in hardware, and if this is to be interfaced to the processor, device drivers must be written. If no serial processor is used, all of the user interface must be built in hardware (Buhler, 2007).

For debugging and initial development, the stand-alone configuration may have a host computer. This can be used for directly downloading the FPGA configuration and may provide some support for debugging. However, for normal running the FPGA configuration is often loaded directly from flash memory, avoiding the need for a host processor.

Also at the system level is the memory architecture. A key factor to consider is the degree of coupling between the memory, FPGA and any CPU. Only some of the arrangements are shown in Figure 4.3. Distinction also needs to be made between shared system memory and memory local to the FPGA. For local memory, the technology is also an important consideration. Many high speed memories, especially those with large capacity, operate most efficiently in burst mode. In this mode, a predefined number of successive memory locations may be read or written in successive clock cycles. Dynamic memories require separate addressing of the row and column within the memory, sometimes with several clock cycles between the row address and the column address. Dynamic memories also must be refreshed regularly, with access to the memory unavailable during the refresh period.

Other memory technologies may allow random access on successive clock cycles, although for high speed operation they are often pipelined with the address having to be provided one or two clock cycles before the data is available. This may have implications for the speed of some operation level algorithms, especially where the memory address is data dependent.

4.3.2 Computational Architecture

The computational architecture defines how the computational aspects of the algorithm are implemented. A particular focus is the form of parallelism being exploited to achieve the algorithm acceleration. Not all aspects of the algorithm may be amenable to hardware implementation, so a later section focuses more on the split of the application level algorithm between hardware and software execution.

4.3.2.1 Stream Processing

The main bottleneck of software-based image processing is that it is memory bound. An operation reads the pixel values from memory, processes them, and writes the results back to memory. Since each operation performs this, the speed of the algorithm will be largely limited by the number and speed of memory accesses. Stream processing can overcome this bottleneck by pipelining operations. The input data is read once from memory, or streamed from the camera and passed into the first operation. Rather than write the results back to main memory, they are directly passed on to the next operation. This will eliminate at least two memory accesses for each stage of the pipeline: writing the results and reading them again as input for the next operation. The benefits of streaming can therefore be significant.

For embedded vision applications, at some stage the data must initially be read from the camera. If possible, as much processing should be performed on this data while it is passing through the FPGA before it is written to memory. If the whole application can be implemented as a single streamed pipeline then it may not even be necessary to use a frame buffer to hold the image data. Similarly, if an image is being displayed, the pixels need to be streamed to the display. Again, much benefit can be gained by performing as much processing on-the-fly as the data is being streamed out.

One of the key characteristics of stream processing is a fixed clock rate, usually one clock cycle per pixel. This clock rate is constrained by the input (from a video camera) or output (to a display). At the input, each pixel must be processed as it arrives, otherwise the data is lost. Similarly, a pixel must be produced on the output at each clock cycle; otherwise there will be no data to display and the output will appear blank for that pixel. Consequently, the design of stream processing systems is synchronous, at least on the input and output. If every stage of the pipeline can be operated synchronously, then the design is simplified.

Stream processing is well suited to low level image processing operations, such as point operations and local filters, where processing can be performed during a raster scan through the image. If an operation, for example a filter, requires data from more than one pixel, it is necessary to design local buffers for each operation to cache the necessary data. This is because the synchronous nature of stream processing limits memory access to one read or write per clock cycle. To satisfy this constraint, the algorithms for some operations may need to be significantly redesigned from that conventionally used in software. The fixed timing constraint arising from the predetermined constant throughput of stream processing may be overcome by using low level pipelining. This splits a more complex calculation over several clock cycles while maintaining the throughput of one pixel per clock cycle.

4.3.2.2 Systolic Arrays and Wavefront Processing

The idea of stream processing may be extended further to systolic arrays (Kung, 1985). A systolic array is a one or two dimensional array of processors, with data streamed in and passed between adjacent processors at each clock cycle. It is this latter characteristic that gives systolic arrays their name. Since the communication is regular and local, communication overheads are low. A systolic array differs from pipeline processing in that data may flow in any direction, depending on the type of computation being performed. For many systolic arrays, the processing elements in the array are often relatively simple and perform the same operation at each clock cycle. This makes systolic arrays well suited to FPGA implementation.

The difference between a systolic array and a wavefront array is that systolic arrays are globally synchronous, whereas wavefront arrays are asynchronous. A wavefront array therefore requires handshaking with each data transfer. This means that a wavefront array is self-timed and is able to operate as fast as the data is able to be processed. Unfortunately, such asynchronous systems are not well suited to implementation on synchronous FPGAs. However, they may be able to be implemented by self-timed architectures such as the Achronix Speedster (Section 2.4.4).

Systolic arrays work best where the processing is recursive with a regular data structure. The systolic array effectively unrolls the recursive loop to exploit concurrency in a way that uses pipelining to make use of the regular data structure. This makes such an architecture well suited for matrix operations (Kung and Leiserson, 1978) and for implementing many low level image processing operations (Kung and Webb, 1986). They are most suited where the operations are relatively simple and regular, such as filtering (Hwang and Jong, 1990; Diamantaras and Kung, 1997; Torres-Huitzil and Arias-Estrada, 2004), connected component labelling (Nicol, 1995; Ranganathan *et al.*, 1995) and computing the Fourier transform (Nash, 2005). However, a single array is not well matched for implementing a complete application level image processing algorithm. The streamed nature of the inputs and outputs means that a systolic array processor would fit well with stream processing.

4.3.2.3 Random Access Processing

In contrast with stream processing, random access processing allows pixels to be accessed from anywhere in the memory as needed by an operation. Consequently, there are no explicit requirements in the pattern of data access. It requires the image to be available in a frame buffer rather than as an input stream. Random access processing relaxes the hard timing constraint associated with stream processing. If necessary, several clock cycles may be used to process each pixel accessed or output.

Random access processing is, therefore, most similar to software processing. Consequently, it is generally easier to map a software algorithm to random access processing than to stream processing. However, simply porting a software algorithm to hardware using random access processing will generally give disappointing performance. This is because the underlying software algorithm will usually be memory bound and performance will be limited by memory bandwidth. Where possible, steps must be taken to minimise the number of memory accesses by using local buffers. It may also be necessary to design an appropriate memory architecture to increase the memory bandwidth. Some of these mapping techniques are described in the next chapter.

For complex algorithms, the output may not be regular in the sense that there may be a variable number of clock cycles for each pixel produced. This can make it more difficult to synchronise the dataflow between successive operations in the application level algorithm. Therefore, it may be necessary to design synchronisation buffers to manage the uneven data flows. If the inputs and outputs are in a sequential order, then a FIFO buffer provides such synchronisation. Otherwise, synchronisation may require a frame buffer.

The clock speed for random access designs is generally not constrained. Therefore, the throughput can be increased by maximising the clock speed. This may be accomplished by using low level pipelining to reduce the combinatorial delay. The operation level algorithm may be further accelerated by examining the dataflow to identify parallel branches and implementing such operations in parallel.

One form of this data parallelism is to partition different parts of the image to separate processors. This requires multiple copies of the hardware, one for each parallel processor. Each copy will usually have its own local memory to avoid problems with limited bandwidth. However, when considering such parallelism, it is also important to consider not only the operation being parallelised, but also the operations before and after it in the processing chain. One overhead to consider is that required to transfer the data from system memory (such as a frame buffer) to local memory and back. If care is not taken, these overheads can remove much of the performance improvement gained by partitioning the processing. Often the processor will need to be idle during the data transfer phase unless there are sufficient memory resources to enable some form of bank switching to be used.

4.3.2.4 Massively Parallel Architectures

The extreme case of such data parallelism is to use a massively parallel architecture. Such architectures exploit the fact that many image processing operations consist of a series of operations performed independently on each pixel. In software, the outermost loop for such operations will iterate through the pixels within the image. A massively parallel architecture effectively unrolls this outermost loop and implements the iterations on each pixel in parallel. The result is a separate processor for each pixel or block of pixels. A massively parallel processor is able to process whole images in a few clock cycles, regardless of the size of the image. Algorithms that rely primarily on local information can readily be mapped onto such architectures.

While nice in theory, there are two limitations to massively parallel architectures. The first is the communication bottleneck. Massively parallel algorithms work well when the data is already present within the processors. However, getting the data into the array (from the camera) and out of the array (to a display if necessary) can often take longer than the actual processing of the image. A second communication problem occurs if the processing requires sending or receiving data from outside a local region. The communications must either be broken down into a series of local steps, with the processors spending a significant proportion of their time forwarding data, or a hierarchical communication scheme established for longer distance communication and data collection.

The second limitation is that a massively parallel processor requires significant resources on the FPGA, even if each processor is relatively simple. This limits the use of such architectures to small images, or allocating a block of pixels to each processor. For example, both the communication and resource limitations are reduced by allocating one processor to each column or row of pixels within the image.

As a result of these limitations, massively parallel architectures are not considered in detail any further in this book.

4.3.2.5 Hybrid Processing

The best performance can usually be achieved when the whole application can be implemented in a single processing mode and memory access to a frame buffer is minimised. This may require considerable redesign and development of the operation level algorithms to make them architecturally compatible. It may also require modifying the application level algorithm to avoid using operations that do not integrate well with the others.

In some applications, it may not be possible to implement the entire algorithm using stream processing. In such cases, it may be necessary to combine stream and random access processing. The introduction of one or more frame buffers will usually increase the latency of the algorithm. However, for some operations where each output pixel may depend on data from anywhere within the image, this cannot be avoided. With hybrid processing, frame buffer access may be minimised by maximising the use of stream processing.

4.3.2.6 Computational Architecture Design

The key to an efficient hardware implementation is not to port an existing serial algorithm, but to transform the algorithm to produce the computational architecture. There are two stages to this process (before implementation), as shown in Figure 4.4.

The first stage is to analyse the algorithm for the underlying algorithmic architecture. In this process, it is important to look at the whole algorithm, not just the individual operations. This is because gains can be made by considering the interactions between operations, especially if the operations can be made architecturally compatible. A dataflow analysis is one common method of identifying the underlying computational architecture (Johnston *et al.*, 2006b; Sen *et al.*, 2007). This allows the developer to get an overview of the design and investigate how it can be modified. In this process of analysing the algorithm, the underlying algorithm needs to be changed or transformed to one that is more readily able to map efficiently to a hardware implementation.

Several such transformations may be applied to the algorithm (Bailey and Johnston, 2010). The first is to convert as many operations as possible to pipelined stream processing to eliminate intervening frame buffers. Any operation that may be implemented by processing the data in a raster-scanned order through the image can be streamed. Some algorithms that are not naturally streamed may be redesigned to use stream processing (for example chain coding, as described in Section 11.3).

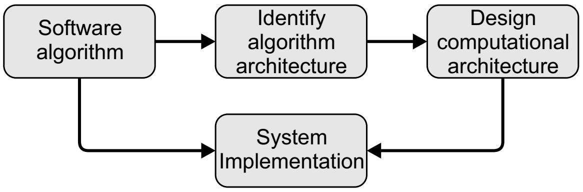

Software algorithm
Identify algorithm architecture
Design computational architecture
System Implementation

Figure 4.4 Transforming an algorithm to produce the computational architecture.

The iterations present in many loops may readily be parallelised. Loop unrolling replaces the innermost loops of an algorithm with multiple sequential copies of the operations within the loop construct. This allows longer pipelines to be constructed at the expense of additional hardware. With many image processing operations, these innermost loops are inherently parallel (for example weighting the pixels within a linear filter) and it is only the sequential nature of a serial processor that requires a loop. Strip mining is related to loop unrolling in that it creates multiple parallel copies of the loop body. The difference is that each copy operates on a different section of the data. The computation is accelerated by partitioning the input data over different hardware processors.

A less common transformation is strip rolling and multiplexing. Sometimes, after strip mining, only one of the parallel processing units is active in any clock cycle. For example, using stream processing to find the bounding boxes of objects requires operating on multiple objects in parallel. However, a pixel can only belong to one object, so in any clock cycle, only one bounding box processor will be operating. In this case, strip rolling replaces the multiple processing units with a single processor that is multiplexed between the different instances.

In some algorithms, it is possible to rearrange the order of the operations to simplify the processing complexity, or even eliminate some steps. A classic example of this is greyscale morphological filtering followed by thresholding. This sequence of operations is equivalent to thresholding first, followed by binary morphological filtering. The algorithms for binary filtering are significantly simpler than those for greyscale filtering, resulting in considerable hardware savings and often a reduction in latency.

The data can sometimes be coded to reduce the volume, and hence the processing speed. Run length coding is a good example. After the data has been run length coded, whole runs of pixels can be processed as a group, rather than as individual pixels. For example, Appiah *et al.* (2008) used run length coding to accelerate the second pass through the image for connected components labelling.

Gains can often be made by substituting one or more image processing operations with other, functionally similar, operations. Substitutions may involve approximating operations with similar but computationally less expensive operations. Common examples are replacing the L_2 norm in the Sobel filter with the computationally simpler L_1 or L_∞ norm (Abdou and Pratt, 1979), or modifying the coefficients used for colour space conversion to be powers of two (Sen Gupta *et al.*, 2004). Note that substituting an operation may change the results of the algorithm. The resulting algorithm will be functionally similar but not necessarily equivalent. In many applications, this does not matter, and the substitution may enable other transformations to be applied to improve the computational efficiency.

The second stage is to design the computational architecture that is implied by the algorithm architecture. This may require further transformation of the algorithm to result in a resource efficient architecture that can effectively exploit the parallelism available through a hardware implementation. Some of the techniques that may be used for this will be described in more detail in the next chapter. Operation specific caching of data ensures that data which is used multiple times will be available when needed. Low level pipelining and retiming can be used to reduce the combinatorial delay and increase the clock speed. It is also necessary to select appropriate hardware structures that correspond with data structures used in the software algorithm.

The design process can be iterative with the algorithm and architecture mappings being updated as different design decisions are made. However, with a thorough design at this stage, the final step of mapping the algorithm onto the architecture to obtain the resultant implementation is straightforward.

4.3.3 Partitioning between Hardware and Software

Low level image processing operations that can readily exploit parallelism are ideal for hardware implementation. Such operations are relatively slow when implemented in software simply because of the volume of data that needs to be processed.

However, not all algorithms map well to a hardware implementation. Tasks that have dynamically variable length loops can be clumsy to implement in hardware. Those with complex control sequences (Compton and Hauck, 2002) are more efficiently implemented in software. Such tasks are characterised by large amounts of complex code that is primarily sequential in nature. If implemented in directly in hardware, this would require significant resources because each operation in the sequence would need to be implemented with a separate hardware block. If there is little scope for exploiting parallelism, much of the hardware will be sitting idle much of the time. The same applies for functions that are only called occasionally, for example once or twice per frame (or less). These tasks or operations can be implemented more easily and more efficiently in software.

Two broad classes of tasks or operations that are best implemented in software are high level image processing operations (for example object tracking; Arias-Estrada and Rodríguez-Palacios, 2002) and managing complex communications protocols. High level image processing operations are characterised by a lower volume of data and more complex control patterns. Many provide limited opportunities for exploiting parallelism and often the computation path is dependent on the data being processed. Similarly, communications such as TCP/IP require a complex protocol stack that can be unwieldy to implement in hardware. While it is possible to implement these tasks in hardware (see, for example Dollas *et al.*, 2005, for a TCP/IP implementation), it can be more efficient in terms of resource utilisation to implement them in software. Software also has the advantage that it is generally easier to programme.

The approach taken to partitioning the application between hardware and software will depend significantly on the system level architecture and, in particular, the level of coupling between the software processor and programmable logic. The software processor may consist of a separate CPU or be integrated on the FPGA as a hardware core (such as the PowerPC core within the Xilinx Virtex Pro) or as a soft core implemented using the FPGAs programmable logic. There is also a wide range of standard processor cores available that have FPGA implementations, including common microcontrollers (8051, 68HC11, PIC) and digital signal processors (320C25). One advantage of using a standard processor is that all of the application development tools for that processor can be used. In particular, using the integrated high level language development and emulation tools for the processor can speed code development and debugging of the software components.

Therefore, the choice is not just whether to implement the application in either software or hardware. There is a spectrum of partitioning of the application between the two, ranging between full software and full hardware implementations as illustrated in Figure 4.5.

At the instruction level of granularity, the application can be largely implemented in software, with the FPGA used to implement custom instructions. This allows the programme to 'call' hardware acceleration blocks (Arnold and Corporaal, 2001). For image processing, it is impractical to implement the whole algorithm this way. However, this approach can be used to augment the processor for the software component of the application where groups of instructions are clumsy or inefficient when implemented purely in software.

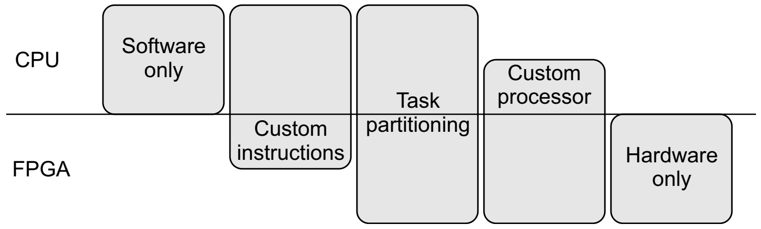


Figure 4.5 Spectrum of hardware and software mix. Note that the CPU can be either external or integrated within the FPGA.

The partitioning can also be at the task level of granularity, where whole tasks are assigned either to a software or hardware processor. The hardware and software tasks operate relatively independently of each other, only communicating when necessary. This would allow, for example, the image capture and low level image processing operations to be implemented in hardware, with the results being passed to high level image processing operations implemented in software.

If the software tasks are of low complexity, then an alternative to using a full CPU is to design a simplified serial processor. This can be either a lightweight soft core (such as the picoBlaze) or a completely custom processor. One limitation to using a custom processor is that compilers and other development tools will not be available and the software for the processor must generally be written in assembly language. At the extreme end, the sequential components of the algorithm could be implemented using a finite state machine. Quite sophisticated, memory-based, finite state machines can be implemented with a small resource footprint (Sklyarov, 2002; Sklyarov and Skliarova, 2008), effectively making them custom processors.

Regardless of the level of partitioning, it is essential to clearly define the interface and communication mechanisms between the hardware and software components. In particular, it is necessary to design the synchronisation and data exchange mechanisms to facilitate the smooth flow of data. In the final design, it is also important to take into account the communication overhead between the hardware and software portions of the design.

One advantage of using a standard processor is the availability of standard operating systems such as Linux. Using an operating system such as embedded Linux has an advantage that all of the features of the standard operating system are available, subject to the limitations of the memory available within the system (Williams, 2009). It also provides a familiar development environment for the software components, including the ability to develop and test the software on one platform before transferring it to the FPGA. One feature of particular interest is that Linux has a full TCP/IP stack enabling the complete system to be networked relatively easily. This can significantly extend the functionality of the system, even if the software processor is not used directly for the application. Having a TCP/IP interface enables:

- an efficient means of communicating images and other data between devices in a distributed system;
- a Web interface for control and setting of algorithm parameters, and for viewing the results of processing;
- remote debugging through telnet, and even mounting the file system across the network;
- the ability to remotely update the configuration.

A minor disadvantage of using an embedded Linux is that even a minimal system will require off-chip RAM. A minimal Linux kernel will require approximately 4 MB of additional flash memory and 16 MB of DDR system memory. The system memory should be solely for the use of the embedded processor and be separate from any memory directly accessed from the user logic on the FPGA. As the number of support applications (within Linux) is increased, the memory requirements (both flash and RAM) will increase accordingly. The boot process of the system is as follows (Figure 4.6) (Gregori, 2009):

1. The FPGA configuration is loaded from flash memory. This configures the FPGA, including the processor on the FPGA.
2. When the system starts running, the processor executes the boot loader from the flash memory.
3. The boot loader decompresses both the Linux kernel and the root file system from flash, and copies them into RAM.
4. Using the RAM image, the processor starts execution of the Linux kernel and associated services.

When interfacing between the hardware and software running under embedded Linux, it is best to make the interface through a Linux device driver (Williams, 2009). While it is possible to interface directly from

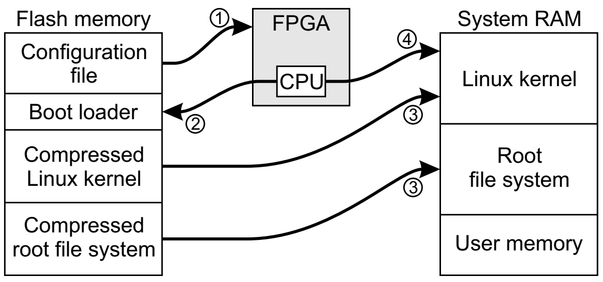


Figure 4.6 Embedded Linux boot process.

the software to the hardware, using a device driver will make access to the hardware more portable, and make the interface easier to maintain and manage. The Linux kernel effectively provides an API and will provide locking and serialisation on the device.

One limitation of using a multitasking operating system is the variable latency of the software components, especially if virtual memory is used. Care must be taken when using such systems in a hard real-time context, especially where software latency is critical. In other cases an embedded Linux can simplify the development and debugging, especially of stand-alone systems.

The key requirement of architecture selection is choosing the most appropriate architecture for the algorithm, both at the computational and system levels. In an embedded application, most of the performance gains come through exploiting parallelism. Power considerations often require maintaining a low clock speed. Careful design of the image processing algorithm can often allow the low level vision to operate at the input or output video rate, which is often up to two orders of magnitude slower than current high end serial processors.

Image capture and display almost always operate on streamed data. Therefore, as much processing as possible should be performed on-the-fly either as the image is streamed in, or streamed out. This will minimise the number of memory accesses required and the sequential nature of the accesses will simplify memory system design.

4.4 System Implementation

The final stage within the design process is system implementation. This is where the image processing algorithm is mapped onto the chosen hardware resources as defined by the architecture. The steps here differ significantly from software-based design, where implementation primarily consists of coding the algorithm. An FPGA-based implementation requires designing the specific hardware that will perform the required image processing operations.

Not all of the operations in the algorithm will necessarily map well to hardware. To gain maximum benefit of the parallelism offered by the FPGA, all of the individual operations must be architecturally compatible. If the image processing algorithm has not been developed with the computational architecture in mind, then it may be necessary to modify the algorithm so that the operations better match the selected architecture. This may require the selection of alternative operations where necessary.

The application level algorithm is created at a relatively high level of abstraction, in terms of individual image processing operations. Some of these operations can be quite complex, and all have their own operation level algorithms at a lower level of abstraction. Many of these operations can be implemented using several possible algorithms. The operations within the software-based image processing environment will have been optimised for serial implementation within that environment. Simply porting that algorithm onto the FPGA will generally give relatively poor performance, because it will still be predominantly a serial algorithm. Sometimes the serial algorithm may have a relatively simple transformation to make it suitable for parallel hardware. More often, however, the underlying algorithm may need to be completely redesigned to make it more suitable for FPGA implementation, as described in the previous section. In this case, it may be better to first re-implement the operation using the new algorithm within the image processing environment, where it can be tested more easily, rather than attempt to directly develop and test it on the FPGA.

Since each image processing operation is considerably simpler than the complete application, this can make the implementation easier to test and debug. Standard test bench techniques can be used to exercise the operation through simulation within the FPGA development environment. For individual operations, most of these tests require a much smaller range of possible inputs than the complete application, so most operations can be tested with relatively small data sets rather than with complete images. The FPGA development platform is also implemented on a standard serial computer, so the simulation of a parallel architecture on a serial computer is usually quite time consuming. This is especially the case for clock-accurate simulations of complex circuits with large data sets (images). Therefore, it is better to test as much of each operation as possible using much smaller data sets.

If the algorithm for an operation is complex, or the mapping from software to hardware is not straightforward, incremental implementation techniques are recommended. For this, each stage or block of the algorithm is simulated as it is completed, rather than leaving testing of the complete operation to the end. This approach generally enables errors in the implementation to be found earlier and corrected more easily.

Another task that is required for FPGA-based designs is to develop the interface logic required by peripherals attached to the FPGA. Intellectual property cores may be available for some devices, such as common memory chips, USB ports and so on. For others, these will need to be developed as part of the implementation process. Example interfaces for some specific devices are developed in Chapter 12. These interface blocks may be thought of as device drivers within a conventional operating system. Even if the design uses an embedded CPU running a conventional operating system such as Linux, it will still be necessary to develop the hardware interface between the processor and the external device.

Despite efforts to make programming FPGAs more accessible and more like software engineering (Alston and Madahar, 2002), efficient FPGA programming is still very difficult. It is important to realise that programming FPGAs is hardware design, not software design. Although the languages may look like software, the compilers use the language statements to create the corresponding hardware. Rather than executing the statements on a common, shared, ALU, each statement builds independent, parallel, hardware. Concurrency and parallelism are implicit in the hardware that is built and the compilers have to build additional control circuitry to make them execute sequential algorithms (Page and Luk, 1991). Consequently, sequential algorithms ported from software will by default run sequentially, usually at a significantly lower clock speed than high end CPUs. Some compilers can automatically make minor changes to the algorithm to exploit some of the parallelism available, and this is often sufficient to compensate for the lower clock speed. However, the algorithm is still largely sequential unless it has been specifically adapted for parallel execution. The software-like languages are in reality hardware description languages, and efficient design requires a hardware mindset.

4.4.1 Mapping to FPGA Resources

While it is not necessary to perform the design at the transistor or gate level, it is important to be aware of how the resources of the FPGA are used to implement the design, whether explicitly or implicitly.

Both arithmetic and logic functions are implemented as combinatorial hardware using the configurable logic blocks on the FPGA. Depending on the size of the lookup tables, each logic cell can implement any arbitrary logic function of between three and seven binary inputs, with or without a register on the output. Functions with more inputs are implemented by cascading two or more logic cells.

A simple logic function, such as AND or OR between two N-bit operands may be implemented using N LUTs, one for each bit of the operands. For more complex logic operations, the number of LUTs required will depend primarily on the number of inputs rather than the actual complexity of the logic. In many situations, logical inversion may be obtained for free, simply by modifying the contents of the corresponding LUTs.

Addition and subtraction also map onto combinatorial logic. The carry propagation may be implemented using circuits for ripple carry, or more advanced look-ahead carry or carry select designs (Hauck *et al.*, 2000). Most FPGAs implement dedicated carry chain logic that can be used to speed up carry propagation by reducing routing delays and avoiding the need for additional hardware to implement the carry logic. Counters are simply adders (usually adding a constant) combined with a register to hold the current count value.

Relational operators compare the relative values of two numbers. The simplest is testing for equality or inequality, which requires a bit-wise comparison using exclusive NOR, and combining all of the outputs with a single gate to test that all bits in the two numbers are equal. A comparison operation may be implemented efficiently using a subtraction and testing the sign bit. This has longer latency than testing for equality because of the carry propagation. Therefore, if the algorithm involves any loops, tests should be made for equality rather than comparison, if possible, to give the smallest and fastest circuit.

Multiplication can be performed using a combination of shift and addition. If time is not important, this can be implemented serially using not much more logic than an adder. Unrolling the loop results in an array of adders that combines the partial products in parallel. Although, in principle, the length of the carry propagation may be reduced using a Wallace (Wallace, 1964) or Dadda (Dadda, 1965) adder tree, in practise this gives a larger and slower implementation than using the dedicated carry logic of the FPGA (Rajagopalan and Sutton, 2001). Additional logic is required to perform the carry save additions, and the structure is not as regular as a simple series of adders resulting in increased routing delays. If necessary, the parallel multipliers can be pipelined to increase the throughput. Most current devices (especially those targeted at DSP applications) have dedicated hardware multiplication circuits. These provide faster and smaller multipliers because they do not suffer from FPGA routing delays between stages. However, the multipliers are available only in fixed bit widths. A single multiplier may readily be used for smaller sized words, but for larger word widths either several multipliers must be combined or a narrower multiplier supplemented with programmable fabric logic to make up the width.

In principle, division is only slightly more complex than multiplication, with a combination of shift and subtraction to generate each bit of the quotient. With the standard long division algorithm (for unsigned divisor and dividend), if the result of the test subtraction is positive the quotient bit is 1 and the result of the test subtraction retained. However, if the test subtraction result is negative then the corresponding quotient bit is 0 and the test subtraction discarded, restoring the partial remainder to a positive value. This requires that the carry from each subtraction must completely propagate through the partial remainder before the sign bit can be determined. The algorithm speed may be improved (Bailey, 2006) by using non-restoring division, which allows the partial remainder to go negative. With this scheme, the divisor is added rather than subtracted for negative partial remainders. The major limitation to the speed of division is the propagation delay of the carry rippling through the whole partial remainder at each iteration. An obvious solution is to reduce the number of iterations by determining more quotient bits per iteration (using a higher radix). Another approach is to reduce the propagation delay by reducing the length of the critical path. The SRT algorithm (named after Sweeney, Robertson (1958) and Tocher (1958)) achieves this by introducing redundancy in the quotient digit set. This gives an overlap in selecting between adjacent digits, which allows a quotient digit to be determined from an approximation of the partial remainder (hence does not need to wait for the carry to propagate fully). Two disadvantages of the SRT method are that the divisor must be normalised (so that the MSB is 1) and the quotient digits must be converted to standard binary, both of which require additional logic and time. All of these algorithms may be implemented sequentially to reduce hardware requirements at the expense of throughput, or can be pipelined to increase the throughput. There are also many other advanced algorithms (Tatas *et al.*, 2002;

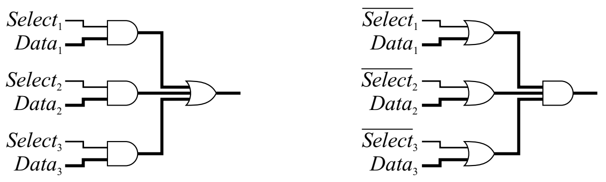


Figure 4.7 Multiplexer circuits for data selection. Left: using positive logic; right: using negative logic.

Trummer, 2005). An alternative approach if high speed multipliers are available is to take the reciprocal of the divisor (using lookup tables or fast iterative methods) and then multiply.

In all of the operations described so far, if one of the operators is always a constant, the hardware can be simplified. This can often result in a reduction in both the logic required and the propagation delay of the circuits.

Another logic circuit that is commonly a part of FPGA designs is the multiplexer. These are required when writing to a register or memory from multiple hardware blocks to select the data source being written. A multiplexer is also required on the address lines of memory to select the address source when accessing memory from multiple points in the hardware. Typically, two input bits are required for each bit of the source word (Figure 4.7); one bit for the data value, and one bit to select or gate that source to the output. When many sources are used, the multiplexers may require a significant fraction of the FPGA resources available. There are two types of multiplexer commonly used, corresponding to the use of positive and negative logic (the differences are shown in Figure 4.7). With positive logic, the default output is 0 and a logic 1 is used to gate (or select) the inputs

$$Out = \bigvee_i (Data_i \wedge Select_i) \tag{4.1}$$

With negative logic, the levels of the input and output are inverted making the default output a 1. De Morgan's theorem is used to convert the expression into a product of sums form:

$$\begin{aligned}\overline{Out} &= \bigvee_i \left(\overline{Data_i} \wedge Select_i\right)\\ Out &= \bigwedge_i \left(Data_i \vee \overline{Select_i}\right)\end{aligned} \tag{4.2}$$

This would normally have little effect on the operation of the circuit (provided the correct polarity select signals are generated) except when multiple sources are enabled simultaneously. In this error condition, the logic levels from the two sources are combined; for positive logic the output is the OR of the enabled inputs, whereas for negative logic it is the AND of the inputs.

Modern FPGAs also provide memory resources at a range of granularities. At the lowest level, registers may be built up of from the flip-flops associated with the logic blocks. In most devices, each logic cell has one flip-flop available on its output, so an N-bit register would require N logic cells. The outputs of these flip-flops are always available for use by other logic, so all of the registers in an array would be available at every clock cycle. An array constructed of registers, if accessed using a variable index, would require one or more multiplexers on the output to select the data from the required register. Similarly, writing to an array may also require a multiplexer on the input of each register in the array.

The LUTs within the logic cells of some FPGAs can also be configured as fabric memory, providing 16, 32, or 64 memory locations depending on the available size of the LUTs. These are true memories, with the address decoding and data multiplexing performed using dedicated logic within the logic cell. Note that only one bit is accessible for either reading or writing per clock cycle. Some FPGAs allow the inputs to an adjacent logic cell to be configured as a second port, enabling two accesses per clock cycle. Multiple logic cells can be combined in parallel to create arbitrary width memories. In doing so, each parallel memory will share the same address inputs. Fabric RAM is best suited for small, shallow memories because any increase in the depth requires the use of external decoding and multiplexing. Examples where fabric RAM would be used include storing coefficients or parameters that do not have to be accessed simultaneously, and short FIFO buffers.

Modern FPGAs also have available larger blocks of RAM. These generally have flexible word widths from 1 to over 64 bits wide. Most such RAM blocks are dual-port. The size makes them well suited for row buffers for image processing, for larger size FIFO buffers or for distributed data caching. The RAM can also be used as instruction and data memory for internal CPUs (both hard and soft core).

Some of the larger FPGAs have sufficient memory blocks to hold whole images on chip. However, image frame buffers are more likely to be held off-chip. The number of parallel external RAMs (and ROMs) and their size are limited only by the I/O resources available. Off-chip RAM can also be used for other large blocks of intermediate data storage or be used by an internal CPU to hold data and instructions for larger programmes.

A software processor, if used, can be implemented either on the FPGA or as a separate chip. Large modern FPGAs have sufficient resources on chip not to need an external processor. However, when using an operating system such as an embedded Linux, an external memory is essential.

4.4.2 Algorithm Mapping Issues

As mentioned earlier in this chapter, mapping is not simply a matter of porting the set of operation level algorithms onto the FPGA, unless they were developed with hardware in mind. The key thing is to map what the software is doing by adapting the algorithm to the hardware.

Again, it is important to keep in mind the different levels of abstraction. At the higher level, the application level algorithm should be thoroughly tested on a software-based platform before mapping the operations to hardware. In some situations, it may be necessary to substitute different image processing operations to maintain a uniform computational architecture if possible. Generally, though, the application level algorithm should not change much through the mapping process.

Mapping the individual image processing operations is a different story. The operation level algorithm may change significantly to exploit parallelism. For some operations, the transformation from a serial to parallel algorithm is reasonably straightforward, while for others a completely new approach, and algorithm, may need to be developed.

During the mapping process, it can be worthwhile to consider not just the operations on their own, but also as a sequence. It may be possible to simplify the processing by combining adjacent operations and developing an algorithm for the composite operation. An example of this is illustrated in Section 11.4 with connected components analysis. In other instances, it may be possible to simplify the processing by splitting a single operation into a sequence of simpler operations. An example of this is splitting a separable filter into two simpler filters. In some situations, swapping the order of operations can reduce the hardware requirements. For example, following a greyscale morphological filter with thresholding requires a more complex filter than its equivalent of first performing the thresholding and then using a binary morphological filter.

One of the major issues facing the algorithm mapping process for embedded, real-time image processing is meeting the various constraints. There are three main constraints facing the system designer: timing constraints, memory bandwidth constraints and resource constraints.

Any real-time application will have timing constraints. These are of two types: throughput and latency. Stream processing has a throughput constraint, usually of processing one pixel every clock cycle. If this rate is not maintained, then data streamed from the camera may be lost, or data streamed to the display may be missing. Since the processing performed per pixel will usually take significantly longer than one clock cycle, it is necessary to use pipelining to maintain the throughput. The other timing constraint is latency. This is the maximum permissible time from when the image is captured until the results are output or an action is performed. Latency is particularly important for real-time control applications where image processing is used to provide feedback used to control the state. Delays in the feedback of a control system (caused in this case by the latency of the processing) will limit the response time of the closed loop system and can make control more difficult. In particular, excessive delay can compromise the stability of the system.

Closely related to the timing constraint is the memory bandwidth constraint. Each memory port may only be accessed once per clock cycle. On-chip memory is usually less of a problem because such memory consists of a number of relatively small blocks, each of which can be accessed independently. Off-chip memory, on the other hand, tends to be in larger monolithic blocks. Operations where the computation order does not correspond to the raster order usually require the whole frame to be buffered. The large size of the image buffer generally restricts the frame buffer to be in off-chip memory. Consequently, it can be difficult to read more than one pixel per clock cycle. This can be further complicated by pipelined memory architectures, where the address needs to be provided several clock cycles before the data is available. Operations where the next memory location depends on the results of processing at the current location (for example chain coding) exacerbate this problem. For the memory bandwidth not to seriously limit the processing speed, it is often necessary to transfer data from off-chip memory into multiple blocks of on-chip memory where the bandwidth constraints are more relaxed.

The third area of constraint relates to the resources available on the FPGA. The cost of an FPGA generally increases with increasing size (in terms of the number of logic blocks, memory etc.). When targeting a commercial application, to minimise the product cost it is desirable to use an FPGA that is only as large as necessary. However, even the largest FPGA has finite resources. It is therefore important to make efficient use of the available resources. Using fewer resources can also improve the timing in two ways. The first, and more obvious, is that an algorithm that uses fewer resources will generally have a shorter delay simply because of the shallower logic depth. The second is that a more regular design will be easier to route on the FPGA, and will have a shorter routing delay. As the design approaches the capacity of the FPGA, it becomes increasingly harder to route efficiently and the maximum clock speed of the design decreases.

A second issue relating to resources is contention between shared resources. A parallel implementation will have multiple parallel blocks working concurrently. In particular, only one process can access memory or write to a register in any clock cycle. Concurrent access to shared resources must either be scheduled to avoid conflicts, or additional arbitration circuitry must be designed to restrict access to a single process. Arbitration has the side effect of delaying some processes, making it more difficult to determine the exact timing of tasks.

Techniques for addressing timing, bandwidth and resource constraints are discussed in more detail in the next chapter.

4.4.3 Design Flow

A more detailed sequence of steps for the design process that we follow in our research group is shown in Figure 4.8. The image processing algorithm is developed either using VIPS (Bailey and Hodgson, 1988) or MATLAB® (Gonzalez and Woods, 2004; Solomon and Breckon, 2011). Both are rich algorithm development environments. VIPS was designed specifically for image processing; MATLAB® is more general, but with the image processing toolbox is also targeted for imaging applications. Both are extensible, operations for VIPS are written in C++, whereas for MATLAB® extensions may be written in C or as MATLAB® scripts (m-files).

At Massey University Handel-C has been used for the hardware description language because it describes hardware at a higher level of abstraction than Verilog or VHDL. It also allows hardware to be described at a low level where necessary. Where possible, Handel-C is also used for any device drivers. VHDL is only used at the very lowest levels, where Handel-C does not provide sufficient control over the detailed implementation.

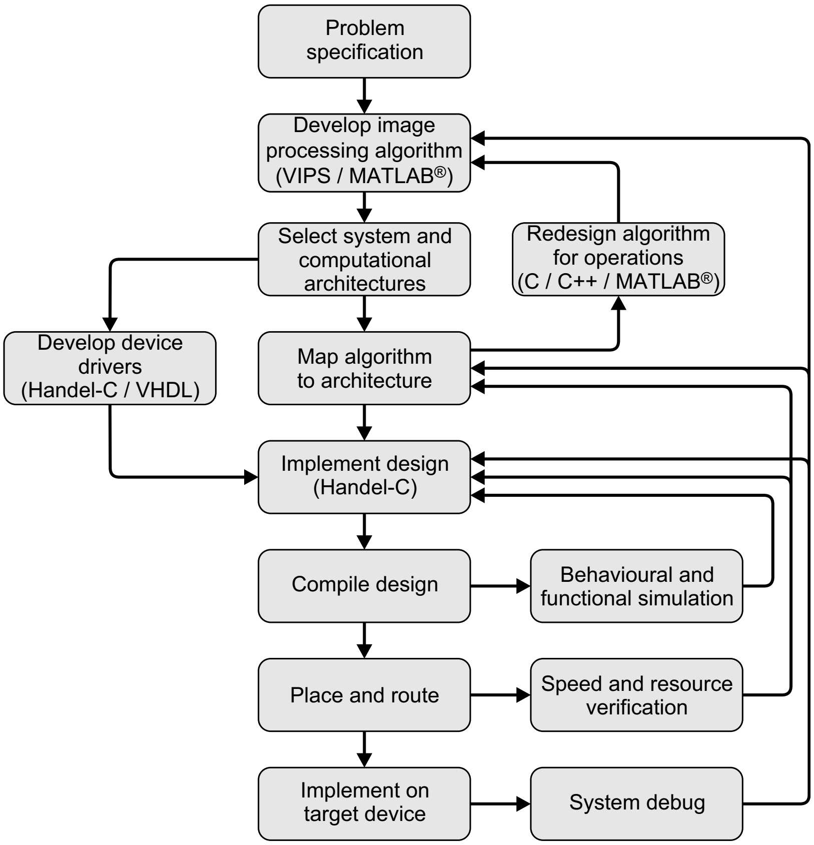

Problem specification
Develop image processing algorithm (VIPS / MATLAB®)
Select system and computational architectures
Redesign algorithm for operations (C / C++ / MATLAB®)
Develop device drivers (Handel-C / VHDL)
Map algorithm to architecture
Implement design (Handel-C)
Compile design
Behavioural and functional simulation
Place and route
Speed and resource verification
Implement on target device
System debug

Figure 4.8 Design flow for FPGA-based implementation of image processing algorithms.

Incremental development techniques are used. As each operation is mapped to Handel-C, the simulator within the development environment is used to verify that the operation functions correctly. If each operation performs as designed, then the complete application level algorithm will function as developed and tested within the image processing environment.

FPGA vendor tools are used to place and route the design onto the FPGA resources. The output from this process will indicate whether there are sufficient resources available on the target FPGA. It also provides timing information based on the actual resources used. Feedback from this process is used to modify the algorithms or implementation where necessary to meet timing constraints.

The final step is to download the configuration file onto the target device for system testing. As part of the debugging process, sections may require re-implementation, remapping or, in the worst case, redevelopment.

4.5 Designing for Tuning and Debugging

4.5.1 Algorithm Tuning

Many application level algorithms require tuning to function correctly. Most algorithms have threshold levels or other parameters that must be optimised to obtain the best performance.

If these parameters are hard wired into the implementation, then whenever a parameter is changed, the whole algorithm needs to be recompiled and remapped to hardware – a process that is very time consuming. Therefore, it is best to store these parameters in registers, where they may be readily modified. While it is technically possible to modify the register values directly in the bitstream before downloading onto the FPGA, for example using JBits (Guccione *et al.*, 1999) or RapidSmith (Lavin *et al.*, 2010) with Xilinx FPGAs, Xilinx is currently the only manufacturer to provide the tools for doing this (Graham *et al.*, 2001). In some ways, tuning an algorithm is a little like tuning a car. Recompiling the application is a little like rebuilding the complete engine each time an adjustment is required. Modifying the bitstream is like turning the engine off, making the adjustments, and then restarting the engine with each modification. Instead, it is much more convenient to tune the engine while it is running. Similarly, the best time to modify many of an algorithm's parameter values is while the algorithm is running.

Changing parameters dynamically requires building the hardware to update the parameters. It also requires some form of user interface that enables the developer to adjust or provide new values. When running in a hosted configuration, the host system can provide the user interface and write the control parameters to registers on the FPGA. However, with a stand-alone configuration, the user interface must also be built on the FPGA. This can be either in hardware or on an embedded processor. At the minimum, it must provide a mechanism for selecting and modifying the parameters. With many applications, though, it is usually necessary to provide additional information on the context of the parameters being adjusted. This often involves displaying the image or other data structures at intermediate stages within the algorithm, as illustrated in Figure 4.9 for a simple colour tracking application.

In this application, the complete algorithm can be implemented using stream processing, without the need of a frame buffer. However, to set the threshold levels used to detect the coloured targets, it may be

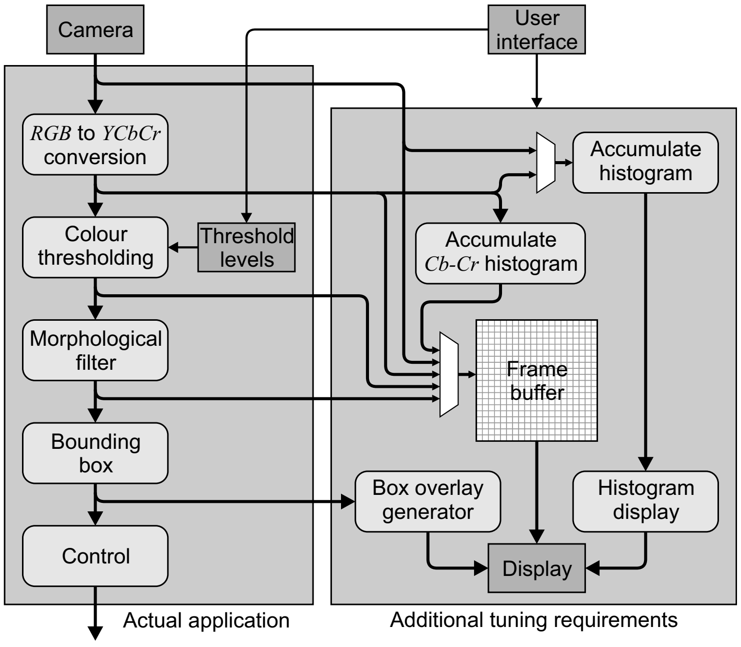


Figure 4.9 Algorithm tuning may require considerable additional hardware to provide appropriate feedback or interaction with the user.

necessary to display histograms of red, green and blue components of the image, or of the *YCbCr* components used for colour thresholding. Since the primary thresholding will be performed in the *Cb*−*Cr* plane, it is also useful to accumulate and display a two-dimensional *Cb*−*Cr* histogram of the image. One way of setting the threshold levels is to click on the coloured objects of interest in the input image and use these to initialise or adjust the threshold levels. It is also useful to see the output image after thresholding to see the effects of adjusting the levels, particularly on the boundaries of coloured regions. Similarly, it is also useful to see the effects of the morphological filter in cleaning up the misclassification noise. Finally, it may be useful to include a box generator to overlay the bounding boxes of the detected regions over the top of the original image (or indeed after any of the other processing stages).

Although this example probably provides more contextual information than is really needed, it does illustrate that the additional logic and resources required to support tuning may actually be more extensive than those of the original application. One way to manage these extra requirements is to use two separate FPGA configurations – one for tuning and the other for running the application. The tuning configuration does not need all of the application logic, for example the final control block in Figure 4.9 may be omitted. The tuning configuration can then store the tuning parameters in flash memory (or other non-volatile memory) where they can be retrieved by the actual application configuration.

4.5.2 System Debugging

In many ways, the requirements for debugging the algorithm and system are similar to tuning. In particular, it may be important to see the results of processing after each of the operations to verify that each operation is functioning correctly. This may require the stand-alone configuration to support video output even if the application does not explicitly require it.

System failure can be the result of two main causes. The first is algorithm failure and the second is a result of not meeting timing constraints.

Since it is much harder to debug a failing algorithm in hardware than it is in software, it is important to ensure that the algorithm is working correctly in software before mapping it onto the FPGA. It is essential that any modifications made to the operation level algorithms during the mapping process be tested in software to ensure that the changes do not affect the correct functioning. However, while it is possible to test the individual operations to check that they function correctly, it is not possible to exhaustively test the complete application level algorithm. In many real-time video processing applications, every image that is processed is different. Failures, therefore, should not be a result of the algorithm not working at all, but be intermittent failures resulting from the algorithm not being completely robust.

Unfortunately, the intermittent nature of such algorithm failures makes them hard to detect. Many, if not most, of the images should be processed correctly. Therefore, to investigate the failures, it is necessary to capture the images that cause the algorithm to fail. One way of accomplishing this is to store each of the images before it is processed. Then when a failure occurs during testing, the input image that caused the failure will be available for testing. Such testing is actually easier to perform in software, either in the image processing development environment or through simulation of the hardware in the FPGA development environment.

Algorithm debugging in hardware is much more complex. Additional user controls may be required to freeze the input image to enable closer examination of what is happening for a particular input. It may be important to view the image after any of the operations, not just the output. This may require stepping through the algorithm one operation at a time, not only in the forward direction, but also stepping backwards through the algorithm to locate the operation that is causing the algorithm to fail. The additional hardware to support these controls is shown in Figure 4.10.

Note that if stream processing is used, it is not necessary to have a frame buffer on the output; the output is generated on-the-fly and routed to the display driver. However, non-streamed operations or non-image based information may require a frame buffer to display the results.

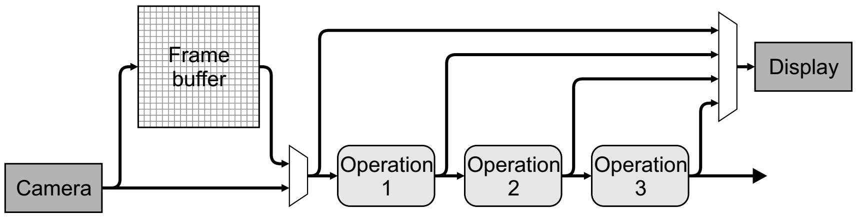


Figure 4.10 Instrumenting an image processing algorithm for debugging. The frame buffer enables the input image to be frozen and fed through the algorithm. The results of any of the steps may be routed to the display (stream processing is assumed here).

At a lower level, debugging the internals of an operation is more difficult. Again, it is emphasised that such debugging should be performed in software or through simulation if possible. The advantage of software debugging is that in software each clock cycle does relatively little compared with hardware where a large number of operations are performed in parallel, and the system state can change considerably (Graham, 2001).

Simulation is only as reliable as the simulation model used. The models can be validated by directly comparing the results of simulation with hardware execution over the full range of situations that may be encountered (Graham, 2001). This is particularly useful for identifying problems (both functional and timing) associated with device drivers or other interfaces.

If the design must be debugged in hardware, there are a number of ways in which this may be performed (Graham *et al.*, 2001). Perhaps the simplest approach is to route key signals to unused I/O ports to make them observable from outside the FPGA. An external oscilloscope or logic analyser is then required to monitor the signals. This concept may be extended further by embedding a logic analyser within the design. Both Xilinx (ChipScope; Xilinx, 2008a) and Altera (SignalTap; Altera, 2008e) provide cores that use memory within the FPGA to record internal signals, and later read these recorded signals out to an external companion software package. It is also possible to build dedicated hardware to perform a similar function, recording signals of interest within a block RAM and using the configuration readback mechanism to obtain the data. One limitation of modifying the circuit to include signal probes and additional signal recording hardware is that that these can affect how the connections are routed on the FPGA. These affect the timing and can even change the critical paths, possibly masking or creating subtle timing problems that may occur when the modifications are either present or absent (Graham, 2001; Tombs *et al.*, 2004).

Configuration readback can be used to sample the complete state of a design at one instant. The difficulty with this approach is to associate the logical design with the hardware state of the mapped resources on the FPGA. Signal names, or even the circuit, can be altered for optimisation or other reasons, and with fabric RAM the address pins may be reordered to improve design routability (Graham, 2001). In spite of these limitations, configuration readback provides the widest observability of the design (Graham, 2001). It is possible to add only a little logic to stop and restart execution by gating the clock (Tombs *et al.*, 2004). These features enable hardware single stepping and, with a little extra logic to detect key events, can effectively provide hardware breakpoints. A key limitation of readback-based methods is that they require the clock to be stopped. This makes them impractical for timing verification and, in particular, for testing of real-time systems. For example, data streamed from a camera or output to a display must be routed via a frame buffer to allow the execution to be stopped and restarted without losing external context and synchronisation.

One technique that enables internal timing to be verified while repeatedly stopping the design to enable configuration readback has been described by Vuillemin *et al.* (1996). By double stepping the design (advancing the clock two cycles at a time), the second cycle can run at full speed, with normal timing.

The system can then be stopped after the second cycle and the configuration read back out to detect timing failures. To verify the timing of the complete design, two execution cycles are required, one for the even cycles and one for the odd cycles (Graham, 2001).

In any complex design, it is important that provision for tuning and debugging be considered as part of the design process, rather than something that is added to the implementation at the end.

5

Mapping Techniques

The previous chapter described the process of designing an embedded image processing application for implementation on an FPGA. Part of the implementation process involves mapping the algorithm onto the resources of the FPGA. Three types of constraints to this mapping process were identified as timing (limited processing time), memory bandwidth (limited access to data) and resource (limited system resources) constraints (Gribbon *et al.*, 2005; 2006). In this chapter, a selection of techniques for overcoming or alleviating these constraints is described in more detail.

5.1 Timing Constraints

The data rate requirements of real-time applications impose a strict timing constraint. At video rates, all required processing for each pixel must be performed at the pixel clock rate (or faster). This generally requires low level pipelining to meet this constraint.

When considering stream processing, it is convenient to distinguish between processes that are source driven and those that are sink driven, because each has slightly different requirements on the processing structure (Johnston *et al.*, 2008). With a source-driven process, the timing is constrained primarily by the rate at which data is produced by the source. The processing hardware has no direct control over the data arrival. The system needs to respond to the data and control events as they arrive. For video data, the events of concern are new pixels, newline and newframe events. Based on the event, different parts of the algorithm need to run; for example, when a new frame is received, registers and tables may need to be reset. The processors in source-driven systems are usually triggered by external events.

Conversely, a sink-driven process is controlled primarily by the rate at which data is consumed. An example of this is displaying images and other graphics on a screen. In this case, processors must be scheduled to provide the correct control signals and data to drive the screen. Processors in a sink-driven system are often triggered to run at specified times, using internal events. Another form of timing constraint occurs when asynchronous processes need to synchronise to exchange data.

5.1.1 Low Level Pipelining

In all non-trivial applications, the propagation delay of the logic required to implement all the image processing operations exceeds the pixel clock rate. Pipelining splits the logic into smaller blocks, spread over multiple clock cycles, reducing the propagation delay in any one clock cycle. This allows the design to be clocked faster in order to meet timing constraints.

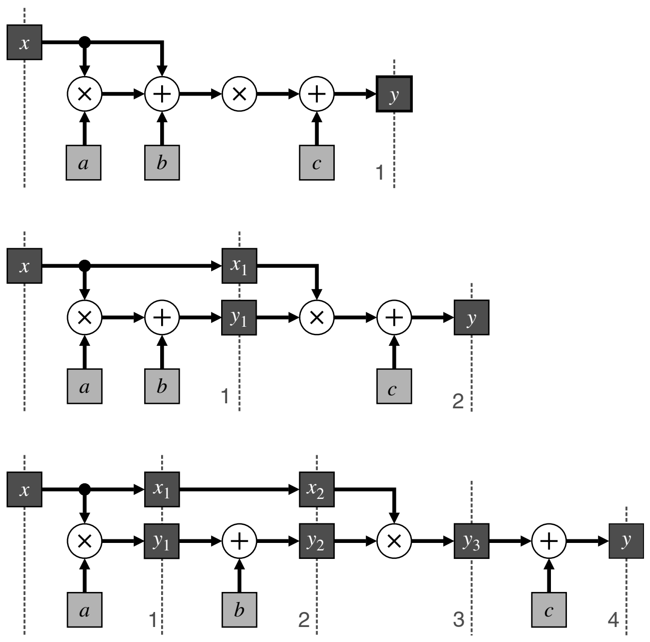


Figure 5.1 A simple pipelining example. Top: performing the calculation in one clock cycle; middle: a two-stage pipeline; bottom: spreading the calculation over four clock cycles.

Consider a simple example. Suppose that the following quadratic must be evaluated for a range of values of x at every clock cycle:

$$\begin{aligned} y &= ax^2 + bx + c \\ &= (a \times x + b) \times x + c \end{aligned} \tag{5.1}$$

If implemented within a single clock cycle, as shown in the top panel of Figure 5.1, the propagation delay is that of two multiplications and two additions. If this is too long, the calculations may be spread over two clock cycles, as shown in the middle panel. Register y_1 splits the computation path allowing the second multiplication and addition to be moved into a second clock cycle. Note that register x_1 is also required to delay the input for the second clock cycle. Without it, the new value of x on the input would be used in the second multiplication, giving the wrong result. The propagation delay per clock cycle is almost halved, although a small additional propagation delay is introduced with the additional register. The two-stage pipeline would enable the clock speed to be almost doubled. Operating as a pipeline, the throughput is increased as a result of the higher clock speed. One new calculation begins every clock cycle, even though each calculation now takes two clock cycles. This increase in throughput comes at the expense of a small increase in latency, which is the total time the calculation takes. This is a result of the small increase in total propagation delay.

Pipelining may result in a small increase in logic block usage. This results from the need to introduce registers to construct the pipeline stages. These registers are implemented using flip-flops within the logic blocks. They are typically mapped onto unused flip-flops within the logic cells that are already used for implementing the logic. For example, the flip-flops to implement y_1 will most likely be mapped to the flip-flops on the output of the logic cells implementing the adder. Register x_1 does not correspond to any logic outputs, although it may be mapped to unused flip-flips of other logic within the design.

In the bottom panel of Figure 5.1 the pipeline is extended further, splitting the calculation over four clock cycles. This time, however, the clock period cannot be halved because the propagation delay of the multiplication is significantly longer than that of the adder. Stages 1 and 3 of the pipeline, containing the multipliers, will govern the maximum clock frequency. Consequently, although the maximum throughput can be increased a little, there is a larger increase in the latency because stages 2 and 4 have significant timing slack.

This imbalance in the propagation delay in different stages may be improved through *retiming* (Leiserson and Saxe, 1991). This requires some of the logic from the multiplication to be moved from stages 1 and 3 through registers y_1 and y_3 to stages 2 and 4, respectively. Depending on the logic that is shifted, and how it is shifted, there may be a significant increase in the number of flip-flops required as a result of retiming. Performing such fine-grained retiming manually can be tedious and error prone. Fortunately, retiming can be automated, for example it is provided as an optimisation option when compiling Handel-C designs (Mentor, 2010a). Retiming can even move the registers within the relatively deep logic created for multipliers and dividers; places that are not accessible within the source code. Such automated retiming can make debugging more difficult because it affects both the timing and the visibility of the intermediate signals.

If retiming is not available, or practical (for example where the output is fed back to the input of the same circuit for the next clock cycle), there are several techniques that can be used to reduce the logic depth (Mentor, 2010b). Of the arithmetic operations, division produces the deepest logic, followed by multiplication. Multiplication and division by constants (especially powers of two) can often be replaced by shifts or shifts and adds, while modulo operations based on powers of two can be replaced by simple bit selection. While long multiplication and division can be implemented efficiently with a loop, if a high throughput is required, the loop can be unrolled and pipelined (Bailey, 2006), with each loop iteration put in a separate pipeline stage. The ripple carry used for wide adders can increase the depth of the logic. Carry propagation can be pipelined by using narrower adders and propagating the carry on the next clock cycle. If carried to the extreme, this results in a form of carry save adder (see Parhami (2000) for some of the different adder structures). Greater than and less than comparisons are based on subtraction. If they can be replaced with equality comparisons, the logic depth can be reduced because the carry chain is no longer required.

Although a pipeline is a parallel computational structure in that each stage has separate hardware that operates in parallel with all of the other stages, from a dataflow perspective it is better to think of pipelines as sequential structures (Johnston *et al.*, 2006b; 2010). Directly associated with the sequential processing is the *pipeline latency*. This is the time from when data is input until the corresponding output is produced. This view of latency is primarily a source-driven concept.

From a sink-driven perspective, the latency implies that data must be input to the pipeline at a predetermined time before the output is required. This period is referred to as *priming* the pipeline. If the output is synchronous, priming can be scheduled to begin a preset time before data is required at the output. Unfortunately, this is not an option when data is requested asynchronously by a sink. In this case, it is necessary to prime the pipeline with data and then *stall* the pipeline in a primed state until the data is requested at the output. This is discussed in more detail in the next section on synchronization. Source-driven processes may also be stalled if valid data is not available on every clock cycle.

During priming, the output is invalid, while the valid data propagates through the pipeline. The corresponding period at the end of processing is pipeline *flushing*, continuing the processing after the last valid data has been input until the corresponding data is output. Note that this use of the term pipeline flushing differs from that used with instruction pipelining of the fetch/decode/execute cycle; there flushing refers to the dumping of the existing, invalid, contents of the pipeline after a branch is taken, and repriming with valid data. The consequence of both priming and flushing is that the pipelined process must be clocked for more clock cycles than there are data elements.

5.1.2 Process Synchronisation

Whenever there are operations working in parallel, a recurring problem is the coordination of behaviour and communications between the different processes. This is relatively easy if all of the operations are streamed within a pipeline. However, it is more difficult for random access or hybrid processing modes, especially where the latencies may be different or where the time taken by the operations may vary. When there is a mix of processing modes, it is often necessary to synchronise the data transfer between operations.

If all of the external events driving the system are regular and each operation has a constant processing time or latency, then global scheduling can be used. This can be accomplished by using a global counter to keep track of events and the algorithm state, as shown in Figure 5.2. The various operations are then scheduled by determining when each operation requires its data and matching the count. An example of global scheduling is processing an image on-the-fly as it is being displayed. Each line is the same length, with regular and synchronous events – the horizontal and vertical sync pulses. This enables the processing for each line to be scheduled to start so that the pipeline is primed when the data must be available for output to the display. Global scheduling requires all of the operations in the algorithm to be both predictable in their execution time, and tightly coupled.

Other communication and synchronisation methods must be used when each operation or module is loosely coupled. For source-driven processes, one way to achieve this is to associate a ‘data valid’ flag or token with the output. This token is then used to control the downstream operations or processes, by either stalling the process or continuing to run but ignoring the invalid data and marking the corresponding outputs as being invalid. For sink-driven applications, in addition to the data valid flag propagated downstream, it is also necessary to have a ‘wait’ flag or token propagated upstream (Mentor, 2010c). This is asserted when a processor is unable to accept further data on its input and requests the upstream process to stop providing data. Depending on the nature of each operation, it may be necessary to use FIFO buffers to smooth the flow of data between the processes. In particular, use of such buffers can simplify the management of pipeline priming and flushing. The interface between the operations within the Pixel-Streams library is based primarily on this form of synchronisation (Mentor, 2010c), and enables the operations to be kept separate and developed independently.

Another mechanism that allows point-to-point communication and synchronisation between concurrent processes is the CSP (communicating sequential processes) model (Hoare, 1985). One process is the transmitter, the other is the receiver, and they are connected by a data channel. Both processes must be ready before the communication or data transfer takes place. If one process is not ready, the other process will block or stall until both ends of the channel are ready. Thus, a CSP channel will ensure that both processes are synchronised at the point of data transfer. Channels can be used to pass data between operations in a pipeline, where they will automatically synchronise the processes. The fact that channels block when they are not ready allows them to be used with both source and sink driven pipelines. With source-driven flow, the downstream operations will stall when data is not available at the input.

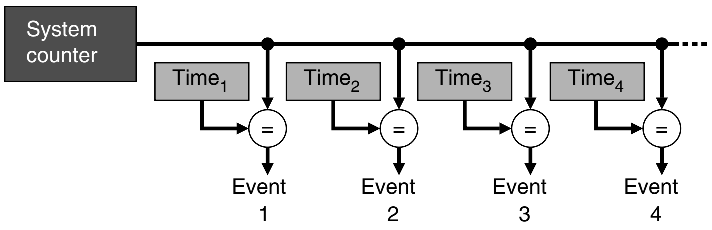


Figure 5.2 Global scheduling derives synchronisation signals from a global counter.

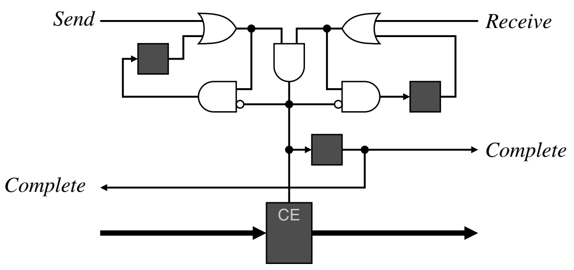


Figure 5.3 Channel synchronisation logic. (Adapted from Page and Luk, 1991.)

With sink-driven flow, the upstream operations will automatically prime and, then when the data is not read from the output, will stall waiting for the available data to be transferred. Note that because channels can block the execution of a process, care must be taken to avoid deadlocks caused by circular dependencies (Coffman *et al.*, 1971).

An example of channel synchronisation logic is shown in Figure 5.3. Identical logic is used to synchronise both the sender and receiver, so only the send logic will be described. If the receiver is not ready, then the *Send* signal is latched to remember that data is waiting to transfer. The *Complete* signal is clocked low to indicate that the sending process should wait. When the receiver becomes ready, the data will be transferred into the register on the next clock edge and the *Complete* signal is latched high for one clock cycle.

Combining the channel with a FIFO buffer will relax the timing constraints. It allows a source process to continue producing outputs until the FIFO buffer is full before it stalls. Similarly, a sink process will read available data from the FIFO buffer until the buffer is empty before it stalls.

The use of FIFO buffers enables operations with variable latency to be combined within a pipeline. In many video-based applications, where data is read from a camera or written to a display, the blanking intervals between rows and between frames can occupy a significant proportion of the total time. The strict timing of one pixel per clock cycle may be relaxed somewhat by enabling operations to work ahead during the blanking period, storing the results in a FIFO buffer, which can then be used to smooth out the timing bumps later.

5.1.3 Multiple Clock Domains

With some designs, using separate clocks and even different clock frequencies in different parts of a design are unavoidable. This is particularly the case when interfacing with external devices that provide their own clock. It makes sense to use the most natural clock frequency for each device or task, with different sections of the design running in separate clock domains. A problem then arises when communicating or passing data between clock domains, especially when the two clocks are independent, or when receiving asynchronous signals externally from the FPGA.

If the input of a flip-flop is changing when the flip-flop is clocked (violating the flip-flop's setup or hold times) the flip-flop can enter a metastable state where the output is neither a 0 nor a 1. When in this state, the output will eventually resolve to either a 0 or 1, but this can take an indeterminate time. Any logic connected to the output of the flip-flop will be affected, but because the resolution can take an arbitrarily long time, there is a danger that with the propagation delay through the logic, the setup times of downstream flip-flops may also be violated. The probability of staying in the metastable state decreases exponentially with time, so waiting for one clock cycle is usually sufficient to resolve the metastability. A *synchroniser* does just this; the output is fed to the input of

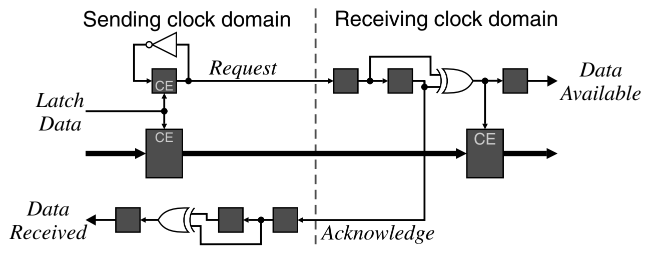


Figure 5.4 Bidirectional handshaking circuit for data synchronisation across clock domains.

a second flip-flop. The reliability of a synchroniser is expressed as the mean time between failures (Dike and Burton, 1999):

$$MTBF = \frac{e^{T/\tau}}{T_W f_S f_T} \tag{5.2}$$

where τ is the settling time constant of the flip-flop, T is the period of the settling window allowed for metastability to resolve; the numerator gives the probability that any metastability will be resolved within the period T. T_W is the effective width of the metastability window and is related to the setup and hold times of the flip-flop, f_S is the frequency of the synchroniser's clock, and f_T is the frequency of asynchronous data transitions on the clock domain boundary. The denominator of Equation 5.2 is the rate at which metastable transitions are generated. T is usually chosen so that the *MTBF* is at least ten times the expected life of the product (Ginosar, 2003). This is usually satisfied by setting T to be one clock period.

To guarantee that data is transferred regardless of the relative clock frequencies of the transmitter and receiver, bidirectional synchronised handshaking is required (Ginosar, 2003). Such a circuit is shown in Figure 5.4. As the data is latched at the sending end, the *Request* line is toggled to indicate that new data is available. This *Request* signal is synchronised using two flip-flops. The resolution time is one clock period less the propagation delay of the exclusive OR gate and the setup time on the clock enable of the data latch. The exclusive OR gate detects the transition on the *Request* line, which is then used to enable the clock on the data latch (the data from the sending clock domain will have been stable for at least one clock cycle and does not require separate synchronisation), and provides a single pulse out on the *Data Available* line. An *Acknowledge* signal is sent back to the sending domain, where again it is synchronised and a *Data Received* pulse generated. It is only after this signal has been received that new data may be safely latched into the sending register. Otherwise, new data may overwrite the old in the sending data register before the contents have been latched at the receiver.

This whole process can take either three or four clock cycles of the slowest clock domain, limiting the throughput to this rate. If the sending domain is always slower, then the acknowledgement logic may be saved by limiting the input to send every three clock cycles. The danger with this, though, is that if the clock rates are later changed, the synchronisation may no longer be valid (Ginosar, 2003).

An alternative approach based on using dual-port fabric RAM as a form of FIFO buffer is shown in Figure 5.5. The RAM has one port in each clock domain and enables data to be queued while the synchronisation signal is sent from one domain to the other. This enables a sustained throughput of one data element per slow clock cycle. The most significant bit of the sending domain's address counter is sent across the domain boundary using a pair of synchronisation flip-flops to resolve metastability. This is then used to indicate when one block of data (half of the RAM) is available and the receiver can start to read. A small amount of logic prevents the address counter from addressing past the end of the available block. To provide safe transfer regardless of which clock domain is slower, both address counters should be interlinked to make access to the two halves of the address space mutually exclusive.

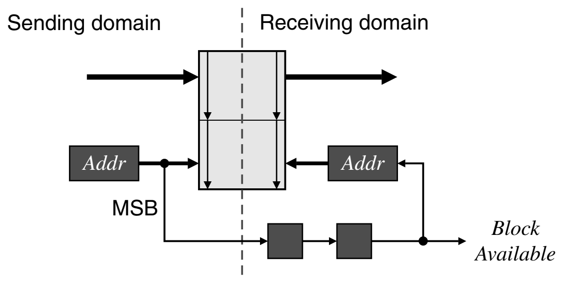


Figure 5.5 Transfer from a slower sending domain using a dual-port RAM.

5.2 Memory Bandwidth Constraints

Some operations require images to be partially or wholly buffered. While current high-capacity devices have sufficient on-chip memory to buffer a single image, in most applications it is poor use of this valuable resource to simply use it as an image buffer. For this reason, image frames and other large data sets are more usually stored in off-chip memory. This effectively places large amounts of data behind limited bandwidth and serialised (at the data word level) connections.

For this reason, it is essential to process as much of the data as possible while it passes through the FPGA, and to limit the data traffic between the FPGA and external devices. The standard software approach is to read each pixel required by an operation from memory, apply the operation and write the results back to memory. This is then repeated for every pixel within the image. A similar process is repeated for every operation that is applied to the image. The continual reading from and writing to memory may be avoided in many places by using a pipelined stream processing architecture.

Some operations cannot achieve this and may require an appropriate design of the memory architecture to overcome bandwidth problems. Other operations may be modified to cache data that is reused. For example, row buffering is a form of caching where pixel values read at one time are saved for reuse when processing subsequent rows.

5.2.1 Memory Architectures

The simplest way to increase memory bandwidth is to have multiple parallel memory systems. If each memory system has separate address and data connections to the FPGA, then each can be accessed independently and in parallel.

The crudest approach is to have multiple copies of the image in separate memory banks. Each bank can then be accessed independently to read the required data. For off-chip memory this is not the most memory efficient architecture, but it can be effective for on-chip caching when multiple processors require independent access to the same data.

With many operations, it is likely that adjacent pixels need to be accessed simultaneously. By partitioning the address space over multiple banks, for example using one memory bank for odd addresses and one for even addresses, adjacent addresses may be accessed simultaneously. This can be applied to both the row and column addresses, so, for example, four banks could be used to access the four pixels within a 2×2 block (top panel of Figure 5.6). Rather than simply partitioning the memory, the addresses may be scrambled to enable a larger number of useful subsets to be accessed simultaneously (Lee, 1988; Kim and Kumar, 1989). For example, the bottom panel of Figure 5.6 enables four adjacent pixels in either a row or a column to be accessed simultaneously. It also allows the main diagonal and the anti-diagonal of a 4×4 block, and many (but not all) 2×2 blocks to be accessed concurrently (Kim and

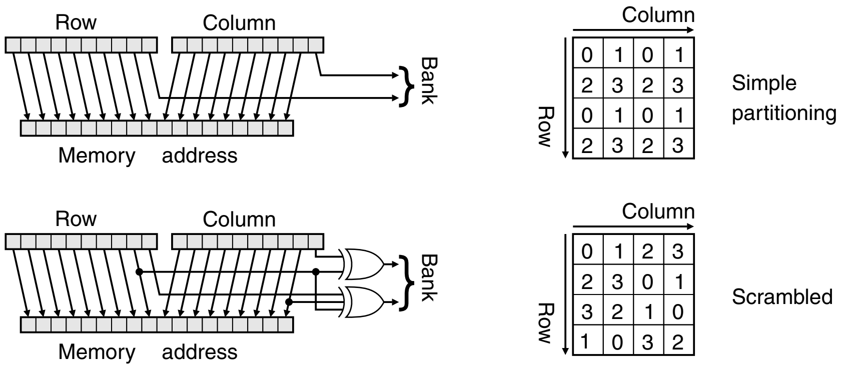


Figure 5.6 Memory partitioning schemes. Top: simple partitioning allows access of 2×2 blocks; bottom: scrambling the address allows row, column and main diagonal access.

Kumar, 1989). It is not possible to have all 4×1, 1×4 and 2×2 blocks all simultaneously accessible with only four memory banks. This can be accomplished with five banks, although the address logic is considerably more complex as it requires division and modulo 5 remainders (Park, 1986). In addition to the partitioning and scrambling logic, a series of multiplexers is also required to route the address and data lines to the appropriate memory bank.

A further approach to increasing bandwidth is to increase the word width of the memory. Each memory access will then read or write several pixels. This is effectively connecting multiple banks in parallel but using a single set of address lines common to all banks. Additional circuitry, as shown in Figure 5.7, is usually required to pack the multiple pixels before writing to memory, and to unpack the pixels after reading, which can increase the propagation delay or latency of this scheme. Data packing is effective when pixels that are required simultaneously or on successive clock cycles are packed together. This will be the case when reading or writing streamed images.

Another common use of multiple memory banks is bank switching or double buffering. Bank switching is used between two successive image processing operations to decouple the memory access patterns. This makes it useful when pipelining random access image processing operations, or with the hybrid processing mode to connect between stream processing and random access processing. It uses two coupled memory banks, as shown in Figure 5.8. The upstream process has access to one bank to write its results while the downstream process accesses data from within a second bank. When the frame is complete, the role of the two banks is reversed, making the data just loaded by the upstream process available to the downstream process. Bank switching therefore adds one frame period to the latency of the algorithm.

If the upstream and downstream processes have different frame rates, or are not completely synchronised, then the two processes will want to swap banks at different times. If the bank switching is forced by one process when the other is not ready, this can result in tearing artefacts, with part of

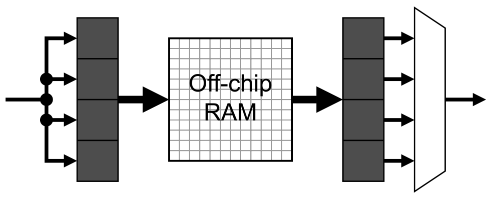


Figure 5.7 Packing multiple pixels per memory location.

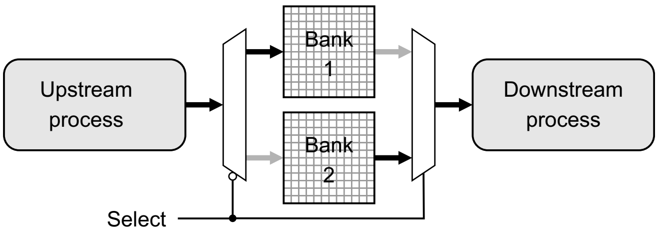


Figure 5.8 Bank switching or double buffering.

the frame containing old data and part new. This problem may be overcome by triple buffering, using a third bank to completely separate the reading and the writing (Khan *et al.*, 2009). This has the obvious implication of increasing the component count of the system and will use a larger number of I/O pins.

Bandwidth can also be increased by using multiport memory. Here the increase is gained by allowing two or more concurrent accesses to the same block of memory through separate address and data ports. While this is fine in principle, in practise it is not used with external memory systems because multiport memory is more specialised and consequently considerably more expensive.

A more practical approach to mimic multiport memory is to run the memory at a higher clock speed than the rest of the system. The resulting design will allow two or three accesses to memory within a single system clock period. The memory clock should be synchronised with the system clock to avoid synchronisation issues. Double data rate (DDR) memory is one example of memory that allows two data transfers per clock cycle, using both the falling and rising edges of the clock. A higher frequency RAM clock is only practical for systems with a relatively low clock speed, or with high speed memories.

Alternatively, the complete design could be run at the higher clock speed, with the data entering the processing pipeline every several clock cycles. Such approaches are called *multiphase* designs, with several clock cycles or phases associated with the pixel clock. Multiphase designs not only increase the bandwidth, but can also reduce computational hardware by reusing the hardware in different phases. The extreme is multiphase design is the Tabula ABAX FPGA, where the complete logic of the FPGA is changed in each of up to eight phases (Section 2.4.6).

5.2.1.1 Memory Technologies

It is generally best to use static memory for frame buffers, especially where they are accessed randomly and frequently. Static memory is faster than dynamic memory and is usually lower power. However, because it uses six transistors per bit rather than the one used by dynamic memory it is also more expensive. Many modern high speed static memories have been designed for use as instruction and data caches for high end serial computers. Consequently, they operate best in a burst mode and can take one or more clock cycles to switch between reading and writing (bus turnaround). Zero bus turnaround (ZBT) memories are designed with no dead period, enabling true low latency random access. This makes them easier to interface to an application, making them a good choice for frame buffers.

Dynamic memory is better for large volumes of data where there is not a strict time constraint. This is because dynamic memory is slower and generally requires several clock cycles latency from when the address is supplied to when the data is available. Dynamic memories have their highest bandwidth when operated in burst mode, with a series of reads or writes on the same row. They are also significantly harder to interface to for two reasons. They must be refreshed periodically (usually every 64 ms) by accessing every row once within this period. The internal structure means that when a row is accessed, all values on that row are available internally, with the column address selecting the entry on the row. Because of this segmented address structure, the row address is required before the column address. Usually, the row and column addresses share the same pins. All of these factors mean that some form of caching may be required to smooth the flow of data to or from the memory.

5.2.2 Caching

A cache memory is a small memory located close to a processor that holds data loaded from an external data source, or results that have been previously calculated. It provides rapidly accessible temporary storage for data that is used multiple times. The benefit of a cache is that it has faster access than re-reading the data from external memory (or hard disk) or recomputing the data from original values.

On an FPGA, a cache can be used to buffer accesses to external memory, particularly memory with a relatively long latency or access time, or operates in burst mode. In this capacity, it can also be used to smooth the flow of data between off-chip memory and the application. The cache can significantly reduce the bandwidth required between the FPGA and memory if the data in the cache is used multiple times (Weinhardt and Luk, 2001a). In some applications, the cache is positioned between the processor and the main memory, as shown in Figure 5.9, with accesses made to the cache rather than the memory. The cache controller in this case is then responsible for ensuring that the required data will be available in the cache. A separate cache is often used to hold the results to be written back to memory. With some operations, for example clustering or classifier training algorithms, the task may be accelerated by using multiple parallel processors working independently from the same data. Since the block RAMs on the FPGA can all be accessed independently, the cache bandwidth can be increased by having multiple separate copies of the cached data (Liu *et al.*, 2008), one for each process. This allows each process to access the data without conflict with other processes. Whether there is a single results cache or multiple distributed results caches will depend primarily on how the results are reused by the processes. In Figure 5.9 it is assumed that results are only reused locally.

One possible structure for the caches is an extended FIFO buffer. With a conventional FIFO buffer, only the item at the head of the buffer is available, and when it is read it is removed from the buffer. However, if the access and deletion operations are separated, and the addressing made external so that not just the head of the buffer is accessible, then the extended FIFO buffer can be used as a cache (Sedcole, 2006). With this arrangement, data is loaded into the buffer in logical blocks. Addressing is relative to the head of the buffer, allowing local logical addressing and data reuse. When the processing of a block is completed, the whole block is removed from the head of the FIFO buffer, giving the next block the focus.

Another arrangement is to have the cache in parallel with the external memory, as shown in Figure 5.10, rather than in series. This allows data to be read from the cache in parallel with the memory, increasing the bandwidth. The cache can be loaded either by copying the data as it is loaded from memory or by the process explicitly saving the data that it may require later into the cache.

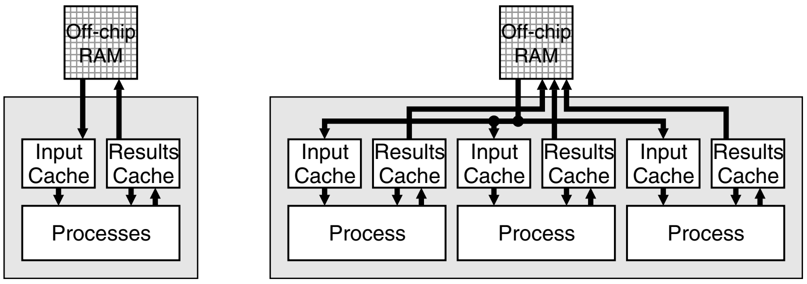


Figure 5.9 Caching as an interface to memory. Left: a single cache is shared between processes; right: data is duplicated in multiple input caches, but each process has a separate results cache.

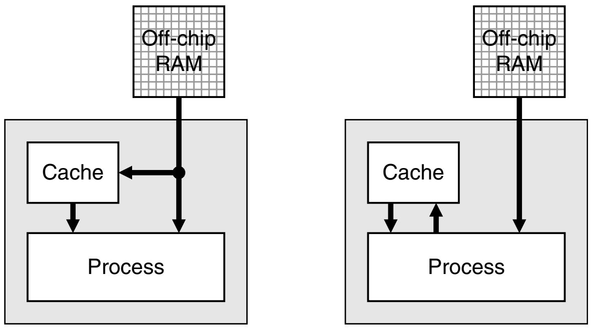


Figure 5.10 Parallel caches.

5.2.3 Row Buffering

One common form of caching is row buffering. Consider a 3×3 window filter – each output sample is a function of the nine pixel values within the window. Without caching, nine pixels must be read for each window position (each clock cycle for stream processing) and each pixel must be read nine times as the window is scanned through the image. Pixels adjacent horizontally are required in successive clock cycles, so may be buffered and delayed in registers. This reduces the number of reads to three pixels every clock cycle. A row buffer caches the pixel values of previous rows to avoid having to read the pixel values in again (Figure 5.11). A 3×3 filter spans three rows, the current row and two previous rows; a new pixel is read in on the current row, so two row buffers are required to cache the pixel values of the previous two rows.

Each row buffer effectively delays the input by one row. An obvious implementation of such a digital delay would be to use an N-stage shift register, where N is the width of the image. There are several problems with such an implementation on an FPGA. Firstly, a shift register is made up of a chain of registers, where each bit uses a flip-flop. Each logic cell only has one flip-flop, so a row buffer would use a significant proportion of the FPGAs resources. Some FPGAs allow the logic cells to be configured as shift registers. This would use fewer of the available resources, since each cell could provide a delay of up to 16 (or 32 for the Virtex 5; Xilinx, 2008f) clock cycles.

A better use of resources would be to use a block RAM as a row buffer. Block RAMs cannot be directly configured as shift registers (one exception is Altera's Cyclone and Stratix devices), although most of the newer FPGAs can configure them as FIFO buffers, which can be arranged to provide an equivalent functionality. Those that natively provide FIFO buffers will automatically manage the addressing of the RAM. Otherwise, a FIFO buffer may be built from a dual-port memory, as shown in Figure 5.12. Two counters are needed, one for providing the address for writing to the memory and one for reading. The memory is used as a *circular buffer*, with the first address appearing sequentially after the last address within the memory. For blocks of memory that are a power of two long, this is achieved by allowing the address counters to wrap around. If the memory is other than a power of two long, then the counter has to

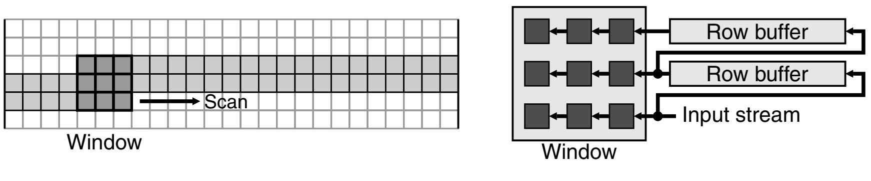


Figure 5.11 Row buffering caches the previous rows so they do not need to be read in again.

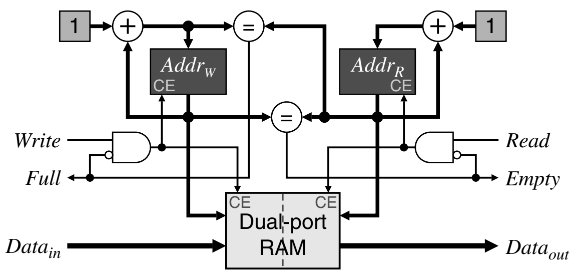


Figure 5.12 Circular memory based FIFO buffer. The full and empty detection logic is not required for use as a row buffer.

be modified to reset to 0 after the last address is reached. A small amount of logic is also required to detect when the buffer is full or empty, although for use as a row buffer this is not necessary. The buffer is empty when the read and write addresses are identical, and full if incrementing the write address would make it equal to the read address.

A FIFO buffer can be used as a shift register as follows. Assume that the buffer is initially empty. A series of N writes to the FIFO buffer will advance the write position to N locations ahead of the read position. Then, if a read and write are preformed simultaneously every clock cycle, the value read will be delayed from that written by N clock cycles. If the counters are provided explicitly, then resetting the read counter to 0 and the write counter to N would create an N-stage shift register. Since both counters are incremented at the same time, a single counter could be used, with an adder to offset it by N to get the write address. If the size of the RAM is exactly the same as the width of the image then the read address will always be the same as the write address, simplifying the circuit further.

Note that having multiple buffers in parallel, such as in Figure 5.11, is the same as a single buffer but with a wider data path. If implemented using separate RAM blocks, the address counters may be shared.

In many applications, it is necessary to maintain a column address, even if only for the purposes of reading from or writing to a frame buffer. This column address may be used to index the buffer rather than using a separate counter. The buffer is then no longer circular, but is automatically indexed to the current position in the image.

5.2.4 Other Memory Structures

With current technologies, memory is inherently a serial device. Although internally, data is stored in parallel, with a separate latch (or capacitor for dynamic memory) per bit stored. (While some more recent dynamic RAMs can store multiple bits per capacitor using different levels, this does not affect the argument here.) However, the addressing and access mechanisms mean that only one memory location can be selected for reading or writing at a time. Multiple data items can, therefore, only be accessed serially.

FIFO and circular buffers have been described in some detail in the previous section on row buffering. A FIFO buffer allows data items to be visited twice, the first time when the data items are loaded into the buffer, and the second time when they are retrieved.

Other useful memory structures are stacks, linked lists, trees and hash tables.

5.2.4.1 Stacks

Stacks also allow data items to be visited twice, although the second time the items are accessed in the reverse order. Items are *pushed* onto a stack by storing them in successive memory locations. When an

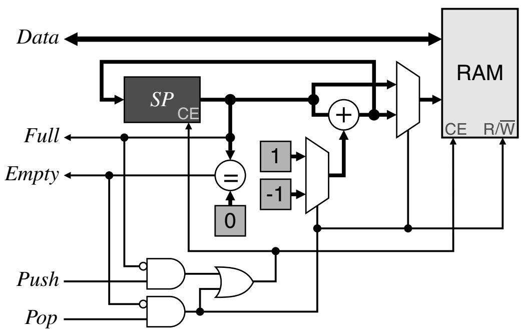


Figure 5.13 Implementation of a stack.

item is *popped* from the stack, the most recently saved item is retrieved first. Stacks are, therefore, a form of first-in last-out or last-in first-out queue.

A stack can be implemented using a single-port memory, combined with a single address counter as the stack pointer, *SP*, as shown in Figure 5.13. It is usual for the stack pointer to address the next available free location. A push writes to the top of stack, as pointed to by *SP* and increments the stack pointer to point to the next free location. A pop needs to decrement the stack pointer before reading to access the data on the top of the stack.

In the implementation shown above, a multiplexer is used to add either 1 or -1 to *SP* depending on whether a *Push* or *Pop* operation is used respectively. A *Push* uses *SP* directly as the RAM address, whereas a *Pop* uses the decremented *SP* as the address. The stack is empty if the address is zero; in this condition, a *Pop* is not allowed because there is nothing left on the stack to pop. Note that the test for zero can be implemented by simply taking the NOR of all of the bits. The stack is full after the last memory element is written. Incrementing *SP* will wrap it past the end. This is prevented by making *SP* one bit longer than needed to address the memory. The stack is then full when the most significant bit of *SP* is set.

5.2.4.2 Linked Lists

A linked list is useful for maintaining a dynamic collection of data items. Each item is stored in a separate location in the memory; however, successive items are not necessarily in sequential memory locations. Associated with each item is a pointer to the next item (a pointer to an item is just the memory address of that item), resulting in an implicit ordering of the items within the list. Linked lists have an advantage over arrays when inserting an item into the list. To insert an item into an array, all the successive items in the array must be moved to new memory locations to make space at the appropriate position in the array. However, with a linked list, the new item can be stored in any free memory location and the pointers adjusted to put it into the correct place in the sequence. Similarly, when deleting an item from an array, all subsequent items must be shifted back. With a list it is only necessary to adjust the pointers to skip the item that is removed.

The first item in the list is called the *head* of the list. The last item in the list has a *null pointer*, Ø, indicating that there is no next item. On software systems, 0 is often used as the null pointer because 0 is not usually a valid data address. However, on FPGAs 0 usually is a valid data address. This may be resolved either by using a separate flag to indicate a null pointer, or by not using location 0 to hold data. The former approach requires one extra bit per pointer to contain the null pointer flag but it does mean that only one bit needs to be tested to determine if it is a null pointer. Not using location 0 reduces the number of items that may be stored in a block of memory by one item and a null pointer can be tested by checking if all bits are 0.

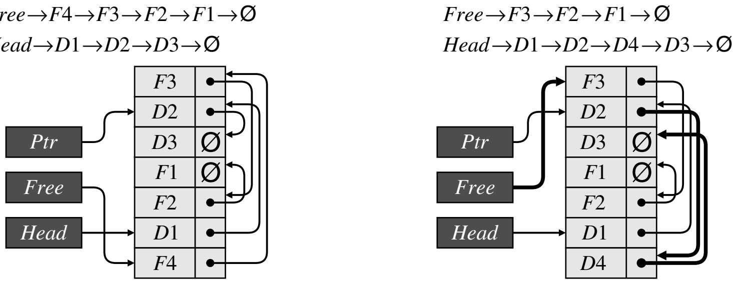


Figure 5.14 Inserting an element into a linked list. Changed links are shown in bold.

One problem with linked lists is keeping track of which memory locations are free. There are two approaches to solving this problem. One is to maintain a series of one bit registers indicating that a memory location is free. Associated with this is a priority encoder to translate the first 1 into the corresponding address. While this is suitable for short lists, it can quickly get expensive as the address space gets larger. A second approach is to reuse the linking mechanism to maintain a second list of unused entries. This requires that the memory be initialised with the links before being used.

Inserting new data into a list requires four memory accesses (Figure 5.14). Assume that it is necessary to insert the new data item, $D4$, after $D2$ and that there is currently a pointer to $D2$. Also assume that unused entries are in a list pointed to by *Free*. The first access is to read the pointer associated with $D2$ (the pointer to $D3$). The pointer is then changed to point to the head of the free list (which will become the new data item), and is written back to $D2$. The item pointed to by the head of the free list (the pointer associated with $F4$) becomes the new head, so it must be read from the memory. Finally, the new data and pointer to $D3$ is written to the new element.

To implement even this relatively simple operation on an FPGA would require a finite state machine to sequence each of the steps. The controlling state machine would require a series of branches to manage each different operation: initialisation, traversing a list, insertion and deletion. Since each operation will require several clock cycles, it will be necessary to include synchronisation logic between the linked list and the rest of the design.

Note that the sequential ordering means it is necessary to insert an item either after a specified item in the list or at the head of the list. Items cannot be directly inserted before a specified item because there is no direct reference to the previous item in a list. The previous item points to the specified item; to insert an entry this pointer needs to be changed to point to the new item. With a singly linked list, the only way to locate the previous item is to sequentially scan through the list from the head.

This limitation can be overcome by using a doubly linked list; this has bidirectional links, to both the previous and the next items in the list. Having the extra link also means that extra memory accesses are required when inserting and deleting links. The extra pointer associated with each data item would also require a wider memory.

5.2.4.3 Trees

Another data structure commonly used in computer vision to represent hierarchically structured data is a tree. An example tree is shown on the left in Figure 5.15. The node at the top of the tree is called the *root* node; node A is the root in this example. The nodes immediately below a node are called the *child* nodes. For example, D, E and F are children of B, and B is their parent. Nodes that have no children are *leaf* nodes. In this example, D, F, G, I, J and K are the leaf nodes.

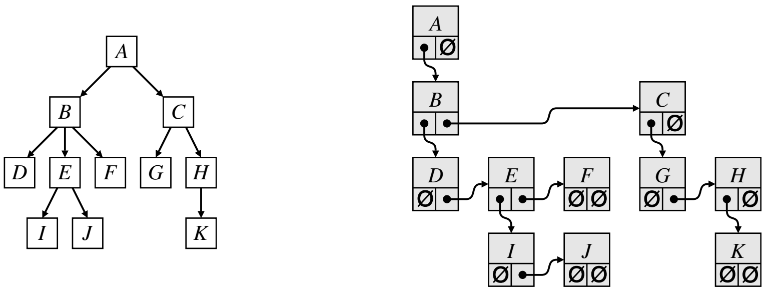


Figure 5.15 A representation of a tree.

In a general tree, a node can have an arbitrary number of children. Therefore, it is inconvenient for a node to point to all of its children. One general representation of a tree is as a form of two-dimensional linked list. Every node has two pointers: one to its first child (or null if it is a leaf node) and one to its next sibling. In this way, all of the children of a node are contained in a linked list. This approach to representing a tree is shown in Figure 5.15.

Note that this method is suitable for navigating from the root of a tree down to a node, but there is no pointer from a child to its parent so navigating back up the tree is more difficult. This may be overcome by using doubly linked lists, but would require every node to maintain three pointers. A more efficient alternative in many applications is to use a separate memory structure to remember the context of a node. When performing a depth-first traversal of the tree, a stack can be used to remember parent nodes. As the link from a node to its first child is taken, a pointer to the node can be pushed onto a stack. Then to move back up the tree after reaching a leaf, it is only necessary to pop the most recently visited parent from the stack. The stack therefore provides the required link from a child back to its parent. If performing a breadth-first search, it is more appropriate to use a FIFO queue. As a node is visited, if it has any children the pointer to the first child node is stored in the queue. Then when the last child is reached, the first entry in the queue is the next node to visit.

One particular type of tree deserves special mention is a *binary tree*. It has at most two children, so for a binary tree it is possible to (and indeed useful to) directly point to the two children rather than having a list to represent the children.

5.2.4.4 Hash Tables

A sparse array is an array with a large address space, but only a small proportion of the array has data at any one time. If the whole array is reserved, this can be wasteful of the memory resources available on an FPGA. While the memory requirement of the array may be significantly reduced using a linked list or tree to represent the occupied data elements, these structures incur a significant overhead of having to search through the list or tree to find a particular element.

An alternative approach is to store the data using a hash table. As shown in Figure 5.16, the original address is mapped using a hash function to a much smaller address space, that of the hash table, where the data is actually stored. The reduction memory comes at the cost of evaluating the hash function. In most applications, this cost can be lower than that of searching through a linked list or tree.

A good hash function is one that maps the different input addresses that are actually used to different slots within the hash table. If the input addresses can be considered random, then a simple and effective hash function can be to use a subset of the bits of the input address. However, if the set of input addresses is likely to be clustered, then a good hash function will distribute these to quite different slots to reduce the

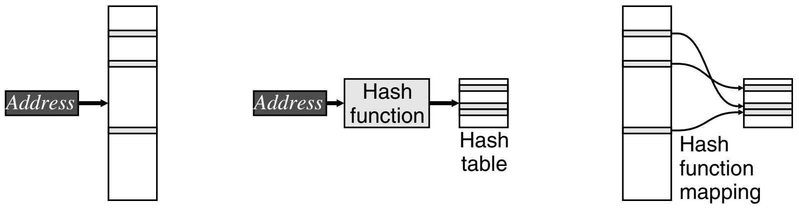


Figure 5.16 Address hashing. Left: original sparse address space; centre: using a hash function to reduce address space; right: the mapping between address spaces by the hash function.

chance that multiple input addresses will map to the same slot. For this reason, a common technique is to base the hash function on a pseudorandom number generator, using the input address as the seed. This will tend to map the input addresses randomly throughout the hash table.

Inevitably, though, there will be address collisions, with multiple input addresses mapping to a common slots. There are two main approaches for resolving such collisions. One is to store multiple elements per slot, using a linked list for example. This requires additional data structures to the hash table. Another alternative is to use a so-called open addressing scheme to store the data in the next available slot (Peterson, 1957). Both methods require a search in the event of collisions. The search is kept small by ensuring that the used slots are randomly distributed and the hash table is not too full. Open addressing is sensitive to the fill factor of the hash table (the proportion of slots that are used) and it is generally better to keep the maximum fill factor below about 75%. There are several advanced techniques (Askitis, 2009) that can be used to increase the fill factor, but these generally come at the expense of more complex hash functions.

A hash table can be considered as a form of associative memory. The sparse addressing of hash tables makes them appropriate for implementing caches. When a hash table is used for a cache, whenever a collision occurs the previous entry can be discarded, avoiding the need for searching.

The input addresses do not necessarily need to be memory locations, but could be strings or other complex data structures that are reduced using the hash function to provide an index into the hash table.

5.3 Resource Constraints

The finite number of available resources in the system such as function blocks or local and off-chip RAM imposes a constraint. On an FPGA, there may be a number of concurrent processes that need access to a particular resource in a given clock cycle, which can result in contention, or undefined behaviour if both are permitted access.

5.3.1 Resource Multiplexing

An expensive resource may be shared between multiple parts of a design by using a multiplexer. There are three conditions required for this to be practical. Firstly, the cost of the resource must be less than the cost of the multiplexers required to share the resource. For example, sharing an adder in most instances is not practical because the cost of building a separate adder will usually be less that the cost of the multiplexers required to share the adder. However, sharing a complex functional block may well be practical. The second condition is that the resource must only be partially used in each of the locations that it is used. If the instances of the resource are fully used, then sharing a single instance between the multiple locations will slow the system down. The third consideration is the timing of when the resource is required. If the resource must be used simultaneously in multiple parts of the design, then multiple copies must be used.

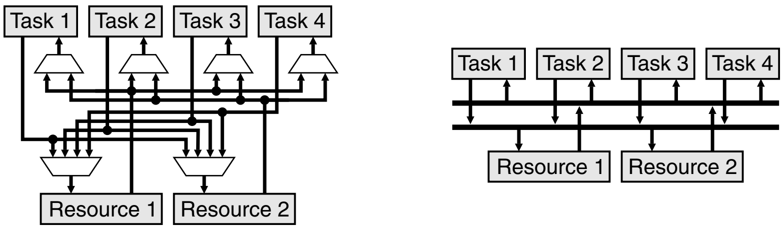


Figure 5.17 Sharing a pool of resources amongst several tasks. Left: using multiplexers; right: using busses.

The limitation of sharing a resource is that multiple simultaneous accesses to the resource are prohibited. Sharing often requires associating some form of arbitration logic with the resource to manage the access; some of the options for this are explored in more detail in the next section.

An option when only a few instances of a resource are required simultaneously with instances required by many tasks is to have a pool of resources. This approach is illustrated in Figure 5.17. When the resource pool is shared by a large number of tasks, the multiplexing requirements can become prohibitive. An alternative is to connect to the resources via busses. Note that the busses are additional resources that must also be shared, because only one task or resource may use a bus at a time.

Sometimes, careful analysis of a problem can identify potential for resource sharing where it may not be immediately obvious. Consider obtaining the bounding boxes of a set of objects from a streamed input image consisting of labelled pixels, where the label indicates the associated object (see also Section 11.1). Since each object can appear anywhere in the image, and the bounding boxes for different objects may overlap, it is necessary to build the bounding boxes for each label in parallel. The computational structure for performing this is shown on the left in Figure 5.18. This has a separate processor for each label, building the bounding box for that label. A multiplexer is required on the output to access the data for a particular label for downstream processing. Careful analysis reveals that it is not the processor that needs to be separate for each label, because any given pixel input will have only one label, but rather it is the data that must be maintained separately for each label. Only a single bounding box processor is needed and this may be multiplexed between the different labels. Equivalently, the data registers for each label need to be multiplexed for the single instance of the bounding box processor. Multiplexing more than a few data registers can get quite expensive. Where the memory bandwidth allows, an efficient alternative is to use a RAM rather than registers because the RAM addressing effectively provides the multiplexing for free.

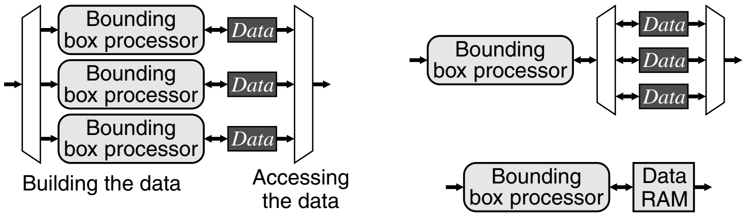


Figure 5.18 Sharing data with a resource. Left: processing each label with a separate bounding box processor; right: sharing a single processor but with separate data registers, which can be efficiently implemented using a RAM.

5.3.1.1 Busses

The example in Figure 5.17 highlights one of the problems with using multiplexers to connect multiple outputs to multiple inputs. Each input requires a multiplexer, with the width of the multiplexer dependent on the number of outputs connected. The complexity is therefore proportional to the product of the number of inputs and the number of outputs. This does not scale well as a new input or output is added.

Such growth in complexity may be overcome by using a bus as shown in the top of Figure 5.19. With a bus, all of the inputs are connected directly to the bus, while the outputs are connected via a tri-state buffer. Enabling an output connects it to the bus, while disabling it disconnects it. The logic scales well, with each new device requiring only a tri-state buffer to connect.

This reduction in logic does not come without a cost. Firstly, with multiplexers, each connection is independent of the others. A bus, however, is shared amongst all of the devices connected to it. While a bus does allow any output to be connected to any input (in fact it connects to all of the inputs), only one output may be connected at a time. If multiple outputs attempt to connect simultaneously to the bus, this could potentially damage the FPGA (if the outputs were in opposite states). Therefore, an additional hidden cost is the arbitration logic required to control access to the bus.

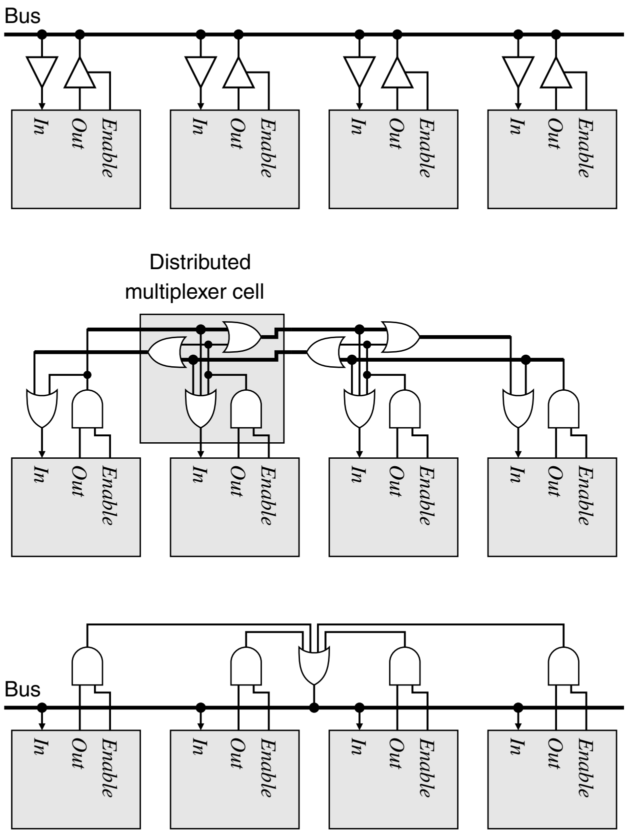


Figure 5.19 Bus structures. Top: connecting to the bus with tri-state buffers; middle: implementing a bus as a distributed multiplexer; bottom: implementing a bus as a shared multiplexer. (Sedcole, 2006; reproduced by permission of P. Sedcole.)

Unfortunately, not many FPGAs have an internal bus structure with the associated tri-state buffers. An equivalent functionality may be constructed using the FPGAs logic resources (Sedcole, 2006), as shown in the middle of Figure 5.19. The bus is replaced by two signals, one propagating to the right and the other to the left. An enabled output from a device is ORed onto both signals, to connect it to devices in both directions. An input is taken as the OR of both signals. Such an arrangement is effectively a distributed multiplexer; if the expression on the input is expanded, it is simply the OR of the enabled outputs. If multiple outputs are enabled, the signal on all of the inputs may not be what was intended, but it will not damage the FPGA.

The disadvantage of the distributed multiplexer is that it requires the devices to be in a particular physical order on the FPGA. This was not a problem where it was initially proposed – for connecting dynamically reconfigurable functional blocks to a design (Sedcole, 2006). There the logic for each block connecting to the 'bus' was constrained to a particular region of the FPGA. For general use, a bus may be simply constructed using a shared multiplexer, as shown in the bottom of Figure 5.19.

When a bus is used to communicate between processes, it may be necessary for each process to have a FIFO buffer on its both its input and output (Sedcole, 2006). Appropriately sized buffers reduce the time lost when stalled waiting for the bus. This becomes particularly important with high bus use to cope with time varying demand while maintaining the required average throughput for each process.

5.3.2 Resource Controllers

There are potential resource conflicts whenever a resource is shared between several concurrent blocks. Any task that requires exclusive access to the resource can cause a conflict. Some common examples are:

- Writing to a register: Only one block may write to a register in any clock cycle. The output of the register, though, is available to multiple blocks simultaneously.
- Accessing memory, whether reading or writing: A multiport memory may only have one access per port. The different ports may write to different memory locations, writing to the same location from multiple ports will result in undefined behaviour.
- Sending data to a shared function block for processing: Note that if the block is pipelined, it may be processing several items of data simultaneously, each at different stages within the pipeline.
- Writing to a bus: While a bus may be used to connect to another resource, the bus is a resource in its own right, since only one output may write to the bus at a time.

Since the conflicts may occur between quite different parts of the design that may be executing independently, none of these conflicts can be detected by a compiler. The compiler can only detect potential conflicts through the resource being connected in multiple places and, as a result, it builds the multiplexer needed to connect to the shared resource. The conflicts are run-time events; therefore, they must be detected and resolved as they occur.

A good design principle is to reduce the coupling as much as possible between parallel sections of an algorithm. This will tend to reduce the number of unnecessary conflicts. However, some coupling between sections is inevitable, especially where expensive resources are shared.

One way of systematically managing the potential conflicts is to encapsulate the resource with a resource manager (Vanmeerbeeck *et al.*, 2001) and access the resource via a set of predefined interfaces, as shown in Figure 5.20. If necessary, the interface should include handshaking signals to indicate that a request to access the resource is granted. Combining the resource with its manager in this way makes the design easier to maintain, as it is only the manager process that interacts directly with the resource. It also makes it easier to change the conflict resolution strategy because it is in one place, rather than distributed throughout the code wherever the resource is accessed.

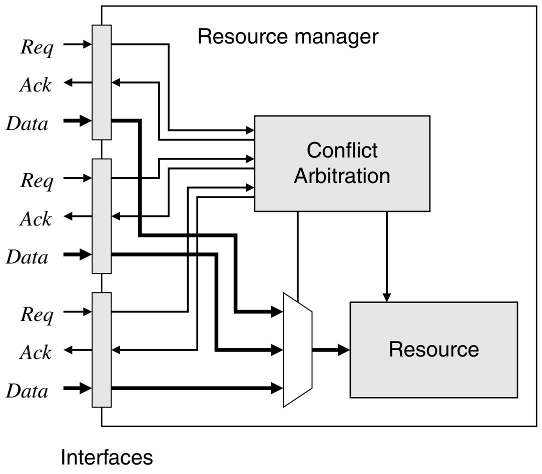


Figure 5.20 Encapsulating a resource with a resource manager simplifies the design.

For example, Figure 5.21 shows a possible implementation of the double-buffered bank switching introduced in Figure 5.8. In this example, the conflict resolution mechanism is scheduled separate access, so no handshaking is required. The *State* variable controls which RAM bank is being written to and which is being read from at any one time. The specific banks are hidden from the tasks performing the access. The upstream task just writes to the write port as though it was ordinary memory, and the resource controller directs the request to the appropriate RAM bank. Similarly, when the downstream task reads from the buffer, it is directed to the other bank. The main difference between access via the controller and direct access to a dedicated RAM bank is the slight increase in propagation delay through the multiplexers.

A CSP model (Hoare, 1985) is another approach for managing the communication between the resource controller and the processes that require access to the resource. This effectively uses a channel for each read and write port, with the process blocking until the communication can take place. The blocking is achieved in hardware by using a handshake signal to indicate that the data has been transferred.

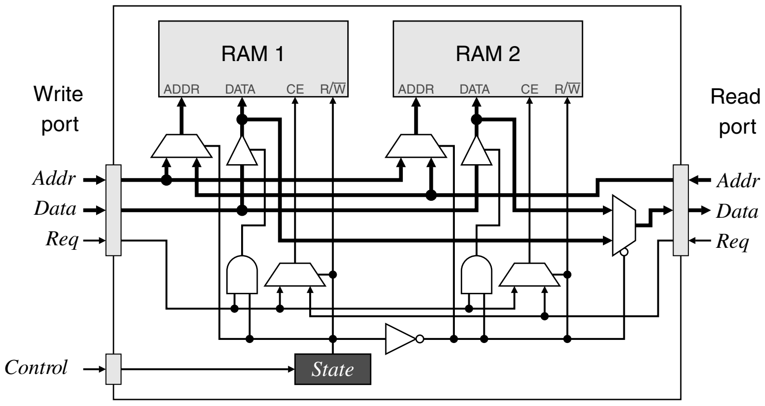


Figure 5.21 Resource manager for bank switched memories.

5.3.2.1 Conflict Arbitration

A conflict occurs when multiple processes attempt to access a resource at the same time. If there is no conflict resolution in place, the default will be for all of the processes to access the resource simultaneously. The multiplexers used to share the resource will prevent any physical damage to the FPGA (although care needs to be taken with busses). However, the multiple selections will be combined in a manner that depends on the multiplexer design (Figure 4.7). Generally, this will have undesired results and could be a cause of bugs within the algorithm that could be difficult to locate.

With conflict resolution, only one process will be granted access to the resource and the other processes will be denied access. These processes must wait for the resource to become free before continuing. While it may be possible to perform some other task while waiting, this can be hard to implement in practise. In some circumstances, it may be possible to queue the access (for example using a FIFO buffer) and continue if processing time is critical. It is more usual, however, simply to stall or block the process until the resource becomes free.

The simplest form of conflict resolution is scheduled access. With this, conflict is avoided by designing the algorithm such that the accesses are separated or scheduled to occur at different times. In the bank-switch example above, accesses to each bank of RAM are separated in time. While data is being stored in one bank, it is being consumed from the other. At the end of the frame, the control signal is toggled to switch the roles of the banks. When stream processing is used, a shared resource may be available to other processes during the horizontal or vertical blanking periods. If multiple processes need to use the resource, simple synchronisation mechanisms may be used to pass access from one process to the next.

Scheduled access can also be employed at a fine grain by using a multiphase pipeline. Consider normalising the colour components of YCbCr data (Johnston *et al.*, 2005b) (on the left in Figure 5.22): each of *Cb* and *Cr* are divided by *Y* to normalise for intensity as part of a stream processing pipeline:

$$\begin{aligned} Cb_n &= Cb/Y \\ Cr_n &= Cr/Y \end{aligned} \tag{5.3}$$

The divider is a relatively expensive resource, so sharing its access may be desirable. Since the pixel clock is often slow compared with the capabilities of modern FPGAs, the clock frequency can be doubled, creating a two-phase pipeline. New data is then presented at the input (and is available at the output) on every second clock cycle. However, with two clock cycles available to process every pixel, the divider (and other resources) can now be shared, as shown on the right in Figure 5.22. In one phase, *Cb* is divided by *Y* and stored in Cb_n, and in the other phase *Cr* is similarly processed. If necessary, the divider can be pipelined to meet the tighter timing constraints, since each of *Cb* and *Cr* are being processed in separate clock cycles. It is only necessary to use the appropriate phase of clock to latch the results into the corresponding output register.

When the processes are able to be synchronised, scheduled access works well and is easy to implement. However, scheduling requires knowing in advance when the accesses will occur. It is difficult to schedule asynchronous accesses (asynchronous in the sense that they can occur in any clock cycle; the underlying resources must be in one clock domain so are still driven by a common clock).

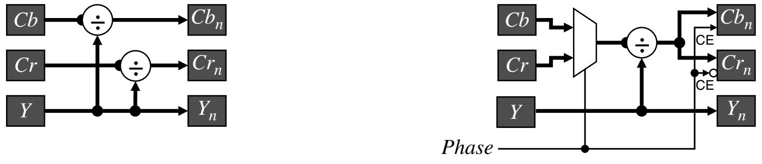
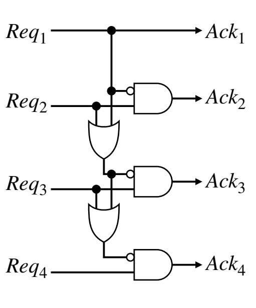


Figure 5.22 Sharing a divider in a two-phase pipeline.



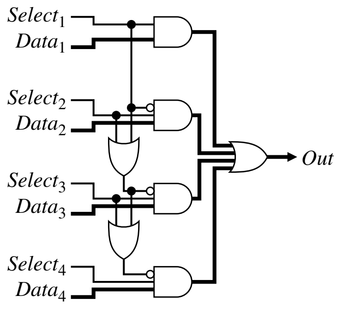


Figure 5.23 Priority-based conflict resolution. Left: a simple prioritised acknowledgement; right: a prioritised multiplexer.

With asynchronous accesses, it is necessary to build hardware to resolve the conflict. The type of hardware required depends on the duration of the access. If the duration is only a single clock cycle, then relatively simple conflict resolution such as a simple prioritised acknowledgement (left panel of Figure 5.23) may be used. With a prioritised access, different parts of the algorithm have different priority with respect to accessing a resource. The highest priority section is guaranteed access, while a lower priority section can only gain access if the resource is not being used by a higher priority section. That is:

$$Ack_i = Req_i \wedge \overline{\left(\bigvee_{j=1}^{i-1} Req_j \right)} \tag{5.4}$$

where a smaller subscript corresponds to a higher priority. The implementation shown in Figure 5.23 is suitable for providing a clock enable signal for a register or some other circuit where the output is sampled on the clock. This is because the propagation delays of the circuit generating the *Req* signals mean that there can be hazards on the *Ack* outputs. If such hazards must be avoided, then the *Ack* outputs can be registered, delaying the acknowledgement until the following clock cycle.

One example of a prioritised access is the prioritised multiplexer shown in the right panel of Figure 5.23. The output will correspond to the highest priority selected input. This could be used, for example, in selecting the pixel to display. Only one pixel may be displayed at any location and a standard multiplexer would combine the pixel values if multiple inputs were selected. By assigning each source a priority, only a single value will be selected. This effectively creates a series of layers or overlays on the display, with the background (lower priority) layers only being visible when a foreground layer is not selected.

Another example of prioritised access is lookahead caching. Consider a streamed operation that requires zero, one or two accesses to memory in a clock cycle depending on the local context. Also assume that the average number of memory accesses per clock cycle is less than or equal to one. The operation may be split into two processes: one that produces the streamed output at one pixel per clock cycle, and one that looks ahead to where two accesses are required and preloads one of those values into a cache. The output process will have the higher priority for memory access because it must produce its streamed output on time. The cache process, however, can access memory whenever the output process does not require it.

If the conflict duration is for many clock cycles, then a semaphore is required to lock out other tasks while one task has access. Equation 5.4 then becomes:

$$Ack_i = Req_i \wedge \overline{\left(\bigvee_{j=1}^{i-1} Req_j \right)} \wedge \left(\overline{Sema} \vee Release \right) \tag{5.5}$$

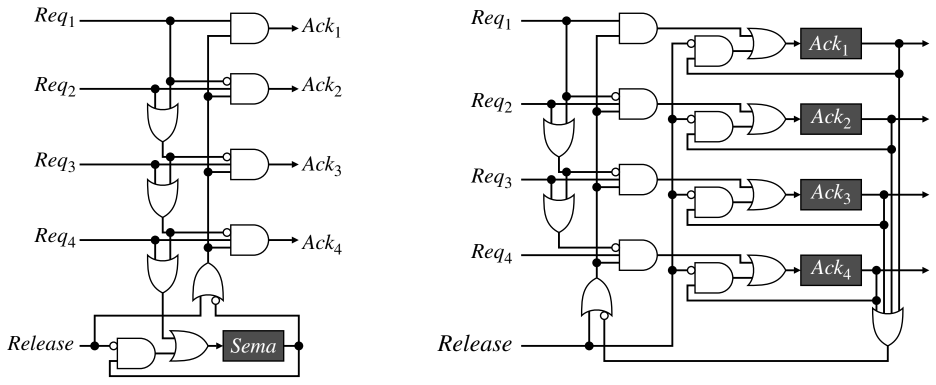


Figure 5.24 A prioritised semaphore. Left: a single semaphore flag; right: a separate flag for each task, to indicate which task currently has the semaphore.

where the first two terms determine the prioritised input as before. The third term selects the prioritised input if either the semaphore, *Sema*, is not set or when the semaphore is released. In the latter case, the highest priority waiting task is then acknowledged. *Sema* is effectively a flag that indicates whether a resource is currently being used (on the left in Figure 5.24). Any request will set the semaphore, which will remain set until released. The new value of the semaphore, *Sema*′, is:

$$Sema' = \left(\bigvee_{j=1}^{N} Req_j\right) \vee \left(Sema \wedge \overline{Release}\right) \tag{5.6}$$

When a request is acknowledged, the semaphore is set, which prevents further requests from being acknowledged. Note that this also means that the *Ack* output only goes high for one clock cycle to acknowledge the request. Again, with this circuit there may be hazards on the *Ack* outputs, which may be removed if necessary by adding a register on each output.

If it is necessary to know which task has the semaphore, one method is to hold the corresponding *Ack* output high while that task has access. This requires a separate flag for each task, as shown on the right in Figure 5.24, where each flag is set by:

$$Ack_i{}' = \left(Req_i \wedge \overline{\left(\bigvee_{j=1}^{i-1} Req_i\right)} \wedge \left(\overline{\bigvee_{j-1}^{N} Ack_j} \vee Release\right)\right) \vee \left(Ack_i \wedge \overline{Release}\right) \tag{5.7}$$

This works on the same principle as before, with any of the semaphores blocking the requests until the *Release* input is asserted. With this circuit, once a lower priority task gains the semaphore, it will block a high priority task. If necessary, it may be modified further to enable a high priority task to pre-empt a lower priority task:

$$Ack_i{}' = \overline{\bigvee_{j=1}^{i-1} Req_i} \wedge \left(\left(Req_i \wedge \overline{\bigvee_{j=1}^{i-1} Ack_j}\right) \vee (Req_i \wedge Release) \vee \left(Ack_i \wedge \overline{Release}\right)\right) \tag{5.8}$$

Care must be taken when using semaphores to prevent deadlock. A deadlock may occur when the following four conditions are all true (Coffman *et al.*, 1971):

- Tasks claim exclusive control of the resources they require.
- Tasks hold resources already allocated while waiting for additional resources.

- Resources cannot be forcibly removed by other tasks. They must be released by tasks using the resource.
- There is a circular chain of tasks, where each task holds resources that are being requested by another task.

Deadlock may be prevented by ensuring that any one of these conditions is false. For complex systems, this is perhaps easiest achieved by requiring tasks to request all of the resources they require simultaneously. If not all of the resources are available, then none of the resources are allocated. This requires modification of semaphore circuit to couple the requests. It still does not necessarily avoid deadlocks unless care is taken with the priority ordering. For example, consider two tasks that both require the same two resources simultaneously. If the priority order is different for the two resources, then, because of the prioritisation, each task will prevent the other from gaining one of the resources, resulting in a deadlock.

5.3.3 Reconfigurability

In most applications, a single configuration is loaded onto the FPGA prior to executing the application. This configuration is retained, unchanged, until the application is completed. Such systems are called *compile-time reconfigurable*, because the functionality of the entire system is determined at compile time and remains unchanged for the duration of the application (Hadley and Hutchings, 1995).

Sometimes, however, there are insufficient resources on the FPGA to hold all of a design. Since the FPGA is programmable, the hardware can be reprogrammed or reconfigured dynamically, while still executing an application. Such systems are termed *run-time reconfigurable* (Hadley and Hutchings, 1995). This requires the algorithm to be split into multiple sequential sections, where the only link between the sections is the partially processed data. Since the configuration data for an FPGA includes the initial values of any registers and the contents of on-chip memory, any data that must be maintained from the old configuration to the new must be stored off-chip (Villasenor *et al.*, 1996) to prevent it from being overwritten. Unfortunately, reconfiguring the entire FPGA can take a significant time– large FPGAs typically have reconfiguration times of tens to hundreds of milliseconds. This latency must be taken into account in any application. With smaller FPGAs, the reconfiguration time may be acceptable (Villasenor *et al.*, 1995; 1996). The latency is usually less important when switching from one application to another, for example between an application used for tuning the image processing algorithm parameters and the application that runs the actual algorithm.

The time required to reconfigure the whole FPGA has spurred research into *partially reconfigurable* systems, where only a part of the FPGA is reprogrammed. Partial reconfiguration has two advantages over reconfiguring the entire FPGA. The first is that the configuration data for part of the FPGA is smaller than that for the whole chip, so will usually take significantly less time to load than a complete configuration file. It also has the advantage that the rest of a design can continue to operate while part of the design is being reconfigured. Not all FPGAs are partially reconfigurable; the bitstream must be able to split into a series of frames, with each frame including an address that specifies its target location within the FPGA. At present only some of the devices with the Xilinx range and the older Atmel AT40K support partial reconfiguration.

Partial reconfiguration requires a modular design, as shown in Figure 5.25, with a static core and a set of dynamic modules (Mesquita *et al.*, 2003). The static core usually deals with interface and control functions and holds the partial results, while the reconfigurable modules typically provide the computation. The dynamic modules can be loaded or changed as necessary at run-time, a little like overlays or dynamic link libraries in a software system (Horta *et al.*, 2002). Only the portion of the FPGA being loaded needs to be stopped during reconfiguration; the rest of the design can continue executing.

Between the dynamic modules and the static core, a standard interface must be designed to enable each dynamic module to communicate with the core or with other modules (Sedcole, 2006).

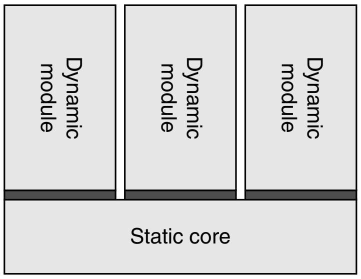


Figure 5.25 Designing for partial reconfigurability. A set of dynamic modules 'plug in' to the static core.

Common communication mechanisms are to use a shared bus (Hagemeyer *et al.*, 2007) or predefined pipeline ports connecting from one module to the next (Sedcole *et al.*, 2003). Both need to be in standard locations to enable them to connect correctly when the module is loaded.

Two types of partial reconfiguration have been considered in the literature. One restricts the reconfigurable modules to being a fixed size that can be mapped to a small number of predefined locations. The other allows modules to be of variable size and be placed more flexibly within the reconfigurable region. The more flexible placement introduces two problems. Firstly, the heterogeneity of resources restricts possible locations that a dynamic module may be loaded on the FPGA. The control algorithm must not only search for a space sufficiently large to hold a module, but the space must also have the required resources in the right place. Secondly, with the loading and unloading of modules, it is possible for the free space to become fragmented. To overcome this, it is necessary to make the modules dynamically relocatable (Koester *et al.*, 2007). To relocate a module (Dumitriu *et al.*, 2009) it must first be stopped or stalled, then disconnected from the core. The current state (registers and memory) must be read back from the FPGA, before writing the module (including its state) to a new location, where it is reconnected to the core, and restarted.

Partial reconfigurability allows adaptive algorithms. Consider an intelligent surveillance camera tracking vehicles on a busy road. The lighting and conditions in the scene can vary significantly with time, requiring different algorithms for optimal operation (Sedcole *et al.*, 2007). These different subalgorithms may be designed and implemented as modules. By having the part of the algorithm that monitors the conditions in the static core, it can select the most appropriate modules and load them into the FPGA based on the context.

Simmler *et al.* (2000) have taken this one step further. Rather than consider switching between tasks that are part of the same application, they propose switching between tasks associated with different applications running on the FPGA. Multitasking requires suspending one task and resuming a previously suspended task. The state of a task is represented by the contents of the registers and memory contained within the FPGA (or dynamic module in the case of partial reconfigurability). Therefore, the configuration must be read back from the FPGA to record the state or context. To accurately record the state, the task must be suspended while reading the configuration and while reloading it. However, not all FPGAs support readback, and therefore not all will support dynamic task switching.

At present, the tools to support partial dynamic configuration are quite low level. Each module must be developed with quite tight placement constraints to keep the module within a restricted region and to ensure that the communication interfaces are in defined locations. The system core also requires similar placement constraints to keep out of the reconfigurable region and to fix the communication interfaces. Xilinx provides an internal configuration access port (ICAP) to enable an embedded CPU to dynamically reconfigure part of the FPGA (Blodget *et al.*, 2003). This can also be accessed directly by logic for partial reconfiguration without requiring a CPU (Lai and Diessel, 2009).

Regardless of whether full or partial reconfigurability is used, run-time reconfigurability can be used to extend the resources available on the FPGA. This can allow a smaller FPGA to be used for the application, reducing both the cost and power requirements.

5.4 Computational Techniques

An FPGA implementation is amenable to a wider range of computational techniques than a software implementation. Some of the computational considerations and techniques are reviewed in this section.

5.4.1 Number Systems

In any image processing computation, the numerical values of the pixels, coordinates, features and other numerical data must be represented and manipulated. There is a wide range of number systems that can be used. A full discussion of the range of number systems and associated logic circuits for performing basic arithmetic operations is beyond the scope of this book. The interested reader should refer to one of the many good books available on computer arithmetic (for example Parhami, 2000).

Arguably, the most common representation is based on binary or base-two numbers. As shown in Figure 5.26, a positive binary integer, B, is represented by a set of N binary digits, b_i, where each digit represents a successive power of two. The numerical value of the number is then:

$$B = \sum_{i=0}^{N-1} b_i 2^i \tag{5.9}$$

In most software representations, N is fixed by the underlying computer architecture at 8, 16, 32, or 64. On an FPGA, however, there is no necessity to constrain the representation to these widths. In fact, the hardware requirements can often be significantly reduced by tailoring the width of the numbers to the range of values that need to be represented (Constantinides *et al.*, 2001). For example, for a 640×480 image the row and column coordinates can be represented by 9-bit and 10-bit integers respectively.

When representing signed integers, there are three common methods. The first is the *sign-magnitude* representation. This uses a sign bit to indicate whether the number is positive or negative, and the remaining $N-1$ bits to represent the magnitude (absolute value) of the number. With this representation, addition and subtraction are slightly complicated because the actual operation performed will depend on the combination of sign bits and the relative magnitudes of the numbers. Multiplication and division are simpler, as the sign and magnitude bits can be operated on separately. The range of numbers represented is from $-(2^{N-1}-1)$ to $2^{N-1}-1$. Note that there are two representations for zero: -0 and $+0$.

The *two's complement* representation is the format most commonly used by computers. Numbers in the range -2^{N-1} to $-2^{N-1}-1$ are represented modulo 2^N, which maps the negative numbers into the range 2^{N-1} to -2^N-1. This is equivalent to giving the most significant bit a negative weight:

$$B = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i \tag{5.10}$$

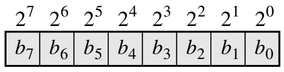


Figure 5.26 Binary number representation.

Therefore, negative numbers can be identified by a 1 in the most significant bit. Addition and subtraction of two's complement numbers is exactly the same as for unsigned numbers, which is why the two's complement representation is so attractive. Multiplication is a little more complex; it has to take into account that the most significant bit is negative.

The third representation is *offset binary*, where an offset or bias is added to all of the numbers:

$$B = Offset + \sum_{i=0}^{N-1} b_i 2^i \tag{5.11}$$

where *Offset* is usually (but not always) 2^{N-1}. Offset binary is commonly used in image processing to display negative pixel values. It is not possible to display negative intensities, so offsetting zero to mid-grey shows negative values as darker and positive values as lighter. Offset binary is also frequently used with A/D converters to enable them to handle negative voltages. Most A/D converters are unipolar and will only convert positive voltages (or currents). By adding a bias to a bipolar signal, negative voltages can be made positive enabling them to be successfully converted. A limitation with offset binary is that the offset needs to be taken into account when performing any arithmetic.

5.4.1.1 Real Numbers

One limitation of integer representations is their inability to represent real numbers. This may be overcome by scaling the integer by a fraction, so that the integer represents the number of fractions. If the fraction is made a negative power of two, for example 2^{-k}, then the result is *fixed-point* notation. The resulting number is then:

$$B_{FP} = B \times 2^{-k} = \pm b_{N-1} 2^{N-1-k} + \sum_{i=0}^{N-2} b_i 2^{i-k} \tag{5.12}$$

where the integer B can be either unsigned or signed (usually two's complement). A convenient shorthand notation for the format of a fixed-point number is U$N.k$ or S$N.k$, where the letter is either U or S to indicate an unsigned or signed N-bit integer is used with k fraction bits. As seen in Figure 5.27, the N-bit integer representation of the fixed-point number corresponds to shifting it k bits to the left.

Most real numbers can only be represented approximately with a fixed-point representation. While increasing the number of bits to the right of the binary point will increase the accuracy of the approximation, many even relatively simple numbers cannot be represented exactly with a finite number of bits. For example, the fraction ⅓ has an infinitely repeating binary representation, 0.01010101010101…, so any fixed-point (or even floating-point) representation will only ever be

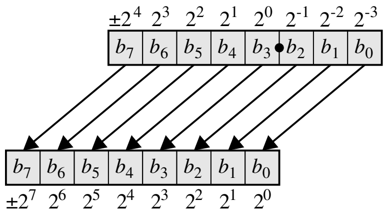


Figure 5.27 Fixed-point number representation with resolution 2^{-3}. Shifting the number 3 bits to the left gives the corresponding integer representation. This number is of type U8.3 or S8.3 depending on whether the weight of the most significant bit was positive or negative respectively.

an approximation. The absolute error of the representation, or resolution, will be determined by the position of the binary point.

The advantage of a fixed-point representation over a more complex floating-point representation is that the location of the binary point is fixed. All arithmetic operations are the same as for integers, apart from aligning the binary point for addition and subtraction. This corresponds to an arithmetic shift, by a fixed number of bits since the location of the binary point is fixed. Such a shift is effectively free in hardware (it is simply a case of routing the signals appropriately).

A problem with fixed-point numbers is that they have limited dynamic range. The fixed location of the binary point fixes the resolution, so while they are able to represent some real numbers, the underlying representation is still as an integer. The same representation is unable to represent both very small and very large numbers. This may be overcome by allowing the location of the binary point to be variable, which requires another parameter. A *floating-point* number therefore has two components: the mantissa or *significand*, which are the binary digits of the number, and the *exponent*, which gives the power of two that the significand is multiplied by, and effectively specifies the position of the binary point.

The floating-point representation is still an approximation to a real number. The absolute error will vary depending on the position of the binary point. However, the relative error of the representation will be constant regardless of the size of the number and is determined by the number of bits in the significand. Increasing the number of bits in the significand will increase the accuracy of the approximation. The relative error of the representation should not be confused with the relative error after an operation. Consider, for example, subtracting two very similar numbers. Many of the significant bits will cancel, giving a much larger relative error for the difference.

Even when a number can be represented exactly, in general it will not be unique; the number six, for example, can be represented with a 4-bit significand as $0.011 \times 2^4 = 0.110 \times 2^3 = 1.100 \times 2^2$. To give a unique representation, a *normalised floating-point* number has a 1 in the most significant bit of the significand. Requiring numbers to be normalised will maximise the number of significant digits within a number, and consequently minimise any errors in the approximation. However, a big disadvantage is that there is no representation for zero (Goldberg, 1991). It also requires a sign-magnitude representation of the significand because the two's complement representation of a negative number would have a leading 1 regardless of the position of the binary point.

The normalised representation therefore has the significand greater than or equal to 1, and less than 2. Since the leading digit is always 1, it can be assumed, giving an extra bit of resolution for free. The significand bits therefore represent a binary fraction. The number of bits used for the exponent will determine the dynamic range of numbers (the ratio between the largest and smallest number) that can be represented. While it could potentially be represented using two's complement, an offset binary representation is commonly used because it enables a simpler unsigned integer comparison to be used for comparing magnitudes. The common representation is shown in Figure 5.28, giving the floating-point number, F, as

$$F = (-1)^s \times 2^{Exponent} \times \left(1 + \sum_{j=1}^{N_b} b_j 2^{-j}\right)$$

where (5.13)

$$Exponent = \sum_{j=0}^{N_e-1} e_j 2^j - \left(2^{N_e-1} - 1\right)$$

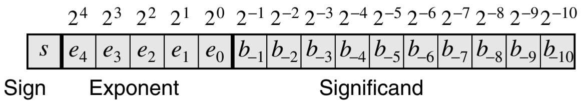


Figure 5.28 Floating-point representation.

N_b is the number of bits in the significant and N_e is the number of bits in the exponent. The term subtracted from the exponent is the standard offset or bias that balances the dynamic range so that both a number and its reciprocal may be represented.

The IEEE standard for floating-point numbers (IEEE, 2008) enables reproducibility of floating-point calculations between machines. It does this by specifying the representation (how the bits are encoded) making the floating-point numbers portable. For example, a single precision (32 bit) floating-point number uses 1 bit for the sign, 8 bits for the exponent with an offset of 127, and 24 bits (including the implicit leading 1) for the significand. Two exponent values are reserved for special cases. All zeros are used to represent denormalised numbers (numbers smaller than the smallest normalised number), including zero. All ones are used to represent overflow, infinity and not a number (NaN – for example the result of 0/0 or $\sqrt{-1}$). The standard also specifies how operations should be performed, so that the same calculation should give the same result regardless of the processor performing the computation.

When performing an image processing calculation on an FPGA, the size of the significand and exponent can be tailored to the accuracy of the calculation being performed, to reduce the hardware required. The consequence of this, however, is that the result will, in general, be different from performing the same calculation in software using the IEEE standard. As with fixed-point, this requires a careful analysis of the approximation errors to ensure that the results are meaningful.

Multiplication and division are the most straightforward operations for floating-point numbers. For multiplication, the significands are multiplied and the exponents added. If the resulting significand is greater than two, renormalisation requires shifting it one bit to the right and incrementing the exponent. The product will have more digits than can be represented, so the result must also be rounded to the required number of digits. The sum of the two exponents will include two offsets, so one must be subtracted off. The sign of the output is the exclusive-OR of the input sign bits. Additional logic is required to detect underflow, overflow and handle other error conditions such as processing infinities and NaNs. Division is similar except the significands are divided and the exponents subtracted, and renormalisation may involve a left shift.

Addition and subtraction are more complex. As with the sign-magnitude representation, the actual operation performed will also depend on the signs of the inputs. The numbers must be aligned so that the exponents are the same. This will involve shifting the significand of the smaller number to the right by the difference in the exponents. When implemented on an FPGA, such a shift is either slow (performed as a series of smaller shifts) or expensive (implemented as a large number of wide multiplexers). If available, a multiplier block may also be used to perform the shift (Gigliotti, 2004). It is necessary to keep an extra bit in the shifted smaller number (a guard bit) to reduce the error introduced by the operation (Goldberg, 1991). The significands are then added or subtracted depending on the operation and the result renormalised. If two similar numbers are subtracted, many of the most significant bits may cancel. Renormalisation therefore requires determining the position of the leftmost one, moving this to the most significant bit (both operations may be combined by using the prioritised multiplexer shown in Figure 5.24) and adjusting the exponent accordingly. Again, additional logic is required to handle error conditions.

Floating-point operations require significantly more resources than the corresponding fixed-point operations. It is not that floating point operations are that complicated, but managing the exceptions requires a large proportion of the logic, especially if compliance to the IEEE standard is required. For this reason, most developers prefer to use fixed-point unless the wider dynamic range of floating-point is needed for an application. Where dynamic range is required, one compromise that has been proposed is to use two fixed-point ranges (Ewe *et al.*, 2004, 2005).

5.4.1.2 Logarithmic Number System

An alternative to floating-point is the logarithmic number system, first proposed by Kingsbury and Rayner (1971). This achieves a wide dynamic range by representing a number by its logarithm. A sign-magnitude representation is required because a real logarithm is not defined for negative numbers. The logarithm of zero is also not defined; this requires either a separate flag, a special code or approximating zero by the smallest available number.

The logarithms are usually base-two, although using another base will simply scale the logarithm. The logarithm is represented using fixed-point notation, with N_I bits for the integer component and N_F bits for the fractional component. The integer component will be the same value as the exponent for a normalised floating-point representation (apart from the offset).

The attraction of the logarithmic number system is that multiplication and division are simple additions and subtractions. If $L_A = \log_2 A$ and $L_B = \log_2 B$ then multiplication becomes:

$$\log_2(A \times B) = \log_2 A + \log_2 B = L_A + L_B \tag{5.14}$$

Addition and subtraction, however, are more complex. Without loss of generality, assume that $A > B$. Then:

$$\begin{aligned} \log_2(A \pm B) &= \log_2 A\left(1 \pm \frac{B}{A}\right) \\ &= \log_2 A + \log_2\left(1 \pm 2^{\log_2(B/A)}\right) \\ &= L_A + \log_2\left(1 \pm 2^{L_B - L_A}\right) \end{aligned} \tag{5.15}$$

This requires two functions to represent the second term, one for addition:

$$\begin{aligned} &\log_2(A + B) = L_A + f_{Add}(L_B - L_A) \\ \text{where} \quad &f_{Add}(r) = \log_2(1 + 2^r) \end{aligned} \tag{5.16}$$

and one for subtraction:

$$\begin{aligned} &\log_2(A - B) = L_A + f_{Sub}(L_B - L_A) \\ \text{where} \quad &f_{Sub}(r) = \log_2(1 - 2^r) \end{aligned} \tag{5.17}$$

These functions can either be evaluated directly (by calculating 2^r, performing the addition or subtraction, then taking the logarithm), by using table lookup or by piece-wise polynomial approximation (Fu *et al.*, 2006). Since $A > B$ then $2^{L_B - L_A}$ will be between 0 and 1. Therefore, f_{Add} will be relatively well behaved, although f_{Sub} is more difficult as $\log_2 0$ is not defined. The size of the tables depends on N_F, and by cleverly using interpolation can be made of reasonable size (Lewis, 1990).

As would be expected, if the computation involves more multiplications and divisions than additions and subtractions then the logarithmic number representation has advantages over floating-point, in terms of both resource requirements and speed (Haselman, 2005; Haselman *et al.*, 2005).

5.4.1.3 Residue Number Systems

A residue number system is a system for reducing the complexity of integer arithmetic by using modulo arithmetic. A set of k co-prime moduli, $\{M_1, M_2, \ldots M_k\}$, is used and an integer, X, is represented by its residues or remainders with respect to each modulus: $\{x_1, x_2, \ldots x_k\}$. That is:

$$X = n_i M_i + x_i \tag{5.18}$$

where each n_i is an integer and each x_i is a positive integer. Negative values of X may be handled just as easily because modulo arithmetic is used. The representation is unique for:

$$0 \leq X < \prod_{i=1}^{k} M_i$$

or

$$-\frac{1}{2}\prod_{i=1}^{k} M_i \leq X < \frac{1}{2}\prod_{i=1}^{k} M_i \tag{5.19}$$

Addition, subtraction and multiplication are performed respectively by using modulo addition, subtraction and multiplication independently on each of the residues. This makes the residue number system attractive for FPGA implementation (Tomczak, 2006) because it replaces a single wide integer with a set of smaller integers, and the calculations on each of the residues may be implemented in parallel. This can reduce both the propagation delay and logic footprint compared with using wide adders and multipliers.

A disadvantage of the residue number system, however, is that division is impractical. The main application is in digital filtering, which consists of fixed-point multiplications and additions, so the lack of division is not serious.

To be useful, it is necessary to be able to convert from the residue number system back to binary. This is performed using the Chinese remainder theorem or variants (Wang, 1998). A common choice of moduli is $\{2^n-1, 2^n, 2^n+1\}$ (Wang *et al.*, 2002), for which the logic required to perform modulo arithmetic is only marginally more complex than for performing standard arithmetic. For this set, there are also efficient techniques for converting between binary and the equivalent residue number representation (Fu *et al.*, 2008).

5.4.1.4 Redundant Representations

Arithmetic operations are slow compared to simple logic operations because of the propagation delay associated with the carry from one digit to the next. Another technique that can be used to speed arithmetic is the use of a redundant representation. A *redundant number system* is one that has more digits than the size of the radix (Avizienis, 1961). Standard binary (radix-2) uses the digits $b_i \in \{0, 1\}$. A commonly used redundant system is a *signed digit representation*, where $b_i \in \{-1, 0, 1\}$.

With redundancy, carry propagation may be considerably shortened, giving significant speed improvements for additions. A particular signed digit representation that maximises the number of zeroes in the number is the *canonical signed digit* form (Arno and Wheeler, 1993). This has at least one zero between every non-zero digit, so has at least half of its digits zero. The carry from adding two canonical signed digit numbers will propagate at most one bit. Minimising the number of digits will also reduce the number of partial products when multiplying numbers (Koc and Johnson, 1994).

The disadvantage of redundant representations is that the additional digits require wider signals (at least two bits are required for the radix-2 signed digit representation), wider registers for storing the numbers and, usually, more logic to process the wider signals. While addition of two canonical signed digit numbers will give a signed digit representation, additional logic and propagation delay may be required to convert the result back to canonical form. With any redundant representation, additional logic is required to convert any output back to normal binary.

The signed digit idea can be extended further, for example to $b_i \in \{-2, -1, 0, 1, 2\}$ (Sam and Gupta, 1990), to further improve the performance of multipliers in particular. The trade-off is primarily between incremental performance improvements and rapidly increasing circuit complexity. An asymmetric signed digit representation using the digits $b_i \in \{-1, 0, 1, 3\}$ has been proposed for reducing the number of non-zero digits (Kamp *et al.*, 2006). It also has the advantage that each digit only requires two bits to represent; a single 4-LUT is sufficient to perform an arbitrary operation on two digits (Kamp *et al.*, 2006).

Signed digits form the basis of the Booth multiplication algorithm (Booth, 1951), which uses a combination of addition ($+1$) and subtraction (-1) and nothing (0) to reduce the number of additions required. It is also used by non-restoring division algorithms to avoid having to restore the result when a test subtraction gives a negative result (Bailey, 2006). Redundancy is exploited by SRT division techniques (Robertson, 1958; Tocher, 1958) by giving an overlap when selecting the correct quotient digit. The overlap means that the quotient digit may be determined from an approximation of the partial remainder and does not need to wait for the carry to propagate fully. Similar techniques have also been applied to shift and add methods for the calculation of logarithms and exponentials (Nielsen and Muller, 1996).

5.4.2 Lookup Tables

Many functions can be quite expensive to calculate. In some circumstances, an alternative is to precalculate the function for the range of expected inputs and store the results in a table. Then, rather than calculate the function, the input is simply looked up in the table, which will return the result.

There are two primary instances when the use of lookup tables can be particularly valuable. The first is when the latency of performing the calculation exceeds the available time constraint. Using a lookup table requires a single clock cycle for the memory access regardless of the complexity of the function. The second is when the complexity of the calculation is such that excessive resources are used on the FPGA. A lookup table only requires a memory sufficiently large to provide an accurate representation of the function.

There are two measures commonly used for assessing the accuracy of the representation. Let $f(x)$ be the function that is being approximated and $\tilde{f}(x)$ the approximation in the domain $a \le x \le b$. The error of the approximation is then:

$$\varepsilon(x) = \tilde{f}(x) - f(x) \tag{5.20}$$

The total squared error between the function and its approximation is then:

$$SE = \int_a^b \varepsilon^2(x)dx \tag{5.21}$$

A small squared error is one indication of a good approximation.

One limitation of using the square error to measure the difference between the function and its approximation is that minimising the total error can make the absolute error large at some points. Therefore, it is usually better to minimise the maximum absolute error (de Dinechin and Tisserand, 2001), given by:

$$MaxE = \max_{a \le x \le b} |\varepsilon(x)| \tag{5.22}$$

The difference between minimising the squared error and maximum error is illustrated in Figure 5.29, in particular in the first segment.

With lookup tables, there is a trade-off between accuracy and the size of the table. While lookup tables can be very efficient for low to moderate accuracy, they do not scale well, with the table size increasing exponentially with the width of the input. Practical table sizes require reducing the accuracy of the input. This can be achieved simply by dropping the least significant bits. It is not necessary to round the input, as the content of the table can be set to give the best result over the domain implied by the bits that are kept.

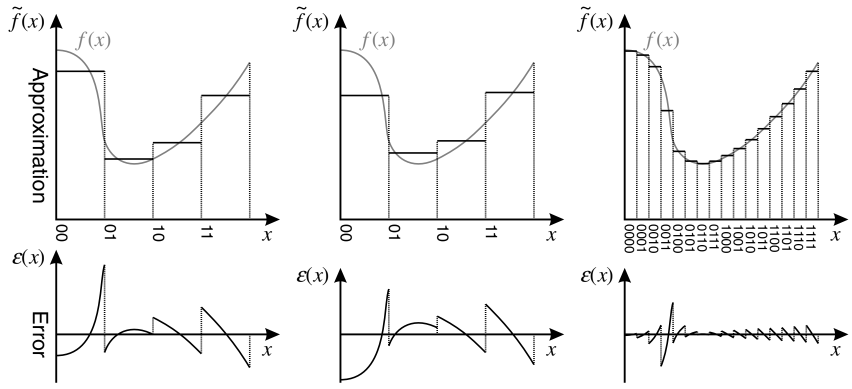


Figure 5.29 Approximating a function by lookup table. Left: minimising the mean square error of the function; centre and right: minimising the maximum error; right: reducing the error by increasing the table size.

This is clearly shown in the centre panel of Figure 5.29, where the table entry is set to the middle of the range for a particular bit combination. The error depends on the slope of $f(x)$, with the maximum slope determining the number of bits required on the input to achieve a particular accuracy on the output.

5.4.2.1 Interpolated Lookup Tables

Observe that the error function consists of a set of approximately linear segments. This can be exploited by using linear interpolation. Rather than each entry in the table being a constant, each entry can be a line segment containing an intercept, c, and slope, m. The most significant bits are looked up to get the intercept and slope for the corresponding segment, with the least significant bits used to scale the slope to interpolate the values:

$$\tilde{f}(x) = c\left[x|_{MSB}\right] + m\left[x|_{MSB}\right] \times x|_{LSB} \tag{5.23}$$

The linear segment approximation and corresponding hardware (Mencer and Luk, 2004) are shown in Figure 5.30. The lookup tables for the intercept and slope have the same addresses. Such tables in parallel are effectively a single table with a wider data path. Note that it is not necessary for the piece-wise linear segments to be continuous; smaller errors can in general be obtained by fitting each segment independently. The slope with the smallest error in the sense of Equation 5.22 will, in general, be the slope between the two endpoints, with the segment offset so that the maximum error within the segment is the negative of the error at the ends, as shown on the right in Figure 5.30. Additional errors (not shown in the figure) will arise from truncation of the least significant bits after the multiplication and also the finite precision of the slopes stored in the table.

An interpolated table effectively trades the size of the lookup table for increased computational resources and latency. The accuracy of the approximation will depend on the curvature or the second derivative of $f(x)$.

Where the curvature is higher, a smaller step size (more bits used to look up the segment) is required to maintain accuracy. However, over much of the table, the accuracy is more than adequate and a larger step size could be used. A variable step size (effectively varying the partition between the most significant bits looked up and the least significant bits used for the interpolation) can significantly reduce the table size

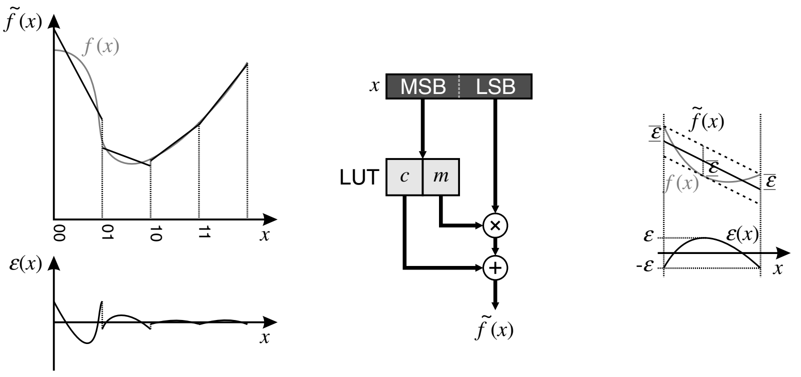


Figure 5.30 Linearly interpolated lookup table. The table contains both the slope and the intercept. Centre: the computational architecture; right: method of construction that minimises the maximum error.

(Lee *et al.*, 2003b; Lachowicz and Pfleiderer, 2008). The cost is a little more logic to manage the variable partitioning. The variable step size approach may be extended to functions of two variables (Nagayama *et al.*, 2008). With two variables, it is essential to partition the space carefully to avoid excessive table sizes, especially for higher precision outputs.

If the approximation is made continuous, as shown in Figure 5.31, and dual-port memory is available for implementing the lookup table, then the slope table is not needed. The current intercept and the next intercept are both looked up using separate ports, and the slope calculated from the difference (Gribbon *et al.*, 2003):

$$\tilde{f}(x) = c\left[x|_{MSB}\right] + \left(c\left[x|_{MSB}+1\right] - c\left[x|_{MSB}\right]\right) \times 2^{-M} \times x|_{LSB} \tag{5.24}$$

where 2^{-M} is the weight of the least significant bit used to address the lookup table. The 2^{-M} is a bit shift, which is free in hardware (hence it is not shown in Figure 5.31). The approximation error may be slightly larger than if each segment is fitted independently, but this will only occur if the second derivative changes sign, as shown between the first two segments in Figure 5.31.

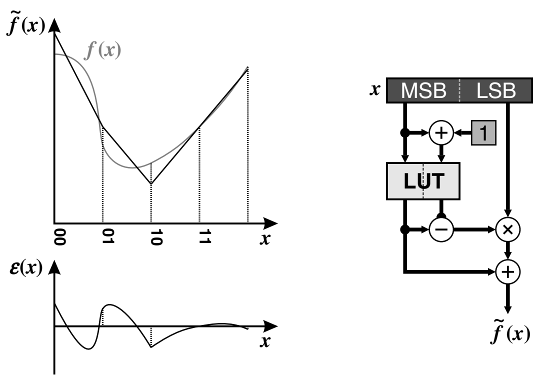


Figure 5.31 Linearly interpolated lookup table using dual-port RAM. The slope is calculated from successive table entries.

For higher accuracy, each segment of the table may be represented by a higher order polynomial approximation. This extends Equation 5.23 to:

$$\tilde{f}(x) = \sum_{i=0}^{d} p_i\left[x|_{MSB}\right] \times \left(x|_{LSB}\right)^i \tag{5.25}$$

where d is the order of the polynomial with coefficients p_i. In practise, to control the accumulation of round-off errors, the polynomial should be evaluated using Horner's method:

$$\tilde{f}(x) = \left(\left(p_d \times x|_{LSB} + p_{d-1}\right) \times x|_{LSB} + \cdots\right) \times x|_{LSB} + p_0 \tag{5.26}$$

For each segment, the optimum polynomial coefficients may be determined using Remez's exchange algorithm (Lee *et al.*, 2003a). As with a linear approximation, the table size can often be significantly reduced by using non-uniform segment widths at the expense of additional decoding logic (Lee *et al.*, 2003a). A higher order polynomial requires fewer segments to fit to a given level of accuracy, but requires more multipliers and, therefore, more resources. It is possible to implement a second order approximation using only a single multiplier by making careful use of lookup tables (Detrey and de Dinechin, 2004).

5.4.2.2 Bipartite and Multipartite Tables

Observe that in Figure 5.30 the slope of the last two segments is very similar. This is often the case, especially with smoothly varying functions. The average slope of the last two segments would also give reasonable errors. If the offsets resulting from multiplying the slope by the least significant bits were stored in a table, the multiplication can be avoided. Unfortunately, if the offsets were stored for every segment, the length of the offset table would be the same as looking up every value. However, since the offsets for adjacent segments are similar (because the slopes are similar) then several of these offsets can be shared between segments, giving a significant reduction in the size of the table.

This is the same concept that is used in tables of logarithms (Hassler and Takagi, 1995). Consider a standard four digit (decimal) table of logarithms. Such a table would require 10^4 or 10 000 entries. Instead, it is arranged as a two-dimensional grid, with 100 rows indexed by the first two digits and 10 columns indexed by the third digit. A second table, which contains offsets, has the same 100 rows indexed by the first two digits, also with 10 columns, but this time indexed by the fourth digit. The same offsets apply to the whole row of the first table. This dual table approach only requires 1000 entries in each table, for a total of 2000 entries. The savings are even greater than first appear, because the numbers in the second table are smaller, so the bit width can be narrower.

This approach to bipartite tables (Das Sarma and Matula, 1995; Hassler and Takagi, 1995) is shown in Figure 5.32. In this example, the eight segments are split into two groups of four for the offset table. Note that at this scale the bipartite method is not particularly effective for the left group, but works well for the right group where the slopes are similar. In general, the input word is split into three approximately equal groups of bits. The most significant two groups are used to index the main table, and define the segments. The most significant and least significant groups are used to give the offsets that apply to the set of segments defined by the first group.

Since no multiplication is used, only an addition, bipartite tables are fast and relatively resource efficient. In addition, since the offsets are usually small, the width of the offset table is usually less than that of the main table. If the value in the main table represents the centre of the segment, the entries in the offset table may be made symmetrical. With a little extra logic, this symmetry may be exploited to halve the size of the offset table (Schulte and Stine, 1997). One of the input bits indicates whether the input value is in the left or right part of a segment. It can therefore be used to invert (using exclusive

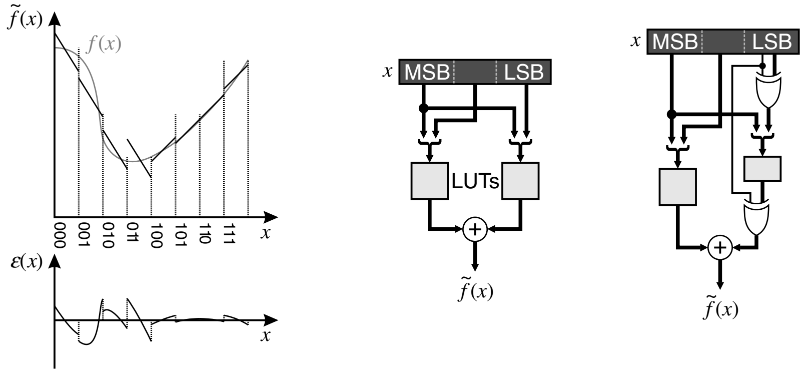


Figure 5.32 Bipartite lookup tables. Right: symmetry is exploited to reduce the table size.

OR gates) the other address bits within the segment to get the corresponding symmetrical position. The offset from the table is also inverted to give the symmetrical offset, as shown on the right in Figure 5.32.

This idea may be extended further to give multiple tables in parallel (Stine and Schulte, 1999; de Dinechin and Tisserand, 2001), which enables the size of the tables to be reduced further, at the cost of an increased number of additions.

Overall, it has been shown (Boullis *et al.*, 2001; Mencer and Luk, 2004) that for short inputs (up to about 8–10 bits) a direct table lookup is best. For intermediate size inputs (10–12 bits) the bipartite tables require the smallest resources. However, for larger inputs (12–20 bits) interpolated lookup tables are the best compromise.

5.4.3 CORDIC

CORDIC (Coordinate Rotation Digital Computer; Volder, 1959) is an iterative technique for evaluating elementary functions. It was originally developed for trigonometric functions (Volder, 1959), but was later generalised to include hyperbolic functions and multiplication and division (Walther, 1971). For trigonometric functions, each iteration is based on rotating a vector (x, y) by an angle θ_k:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} \cos\theta_k & -\sin\theta_k \\ \sin\theta_k & \cos\theta_k \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix} \tag{5.27}$$

The trick is choosing the angle and rearranging Equation 5.27 so that the factors within the matrix are successive negative powers of two, enabling the multiplications to be performed by simple shifts.

$$\begin{aligned} \begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} &= \frac{1}{\cos\theta_k} \begin{bmatrix} 1 & -d_k \tan\theta_k \\ d_k \tan\theta_k & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix} \\ &= \sqrt{1+2^{-2k}} \begin{bmatrix} 1 & -d_k 2^{-k} \\ d_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix} \end{aligned} \tag{5.28}$$

where the direction of rotation, d_k, is $+1$ for an anticlockwise rotation and -1 for a clockwise rotation, and the angle is given by:

$$\tan\theta_k = 2^{-k} \tag{5.29}$$

After a series of K rotations, the total angle rotated is the sum of the angles of the individual rotations:

$$\theta = \sum_{k=0}^{K-1} d_k \tan^{-1} 2^{-k} \tag{5.30}$$

The total angle will depend on the particular rotation directions chosen at each iteration. Therefore, it is useful to have an additional register (labelled z) to accumulate the angle.

The resultant vector is:

$$\begin{aligned} \begin{bmatrix} x_K \\ y_K \end{bmatrix} &= \prod_{k=0}^{K-1} \left(\sqrt{1+2^{-2k}} \begin{bmatrix} 1 & -d_k 2^{-k} \\ d_k 2^{-k} & 1 \end{bmatrix} \right) \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \\ &= G \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \end{aligned} \tag{5.31}$$

where the scale factor, G, is the gain of the rotation and is constant for a given number of iterations.

$$G = \prod_{k=0}^{K-1} \sqrt{1+2^{-2k}} \approx 1.64676 \tag{5.32}$$

The iterations are, therefore:

$$\begin{aligned} x_{k+1} &= x_k - d_k 2^{-k} y_k \\ y_{k+1} &= y_k + d_k 2^{-k} x_k \\ z_{k+1} &= z_k - d_k \tan^{-1}\left(2^{-k}\right) \end{aligned} \tag{5.33}$$

with the corresponding circuit shown in Figure 5.33. On *Load*, the initial values are loaded into the x, y and z registers, and k is reset to zero. Then with every clock cycle, one iteration of summations is performed and k incremented. Note that it is necessary for the summations and the registers to have an extra $\log_2 N$ guard bits to protect against the accumulation of rounding errors (Walther, 1971). The most expensive part of this circuit is the shifters, which are often implemented using multiplexers. If the FPGA has available spare multipliers, these may be used to perform the barrel shifts (Gigliotti, 2004). A small ROM is required to store the small number of angles given by Equation 5.29. There are two modes for choosing the direction of rotation at each step (Volder, 1959): rotation mode and vectoring mode.

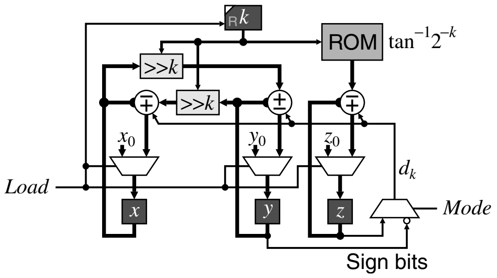


Figure 5.33 Iterative hardware implementation of the CORDIC algorithm.

The rotation mode starts with an angle in the z register and reduces this angle to zero by choosing d_k as the sign of the z register as the vector is rotated:

$$d_k = \mathrm{sign}(z_k) \tag{5.34}$$

The result after K iterations is:

$$\begin{aligned} x_K &= G(x_0 \cos z_0 - y_0 \sin z_0) \\ y_K &= G(x_0 \sin z_0 + y_0 \cos z_0) \\ z_K &= 0 \end{aligned} \tag{5.35}$$

This converges (z_K can be made to go to zero within the precision of the calculation; Walther, 1971) if:

$$\theta_k - \sum_{j=k+1}^{K-1} \theta_j < \theta_{K-1} \tag{5.36}$$

In other words, if the angle is zero, and a rotation moves it away, the remaining rotations must be able to move it back to zero again. In this case, the series converges because:

$$\tan^{-1} 2^{-i} < 2 \tan^{-1} 2^{-(i+1)} \tag{5.37}$$

For Equation 5.33 the domain of convergence is:

$$|z_0| < \sum_{k=0}^{K-1} \tan^{-1} 2^{-k} \approx 99.88^\circ \tag{5.38}$$

Starting with $x_0 = 1/G$ and $y_0 = 0$, the rotation mode can be used to calculate the sine and cosine of the angle in z_0. It may also be used to rotate a vector by an angle, subject to the extension of the magnitude by a factor of G. For angles larger than 90°, the easiest solution is to perform an initial rotation by 180°:

$$\begin{aligned} x_0 &= -x_{-1} \\ y_0 &= -y_{-1} \\ z_0 &= z_0 \pm 180^\circ \end{aligned} \tag{5.39}$$

The second mode of operation is vectoring mode. This chooses d_k as the opposite of the sign of the y register, determining the angle required to reduce y to zero:

$$d_k = -\mathrm{sign}(y_k) \tag{5.40}$$

which gives the result:

$$\begin{aligned} x_K &= G\sqrt{x_0^2 + y_0^2} \\ y_K &= 0 \\ z_K &= z_0 + \tan^{-1}\left(\frac{y_0}{x_0}\right) \end{aligned} \tag{5.41}$$

This has the same domain of convergence as the rotation mode, given in Equation 5.38. Therefore, if $x_0 < 0$ it is necessary to bring the vector within range by rotating by 180° (Equation 5.39). The vectoring mode effectively converts a vector from rectangular to polar coordinates. If x_0 and y_0 are both small, then rounding errors can become significant, especially in the calculation of the arctangent (Kota and Cavallaro, 1993). If necessary, this may be solved by renormalizing x_k and y_k by scaling by a power of two (a left shift).

With both the rotation mode and vectoring mode, each iteration improves the accuracy of the angle by approximately one binary digit.

An additional mode of operation has been suggested by Andraka (1998) for determining arcsine or arccosine. It starts with $x_0 = 1$ and $y_0 = 0$ and rotates in the direction given by:

$$d_k = \text{sign}(s - y_k) \tag{5.42}$$

where s is the sine of the angle. The result after converging is:

$$\begin{aligned} x_K &= \sqrt{G^2 x_0^2 - s^2} \\ y_K &= s \\ z_K &= z_0 - \sin^{-1}\left(\frac{s}{Gx_0}\right) \end{aligned} \tag{5.43}$$

This result would be simpler if initialised with $x_0 = 1/G$. This mode is effective for $|s/Gx_0| < 0.98$. Outside this range, the gain of the rotation can cause the wrong direction to be chosen, leading to convergence failure (Andraka, 1998). An alternative two-step approach to calculating arcsine and arccosine will be described shortly.

One way of speeding up CORDIC is to consider the Taylor series expansions for sine and cosine for small angles (Walther, 2000):

$$\begin{aligned} \sin\theta &= \theta\left(1 - \frac{1}{6}\theta^2 + \frac{1}{120}\theta^4 + \cdots\right) \approx \theta \\ \cos\theta &= 1 - \frac{1}{2}\theta^2 + \frac{1}{24}\theta^4 + \cdots \approx 1 \end{aligned} \tag{5.44}$$

After $N/2$ iterations in the rotation mode, the angle in the z register is such that θ^2 is less than the significance of the least significant guard bit, so can be ignored. The remaining $N/2$ iterations may be replaced by small multiplications:

$$\begin{aligned} x_N &= x_{N/2} - z_{N/2} y_k \\ y_N &= y_{N/2} + z_{N/2} x_k \end{aligned} \tag{5.45}$$

(The assumption here is that the angle is in radians.) Note that the same technique cannot be applied in the vectoring mode because it relies on the angle approaching zero.

The implementation in Figure 5.33 is word-serial in that each clock cycle performs one iteration at the word level. Other implementations are the bit-serial (Andraka, 1998) and the unrolled parallel versions (Wang *et al.*, 1996; Andraka, 1998). The bit-serial implementation uses significantly fewer resources, but requires N clock cycles per iteration, where N is the width of the registers. The unrolled implementation (Figure 5.34) uses separate hardware for each iteration and can perform one complete calculation per clock cycle. Unrolling enables two simplifications to the design. Firstly, the shifts at each iteration are fixed and can be implemented simply by wiring the appropriate bits to the inputs to the adders. Secondly, the angles that are added are constants, simplifying the z adders.

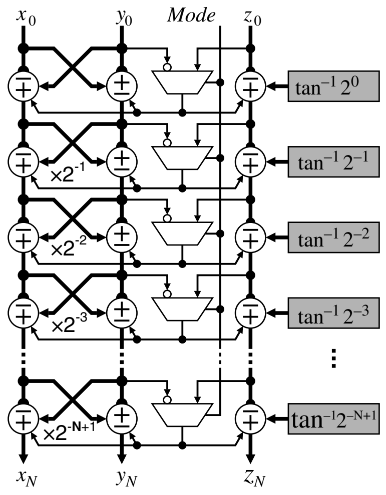


Figure 5.34 Unrolled CORDIC implementation.

The propagation delay of the unrolled implementation is long because the carry must propagate all of the way to the sign bit at each iteration before selecting the next operation. However, the design can be readily pipelined with a throughput of one calculation per clock cycle. Several attempts have been made at reducing the latency by investigating ways of overcoming the carry propagation. It cannot be solved using redundant arithmetic, because to determine the sign bit with redundant arithmetic it is still necessary to propagate the carry. Two approaches have been developed, although they only work in rotation mode. The first is to work with the absolute value in the z register and detect the sign changes (Dawid and Meyr, 1992), which can be done with redundant arithmetic, working from the most significant bit down, significantly improving the delay. The second approach relies on the fact that the angles added to the z register are constant, enabling d_k to be predicted, significantly reducing the latency (Timmermann *et al.*, 1992).

5.4.3.1 Compensated CORDIC

A major limitation of the CORDIC technique is the rotation gain. If necessary, this constant factor may be removed by a multiplication either before or after the CORDIC calculation. Compensated CORDIC algorithms integrate this multiplication within the basic CORDIC algorithm in different ways.

Conceptually the simplest (and the original compensated CORDIC algorithm) is to introduce a scale factor of the form 1 ± 2^{-k} with each iteration (Despain, 1974). The sign is chosen so that the combined gain approaches one. This can be generalised in two ways. The first is by applying the scale factor only on some iterations, and choosing the terms so that the gain is two, enabling it to be removed with a simple shift. Since the signs are predetermined, this results in only a small increase in the propagation delay, although the circuit complexity is considerable increased.

The second approach is to repeat the iterations for some angles to drive the total gain, G, to two (Ahmed, 1982). This uses the same hardware as the original CORDIC, with only a little extra logic required to control the iterations. The disadvantage is that it takes significantly longer than the uncompensated algorithm. A hybrid approach is to use a mixture of additional iterations and scale factor terms to reduce the total logic required (Haviland and Tuszynski, 1980). This is probably best applied to the unrolled implementation.

Another approach is to use a double rotation (Villalba *et al.*, 1995). This technique is only applicable to the rotation mode; it performs two rotations in parallel (with different angles) and combines the results. Consider rotating by an angle $(\theta + \beta)$:

$$\begin{aligned} x_N^+ &= G(x_0(\cos\theta\cos\beta - \sin\theta\sin\beta) - y_0(\sin\theta\cos\beta + \cos\theta\, sin\,\beta)) \\ y_N^+ &= G(x_0(\sin\theta\cos\beta + \cos\theta\sin\beta) + y_0(\cos\theta\cos\beta - \sin\theta\sin\beta)) \end{aligned} \tag{5.46}$$

Similarly, rotating by an angle $(\theta - \beta)$ gives:

$$\begin{aligned} x_N^- &= G(x_0(\cos\theta\cos\beta + \sin\theta\sin\beta) - y_0(\sin\theta\cos\beta - \cos\theta\sin\beta)) \\ y_N^- &= G(x_0(\sin\theta\cos\beta - \cos\theta\sin\beta) + y_0(\cos\theta\cos\beta + \sin\theta\sin\beta)) \end{aligned} \tag{5.47}$$

Then adding these two results cancels the terms containing $\sin\beta$:

$$\begin{aligned} \frac{1}{2}\left(x_N^+ + x_N^-\right) &= G(x_0\cos\theta\cos\beta - y_0\sin\theta\cos\beta) \\ \frac{1}{2}\left(y_N^+ + y_N^-\right) &= G(x_0\sin\theta\cos\beta + y_0\cos\theta\cos\beta) \end{aligned} \tag{5.48}$$

Therefore, by setting $\cos\beta = 1/G$, the gain term can be completely cancelled. It takes the same time to perform the double rotation as the original CORDIC apart from the extra addition at the end. However, it does require twice the hardware to perform the two rotations in parallel, and this technique does not work for the vectoring mode.

5.4.3.2 Linear CORDIC

So far the discussion of CORDIC has been in a circular coordinate space. The CORDIC iterations may also be applied in a linear space (Walther, 1971):

$$\begin{aligned} x_{k+1} &= x_k \\ y_{k+1} &= y_k + d_k 2^{-k} x_k \\ z_{k+1} &= z_k - d_k 2^{-k} \end{aligned} \tag{5.49}$$

Again, either the rotation mode or vectoring mode may be used. Rotation mode for linear CORDIC reduces z to zero using Equation 5.34, and in doing so is performing a variation on long multiplication with the result:

$$\begin{aligned} x_K &= x_0 \\ y_K &= y_0 + x_0 z_0 \\ z_K &= 0 \end{aligned} \tag{5.50}$$

The domain of convergence depends on the initial value of k.

$$|z_0| < \sum_{k=k_i}^{K-1} 2^{-k} \approx 2^{1-k_i} \tag{5.51}$$

Vectoring mode reduces y to zero using Equation 5.40 and effectively performs the same algorithm as non-restoring division, with the result:

$$\begin{aligned} x_K &= x_0 \\ y_K &= 0 \\ z_K &= z_0 + \frac{y_0}{x_0} \end{aligned} \tag{5.52}$$

Usually, $k_i = 1$ for which Equation 5.52 will converge provided $y_0 < x_0$. Note that there is no scale factor or gain term associated with linear CORDIC.

5.4.3.3 Hyperbolic CORDIC

Walther (Walther, 1971) also unified CORDIC with a hyperbolic coordinate space. The corresponding iterations are then:

$$\begin{aligned} x_{k+1} &= x_k + d_k 2^{-k} y_k \\ y_{k+1} &= y_k + d_k 2^{-k} x_k \\ z_{k+1} &= z_k - d_k \tanh^{-1}\left(2^{-k}\right) \end{aligned} \tag{5.53}$$

There are convergence issues with these iterations that require certain iterations to be repeated ($k = 4, 13, 40, \ldots, k_r, 3k_r + 1, \ldots$) (Walther, 1971). Note, too, that the iterations start with $k = 1$ because $-1 < \tanh x < 1$.

Subject to these limitations, operating in rotation mode gives the result:

$$\begin{aligned} x_K &= G_h(x_0 \cosh z_0 + y_0 \sinh z_0) \\ y_K &= G_h(x_0 \sinh z_0 + y_0 \cosh z_0) \\ z_K &= 0 \end{aligned} \tag{5.54}$$

where the gain factor (including the repeated iterations) is:

$$G_h = \prod_{k=1}^{K-1} \sqrt{1-2^{-2k}} \approx 0.82816 \tag{5.55}$$

and the domain of convergence is:

$$|z_0| < \sum_{k=1}^{K-1} \tanh^{-1} 2^{-k} \approx 1.118 \tag{5.56}$$

Operating in vectoring mode gives the result:

$$\begin{aligned} x_K &= G_h \sqrt{x_0^2 - y_0^2} \\ y_K &= 0 \\ z_K &= z_0 + \tanh^{-1}\left(\frac{y_0}{x_0}\right) \end{aligned} \tag{5.57}$$

As with circular CORDIC, the gain may be compensated either by appropriate initialisation of the x or y register (for rotation mode) or by introducing additional, compensating operations. Much of the discussion of compensated CORDIC above may be adapted to hyperbolic coordinates.

The CORDIC iterations from the different coordinate systems may be combined to use the same hardware (with a multiplexer to select the input to the x register). With all three modes available, other common functions may also be calculated (Walther, 1971):

$$\tan\theta = \frac{\sin\theta}{\cos\theta} \tag{5.58}$$

$$\tanh\theta = \frac{\sinh\theta}{\cosh\theta} \tag{5.59}$$

$$\exp x = \sinh x + \cosh x \tag{5.60}$$

$$\ln x = 2\tanh^{-1}\left(\frac{x-1}{x+1}\right) \tag{5.61}$$

$$\sqrt{x} = \sqrt{\left(x+\frac{1}{4}\right)^2 - \left(x-\frac{1}{4}\right)^2} \tag{5.62}$$

Other common functions that may be calculated in two steps are:

$$\sin^{-1}x = \tan^{-1}\left(\frac{x}{\sqrt{1-x^2}}\right) \tag{5.63}$$

$$\cos^{-1}x = \tan^{-1}\left(\frac{\sqrt{1-x^2}}{x}\right) \tag{5.64}$$

5.4.3.4 CORDIC Variations and Generalisations

The basis behind CORDIC is to reduce multiplications by an arbitrary number to a shift by constraining the system to powers of two. This can be extended to the direct calculation of other functions. For example, looking at logarithmic functions (Specker, 1965), the relation:

$$\ln x\left(1+2^k\right) = \ln x + \ln\left(1+2^k\right) \tag{5.65}$$

allows a shift and add algorithm for the calculation of logarithms. The idea is to successively reduce x to one by a series of multiplications, and adding the corresponding logarithm terms obtained from a small table.

The iteration is given by:

$$\begin{aligned} x_{k+1} &= x_k\left(1+d_k 2^{-k}\right) \\ z_{k+1} &= z_k - \ln\left(1+d_k 2^{-k}\right) \end{aligned} \tag{5.66}$$

where
$$d_k = \begin{cases} 1, & x_k + x_k 2^{-k} < 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.67}$$

will converge for $0.4195 < x_0 \leq 1$ to:

$$\begin{aligned} x_K &= 1 \\ z_K &= z_0 + \ln x_0 \end{aligned} \tag{5.68}$$

Alternatively, if a factor of $(1-d_k 2^{-k})$ is used, then the iteration converges for $1 \leq x_0 < 3.4627$. The iteration of Equation 5.66 may also be readily adapted to taking logarithms in other bases simply by changing the table of constants added to the z register. The algorithm can also be run the other way, by reducing z_k to zero, giving the exponential function.

Muller (1985) showed that this iteration shared the same theory as the CORDIC iteration and placed them within the same framework. Since trigonometric functions can be derived from complex exponentials using the Euler identity:

$$e^{j\theta} = \cos\theta + j\sin\theta \tag{5.69}$$

the iteration of Equation 5.66, can be generalised to use complex numbers (Bajard *et al.*, 1994) enabling trigonometric functions to be calculated without the gain factors associated with CORDIC. Another advantage of this implementation is that it allows redundant arithmetic to be used (Nielsen and Muller, 1996) by exploiting the fact that if $d_k \in \{-1, 0, 1\}$ then there is an overlap between the selection ranges for the d_k. This means than only the few most significant bits need to be calculated in order to determine the next digit, reducing the effects of carry propagation delay.

While the CORDIC and other related shift-and-add based algorithms have relatively simple circuitry, their main limitation is their relatively slow rate of convergence. Each iteration gives only a single bit improvement of the result. A further limitation is that only a few elementary functions may be calculated. More complex or composite functions require the evaluation of many elementary functions.

5.4.4 Approximations

An alternative to direct function evaluation is to approximate the function with another simpler function that gives a similar result over some domain of interest. One popular method of approximating functions is to use a minimax polynomial. This chooses the polynomial coefficients to minimise the maximum approximation error within the domain of interest:

$$\tilde{f}(x) = \sum_{i=0}^{N} a_i x^i \tag{5.70}$$

Typically, Remez's exchange algorithm is used to find the coefficients. However, such coefficients do not take into account the fact that it is desirable to perform the calculation with limited precision or small multipliers, and simply rounding the coefficients may not give the coefficients with minimum error (Brisebarre *et al.*, 2006). This requires a search for the appropriate coefficients.

For low precision outputs (less than 14–20 bits) table lookup methods are both faster and more efficient (Sidahao *et al.*, 2003). However, the resources to implement a lookup table grow exponentially with output precision, whereas those required for polynomial approximations grow only linearly.

An extension of polynomial approximations are rational approximations where:

$$\tilde{f}(x) = \frac{\sum_{i=0}^{N} a_i x^i}{\sum_{i=0}^{M} b_i x^i} \tag{5.71}$$

For a given level of accuracy, a lower order rational polynomial is required (Koren and Zinaty, 1990) than a regular polynomial, although it does require a division operation. This makes rational polynomial approximations best suited for high precision.

A further approach for high precision is to use an iterative approximation. This starts with an approximate solution and with each iteration improves the accuracy. A common approach is the Newton–Raphson iteration, which is a root-finding algorithm. The Newton–Raphson iteration is given by:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{5.72}$$

It works, as illustrated in Figure 5.35, by projecting the slope at x_k to obtain a more accurate estimate.

Newton–Raphson has quadratic convergence. This means that if the input is close to the solution, the relative error will square with each iteration. This effectively means that the number of significant bits will approximately double with each iteration.

For example, to evaluate the square root of an input number, X, an equation is formed for which $\sqrt{X}$ is a root:

$$f(x) = x^2 - X = 0 \tag{5.73}$$

Then, the iteration is formed from Equation 5.72:

$$\begin{aligned} x_{k+1} &= x_k - \frac{x_k^2 - X}{2x_k} \\ &= \frac{1}{2}\left(x_k + \frac{X}{x_k}\right) \end{aligned} \tag{5.74}$$

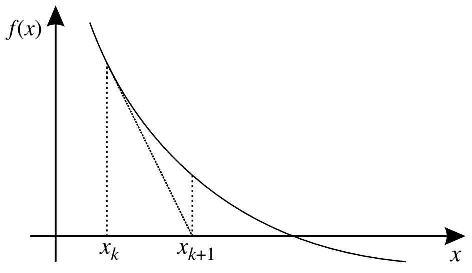


Figure 5.35 Newton–Raphson iteration.

Note that the iteration of Equation 5.74 requires a division, which is relatively expensive. A variation on this technique is to rearrange the equation to be solved. For example:

$$\sqrt{X} = \frac{X}{\sqrt{X}} \tag{5.75}$$

so solving $1/\sqrt{X}$ enables $\sqrt{X}$ to be calculated with a final multiplication. The corresponding function to solve would be:

$$f(x) = x^{-2} - X = 0 \tag{5.76}$$

with the corresponding iteration:

$$\begin{aligned} x_{k+1} &= x_k - \frac{x_k^{-2} - X}{-2x_k^{-3}} \\ &= \frac{1}{2} x_k \left(3 - x_k^2 X\right) \end{aligned} \tag{5.77}$$

Although Equation 5.77 is more complex than Equation 5.74, it does not involve a division and can almost certainly be implemented with lower latency.

The number of iterations required to reach a desired level of accuracy depends on the accuracy of the initial estimate. Therefore, other techniques, such as lookup tables or other approximations, are helpful to give a more accurate initial approximation and hence reduce the number of iterations (Schwarz and Flynn, 1993). Since the iteration for many functions is a polynomial (for example Equation 5.77), the same hardware may be used to form the initial approximation using a low order polynomial (Habegger *et al.*, 2010).

5.4.5 Other Techniques

5.4.5.1 Bit-Serial Processing

In situations where resources are scarce but latency is less of an issue, bit-serial processing can provide a solution. Conventionally, operations are performed a whole word at a time, processing all of the bits within a data word in parallel. In contrast to this, bit-serial techniques process the data one bit per clock cycle (Denyer and Renshaw, 1985). This requires serialising the input data and processing the data as a serial bit-stream. The order that the bits should be presented (least significant bit first, or most significant bit first) will depend on which input bits affect which output bits (Andraka, 1996). For example, most arithmetic operations should be presented least significant bit first because carry propagates from the lower bits to the higher bits. This is illustrated in Figure 5.36 for a bit-serial adder.

Processing only one bit at a time can significantly reduce the propagation delay, although more clock cycles are required to process the whole word. For many operations, bit-serial processing can lead to efficient pipelined designs or systolic structures. Another significant advantage of bit-serial designs is that, because the logic is more compact, routing is also simplified and more efficient (Isshiki and Dai, 1995).

A compromise between bit-serial and word-parallel systems is digit-serial designs (Parhi, 1991). These process a small number of bits (a digit) in parallel but have the digits presented serially. This can reduce the high clock speeds required for bit-serial designs while maintaining efficient use of logic resources. Digit-serial designs therefore fall within the spectrum between bit-serial and word-parallel systems.

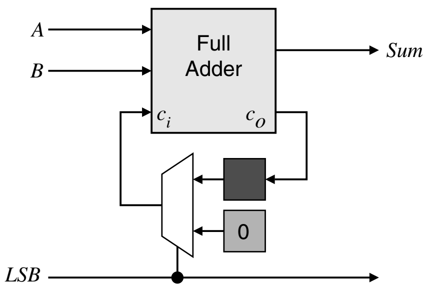


Figure 5.36 Bit-serial adder.

5.4.5.2 Incremental Update

Another technique that may be employed, particularly with stream processing, is incremental update. Rather than perform a complete calculation independently for each pixel, incremental update exploits the fact that often part of the calculation performed for the previous pixel may be reused. A classic example (Gribbon *et al.*, 2003) is the calculation of x^2 or y^2 where x and y are pixel coordinates. When moving to the next pixel, the identity:

$$(x+1)^2 = x^2 + 2x + 1 \tag{5.78}$$

can be used to maintain the value of x^2 without performing a multiplication. The same concept may be extended to calculate higher order moments (Chan *et al.*, 1996).

A similar technique may be used to determine the input location for an affine transformation incrementally. Consider the transformation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.79}$$

The input location corresponding to the next pixel in the scan line is then given by:

$$\begin{aligned} x' &\Leftarrow x' + A \\ y' &\Leftarrow y' + D \end{aligned} \tag{5.80}$$

Note that in this case it may be necessary to keep extra guard digits to prevent accumulation of round-off errors in the approximation of A and D.

Another common example of incremental update is with local filters. As the window moves from one pixel to the next, there is a significant overlap between the corresponding input windows. Many of the pixels within the window will be the same as the window moves. This may be exploited, for example, by median filters by maintaining a histogram of the values within the window and updating the histogram only for the pixels that change (Garibotto and Lambarelli, 1979; Huang *et al.*, 1979; Fahmy *et al.*, 2005).

5.4.5.3 Separability

Image processing often involves operations that require inputs from two (or more) dimensions. Separable operations are those that can split the two-dimensional operations into separate operations in each of the X

and *Y* dimensions. The separate one-dimensional operations are usually significantly simpler than the composite two-dimensional operation. The most common application of separability is with filters, but it can also be applied to two-dimensional transformations such as the fast Fourier transform (FFT), or be used with image warping (Catmull and Smith, 1980).

5.5 Summary

This chapter has reviewed a range of techniques for mapping from a software algorithm to a hardware implementation. These techniques may be considered as a set of design patterns that provides ways of overcoming common problems or constraints encountered in the mapping process (DeHon *et al.*, 2004; Gribbon *et al.*, 2005; 2006).

Timing constraints focussed particularly on the limited time available for performing the required operations. One technique for improving throughput is to use low level pipelining. Another timing issue encountered with concurrent systems is the synchronisation of the different processes. For synchronous processes, global scheduling can derive synchronisation signals from a global counter. With asynchronous systems one synchronisation technique is to use a channel based on CSP. If necessary, a FIFO can be used to smooth the flow of data between processes.

Memory bandwidth constraints result from the fact that only one memory access may be made per clock cycle. A range of techniques were reviewed that improved the bandwidth of external memories. The largest effect can be obtained by caching data on the FPGA where the data may be distributed over several parallel block RAMs, significantly increasing the bandwidth. Custom caching can significantly reduce the number of times that the data needs to be loaded into the FPGA.

The third form of constraint was for resources. Any concurrent system with shared resources will face contention when a resource is used by multiple processes. The resource needs to be multiplexed between the various processes, with appropriate conflict arbitration. This is best managed by combining the resource with its manager. Efficient use of the limited logic and memory resources available on an FPGA requires appropriate data structures and computational techniques to be used by the algorithm. If necessary, the FPGA can be dynamically reconfigured to switch from one task to another. While partial reconfigurability has received a lot of research, not every FPGA supports it, and those that do require quite low level programming to manage the placement constraints.

The next several chapters explore how these may be specifically applied to mapping image processing operations onto an FPGA. Every operation may be implemented in many different ways. However, the primary focus is on stream processing where appropriate. This is because, for an embedded system, the data is inevitably streamed from a camera or other device. If the data is processed as it is streamed in, the memory requirements of the algorithm can often be significantly reduced. Using stream processing does place a strict time constraint of a one pixel per clock cycle (although this may be relaxed a little for slower pixel clocks by using multiphase techniques), so not every operation may be able to be streamed.

6

Point Operations

Beginning with this chapter, the focus is on how specific image processing operations may be implemented on an FPGA. The next few chapters describe preprocessing and other low level operations. Later chapters move to intermediate level operations.

An assumption in the next few chapters (unless stated otherwise) is that the N-bit pixels represent unsigned values from 0 to $V = 2^N - 1$. Most of the examples here will be given with $N = 8$ bits.

6.1 Point Operations on a Single Image

The simplest class of image processing operations is that of point operations. They are so named because the output value for a pixel depends only on the corresponding pixel value of the input image:

$$Q[x,y] = f(I[x,y]) \tag{6.1}$$

where f is some arbitrary function. Since the output value depends only on the input value and not on the location of the pixel in the image, point operations may be represented by a mapping or transfer function, as shown in Figure 6.1.

From a hardware implementation perspective, point operations may be easily implemented in any processing mode. However, because each operation is applied exactly once to each pixel in the image, the simplest approach is to systematically pass each input pixel through a single hardware block implementing the function. This corresponds to the streamed mode of processing and is illustrated in the right panel of Figure 6.1. Since each pixel is processed independently, point operations can also be easily implemented in parallel. The image may be partitioned and a separate processor used with each partition.

In spite of their simplicity, point operations have wide use in terms of contrast enhancement, segmentation, colour filtering, change detection, masking and many other applications.

If the subsequent imaging operation requires random access of pixels (for example from a frame buffer), the hardware block implementing the point operation may be placed between the source and subsequent operation. In this way, as the pixels are read, they are transformed by the point operation as required for the subsequent process.

6.1.1 Contrast and Brightness Adjustment

One interpretation of the mapping function is to consider its effect on the brightness and contrast of the image. To make an image brighter, the output pixel value needs to be increased. This may be accomplished

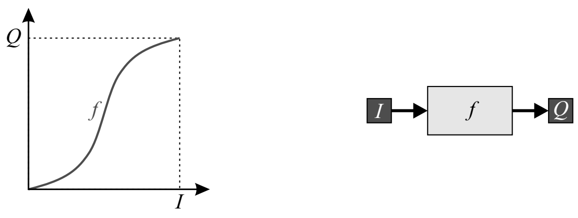


Figure 6.1 A point operation maps the input pixel value, I, to the output value, Q, via an arbitrary mapping function, f. Right: implementation of a point operation using stream processing.

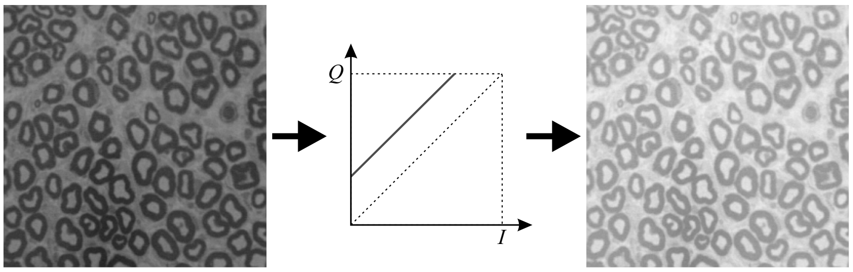


Figure 6.2 Increasing the brightness of an image by adding a constant.

by adding a constant, as illustrated in Figure 6.2. Similarly, an image may be made darker by decreasing the pixel value, which may be accomplished by subtracting a constant. In practise, adjusting the brightness is more complicated than this because the human visual system is nonlinear and does not consider each point in an image in isolation but relative to its context within the overall image. This is shown in the optical illusion in Figure 6.3, where the band through the image appears brighter on the left and darker on the right (actually the pixel value is the same).

The contrast of an image is affected or adjusted by the slope of the mapping function. A slope greater than one corresponds to an increase in the contrast (Figure 6.4) and a slope less than one corresponds to a contrast reduction. This may be accomplished by multiplying by a constant greater than or less than one respectively.

A relatively simple point operation for adjusting both the brightness and contrast is:

$$Q = aI + b = a(I + b') \tag{6.2}$$

where a and b are arbitrary constants that control the brightness and contrast. One issue is what to do when the value for Q exceeds the range of representable values. For example, with an eight bit per pixel representation, what should be output if Q is outside the range of 0–255? The default of just taking the

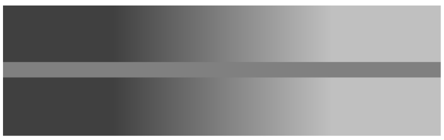


Figure 6.3 Brightness illusion.

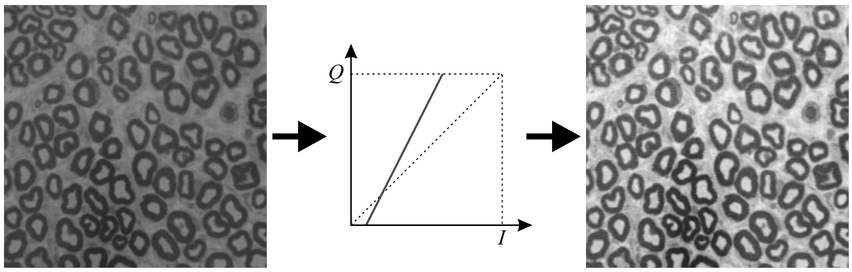


Figure 6.4 Increasing the contrast of the image.

eight least significant bits and ignoring any overflow would cause values outside this range to wrap around. If the goal is to enhance the brightness or contrast of an image, this is seldom the desired effect. More sensible would be for the output to saturate or clip to the limits. Note that clipping the output is not invertible.

Clipping requires building hardware to detect when the limits are exceeded and adjust the output accordingly. This is shown in block diagram form in Figure 6.5. Note that either the brightness or contrast operation may be performed first depending on the form of Equation 6.2.

On the left in Figure 6.5, the clipping tests are performed after the calculation of Equation 6.2. This means the propagation delay for the complete operation includes the time required for the clipping test. If, however, the clipping test was performed on the input, it would be computed in parallel with the contrast enhancement operation (as shown on the right), and the propagation delay would be reduced. The clip limits referred to the input are:

$$I_{\min} = \frac{0-b}{a},\ I_{\max} = \frac{V-b}{a} \tag{6.3}$$

This form is most practical if the gain and offset values are constant, so that the input range can be determined at compile time.

This circuit may be further simplified by relying on the fact $V = 2^N - 1$. With positive gain, it is only the offset that can cause the result to go negative. Using the second form of Equation 6.2, the sign bit of the addition can be used to set the output to zero. Similarly, the carry from the multiplication can be used to clip to the maximum by setting all the bits to one. This scheme is shown in Figure 6.6.

Further optimisations may be made. Obviously if the gain and offset are constant then the multiplication may be replaced by a series of fixed additions. Representing the gain in canonical signed digit form will reduce the number of terms that need to be added.

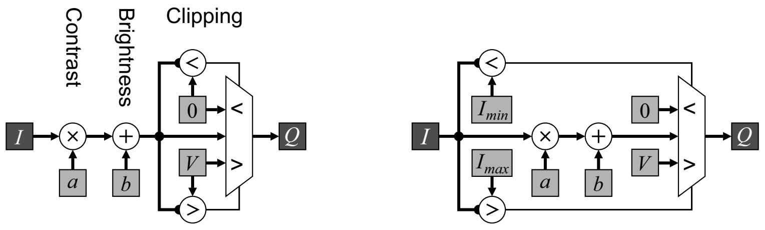


Figure 6.5 Schematic representation of a simple contrast enhancement operation. Right: improving the performance by moving the comparisons to the input.

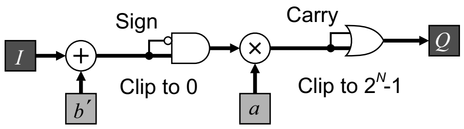


Figure 6.6 Simplified circuit. The sign bit from the addition sets the result to zero if the number is negative, and the carry from the multiplication sets the result to all ones.

A common contrast enhancement method is to derive the gain and offset from the minimum and maximum values available in the input image. The contrast is then maximised by stretching the pixel values to occupy the full range of available pixel values. This obviously requires that the whole image be read to determine the extreme pixel values, with consequent timing issues. Such issues are discussed in Section 7.1.2 with histogram equalisation, another adaptive contrast enhancement technique.

A contrast reversal is obtained if the slope of the transfer function is negative. For example to invert the image, a gain of -1 is used, as shown in Figure 6.7, where:

$$Q = V - I = \left(2^N - 1\right) - I = \bar{I} \tag{6.4}$$

Logically, this may be obtained by taking the one's complement (inverting each bit) of the input.

The transformation does not necessarily have to be linear. The same principle applies with a nonlinear mapping: raising or lowering the output will increase or decrease the brightness of the corresponding pixels; slopes greater than one correspond to a local increase in contrast, while slopes less than one correspond to a decrease in contrast; a negative slope will result in contrast inversion. A nonlinear mapping allows different effects at different intensities.

A common mapping is the gamma transformation. This is easier to represent when considering normalised values (where Q_N and I_N represent fractions between zero and one).

$$Q_N = I_N^{\gamma} \tag{6.5}$$

The gamma curve was designed primarily to compensate for nonlinearities in the transfer function at various stages within the imaging system. For example, the relationship between the drive voltage of a cathode ray tube (CRT) and the output intensity is not linear, but is instead a power law. Therefore, a compensating gamma curve is required to make the contrast appear more natural on the display. (This is complicated by the fact that our eyes are not linearly sensitive to intensity either.)

The effects of the gamma transformation can be seen in Figure 6.8. When γ is less than one, the curve moves up, increasing the overall brightness of the image. From the slope of the curve, the contrast is

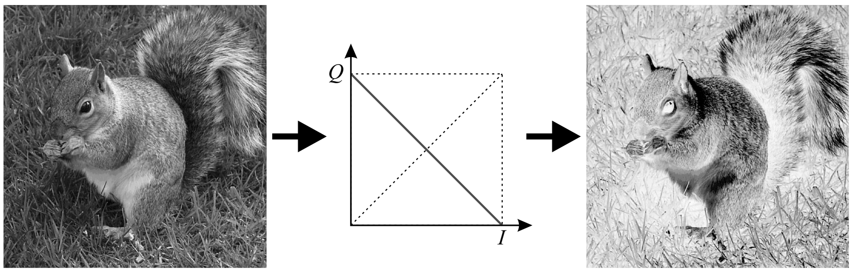


Figure 6.7 Inverting an image (Photo courtesy of Robyn Bailey).

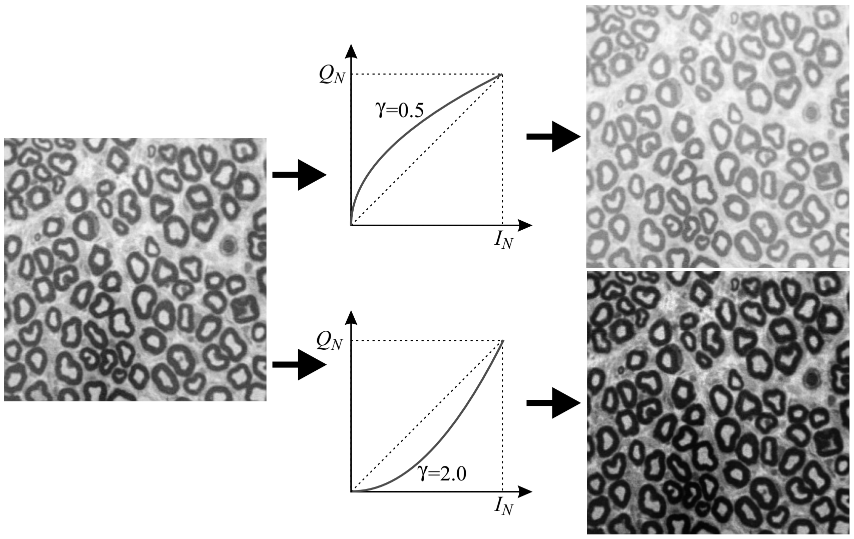


Figure 6.8 The effects of a gamma transformation.

increased for smaller pixel values at the expense of the lighter pixels, which have reduced contrast. For γ greater than one, the opposite occurs. The image becomes darker and the contrast is enhanced in the lighter regions at the expense of the darker regions.

A sigmoid curve is also sometimes used to selectively enhance the contrast of the mid-tones at the expense of the highlights and shadows (Braun and Fairchild, 1999).

6.1.2 Global Thresholding and Contouring

Thresholding, in its basic form, compares each pixel in the image with a threshold level and assigns the output to true or false (white or black) as shown in Figure 6.9. Since each pixel is treated identically, thresholding is a point operation.

$$Q = \begin{cases} 1, & I \geq thr \\ 0, & I < thr \end{cases} \tag{6.6}$$

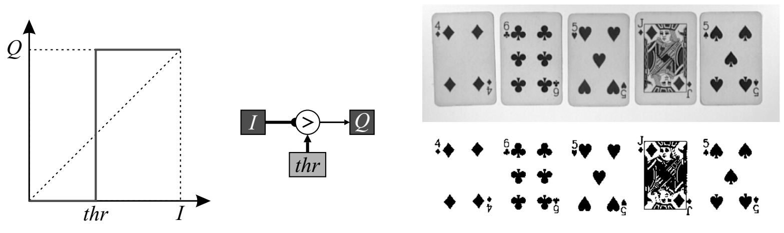


Figure 6.9 Simple global thresholding.

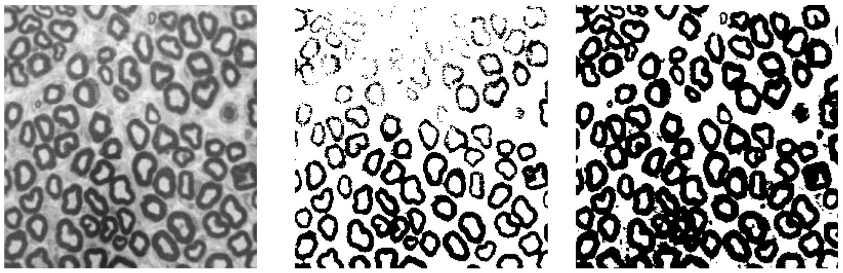


Figure 6.10 Global thresholding is not always appropriate. Left: input image; centre: optimum threshold for the bottom of the image; right: optimum threshold for the top of the image.

Thresholding effectively classifies each pixel into one of two classes and is commonly used to segment between object and background. An appropriate threshold level can often be selected by analysing the statistics of the image, for example as contained in a histogram of pixel values. Several such threshold selection methods are described in Section 7.1.4 under histogram processing.

Sometimes, however, a single global threshold does not suit the whole image. Consider the example in Figure 6.10. The optimum threshold for one part of the image is not the optimum for other parts. Using a global threshold to process such images will involve a compromise in the quality of the thresholding of some region. A point operation is unsuitable to process such images, because to give good results over the whole image the operation needs to take into account not only the input pixel value but also the local context. Such images require adaptive thresholding, a form local filtering, and is considered further in Section 8.7.

Simple thresholding may be generalised to operate with more than one threshold level.

$$Q = \begin{cases} 2, & thr_2 \le I \\ 1, & thr_1 \le I < thr_2 \\ 0, & I < thr_1 \end{cases} \tag{6.7}$$

An example of this is shown in Figure 6.11. Here, two thresholds have been used to separate the image into three regions: the background, the cards and the pips. Each output pixel is assigned a label based on its relationship to the two threshold levels.

In this example, the darkest region is never adjacent to the lightest region. This it not the case in Figure 6.12, where the interior of the white bubbles is black. There is also a gradation in contrast between the white and black giving a ring of pixels within the bubbles detected as the intermediate background level. However, even if there was a sharp change in contrast between the black and white, many of

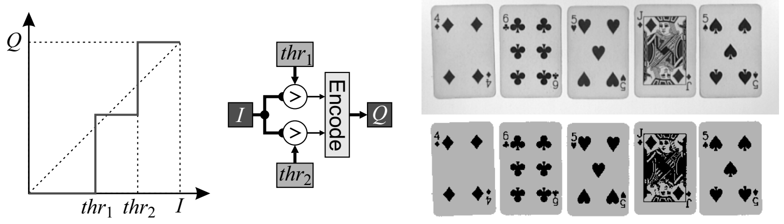


Figure 6.11 Multilevel thresholding.

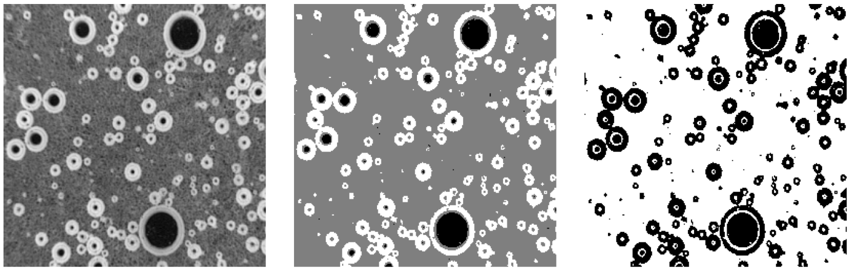


Figure 6.12 Misclassification problem with multilevel thresholding. Left: original image; centre: classifying the three ranges; right: the background level labelled with white to show the misclassifications within each bubble.

the boundary pixels are still likely to have intermediate levels. This is because images are area sampled; the pixel value is proportional to the average light falling within the area of the pixel. Therefore, if the boundary falls within a pixel, as illustrated in Figure 6.13, part of the pixel will be white and part black, resulting in intermediate pixel values. Any blurring or lens defocus will exacerbate this effect.

Consequently, with multilevel thresholding, it is inevitable that some of the boundary pixels will fall between the thresholds and will be assigned an incorrect label. Such misclassifications require the information from the context (filtering) either as preprocessing before the thresholding, or after thresholding to reclassify the incorrect incorrectly labelled pixels.

Another form of multilevel thresholding is contouring. This selects pixel values within a range, as seen in Figure 6.14, and requires a minor modification to Equation 6.7:

$$Q = \begin{cases} 0, & thr_2 \leq I \\ 1, & thr_1 \leq I < thr_2 \\ 0, & I < thr_1 \end{cases} \tag{6.8}$$

Contouring is so named because, in an image with slowly varying pixel values, the pixels selected appear like contour lines on a map. However, contouring is less effective on high contrast edges, because it is detecting the intermediate pixel values illustrated in Figure 6.14. For a given threshold range, only a few edge pixels may fall within the range, and there is no guarantee that these pixels would be connected. While the pixels detected are likely to be edge pixels (depending on the threshold range), contouring makes a poor edge detector because it lacks the context required to give connectivity.

As the output after thresholding is binary, it only requires a single bit of storage per pixel. (More bits would be required for the output from multilevel thresholding, although this is usually significantly less than that of the input image.) Therefore, if an image is able to be successfully thresholded as it is streamed from the camera, the memory requirements for buffering the image are significantly reduced.

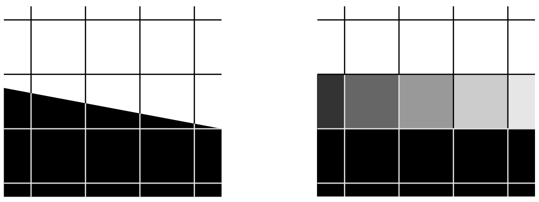


Figure 6.13 Edge pixels have an intermediate pixel value.

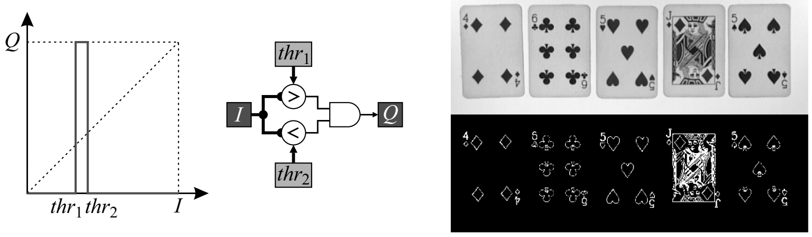


Figure 6.14 Contouring selects pixels within a range of pixel values.

6.1.3 Lookup Table Implementation

Most of the point operations considered so far are relatively simple mappings and can be implemented directly using logic. More complex mappings, such as gamma adjustment, require considerable logic to implement directly. If necessary, if such operations were implemented in logic, pipelining can be used to improve the speed.

Since the output value depends only on the input, the mapping for any point operation can be precalculated and stored in a lookup table. Then it is simply a case of using the input pixel value to index into the table to find the corresponding output. An example of this is shown in Figure 6.15.

The biggest advantage of a lookup table implementation is the constant access time, regardless of the complexity of the function. The size of the table depends on the width of the input stream, with each extra bit on the input doubling the size of the table.

A disadvantage of using a lookup table is that once the table has been set, the mapping is fixed. If the mapping is parameterised (for example contrast enhancement or thresholding) then it is also necessary to build logic to construct the lookup table before the frame is processed. Constructing the table will use the same (or similar) logic as implementing the point operation directly. The only difference is that timing is less critical. In a system with a software coprocessor, it may also be possible to construct the table in software rather than building hardware to calculate the table values.

If the parameter can be reduced to a small number of predefined values, then one alternative is to use the parameter to select between several preset lookup tables (as shown on the left in Figure 6.16). This is achieved by concatenating the parameter with the input pixel value to form the table address. Another possibility for more complex parameterised functions is to combine one or more lookup tables with logic, as shown for the gamma mapping on the right in Figure 6.16.

A sequence of two or more point operations is in itself a point operation. For example, contrast enhancement followed by thresholding is equivalent to thresholding with a different threshold level.

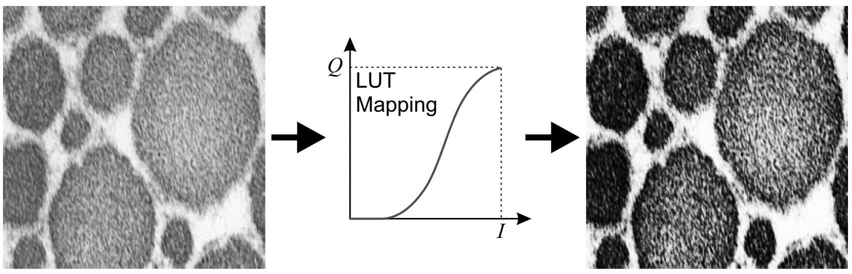


Figure 6.15 Performing an arbitrary mapping using a lookup table.

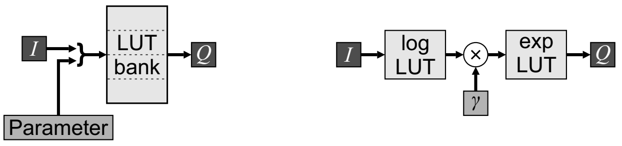


Figure 6.16 Parameterised lookup tables. Left: using the parameter as an address; right: combining tables with logic (implementing the gamma mapping).

This property of point operations implies that a complex sequence of point operations may be replaced by a single lookup table that implements the whole sequence in a single clock cycle.

6.2 Point Operations on Multiple Images

Point operations can be applied not only to single images, but also between multiple images. For this, Equation 6.1 is extended to:

$$Q[x,y] = f(I_1[x,y], I_2[x,y], \ldots, I_k[x,y]) \tag{6.9}$$

Corresponding pixels of the images are combined by an arbitrary function, f, to give the output pixel value at the corresponding pixel location.

There are two main ways in which multi-image point operations are used. One is between two or more images derived from the same input image. A variation of this is between images derived from different, but synchronised, sensors. Stream processing requires that the streams for each of the images be synchronised. If the processing branches prior to the point operation have different latencies, then it will be necessary to introduce a delay within the faster branches to equalise the latency with the slowest branch. This is shown in Figure 6.17 for a three-input point operation. Note that the delay does not necessarily need to follow the faster process (as with Process 2); it may be inserted before it (as with Process 3), or even distributed throughout the process. However, keeping the delay together enables a more efficient implementation using shift registers or a FIFO buffer.

The other way is to apply the point operation between images captured at different times. This requires one or more frame buffers to hold the data from previous frames, as shown in Figure 6.18. Direct sequential access operates directly on the input image and one or more previous images. Recursive access feeds the output image back to be combined in some way with the incoming input image. In both cases, the frame buffer is used for both reading and writing. Some of the techniques of Section 5.2.1 may need to be used to enable the multiple parallel accesses.

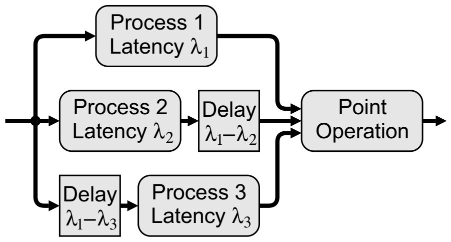


Figure 6.17 Two-image point operation applied to the same input image. A delay is inserted into the faster branch to equalise the latency.

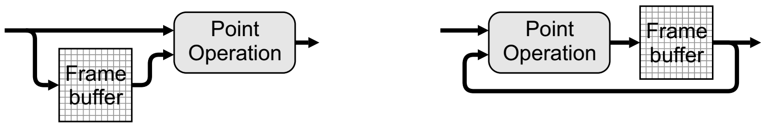


Figure 6.18 Two-image point operation applied to successive images. Left: sequential access; right: recursive access. A frame buffer is required to hold the image from the previous frame.

Just as single image point operations may be implemented using a lookup table, lookup tables may also be used to implement multiple image point operations. The table input or address is formed by concatenating the individual inputs. The main limitation of this approach is that the memory requirement of a table grows exponentially with the number of address lines. This makes multiple input point operation lookup tables impractical for all but the most complex operations, or where the latency is critical. As with single input lookup tables, the latency of multiple input LUTs is constant at one clock cycle.

While there are potentially many multi-image point operations, this section focuses primarily on some of the more common ones.

6.2.1 Image Averaging

Real images inevitably contain noise, both from the imaging process and also from quantisation when creating a digital image. Let the captured image be the sum of the ideal, noise-free image and a noise image:

$$I[x,y] = \tilde{I}[x,y] + n[x,y] \tag{6.10}$$

By definition, the noise-free image is the same from frame to frame. Let the noise have zero mean, and be independent from frame to frame:

$$E\big(n_i[x,y]n_j[x,y]\big) = 0 \quad \text{for } i \neq j \tag{6.11}$$

where E is the expectation operator. The signal-to-noise ratio (SNR) of the noisy image is then defined by the ratio of the energy in the signal to the energy in the noise. One definition of this is:

$$SNR_I = \frac{\sigma_{\tilde{I}}^2}{\sigma_n^2} \tag{6.12}$$

If several images of the same scene are averaged:

$$Q[x,y] = \sum_i w_i I_i[x,y] \tag{6.13}$$

where:

$$\sum_i w_i = 1 \tag{6.14}$$

then:

$$\begin{aligned} Q[x,y] &= \sum_i w_i \tilde{I}[x,y] + \sum_i w_i n[x,y] \\ &= \tilde{I}[x,y] + \hat{n}[x,y] \end{aligned} \tag{6.15}$$

The image content is the same from frame to frame, so will reinforce when added. However, since the individual noise images are independent, the noise in one frame will partially cancel the noise in other frames. The noise variance will therefore decrease and is given by (Mitra, 1998):

$$\sigma_{\hat{n}}^2 = \sigma_n^2 \sum_i w_i^2 \tag{6.16}$$

If averaging a constant number of images, N, the greatest noise reduction is given when the weights are all equal. The output signal-to-noise ratio is then:

$$SNR_Q = \frac{\sigma_{\tilde{I}}^2}{\sigma_{\hat{n}}^2} = N\frac{\sigma_{\tilde{I}}^2}{\sigma_n^2} = N \times SNR_I \tag{6.17}$$

Note that the signal-to-noise ratio depends on the noise variance. The noise amplitude will therefore decrease by $\sqrt{N}$.

There is also a limitation with averaging multiple frames to reduce quantisation noise. If the noise from other sources (before quantisation) is significantly less than one pixel value, then the quantisation noise will be similar from one frame to the next. The noise term will no longer be independent and the reduction given by Equation 6.16 will no longer apply. On the other hand, if the noise standard deviation from other sources has a standard deviation larger than one pixel value, then the quantisation noise will be independent and will be reduced by averaging.

The main problem with averaging multiple frames is the requirement to store the previous $N-1$ frames in order to perform the averaging. Alternatively, this limitation may be overcome if the output frame rate is reduced by a factor of N. In this latter case, the images are accumulated until N have been summed, then the accumulator is reset to begin accumulation of the next N.

To overcome this limitation, one technique is to have an exponentially decreasing sequence of weights.

$$Q[x,y] = \sum_{i=0}^{\infty} \alpha(1-\alpha)^i I_{-i}[x,y] \tag{6.18}$$

This may be implemented efficiently by calculating the result recursively:

$$\begin{aligned} Q_i[x,y] &= (1-\alpha)Q_{i-1}[x,y] + \alpha I_i[x,y] \\ &= Q_{i-1}[x,y] + \alpha(I_i[x,y] - Q_{i-1}[x,y]) \end{aligned} \tag{6.19}$$

The output noise variance, from Equation 6.16, is then given by:

$$\begin{aligned} \sigma_{\hat{n}}^2 &= \sigma_n^2 \sum_{i=0}^{\infty} \alpha^2 (1-\alpha)^{2i} \\ &= \sigma_n^2 \frac{\alpha}{2-\alpha} \end{aligned} \tag{6.20}$$

with the resulting signal-to-noise ratio:

$$SNR_Q = \left(\frac{2}{\alpha} - 1\right)\frac{\sigma_{\tilde{I}}^2}{\sigma_n^2} \tag{6.21}$$

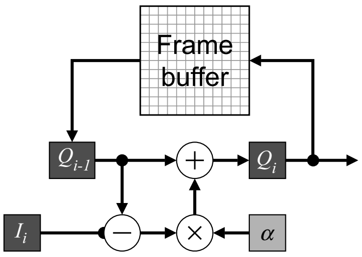


Figure 6.19 An implementation of weighted image averaging using Equation 6.19.

By equating Equation 6.21 with Equation 6.17, the weight to give an equivalent noise smoothing to averaging N images is:

$$\alpha = \frac{2}{N+1} \tag{6.22}$$

The assumption with this analysis is that no additional noise is introduced by the averaging process. In practise, the result of multiplication by α will require truncating the result. This will introduce additional noise, limiting the improvement in signal-to-noise ratio. To reduce this additional noise, it is necessary to maintain additional guard bits on the accumulator image.

An example implementation of Equation 6.19 is shown in Figure 6.19. Since the frame buffer will almost certainly be off-chip memory, one of the schemes of Section 5.2.1 will need to be used to enable the accumulated image to be both read and written for each pixel. For practical reasons, α can be made a power of two, enabling the multiplication to be implemented with a fixed binary shift. If $\alpha = 2^{-k}$ then the effective noise averaging window, from Equation 6.22, is:

$$N = \frac{2}{\alpha} - 1 = 2^{k+1} - 1 \tag{6.23}$$

In most applications, this restriction is not a problem.

6.2.2 Image Subtraction

The main purpose for subtracting images is to determine the similarity between two images. Two global metrics are commonly used to gauge similarity: the sum of absolute differences (SAD):

$$SAD = \sum_{x,y} \left| I_1[x,y] - I_2[x,y] \right| \tag{6.24}$$

and the sum of squared differences (SSD):

$$SSD = \sum_{x,y} (I_1[x,y] - I_2[x,y])^2 \tag{6.25}$$

By squaring the difference, the *SSD* effectively gives more weight to large differences.

One application of these metrics is in image registration. The spatial offset of one of the images is adjusted to obtain an estimate of the similarity as a function of position.

$$SSD[i,j] = \sum_{x,y} (I_1[x+i, y+j] - I_2[x,y])^2 \tag{6.26}$$

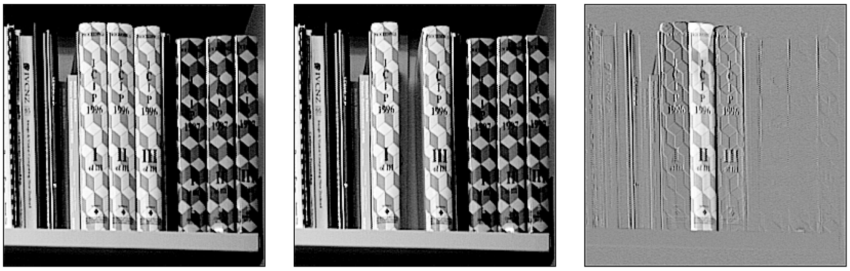


Figure 6.20 Image subtraction to detect change. Left: original scene; centre: the changed scene; right: the difference image (offset to represent zero difference as a mid-grey).

The offset corresponding to the minimum *SAD* or *SSD* is therefore the offset that makes the images most similar, and therefore provides an estimate of the relative position of the images to the nearest pixel. Image registration is discussed in more detail in Section 9.5.

Another application of image subtraction is to detect changes within the scene. This is illustrated in Figure 6.20. Where there is no change, the difference between pixel values will be zero and any changes in the image will result in non-zero difference. This allows not only the addition or removal of objects to be detected, but also subtle shifts in the position of the object, which result in differences at intensity edges within the image.

Offsetting the pixel values so that no difference is represented by mid-grey enables both positive and negative differences to be represented. Any small shift in the image contents between the compared frames results in a bas relief effect, as seen in the difference image in Figure 6.20. Analysing the pixel values and thickness of these features enable even subpixel shifts to be detected.

Note that it is important when subtracting images taken at different times that the lighting remains constant or the images be normalised. This is particularly difficult with outdoor scenes, where there is little control over the lighting. There are two basic approaches to managing this problem. One is to take differences only between images closely spaced in time, where it is assumed that conditions do not change significantly from frame to frame, and the other is to maintain a dynamic estimate of the background that adapts as conditions change.

The principle behind frame differencing is that, when an object moves, part of the background in the previous image becomes obscured, resulting in a change in pixel value (assuming the object and background have different pixel values). One of the limitations of frame differencing is that there will also be a difference where the object was, as the background is uncovered. This is shown clearly in Figure 6.21. While in this case the previous position and current position can be distinguished by the sign of the difference, in general, with a complex background and objects this will not be the case. Significant differences are detected by thresholding the absolute difference. Double differencing (Kameda and Minoh, 1996) then detects regions that are common between successive difference images using a logical AND. It therefore detects objects that are arriving in the first difference and leaving in the second.

Although the double difference requires data from three frames, the recursive architecture introduced in Figure 6.19 can be readily adapted, as illustrated in Figure 6.22. The input pixel value is augmented with one extra bit (Δ) to indicate that the pixel has been detected as different between the last two frames. This difference bit is then combined with the difference bit stored with the previous frame to give the detected output pixels.

For more slowly moving objects (that obscure the background for several successive images) it is necessary that the object has sufficient texture to create differences in both difference frames. Double differencing has been applied to vehicle detection using an FPGA implementation (Cucchiara *et al.*, 1999).

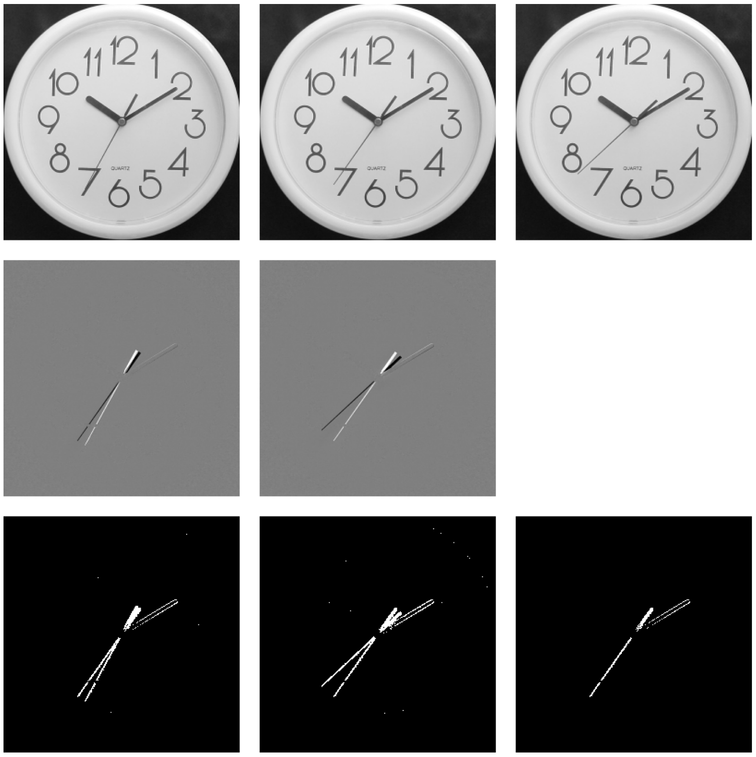


Figure 6.21 Double difference approach. Top: three successive input images; middle: differences between successive images; bottom, left and centre: absolute differences above the threshold; bottom, right: the double difference.

The alternative is to construct a model of the static background and take the difference between each successive image and the background. To account for changing conditions, it is necessary for the background image (or model) to be adaptive. Such background models can vary significantly in complexity (McIvor *et al.*, 2001).

The simplest model is to represent the background image by the mean (Heikkila and Silven, 1999) or median (Cutler and Davis, 1998) of the previous several images. The median is less sensitive to outliers,

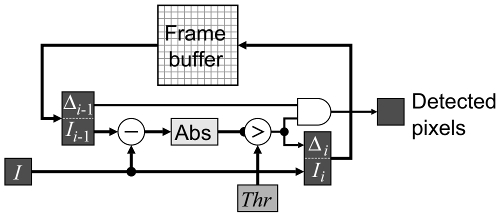


Figure 6.22 Implementing double differencing.

but requires maintaining a large number of images to calculate. The mean, however, can be estimated using the recursive update of Equation 6.19. In this equation, α controls how quickly changes to the scene are updated into the background. Larger values of α result in faster adaption of the background and rapid assimilation of objects into the background, but may result in artefacts of trails behind moving objects as they are partially assimilated into the background (Heikkila and Silven, 1999).

A problem with such a simple model is that it is unable to cope with regions of the image that are naturally variable, for example leaves fluttering in the wind. By estimating the variance associated with each pixel, pixels that vary significantly may be either masked out or only be detected as foreground object pixels if the difference exceeds three standard deviations (Orwell *et al.*, 1999). Equation 6.19 may be adapted to efficiently derive an estimate of the variance:

$$\sigma_i^2[x,y] = (1-\alpha)\sigma_{i-1}^2[x,y] + \alpha(I_i[x,y]-Q_i[x,y])^2 \tag{6.27}$$

Calculating the standard deviation requires taking the square root of the variance. A simpler alternative is to replace the standard deviation with the mean absolute deviation:

$$d_i[x,y] = (1-\alpha)d_{i-1}[x,y] + \alpha\left|I_i[x,y]-Q_i[x,y]\right| \tag{6.28}$$

Again, the recursive architecture of Figure 6.19 can be extended to detect foreground pixels. Figure 6.23 replicates the mean update logic to calculate the mean absolute deviation. All pixels that differ from the mean by more than three deviations are detected as foreground pixels. As before, it is desirable to have α to be a power of two to reduce the logic. The multiplication of the deviation by three can also be implemented with a shift and add.

A limitation of the mean and variance approach is that the large variance (or mean absolute deviation) is often the result of two or more different distributions being represented by a pixel. For example, with a flag waving in the wind a pixel may alternate between the background and one of the colours on the flag. More sophisticated models account for the bimodal and multimodal pixel value distributions resulting from such effects. They represent each pixel by a weighted mixture of Gaussians (Stauffer and Grimson, 1999). To be effective, the mixture and Gaussian parameters need to be updated online as each image is acquired. The basic approach (Stauffer and Grimson, 1999) is to determine which of the Gaussians is represented by the current pixel (starting with the Gaussian that has the most weight). If it is within 2.5 standard deviations of the mean of one of the Gaussians, the pixel value is incorporated into that Gaussian, using Equations 6.19 and 6.27. The weights for all of the Gaussians in the mixture are updated using a similar weighted update, increasing the weight of the matched Gaussian and decreasing the weights of the others. If none of the Gaussians in the mixture matches the current pixel, the Gaussian with the least weight is discarded and replaced by a new Gaussian with a large standard deviation and low weight. The most probable Gaussians are considered background and the lowest weight Gaussians represent new events, so are considered to be foreground object pixels.

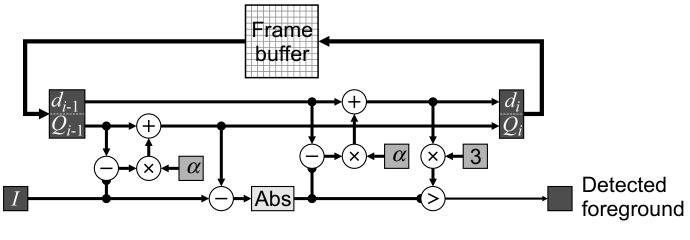


Figure 6.23 Detecting foreground pixels using the mean and mean absolute deviation.

A significant advantage of the mixture model over a single Gaussian occurs when an object stops long enough to become part of the background, and then moves away again. The single Gaussian model will gradually drift to the new pixel value and then drift away again, resulting in an object being detected for a considerable time after it moves. With the mixture model, however, the mean does not drift, but a new distribution is begun for the object. When this receives sufficient weight, it is considered background. The model for the original background is retained, so when the object moves again, the pixels are correctly classified as background.

Typically, three to five Gaussians are used to model most images (McIvor *et al.*, 2001). An FPGA implementation would be a relatively straightforward extension of Figure 6.23. Using more terms for each pixel requires more storage and logic to maintain the model. This may require multiple external RAM banks to give the required memory width. Alternatively, the frame rate can be reduced to enable sequential memory locations to be used for the component distributions. Appiah and Hunter (2005) have implemented a slightly simpler version of multimodal background modelling using a fixed width rather than explicitly modelling Gaussians. Their implementation considered both greyscale and colour images.

Other, more advanced and more complex methods of background estimation and subtraction have been reviewed by Piccardi (2004).

6.2.3 Image Comparison

Image addition and subtraction of successive images may be considered as a form of temporal filtering. Another type of filter useful for background estimation is to select the median (Cutler and Davis, 1998), maximum or minimum pixel value of successive images. The median is used in a similar manner to image averaging as described earlier. A problem with using the median, however, is that multiple images must be stored to enable its calculation on a pixel-by-pixel basis. The memory storage and consequent bandwidth issues restrict the usefulness of median calculation to short time windows for real-time operation.

If it is desired to detect dark objects moving against a light background, then the maximum of successive images may be able to estimate the background (Shilton and Bailey, 2006). Sufficient images must be combined so that for each pixel, at least one image within the set contains a background pixel. This may be implemented directly with the recursive architecture.

$$M_i[x,y] = \max(M_{i-1}[x,y], I_i[x,y]) \tag{6.29}$$

This makes using the maximum (or conversely using the minimum to estimate a dark background with light objects) practical. Only the maximum (or minimum) found so far needs to be stored.

Two limitations of Equation 6.29 are that any outlier will automatically become part of the background and, if conditions change, the background is unable to adapt. These may be overcome by building a decay into the expression:

$$M_i[x,y] = \max(\alpha M_{i-1}[x,y], I_i[x,y]) \tag{6.30}$$

where α is slightly less than one. The multiplication may be replaced by a subtraction by choosing:

$$\alpha = 1-2^{-k} \tag{6.31}$$

When working with the minimum, such a multiplication does not work as well. In this case, it is necessary to invert the image before scaling, so that the decay is toward white rather than black:

$$\begin{aligned} m_i[x,y] &= \min\left((2^N-1)-\alpha\left((2^N-1)-m_{i-1}[x,y]\right), I_i[x,y]\right) \\ &= \min\left(\overline{\alpha\overline{m_{i-1}[x,y]}}, I_i[x,y]\right) \end{aligned} \tag{6.32}$$

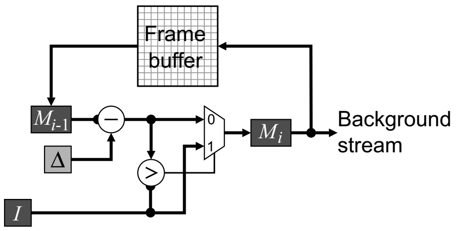


Figure 6.24 Estimating a light background using a recursive maximum.

In practise, the inversion may be performed efficiently by taking the one's complement. Expressing Equation 6.32 in terms of its dual puts it in the same form as Equation 6.30:

$$\overline{m_i[x,y]} = \max\left(\alpha\overline{m_{i-1}[x,y]}, \overline{I_i[x,y]}\right) \tag{6.33}$$

The disadvantage of the multiplication is that the amount of the decay depends on the pixel value. An alternative is simply to subtract a constant:

$$M_i[x,y] = \max(M_{i-1}[x,y] - \Delta, I_i[x,y]) \tag{6.34}$$

where Δ is related to the expected noise level. The constant offset may also be used for estimating a dark background:

$$m_i[x,y] = \min(m_{i-1}[x,y] + \Delta, I_i[x,y]) \tag{6.35}$$

An implementation of this using the recursive architecture is shown in Figure 6.24.

Image comparison is also used for adaptive thresholding. Instead of using a constant global threshold as in Equation 6.6, the threshold level is made to depend on the local context:

$$Q[x,y] = \begin{cases} 1, & I[x,y] \geq thr[x,y] \\ 0, & I[x,y] < thr[x,y] \end{cases} \tag{6.36}$$

Equivalently, the comparison may be performed by subtracting the two images and then using a single global threshold.

Determining the threshold from the context requires some form of local filtering. Adaptive thresholding are therefore discussed in more detail in Section 8.7 in the context of filters.

6.2.4 *Intensity Scaling*

Multiplication or division can be used to selectively enhance the contrast within an image. One image is usually the input image being enhanced, and the other image represents the pixel dependent gain. As described earlier, a gain greater than one will increase the contrast and a gain less than one will reduce the contrast. The gain image is often obtained by preprocessing the input image in some way.

One application of pixel-by-pixel division is correcting for non-uniform pixel sensitivity within the image sensor, or vignetting caused by the lens. If the pixel response is nonlinear, then correction requires characterising the nonlinearity function, which is potentially different for each pixel (Sawchuk, 1977). Fortunately, most modern solid-state sensors are linear, enabling a simpler correction. The basic principle is to characterise the response by capturing an image of a uniform field. This approach is also able to correct for uneven illumination, provided the illumination is unchanged in the images being corrected.

Firstly, it is necessary to capture a reference image of a uniform field or a plain, non-textured background. Since the image should be uniform, any variation in pixel value is a result of deficiencies in the capture process, whether caused by variations in sensor gain or illumination. An image of a scene can then be corrected by dividing by the reference image (Aikens *et al.*, 1989):

$$Q[x,y] = k\frac{I[x,y]}{Ref[x,y]} \tag{6.37}$$

where the constant k controls the dynamic range of the output image and is chosen so that the full range of pixel values is used. If the input and reference images are captured under identical conditions, then k is typically set to 255 (or V). For practical implementation, k can be combined with the reference image to avoid the extra operation. After calibration, the reference image does not change. Therefore, the processing complexity of Equation 6.37 may be reduced by converting the division to a multiplication and precalculating $k/Ref[x,y]$.

For best accuracy, when capturing the reference image the exposure should be as large as possible without actually causing any pixels to saturate. In practise, a uniform illumination field is difficult to achieve (Schoonees and Palmer, 2009) but a similar reference image may be obtained by processing a sequence of images of a moving background containing a modulation pattern. The processing enables the variations caused by the imaging process to be separated from the variations in the illumination (Schoonees and Palmer, 2009).

Note that any errors or noise present in the reference image will be introduced into the image through Equation 6.37. It is therefore important to minimise the noise. This may be accomplished, if necessary, by averaging several images or applying an appropriate noise smoothing filter.

Once calibrated, the reference image needs to be stored so that it can be made available as needed for processing. The large size of an image requires an external memory to use as a frame buffer in most circumstances. However, as the reference image is generally slowly varying, it can be compressed readily. A simple compression scheme would be to down-sample the reference image and reconstruct the reference using interpolation. Down-sampling by a factor of 8 or 16 would give a significant data reduction, enabling the reference to be stored directly on the FPGA. See Section 9.3 for an implementation of the reference reconstruction process.

When considering only vignetting, an alternative approach is to model the intensity fall off with radius (Goldman and Chen, 2005). The model can then be used to perform the correction, rather than requiring a reference image.

Other applications of normalisation include colour balancing and colour space conversion. For example, calculating the colour saturation requires normalising by the intensity. This, and other related colour transformations are described in more detail in Section 6.3.

Correlation is another image operation that requires multiplication of images on a pixel-by-pixel basis. The resulting product is then summed over the image:

$$COR = \sum_{x,y} I_1[x,y]I_2[x,y] \tag{6.38}$$

Like the sum of squares or absolute differences, this is a measure of the similarity of two images, with a larger correlation indicating a better match. Like the sum of differences, correlation is also used for image registration, by adjusting the offset of one of the images to maximise the correlation. It can be shown that correlation is closely related to the minimum square difference (Jain, 1989). Expanding Equation 6.25 gives:

$$\begin{aligned} SSD &= \sum_{x,y} (I_1[x,y]-I_2[x,y])^2 \\ &= \sum_{x,y} \left(I_1^2[x,y]+I_2^2[x,y]-2I_1[x,y]I_2[x,y]\right) \\ &= \sum_{x,y} I_1^2[x,y] + \sum_{x,y} I_2^2[x,y]-2COR \end{aligned} \tag{6.39}$$

The first two terms are constant when registering images, so maximising the correlation is equivalent to minimising the sum of squared difference.

The problem with simple correlation for image registration is that images are finite. Therefore, if the image is darker on one side than the other, this can introduce a bias that offsets the correlation peak. This may be overcome by normalising the correlation by the overlap area and average pixel value (Jain, 1989):

$$COR_N[i,j] = \frac{\sum_{x,y} f[x+i,y+j]g[x,y]}{\sqrt{\sum_{x,y} f^2[x+i,y+j]}\sqrt{\sum_{x,y} g^2[x,y]}} \tag{6.40}$$

This requires maintaining three accumulators for each correlation output: one for fg, one for f^2 and one for g^2. The division and square root are only performed at the end of the image, rather than at every pixel, so their speed is much less critical.

6.2.5 Masking

The final operation that will be considered in this section is image masking. In its most basic form it consists of a logical AND or OR operation. Masking is commonly used to select a region of an image to process, while ignoring irrelevant regions within the image. The choice to use an AND or OR depends on the desired level for the background. ANDing with zero will result in a black background, while ORing with one will make the background white. These options are shown schematically in Figure 6.25.

Logical OR and AND can also be used to combine multiple regions into a single image. OR is used with a black background and AND with a white background. The result is a generalisation of multiplexing, using a set of mask images to select the corresponding image data for the output.

One application of this is image compositing, creating a single image from a series of offset images, for example when making a panorama from a series of panned images. To ensure a good result, it is necessary to correct for lens and other distortions and to ensure accurate registration in the region of overlap.

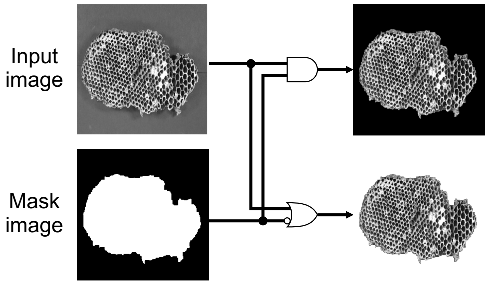


Figure 6.25 Masking, with setting of the background to black or white.

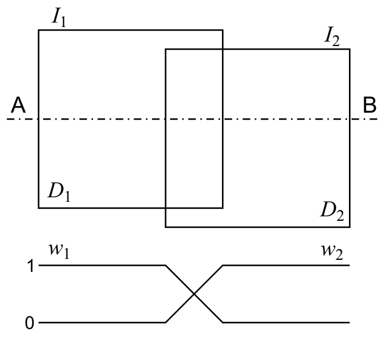


Figure 6.26 Merging of images in the overlap region. The weights corresponding to the input images are shown for the section A–B.

Where the scene geometry is known, this may be accomplished by rectifying the images (Bailey and Shand, 1996). If the images are subject to vignetting, the seams between the images can be visible. Correcting for vignetting (Goldman and Chen, 2005) can significantly improve the results by making the pixel values of on each side of the join similar.

Rather than switch from one image to the other at the border between the images, the seam resulting from mismatched pixel values between the images may be significantly reduced by merging the images in the region of the overlap. Consider two overlapping frames, I_1 and I_2, as illustrated in Figure 6.26. In the region of overlap the two frames are merged with a smooth transition from one to the other. The weights applied to each of the images, w_1 and w_2 respectively, depend on the width of overlap, which may vary from image to image, and also with position within the image.

One relatively simple technique is to apply the distance transform to each of the mask images. This labels each pixel within the mask image with the distance to the nearest edge of the mask (distance transforms are covered later in Section 11.5). The weights may then be calculated from the distance transformed masks, D_1 and D_2:

$$w_1 = \frac{D_1}{D_1 + D_2}, \; w_2 = \frac{D_2}{D_1 + D_2} = 1 - w_1 \tag{6.41}$$

The output image is then given by:

$$\begin{aligned} Q &= w_1 I_1 + w_2 I_2 \\ &= w_1 I_1 + (1 - w_1) I_2 \\ &= I_2 + w_1 (I_1 - I_2) \end{aligned} \tag{6.42}$$

An implementation of Equation 6.42 suitable for streamed operation is shown in Figure 6.27. Note that these operations are applied for each pixel within the image. The rearrangement in Equation 6.42 reduces

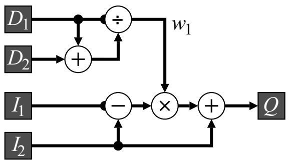


Figure 6.27 Schematic for merging images, given distance weighted masks.

the computation to a single division to calculate the weight and a single multiplication to combine the two images. The calculation may require pipelining to meet timing constraints.

6.3 Colour Image Processing

Colour image processing is a logical extension to the processing of greyscale images. The main difference is that each pixel consists of a vector of components rather than a scalar. A vector-based image can be defined as **I**, where:

$$\mathbf{I}[x,y] = \{I_1[x,y], I_2[x,y], I_3[x,y]\} \tag{6.43}$$

or

$$\mathbf{I}[x,y] = \begin{bmatrix} I_1[x,y] \\ I_2[x,y] \\ I_3[x,y] \end{bmatrix} \tag{6.44}$$

and I_1, I_2, I_3 are the component images. If the colour image is split into its component parts, then a point operation on a colour image can be considered to be a point operation on multiple images.

The most common native representation of an image is for each pixel to have red, green and blue components, $\mathbf{I}_{RGB}\{R, G, B\}$, corresponding approximately with the sensitivity of the cones within the human visual system. As described in Section 1.2, a colour image is usually formed by capturing images with sensors that are selectively sensitive to the red, green and blue regions of the spectrum. Colour is typically represented by a three-dimensional vector, and the most common standard is to have eight bits per component, resulting in a 24-bit colour system.

Sensors are not solely restricted to the visible region of the electromagnetic spectrum. Each spectral band conveys different information about the properties of the object being imaged. Such multispectral imaging, particularly in the infrared, has been found useful in a wide range of applications, including land use and vegetation classification in remote sensing (Ehlers, 1991) and produce grading (Duncan and Leeson, 1999) for detecting blemishes that may not be as apparent in the visible spectrum.

Colour image processing, therefore, involves the processing of such vector-valued images.

6.3.1 False Colouring

The first operation considered is false colouring. This is so named because the colours seen in the output image are false in that they do not reflect the true underlying colour of the scene. The purpose of false colouring is to make apparent to a human viewer that which may not be apparent or visible in the original image. It is seldom used as part of an automatic image processing algorithm.

False colouring is applied in two distinct ways. The first is to map the non-visible components of a multispectral image into the visible region of the spectrum. This enables human viewing (and interpretation) of the resultant image. For example, in remote sensing, different types of vegetation have different spectral signatures that are most obvious in the near infrared, red and green components (Duncan and Leeson, 1999). Therefore, a mapping commonly used within remote sensing is to map these onto the red, green and blue components respectively of the output image or display. The resulting colours are not the same as seen by the unaided human eye. However, they do allow subtle distinctions to be more readily seen and distinguished.

A related technique is to combine different images of the same scene as the different components of a colour image. One example where this has been used is to verify the correct registration of images taken from different viewpoints (Reulke *et al.*, 2008). The images from each viewpoint are registered to a common coordinate system, and when combined, they should align. Any registration errors will show as coloured fringes. A similar application is to combine images taken at different times as different

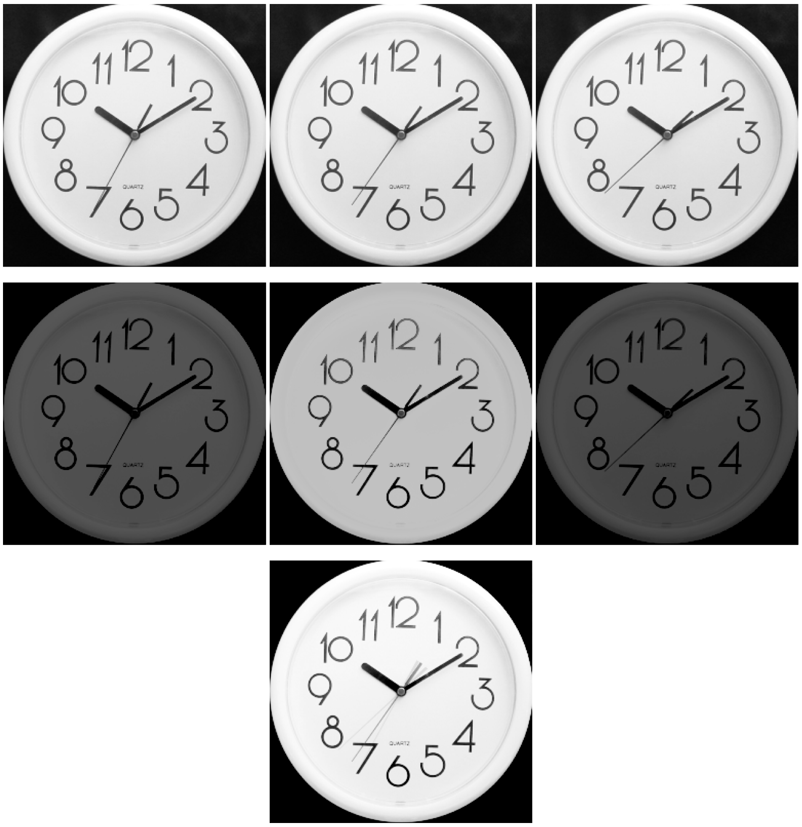


Figure 6.28 Temporal false colouring. Images taken at different times are assigned to different channels, with the resultant output showing coloured regions where there are temporal differences. (*See colour version of this figure in colour plate section*)

components, as shown in Figure 6.28. Common regions within the set will show up as grey because they have equal red, green and blue components. However, any differences will cause an imbalance in the components, resulting in a coloured output. In the example here (Figure 6.28), dark objects appear in their complimentary colour: the dark second hand in the red channel appears cyan, dark in the green channel appears magenta, and dark in the blue channel appears yellow.

The second way in which false colouring is commonly used is to map a greyscale image onto a colour image. Each pixel value is assigned a separate colour. This is implemented by using a lookup table to produce each of the red, green and blue components, as shown in Figure 6.29. Its usefulness relies on the ability of the human visual system to more readily distinguish colours than shades of grey, particularly when the local contrast is different (see, for example, Figure 6.3). An appropriate pseudocolour both enhances the contrast, and can facilitate the manual selection of an appropriate threshold level.

6.3.2 Colour Space Conversion

While the RGB colour space is native to most display devices, it is not necessarily the most natural to work in. Colour space conversion involves transforming one vector representation into another that makes

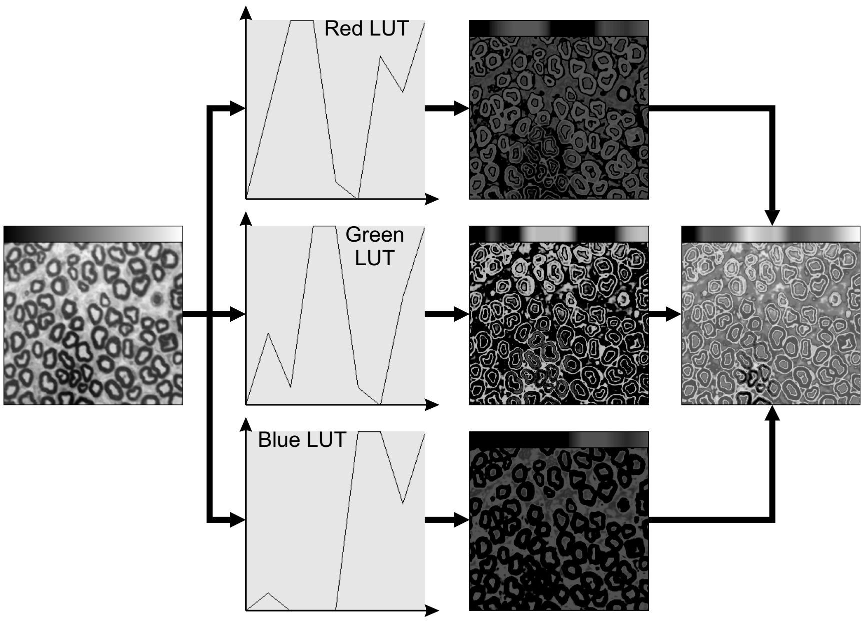


Figure 6.29 Pseudocolour or false colour mapping using lookup tables. (*See colour version of this figure in colour plate section*)

subsequent analysis or processing easier. Colour space conversion is a point operation, since each pixel is operated on independently.

6.3.2.1 RGB

The RGB colour space is referred to as *additive*, because a colour is made by adding particular levels of red, green and blue light. Figure 6.30 illustrates this with the RGB components of a colour image.

The RGB colour components (whether from capture or display) are device dependent. In a scene captured by a camera, the colour vector for each pixel depends not only on the colour in the scene and the illumination, but also on the spectral response of the filters used to measure the red, green and blue components. Similarly, the actual colour produced on a display will depend on the spectral content of the red, green and blue light sources within the display. Therefore, there are many different RGB colour spaces depending on the particular wavelengths (or spectral mix) used for each of the red, green and blue primaries.

Two variations of RGB are of particular note. The first is 16-bit RGB. This is used where the entire colour vector must be contained within 16 bits, usually as a result of bandwidth limitations. With 16-bit RGB, five bits are allocated for each of the R and B components, and six bits are allocated for G (to take into account the increased sensitivity of the human visual system to green). To convert from 16-bit to 24-bit RGB, rather than append zeros, it is better to append the two (for G) or three (for R and B) most significant bits of the component as shown in Figure 6.31.

The main limitation of the RGB colour space is that it is device dependent. To combat this, a device independent colour space was defined: sRGB (Stokes *et al.*, 1996). This is a defacto standard for consumer colour devices, including cameras, printers and displays. It not only defines the specific colours of the three primaries used for red, green and blue, but also defines a nonlinear gamma-like mapping between the intensity and the numerical values that closely approximates the response of CRT-based displays.

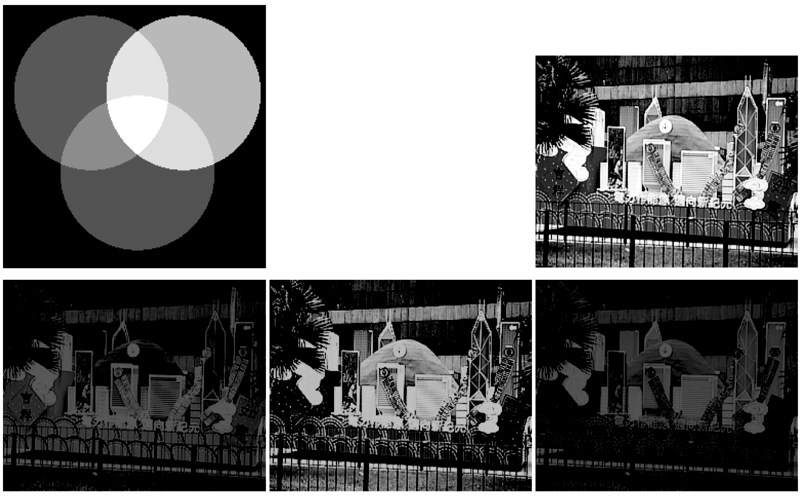


Figure 6.30 RGB colour space. Top left: combining red, green and blue primary colours; bottom: the red, green and blue components of the colour image on the top right. (*See colour version of this figure in colour plate section*)

Conversion from a device-dependent RGB to sRGB requires first multiplying the device dependent colour vector by a 3 × 3 matrix that depends on the red, green and blue spectral characteristics of the device. This transformation, determined by calibration, is sometimes called the *colour profile* of the device. The result of the transformation represents a vector of normalised (between zero and one) linear RGB values. Any component that is outside this range is outside the gamut of colours that may be reproduced within sRGB and is clipped to fall within this range. Each component, C_N, is then mapped from linear RGB to sRGB values by (Stokes *et al.*, 1996):

$$C_{NsRGB} = \begin{cases} 12.92 C_{NRGB}, & C_{NRGB} \leq 0.0031308 \\ 1.055 C_{NRGB}^{1/2.4} - 0.055, & C_{NRGB} > 0.0031308 \end{cases} \tag{6.45}$$

The final values can then be represented as 8-bit quantities by scaling by 255.

Equation 6.45 can be easily inverted to remove the gamma component and return to linear components:

$$C_{NRGB} = \begin{cases} \dfrac{C_{NsRGB}}{12.92}, & C_{NRGB} \leq 0.04045 \\ \left(\dfrac{C_{NsRGB} + 0.055}{1.055}\right)^{2.4}, & C_{NRGB} > 0.04045 \end{cases} \tag{6.46}$$

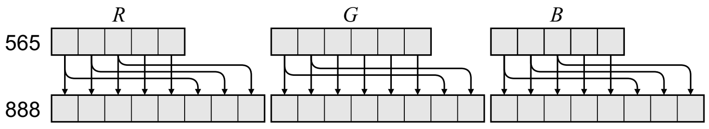


Figure 6.31 Converting from RGB565 to RGB888.

6.3.2.2 CMY and CMYK

In printing, rather than actively producing light, the image begins with the white paper, and colour is produced by filtering or blocking some of the colour. The *subtractive* primaries are cyan, magenta and yellow, and usually consist of appropriate inks, dyes or filters. Consider the yellow dye; it will allow the red and green spectral components to pass through, but will attenuate the blue, making that part of the scene appear yellow. The more yellow, the more the blue is attenuated, the yellower the scene will appear. Similarly, the magenta dye attenuates the green spectral component, and the cyan dye attenuates the red component. A CMY image is therefore formed from the cyan, magenta and yellow components:

$$\mathbf{I}_{CMY} = \{C, M, Y\}$$

Therefore, as illustrated in Figure 6.32, the colours of a scene may be produced by mixing different quantities of yellow, magenta and cyan dyes to give the required spectral content at each point. If the RGB components are normalised then the corresponding normalised CMY components are approximately given by:

$$\begin{aligned} C_N &= 1 - R_N \\ M_N &= 1 - G_N \\ Y_N &= 1 - B_N \end{aligned} \tag{6.47}$$

The exact relationship depends on the spectral content of the RGB components and the spectral transmissivity of the CMY components. While the simple conversion of Equation 6.47 works reasonably well with lighter, unsaturated colours, it becomes less accurate for darker and more saturated colours where one (or more) of the CMY components is larger. There are two reasons for this. Firstly, the attenuation is not linear with the amount of dye used, but is exponential. This makes the relationship approximately linear for lower levels of CMY components, but deviates more for the higher levels. Consequently, a particular spectral component cannot be completely removed, making it difficult to produce fully saturated colours and dark colours. The second reason is that equal levels of yellow, magenta

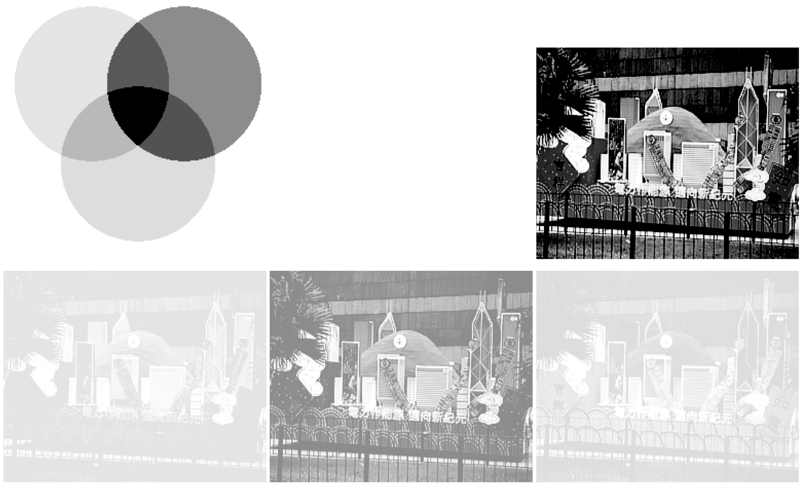


Figure 6.32 CMY colour space. Top left: combining yellow, magenta and cyan secondary colours; bottom: the yellow, magenta and cyan components of the colour image on the top right. (*See colour version of this figure in colour plate section*)

and cyan seldom result in a flat spectral response. Both of these factors make black appear as a muddy colour often with a colour cast.

In printing, this problem is overcome with the addition of a black dye, resulting in the CMYK colour space. Equal amounts of yellow, magenta and cyan dyes are replaced by the appropriate amount of black dye. This changes the approximate conversion from Equation 6.47 to:

$$\begin{aligned} K_N &= 1-\max(R_N, G_N, B_N) \\ C_N &= 1-K_N-R_N = \max(R_N, G_N, B_N)-R_N \\ M_N &= 1-K_N-G_N = \max(R_N, G_N, B_N)-G_N \\ Y_N &= 1-K_N-B_N = \max(R_N, G_N, B_N)-B_N \end{aligned} \tag{6.48}$$

Note that one of the CMYK components will always be zero. Other than for printing, the CMY or CMYK colours spaces are not often used for image processing.

6.3.2.3 YUV, YIQ and YCbCr

When colour television was introduced, backward compatibility with existing black and white television was desired. The luminance signal, Y, is a combination of RGB components, with the colour provided by two colour difference signals, $B-Y$ and $R-Y$. Assuming normalised RGB, the image can be represented in YUV colour space, $\mathbf{I}_{YUV} = \{Y, U, V\}$, with the components given by:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ U &= 0.492(B-Y) \\ V &= 0.877(R-Y) \end{aligned} \tag{6.49}$$

The particular weights given for the Y component reflect the relative sensitivities of the human visual system. Representing Equation 6.49 in matrix form gives:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.50}$$

The range of the V component is from -0.615 to 0.615, which requires an extra bit to represent. When processing digitally, it is more common to adjust the scale factors of the U and V components so both are in the range -0.5 to 0.5 (for example as used by the JPEG2000 standard (ISO, 2000)). This makes the transform matrix:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.51}$$

The YIQ colour space is very similar to the YUV, except for different weights in Equation 6.50:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.52}$$

The difference is that the chrominance components (U, V) are rotated by 33 degrees. This requires a full matrix multiplication (nine multiplications) rather than the five of Equation 6.49. The advantage gained is a reduction in bandwidth required for television broadcasting because the human visual system is less sensitive to the Q component than to the I component. The YIQ colour space, however, is seldom used for digital image processing.

Strictly speaking, the YUV colour space is an analogue representation. The corresponding digital representation is called YCbCr, with $\mathbf{I}_{YCbCr} = \{Y, Cb, Cr\}$. There are two commonly used YCbCr formats: one scales the values by less than 255 to give 8-bit quantities with headroom and footroom and offsets the chrominance components to enable an unsigned representation:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.0 \\ 112.0 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R_N \\ G_N \\ B_N \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \tag{6.53}$$

and the other uses the full range of 8-bit outputs for each of the components. This latter case is often used with 8-bit input images (from 0–255) where the headroom and footroom are not considered as important as maximising the dynamic range:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{6.54}$$

The implementation of Equation 6.54 will be considered here. This matrix can be factorised to use four multiplications:

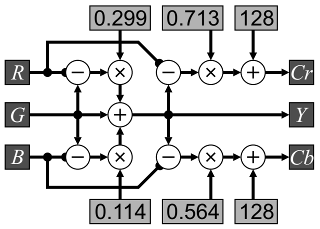
$$\begin{aligned} Y &= 0.299(R-G) + 0.114(B-G) + G \\ Cb &= \frac{0.5}{(1-0.114)}(B-Y) + 128 = 0.564(B-Y) + 128 \\ Cr &= \frac{0.5}{(1-0.299)}(R-Y) + 128 = 0.713(R-Y) + 128 \end{aligned} \tag{6.55}$$

with the implementation shown in Figure 6.33. The inverse similarly requires four non-trivial multiplications.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ Cb-128 \\ Cr-128 \end{bmatrix} \tag{6.56}$$

Since the multiplications are by constants, they may be implemented efficiently using a few additions. The addition and subtraction of 128 can be accomplished by just inverting the most significant bit. The YCbCr components of a colour image are illustrated in Figure 6.34.

From an image processing perspective, the advantage of using YUV or YCbCr over RGB is the reduction in correlation between the channels. This is particularly useful with colour thresholding as described in the next section. It also simplifies the enhancement of colour images. For example, contrast enhancement (such as histogram equalisation) may be performed on the Y component of the image.



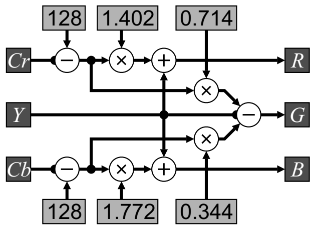


Figure 6.33 Left: conversion from RGB to YCbCr; right: conversion from YCbCr to RGB.

There are two limitations of the YUV (or YCbCr colour space). The first is that the transformation is a (distorted) rotation of the RGB coordinates. The scale factors are chosen to ensure that every RGB combination has a legal YCbCr representation; however, the converse is not true. Some combinations of YCbCr fall outside the legal range of RGB values. The implication is that more bits must be kept in the YCbCr representation if the transform is to be reversed. Note that this cannot be avoided when using rotation based transformation of the colour space.

The second limitation is that it requires either floating-point or fixed-point multiplications to perform the transformation. This problem is primarily caused by the different weights for the Y component, which are based on human perception of the luminance. From an image processing point of view, this is often less relevant and can be relaxed.

The obvious solution to this problem is to restrict the coefficients to powers of two. There are several ways of accomplishing this. One is the reversible colour transform (RCT) used by lossless coding in JPEG2000 (ISO, 2000). This has:

$$\begin{aligned} Y &= \left\lfloor \frac{R+2G+B}{4} \right\rfloor \\ Cr &= R-G \\ Cb &= B-G \end{aligned} \tag{6.57}$$

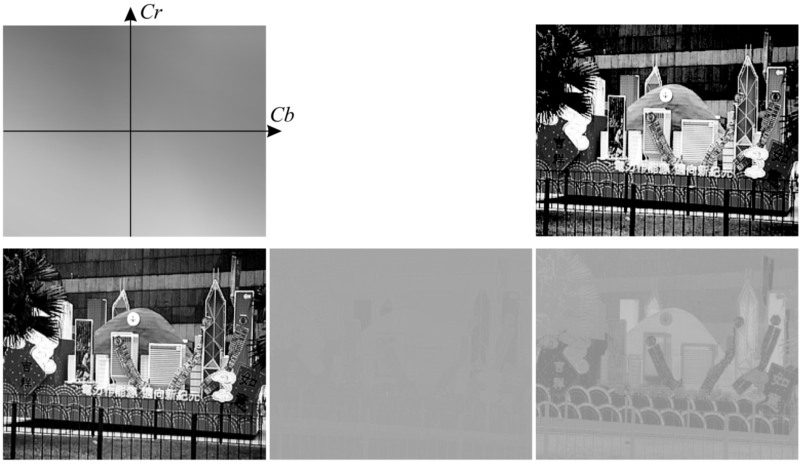


Figure 6.34 YCbCr colour space. Top left: the $Cb-Cr$ colour plane at mid luminance; bottom: the luminance and chrominance components of the colour image on the top right. (*See colour version of this figure in colour plate section*)

where $\lfloor\ \rfloor$ represents truncation. Although information appears to be lost by the truncation, it is retained in the other two terms and can be recovered exactly (hence reversible colour transform). The *Cr* and *Cb* terms would appear to require an extra bit to prevent wrap around, and this is mandated in the JPEG2000 standard, but this is actually unnecessary, since the *Y* term provides sufficient information to resolve the ambiguity. The reverse transformation is:

$$\begin{aligned} R &= Y - \left\lfloor \frac{Cr + Cb}{4} \right\rfloor \\ G &= Cr + G \\ B &= Cb + G \end{aligned} \tag{6.58}$$

While this solution is good from a coding perspective, for many image processing algorithms the ambiguity between positive and negative values of *Cr* and *Cb* would need to be resolved by retaining the extra bit.

The RCT is not orthogonal; in particular, there is a strong correlation caused by the significant sharing of the *G* component. One YUV-like orthogonal transformation is (Sen Gupta *et al.*, 2004; Sen Gupta and Bailey, 2008):

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.59}$$

A similar transformation is:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 1 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.60}$$

The problem with these is that the inverse transformations require division by three. An orthogonal transformation that uses powers of two for both the forward and inverse transformations is not possible. A minor adjustment to Equations 6.59 and 6.60 gives a simple transform that is nearly orthogonal and is also easily inverted. Here it is represented in normalised form:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.61}$$

This may be further factorised to reduce the complete transformation to four additions, with the ÷2 performed by a shift, which is free in hardware. The implementation of this, and its inverse:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 0 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \tag{6.62}$$

are shown in Figure 6.35.

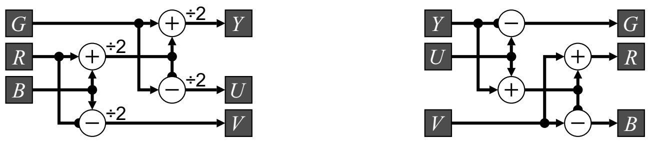


Figure 6.35 A multiplier-less YUV-like transformation and its inverse.

6.3.2.4 HSV and HLS

The RGB or YUV colour spaces do not reflect the psychological way in which colour is interpreted or thought about. When a colour is interpreted, it is considered primarily in terms of its hue and the strength of the colour (saturation) rather than the individual RGB components. Therefore, it is reasonable to have a colour space with hue, saturation and intensity as the three components. Two such colour spaces are $\mathbf{I}_{HSV} = \{H, S, V\}$ (hue, saturation and value) and $\mathbf{I}_{HLS} = \{H, L, S\}$ (hue, lightness, and saturation) (Foley and Van Dam, 1982).

The HSV colour space is typically represented as a cone as shown in the left panel of Figure 6.36. The hue represents the angle around the cone, resulting in the colour wheel on the right panel of Figure 6.36. By definition, the black-white axis up the centre of the cone has a hue of zero. The remainder of the colour wheel is split into three sectors based on which component is the maximum, and the proportion between the other two components is used to give the angle. This is illustrated graphically in the colour wheel in Figure 6.36.

Mathematically, the hue is defined as:

$$H = \begin{cases} 0, & R = G = B \\ \dfrac{(G-B)60^\circ}{\max(R,G,B)-\min(R,G,B)} \bmod 360^\circ, & R \geq G, B \\ \dfrac{(B-R)60^\circ}{\max(R,G,B)-\min(R,G,B)} + 120^\circ, & G \geq R, B \\ \dfrac{(R-G)60^\circ}{\max(R,G,B)-\min(R,G,B)} + 240^\circ, & B \geq R, G \end{cases} \tag{6.63}$$

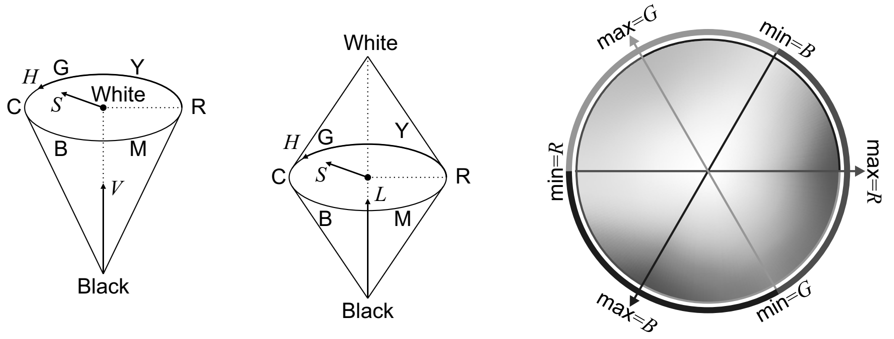


Figure 6.36 HSV and HLS colour spaces. Left: the HSV cone; centre: the HLS bi-cone; right: the hue colour wheel. (*See colour version of this figure in colour plate section*)

With a binary number system, the use of degrees (or even radians) is inconvenient. There are two alternatives. One is to normalise the angle so that a complete cycle goes from zero to one, which maximises the hue resolution for a given number of bits and avoids the need to manage wraparound of values outside the range of 0–360°. This may be preferable when performing many manipulations on the hue. The other is to represent 60° by a power of two (for example 64) to simplify the multiplication. It also makes it easier to convert back to RGB.

The value is similar to Y, except that equal weight is given to the red, green and blue components, rather than weighting them based on human perception. The V component represents the height up the cone. The value is usually represented normalised between zero and one.

$$V_N = \max(R_N, G_N, B_N) \tag{6.64}$$

The saturation represents the strength of the colour and is represented by the radius of the cone relative to the maximum radius for a given value:

$$S_{HSV} = \begin{cases} 0, & \max(R,G,B) = 0 \\ \dfrac{\max(R,G,B) - \min(R,G,B)}{\max(R,G,B)}, & \text{otherwise} \end{cases} \tag{6.65}$$

This normalisation means only the value changes as the intensity of the image is scaled. The HSV components of a sample image are shown in the middle row of Figure 6.37. Note that for greys in the input image the hue is meaningless, because even a small amount of noise on the components will determine the hue.

The HLS representation is slightly different from the HSV colour space in that it is a bi-cone rather than a cone, as seen in the centre panel of Figure 6.36. The main advantage of HLS over HSV is that it is symmetric with respect to black and white. The hue is the same for both representations, given by Equation 6.63, but the lightness and saturation differ from the HSV space. Again, the lightness and saturation are usually represented as normalised values.

$$L_N = \tfrac{1}{2}(\max(R_N, G_N, B_N) + \min(R_N, G_N, B_N)) \tag{6.66}$$

$$S_{HLS} = \begin{cases} 0, & L_N = 0 \\ \dfrac{\max(R,G,B) - \min(R,G,B)}{2L_N}, & L_N < \frac{1}{2} \\ \dfrac{\max(R,G,B) - \min(R,G,B)}{2 - 2L_N}, & L_N \geq \frac{1}{2} \end{cases} \tag{6.67}$$

The HLS components of the sample image are shown in the bottom row of Figure 6.37.

Equation 6.66 places fully saturated primary colours at half lightness, at the intersection of the two cones. In general, this makes L less than V, although this has no real consequence from an image processing perspective.

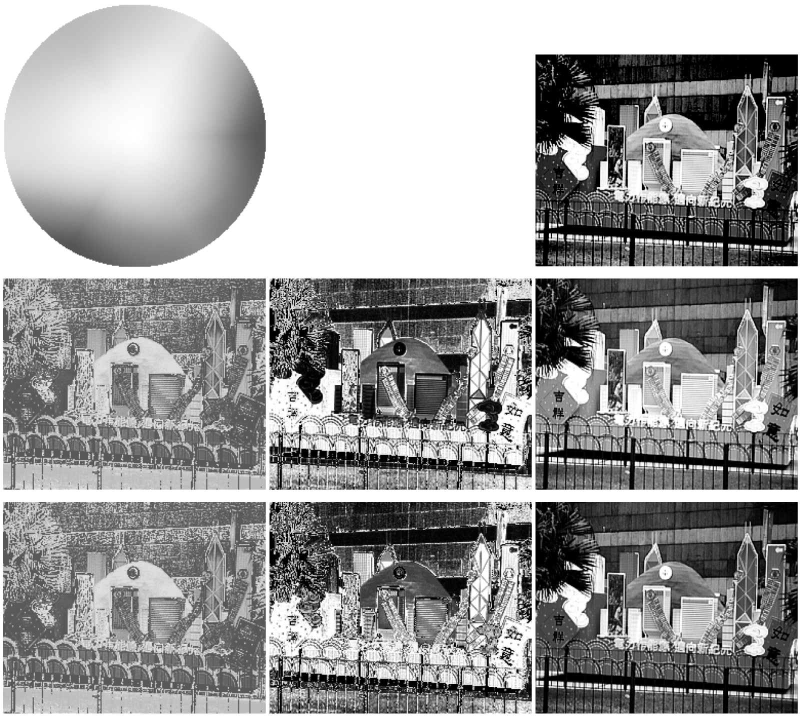


Figure 6.37 HSV and HLS colour spaces. Top left: HSV hue colour wheel, with saturation increasing with radius; middle row: the HSV hue, saturation and value components of the colour image on the top right; bottom row: the HLS hue, saturation and lightness components. (*See colour version of this figure in colour plate section*)

The major limitation of HLS is that the saturation is only constant with scaling lightness only in the lower cone ($L_N \leq \frac{1}{2}$). This gives the anomaly that pastel colours can appear as fully saturated. In particular, this can be seen in the lighter areas of Figure 6.37, where the saturation is high.

For these reasons, only the implementation of HSV will be considered here. Figure 6.38 shows an implementation of the conversion from RGB to HSV. The minimum and maximum components are determined from the sign bits of the differences between the input colour channels. These differences need to be calculated anyway to give the numerators of Equation 6.63, so this information is effectively obtained for free. Apart from the multiplexers to select the minimum and maximum components, the main complexity is the two dividers for normalising the hue and the saturation. The colour wheel is rotated so that a hue of zero corresponds to magenta to save the modulo normalisation when the maximum pixel is red. To achieve this, the offset added to the hue is given by:

$$Offset = 256 \cdot B_{\max} + 128 \cdot G_{\max} + 64 \tag{6.68}$$

with the hue taking a value between zero and 383. If necessary, this can be shifted back by subtracting 64, and then adding 384 if the result goes negative.

To convert HSV back to RGB, the three most significant bits of the hue represent the sector, hence can be used to select the appropriate values for the RGB components:

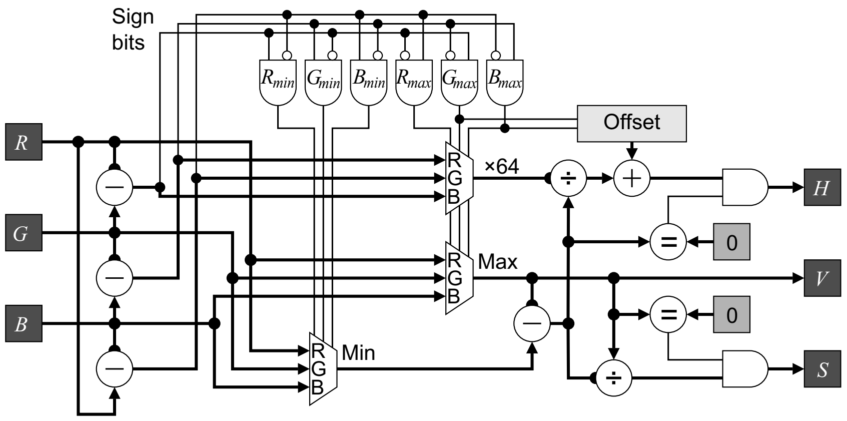


Figure 6.38 Conversion from RGB to HSV.

$$\mathbf{I}_{RGB} = \begin{cases} \{V, V-VS, V-VSH_{LSB}\}, & H_{MSB}=0 \\ \{V, V-VS(1-H_{LSB}), V-VS\}, & H_{MSB}=1 \\ \{V-VSH_{LSB}, V, V-VS\}, & H_{MSB}=2 \\ \{V-VS, V, V-VS(1-H_{LSB})\}, & H_{MSB}=3 \\ \{V-VS, V-VSH_{LSB}, V\}, & H_{MSB}=4 \\ \{V-VS(1-H_{LSB}), V-VS, V\}, & H_{MSB}=5 \end{cases} \tag{6.69}$$

where the least significant bits of hue are considered as a fraction. An implementation of Equation 6.69 is shown in Figure 6.39.

A form of hue and saturation may also be derived from the YCbCr colour space by converting the chrominance components into polar coordinates (Punchihewa *et al.*, 2005). This may be readily implemented using a CORDIC transformation.

$$H = \tan^{-1}\frac{Cb}{Cr} \tag{6.70}$$

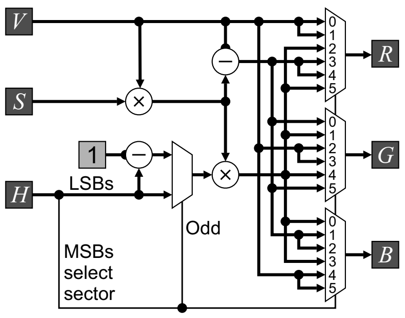


Figure 6.39 Conversion from HSV to RGB.

$$S = \sqrt{Cb^2 + Cr^2} \tag{6.71}$$

In this form, the saturation will scale with intensity, which is generally not desired. To make the saturation more meaningful, it is necessary to scale the saturation by either the Y or the maximum RGB component so that it is not intensity dependent.

The HSV colour space is useful in image processing for colour detection and enhancement. The advantage gained is the intensity independence of hue and saturation, enabling more robust segmentation. One limitation, however, is the need to have a correct colour balance. The hue (and also the saturation to a lesser degree) is affected by any colour cast, especially for colours with low saturation. Section 6.3.4 considers correcting the colour balance of an image.

In some applications, only one part of the colour wheel is of interest. For example, many fruits vary in colour from green to yellow or red. In this case the sectors between green and red are of particular interest, whereas outside this range it less important. In these cases, a simplified substitute for hue may be used. For example, when analysing the colour of limes, colours in the range from green (chlorophyll dominates) to yellow (carotenoids dominate) are important (Bunnik *et al.*, 2006); these may be captured by a ratio of the components:

$$H_{G-Y} = \frac{R}{G} \tag{6.72}$$

This measure ranges from zero for pure green to one for yellow, enabling the health of the fruit to be assessed. Such a measure is less suitable for reds (such as would be encountered when grading tomatoes, where the red pigmented lycopene dominates), because the red component can be much larger than the green. A small change in the green component can result in a large change in the measure. An alternative hue measure that is more uniform is (Bunnik *et al.*, 2006):

$$H_{G-R} = \frac{R-G}{R+G} \tag{6.73}$$

which ranges from -1 for pure greens through 0 for yellow to $+1$ for pure reds. Note that, as with the hue, both the measures of Equations 6.72 and 6.73 require the correct scaling or balance between the red and green channels to accurately reflect the colour.

In other applications, other colour distinctions may be important, enabling similar simplifications to be used.

6.3.2.5 CIE XYZ and xyY

The limitation, as mentioned before, of RGB is that it is device dependent. In 1931, the International Commission on Illumination (CIE) established the XYZ standard colour space. The X, Y and Z are imaginary (in the sense that they do not correspond to any real illumination) tristimulus values chosen such that the Y corresponds with perceived intensity, and all possible visible colours are represented by a combination of positive values of XYZ (Hoffmann, 2000). (It is impossible to produce all visible colours with only three real sources.)

One limitation of a three-dimensional colour space is that it is difficult to represent graphically in two dimensions. Since the underlying colour does not change with changing intensity, a normalised colour space $\mathbf{I}_{xyz} = \{x, y, z\}$ may be derived:

$$\begin{aligned} x &= \frac{X}{X+Y+Z} \\ y &= \frac{Y}{X+Y+Z} \\ z &= \frac{Z}{X+Y+Z} = 1-x-y \end{aligned} \tag{6.74}$$

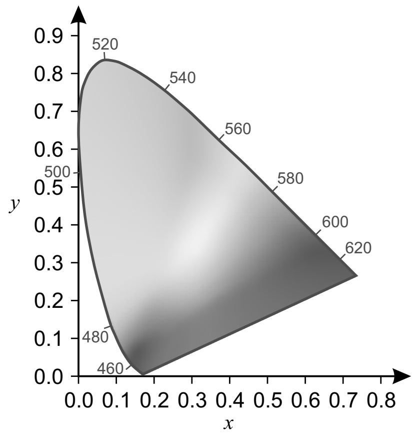


Figure 6.40 Chromaticity diagram. The numbers are wavelengths of monochromatic light in nanometres. (*See colour version of this figure in colour plate section*)

which then allows the colour or chromaticity to be represented by two of the components.

The x–y plane is usually used to define the colour, resulting in the chromaticity diagram of Figure 6.40. Given two points on the chromaticity diagram corresponding to two colour sources, a linear combination of those sources will lie on a straight line between those points. Therefore, any three points, corresponding for example to red, green and blue light sources, can be mixed to produce only the colours within the triangle defined by those three points. To go outside the triangle would require one (or more) of the sources to have a negative intensity, which is clearly impossible. The triangle, therefore, defines the gamut of colours that can be produced by those primaries, for example on a display.

To convert from a chromaticity back to XYZ tristimulus values, it is also necessary to know the intensity, Y, resulting in the xyY representation. The X and Z values may be recovered from

$$\begin{aligned} X &= \frac{Y}{y}x \\ Z &= \frac{Y}{y}(1-x-y) \end{aligned} \tag{6.75}$$

With any linear tristimulus colour space, the component values may be transformed into any other linear colour space simply by a matrix multiplication (Hoffmann, 2000). The matrix used depends, of course, on the spectral characteristics of the particular components used. For example, the primaries used in HDTV have the chromaticity values listed in Table 6.1. The white point defines the chromaticity coordinates of white, defined as the colour observed when all three RGB components are equal.

The corresponding transformation matrix and its inverse can then be derived as (Hoffmann, 2000):

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4984 \\ -0.9692 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{6.76}$$

Table 6.1 Chromaticity values for HDTV and sRGB

	x	y	z
Red	0.6400	0.3300	0.0300
Green	0.3000	0.6000	0.1000
Blue	0.1500	0.0600	0.7900
White point	0.3127	0.3290	0.3583

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.4124 & 0.3576 & 0.1805 \\ 0.2126 & 0.7152 & 0.0722 \\ 0.0193 & 0.1192 & 0.9505 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{6.77}$$

If device independence is not a requirement, a device-dependent chromaticity may also be formed. This normalises the RGB components in a similar manner to Equation 6.74:

$$\begin{aligned} r &= \frac{R}{R+G+B} \\ g &= \frac{G}{R+G+B} \\ b &= \frac{B}{R+G+B} = 1-r-g \end{aligned} \tag{6.78}$$

with the device dependent $r-g$ chromaticity shown in Figure 6.41.

6.3.2.6 CIE L*a*b* and CIE L*u*v*

One limitation of the colour spaces mentioned so far is that they are not perceptually uniform. In estimating colour differences or colour errors, it is useful if the distance between two points is a measure of

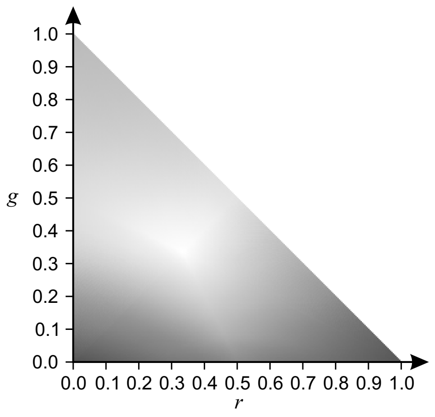


Figure 6.41 Device dependent $r-g$ chromaticity. (*See colour version of this figure in colour plate section*)

the perceived colour difference. Several colour spaces have been derived from the CIE XYZ space that are perceptually more uniform. Two considered here are the CIE L*a*b* and CIE L*u*v* colour spaces.

The L*a*b* space is defined relative to a reference white point $\{X_n, Y_n, Z_n\}$ and introduces the following nonlinearity to make the space more uniform:

$$f(C) = \begin{cases} C^{1/3}, & C > \left(\frac{6}{29}\right)^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 C + \frac{4}{29}, & \text{otherwise} \end{cases} \tag{6.79}$$

The conversion is to $\mathbf{I}_{L^*a^*b^*} = \{L^*, a^*, b^*\}$ is then:

$$\begin{aligned} L^* &= 116 f(Y/Y_n) - 16 \\ a^* &= 500(f(X/X_n) - f(Y/Y_n)) \\ b^* &= 200(f(Y/Y_n) - f(Z/Z_n)) \end{aligned} \tag{6.80}$$

resulting in the lightness, L^*, ranging from 0 to 100. A different range may be arranged by appropriate scaling. The a^* axis goes from green (negative) to red or magenta (positive), and the b^* axis goes from blue (negative) to yellow (positive).

The function, f, may be implemented using a lookup table, although three will be required to convert one pixel per clock cycle, one for each component. If the reference white point does not change, the division can be combined into the tables.

The inverse conversion:

$$\begin{aligned} f(Y/Y_n) &= \frac{L^* + 16}{116} \\ f(X/X_n) &= \frac{a^*}{500} + f(Y/Y_n) \\ f(Z/Z_n) &= f(Y/Y_n) - \frac{b^*}{200} \end{aligned} \tag{6.81}$$

requires three divisions before inverting Equation 6.79:

$$C = \begin{cases} f(C)^3, & f(C) > \frac{6}{29} \\ 3\left(\frac{6}{29}\right)^2 \left(f(C) - \frac{16}{116}\right), & \text{otherwise} \end{cases} \tag{6.82}$$

which is again best implemented as a lookup table. The resultant components then need to be scaled by the reference white point components, which can again be combined into the lookup table if constant.

The L*u*v* colour space was designed to be a little easier to calculate than L*a*b*, although on an FPGA there may be little difference. The lightness is the same as Equation 6.80, with the two chrominance components given by:

$$\begin{aligned} u^* &= 13L^*(u' - u'_n) \\ v^* &= 13L^*(v' - v'_n) \end{aligned} \tag{6.83}$$

where
$$\begin{aligned} u' &= \frac{4X}{X+15Y+3Z} \\ v' &= \frac{9Y}{X+15Y+3Z} \end{aligned} \tag{6.84}$$

The chrominance components range from −100 to 100. Again, a more convenient range may be achieved by scaling L^*.

Conversion from L*u*v* back to XYZ is:

$$\begin{aligned} u' &= \frac{u^*}{13L^*} + u'_n \\ v' &= \frac{v^*}{13L^*} + v'_n \\ Y &= \begin{cases} Y_n\left(\dfrac{3}{29}\right)^3 L^*, & L^* \le 8 \\ Y_n\left(\dfrac{L^*+16}{116}\right)^3, & L^* > 8 \end{cases} \\ X &= Y\frac{9u'}{4v'} \\ Z &= Y\frac{12-3u'-20v'}{4v'} \end{aligned} \tag{6.85}$$

The colour difference between two points in either the L*a*b* or L*u*v* colour space may be determined by calculating the Euclidean distance between the corresponding vectors.

6.3.3 Colour Thresholding

Colour thresholding, like scalar thresholding, assigns a label to each pixel. With colour images, the purpose is to detect which pixels belong to each of a set of colours of interest. The output of colour thresholding is an image of labels, with each label corresponding to a colour class.

A computationally simple approach is to associate a rectangular box in the colour coordinates with a colour class. This corresponds to using a pair of thresholds for each component to define the boundaries of the box along that component.

It is generally not appropriate to use RGB space for this unless the illumination is fixed. This is because there is a strong correlation between the red, green and blue components, and all three will scale with illumination. This means that as the intensity changes, points will move diagonally in RGB space, requiring the box to be large. Consequently, only a few very different colours can be detected, and there is poor discrimination between colours.

Converting to YCbCr will give some improvement because rectangular boxes aligned with the YCbCr axes will be diagonal in RGB space. However, the chrominance components still scale with the luminance. Better discrimination may be obtained by scaling the chrominance by the luminance, or alternatively scaling the chrominance components by the maximum of the red, green and blue components (Johnston *et al.*, 2005b).

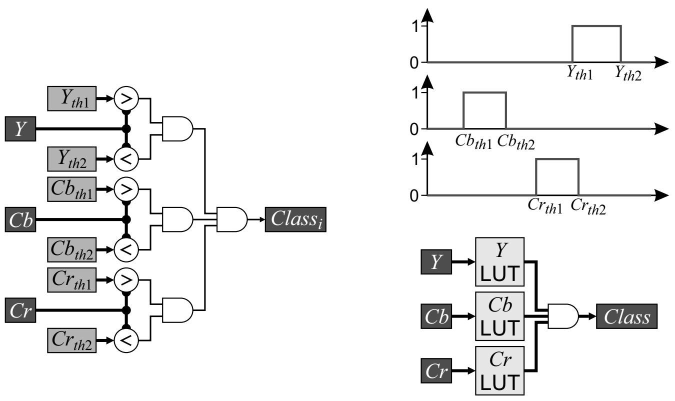


Figure 6.42 Colour thresholding. Left: comparisons required for each colour class using a rectangular block; right: lookup table approach, with one lookup table per colour channel, detects multiple classes in parallel.

Alternatively, the thresholding may be performed directly in the HSV colour space where the hue and saturation are independent of the luminance. Note that the HLS colour space is usually less suitable than HSV because the saturation changes with lightness when the lightness is greater than 50%.

The implementation of colour thresholding extends the contouring operation of Equation 6.8 and Figure 6.14 to each of the three channels, as shown in the left in Figure 6.42. This circuit will need to be repeated for each colour class to be detected.

An alternative is to use a lookup table to implement the thresholding. This requires a separate LUT for each channel, but has the advantage that multiple classes may be segmented simultaneously by having a separate bit-plane associated with each colour class in the lookup tables (Sen Gupta *et al.*, 2004).

The set of pixel values corresponding to a colour class do not necessarily need to fall within a rectangular box aligned with the component axes. However, the logic becomes increasingly more complex to handle arbitrary shaped regions. One approach is to use a single large lookup table taking as input the concatenated components of the colour input vector (Sen Gupta *et al.*, 2003). The mapping can then be arbitrarily complex and can include any necessary colour space conversions. Unfortunately, such a lookup table is prohibitively large ($2^{24} = 16\ 777\ 216$ entries for 24-bit colour) and would need to be implemented in external memory. The table size may be reduced by using fewer bits of the input at the cost of poorer discrimination.

A composite approach can also be used. For example, Johnston *et al.* (2005a) use two lookup tables, for each of *Cb* and *Cr*, and combine the normalisation division by *Y* into those tables, with a reduced number of bits. The resultant tables had only 512 entries of four bits (for four colour classes) enabling both tables to be contained within a single 4 Kbit block RAM.

6.3.4 Colour Correction

The colour of each pixel observed in an image depends not only on the spectral reflectivity of the corresponding object point, but also on the illumination. For example, the colours of objects within a scene illuminated by incandescent light can be quite different from those illuminated by fluorescent light. For outdoor scenes, the spectral characteristics of the illumination depend on whether the object is illuminated by sunlight, or in shadow, or whether the sky is clear or overcast, or even on the time of day.

The colour constancy problem is that of estimating how the scene would appear given canonical illumination. Unfortunately, without prior knowledge of the illumination, this is an ill-posed problem. In practise, it is not necessary to estimate the full spectral characteristics of the illumination; it is sufficient to remove the effects of varying illumination on the image. Colour correction involves transforming the RGB value of each pixel in the image to remove the effects of illumination. In general, this requires determining the components of a 3×3 linear transformation matrix. However, given that the spectral distribution at the sensor is the product of the reflectivity and illumination distributions, this is often simplified to a diagonal matrix, independently scaling the R, G, and B components (Finlayson *et al.*, 1994). Note that if any of the components are saturated, this will result in an additional colour change that cannot easily be corrected.

Perhaps the simplest model is to assume that, on average, the image is achromatic (Finlayson *et al.*, 2001). This implies that the average colour within the image should be grey. The effective illumination colour is then given by the mean of the red, green and blue components of the image. Correction is performed by dividing by the mean value and scaling to ensure that all pixel values are within the range of allowed values.

$$\begin{aligned}\hat{R} &= kR/\mu_R \\ \hat{G} &= kG/\mu_G \\ \hat{B} &= kB/\mu_B\end{aligned} \tag{6.86}$$

where $\hat{R}\hat{G}\hat{B}$ are the corrected colour components and the scaling factor, k, that maximises the output contrast is given by:

$$k = V \min\left(\frac{\mu_R}{\max\limits_{x,y}(R)}, \frac{\mu_G}{\max\limits_{x,u}(G)}, \frac{\mu_B}{\max\limits_{x,y}(B)}\right) \tag{6.87}$$

where $V = 2^N - 1$. Alternatively, the scaling factor can be set to $V/2$ (Funt *et al.*, 1998), although this has the danger that one or more of the components may saturate.

The major limitation of this approach is that if the image contains a dominant colour, the estimate of the illumination colour will be biased towards that colour (Gershon *et al.*, 1987). As a result, a colour cast complementary to the dominant colour will be introduced into the image. This is clearly seen in the centre panel of Figure 6.43, where a bluish cast is introduced. As it is not known in general whether or not there is a dominant colour, this approach to colour correction is impractical.

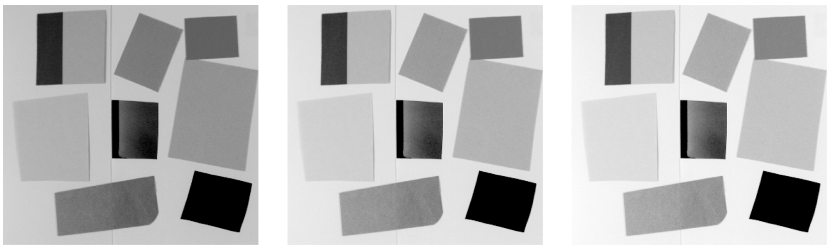


Figure 6.43 Simple colour correction. Left: original image captured under incandescent lights, resulting in a yellowish-red cast; centre: correcting assuming the average is grey, using Equation 6.86; right: correcting assuming the brightest pixel is white, using Equation 6.88. (*See colour version of this figure in colour plate section*)

Another assumption that is sometimes used for colour correction is that the image contains at least one white pixel within the image, and that the white region is the brightest in the image. In this case, the white pixel may be found simply by finding the maximum of each component. Since this should be white (achromatic, with all components equal), the input image can be corrected by scaling each component by its corresponding maximum (Funt *et al.*, 1998):

$$\begin{aligned}\hat{R} &= VR/\max_{x,y}(R)\\ \hat{G} &= VG/\max_{x,y}(G)\\ \hat{B} &= VB/\max_{x,y}(B)\end{aligned} \tag{6.88}$$

The results of this scheme are shown in the right panel of Figure 6.43.

One limitation of this approach is that it is easy for the assumptions to be violated. If the white pixel is the brightest in the image, there is a strong possibility that one or more of the channels will be saturated at V. The true value of that component will be underestimated, affecting the correction. In many cases, the brightest point in an image is the result of specular reflection with the resulting pixel affected by the colour of the reflecting surface. Also, basing the correction on a single pixel will inevitably introduce noise into the estimate of the illumination colour, affecting the accuracy of any correction. These limitations may be mitigated by requiring a region of adjacent pixels be detected, and using the mean of that region. The logic for detection, however, becomes considerably more complex.

The brightest object in the image may not be white, and if it is white it may not be pure white. If the colour is slightly off-white, making it white will introduce a cast. This limitation is harder to overcome, because it is impossible to distinguish between an off-white surface or slightly coloured illumination.

This approach may be extended by deliberately placing a white patch within the image. If the illumination and camera settings do not change with time, this may even be captured off-line as a calibration image. The white patch can then be detected either automatically, or by having the patch in a known location, or by manually selecting the region of interest. Let $\mu_{(white)}$ be the mean of the pixels within the white patch, then the correction may be performed as:

$$\begin{aligned}\hat{R} &= VR/\mu_{R(white)}\\ \hat{G} &= VG/\mu_{G(white)}\\ \hat{B} &= VB/\mu_{B(white)}\end{aligned} \tag{6.89}$$

If the white patch is not the brightest region within the image (for example, a light grey is used instead to prevent any of the components of the calibration patch from saturating), then the V in Equation 6.89 may be replaced by k similar to Equation 6.86.

This will ensure that the white patch will appear achromatic. However, the camera may also introduce a DC offset in the output (for example as an offset in the amplifier or A/D converter, or not completely estimating the dark current). The colour correction may be enhanced by introducing a black patch into the model. Let $\mu_{(black)}$ be the mean of the pixels within the black patch. A colour correction using both patches is:

$$\begin{aligned}\hat{R} &= V\frac{R-\mu_{R(black)}}{\mu_{R(white)}-\mu_{R(black)}}\\ \hat{G} &= V\frac{G-\mu_{G(black)}}{\mu_{G(white)}-\mu_{G(black)}}\\ \hat{B} &= V\frac{B-\mu_{B(black)}}{\mu_{B(white)}-\mu_{B(black)}}\end{aligned} \tag{6.90}$$

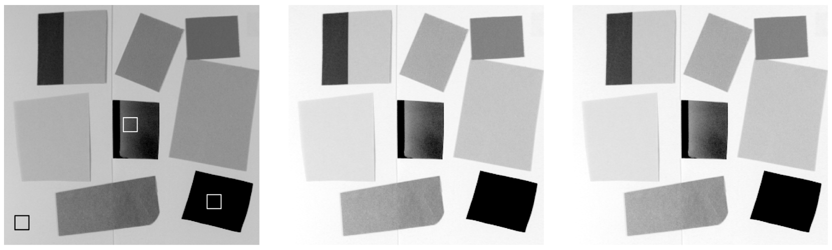


Figure 6.44 Correcting using black, white and grey patches. Left: original image with the patches marked; centre: stretching each channel to correct for black and white, using Equation 6.90; right: adjusting the gamma of the red and blue channels using Equation 6.91 to make the grey patch grey. (*See colour version of this figure in colour plate section*)

The results of this correction are shown in the centre panel of Figure 6.44.

If the calibration performed off-line, then the colour correction is making a brightness and contrast adjustment to each of the channels, as described in Section 6.1.1. If the calibration is performed on-line, then the best approach is to place the black and white patches at the top of the image, so the mean values may be used to correct the rest of the image. Otherwise, the whole image needs to be stored in a frame buffer to enable the whole image to be corrected. If the illumination is changing slowly, then it may be sufficient to use the channel gains and offsets calculated from the previous frame.

A more sophisticated model adds a third mid-grey patch, with mean value $\mu_{(grey)}$. This allows slight variations in gamma to be corrected between the channels. The green pixel value is left unchanged, but a gamma correction is applied to the red and blue channels to make the components equal for the grey patch. Considering just the red component (a similar equation results for blue):

$$\hat{R} = V\left(\frac{R-\mu_{R(black)}}{\mu_{R(white)}-\mu_{R(black)}}\right)^{\gamma_R} \tag{6.91}$$

$$\text{where} \qquad \gamma_R = \frac{\ln\left(\dfrac{\mu_{G(grey)}-\mu_{G(black)}}{\mu_{G(white)}-\mu_{G(black)}}\right)}{\ln\left(\dfrac{\mu_{R(grey)}-\mu_{R(black)}}{\mu_{R(white)}-\mu_{R(black)}}\right)} \tag{6.92}$$

The results of adjusting the gamma are shown in the right panel of Figure 6.44.

Correcting for grey requires the addition of two gamma correction blocks, one for each of red and blue as shown in Figure 6.45. The circuit for these is complicated by the fact that the gamma values are not constant, but are a result of calibration. If implemented on-line, the calculation of the gamma is also required. However, since this is only performed once per frame, it is probably best implemented in software rather than directly in hardware. In that case, the software can also calculate the values for a lookup table implementation of the gamma correction blocks.

There is a wide range of more complex illumination estimation methods, which are not considered here. Several of these are discussed elsewhere (Funt *et al.*, 1998; Finlayson *et al.*, 2001).

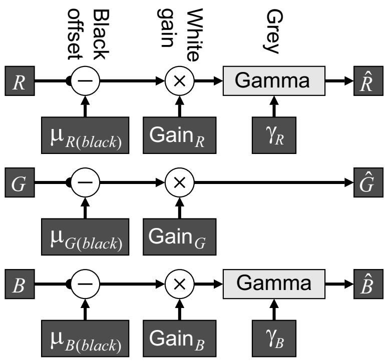


Figure 6.45 Implementing colour correction; the various parameters are stored in registers. Overflow detection and clipping has been omitted for clarity.

6.3.5 Colour Enhancement

When performing colour enhancement, is often easier to work in HSV or HLS space. Since the hue represents the basic colour, generally it is left unchanged (unless correcting colour errors as described above). Colours can be made move vivid by increasing the saturation. This can be achieved by multiplying by a constant, or using a gamma enhancement (with $\gamma < 1$). Overall the contrast of the image may be enhanced by enhancing the lightness or value component.

6.4 Summary

Point operations are some of the simplest to implement on an FPGA because each output pixel value depends only on the corresponding input pixel value. This enables them to be implemented using any processing mode. Since there are no dependencies between pixels, point operations can readily be parallelised by building multiple processing units. It is also common to pipeline point operations, processing data as it is being read from or written to memory.

Point operations on a single image are primarily concerned with adjusting the brightness or contrast (or equivalently adjusting the colour balance or contrast of a colour image). Thresholding is another common point operation, used for segmenting an image on the basis of intensity or colour. Any point operation may be directly implemented using a lookup table, giving constant processing time regardless of the complexity of the operation.

Point operations may also be applied between multiple images. Virtually any operation may be applied between the pixels of the images being combined. Most commonly, these are images captured at different times, so requires frame buffering to store the history. A common architecture is recursive processing, where a newly acquired image is combined with one or more images held in a frame buffer, with the result written back to the frame buffer.

The simplicity of point operations is also one of the main limitations. Frequently, global data is required to set appropriate parameters (for example for contrast stretching or selecting an appropriate threshold level). The use of histograms to capture and derive some of these parameters is described in the next chapter. A point operation considers each pixel in isolation and does not take into account context. Local filters, where the output pixel value is dependent on several pixels within a local neighbourhood, are described in Chapter 8.

7

Histogram Operations

As indicated in the previous chapter, the parameters for many point operations can be derived from the histogram of pixel values of the image. This chapter is divided into to two parts: the first considers greyscale histograms and some of their applications, and the second extends this to multidimensional histograms.

7.1 Greyscale Histogram

The histogram of a greyscale image, as shown in Figure 7.1, gives a count of the number of pixels in an image as a function of pixel value.

$$H[i] = \sum_{x,y} \begin{cases} 1, & I[x,y] = i \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

There are two main steps associated with using histograms for image processing. The first step is to build the histogram, and the second is to extract data from the histogram and use it for processing the image. Building the histogram is discussed in this section, whereas the applications are described in subsequent sections.

To build the histogram, it is necessary to accumulate the counts for each pixel value, as indicated in Equation 7.1. Since each pixel must be visited once, building the histogram is ideally suited to stream processing. To accumulate the pixel count for each pixel value requires maintaining the count so far for each value. This may be accomplished by an array of counters, with the input pixel used to enable the clock of the corresponding counter (Figure 7.2). The output counts need to be indexed using a bus or multiplexer.

A disadvantage of this approach is that the decoding and output multiplexing are relatively expensive. The decoder of the input pixel stream may be re-used as part of the output multiplexer; however, the number of different pixel values (or histogram bins) means that a large amount of logic cannot be avoided. The counters and associated registers must also be built using the logic resources of the FPGA, with each counter register built from flip-flops. This approach is, therefore, best suited if only a few counters are required (for example after thresholding), or if processing requires the individual counters to be accessed in parallel.

Each pixel can have only one pixel value, so only one counter is incremented in any clock cycle. This implies that the accumulators may be implemented in memory. Incrementing an accumulator requires reading the associated memory location, adding one, and writing the sum back to memory. This requires a

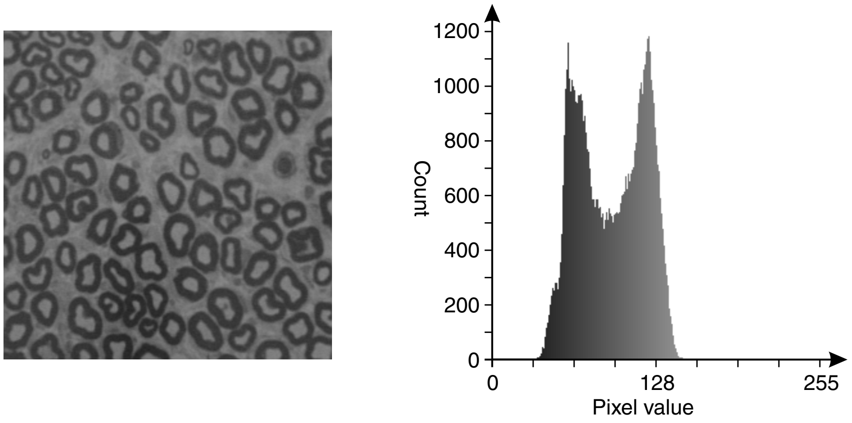


Figure 7.1 An image and its histogram.

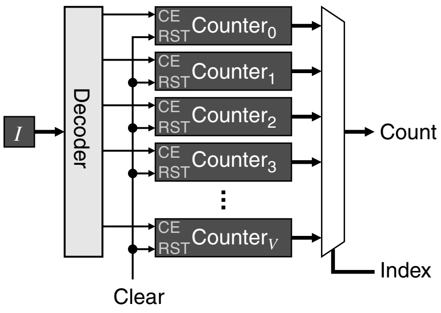


Figure 7.2 Histogram accumulator built from counters.

dual-port memory, with one port being used to read the memory and one port to write the result. This scheme is shown in Figure 7.3. Memory is effectively a multiplexed register bank. However, from an implementation perspective, the multiplexers are built within the memory addressing, so significantly fewer logic resources are required.

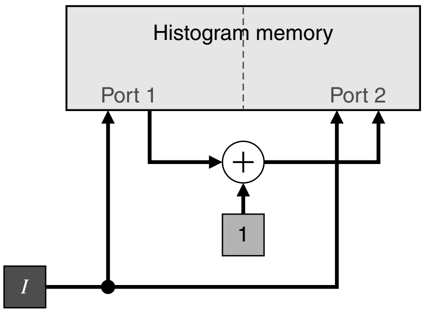


Figure 7.3 Histogram accumulation using dual-port memory.

One disadvantage of the memory-based approach is that resetting the registers at the start of a frame requires sequentially cycling through memory locations to set them to zero. With the register-based approach of Figure 7.2, all of the registers may be reset in parallel.

The circuit of Figure 7.3 requires a late write to the second port. This requires the timing to allow the data to be read near the start of the clock cycle, so the incremented value may be written at the end of the clock cycle. This may be readily achieved with asynchronous memory or by delaying the write pulse for synchronous memory.

If a late write is not available, then both the pixel value and the accumulated count must be buffered in registers to enable the updated count to be written in the following clock cycle. It is slightly more complicated than this, as shown in Figure 7.4, because if the subsequent pixel has the same value, the count read from memory will not have been updated yet. This requires checking if the current pixel is the same as the previous value, and if so incrementing the registered value rather than that read from memory.

As each pixel is processed independently, it is also possible to partition the image over several parallel accumulators (Figure 7.5). This approach then requires a final pass through the accumulators to combine the results from each partition to give the histogram for the complete image.

The circuitry that processes the histogram to extract data from it must operate on the same histogram. Similarly, the circuitry that resets the histogram must also be connected to the histogram memory. These require multiplexing the address and data lines to the appropriate data structures. These multiplexers are assumed in the following sections.

7.1.1 Data Gathering

Histograms may be used to gather data from the objects within the image. Consider an image of a single object, and that object can be separated from the background by thresholding. Then the area of the object is simply given by the number of pixels detected by thresholding. This may be obtained from the histogram of the corresponding binary image. Obviously, it is unnecessary to build a complete histogram for this case, a single counter will suffice (compare with Figure 7.2). However, if the threshold level is obtained by processing the histogram, then the area may be found directly from the histogram without necessarily thresholding the input image:

$$\text{Area}(I_{min} \leq I \leq I_{max}) = \sum_{i=I_{min}}^{I_{max}} H[i] \tag{7.2}$$

This idea may be generalised to measuring the area of many objects within an image. If the image is processed in such a way that the pixels associated with each object within the image are assigned a unique label, then each histogram bin will be associated with a different label, effectively giving the area of each object.

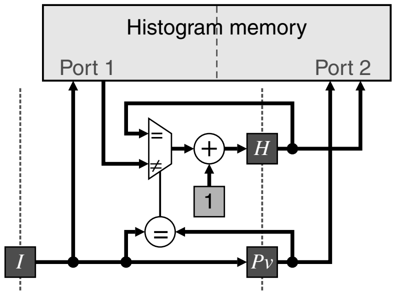


Figure 7.4 Performing histogram accumulation with caching when late write is not available.

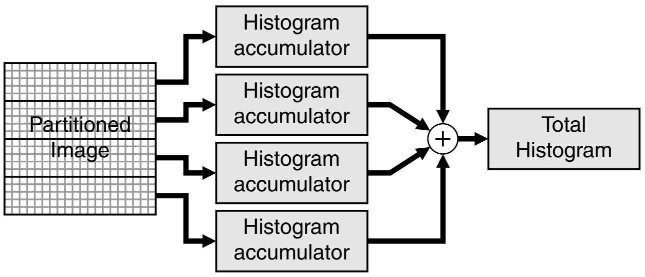


Figure 7.5 Partitioning the accumulation over several parallel processors.

Other useful statistics of the image, such as the mean and variance, may be extracted from the histogram, rather than directly from the image. While these may be calculated directly from the image, if the statistics are taken from a restricted range of pixel values that is chosen dynamically, it is more efficient to obtain the histogram as an intermediate step.

The mean pixel value of an image is given by:

$$\mu_I = \frac{\sum\limits_{x,y} I[x,y]}{\sum\limits_{x,y} 1} = \frac{\sum\limits_{i} iH[i]}{\sum\limits_{i} H[i]} \tag{7.3}$$

with the variance as:

$$\sigma_I^2 = \frac{\sum\limits_{x,y} (I[x,y]-\mu_I)^2}{\sum\limits_{x,y} 1} = \frac{\sum\limits_{i} (i-\mu_I)^2 H[i]}{\sum\limits_{i} H[i]} = \frac{\sum\limits_{i} i^2 H[i]}{\sum\limits_{i} H[i]} - \mu_I^2 \tag{7.4}$$

The last form of Equation 7.4 allows the variance to be calculated in parallel with the mean, rather than having to calculate the mean first. Often, the mean and variance within a restricted range of pixel values is required rather than over the full range. In this case, the summations of Equations 7.3 and 7.4 are performed over that range (i_{min} to i_{max}).

A direct implementation of these summations is shown in Figure 7.6. The start signal selects i_{min} into i and resets the summations to zero. On subsequent clock cycles, $H[i]$ is read and added to the appropriate accumulators. If the delay through the multiplications is too long, then this may be pipelined. When i reaches i_{max}, the final summation is performed and the divisions to calculate the mean and variance carried out. Again, if the divisions take too long, they may be pipelined. The *Done* output goes high to indicate the completion of the calculation.

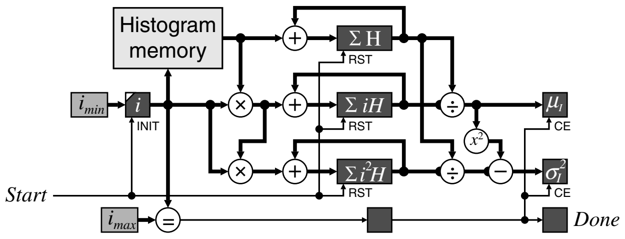


Figure 7.6 Calculating the mean and variance from a range of values within the histogram.

While the divisions on the output cannot easily be avoided, it is possible to share the hardware for a single division between the two calculations. The division can also be pipelined if necessary. If the standard deviation is required (rather than the variance) then a square root operation must be applied to the variance output. This may be accomplished with less logic than the division (Li and Chu, 1996).

The multiplications associated with the summations may be eliminated by using incremental update. Firstly, consider the summation, where K is an arbitrary constant:

$$\sum_i (K-i)H[i] = K\sum_i H[i] - \sum_i iH[i] \tag{7.5}$$

Therefore:

$$\mu_I = K - \frac{\sum (K-i)H[i]}{\sum H[i]} \tag{7.6}$$

Similarly:

$$\begin{aligned} \frac{\sum_i (K-i)^2 H[i]}{\sum_i H[i]} &= \frac{\sum_i i^2 H[i]}{\sum_i H[i]} - 2K\frac{\sum_i iH[i]}{\sum_i H[i]} + K^2\frac{\sum_i H[i]}{\sum_i H[i]} \\ &= \sigma_I^2 + \mu_I^2 - 2K\mu_I + K^2 \\ &= \sigma_I^2 + (\mu_I - K)^2 \end{aligned} \tag{7.7}$$

Therefore:

$$\sigma_I^2 = \frac{\sum_i (K-i)^2 H[i]}{\sum_i H[i]} - \left(\frac{\sum_i (K-i)H[i]}{\sum_i H[i]}\right)^2 \tag{7.8}$$

Next, consider the incremental calculation of Equations 7.5 and 7.7:

$$\begin{aligned} \sum_i^K (K-i)H[i] &= \sum_i^{K-1} (K-i)H[i] \\ &= \sum_i^{K-1} ((K-1)-i)H[i] + \sum_i^{K-1} H[i] \end{aligned} \tag{7.9}$$

From

$$\sum_i^{K-1} ((K-i)-1)^2 H[i] = \sum_i^{K-1} (K-i)^2 H[i] - 2\sum_i^{K-1} (K-i)H[i] + \sum_i^{K-1} H[i] \tag{7.10}$$

$$\begin{aligned} \sum_i^K (K-i)^2 H[i] &= \sum_i^{K-1} (K-i)^2 H[i] \\ &= \sum_i^{K-1} ((K-i)-1)^2 H[i] + 2\sum_i^{K-1} (K-i)H[i] - \sum_i^{K-1} H[i] \\ &= \sum_i^{K-1} ((K-1)-i)^2 H[i] + 2\sum_i^{K-1} ((K-1)-i)H[i] + \sum_i^{K-1} H[i] \end{aligned} \tag{7.11}$$

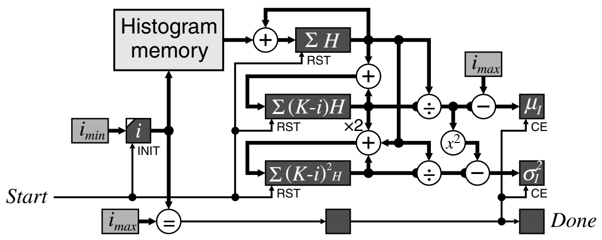


Figure 7.7 Using incremental calculations for the mean and variance.

Therefore, starting K with i_{min} and incrementing until i_{max} allows the summations to be performed without any multiplications (the factor of two in Equation 7.11 is simply implemented with a left shift). The resulting circuit is shown in Figure 7.7.

The circuits of Figures 7.6 and 7.7 are fine if only a single calculation is required. However, if the mean and variance are required for several different ranges, then the repeated summation can become a time bottleneck. If timing becomes a problem, then this may be overcome by building sum tables that contain cumulative moments of the histogram:

$$\begin{aligned} S_0H[j] &= \sum_{i=0}^{j} H[i] \\ S_1H[j] &= \sum_{i=0}^{j} iH[i] \\ S_2H[j] &= \sum_{i=0}^{j} i^2H[i] \end{aligned} \tag{7.12}$$

The first, S_0H, is also called the cumulative histogram, an example of which is shown in Figure 7.8. These tables require a single pass through the histogram to construct and, after that, the mean and variance of any range may be calculated with two accesses to each table:

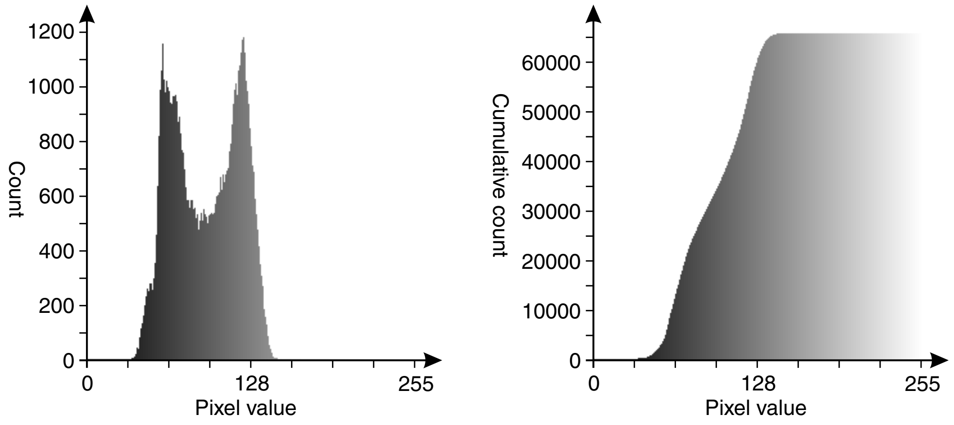


Figure 7.8 Left: a histogram; right: its cumulative histogram.

$$\begin{aligned}\mu_I(i_{min} \le I \le i_{max}) &= \frac{S_1H[i_{max}]-S_1H[i_{min}-1]}{S_0H[i_{max}]-S_0H[i_{min}-1]}\\ \sigma_I^2(i_{min} \le I \le i_{max}) &= \frac{S_2H[i_{max}]-S_2H[i_{min}-1]}{S_0H[i_{max}]-S_0H[i_{min}-1]} - \left(\frac{S_1H[i_{max}]-S_1H[i_{min}-1]}{S_0H[i_{max}]-S_0H[i_{min}-1]}\right)^2\end{aligned} \tag{7.13}$$

Higher order moments (skew and kurtosis) may also be calculated in a similar manner by extension of the above circuit.

The statistical range of pixel values within an image is probably best calculated directly from the image as it is streamed. However, if the histogram is already available for other processing then it is a simple manner to extract the range from the histogram. The minimum pixel value, I_{min}, is the smallest index that has a non-zero count, and the maximum pixel value, I_{max}, is the largest. A common operation using the range is contrast expansion using Equation 6.2 with:

$$\begin{aligned}b' &= I_{min}\\ a &= \frac{V}{I_{max}-I_{min}}\end{aligned} \tag{7.14}$$

Calculating the median of an image is probably easiest using the cumulative histogram of the image. The image median is defined as the bin that contains the fiftieth percentile. Ideally, an inverse cumulative histogram could be used; this maps from the cumulative count back to the corresponding pixel value. However, this can be derived from the cumulative histogram:

$$med_I = \min_i \left\{ S_0H[i] \ge \frac{N_P}{2} \right\} \tag{7.15}$$

where N_P is the number of pixels in the image and:

$$S_0H[V] = N_P \tag{7.16}$$

Rather than scan through the histogram to find the median, Equation 7.15 may be solved with a binary search, with each successive iteration giving one bit of the median.

The image median may be generalised to determine the median value within a range of pixel values:

$$med_I(i_{min} \le I \le i_{max}) = \min_i \left\{ S_0H[i] \ge \frac{S_0H[i_{max}] + S_0H[i_{min}-1]}{2} \right\} \tag{7.17}$$

The architecture of Figure 7.2 may be adapted to build the cumulative histogram directly (Fahmy *et al.*, 2005). Figure 7.9 demonstrates how this works. The decoder enables the clock of all of the counters greater than or equal to the pixel value, causing them to be incremented. After the whole image has been accumulated, the counters represent the cumulative histogram. As the outputs of all these counters are available, they can be compared with the median count (indicated by '50%' in the figure) in parallel. The priority encoder finds the minimum output that has a one, as required by Equation 7.15, and returns the corresponding index.

Another statistic that requires the histogram is the mode. The mode is defined as the index of the highest peak within the histogram. This requires a single scan through the histogram, for example as in Figure 7.10.

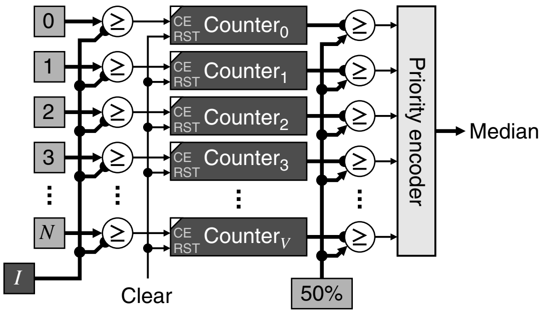


Figure 7.9 Parallel architecture for measuring the median.

There are two limitations to this approach for finding the mode. If the histogram is multimodal, only the global mode is found. The global mode is the one with the highest peak, which is not necessarily the most significant peak. Finding multiple peaks requires detecting the significant valleys and finding the mode between these valleys. Valley detection is described in Section 7.1.4 on threshold selection.

A second limitation is that histograms tend to be noisy. This is clearly seen in Figure 7.1. Consequently, the mode may not necessarily be representative of the true location of the peak. The accuracy of the peak location may be improved through filtering the histogram before finding the mode. A one-dimensional linear smoothing filter may be pipelined between reading the histogram values and finding the maximum. Suitable filters are described in detail in the next chapter. Note that care must be taken when designing the filter response to avoid bias being introduced if the peak is skewed.

7.1.2 Histogram Equalisation

Histogram equalisation is a contrast enhancement or contrast normalisation technique. It uses the histogram to construct a monotonic mapping that results in a flattened output histogram. From an information theoretic perspective, histogram equalisation attempts to maximise the entropy of an image by making each pixel value in the output equally probable.

From a contrast enhancement perspective, the assumption behind histogram equalisation is that the peaks of the input histogram contain the information of interest and the contrast between the peaks is of lesser importance. To make the output histogram flat, it is necessary to reduce the average height of the peaks by spreading the pixel values out. This increases the contrast of these regions. The valleys between the peaks are compressed to raise the average value, effectively reducing the contrast of these features.

These factors are clearly seen in Figure 7.11, where the regions with counts greater than the average are enhanced at the expense of those below the average. In this image, the effect has been to enhance the contrast within the background material; this is probably not the intended consequence for this image,

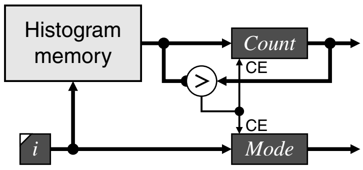


Figure 7.10 Finding the mode of a histogram.

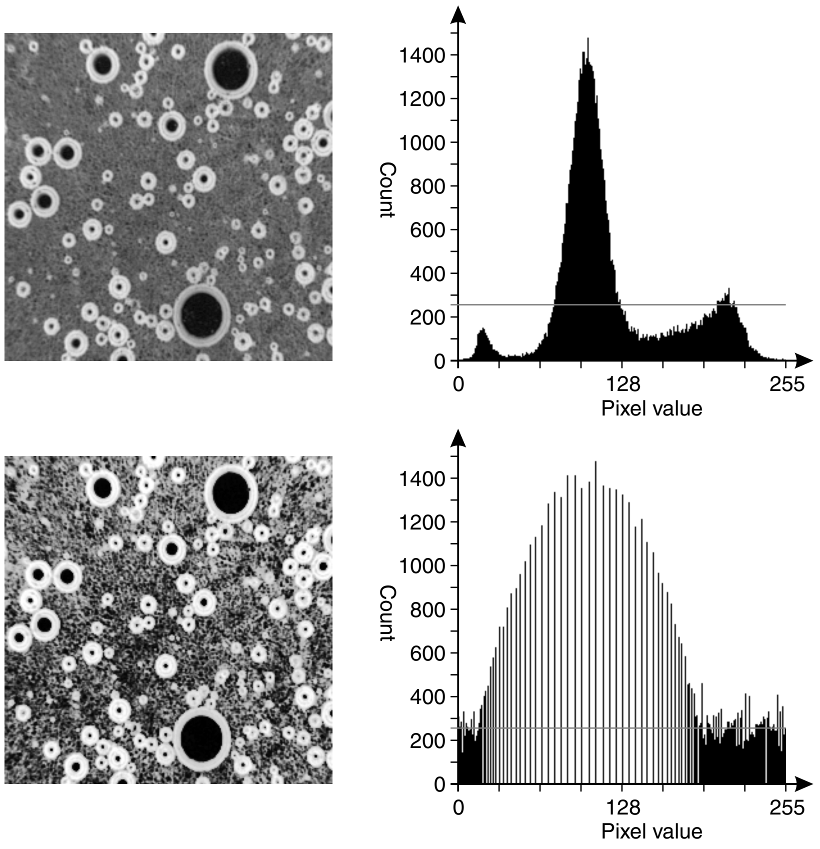


Figure 7.11 Histogram equalisation. Top: input image with its histogram; bottom: the image and histogram after histogram equalisation. The grey line on the histograms shows the average 'flat' level.

where it would have been more useful to enhance the contrast between the regions. However, in many applications, histogram equalisation gives a useful contrast enhancement.

Note that the main peak of the output histogram is exactly the same height as that of the input. This is a consequence of using a point operation to perform the mapping, which requires all pixels with one value in the input to map to the same value in the output. The average is obtained by spacing consecutive input values to more separated values in the output. The opposite occurs for bins with counts below the average. Several consecutive input pixel values may map to the same output pixel.

A flatter histogram may be obtained by using local information to distribute the pixels with an input value with a large count over a range of output pixel values (Hummel, 1975; 1977) although such operations are technically filters rather than point operations.

The mapping for performing histogram equalisation is the normalised cumulative histogram. Intuitively, if the input bin count is greater than the average, the slope of the mapping will be greater than one, and conversely if less than the average. Normalisation requires scaling the histogram by the number of pixels, so that the output maps to the maximum pixel value (V).

The associated division operation is the most expensive part of histogram equalisation, although it can readily be pipelined. An alternative to division is to calculate the inverse off-line (since the number of

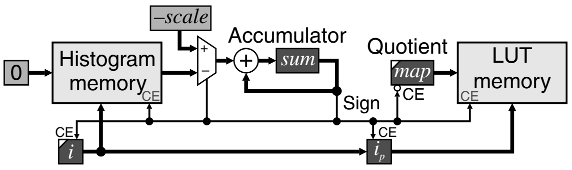


Figure 7.12 Using repeated subtraction to avoid division for histogram equalisation.

pixels in an image is constant) and use a multiplication. Another alternative is to arrange the region of the image that the histogram is taken of to result in a scale factor that is a power of two. The region used is best taken from the centre of the image, with the assumption that the values not accumulated will follow the same statistics.

The division may be performed using repeated subtraction. This takes advantage of the fact that the cumulative histogram is monotonic to extend the calculations performed at lower pixel values to higher values. Therefore, the repeated subtractions calculate the mapping at the same time as the total is accumulating.

The architecture for this is shown in Figure 7.12. The counters *map* and *i* are initialised to zero, and the *sum* is initialised to $-scale$ (or $-scale/2$ for rounding) where:

$$scale = \frac{N_P}{V} \tag{7.18}$$

If the accumulated sum, which represents the current remainder, is negative then the count from the next histogram bin is accumulated. However, if the accumulated sum is positive, the scale factor is subtracted from the sum and the quotient incremented. When the index i is incremented, the previous value is also saved as i_p. As a histogram bin is accumulated, the current quotient, *map*, is saved at the index i_p. For 8-bit images, *map* will be incremented at most 255 times, and 257 clock cycles are required to read the input histogram and write the mapping into the lookup table.

There are several control issues associated with histogram equalisation, or indeed any other histogram processing. Firstly, the circuitry for building the histogram (for example from Figure 7.3 or Figure 7.4) must be combined with that for building the mapping used to perform the histogram equalisation (Figure 7.12). This will require appropriate multiplexers in the address and data lines. It is also necessary to reset the histogram accumulators to zero before the next frame. This may be accomplished in parallel with building the map, by using the histogram memory write port, as shown in Figure 7.12.

Secondly, there is a similar resource sharing issue with the mapping associated with performing the histogram equalisation. The lookup table must be shared between the circuit for building the mapping (Figure 7.12) and applying the mapping to the image. Each of these processes only requires a single port, so use of a dual-port memory would simplify the sharing. Otherwise, the map address lines would require an appropriate multiplexer, with additional control required to provide the memory read or write signal.

Thirdly, the registers for building the map (in Figure 7.12) may be initialised to appropriate values while the histogram is being built.

Finally, since construction of the histogram equalisation map requires data from the complete image, it is necessary to buffer the image before the mapping is applied, as shown in the left panel of Figure 7.13. In many instances, the image changes relatively slowly. An approximation in these circumstances is to apply the mapping acquired from the previous frame while building the histogram for the current frame (McCollum *et al.*, 1988), as illustrated in the right panel of Figure 7.13. The consequence is that the histogram equalisation might not be quite correct, but there is a significant savings in both resources (the frame buffer is not required) and latency.

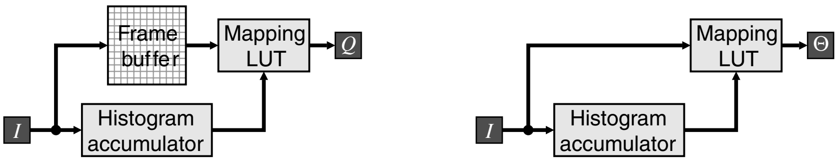


Figure 7.13 Using the histogram for histogram equalisation. Left: buffering the frame during histogram accumulation; right: directly applying table from the last image to the current image.

The procedure for histogram equalisation may readily be adapted to perform other histogram transformations, for example hyperbolisation (Frei, 1977). It is even possible to derive a target histogram from one image and apply a mapping to another image such that the histogram of the output image approximates that of the target. The basic procedure for this is illustrated graphically in Figure 7.14.

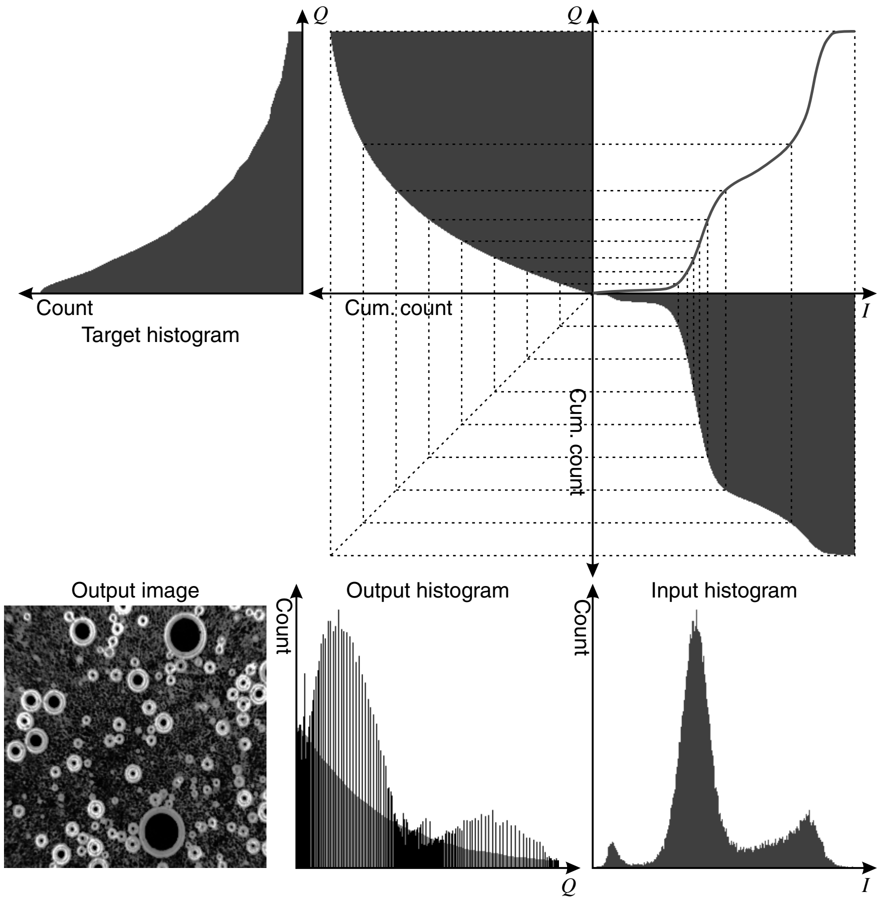


Figure 7.14 Determining the mapping to produce an arbitrary output histogram shape by matching cumulative histograms. Top: target histogram and cumulative histogram; right: input histogram and cumulative histogram; top right: building the mapping; bottom: output image and output histogram (with target histogram in the background).

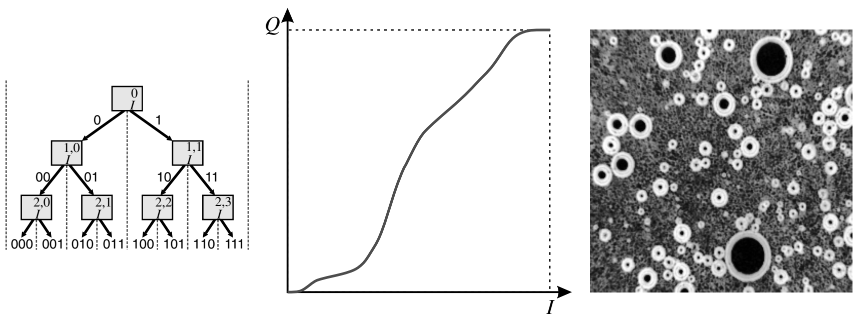


Figure 7.15 Determining the output code or pixel value using the successive mean quantisation transform. Centre: the resultant mapping; right: applying the mapping to the input image. (Adapted with permission from M. Nilsson *et al.*, "The successive mean quantization transform," *IEEE International Conference on Acoustics, Speech and Signal Processing*, **4**, 429–432, 2005. © 2005 IEEE.)

To derive a monotonic mapping, it is necessary for all of the pixels less than or equal to a pixel value in the input image to map to values that are less than or equal to the corresponding output value. This may be achieved by mapping the cumulative histogram of the input to the cumulative histogram of the target. As with histogram equalisation, the output histogram approximates the target on average, as shown in the bottom of Figure 7.14.

The circuit of Figure 7.12 may be adapted to calculate this mapping. Rather than subtract off the constant *scale*, the value to be subtracted may be obtained from the target histogram, $H_T[map]$ (assuming that both images have the same area).

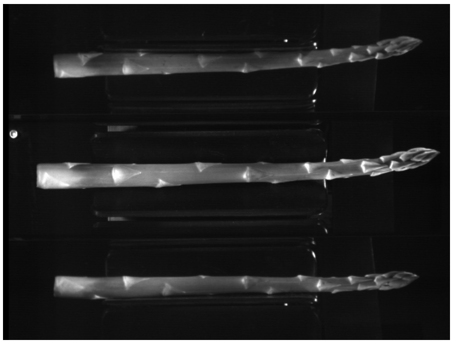
Related to histogram equalisation is the successive mean quantisation transform (Nilsson *et al.*, 2005b). This works by using a series of means to derive successive threshold levels that define successive bits of the output pixel value. Initially, the whole range of pixel values is defined as a single partition. The mean value of the pixels within a partition is used to split the partition into two at the next level, with the corresponding output bit defined by the partition to which a pixel is assigned. This partitioning process is shown in Figure 7.15. Sum tables may be used to calculate successive means efficiently using Equation 7.13.

It has been demonstrated that, in many applications, the successive mean quantisation transform provides a subjectively better enhancement than histogram equalisation, because it preserves the gross structure of the histogram and does not over-enhance the image (Nilsson *et al.*, 2005a; 2005c). In the context of the successive mean quantisation transform, histogram equalisation corresponds to successively partitioning using the median rather than the mean.

7.1.3 Automatic Exposure

Related to contrast enhancement is automatic exposure. This involves using data obtained from the captured image to modify the image capture process to optimise the resultant images. In sensors with electronic shuttering, this may require adjusting the control signals that determine the exposure time. It may also involve automatically adjusting the aperture or even controlling the intensity of the light source.

The histogram of the captured image is able to provide useful data for gauging the quality of the exposure. If the maximum pixel value within the image is significantly less than *V*, then the image is not making the best use of the available dynamic range. This may be improved by increasing the exposure. On the other hand, if the image contains a large number of pixels with pixel value *V*, then there may be important detail lost through saturation. This may be corrected by reducing the exposure.



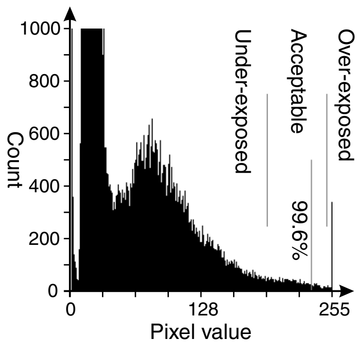


Figure 7.16 Using the histogram for exposure. Left: example image; right: histogram with its 99.6th percentile shown, along with the acceptable bounds.

Although the maximum pixel value may be determined without needing a histogram, there are two major limitations to using the maximum pixel value to assess exposure. Firstly, the maximum value is likely to be a result of noise and may not reflect the true peak of intensities within the image. Secondly, the maximum pixel value may actually reach *V* without being saturated. In many images, the maximum value may result from specular reflections or other highlights that are not significant from an image processing point of view.

An alternative approach is to use the histogram to find the pixel value of the 99th percentile (Bailey *et al.*, 2004) (or even higher depending on the image content). If the histogram is flat, the 99th percentile should have a value of about 252 for an 8-bit image. However, most histograms are non-uniform, and in many applications there may be variability from one frame to the next that must be tolerated.

One approach is to have a target acceptable range for the given level. Consider the image in Figure 7.16. The histogram clearly shows that some pixels are saturated (from the peak at 255). These are the specular reflections and highlights in the figure on the left, and no information of importance is lost if these are saturated. However, the 99.6th percentile (allowing for the equivalent of two rows worth of pixels to be saturated) is within the allowed band of 192–250, indicating that the exposure for this image is acceptable. In this application, the large dead-band allowed for the considerable variability in the produce from one image to the next, while ensuring that an adequate exposure was maintained.

When determining if the exposure is acceptable, it is faster to accumulate the histogram counts starting from the maximum pixel value and counting down, rather than counting up from zero.

Note that changing the camera exposure will not affect the current image. It may also be best to adjust the exposure incrementally over several images rather than in a single step, especially if the object being imaged changes from one image to the next.

7.1.4 Threshold Selection

The purpose of thresholding is to segment the image into two (or more) classes based on pixel value. While it is possible to consider connectivity and other issues (Sezgin and Sankur, 2004), there is a wide range of threshold selection techniques that calculate a threshold level from the histogram rather than the image (Sezgin and Sankur, 2004). Consider an obvious case, where the pixel values of each class are well separated; the resulting pixel value histogram is multimodal. This implies that the distributions of each of the classes, and hence the best places to threshold the image, may be determined by analysing the histogram. Since selecting an appropriate threshold is a common task in many image analysis applications, many of the key methods for dynamically determining the threshold from a histogram will be reviewed.

The most common case is separating objects from a background, where there are two classes: object and background. If the proportion of object pixels is known then the corresponding threshold level may be determined from the inverse cumulative histogram (Doyle, 1962). This may be found from the cumulative histogram in two ways.

The first is to scan through cumulative histogram until the desired proportion is reached. In fact, it is not necessary to actually build the cumulative histogram; the histogram counts may be accumulated until the desired total is reached.

$$\begin{aligned} T &= \max_i \left\{ S_0 H[i] < N_{obj} \right\} \\ &= \max_k \left\{ \sum_{i=0}^{k} H[i] < N_{obj} \right\} \end{aligned} \tag{7.19}$$

The second approach, if the cumulative histogram is available, is to solve Equation 7.19 using a binary search, as described earlier for finding the median.

In many cases, however, the number of object pixels in not known in advance, but is determined as a result of thresholding. An alternative that is more tolerant of variation in the number of object pixels is to set the threshold a given number of standard deviations from the mean:

$$T = \mu_I + \alpha \sigma_I \tag{7.20}$$

where α is chosen empirically based on knowledge of the image being thresholded.

When separating objects from the background, the histogram is often bimodal, with the peaks representing the distributions of object and background pixel values. If the peaks are clearly separated and do not overlap, choosing an appropriate threshold between the peaks is relatively trivial. In most cases, however, the peaks are often broad, with significant overlap between the distributions, as in the image in Figure 7.17. When the histogram has distinct peaks, an appropriate threshold is somewhere in the valley between the peaks. Finding the 'best' threshold is complicated by the fact that the histogram is seldom smooth. Both the peaks and the valley between them are noisy, so simply choosing the deepest valley can result in considerable variation from one image to the next.

One approach to this problem is to smooth the histogram to make the valley between the peaks clearer. This was first proposed by Prewitt and Mendelsohn (1966). The histogram is repeatedly filtered with a

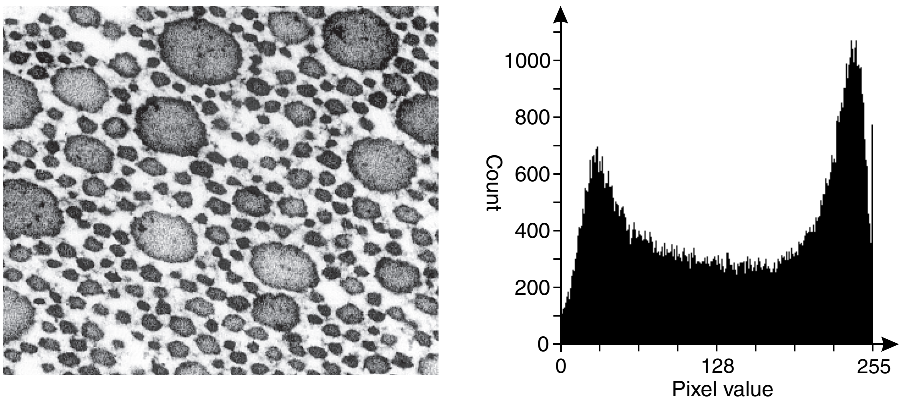


Figure 7.17 An image of objects against a background and its corresponding histogram.

narrow filter until there are only two maxima; the threshold is then in the valley between the peaks. That is, it will be the only pixel value in the histogram that satisfies:

$$H[T-1] > H[T] \leq H[T+1] \tag{7.21}$$

A variation on this theme is to record the location of the peaks and valleys and track these through a range of filter widths. This results in a range of thresholds, depending on the scale of filtering (Carlotto, 1987). Such a multiscale approach suits multilevel thresholding, or the threshold at the lowest scale (most smoothed) may be tracked up the scales to refine it. A disadvantage of such approaches is the repeated filtering required to obtain the necessary smoothness, or range of scales in the latter case.

An alternative approach is to fit a curve to the histogram. A common assumption is that the shape of each of the modes is a Gaussian distribution, modelling the histogram as a sum of Gaussians:

$$H[i] = \sum_j A_j e^{-\frac{(i-\mu_j)^2}{2\sigma_j^2}} \tag{7.22}$$

In the case where the histogram is bimodal, the threshold that gives the minimum misclassification error is (Kittler and Illingworth, 1986):

$$T = \min_t error(t) = \min_t (p_1 \ln \sigma_1 + p_2 \ln \sigma_2 - p_1 \ln p_1 - p_2 \ln p_2) \tag{7.23}$$

where

$$p_1 = \frac{\sum_{i=0}^{t} H(i)}{N_P} = \frac{S_0 H(t)}{N_P}, \quad p_2 = \frac{\sum_{i=t+1}^{V} H(i)}{N_P} = 1 - p_1 \tag{7.24}$$

and

$$\sigma_1 = \sigma_I(0 \leq I \leq t), \quad \sigma_2 = \sigma_I(t < I \leq V) \tag{7.25}$$

This is based on separating the histogram into two parts by the threshold and modelling each part with a Gaussian distribution. The minimum error will correspond to the crossover point between the two distributions. Jiulun and Winxin (1997) showed that the derivation given by Kittler is equivalent to minimising the relative entropy between the actual distributions and the ideal Gaussian distributions and determining the threshold that gives the minimum classification error.

Equation 7.23 may be evaluated with a single pass through the cumulative histogram, although several clock cycles may be needed for each t to evaluate the logarithms. Note the square roots for calculating the standard deviation may be avoided by using:

$$\ln \sigma_1 = \tfrac{1}{2} \ln \sigma_1^2 \tag{7.26}$$

An alternative to fitting to the histogram is to make the distributions on each side of the threshold as compact as possible. This leads to the popular approach by Otsu of minimising the intra-class variance (Otsu, 1979):

$$T = \min_t \left(p_1 \sigma_1^2 + p_2 \sigma_2^2\right) \tag{7.27}$$

which is equivalent to maximising the separation between the means, or the interclass variance:

$$T = \max_t \left(p_1 p_2 (\mu_2 - \mu_1)^2\right) \tag{7.28}$$

where

$$\mu_1 = \mu_I(0 \leq I \leq t) \quad \text{and} \quad \mu_2 = \mu_I(t < I \leq V) \tag{7.29}$$

The advantage of Equation 7.28 over Equation 7.27 is that is only requires calculating the means rather than the variances. Since the criterion function that is maximised in Equation 7.28 can have multiple local maxima (Lee and Park, 1990), iterative methods for finding the maximum may converge to the wrong value. Therefore, it is necessary to search by scanning through the histogram. Equation 7.28 may be calculated incrementally if the global mean is known. Firstly, a pair of registers is initialised to zero:

$$\begin{aligned} p_1[-1] &= 0 \\ S_1[-1] &= 0 \end{aligned} \tag{7.30}$$

Then the following iterations are used:

$$\begin{aligned} p_1[t] &= p_1[t-1] + H[t] & p_2[t] &= N_P - p_1[t] \\ S_1[t] &= S_1[t-1] + tH[t] & S_2[t] &= \mu_I N_P - S_1[t] \\ \mu_1[t] &= S_1[t]/p_1[t] & \mu_2[t] &= S_2[t]/p_2[t] \end{aligned} \tag{7.31}$$

Note that in Equation 7.31 p_1 is not a probability, rather it is the cumulative histogram. The division by N_P is unnecessary, since it is constant and will not affect the location of the maximum.

Alternatively, the criterion function may be evaluated directly from the cumulative histogram and summed first moment:

$$p_1 p_2 (\mu_2 - \mu_1)^2 = \frac{S_0H[t]}{N_P} \frac{(N_P - S_0H[t])}{N_P} \left(\frac{S_1H[t]}{S_0H[t]} - \frac{S_1H[V] - S_1H[t]}{N_P - S_0H[t]}\right)^2 \tag{7.32}$$

Again, the factor of N_P^2 in the denominator of the first two terms may be left out and Equation 7.32 rearranged to reduce the number of division operations:

$$p_1 p_2 (\mu_2 - \mu_1)^2 \propto \frac{(S_1H[t]N_P - S_0H[t]S_1H[V])^2}{S_0H[t](N_P - S_0H[t])} \tag{7.33}$$

Otsu's method works best with a bimodal distribution, with a clear distinct valley between the modes (Lee and Park, 1990). Performance can deteriorate when there is a significant imbalance between the number of object and background pixels, or when the object and background are not well separated, or when there is significant noise (Lee and Park, 1990).

In principle, Otsu's method may be extended to multilevel thresholding (Otsu, 1979), although each threshold added will increase the dimensionality of the search space, making the search for the maximum computationally impractical for more than two or three threshold levels. Peaks within the multidimensional criterion function will correspond to zero partial derivatives, and iterative search techniques may be employed to speed up the search for the zero partial derivatives (Lin, 2005).

An alternative approach that gives similar results to Otsu's method for many images is to iteratively refine the threshold to midway between the means of the object and background classes (Ridler and Calvard, 1978; Trussell, 1979):

$$\begin{aligned} T_k &= \frac{\mu_I(0 \leq i \leq T_{k-1}) + \mu_I(T_{k-1} < i \leq V)}{2} \\ &= \frac{1}{2}\left(\frac{S_1H[T_{k-1}]}{S_0H[T_{k-1}]} + \frac{S_1H[V] - S_1H[T_{k-1}]}{N_P - S_0H[T_{k-1}]}\right) \end{aligned} \tag{7.34}$$

with the initial threshold given from the global mean:

$$T_0 = \mu_I \tag{7.35}$$

The iteration of Equation 7.34 is repeated until it converges ($T_k = T_{k-1}$), although this generally occurs within four iterations (Trussell, 1979). Note that the resultant threshold level is dependent on the initial value. The initialisation of Equation 7.35 works well if there are approximately equal numbers of object and background pixels, or the object and background are well separated. An alternative initialisation if the distribution is very unbalanced is the midrange of the distribution:

$$T_0 = \frac{\min(I) + \max(I)}{2} \tag{7.36}$$

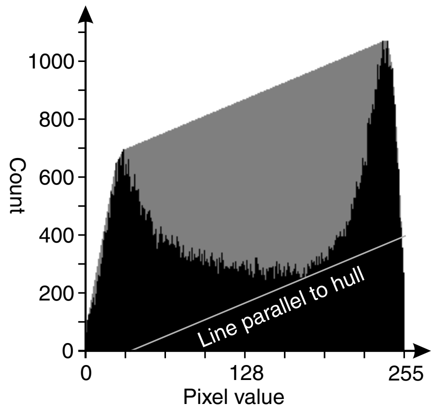
The attractiveness of this iterative method is its simplicity and speed, and with good contrast images it gives good results.

Another approach to thresholding is to use the total entropy of two regions of the histogram as the criterion function (Kapura *et al.*, 1985). The threshold is then chosen to maximise the information between the object and background distributions:

$$\begin{aligned} T &= \max_t \left(-\sum_{i=0}^{t} \frac{H[i]}{S_0H[i]} \ln \frac{H[i]}{S_0H[i]} - \sum_{i=t+1}^{V} \frac{H[i]}{N_P - S_0H[i]} \ln \frac{H[i]}{N_P - S_0H[i]} \right) \\ &= \max_t \left(\ln(S_0H[i](N_P - S_0H[i])) - \frac{\sum_{i=0}^{t} H[i]\ln H[i]}{S_0H[i]} - \frac{\sum_{i=t+1}^{V} H[i]\ln H[i]}{N_P - S_0H[i]} \right) \end{aligned} \tag{7.37}$$

As with the other methods, calculating the optimum threshold using this criterion function may be performed with a single pass through the histogram. However, as with the minimum error method, a hardware implementation is complicated by the need to calculate logarithms. Maximising the entropy has a tendency to push the threshold towards the middle of the distribution, so this method works best when there is a clear distinction between the two distributions. It may also be extended to multilevel thresholding (Kapura *et al.*, 1985) in a similar manner to Otsu's method, with the same limitation of the exponential growth of the search space with the number of threshold levels used.

When the object and background distributions significantly overlap, there may not be two distinct modes to the distribution. Rosenfeld and de la Torre (1983) proposed using the difference between the distribution and its convex hull to identify irregularities on the flanks of a unimodal



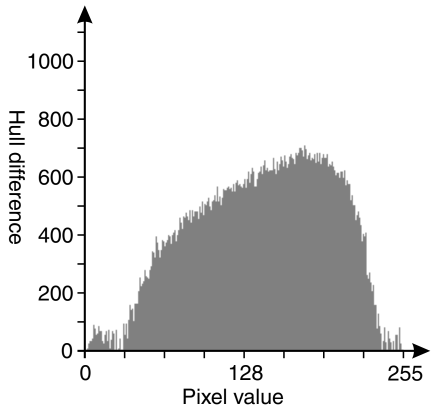


Figure 7.18 Convex hull of histogram. Left: the convex hull superimposed on the histogram, the line parallel to the top of the hull shows how the threshold is biased towards the higher peak; right: the difference between the hull and the histogram.

distribution. The method also works equally well with bimodal distributions, where it will tend to bias the threshold toward the higher peak, as shown in Figure 7.18. Since, in general, the convex hull is sloped, the largest difference will correspond to the pixel value in the valley that has a tangent parallel with the hull. It is this slope that biases the threshold location and enables a bump on the side of a unimodal histogram to be detected.

The convex hull of the histogram is the smallest extension of the shape that has no concavities. A physical analogy is to stretch a rubber band over the histogram and fill in the histogram below the rubber band, as shown in Figure 7.18. The convex hull may be represented by the set of convex vertices, which corresponds to the set of counts in the histogram that are unaffected by this infilling process.

Firstly, observe that the slope of the convex hull decreases monotonically with increasing pixel value. Therefore, consider a sequence of three points with increasing pixel value: $(i_1, H[i_1])$, $(i_2, H[i_2])$ and $(i_3, H[i_3])$. The middle point is concave if the slope between the first two points is less than the slope between the second two points. That is:

$$\frac{H[i_2]-H[i_1]}{i_2-i_1} < \frac{H[i_3]-H[i_2]}{i_3-i_2} \tag{7.38}$$

The division may be avoided by cross-multiplying:

$$(H[i_2]-H[i_1])(i_3-i_2) < (H[i_3]-H[i_2])(i_2-i_1) \tag{7.39}$$

If the middle point is concave, it may be eliminated.

Using this, the convex vertices may be found as follows. Firstly, the histogram is scanned to find the first non-zero count and the corresponding pixel value, i_{min}, is stored in a register. For each successive pixel value, the pixel difference, $\Delta i_{Current} = 1$ and histogram difference, $\Delta H_{Current} = H[i]-H[i-1]$, are determined. A table is used to maintain these offsets between the convex vertices. If the table is empty, the pair $(\Delta i, \Delta H)$ is recorded in the table. If the table is not empty, Equation 7.39 is evaluated between the last entry in the table and the new entry:

$$\Delta H_{Last}\Delta i_{Current} < \Delta H_{Current}\Delta i_{Last} \tag{7.40}$$

If Equation 7.40 is true, then the previous vertex is now concave so must be eliminated. The last entry in the table is therefore removed and combined with the current entry:

$$\begin{aligned}\Delta i_{Current} &\Leftarrow \Delta i_{Current} + \Delta i_{Last} \\ \Delta H_{Current} &\Leftarrow \Delta H_{Current} + \Delta H_{Last}\end{aligned} \tag{7.41}$$

Equation 7.40 is repeatedly re-evaluated with the new last entry in the table and entries removed with Equation 7.41 until all concave vertices to the left of the current point are removed. Then the current point is added to the table. Note that when a count of zero is reached ($H[i] = 0$), the above is not performed in case it is the end of the histogram. However, if a subsequent non-zero count is encountered, $\Delta i_{Current}$ is set to reflect the intervening zero counts.

Once the end of the histogram has been reached, the table will contain the offsets between the convex vertices. This process will take at most $2V$ steps, because for each new pixel value one entry is added to the table, and through the elimination steps each entry can only be removed at most once from the table.

The final stage is to link between the convex vertices to obtain the convex hull. This is complicated slightly by the fact that the pixel values are discrete and the slopes are not necessarily represented by integers. This process starts with the pixel value with the first non-zero count, $i_{\min}$, and initialising a remainder to zero:

$$\begin{gathered} H_{Hull}[i_{\min}] = H[i_{\min}] \\ R = 0 \end{gathered} \tag{7.42}$$

Then each table entry is used to calculate the next Δi hull values:

$$\begin{gathered} H_{Hull}[i] = H_{Hull}[i-1] + \left\lfloor \frac{\Delta H + R}{\Delta i} \right\rfloor \\ R = (\Delta H + R) \bmod \Delta i \end{gathered} \tag{7.43}$$

where $\lfloor\ \rfloor$ is the largest integer value less than its argument, and mod returns the remainder between zero and Δi. Rather than record the actual hull, it is only necessary to determine the pixel value corresponding to the maximum difference between the hull and the original histogram

$$T = \max_t \left(H_{Hull}[t] - H[t]\right) \tag{7.44}$$

that is, the position of the maximum value on the right in Figure 7.18.

A variation on valley finding is the global valley approach of Davies (2008). His criterion function is based on the geometric mean of differences between a histogram point and the peaks on either side:

$$T = \max_t \sqrt{\text{clip}(H_L[t] - H[t])\text{clip}(H_R[t] - H[t])} \tag{7.45}$$

where

$$\begin{aligned} H_L[t] &= \max_{i<t} H[i] \\ H_R[t] &= \max_{i>t} H[i] \end{aligned} \tag{7.46}$$

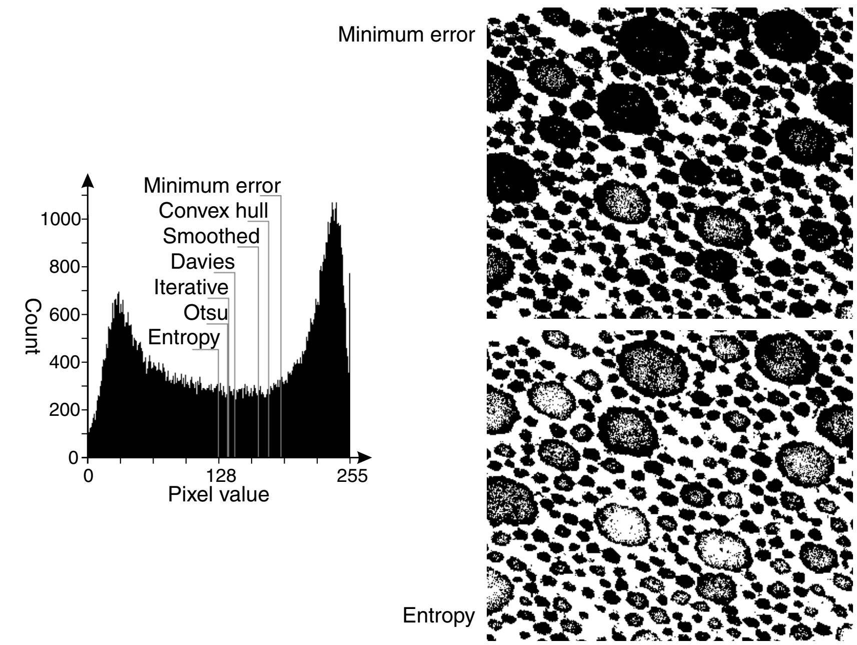


Figure 7.19 Applying the threshold calculation method to the histogram of Figure 7.17. The methods are minimum error (Equation 7.23), convex hull (Equation 7.44), smoothed minimum (Equation 7.21), Davies (Equation 7.45), iterative mean (Equation 7.34), Otsu (Equation 7.28) and entropy (Equation 7.37). Right: results of the two extreme threshold levels are compared; top: the minimum error method; bottom: the maximum entropy method.

and

$$\text{clip}(x) = \begin{cases} 0, & x < 0 \\ x, & x > 0 \end{cases} \tag{7.47}$$

The geometric mean in Equation 7.45 is better than the arithmetic mean because it prevents pedestals at the ends of the distribution (Davies, 2008). Only two passes are required through the histogram to evaluate the criterion function: a reverse pass to incrementally calculate H_R and a forward pass to incrementally calculate both H_L and the criterion function. If searching for a single threshold, the square root operation of Equation 7.45 may be omitted.

This approach may be extended to multilevel thresholding by smoothing the criterion function until the specified number of local maxima remains (Davies, 2008).

The threshold levels calculated for the different methods are compared in Figure 7.19 for the image of Figure 7.17. The different assumptions made, and different criteria functions, result in threshold levels ranging from 127 for the maximum entropy method, through to 188 for the minimum error method. Note that for this image there is no global threshold that provides good separation between the objects and the background. Of all the methods considered here, the iterative mean of Equation 7.34 is computationally the simplest, especially if the cumulative histogram and the cumulative first moment are calculated.

Histogram-based methods of calculating threshold levels generally rely on valley finding techniques. If the number of either object or background pixels is significantly smaller than the other, the valley between the peaks can be indistinct. Filters may be applied to the image to improve the histogram obtained. Obtaining the histogram-only pixels near edges will result in an approximately equal numbers of object and background pixels (Weszka *et al.*, 1974). Noise smoothing may result in narrower peaks, making them more distinct and easier to detect and separate. Applying an edge enhancement filter prior to accumulating the histogram will reduce the number of pixel values in the valley between the peaks, making the result less sensitive to the actual threshold level used (Bailey, 1995).

Many of the threshold selection techniques described here would be easier to implement in software, using an embedded processor, than directly in hardware. The timing is not particularly critical because the threshold could be calculated during the vertical blanking period between frames.

Only global thresholding has been considered in this section. Adaptive thresholding chooses a different threshold at each location based on local statistics. Adaptive thresholding can be considered equivalent to filtering followed by global thresholding. Adaptive thresholding is considered in more detail in the next chapter on filtering.

7.1.5 Histogram Similarity

Histogram similarity is one method of object classification. The intensity histograms of a set of candidate model objects are maintained in a database. The histogram of the test object is obtained and compared with the histograms of the model objects, with the object classified based on which model has the most similar histogram. Unlike most other methods of object classification, the object does not need to be completely separated from the background, as long as it dominates the content of the image.

The histogram may require some normalisation prior to matching (usually limited to contrast stretching) to account for varying illumination intensity. Normalisation such as histogram equalisation, or shaping, is obviously not appropriate. Often the resolution of the histogram is reduced (the bin size is increased) by combining adjacent bins. This reduces the sensitivity to small changes in illumination and reduces problems resulting from gaps in the histogram that result from normalisation. It also helps to speed the search for a match.

Let all the model histograms be normalised to correspond to the same area image and let $M_j[i]$ be the histogram of the jth model. The similarity between the test histogram, $H[i]$, and the model can be found from their intersection (Swain and Ballard, 1991):

$$Match_j = \sum_i \min\left(H[i], M_j[i]\right) \tag{7.48}$$

This measure will be a maximum when the histograms are identical and will decrease as the histograms become more different. Therefore, the input image is classified as the object that has the largest match.

All spatial relationships between the pixel values are lost in the process of accumulating the histogram. Consequently, many quite different images can have the same histogram. Classification based on histogram similarity will, therefore, only be effective when the histograms of all of the models are distinctly different. However, the method has the advantage that the histogram has a considerably smaller volume of data than that of the whole image, significantly reducing comparison times. This can be helped further by reducing the number of bins within the histograms. Where speed is important, the test histogram may be compared with a number of model histograms in parallel.

7.2 Multidimensional Histograms

Just as pixel-value histograms are useful for gathering data from greyscale images, multidimensional histograms can be used to accumulate data from colour or other vector valued images. The simplest approach is simply to accumulate a one-dimensional histogram of each channel. While this is useful for some purposes, for example gathering the data required for colour balancing, it cannot capture the relationships between channels that give each pixel its colour. This requires a true multidimensional histogram.

A multidimensional histogram has one axis or dimension for each channel in the input image. For example, the histogram of an RGB image would require three dimensions. The histogram gives a count of the pixels within the image that have a particular vector value. For an RGB image, Equation 7.1 would be extended to give:

$$H[r,g,b] = \sum_{x,y} \begin{cases} 1, & \mathbf{I}_{RGB}[x,y] = \{r,g,b\} \\ 0, & \text{otherwise} \end{cases} \tag{7.49}$$

Data is accumulated in exactly the same way as for a greyscale histogram. The larger address space of a multidimensional histogram requires a significantly larger memory, which may not necessarily fit within the FPGA. Two histogram memory accesses are required for each pixel: one to read the previous count and the other to write the updated count. This may be accomplished either with dual-port memory, in the same manner as one-dimensional histograms (Figure 7.4), or by running the accumulator memory at twice the pixel clock frequency.

While it is usually possible to implement two-dimensional histograms at the full resolution of the input vector, for three and higher dimensions the size of the histogram memory can become prohibitively large. This problem may be overcome by reducing the number of bins by applying a coarse quantisation to each component. Indexing into the multidimensional array is performed in the same manner as in software:

$$H[x,y,z] = H[x + n_x y + n_x n_y z] \tag{7.50}$$

where n_x and n_y are the number of bins used for the x and y components, respectively. Indexing may be simplified if the range of each component is a power of two, as the memory address may then be formed by concatenating the vector components. The quantisation to accomplish this may be readily implemented by truncating the least significant bits of each vector component.

One limitation with using multidimensional histograms is the time required to both initialise them and to extract the data. Throughput issues may be overcome by using pipelining, although bandwidth issues are harder to deal with. It may require multiple parallel banks to initialise and process the histograms in a realistic time. Bank switching allows one bank to be initialised while another is accumulating and a third is being processed. Once the histogram data has been collected, the histograms can be used and processed as images in their own right if necessary.

7.2.1 Triangular Arrays

Often when processing colour images, intensity independence is often desired. With an appropriate colour space conversion, the intensity may be removed, with a reduction to two dimensions for the chrominance. If chromaticity is used (either $x{-}y$ or $r{-}g$) the resulting histogram accumulator is triangular. This may result in inefficient use of memory, as almost half is not being used. Unless the memory may be shared with another part of the application, an alternative is to pack the memory.

With a triangular accumulator, $n_x = n_y = n$. The number of entries in the triangular array is then $\frac{1}{2}n(n+1)$ rather than n^2 for the full array. When n is a power of two, as suggested earlier for efficiency, say $n = 2^k$, the number of entries becomes:

$$N_E = \frac{1}{2}n(n+1) = 2^{2k-1} + 2^{k-1} \tag{7.51}$$

If using a single memory block, this does not result in any savings, because of the second term in Equation 7.51. Memory savings can only be achieved with multiple small memory blocks.

Instead, if the number of bins is reduced to $n = 2^k - 1$, the number of entries is now:

$$N_E = \frac{1}{2}n(n+1) = 2^{2k-1} - 2^{k-1} \tag{7.52}$$

Since the second term is now subtracted rather than added, the array will fit into a block of size 2^{2k-1}.

The quantisation of the input range into n bins is now more complex and requires scaling the input before truncating the least significant bits. Fortunately, this scaling may be implemented by a single subtraction:

$$x = \left\lfloor x_{in}\frac{n}{2^b} \right\rfloor = \left\lfloor x_{in}\frac{2^k-1}{2^b} \right\rfloor = \left\lfloor \frac{x_{in} - 2^{-k}x_{in}}{2^{b-k}} \right\rfloor \tag{7.53}$$

Two approaches may be used to pack the data into the smaller array. The first successively shifts each row back in memory to remove the unused entries:

$$\begin{aligned} H[x,y] &= H\left[x + ny - \frac{1}{2}y(y-1)\right] \\ &= H\left[x + 2^k y - \frac{1}{2}y(y+1)\right] \end{aligned} \tag{7.54}$$

The multiplication may be avoided by using a small lookup table on the y address, as shown on the left in Figure 7.20.

The second approach is to maintain the spacing but fold the upper addresses into the unused space at the end of each row:

$$H[x,y] = \begin{cases} H[x + (n+1)y], & y \le \dfrac{n}{2} \\ H[(n-x) + (n+1)(n-y)], & y > \dfrac{n}{2} \end{cases} \tag{7.55}$$

Note that $(n-x)$ and $(n-y)$ are just one's complements of x and y respectively, enabling them to be implemented with exclusive OR gates as shown in Figure 7.20.

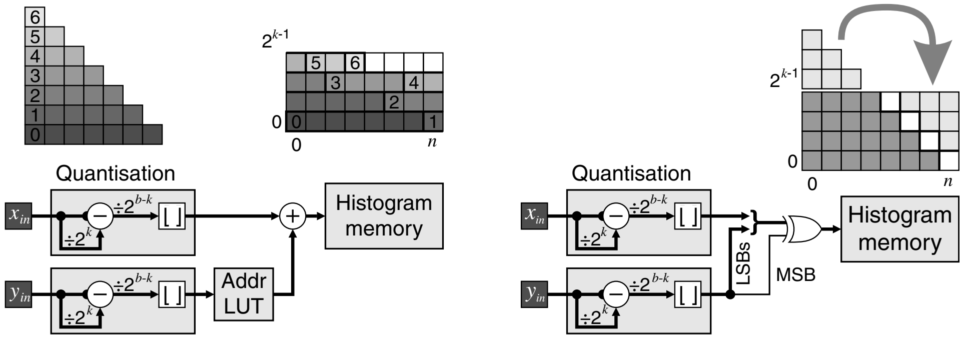


Figure 7.20 Packed addressing schemes for a triangular array with $n = 2^k - 1$. Left: packing the rows sequentially. The numbered memory locations represent the addresses stored in the Addr LUT. Right: folding the upper addresses into the unused space.

7.2.2 Multidimensional Statistics

The scalar mean and variance extend to vector valued quantities as the mean vector and covariance matrix. The mean vector is simply found by taking the mean of the components, for example the mean of a three-dimensional data set is:

$$\boldsymbol{\mu}_\mathbf{I} = \frac{\sum\limits_{x,y} \mathbf{I}[x,y]}{\sum\limits_{x,y} 1} = \begin{bmatrix} \mu_{I_1} \\ \mu_{I_2} \\ \mu_{I_3} \end{bmatrix} \tag{7.56}$$

Similar to the variance of Equation 7.4, the covariance between two variables is:

$$\sigma^2_{I_i,I_j} = \frac{\sum\limits_{x,y}(I_i - \mu_{I_i})(I_j - \mu_{I_j})}{\sum\limits_{x,y} 1} = \frac{\sum\limits_{x,y} I_i I_j}{\sum\limits_{x,y} 1} - \mu_{I_i}\mu_{I_j} \tag{7.57}$$

with the covariance matrix formed from the covariances between every pair of components:

$$\boldsymbol{\sigma}^2_\mathbf{I} = \begin{bmatrix} \sigma^2_{I_1} & \sigma^2_{I_1,I_2} & \sigma^2_{I_1,I_3} \\ \sigma^2_{I_1,I_2} & \sigma^2_{I_2} & \sigma^2_{I_2,I_3} \\ \sigma^2_{I_1,I_3} & \sigma^2_{I_2,I_3} & \sigma^2_{I_3} \end{bmatrix} \tag{7.58}$$

The diagonal elements of $\boldsymbol{\sigma}^2_\mathbf{I}$ correspond to the variances of the individual components. The off-diagonal elements measure how the corresponding components vary with respect to each other, which reflects the correlation between that pair of components. It is clear from Equation 7.57 that the covariance matrix is symmetric. In matrix form:

$$\boldsymbol{\sigma}^2_\mathbf{I} = \frac{\sum\limits_{x,y}(\mathbf{I} - \boldsymbol{\mu}_\mathbf{I})(\mathbf{I} - \boldsymbol{\mu}_\mathbf{I})^\mathrm{T}}{\sum\limits_{x,y} 1} \tag{7.59}$$

where the superscript $^\mathrm{T}$ denotes the vector transpose. (For complex data, the complex conjugate of the transpose is used.)

For a colour image, each sample has three components or dimensions. Calculating the mean therefore requires three accumulators, one for each dimension. If the denominator is not fixed (for example extracting statistics from a region of the image), an additional accumulator is required for the denominator. From the symmetry in the covariance matrix, only six accumulators are required for the numerator of Equation 7.59.

When calculating global statistics for the image, the data for the mean and covariance may be performed directly on the input as it is streamed in. Alternatively, a multidimensional histogram may be used as an intermediary, in a similar manner to that described in Section 7.1.1. While using a histogram as an intermediate step is efficient for one-dimensional histograms, in two and higher dimensions their utility for this purpose becomes marginal. Unless the resolution in each dimension is reduced, the histogram can have more entries than the original image. While this may be overcome by reducing the resolution, this will also reduce the accuracy of the statistics derived from the histograms. Therefore, gathering data from a multidimensional histogram is usually only practical if many separate calculations must be performed for different regions within the multidimensional data space (for example to obtain the mean and covariance for each of a set of colour classes).

The covariance matrix may be used in a number of ways. If the covariance matrix is of a set of points from the same distribution, such as from the same colour class, then the covariance matrix provides information on the spread of distribution within the multidimensional space. If the distribution is

Gaussian, then the shape will be ellipsoidal, with the orientation and extent of the ellipsoid defined by the covariance matrix. In one dimension, the probability that a point, x, belongs to the distribution may be determined from the distance of the point from the centre of the distribution, d, in terms of the number of standard deviations:

$$d = \frac{x-\mu}{\sigma} = \sqrt{(x-\mu)^2(\sigma^2)^{-1}} \tag{7.60}$$

With a multidimensional distribution, in general the spread of the distribution is different in different directions. The Mahalanobis distance (Mahalanobis, 1936) generalises Equation 7.60 to measure the distance of a vector, **x**, from a distribution, **X**, using the covariance:

$$d_{Mahalanobis} = \sqrt{(\mathbf{x}-\boldsymbol{\mu}_\mathbf{X})^\mathrm{T}(\boldsymbol{\sigma}_\mathbf{X}^2)^{-1}(\mathbf{x}-\boldsymbol{\mu}_\mathbf{X})} \tag{7.61}$$

If the distribution is spherical, with a standard deviation of one, then the covariance matrix will be the identity matrix, and this will reduce to the standard Euclidean distance:

$$d_{Euclidean} = \sqrt{(\mathbf{x}-\boldsymbol{\mu}_\mathbf{X})^\mathrm{T}(\mathbf{x}-\boldsymbol{\mu}_\mathbf{X})} \tag{7.62}$$

Another common use for the covariance matrix is for dimensionality reduction. Principal components analysis (PCA) defines a new set of axes where each axis is independent of the others (the covariances are all zero). This effectively rotates the multidimensional space in such a way to best align the axes with the axes of the distribution. In the rotated space, the axes are ordered in terms of decreasing variance. This means that the first principle component is in the direction that accounts for the most variation within the data, and corresponds to the long axis of the ellipsoid if the data is from a multidimensional Gaussian distribution. The dimensionality of the data set may be reduced by discarding the lower order components. It can be shown that for a given number of dimensions or axes retained, this will minimise the error of the approximation in a least squares sense.

The principal component axes are found by determining the eigenvectors of the covariance matrix. It can be shown that the eigenvectors will be orthogonal because the covariance matrix is symmetric. Therefore, the matrix **E** formed from the unit eigenvectors of the covariance matrix will represent a pure rotation of the multidimensional space about the centre (or mean vector). The corresponding eigenvalues represent the variance in the direction of each eigenvector. This is related by:

$$\boldsymbol{\sigma}_{PCA}^2 = \mathbf{E}^\mathrm{T}\boldsymbol{\sigma}_\mathbf{I}^2\mathbf{E} \tag{7.63}$$

where the $\boldsymbol{\sigma}_\mathbf{PCA}^2$ is the diagonal covariance matrix of the transformed data. The transformation is performed by projecting each pixel onto the new axes:

$$\mathbf{I}_{PCA} = \mathbf{E}^\mathrm{T}\mathbf{I} \tag{7.64}$$

The complexity of the control logic for determining the eigenvectors from the covariance matrix implies that this step is probably best performed in software rather than hardware. Of course, the ALU for the processor may be enhanced with dedicated instructions to accelerate the matrix operations. Once the transformation matrix, **E**, has been determined, the projection of Equation 7.64 may be readily implemented in hardware.

The principal components are only independent axes with respect to the underlying data only if the whole data set is Gaussian distributed. This may be the case if it is performed on a cluster of data points within the data set (for example a particular colour class) but is generally not the case for whole images, which consist of multiple objects or classes. In this case, PCA will merely decorrelate the axes. These axes

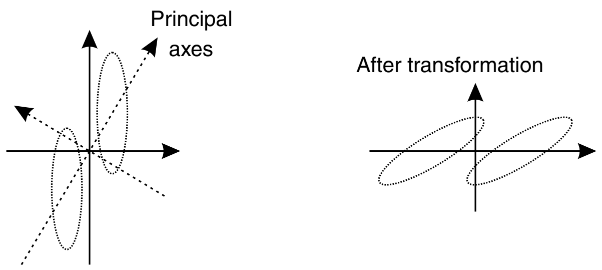


Figure 7.21 Principal components counter-example. Left: the data on the original axes, with the principal component axes overlaid. The ellipses represent different data classes. Right: after PCA transformation; the original axes were better for class separation than the PCA axes.

may not necessarily have any useful meaning; in particular, they may not necessarily be useful features for performing classification. The two-dimensional (contrived) example of Figure 7.21 is such a counter-example that illustrates a case where the original axes were better for separating the two classes than the principal axes.

From an image processing perspective, principal components analysis has two main uses. When applied across the components of an image, it determines the linear combinations of components that account for most of the variation within the image. This may be used, for example, to reduce the number of channels of a multispectral image. It can also be used for image compression to concentrate or compact the energy in the signal. Fewer bits are required for the components that have lower variance. In this regard, the principal components of an RGB image are often similar to the YUV or YIQ components (unless the image is dominated by a strongly coloured region).

The second application of PCA is in object recognition. In this case, rather than consider the components as the axes, the pixel locations are the axes; each image may then be considered a point in this very high dimensional space. When considering a set of images of related objects, each image will result in a separate point, with the resulting distribution representing the variability from one image to another. The dimensionality of this high dimensional space may be significantly reduced by principal components analysis. Each axis in the new space corresponds to an eigen-image – the high dimensional eigenvector of the corresponding covariance matrix. The variation between the set of images may be reduced to a small number of components, representing the projections onto, hence the weights of, each eigen-image. These weights may then be used for classifying the objects. One classic example of this approach is face recognition (Turk and Pentland, 1991), where the eigen-images are often referred to as *eigen-faces*.

Other statistical measures, such as the median and range, are based on an ordering of the data. Since vectors have no natural ordering (Barnett, 1976), the standard definitions no longer apply. The simple approach of taking the median or range of each component gives an approximation but is not the true result. For the median, the resulting point may not be one of the input data points because in general the median of one component will select a different pixel to the median of another component (Astola *et al.*, 1990). When used for median filtering within an image, this can result in coloured fringes around edges.

The problem with operating independently on the components is that it does not take into account the relationship between components. Each vector value should be treated as a single unit and operated on as such. One approach is to convert the vector to a scalar, which can then be used to define an ordering. Such conversions enable a reduced ordering (Barnett, 1976), with the results determined by the particular conversion used. A simple conversion is to use the vector magnitude. For colour images, the luminance, or in fact any weighted combination of the components, could be used as the scalar value for ordering (Comer and Delp, 1999). A limitation of these approaches is that many quite different vectors would be considered equal, with no ordering between them. One way of overcoming this for integer components is to interleave

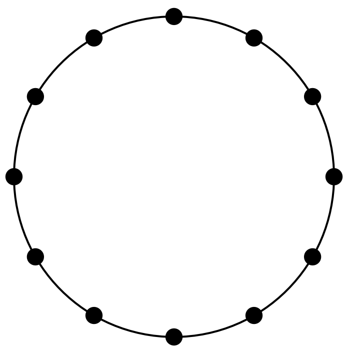


Figure 7.22 Vector median is not unique.

the bits of each of the components (Chanussot *et al.*, 1999). This keeps each distinct vector unique, although the ordering is still dependent on interleaving order.

It is possible to define both a true median (Astola *et al.*, 1988; 1990) and range (Barnett, 1976) for vector data. The median of a set of vectors, **X**, is the vector within the set, **x**, that minimises the distances to all of the other points in the set. That is:

$$\mathbf{x}_{med} = \arg\min_{\mathbf{x}_j \in \mathbf{X}} \sum_i \left\| \mathbf{x}_j - \mathbf{x}_i \right\| \tag{7.65}$$

The distances may be measured either using Euclidean (L_2 norm) or city block (L_1 norm) distances (Astola *et al.*, 1990). The city block distance is easier to calculate in hardware, although the median vector will then depend on the particular coordinates used. Calculating the median requires measuring the distance between every pair of points within the set. This makes the computational complexity of order $O(N^2)$ where N is the number of points in the set. For an image, this is impractical. However, for smaller N, such as filtering, this is more feasible.

One limitation of the median defined in this way is that it is not unique. Consider a set of points distributed symmetrically (for example in two dimensions, evenly around the circumference of a circle as shown in Figure 7.22). Each one of the points is the median as defined by Equation 7.65. This non-uniqueness results from requiring the median to be one of the input samples. Davies (2000) argues that this constraint is unnecessary and results in selecting a point that is not representative of the data as a whole. If this constraint is relaxed, the resulting minimum, also called the *geometric median*, is then:

$$\mathbf{x}_{med} = \arg, \min_{\mathbf{x}_j \in \mathbb{R}^n} \sum_i \left\| \mathbf{x}_j - \mathbf{x}_i \right\|_2 \tag{7.66}$$

where n is the number of components in each vector. It can be shown that the geometric median is unique (Ostresh, 1978) and may be found iteratively by Weiszfeld's algorithm (Ostresh, 1978; Chandrasekaran and Tamir, 1989):

$$\mathbf{x}_{r+1} = \frac{\sum_i \frac{\mathbf{x}_i}{\|\mathbf{x}_i - \mathbf{x}_r\|_2}}{\sum_i \frac{1}{\|\mathbf{x}_i - \mathbf{x}_r\|_2}} \tag{7.67}$$

Each iteration of Equation 7.67 requires summing over the complete set of vectors. Since successive iterations must be performed sequentially, the only scope for parallelism is to partition the set of input vectors over several processors. Calculating the inverse Euclidean distance is also relatively expensive, but is amenable to pipelining using a CORDIC processor. The resulting weight may be shared by the numerator and denominator. The iteration of Equation 7.67 converges approximately linearly, although it can get stuck if one of the intermediate points is a one of the input vectors (Chandrasekaran and Tamir, 1989).

For multidimensional data, the range can be defined as the distance between the two points that are furthermost apart (Barnett, 1976):

$$range = \max_{\mathbf{x}_i,\mathbf{x}_j \in \mathbf{X}} ||\mathbf{x}_i - \mathbf{x}_j|| \tag{7.68}$$

Again the distance between every pair of points must be calculated, giving complexity of order $O(N^2)$. A simpler approximation is to combine the ranges of each component. This assumes that the distribution is rectangular and aligned with the axes, so will tend to over-estimate the true range.

7.2.3 Colour Segmentation

Multidimensional histograms provide a convenient visualisation for segmentation, particularly when looking at colour images. Segmentation is equivalent to finding clusters within the data set, or equivalently finding peaks within the multidimensional histogram.

When considering colour segmentation or colour thresholding, using three-dimensional histograms is both hard to visualise and expensive in terms of resources. In this case, pairs of components are usually considered; this projects the three-dimensional histogram onto a two-dimensional plane. As described before, RGB is generally not the best colour space for segmentation. If colour is of primary interest, then the colour can be converted to YCbCr or HSV and the intensity channel ignored. The corresponding two-dimensional histograms are then taken of either $Cb-Cr$ or $H-S$ respectively. Alternatively, the components can be normalised and the $x-y$ or $r-g$ histogram taken.

Several of these are compared in Figure 7.23. Note the predominantly diagonal structure in the $R-G$ histogram in the left panel. This is even more exacerbated if the lighting varies across the scene. The grid structure is caused from the stretching of each component during colour correction.

Normalising the coordinates by dividing the sum (as in Equation 6.78) gives the $r-g$ histogram. This is a triangular histogram with nothing above the line $r + g = 1$. The intensity dependence has been removed, resulting in tighter groups. The dense group in the centre corresponds to the white background. The isolated 'random' points result from the black region where a small change in one of the components due to noise drastically changes the proportions of the components. Black regions must be distinguished based on luminance rather than colour. The faint lines connecting each cluster with the background result from mixing around the edges of each coloured patch; the edge pixels consist partly of the coloured object and

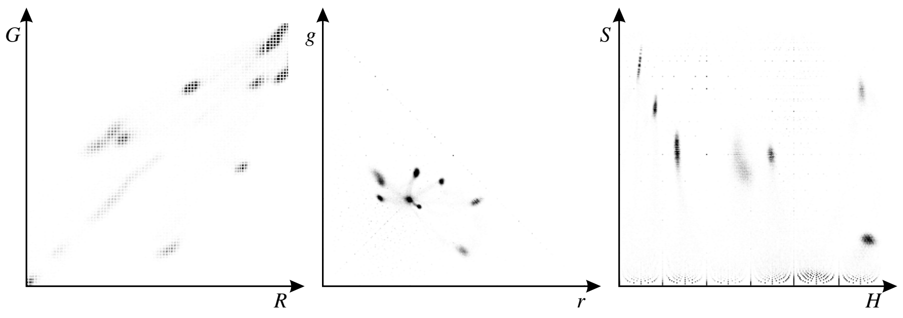


Figure 7.23 Example two-dimensional histograms of the colour corrected image from Figure 6.44. Left: $R-G$ histogram; centre: $r-g$ histogram; right: $H-S$ histogram. (For key features of each histogram, refer to the text.)

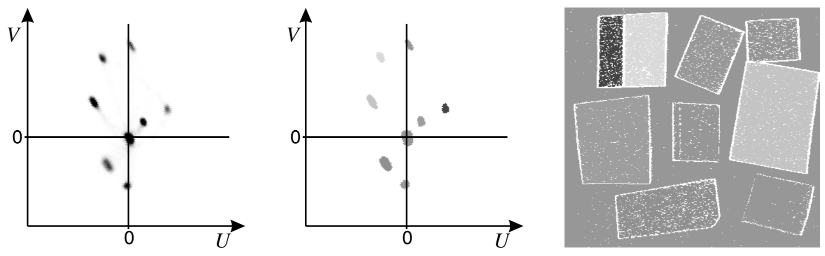


Figure 7.24 Using a two-dimensional histogram for colour segmentation. Left: $U-V$ histogram using Equation 6.61; centre: after thresholding and labelling, used as a two-dimensional lookup table; right: segmented image. (*See colour version of this figure in colour plate section*)

partly of the background. Although present in all of the histograms, these edges are more visible in the $r-g$ or $Cb-Cr$ histograms.

The third example is the $H-S$ histogram on the right. This, too, removes the intensity dependence. The best separation is achieved after proper colour balancing, which has quite a strong effect on both the hue and saturation. The pattern of points at low saturation results from noise affecting both the hue and saturation of grey and white pixels. Similarly, the scattered points at higher saturation result from noise affecting the black pixels.

The colour histogram can then be used as the basis for segmentation. This process is illustrated in Figure 7.24 with the $U-V$ histogram (using Equation 6.61) of the colour corrected image of Figure 6.44. Since each of the coloured patches results in a distinct cluster, simply thresholding the two-dimensional histogram is effective in this case. Each connected component in the histogram is assigned a unique label (Section 11.4), represented here by a colour. This labelled two-dimensional histogram can then be used as a lookup table on the original U and V components to give the labelled, segmented image.

In the example here, the white pixels represent unclassified colour values. The corresponding points within the histogram were below the threshold. The number of unclassified pixels within a patch may be significantly reduced by expanding the labelled regions within the two-dimensional lookup table (using a morphological filter) and filtering the output image to remove isolated unclassified pixels. Note the large number of unclassified pixels around the borders of each patch. These result from edge pixels being a mixture of the patch and background colours. They therefore fall part way between the corresponding clusters in the histogram. The number of such pixels may be significantly reduced

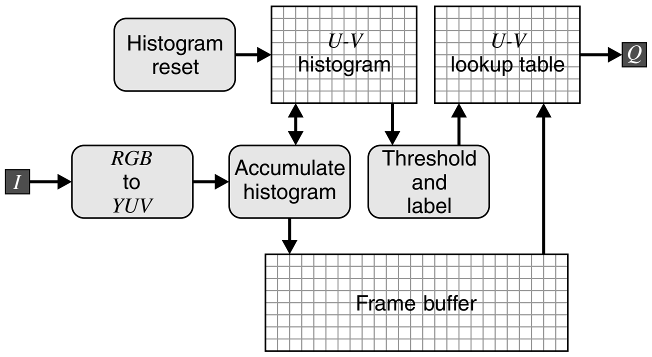


Figure 7.25 Block diagram of a simple colour segmentation scheme.

by using an appropriate edge enhancement filter prior to classification. The grey and black patches cannot be distinguished from the white background in this classification because the Y component is not used for segmentation. The projection onto the $U-V$ plane collapses these regions into a single cluster.

A block diagram for implementing such colour segmentation is shown in Figure 7.25. The scheduling logic is not shown here but effectively controls the sequence as follows:

1. Firstly, the histogram must be initialised by resetting the counts to zero.
2. As the input image is streamed in, it is converted to YUV (or any other convenient colour space). The U and V components are used to increment the corresponding bin in the histogram. The components are also saved in a frame buffer for later classification.
3. Once the histogram is accumulated, the clusters are determined. Here simple thresholding and blob labelling are used, but more sophisticated cluster extraction techniques could be used. The histogram is converted into a lookup table.
4. Finally, the U and V components from the frame buffer are used to index the lookup table to produce the corresponding label for the pixel, producing a streamed output image.

If the labelling is determined offline, then the U and V components may be directly looked up to produce the label. However, the advantage of using the histogram is that the segmentation can be adaptive. The lookup table is also not really required to perform the classification – any of the fixed threshold methods described in Section 6.3.3 could be used, with the data from the histogram used to adapt the threshold levels.

Once common application of multidimensional histograms is for detecting skin-coloured regions within images (Vezhnevets *et al.*, 2003; Kakumanua *et al.*, 2007). While a number of approaches can be used, histograms may be used to represent the colour distributions of skin and non-skin regions. The initial distributions are established through training, with examples of skin and general background. The histograms, once normalised, give the prior probability of observing a particular colour, **I**, given the class, C_i, that is:

$$p(\mathbf{I}|C_i) = \frac{hist_i[\mathbf{I}]}{\sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]} \tag{7.69}$$

where $hist_i[\mathbf{I}]$ is the histogram for class C_i. Bayes' theorem can be used to invert this to give the probability of a given class given the observed colour:

$$p(C_i|\mathbf{I}) = \frac{p(\mathbf{I}|C_i)p(C_i)}{p(\mathbf{I})} \tag{7.70}$$

where

$$p(C_i) = \frac{\sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]}{\sum\limits_{i}\sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]} \tag{7.71}$$

given that the training samples are provided in the expected proportion of observation and $p(\mathbf{I})$ is the probability of observing a particular colour.

The Bayesian classifier then selects the most probable class given the observation, **I**:

$$\begin{aligned} C_i &= \arg\max_i \{p(C_i|\mathbf{I})\} \\ &= \arg\max_i \left\{ \frac{p(\mathbf{I}|C_i)p(C_i)}{p(\mathbf{I})} \right\} \end{aligned} \tag{7.72}$$

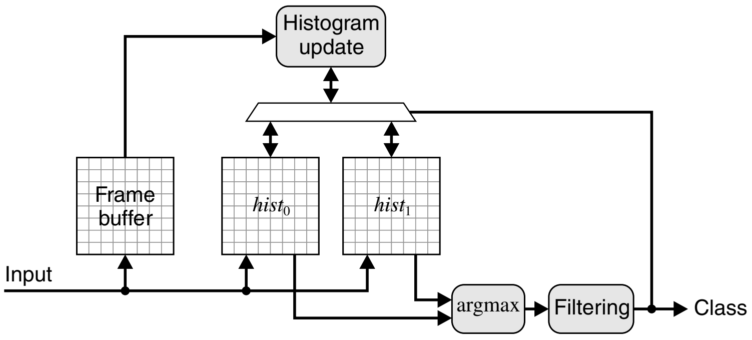


Figure 7.26 Adaptive Bayesian colour segmentation.

Since the denominator does not depend on i, it will not affect which one is the maximum. Simplifying, and then substituting Equations 7.69 and 7.71 gives:

$$\begin{aligned} C_i &= \arg\max_i \{p(\mathbf{I}|C_i)p(C_i)\} \\ &= \arg\max_i \left\{ \frac{hist_i[\mathbf{I}]}{\sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]} \frac{\sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]}{\sum\limits_i \sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]} \right\} = \mathop{argmax}_i \left\{ \frac{hist_i[\mathbf{I}]}{\sum\limits_i \sum\limits_{\mathbf{I}} hist_i[\mathbf{I}]} \right\} \end{aligned} \tag{7.73}$$

Again because the denominator does not depend on i, this may be simplified further:

$$C_i = \arg\max_i \{hist_i[\mathbf{I}]\} \tag{7.74}$$

In other words, the particular class that has the maximum histogram entry for the observation is the most likely class. Alternatively, if the cost of false positives and false negatives differs, then the histograms may be weighted accordingly before selecting the maximum (Chai and Bouzerdoum, 2000).

The process can then be made adaptive, by using the classification results to further train the classifier. For this to provide new data, rather than simply to reinforce the existing classification, the output has to be filtered to remove as many misclassifications (both false positives and false negatives) as possible before being fed back. This is shown in block diagram form (without the control logic) in Figure 7.26.

Using only the chrominance components will make the colour segmentation independent of the light intensity. However, it has also been shown (Kakumanua *et al.*, 2007) that this will also decrease the discrimination ability, and that better accuracy is obtained with a full three-dimensional colour space.

Toledo *et al.* (2006) have implemented on an FPGA a histogram-based skin colour detector using a combination of three-dimensional probability maps (histograms) with 32 bins per RGB and YCbCr channel and a combination of two-dimensional histograms using $I-Q$, and $U-V$ components. They used a software processor to monitor the results, and adapt the segmentation parameters to improve the tracking performance.

7.2.4 Colour Indexing

Histogram similarity, as described in Section 7.1.5, can readily be extended to identification of colour images. This histogram in Equation 7.48 is simply replaced with a multidimensional histogram. The main application of histogram similarity is searching for similar images within an image database. The colour histogram contains much important information that can be used to distinguish different colour images, is invariant to translation and rotation, and is relatively insensitive to the orientation of the object and occlusion. If the colour histogram has a reduced number of bins then the histograms can be much faster to compare than the images themselves. Reasonable results can be achieved with as few as 200 bins (Swain and Ballard, 1991).

One limitation of simple colour indexing is that the search is linear with the size of the database. However, the scaling factor may be improved in a number of ways. The simplest is to reduce the number of bins by combining the counts of groups of adjacent bins. Since:

$$\min(A, A') + \min(B, B') \leq \min(A + B, A' + B') \tag{7.75}$$

the reduced resolution will always give a higher match score than the higher resolution histogram. Therefore, database entries with a low match score may be eliminated at a lower cost.

For many histograms, most of the total count is concentrated in a few bins, corresponding to the dominant colours in the image. The match from these bins has the greatest influence on the match score. An approximate match score may be produced quickly by comparing only those bins with the database histograms. The search speed may be further improved if the database is sorted based on the count in each bin (Swain and Ballard, 1991). This requires a separate database index for each bin, but this index only needs to include those database histograms for which that bin is significant. In this way, large sections of the database that will give a poor match are not even searched. Only those that are likely to give a good match are compared and, even then, only a small number of bins need to be matched. Swain and Ballard (1991) demonstrated that as few as 10 bins actually need to be compared, significantly improving the search time.

If the database contains objects that may be present within the image, then once a match is found, the next step is to locate the actual object within the image. This can be accomplished through histogram back-projection (Swain and Ballard, 1991). The basic idea for this is illustrated in Figure 7.27. Each pixel in the input image is looked up in both the model histogram from the database, $hist_M$, and the histogram for the image, $hist_I$. The value of the back-projected pixel is then:

$$bp[x, y] = \min\left(\frac{hist_M[I[x, y]]}{hist_I[I[x, y]]}, 1\right) \tag{7.76}$$

This effectively assigns a high value to the pixels that have a strong colour match with the model, and a low value to the pixels that are not significant in the model. The back-projected image is then smoothed using a suitable filter and the location of the maximum detected. Since the largest peak does not necessarily correspond to the object (sometimes it will be the second or third largest peak), an alternative is to threshold the filtered image to detect candidate locations.

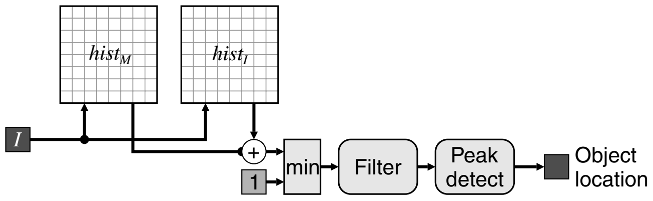


Figure 7.27 Object location through histogram back-projection.

7.2.5 Texture Analysis

Visual texture is characterised by spatial patterns within the pixel values of an image. Since it has a spatial aspect, simple first order statistics are unable to characterise a texture properly. For example, many quite different textures have identical grey-level histograms. It is necessary to capture the spatial relationships using second order statistics. One common method for texture analysis and classification is the based on second order conditional distribution functions, which can be captured in a co-occurrence matrix. This is essentially a two-dimensional histogram, where the first component is the input greyscale image and the second component is the input image offset by distance d in direction θ:

$$H_{d,\theta}[a,b] = \sum_{x,y} \begin{cases} 1, & (I[x,y]=a) \wedge (I[x+\Delta x, y+\Delta y]=b) \\ 0, & \text{otherwise} \end{cases} \tag{7.77}$$

where $||(\Delta x, \Delta y)|| = d$ and $\angle(\Delta x, \Delta y) = \theta$. Typically, the city block distance is used and the directions are limited to 45° increments to simplify calculation.

The spatial grey-level dependence method (SGLDM) (Conners and Harlow, 1980) of texture analysis extracts several features from the co-occurrence matrix. Haralick *et al.* (1973) define 14, of which five are commonly used (Conners and Harlow, 1980). To obtain meaningful results for comparison or classification, the counts within the two-dimensional histogram must be normalised to give probabilities.

$$\text{Energy}: \quad E(H_{d,\theta}) = \frac{\sum_{i,j} \left(H_{d,\theta}(i,j)\right)^2}{\left(\sum_{i,j} H_{d,\theta}(i,j)\right)^2} \tag{7.78}$$

$$\text{Entropy}: \quad H(H_{d,\theta}) = -\frac{\sum_{i,j} H_{d,\theta}(i,j)\log_2 H_{d,\theta}(i,j)}{\sum_{i,j} H_{d,\theta}(i,j)} \tag{7.79}$$

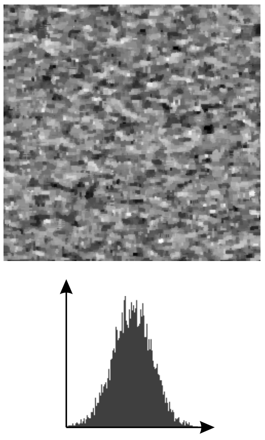
$$\text{Correlation}: \quad C(H_{d,\theta}) = \frac{\sum_{i,j}(i-\mu_i)(j-\mu_j)H_{d,\theta}(i,j)}{\sigma_i\sigma_j} \tag{7.80}$$

$$\text{Local homogeneity}: \quad L(H_{d,\theta}) = \frac{\sum_{i,j} \frac{H_{d,\theta}(i,j)}{1+(i-j)^2}}{\sum_{i,j} H_{d,\theta}(i,j)} \tag{7.81}$$

$$\text{Inertia or contrast}: \quad I(H_{d,\theta}) = \frac{\sum_{i,j} (i-j)^2 H_{d,\theta}(i,j)}{\sum_{i,j} H_{d,\theta}(i,j)} \tag{7.82}$$

where μ_i, μ_j, σ_i, and σ_j are the means and standard deviations of the i and j components respectively of the SGLDM.

Several features are used, over a range of angles, and for a selection of distances that characterise the textures of interest. Some of the features are shown for a sample texture in Figure 7.28, along with the



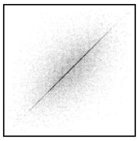
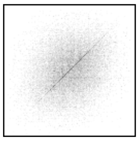
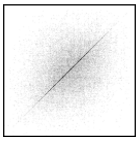
θ	H	$E(H)$	$H(H)$	$C(H)$	$L(H)$	$I(H)$
0°		7.19×10^{-4}	11.57	0.606	0.216	276
45°		3.27×10^{-4}	12.06	0.260	0.095	517
90°		4.69×10^{-4}	11.89	0.356	0.151	451

Figure 7.28 A sample texture with its histogram and some associated SGLDM features for $d = 3$.

associated SGLDMs displayed as images. The one-dimensional histogram is also shown for reference; it is indistinguishable from a Gaussian distribution and is unable to capture any spatial relationships.

If the whole image contains a single texture, the above analysis is suitable. However, for texture segmentation, the co-occurrence matrices must be calculated from smaller regions or windows within the image. The co-occurrence matrix is more accurate when formed from a large number of pixels. However, for small windows, the measures can become meaningless unless the number of bins in the histogram is reduced accordingly. This has an advantage from an implementation perspective, since the memory required to store the matrix is smaller and calculating the texture features is faster. On an FPGA, the different features may be calculated in parallel as the matrix is scanned. If several matrices are used (with different separations and orientations), the features from these may also be calculated in parallel. Tahir *et al.* (2003a) do exactly this; they build 16 co-occurrence matrices in parallel with distance ranges of $d = \{1, 2, 3, 4\}$ and angles $\theta = \{0, 45, 90, 135\}$, with 32 pixel value bins in each dimension of their two-dimensional histogram. They also used the FPGA to calculate seven texture features from each matrix in parallel (Tahir *et al.*, 2003b).

8 Local Filters

Local filters extend point operations by having the output be some function of the pixel values within a local neighbourhood or window:

$$Q[x,y] = f(I[x,y],\ldots,I[x+\Delta x, y+\Delta y]), \quad (\Delta x, \Delta y) \in \mathbf{W} \tag{8.1}$$

where $\mathbf{W}$ is the window or local neighbourhood centred on $I[x,y]$, as illustrated in Figure 8.1. The window can be any shape or size, but is usually square, $W \times W$ pixels in size, with W odd so that the window centre is well defined. As the window is scanned through the input image, each possible position generates an output pixel according to Equation 8.1. Again, f is any function, with the particular function determining the type of filter. Since the output depends not on only the input pixel but also its local context, filters can be used for noise removal or reduction, edge detection, edge enhancement, line detection and feature detection.

The software approach to filtering has both the input and output images stored in frame buffers. The algorithm iterates for each output pixel, retrieving the pixels within the window in the input image and applying the filter function. In this form, the algorithm is ultimately limited by the bandwidth of the input memory.

Any acceleration of filtering must exploit the fact that each pixel is used in multiple windows. With stream processing, this is accomplished through caching pixels as they are read in to enable them to be reused in later window positions.

8.1 Caching

Pixel caching for stream processing of a $W \times W$ filters requires a series of $W - 1$ row buffers, as described in Section 5.2.3. Scanning the window through the image is equivalent to streaming the image through the window. The row buffers can either be placed in parallel with the window or, since the window consists of a set of shift registers, in series with the window as shown in Figure 8.2. Computationally they are equivalent. The parallel row buffers need to be slightly longer (the full width of the image) but have the advantage that they are kept independent of the window and filter.

Variations on this stream access pattern are also possible. If the resources are limited and the row buffer is too long for the available memory, then the image may be scanned down the columns rather than across the rows. Alternatively, the image may be partitioned and processed as a series of vertical strips (Sedcole,

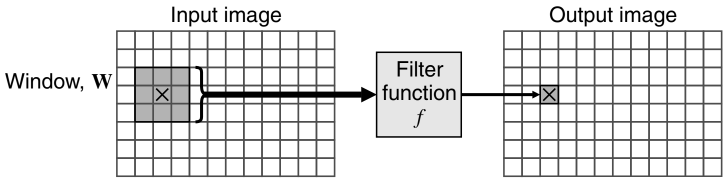


Figure 8.1 A window filter. The shaded pixels represent the input window located at × that produces the filtered value for the corresponding location in the output image. Each possible window position generates the corresponding pixel value in the output image.

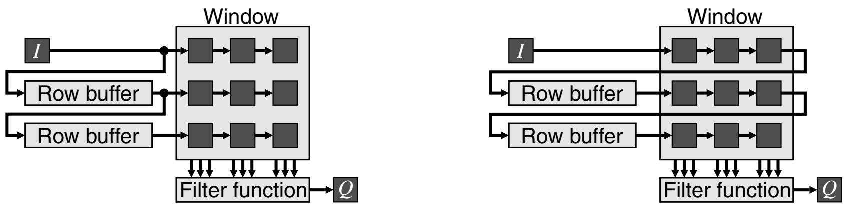


Figure 8.2 Row buffers. Left: in parallel with window; right: in series with window.

2006), as illustrated in Figure 8.3. In the latter case, there is a small overhead because to process the complete image the vertical strips will have to overlap by $W-1$ pixels.

Conversely, if additional resources and memory bandwidth are available, the image may be partitioned with multiple filters operating in parallel. However, rather than partition the image as illustrated on the right in Figure 8.3, it can be more efficient to simultaneously process multiple adjacent rows (Draper *et al.*, 2003). This requires multiple pixels to be input per clock cycle and exploits the fact that the windows for vertically adjacent outputs overlap significantly, as can be seen in Figure 8.4. This is partially unrolling the vertical scan loop through the image. Note that the number of row buffers is unchanged. For an unroll factor of k (processing k rows of pixels in parallel) the combined window size is $W \times (W+k-1)$. Of this, k rows of data are streamed in from memory in parallel, so the remaining $W-1$ rows must come from row buffers. These buffers are arranged with a pitch of k rather than simply being chained. The parallel implementation will require k copies of the filter function. However, with some filters, the overlap in windows can even enable some of the filter function logic to be shared (Lucke and Parhi, 1992), reducing the resource requirements further.

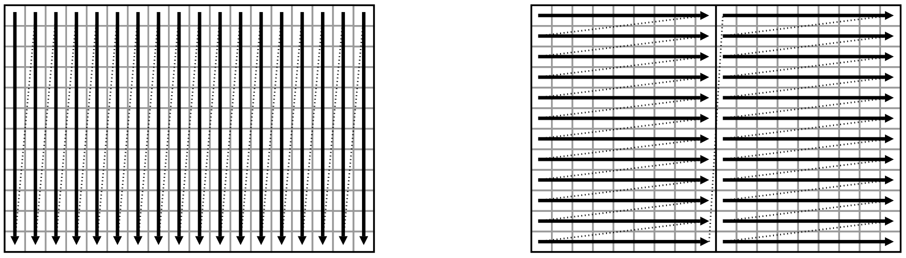


Figure 8.3 Access patterns for stream processing.

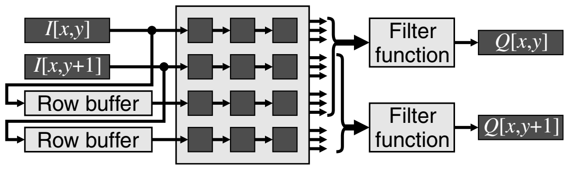


Figure 8.4 Partially unrolling the loop vertically, streaming in multiple rows in parallel.

Partially unrolling the scan loop vertically will require reading pixels from k rows of the image. However, these pixels are not usually stored together in memory, requiring an awkward memory access pattern. Pixels are more likely to be packed in memory, so it may be preferable to read multiple horizontal pixels simultaneously. Figure 8.5 demonstrates a partial horizontal unrolling of the scan loop. This time k horizontally adjacent pixels are processed simultaneously, making the combined window size $(W+k-1) \times W$. This also requires the shifting pitch of the window registers to be k pixels. $W-1$ row buffers are still required, but their aspect ratio must be changed to provide the wider combined data. Again, k copies of the filter function are required, although for some functions some of the filter logic may be able to be shared.

Although the internal memory of most FPGAs is dual-port (only simple dual-port is required for circular memory-based row buffers), it is possible to cache the input rows using only single-port memory. This requires W buffer memories, with the input being written to one buffer and row 0 of the window, while the other rows of the window are read from other row buffers as shown in Figure 8.6. Table 8.1 shows the corresponding control for a 3×3 window. The state is easiest implemented as a ring counter, directly controlling the read/write signals to the memories and the multiplexers.

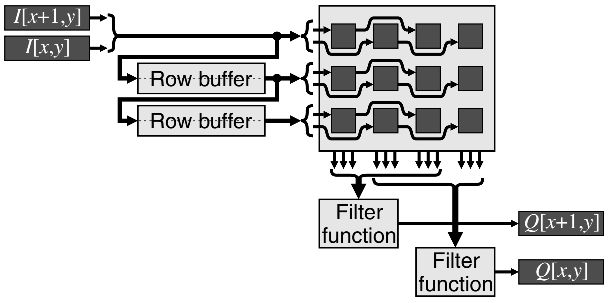


Figure 8.5 Partially unrolling horizontally, streaming multiple pixels each clock cycle.

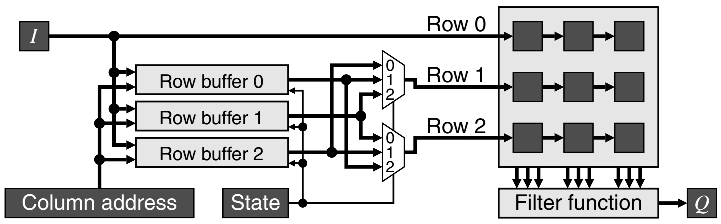


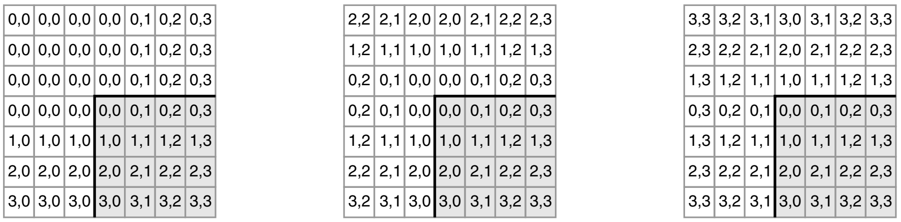
Figure 8.6 Filtering with single-port row buffers (see Table 8.1 for the control sequence).

Table 8.1 Control for single-port row buffer in Figure 8.6

Input row	State	Row buffer 0	Row buffer 1	Row buffer 2	Output row
0	0	Write	(Row 2)	(Row 1)	—
1	1	Row 1	Write	(Row 2)	—
2	2	Row 2	Row 1	Write	1
3	0	Write	Row 2	Row 1	2
4	1	Row 1	Write	Row 2	3
5	2	Row 2	Row 1	Write	4
6	0	Write	Row 2	Row 1	5
	...	...	...	...	...

One problem with filtering is managing what happens when the window is not completely within the input image (Bailey, 2011b). Very few papers consider these boundary conditions – the design to handle them properly can take more effort than to manage the normal case where all of the data is available. For some solutions, the logic required to handle the boundary cases can be just as much or more than the regular window logic. There is a wide range of solutions that can be considered:

- The output could be left uncalculated for the boundary pixels. The output image would become $(W-1) \times (W-1)$ pixels smaller. In many applications and with a small window size this would not be a problem.
- The filter function could be modified to work with the smaller window when part of the window extends beyond the boundary of the image. For many filters, this is not an option and, where it is, the logic requirements can grow rapidly to manage the different cases.
- The input image could be made larger. This requires manufacturing $(W-1)/2$ pixel values beyond each border of the image so that there is sufficient data to fill the window to produce a full-sized image at the output. An original $M \times N$ image is extended to produce the $(M+W-1) \times (N+W-1)$ input image that is processed. The following options describe different techniques for extending the input image in order of complexity.
- The input stream can just be wrapped. This is equivalent to ignoring the problem and simply processing the $M \times N$ image. The pixels at the end of a row are immediately followed by the pixels at the start of the next row. Similarly, the last row of one frame is immediately followed by the first row of the next frame. Since the opposite edges of an image are not likely to be related in most applications, the border pixels are likely to be invalid. However, the advantage over the first case above is that output pixels are produced and the output image is the same size as the input.
- A predefined value could be used for the pixels outside the image. This could be black (0), white, or some other value depending on the type of filter and the expected scene.
- The pixel values on the border of the image could be duplicated, assigning the value to that of the nearest available pixel. This is effectively nearest neighbour extrapolation. This approach has been used with median and rank filters (Choo and Verma, 2008).
- The previous extension made the pixels outside the image flat. With some filter functions, this may result in undesired artefacts around the edge. An alternative is to mirror the rows and columns just inside the border to the outside of the image. Mirroring is commonly used with wavelet filters. There are two approaches to mirroring that have been used. One mirrors using the edge of the image, duplicating the border pixels (Kurak, 1991), and the other mirrors from the border row, without duplicating the border pixels (McDonnell, 1981; Benkrid *et al.*, 2003a). These approaches are compared in Figure 8.7.



0,0	0,0	0,0	0,0	0,1	0,2	0,3
0,0	0,0	0,0	0,0	0,1	0,2	0,3
0,0	0,0	0,0	0,0	0,1	0,2	0,3
0,0	0,0	0,0	0,0	0,1	0,2	0,3
1,0	1,0	1,0	1,0	1,1	1,2	1,3
2,0	2,0	2,0	2,0	2,1	2,2	2,3
3,0	3,0	3,0	3,0	3,1	3,2	3,3

2,2	2,1	2,0	2,0	2,1	2,2	2,3
1,2	1,1	1,0	1,0	1,1	1,2	1,3
0,2	0,1	0,0	0,0	0,1	0,2	0,3
0,2	0,1	0,0	0,0	0,1	0,2	0,3
1,2	1,1	1,0	1,0	1,1	1,2	1,3
2,2	2,1	2,0	2,0	2,1	2,2	2,3
3,2	3,1	3,0	3,0	3,1	3,2	3,3

3,3	3,2	3,1	3,0	3,1	3,2	3,3
2,3	2,2	2,1	2,0	2,1	2,2	2,3
1,3	1,2	1,1	1,0	1,1	1,2	1,3
0,3	0,2	0,1	0,0	0,1	0,2	0,3
1,3	1,2	1,1	1,0	1,1	1,2	1,3
2,3	2,2	2,1	2,0	2,1	2,2	2,3
3,3	3,2	3,1	3,0	3,1	3,2	3,3

Figure 8.7 Edge extension schemes. Left: border pixel duplication; centre: mirroring with duplication; right: mirroring without duplication.

Whatever extension method is used, it is easier to apply it between the row buffering and forming the filter window than to attempt to modify the filter function to handle the boundary conditions. Specific mechanisms for achieving this are described after discussing the priming and flushing requirements of filter pipelines.

It is necessary to load a whole window of data before the output starts. The time required to prime the filter and fill the row buffers for a $W \times W$ window is the time for loading $W-1$ complete rows of $(M+W-1)$ pixels plus an additional W pixels. Therefore:

$$\begin{aligned} t_{Prime} &= (M+W-1)(W-1)+W \\ &= (M+W)(W-1)+1 \end{aligned} \tag{8.2}$$

However, at the top of the image, half of these rows are outside the boundary of the image. The latency of the filter is the time from when a pixel is input to when the corresponding output pixel is produced, and can be broken into two components. The first is that of the windowing operation and the second is the latency of the filter function itself. The latency of windowing is the time from when the centre pixel is loaded to when the last pixel is loaded for that window position, enabling the output to be calculated. This gives:

$$\begin{aligned} t_{Latency} &= t_{Latency,W} + t_{Latency,f} \\ &= (M+W-1)\frac{W-1}{2} + \frac{W+1}{2} + t_{Latency,f} \\ &= \frac{1}{2}(M+W)(W-1)+1+t_{Latency,f} \end{aligned} \tag{8.3}$$

If the system is sink-driven, then it is necessary to start the filtering at least this length of time before the first data is required on the output. The filter must also be clocked for this many clock cycles after the last data is input to flush the contents of the filter. Note the times of Equations 8.2 and 8.3 are further complicated if the input stream has additional horizontal and vertical blanking intervals.

Part of priming the filter window includes loading the extended input image (beyond the borders of the original image). Consider the case of boundary replication – the first and last rows must be loaded $(W+1)/2$ times, and similarly the first and last pixels on each row. Rather than explicitly reload them, it is possible to reuse the loaded values using the scheme of Figure 8.8. As the first pixel of each row is streamed in, the value is loaded into all of the shift register stages on the input. This allows the first pixel to be replicated an additional $(W-1)/2$ times. The rest of the stream also passes through the input shift register to enable the whole row to be streamed continuously. As the last pixel of the row is loaded into the window, the last shift register stage is fed back replicating the last pixel of the row. At this stage the first row buffer contains the whole first row, including the duplicated pixels at the boundaries. This row is recirculated to the input $(W-1)/2$ times through the R input of the multiplexer, repeating the first row

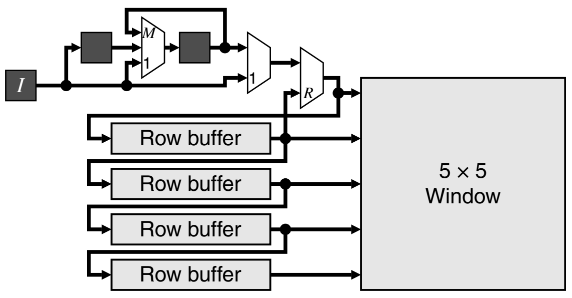


Figure 8.8 Priming the filter. Row and pixel replication at the boundaries.

as necessary. Then the remainder of the image is streamed in, with the last row repeated as before while flushing data out of the window.

A similar approach may be used for mirroring the input, although the multiplexing gets a little more complex. One disadvantage of this approach is that it requires at least $(M + W-1) \times (N + W-1)$ clock cycles to process the whole image, and the row buffers need to be slightly longer than one row. However, since only $M \times N$ pixels are loaded in and $M \times N$ pixels are output, it should be possible to process pixels continuously, without the delays associated with repeating pixels and rows. An approach that does that, using mirroring without duplication, for a 5×5 window is demonstrated in Figure 8.9 (Bailey, 2011b).

The operation of this is a little more complex. Firstly consider the operation along a row. In normal operation (in the centre of the image) the bottom row of shift registers is used to shift the pixels along. At the end of the row, the first pixels of the next row are clocked into the top row of shift registers. Meanwhile, the appropriate pixel from within the window is fed back to the start of the window to give the mirroring. The labels on the input multiplexer refer to the second last (-2) and last (-1) positions of the window on the row. On the next clock cycle, the window should correspond to the first position on the next row. While flushing the end of one row, the initial pixels required to prime the window for the next row have been loaded. These are now transferred to the appropriate window registers (through multiplexer inputs labelled 1) to give the mirrored window values. Note that no clock cycles have been lost here – the window transitions from the last position on one row directly to the first position on the next row.

A similar procedure could be used for transitioning from the bottom of one image to the top of the next. However, this would require two extra row buffers to perform the counterpart of the preload registers along the row. Instead, the outputs from the row buffers are routed directly to the corresponding rows of the window. Mirroring requires feeding back the data at the bottom of the window; multiplexer inputs -2 and -1 refer to the second last and last rows of the image respectively. On the first and second rows of the new

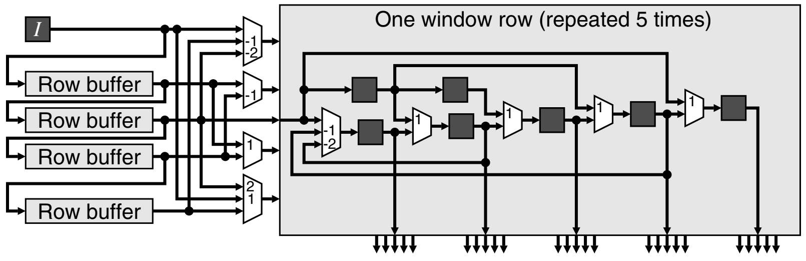


Figure 8.9 Priming the filter with mirroring and no additional delay between rows.

image, multiplexer inputs 1 and 2 respectively feed forward the row buffered and input data to reflect the mirroring.

This section has shown how row buffers can be used to cache the data associated with local filters. Each pixel is only loaded once, with the value reused for all of the window positions that require it. A throughput of one pixel per clock cycle can be maintained (or higher if partially unrolling the raster scan). It is also possible to process the borders of the image without introducing additional latency with only a small amount of additional control logic.

8.2 Linear Filters

The filter function for a linear filter is a weighted sum of the pixel values within the window.

$$Q[x,y] = \sum_{i,j\in\mathbf{W}} w[i,j]I[x+i,y+j] \tag{8.4}$$

The particular set of weights is sometimes called the filter *kernel*, with the filter function determined by the kernel used. Linear filtering is equivalent to performing a two-dimensional convolution with the flipped kernel, $w[-i,-j]$. Since the Fourier transform of a convolution is a product of the respective Fourier transforms (Bracewell, 2000), the operation of linear filters can be also be considered from their effects in the frequency domain. This is considered further in Chapter 10.

In signal processing terminology, the filters described by Equation 8.4 are *finite impulse response* filters. Recursive, *infinite impulse response* filters can also be used; these use the previously calculated outputs in addition to the inputs to calculate the current output:

$$Q[x,y] = \sum_{i,j\in\mathbf{W}} w_q[i,j]Q[x+i,y+j] + \sum_{i,j\in\mathbf{W}} w[i,j]I[x+i,y+j] \tag{8.5}$$

By necessity, w_q must be zero for the pixels that have not been calculated yet. While recursive filters may be used to implement some finite impulse response filters (for example box filters described below), they are otherwise not often used in image processing. The nonlinear phase response results in a directional smearing of information within the image. To avoid this, it is necessary to apply each filter twice, once with a top-to-bottom, left-to-right scan pattern, and again with a bottom-to-top, right-to-left scan pattern. This makes such filters harder to pipeline, although for a given frequency response, infinite impulse response filters require a significantly smaller window (Mitra, 1998). Only finite impulse response filters are discussed further in this section.

8.2.1 Noise Smoothing

One of the most common filtering operations is to smooth noise. The basic principle behind noise smoothing is to use the central limit theorem to reduce the noise variance. With only one image, it is not possible to average with time as was used in Section 6.2.1. Instead, adjacent pixels are averaged. To avoid the output value from being different from the input in uniform regions of the image, a noise smoothing filter requires:

$$\sum_{i,j\in\mathbf{W}} w[i,j] = 1 \tag{8.6}$$

Most images have their energy concentrated in the low frequencies, with the high frequencies containing information relating to fine details and edges (wherever the pixel values are changing quickly).

Random (white) noise has a uniform frequency distribution. Therefore, a low pass filter will remove the most noise while having the least impact on the energy content of the image. Unfortunately, in attenuating the noise, the high frequency content of the image is also attenuated, resulting in blur.

With linear filters, their effect on the noise and the image content can be analysed separately. Assuming that the noise is independent for each pixel, then from Equation 6.16 the best improvement in variance for a given size window will result if the weights are all equal. A larger width window will result in more noise smoothing. These noise suppression effects can be clearly seen in Figure 8.10, where an image with artificially added noise is filtered.

Spatial averaging will result in a loss of fine detail and a blurring of edges. Unfortunately, these effects will become worse with increased noise smoothing. Therefore, there is a trade-off between blurring and noise suppression, as is also seen in Figure 8.10.

The Fourier transform of a uniform filter is a sinc function, which has poor sidelobe performance at high frequencies. These sidelobes mean that significant high frequency noise still remains after filtering. This is apparent in the fine textured region in the centre image in Figure 8.10. A better low pass filter can be obtained by rolling off the weights towards the edge of the kernel. While the noise variance will be higher, there is better attenuation of the high frequency noise and fewer filtering artefacts.

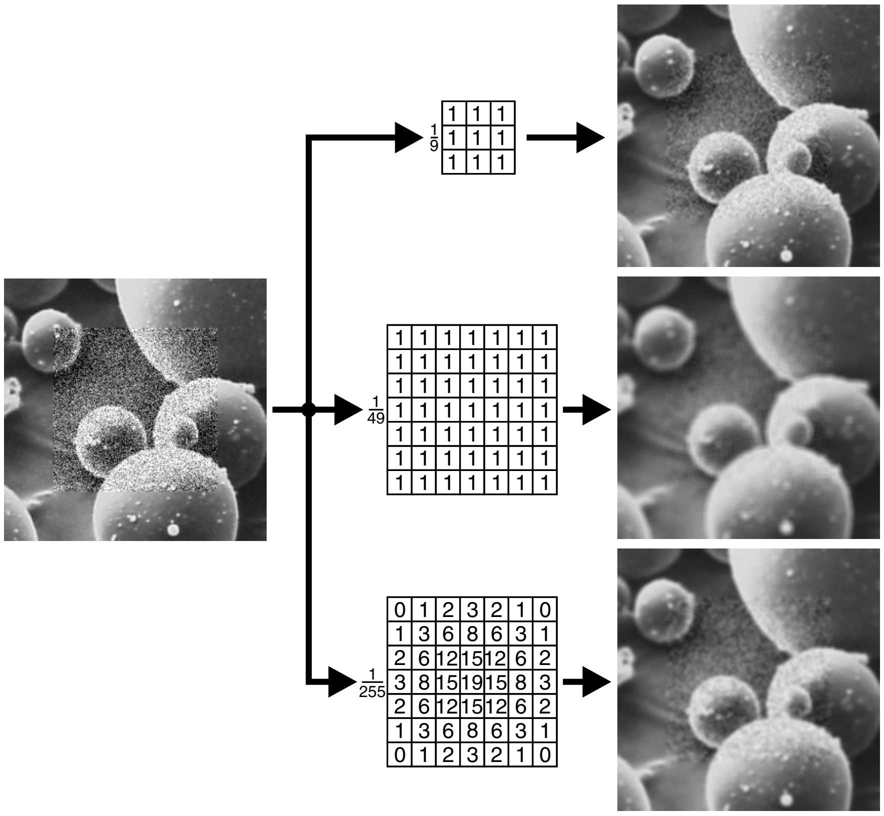


Figure 8.10 Linear noise smoothing filters.

A commonly used filter has Gaussian weights:

$$w_G[i,j] = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}} \tag{8.7}$$

where σ is the standard deviation or equivalent radius of the filter. Note that the Gaussian weights never actually go to zero, so a practical implementation will require truncating the weights within a finite rectangular region. The filter width, W, should generally be greater than about 4σ and, if accuracy is important, then $W \geq 6\sigma$. With truncation, the kernel coefficients may require rescaling to make the total equal to one. The filter in the bottom row of Figure 8.10 corresponds to $\sigma = 1.5$.

One application of Gaussian filtering is to obtain a scale–space representation of an image (Lindeberg, 1994). Filtering by a series of Gaussian filters with successively larger standard deviations will remove from the image features of increasing size or scale.

8.2.2 Edge Detection

Edge detection is another common application of filtering. At the edges of objects, there is often a change in pixel value, reflecting the contrast between the object and the background, or between two objects. Since the edge is characterised by a difference in pixel values, a differencing filter can be used to detect edges. Differencing is sensitive to noise, so the top filter in Figure 8.11 averages vertically (to give noise smoothing) while differencing horizontally to detect vertical edges. Note that differencing like this will only detect edges of particular orientation. Alternatively, since the edges contain high frequency information (this was lost through the low pass noise smoothing filter), a high pass filter can be used to detect edges of all orientations. The main limitation of high pass filters is their strong sensitivity to noise (see the middle filter of Figure 8.11).

The noise sensitivity can be improved significantly by first filtering the image with a Gaussian filter. However, there is a trade-off between accurate detection (improves with smoothing) and accurate localisation (deteriorates with smoothing) (Canny, 1986). With the smoothed image, the peaks of the first derivative will correspond to the location of the edges. These can either be found directly or by finding the zero crossings of the second derivative. (Note that finding the peaks or zero crossings requires a nonlinear filter, which will be described in the next section.)

With linear filters, it does not matter whether the derivatives are taken before or after smoothing. Indeed they can even be combined with the smoothing to give a single filter. With the first derivative, there are two filters for derivatives in each of the x and y directions. The corresponding weights are:

$$\begin{aligned} w_{G_x}[i,j] &= \frac{i}{2\pi\sigma^4} e^{-\frac{i^2+j^2}{2\sigma^2}} \\ w_{G_y}[i,j] &= \frac{j}{2\pi\sigma^4} e^{-\frac{i^2+j^2}{2\sigma^2}} \end{aligned} \tag{8.8}$$

Using the Laplacian for the second derivative makes the second derivative filter (Laplacian of Gaussian or LoG filter) (Marr and Hildreth, 1980):

$$w_{LoG}[i,j] = \frac{i^2+j^2-2\sigma^2}{2\pi\sigma^6} e^{-\frac{i^2+j^2}{2\sigma^2}} \tag{8.9}$$

The LoG filter can be used to detect edges of all orientations. It is effectively a band-pass filter, where the standard deviation, σ, controls the centre frequency or the scale from a scale–space perspective, at which edges are detected. The bottom image of Figure 8.11 shows the Laplacian of Gaussian filter with $\sigma = 1.12$.

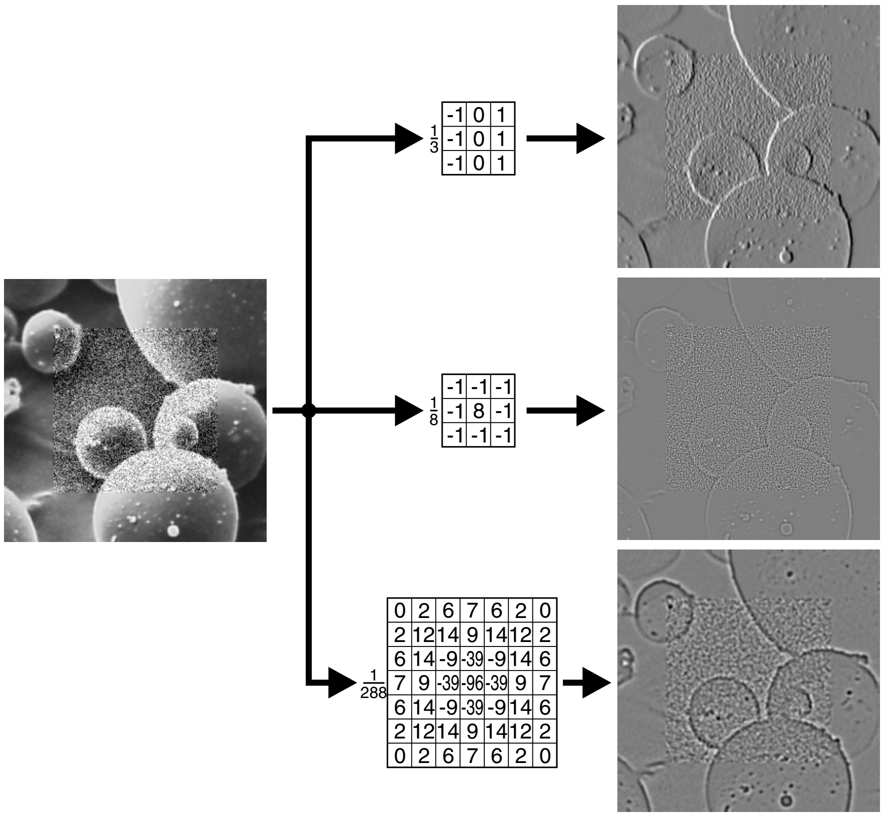


Figure 8.11 Linear edge detection filters. Top: vertical edge detection; centre: high pass filter; bottom: Laplacian of Gaussian filter. The output images are offset to show negative values.

If the Gaussian and Laplacian operations are separated, then one of the Laplacian kernels in Figure 8.12 may be used.

A filter with a similar response to the LoG filter is the difference of Gaussians (or DoG filter). It is formed by subtracting the output of two Gaussian filters with different standard deviations:

$$w_{DoG}[i,j] = \frac{1}{2\pi k^2\sigma^2} e^{-\frac{i^2+j^2}{2k^2\sigma^2}} - \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}} \tag{8.10}$$

where $k \approx 1.5$ is the ratio of standard deviations of the two Gaussians. This corresponds closely to the receptive field response within the human visual system (Wilson and Giese, 1977).

$$\frac{1}{4}\begin{array}{|c|c|c|}\hline 0 & 1 & 0 \\ \hline 1 & -4 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \qquad \frac{1}{8}\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & -8 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Figure 8.12 Filter kernels for a Laplacian filter.

Note that the weights for an edge detection filter should sum to zero, so that the response is zero for regions of uniform pixel value. For edge detection, the weights can be multiplied by an arbitrary constant, since the edges correspond either to the positions of the maximum derivatives, or the locations of the zero crossings of the Laplacian.

8.2.3 Edge Enhancement

An edge enhancement filter sharpens edges by increasing the gradient at edges. In this sense it is the opposite of edge blurring, where the gradient is decreased by attenuating the high frequency content. Edge enhancement works by boosting the high frequency content of the image. One such filter is shown in Figure 8.13. A major limitation of the high frequency gain is that any noise within the image will also be amplified. This cannot be easily avoided, so linear edge enhancement filters should only be applied to relatively noise-free images. Over-enhancement of edges can result in ringing, which will become more severe as the enhancement is increased.

The weights of an edge enhancement filter must sum to one to avoid changing the global contrast of the image.

8.2.4 Linear Filter Techniques

Although most of the examples shown in this section are 3×3 filters, the techniques readily extend to larger filter sizes. The obvious implementation of a linear filter is illustrated in Figure 8.14, especially if the FPGA has plentiful multiplication or DSP blocks. Each pixel within the window is multiplied in parallel by the corresponding filter weight and then added. Note that the bottom right window position is the oldest pixel and corresponds to the top left pixel within the image in the image. The filter weights are shown here as constants, but could also be programmable and stored in a set of registers.

The propagation delay through the multiplication and adders may exceed the system clock cycle and require pipelining. Since each input pixel contributes to several pixels, rather than delaying the inputs and accumulating the output, the transpose filter structure performs all the multiplication with the input and delays the product terms (Mitra, 1998). The transpose filter structure does this by feeding data through the filter backwards (swapping the input and the output) and swapping summing junctions and pick-off points. This is applied to each row in Figure 8.15, and to the whole window (Benkrid *et al.*, 2003a) in Figure 8.16. The advantage of the transposed structures is that the output is automatically pipelined. When transposing the whole window, it is necessary to cache the partial sums using row buffers until the remainder of the window appears. Depending on the filter coefficients, these partial sums may require a

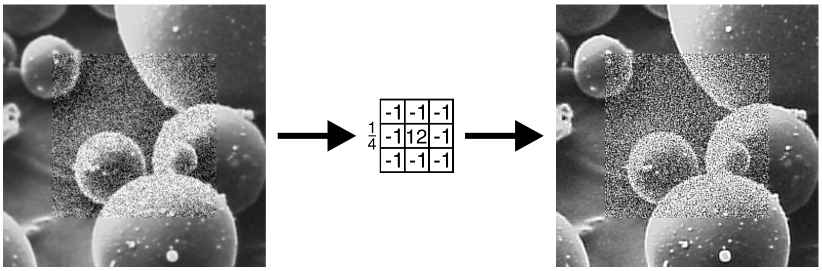


Figure 8.13 Linear edge enhancement filter.

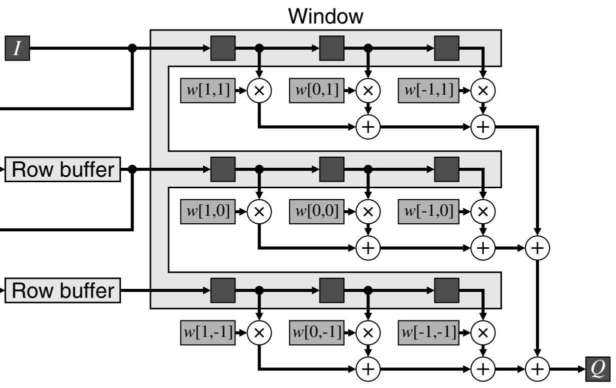


Figure 8.14 Direct implementation of linear filtering.

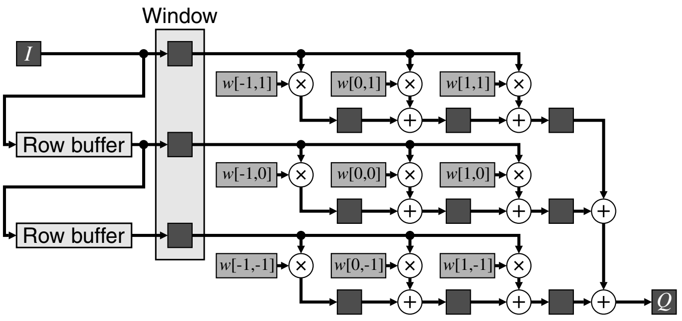


Figure 8.15 Pipelined linear filter. Note the different order of coefficients to Figure 8.14 because the transpose filter structure is used for each row.

few guard bits to prevent accumulation of rounding errors. The row buffers, therefore, may need to be a few bits wider than the input pixel stream. The transposed filter structures may also require a little more effort to manage the image boundaries, although that can also be incorporated into the filter structure (Benkrid *et al.*, 2003a).

There are several techniques that may be used to simplify and reduce the logic required by linear filters. Many filters are symmetric, therefore many of their filter coefficients share the same value. If using the direct implementation of Figure 8.14, the corresponding input pixel values can be added prior to the multiplication. Alternatively, since each input pixel is multiplied by a number of different coefficients, the input pixel value can be multiplied by each unique coefficient once, with the results cached until needed (Porikli, 2008). This fits well with the transposed implementation of Figure 8.16. In both cases, the number of multipliers is reduced to the number of unique coefficients.

If using a relatively slow clock rate, and hardware is at a premium, then significant hardware savings can often be made by doubling or tripling the clock rate but keeping the same data throughput. The result is a multiphase design which enables expensive hardware (multipliers in the case of linear filters) to be reused in different phases of the clock. This is shown for a three-phase system in Figure 8.17. With the accumulator, a 0 is fed back in phase zero to begin the accumulation for a new pixel.

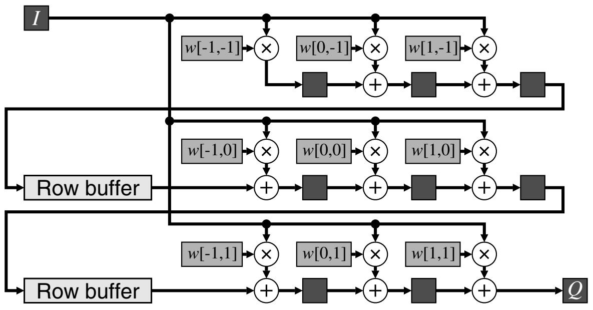


Figure 8.16 Transposed implementation of linear filtering. Note the changed coefficient order.

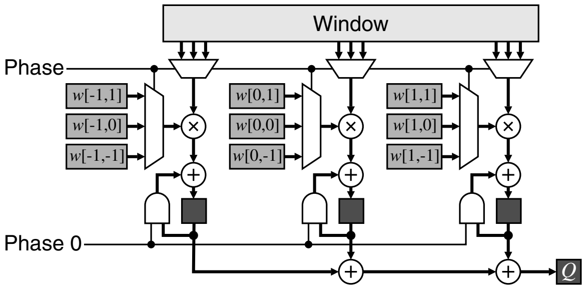


Figure 8.17 Reducing the number of multipliers by using a higher clock speed.

Many useful filters are separable:

$$w[i,j] = w_x[i]w_y[j] \tag{8.11}$$

This means that a two-dimensional filter may be decomposed into a cascade of two one-dimensional filters: a $1 \times W$ filter operating on the columns and a $W \times 1$ filter operating on the rows. This will reduce the number of both multiplications and associated additions from W^2 to $2W$. Note that the column filter does not require a separate pass through the image. It, too, can be streamed by replacing the pixel delays within the window by row buffers, as shown in Figure 8.18. This is effectively implementing the filters

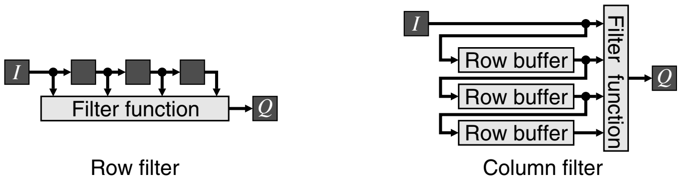


Figure 8.18 Converting a row filter to a column filter by replacing pixel delays with row buffers.

for each column in parallel, but sequentially stepping the filter function from one column to the next as the data is streamed in. Consequently, the row and column filters may be directly pipelined.

While not all filters are directly separable, even arbitrary two-dimensional filters may be decomposed as a sum of separable filters (Lu *et al.*, 1990; Bouganis *et al.*, 2005)

$$w[i,j] = \sum_k w_{kx}[i]w_{ky}[j] \tag{8.12}$$

through singular value decomposition of the filter kernel. The vectors associated with the k largest (significant) singular values will account for most of the information within the filter. With such a decomposition, the column filters should be implemented before the row filters to enable a single set of row buffers to be used for all the parallel filters. For a similar reason, the transpose filter structure cannot be used for the column filters, although it may be used for the row filters. Such a decomposition can give savings if $k \leq W/2$, which will be the case if the original filter is symmetrical either horizontally or vertically.

Also worth considering are series decompositions of filters. The kernel of a composite filter is the convolution of the kernels of the constituent filters:

$$\begin{aligned} w[i,j] &= w_1[i,j] \otimes w_2[i,j] \\ &= \sum_{x,y} w_1[x,y]w_2[i-x,j-y] \end{aligned} \tag{8.13}$$

A common example of this is approximating a Gaussian filter by repeated application of a rectangular box filter. For k repetitions of a window of width W, the standard deviation of the resultant approximate Gaussian is:

$$\sigma = \sqrt{\frac{1}{12}k(W^2-1)} \tag{8.14}$$

For many applications, three or four repetitions will provide a sufficiently close approximation to a Gaussian filter.

Of course, when implementing constant coefficient filters, if any of the coefficients is a power of two, the corresponding multiplication is a trivial shift of the corresponding pixel value. Such shifts can be implemented without logic. If not using multiplier blocks, the multiplications may be implemented with shift and add algorithms. For any given coefficient, the smallest number of shifts and adds (or subtracts) is obtained by using the canonical signed digit representation for the coefficients. Further optimisations may be obtained by identifying common subexpressions and factorising them out (Hartley, 1991). For example, $165 = 101\ 001\ 01_2$ requires three adders for a canonical signed digit multiplication, but only two by factorising as 5×33 ($= 101_2 \times 1\ 000\ 01_2$). A range of techniques has been developed to find the optimal (in some sense) factorisation and arrangement of terms (Dempster and Macleod, 1994; Potkonjak *et al.*, 1996; Martinez-Peiro *et al.*, 2002; Al-Hasani *et al.*, 2011).

A simple example will illustrate the power of these techniques. Consider a 21×21 Gaussian smoothing filter with $\sigma = 3$. A direct implementation would require 441 multipliers and 440 adders. Representing the coefficients with fixed-point numbers with resolution of 2^{-16} has the coefficients in the range from 0 to 1160, requiring 11 bits for the central few coefficients. Of the coefficients, 40 of them are zero, so do not need to be multiplied or added in. This reduces the number of adders to 400. Of the

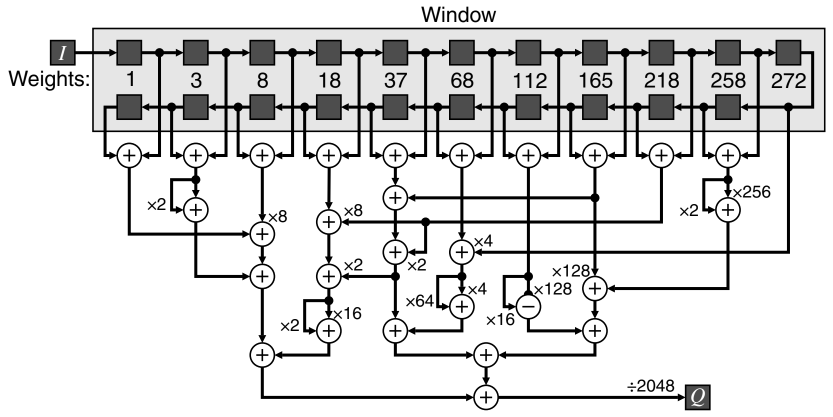


Figure 8.19 Adder only implementation of a 21×1 Gaussian filter with $\sigma = 3$.

coefficients, 96 are direct powers of two (1, 2, 4 and 8); these also will not require multipliers but will still require adders. Since the filter is symmetric, most of the coefficients will be used four or eight times. This can be exploited by adding the corresponding pixel values first and then multiplying by the coefficient. Since there are 40 unique coefficients (not counting the zero or powers of two), this reduces the number of multipliers to 40, although 400 adders are still required. A further 10 of the coefficients can be represented as the sum or difference of two powers of two. This allows those multiplications to be performed by a single addition rather than a multiplication. The number of operations is then 30 multiplications and 410 additions.

A further observation is that the Gaussian filter is separable. This allows the 21×21 filter to be implemented as cascade one-dimensional Gaussian filters (1×21 and 21×1). The fixed-point coefficients with a resolution of 2^{-11} are shown along the top of Figure 8.19. Each one-dimensional filter would require nine multiplications (because of symmetry and two coefficients are powers of two) and 20 additions. A further six coefficients can be represented as a sum or difference of two powers of two, reducing the requirements to three multiplications and 25 additions (one addition can be removed through common subexpression elimination: $272 = 4 \times 68$). The remaining three multiplications can also be eliminated by reusing common subexpressions, in the form of a shifted sum of two existing coefficients ($37 = 2 \times 18 + 1$; $165 = 37 + 128$; $218 = 8 \times 18 + 2 \times 37$) allowing even those multipliers to be replaced with single additions. Therefore, the complete 21×1 filter can be implemented with only 28 additions, as demonstrated in Figure 8.19. A similar filter can be used for the vertical filter, replacing each the register in the chain across the top with a row buffer.

With the increasing prevalence of high speed pipelined multipliers within FPGAs, the need for such decompositions has diminished somewhat in the last few years. The optimised hardware multipliers are hard to out-perform with the relatively slow adder logic of the FPGA fabric (Zoss *et al.*, 2011).

Other decompositions and approximations are also possible, especially for large windows. For example, Kawada and Maruyama (2007) approximate large circularly symmetric filters by octagons and rely on the fact that the pixels with a common coefficient lie on a set of lines. As the window scans, the total for each octagon edge is updated to reflect the changes from one window position to the next.

A simpler version of this can be applied to averaging using a rectangular window with equal weights (McDonnell, 1981). Firstly, a large rectangular window is separable, allowing the filter to be implemented as a cascade of two one-dimensional filters. Secondly, the equal weights make the filter

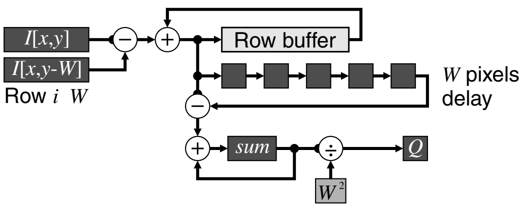


Figure 8.20 Efficient implementation of a $W \times W$ box average filter.

amenable to a recursive implementation:

$$\begin{aligned} S[x+1] &= \sum_{i=-\frac{W-1}{2}}^{\frac{W-1}{2}} I[x+1+i] \\ &= \sum_{i=-\frac{W-1}{2}}^{\frac{W-1}{2}} I[x+i] + I\left[x+1+\tfrac{W-1}{2}\right] - I\left[x-\tfrac{W-1}{2}\right] \\ &= S[x] + I\left[x+\tfrac{W+1}{2}\right] - I\left[x+\tfrac{W+1}{2}-W\right] \end{aligned} \tag{8.15}$$

which reduces the one-dimensional filter to a single addition and subtraction regardless of the size of the window. The same recursive mechanism may also be used for the column filter. An input pixel is then only required twice, once when it first enters the window and again when it leaves the window. For large windows, a large number of row buffers are required to cache the input for its second access. Therefore, if the input stream is from a frame buffer which has sufficient bandwidth to stream from two rows simultaneously, the row buffering requirements can be reduced significantly, with only the current column sum buffered. Such an implementation for a $W \times W$ box average is illustrated in Figure 8.20.

8.3 Nonlinear Filters

While the theory behind linear filters is well established and the filters are relatively simple to implement, restricting f in Equation 8.1 to a linear combination of the input values has some significant limitations (Bailey *et al.*, 1984). In particular, the filters have difficulty distinguishing between legitimate changes in pixel value (for example at an edge) from undesirable changes resulting from noise. Consequently, linear filters will blur edges while attempting to reduce noise, or be sensitive to noise while detecting edges or lines within the image.

Linear filters may be modified to improve their characteristics, for example by making the filter weights adaptive, or dependent on the image content. The simplest of these are *trimmed filters*, which omit some pixels within the window from the calculation. While the mean possesses good noise reduction properties, it is sensitive to outliers. A trimmed mean will discard the outliers by discarding the extreme pixels and calculate the mean of the remainder (Bednar and Watt, 1984). With a heavy tailed noise distribution, this can give an improvement in signal-to-noise ratio. A related application is smoothing noise without significantly blurring edges; it is desirable to average the pixel values only from one side of the edge. By selecting only the pixel values that are similar to the central pixel value, the probability of using pixel values on the opposite side of the edge is reduced, with a corresponding reduction in blur.

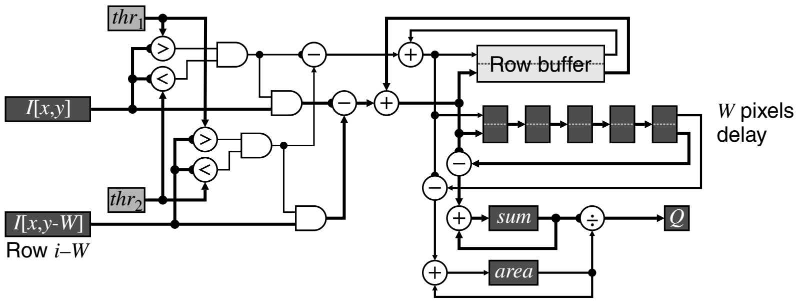


Figure 8.21 The box filter of Figure 8.20 augmented for trimming the input.

Box average filters may also be operated as trimmed filters. For example, the average could be taken only of pixel values within a given range (McDonnell, 1981). The sum of the pixel values is augmented with the count of those that are within range. The corresponding implementation is given in Figure 8.21.

Gated filters are a related form of adaption to trimmed filters. These use one function of the pixel values within the window to select between two or more different filters to produce an output, as represented in Figure 8.22. The gating is determined independently for each output pixel. A simple example of a gated filter is to detect the gradient within the window and only apply smoothing perpendicular to the gradient. Such filters apply smoothing along an edge to reduce noise without significantly blurring the edge. Another example is to determine whether the window is within a uniform region away from an edge or is adjacent to an edge. If an edge is present, an edge enhancement filter may be used, otherwise a noise smoothing filter can be selected. At the extreme, a statistical test may be performed to determine if the pixel values come from a single Gaussian distribution (Gasteratos *et al.*, 2006). If so, the mean can be used to smooth the noise, otherwise the original pixel value is retained to prevent degrading edges (which will generally have a mixed distribution).

Nonlinear combinations of linear filters can also provide more useful outputs than a simple linear filter. Perhaps the most common example of this are the Prewitt and Sobel edge detection filters (Abdou and Pratt, 1979), which combine the outputs of two linear gradient filters; one in the horizontal direction and one in the vertical direction. Let H be the horizontal gradient and V be the vertical gradient. The magnitude of the two-dimensional gradient is ideally given by:

$$Q = \sqrt{H^2 + V^2} \tag{8.16}$$

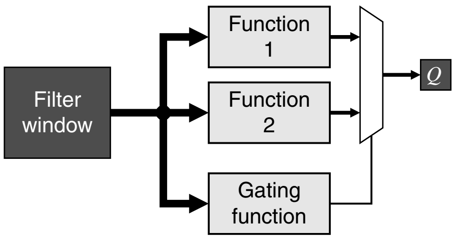


Figure 8.22 Gated filter.

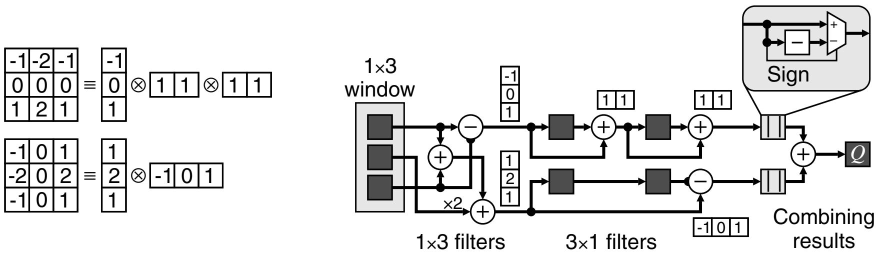


Figure 8.23 Sobel filter implemented as parallel separable filters. Left: filter decomposition; right: implementation using Equation 8.18.

If the edge orientation is also required, this may be conveniently calculated using a CORDIC unit (Section 5.4.3) to calculate both the arctangent and Equation 8.16. However if just the edge strength is required, Equation 8.16 is quite expensive and two simpler alternatives are commonly used (Abdou and Pratt, 1979):

$$Q = \max(|H|, |V|) \tag{8.17}$$

or

$$Q = |H| + |V| \tag{8.18}$$

One example of a Sobel filter implementation is given by Hezel *et al.* (2002); it directly implements the two linear filters and combines the result. However, a simpler implementation may be devised that reduces the number of calculations by exploiting separability. Both the horizontal and vertical filters are separable and may be decomposed, as shown in Figure 8.23. The two filters are combined using Equation 8.18 to reduce the complexity.

One limitation of differencing filters to detect gradients is that differentiating is a high pass operation and is, therefore, sensitive to noise. An alternative that has been proposed are moment-based filters (Suciu and Reeves, 1982), which are based on adding or integrating and are, therefore, inherently less sensitive to noise. The basic principle for edge detection is that the centre of gravity will be moved from the centre of the window in the presence of an edge. The offset of the centre of gravity from the window centre indicates the magnitude of the edge, while the direction is perpendicular to the edge orientation.

While there are many useful nonlinear filters, only a few of these are described in the following sections.

8.3.1 Edge Orientation

The orientation of edges, or intensity gradients, within an image may be determined by taking the derivatives in two perpendicular directions. If necessary, the effects of noise may be reduced by applying a Gaussian or other noise smoothing filter before detecting the gradients. The two derivatives can then be treated as the components of the gradient vector. The orientation of the gradient may then be determined from the angle of the vector by taking an arctangent. This process is illustrated in Figure 8.24.

In many situations, the output angle is required in terms of neighbouring pixels. Rather than perform the arctangent and then quantise the result, the orientation may be distinguished directly. Figure 8.25 shows some of the possibilities. To distinguish between a horizontal or vertical gradient, a threshold of 45° may be evaluated by comparing the magnitudes, as shown on the left. If necessary, the sign bits of the gradients may be used to distinguish between the four compass directions. The logic for this is most efficiently

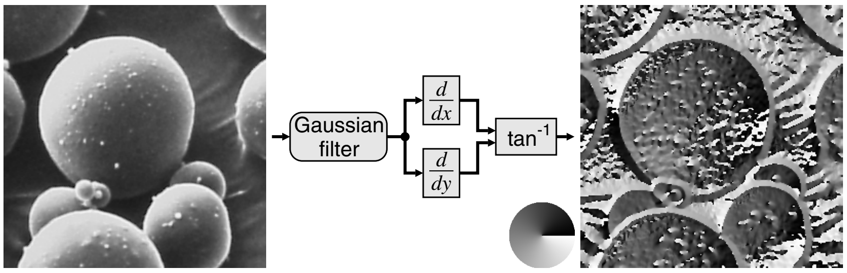


Figure 8.24 Edge orientation. The circle shows the mapping between angle and output pixel value.

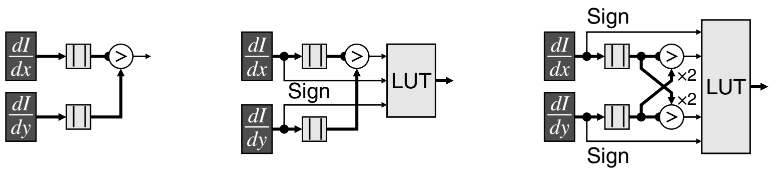


Figure 8.25 Circuits for detecting limited orientations. Left: distinguishing horizontal and vertical; centre: four directions (0°, ±90°, 180°); right: including diagonal directions (0°, 45°, 90°, 135°, or 0°, ±45°, ±90°, ±135°, 180°).

implemented using a lookup table to give the two bits out. If an eight-neighbourhood is required, the threshold angle is 22.5°. A sufficiently close approximation in most circumstances is:

$$Horizontal = \left|\frac{dI}{dx}\right| > 2\left|\frac{dI}{dy}\right| \tag{8.19}$$

which corresponds to a threshold of 26.6°. This will give a small preference to horizontal and vertical gradients over the diagonals. Again, the sign bits are able to resolve the particular direction and which quadrant the diagonals are in.

8.3.2 Non-maximal Suppression

The Canny edge detector (Canny, 1986) first smoothes the image, then detects the gradient, and suppresses points that are not a local maximum along the direction of the steepest gradient. Since the Gaussian smoothing will also spread the edge, the detector defines the position of the maximum gradient as the location of the edge. Non-maximal suppression is therefore thinning the edges, keeping only those points where the edge is the strongest.

Identifying the maximum gradient points first requires determining the orientation of the edge (from the direction of the steepest gradient). This may be determined as either horizontal or vertical or one of the two diagonals using the scheme described in the previous section. If the gradient magnitude at a point is larger than both points on either side in the direction of the gradient, then the magnitude is kept. Otherwise, it is not a local maximum and reset to zero. One possible circuit for implementing this is in Figure 8.26. An alternative for determining both the magnitude and orientation is a CORDIC block, although only two bits are required for the orientation. The orientation must be delayed by one row to compensate for the latency

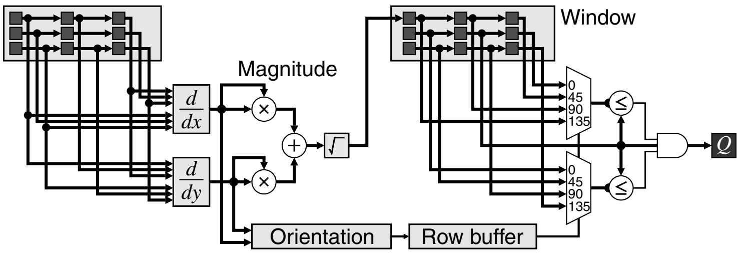


Figure 8.26 Non-maximal suppression for the Canny edge detector.

of getting the magnitude on the row below the current pixel. The orientation is used to select the two adjacent magnitudes for comparison. If either of the neighbours is greater, this masks the output.

8.3.3 *Zero-Crossing Detection*

A zero-crossing detector is required to detect the edge locations from the output of Laplacian of Gaussian or difference of Gaussian filters. Whenever two adjacent pixels have opposite sign, it must cross through zero somewhere in the space between the two pixels (only very occasionally does the zero-crossing occur precisely at a pixel location). The pixel on the positive side of the edge is arbitrarily defined as being the location of the zero crossing. Referring to Figure 8.27, if the centre pixel is positive and any of the four adjacent pixels is negative, then the centre pixel is a zero-crossing. Just using the sign bit also handles the case when the centre pixel is exactly zero.

The zero-crossings will form closed contours within a two-dimensional image. However, many of the detected crossings will correspond to insignificant or unimportant features. A cleaner edge image may be obtained by requiring the adjacent pixels to not only be of opposite sign, but also be significantly different. This thresholding is shown in the right panel of Figure 8.27.

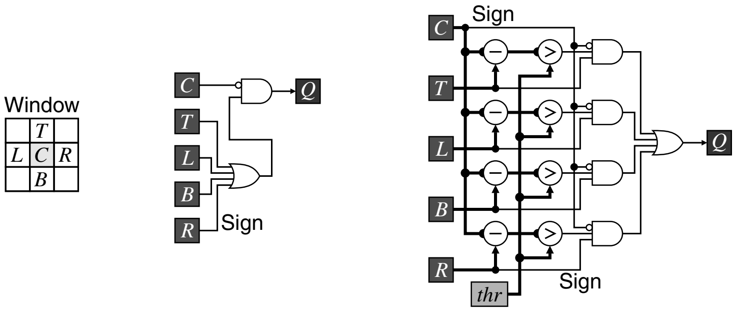


Figure 8.27 Zero-crossing detection filter. Left: definition of pixel locations within the window; centre: all zero crossings; right: those above a threshold.

8.4 Rank Filters

An important class of nonlinear filters is rank filters (Heygster, 1982; Hodgson *et al.*, 1985). These rank the pixel values within the window and use the value from a selected rank in the list, as illustrated in

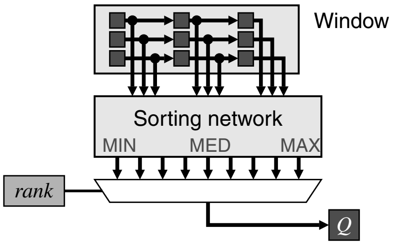


Figure 8.28 Rank filter.

Figure 8.28. They can be considered as a generalisation of the minimum, maximum (Nakagawa and Rosenfeld, 1978) and median filters. The main component of a rank filter is the sorting network that arranges the pixel values in ascending order. Obviously, if only a single, constant rank position is required, then only that sorted output needs to be produced by the sorting network, and the multiplexer is not required.

Rank filters, or combinations of rank filters, can be used in a wide range of image processing operations.

Using ranks close to the median will discard outliers giving them good noise smoothing properties, especially for heavy tailed distributions. This is demonstrated in Figure 8.29 by almost completely removing the salt and pepper noise from a heavily corrupted image. In terms of additive Gaussian noise,

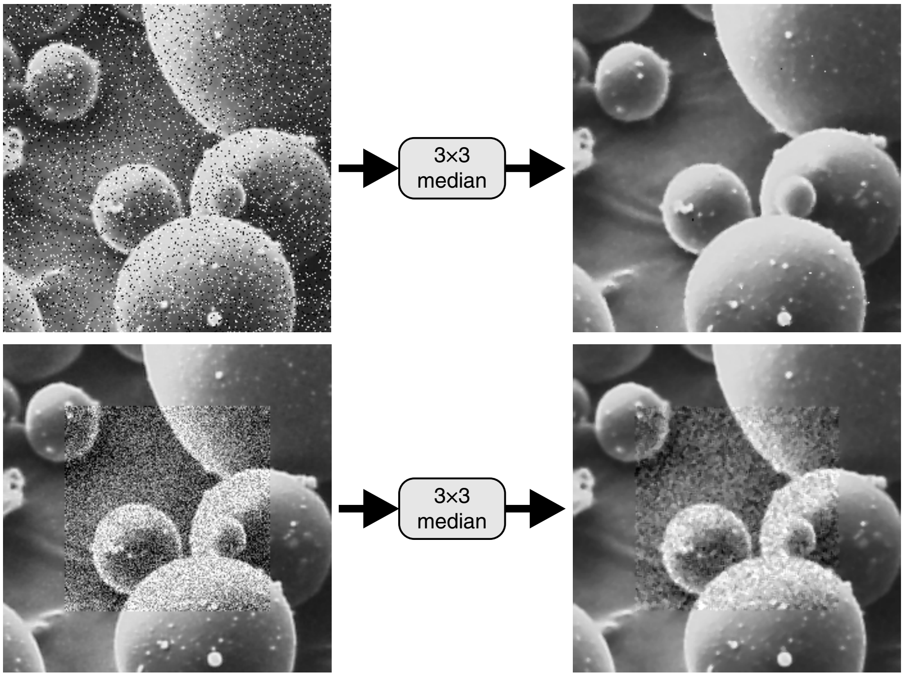


Figure 8.29 Median filter. Top: salt and pepper noise with 12% pixels corrupted; bottom: additive Gaussian noise.

the median is not as effective as an unweighted average of the same size. The best characteristics of both filters may be obtained by taking a weighted average of several ranks (Bovik *et al.*, 1983).

Straight edges are not affected by median filters, although if the edge is curved its position can shift, affecting the accuracy of any measurements made on the object (Davies, 1999). In the absence of noise, median filters do not blur edges (Nodes and Gallagher, 1982). However, any noise will cause edges to be slightly blurred, with the level of blurring dependent on the noise level (Justusson, 1981).

To detect edges, two rank values must be used and the difference between them is taken (Bailey and Hodgson, 1985), effectively using the range or subrange of pixel values within the window. The way this works as an edge detection filter is that each rank filter will shift the edges within the image by differing amounts, so the pixel difference will be larger in the vicinity of an edge. Generally a 3 × 3 window is used for edge detection; using a larger window will result in thick responses at the edges. The ranking of the pixels within the window means that the filter can detect edges of all orientations, and that the output of the filter is always positive. Using rank values closer to the extremes will give a stronger response, but is also more sensitive to noise. Rank values closer to the median are less noise sensitive, but will give a weaker response. These effects are clearly seen in Figure 8.30.

A simple gated rank filter can be used for both enhancing edges while simultaneously suppressing noise (Bailey, 1990). Consider a blurred edge as shown on the left in Figure 8.31. When the centre of the window is over the edge, there are several pixels on either side of the edge. These will be selected by rank values near the extremes. Let the pixel values selected by the rank filters to be representative of the dark and light sides of the edge be D and L respectively. An edge enhancement filter classifies the centre pixel as being either light or dark depending on which is closer:

$$Q = \begin{cases} D, & C-D < L-C \\ L, & \text{otherwise} \end{cases} \tag{8.20}$$

Selecting the extreme values for D and L will be more sensitive to noise and outliers, and will tend to give a slight over enhancement. Ringing is completely avoided because one of the pixel values within the

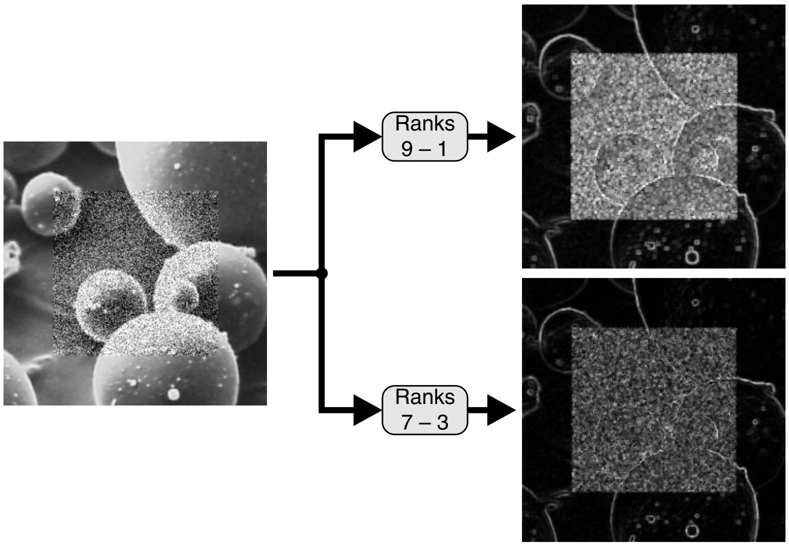


Figure 8.30 Subrange filters for edge detection.

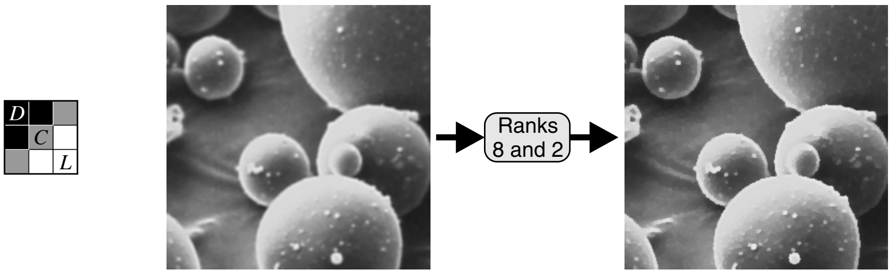


Figure 8.31 Rank-based edge enhancement filter. Left: pixels on either side of a blurred edge; right: edge enhancement using ranks 8 and 2.

window is always selected for output. Selecting rank values closer to the median will give some edge enhancement, while at the same time filtering noise.

The rank-based edge enhancement filter is particularly effective immediately prior to multilevel thresholding and can almost completely eliminate the problems described in Section 6.1.2. It also makes threshold selection easier because the classification process of Equation 8.20 depopulates the spread between the histogram peaks that results from pixels that are a mixture of two regions. The thresholding results are therefore less sensitive to the exact threshold level used. For best enhancement, the size of the window should be twice the width of the blur. Then whenever the window is centred over a blurred edge, there will be representatives of each side of the edge within the window to be selected.

Since rank filtering involves sorting the pixel values within the window, any monotonically increasing transformation of the pixel values will not change the order of the pixel values. Rank filters are therefore commutative with monotonic transformations. Note that the results of rank-based edge detection or edge enhancement will be affected, because the output is a function of multiple selected values, and the relative differences will in general be affected by the transformation.

8.4.1 Rank Filter Sorting Networks

Key to the implementation of rank filters is the sorting network that takes the pixel values within the window and sorts them into ranked order. A wide range of techniques has been developed for implementing this, both in software and in hardware.

Perhaps the simplest and most obvious approach is to use a simple bubble sorting odd–even transposition network, as shown in Figure 8.32. To sort a whole window at the clock rate, it is necessary to pipeline each stage in the network. This simple approach is only really suitable for small windows because the amount of logic grows with the square of the window area.

As with linear filters, running the filter with a higher clock rate (using a multiphase design) can reduce the hardware requirements. In this case it is complicated by the fact that the odd and even layers alternate. A two-phase scheme is shown in Figure 8.33. For sorting a 3×3 window, the number of layers has been reduced from nine to five. At the bottom, the data is looped back for a second pass through the network on the opposite phase. Note that since the bottom layer and the top layer are the same, phase 3b will not actually perform any sorting, but is required to phase shift the data fed back to interleave it with the new incoming data from the window.

An improvement may be gained by exploiting the fact that there is considerable overlap between successive window positions. Significant savings may be gained by sorting a column once as the new pixel comes into the window and then merging the sorted columns (Chakrabarti and Wang, 1994; Waltz *et al.*, 1998). One arrangement of this is shown in Figure 8.34. Note that for three inputs it is both faster and more efficient to perform the comparisons in parallel and use a single, larger multiplexer than it is to

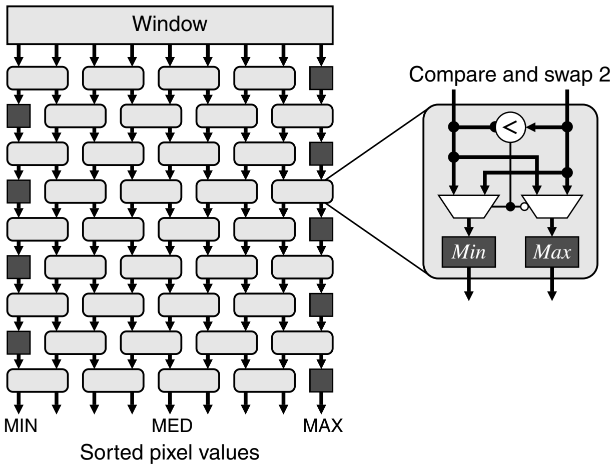


Figure 8.32 Bubble sorting using an odd–even transposition network.

use a series of three two-input compare and swap blocks. In the middle layer of compare and sorts, since the inputs are successively delayed with each window position, the middle comparator can also be removed and replaced with the delayed output of the first comparator. This reduces the number of comparators from 36 required for a bubble sort to 14. Again, this can be further reduced if not every rank value is required.

A similar analysis may be performed for larger window sizes. Although savings can be gained by sorting the columns only once, the logic requirements still increase approximately with the square of the window size.

Rather than sort the actual pixel values, each pixel in the window can have a corresponding rank register which counts the number of other pixels in the window greater than that pixel (Swenson and Dimond, 1999). Then, as new pixels are added to the window, it is necessary to compare the incoming pixels with

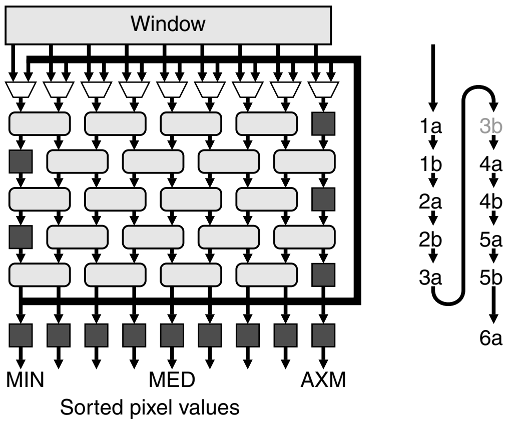


Figure 8.33 Two-phase rank sorting network. Right: phasing of the data as it passes through the network.

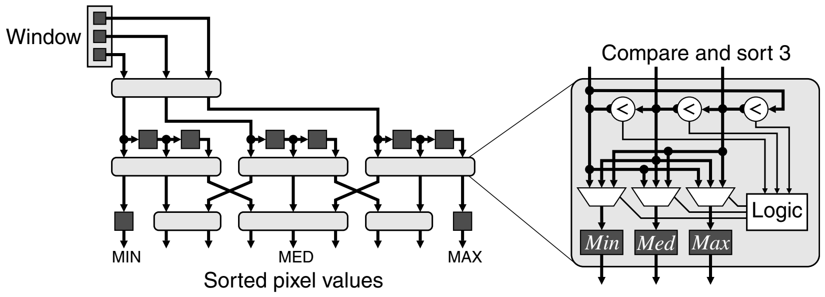


Figure 8.34 Sorting columns first, followed by a merging.

each of the existing window pixels to determine their rank and to adjust the ranks of the existing pixels. For one-dimensional filtering, a systolic array may be used to efficiently compare each pixel with every other pixel in the window and assign a rank value to each sample. One way of extending this to two dimensions is to collapse the two-dimensional window to one dimension by inserting the whole new column at each step (Hwang and Jong, 1990). The output is then taken every W samples after the complete window has been processed.

An alternative approach is to use the threshold decomposition property of rank filtering (Fitch *et al.*, 1984; 1985). Rank filtering is commutative with thresholding, so if the image is thresholded first, a binary rank filter may be used. Therefore, a grey-level rank filter may be implemented by decomposing the input using every possible threshold, performing a binary rank filter, and then reconstructing the output by adding the results. This process is illustrated in Figure 8.35.

A simple implementation of a binary rank filter is to count the number of ones within the window and compare this with the rank value, as shown in Figure 8.36. The counter here is parallel; it is effectively an adder network, adding the inputs to give the count. As with other implementations, it is possible to exploit the significant window overlap for successive pixels. An implementation of this is shown on the right in Figure 8.36. The column counts are maintained, with the new column added and the old column subtracted as the window moves. This reduces the size of the counter from W^2 to W inputs.

Threshold decomposition requires $V = 2^N - 1$ parallel binary rank filters. This is expensive for small windows, but becomes more efficient for larger window sizes. This principle of threshold decomposition may also be generalised to stack filters (Wendt *et al.*, 1986).

A closer look at the left panel of Figure 8.36 reveals that the count produced is actually the bin from cumulative histogram of the window corresponding to the threshold, *thr*. An early software approach to median filtering was based on maintaining the running histogram of the window, updating it as the data

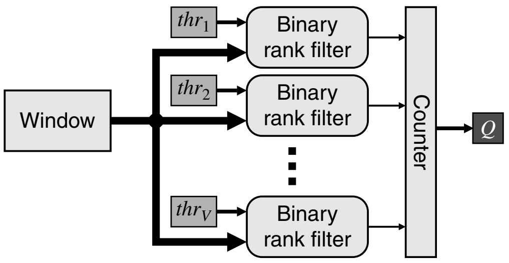


Figure 8.35 Threshold decomposition – the window values are thresholded to enable separate binary filters and reconstructed by combining the outputs.

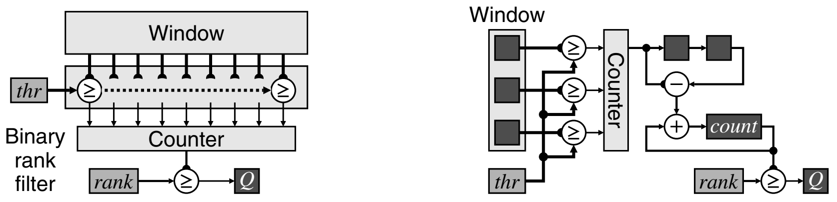


Figure 8.36 Binary rank filter for threshold decomposition. Left: thresholding to create binary values, and counting; right: an efficient implementation exploiting commonality between adjacent window positions.

moved into and out of the window (Garibotto and Lambarelli, 1979; Huang *et al.*, 1979). There are two problems with directly using a histogram. The first is that for two-dimensional windows, multiple bins must be updated simultaneously, although this can be overcome by using a trick similar to that shown in the right panel of Figure 8.36. The second problem is that the histogram must be searched to find the corresponding rank value. The search may take many clock cycles to find the appropriate output value. These problems may be overcome by using the cumulative histogram. This approach was used by Fahmy *et al.*, (2005) for one dimensional windows. An alternative to the direct output shown in Figure 8.35 is to use a binary search of the cumulative histogram (Harber and Neudeck, 1985). This is illustrated in Figure 8.37 for 3-bit data words. Note that in the right panel, the cumulative histogram counts are arranged in a bit-reversed order to minimise the interconnect length. The binary search can easily be extended to wider words, using pipelining if necessary to meet timing constraints. The binary search circuitry needs to be duplicated for each rank value required at the output.

This binary search can be extended to work from the original data using a bit voting approach. The idea is to reduce the logic by performing a binary sequence of threshold decompositions, with each decomposition yielding one bit of the output (Ataman *et al.*, 1980; Danielsson, 1981). The threshold decomposition is trivial, selecting the most significant bit of those remaining, while passing on the remaining bits to the next stage. If the most significant bit of the input differs from the desired rank output, that input pixel value can never be selected for the rank output. This is achieved by setting the remaining bits to zero or one accordingly to prevent them from being selected in subsequent stages. The circuit for this scheme is shown in Figure 8.38. Note that only a single rank value is output; to produce additional rank values, the sorting network must be duplicated.

A scheme similar to this has been implemented on an FPGA by Benkrid *et al.* (2002a); Benkrid and Crookes (2003) for median filtering and by Choo and Verma (2008) for rank filtering.

Minimum and maximum filters with rectangular windows are separable, providing an efficient implementation for large window sizes. A one-dimensional filter may be applied to either the rows or

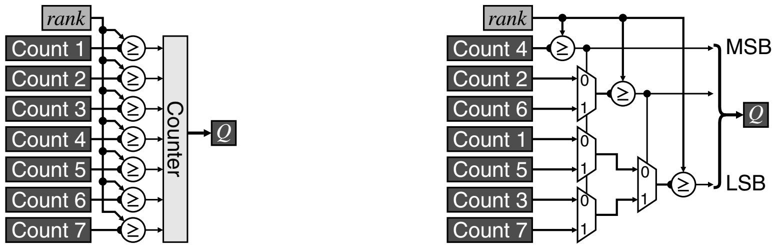


Figure 8.37 Searching the cumulative histogram (illustrated for 3-bit data). Left: parallel approach of Figure 8.35; right: binary search.

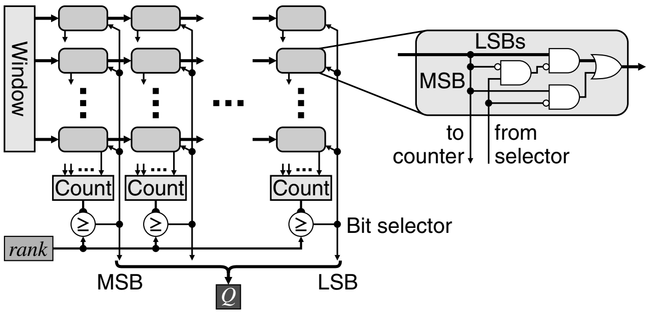


Figure 8.38 Bit sequential rank sorter.

columns first, followed by a second one-dimensional filter in the other direction. The overlap in window positions as the window moves enables the filter to be efficiently implemented using $\lceil \log_2 W \rceil$ operations (Bailey, 2010b). The basic principle is shown in Figure 8.39 for $W = 8$. To find the maximum requires $W-1$ two-input maximum operations. To minimise propagation delay, these may be arranged in a tree as shown in the top left panel. Since the inputs are sequential, the operations shown with a white background have already been calculated and can be buffered from before. The buffered version is shown in the top right panel. The circuit can readily be modified for window lengths that are not a power of two simply by reducing the number of delays in the last stage. The middle and bottom panels show a similar reuse transformation applied to the transposed filter structure (swapping input with output, and maximum operations with pick-off points). In the version which exploits reuse, the number of delays in the first stage can be reduced for window lengths that are not a power of two. Figure 8.39 illustrates a row filter; to implement a column filter, the sample delays are replaced by row buffers in the manner of Figure 8.18.

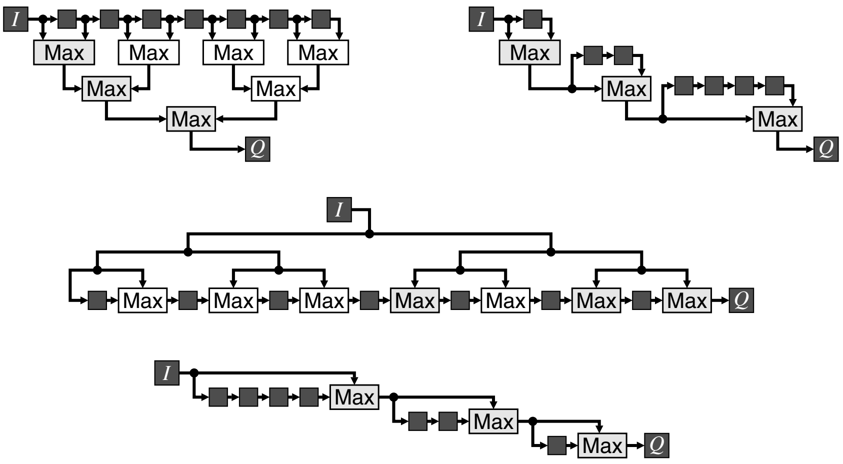


Figure 8.39 Reducing the number of operations in a maximum filter. Top left: a tree structure requires $W-1$ operations; top right: exploiting reuse reduces this to $\log_2 W$ operations; middle: transposed tree structure, requires $W-1$ operations; bottom: exploiting reuse in the transposed structure.

While the median and other rank filters are not separable, a separable median can be defined that applies a one-dimensional median filter first to the rows then to the columns of the result, or vice versa (Shamos, 1978; Narendra, 1981). Although this is not the same as the true two-dimensional median, it is a sufficiently close approximation in most image processing applications. It also gives a significant savings in terms of computational resources. Note that in general a different result will be obtained when performing the row or column median first.

8.4.2 Adaptive Histogram Equalisation

Closely related to rank filtering is adaptive histogram equalisation for contrast enhancement. Global histogram equalisation, as described in Section 7.1.2, is ineffective if there is a wide variation in background level, making the global histogram approximately uniform. Adaptive histogram equalisation overcomes this problem by performing the transformation locally, attempting to make the pixel values within a local window be equally likely. This expands the contrast locally, while making the image more uniform globally.

Adaptive histogram equalisation is therefore a filter that determines the histogram equalisation mapping within a window, with only the centre pixel value being transformed. Each pixel is subject to a separate mapping based on local context (Hummel, 1977). Since each transformation is given by the scaled cumulative histogram within the window, the output is effectively the scaled rank position of central pixel value within the window (Pizer *et al.*, 1987).

What makes adaptive histogram equalisation computationally expensive is that it generally uses a large window. To overcome this, many software implementations divide the image into a series of overlapping blocks and either use a single transformation for all of the pixels within each block or interpolate the transformations between blocks and apply the interpolated transformation to each pixel (Pizer *et al.*, 1987).

Rather than build the histogram or cumulative histogram from scratch at each window position, it is possible to use an approach similar to that for box filtering to exploit adjacency and to update the histogram based on the changes. This approach was taken by Kokufuta and Maruyama (2009) and is represented in Figure 8.40. The basic approach is to maintain column histograms within the window. As the window moves to the next pixel, the histogram for the new column is updated and added to the window histogram. The column that has just left the window is subtracted from the window histogram. The column

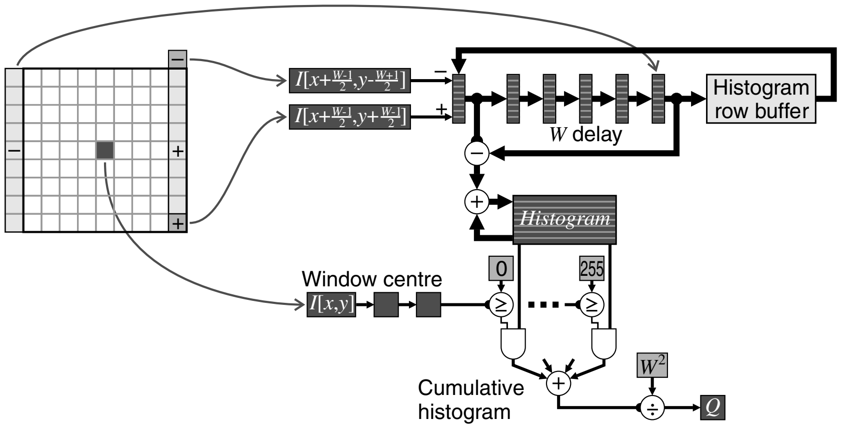


Figure 8.40 Adaptive histogram equalisation by maintaining a running histogram.

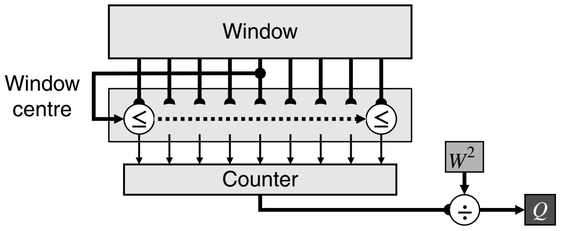


Figure 8.41 Adaptive histogram equalisation through window thresholding.

histograms must be cached in a row buffer until the next line. Since the histograms must be added and subtracted in parallel, they must be stored in registers, although a block RAM may be used for the histogram row buffer. Shallower fabric RAM or shift registers can be used to provide the W pixel delay unless W is large enough to make efficient use of a block RAM. The two delays in the window centre account for the latency of the histogram accumulation. The window histogram entries from bins less than the centre are summed to give the corresponding cumulative histogram bin, which is normalised to give the output pixel value.

The resources required by this approach are independent of the window size. However, significant memory resources are required for the histogram row buffer and for registers for the column and window histograms. This approach is, therefore, suitable for large window sizes, although it does require a larger modern FPGA to provide the required on-chip memory.

For smaller window sizes, an alternative is to modify the threshold decomposition rank filter of Figure 8.36 to calculate the cumulative histogram bin for the current centre pixel value. This approach is demonstrated in Figure 8.41. Unfortunately, it is not possible to exploit the window overlap when moving from one pixel to the next because the threshold level varies with each pixel.

8.5 Colour Filters

Filtering colour images carries its own set of problems. Linear filters may be applied independently to each component of the image.

Problems can be encountered when filtering hue images, because the hue wraps around. For example, the average of 12° and 352° is not 182° obtained by direct numerical averaging, but 2°. If the dominant hue is red (close to 0°), the hues may be offset by 180° before filtering and adjusted back again afterwards. The alternative is to convert from hue and saturation polar coordinates to rectangular coordinates before filtering and convert back to polar after filtering.

When using nonlinear filters, it is important to keep the colour components together. If a trimmed filter removes one component, all components should be trimmed. A gated filter should apply the same gating to all channels. Applying a nonlinear filter independently to each component can lead to colour fringing if the different components are treated differently.

Rank filtering, in particular, is not defined for colour images, because vectors do not have a natural ordering. This problem was discussed in more detail in Section 7.2.2. The median is, however, defined for colour data (Astola *et al.*, 1988; 1990) and can be used for noise smoothing. One problem is the search through all of the points to find the median. For filtering, this must be performed within a pixel clock cycle. To do this efficiently, it is necessary to reuse the large proportion of data that is common from one window position to the next. Even so, efficient search algorithms, such as proposed by Barni (1997), are difficult to map to a hardware implementation. For small windows, the median may be calculated directly, although for larger windows the separable median may be more appropriate.

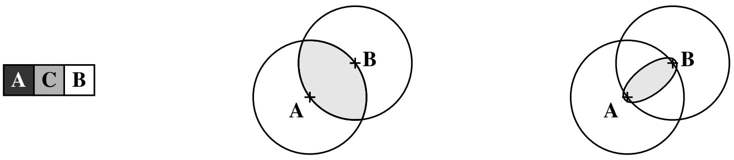


Figure 8.42 Constructs for colour edge enhancement. Left: three pixels, with **C** on the border between regions **A** and **B**; centre: point **C** would be the vector median in the shaded region; right: a tighter criterion defined by Equation 8.25.

Many variations of the vector median filter have been proposed in the literature. The simplest is to interleave the bits from each of the components and perform a scalar median filter (Chanussot *et al.*, 1999) with the wider data word. A wide range of gated vector medians has been proposed (Celebi and Aslandogan, 2008) that remove noise without losing too many fine details.

The rank-based edge enhancement filter can readily be adapted to colour images. The key is to identify when a pixel is on a blurred edge and then identify which of the two regions the edge pixel is most associated with. Consider three pixels across a boundary between two different coloured regions, as illustrated on the left in Figure 8.42. If point **C** is on a blurred edge between regions **A** and **B**, the colour will be a mixture of the colours of **A** and **B**:

$$\mathbf{C} = \alpha\mathbf{A} + (1-\alpha)\mathbf{B} \tag{8.21}$$

In this case, since **C** is between **A** and **B**, the central pixel **C** will be the vector median of the three points. In this case, from Equation 7.65:

$$||\mathbf{C}-\mathbf{A}|| + ||\mathbf{C}-\mathbf{B}|| < \begin{cases} ||\mathbf{A}-\mathbf{B}|| + ||\mathbf{A}-\mathbf{C}|| \\ ||\mathbf{B}-\mathbf{A}|| + ||\mathbf{B}-\mathbf{C}|| \end{cases} \tag{8.22}$$

or

$$\left.\begin{matrix} ||\mathbf{C}-\mathbf{B}|| \\ ||\mathbf{C}-\mathbf{A}|| \end{matrix}\right\} < ||\mathbf{A}-\mathbf{B}|| \tag{8.23}$$

If this is used as the criterion for detecting an edge pixel as proposed in (Tang *et al.*, 1994), a point **C** anywhere in the shaded region in the centre panel of Figure 8.42 would be considered an edge pixel and enhanced. In three dimensions, this is a discus-shaped region, which can include points that deviate considerably from Equation 8.21. If **C** is indeed a mixture of the two colours, then:

$$||\mathbf{C}-\mathbf{A}|| + ||\mathbf{C}-\mathbf{B}|| \approx ||\mathbf{A}-\mathbf{B}|| \tag{8.24}$$

An alternative edge criterion, allowing for some small deviation from the line as a result of noise, is (Gribbon *et al.*, 2004):

$$||\mathbf{C}-\mathbf{A}|| + ||\mathbf{C}-\mathbf{B}|| < ||\mathbf{A}-\mathbf{B}|| + T \tag{8.25}$$

which is an ellipsoidal region, illustrated by the shaded region in the right panel of Figure 8.42. T can either be fixed, or it may be proportional to the distance between **A** and **B**. In Figure 8.42, T is $||\mathbf{A}-\mathbf{B}||/8$, which can easily be calculated using a simple shift.

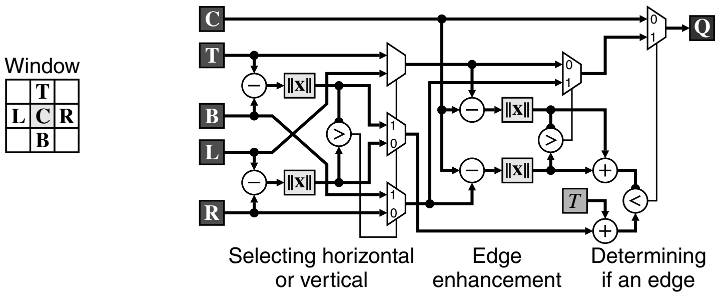


Figure 8.43 Colour edge enhancement.

If **C** is an edge pixel, as determined by either Equation 8.23 or Equation 8.25, then the edge pixel may be enhanced by selecting the closest:

$$\mathbf{Q} = \begin{cases} \mathbf{A}, & ||\mathbf{C}-\mathbf{A}|| < ||\mathbf{C}-\mathbf{B}|| \\ \mathbf{B}, & ||\mathbf{C}-\mathbf{A}|| \geq ||\mathbf{C}-\mathbf{B}|| \end{cases} \tag{8.26}$$

otherwise it is left unchanged.

In extending to two dimensions, using a larger two-dimensional window presents a problem because of the difficulty in determining which two pixels in the window are the opposite sides of the edge (the **A** and **B** above). One alternative is to apply the one-dimensional enhancement filter to the rows and columns (Tang *et al.*, 1994). An alternative is to check if the edge is primarily horizontal or vertical and perform the enhancement perpendicular to the edge (Gribbon *et al.*, 2004). This approach is shown in Figure 8.43 using the edge detection criterion of Equation 8.25.

The distances may be measured using either the L_1 or L_2 norms. The L_1 norm has the advantage that it is simple to calculate, although it is not isotropic. The L_2 norm is isotropic, although for colours it requires three multiplications, two sums and a square root. The alternative is to use CORDIC arithmetic, although two CORDIC units are required for a three-component colour space. Since the norms do not need to be exact, an approximation to the L_2 norm may be obtained by taking a linear combination of the ordered components (Barni *et al.*, 2000). For three components, the relative weighting of the three components that gives the minimum error is (Barni *et al.*, 2000) $1 : (\sqrt{2}-1) : (\sqrt{3}-\sqrt{2})$. A simple approximation that is reasonably close is $1 : 0.5 : 0.25$, with the corresponding calculation represented in Figure 8.44.

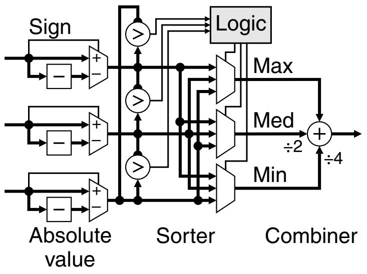


Figure 8.44 Simplified approximation of Euclidean distance (L_2 norm).

If the edge blur is larger, then the same filter can be extended but considering points two or three pixels apart (Tang *et al.*, 1994). If noise is an issue, the filter may be preceded by an appropriate noise smoothing filter (Tang *et al.*, 1994).

For other operations, a common strategy is to apply the enhancement or filtering to the luminance component of the image (the *Y* from YUV or YCbCr, the *V* from HSV, or *L** from L*a*b* or L*u*v*). When doing so, it is important to keep the components together through any selection process to avoid colour fringing artefacts.

8.6 Morphological Filters

Based on set theory, mathematical morphology defines a set of transformations on a set based on a structuring element (Serra, 1986). In terms of image processing filters, the set can be defined by the pixel values within the image and the structuring element may be thought of as an arbitrary window. As implied by the name, morphological filters will filter an input image on the basis of the shape or morphology of objects within an image. The structuring element defines the shape of the filter and effectively defines or controls what is being filtered.

The basic principles of morphological filtering are easiest understood in terms of binary images. These principles can then be extended to filtering greyscale images.

8.6.1 Binary Morphology

A binary image considers each pixel to belong to one of two classes: object and background. In general the object pixels are represented by a binary 1 and the background is represented by 0. The structuring element, **S**, consists of a set of vectors or offsets. It can therefore be considered as another binary image or shape.

The basic operations of morphological filtering are erosion and dilation. With *erosion*, an object pixel is kept only if the structuring element fits completely within the object. Considering the structuring element as a window, the output is considered an object pixel only if all of the inputs are one; erosion is therefore a logical AND of the pixels within the window:

$$\begin{aligned} Q_{erosion}[x,y] &= I \ominus S \\ &= \wedge_{i,j\in \mathbf{S}}\, I[x+i,y+j] \end{aligned} \tag{8.27}$$

It is called an erosion, because the object size becomes smaller as a result of the processing.

With *dilation*, each input pixel is replaced by the shape of the structuring element within the output image. This is equivalent to outputting an object pixel if the flipped structuring element hits an object pixel in the input. In other words, the output is considered an object pixel if any of the inputs within the flipped window is a one, that is dilation is a logical OR of the flipped window pixels:

$$\begin{aligned} Q_{dilation}[x,y] &= I \oplus S \\ &= \vee_{i,j\in \mathbf{S}}\, I[x-i,y-j] \end{aligned} \tag{8.28}$$

Note that the flipping of the window (rotation by 180° in two dimensions) will only affect asymmetric windows. In contrast with erosion, dilation causes the object to expand and the background to become smaller. Erosion and dilation are duals, in that a dilation of the image is equivalent to an erosion of the background and vice versa. This can also be seen in Equations 8.27 and 8.28 by taking the logical complement of both sides of the equation and using De Morgan's theorem.

Many other morphological operations may be defined in terms of erosion and dilation. Two of the most commonly used morphological filters are opening and closing. An ***opening*** is defined as an erosion followed by a dilation with the same structuring element.

$$I \circ S = (I \ominus S) \oplus S \tag{8.29}$$

The erosion will remove features smaller than the size of the structuring element and the dilation will restore the remaining objects back to their former size. Therefore, the remaining object points are only where the structuring element fits completely within the object.

The converse of this is a *closing*, which is a dilation followed by an erosion with the same structuring element.

$$I \bullet S = (I \oplus S) \ominus S \tag{8.30}$$

Background regions are kept only if the structuring element completely fits within them. It is called a closing because any holes or gaps within the object smaller than the structuring element are closed as a result of the processing. Example morphological operations are illustrated in Figure 8.45.

The relatively simple processing and modest storage required by morphological filters has made them one of the most commonly implemented image processing filters, especially in the early days when FPGAs were quite small and the available resources limited what could be accomplished.

The direct implementation of erosion and dilation are relatively trivial, as seen in Figure 8.46. Duality enables both to be implemented with a single circuit, with a control signal used to complement the input and output.

More interesting, however, is to make the structuring element programmable (Velten and Kummert, 2002). Associated with each window element is an additional register which controls whether or not that window element is part of the structuring element. It effectively gates the window data by providing a one from those window elements that are not part of the structuring element. The circuit of Figure 8.46 is modified to give Figure 8.47. The programme registers may be set either as part of the initialisation or programmed on-the-fly by another process.

Opening and closing may be implemented as a pipeline of erosion followed by dilation, or dilation followed by erosion respectively. However, it is possible to combine both processes within a single window structure. Without loss of generality, consider the opening. By definition, if the complete

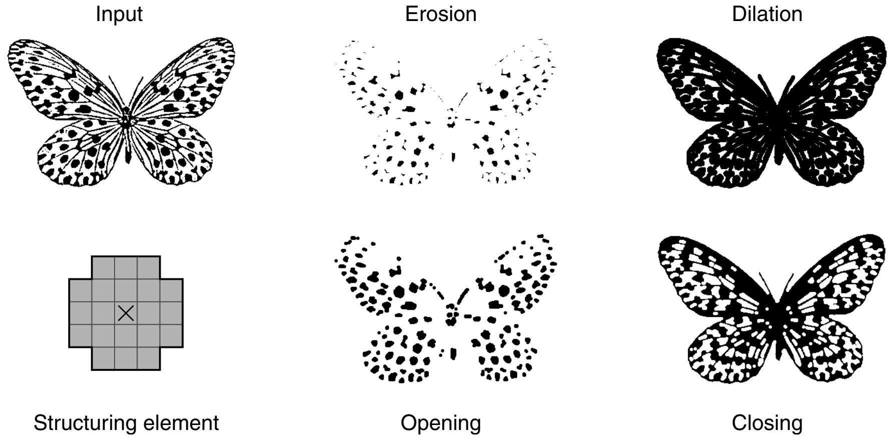


Figure 8.45 Morphological filtering. In this example, black pixels are considered object, with a white background. (Photo courtesy of Robyn Bailey.)

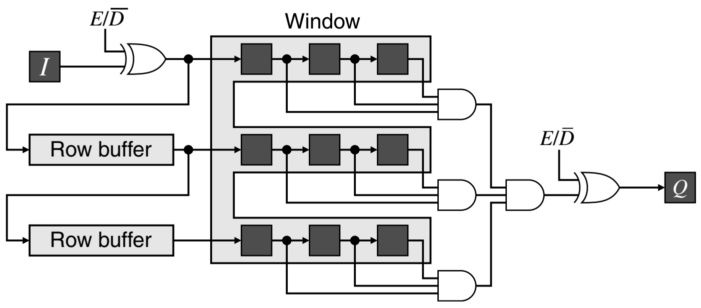


Figure 8.46 Circuit for erosion and dilation. The control signal selects between erosion and dilation.

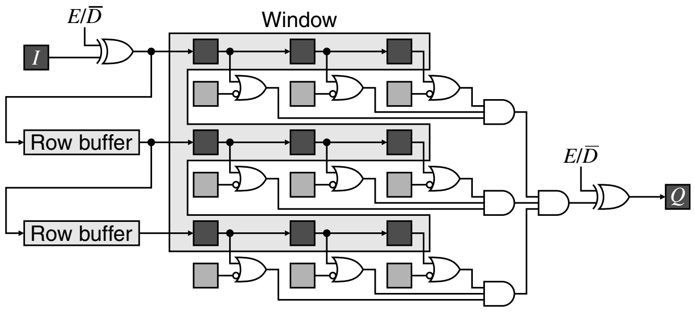


Figure 8.47 Morphological filtering with a programmable structuring element.

structuring element fits within the object, all of those corresponding object pixels will be output. Therefore, by feeding back the erosion output, a parallel set of window registers may be used to hold the opening output. These must be shifted along because any of the window positions may set the output from the opening operation. This scheme is shown in Figure 8.48. Note that the row buffers have been changed to the serial configuration because the output must be shifted serially. The output multiplexer selects between erosion or dilation and opening or closing.

Although a single window structure is used, this filter has the same latency and row buffer memory requirements as a cascade of two filters. The advantage is that, for a programmable window, the structure element control bits are easier to share than if the two window filters were operated separately.

Just as linear filters could be decomposed as a sequence of simpler smaller filters, morphological filters can also be decomposed. There are two types of decomposition of particular interest here. The first is that dilation is associative, allowing a complex structuring element to be made up of a sequence of simpler structuring elements:

$$(I \oplus S_1) \oplus S_2 = I \oplus (S_1 \oplus S_2) \tag{8.31}$$

with a similar chain rule for a sequence of erosions:

$$(I \ominus S_1) \ominus S_2 = I \ominus (S_1 \oplus S_2) \tag{8.32}$$

The most obvious application of the chain rule is to make the operations for rectangular structuring elements separable, with independent filters in each of the horizontal and vertical directions.

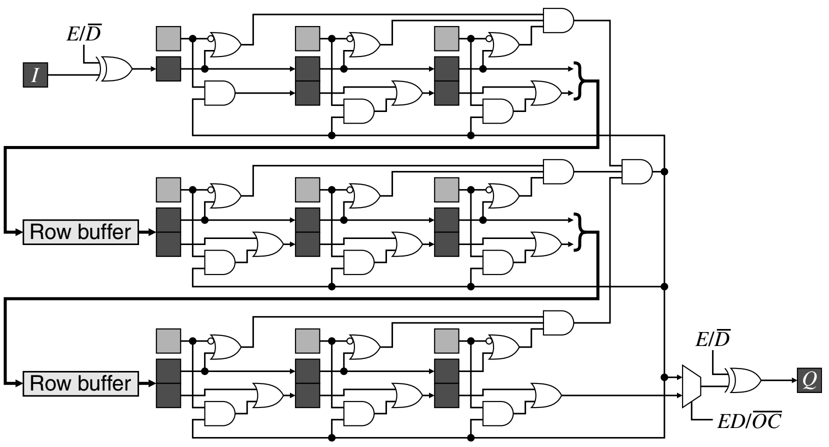


Figure 8.48 Extending the window operations to perform opening or closing as well as erosion or dilation using a programmable structuring element.

The second decomposition is based on combining filters in parallel. For dilations:

$$I \oplus (S_1 \vee S_2) = (I \oplus S_1) \vee (I \oplus S_2) \tag{8.33}$$

and for erosions:

$$I \ominus (S_1 \vee S_2) = (I \ominus S_1) \wedge (I \ominus S_2) \tag{8.34}$$

These decompositions, for example, allow the structuring element in Figure 8.45 to be decomposed as in Figure 8.49. Note that with decompositions, the integrated approach of performing the opening or closing demonstrated in Figure 8.48 cannot be used.

An alternative approach is to implement the filter directly as a finite state machine (Waltz, 1994c). Firstly, consider a one-dimensional erosion filter. The pattern of 1s and 0s in the window can be considered the state, as this information must be maintained to perform the filtering. If the structuring element is continuous, a useful representation of the state is the count of successive 1s within the window. The output is a 1 only when all of the inputs within the window are 1s. Therefore, whenever a 0 is encountered, the count is reset to zero, and when the count reaches the window width, it saturates. An example state transition diagram and corresponding implementation are given in Figure 8.50.

This approach may be extended to arbitrary patterns by maintaining a more complex state machine. Basically the state is a compressed representation of the elements within the window. The finite state machine approach can also be extended to two dimensions by creating two state machines, one for

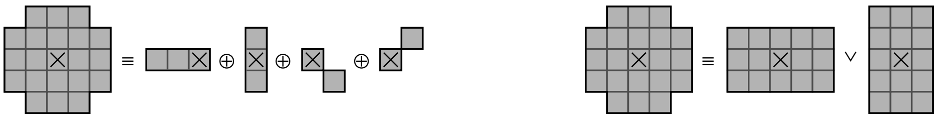


Figure 8.49 Example structuring element decompositions. Left: series decomposition using dilation; right: parallel decomposition using OR.

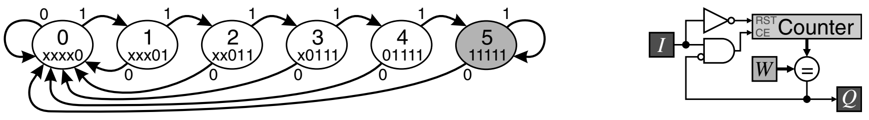
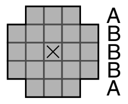


Figure 8.50 Horizontal erosion based on a state machine implementation. The shaded state outputs a 1, the other states output a 0. *W* is the size of the one-dimensional window.

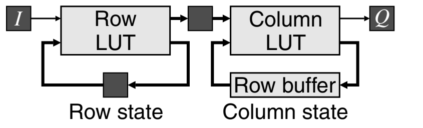
operating on the rows and the other for operating on the columns (Waltz, 1994c; Waltz and Garnaoui, 1994a). It is not necessary for the structuring element to be separable in the normal sense of separability; the row state machine passes pattern information representing the current state of the row to the column state machine. This is illustrated for a non-separable structuring element in Figure 8.51.

Firstly, the required patterns on each of the rows of the structuring element are identified. The row state machine is then designed to identify the row patterns in parallel as the data shifts in. Combinations of the row patterns are then encoded in the output to the column state machine. The column state machine then identifies the correct sequence of row patterns as each column shifts in. In this example, the row machine has only seven states and the column machine six states. The finite state machines may therefore be implemented using small lookup tables.

The technique may be extended to filters using several different structuring elements in parallel. The corresponding state machines are coded to detect all of the row patterns for all of the filters in parallel, and similarly detect the outputs corresponding to all of the different row pattern combinations in parallel.

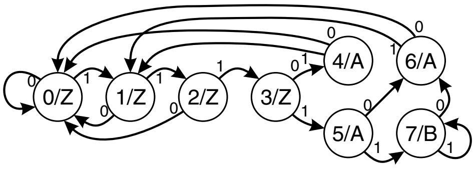


Row patterns
Z: xxxxx
A: x111x
B: 11111



Row state machine

		Next		
State	Pattern	0	1	Output
0	xxxx0	0	1	Z
1	xxx01	0	2	Z
2	xx011	0	3	Z
3	x0111	4	5	Z
4	01110	0	1	A
5	01111	6	7	A
6	11110	0	1	A
7	11111	6	7	B



Column state machine

		Next			
State	Pattern	Z	A	B	Output
0	xxxxZ	0	1	1	0
1	xxxZA	0	1	2	0
2	xxZAB	0	1	3	0
3	xZABB	0	1	4	0
4	ZABBB	0	5	5	0
5	ABBBA	0	1	2	1
6	ABBBB	0	5	6	1

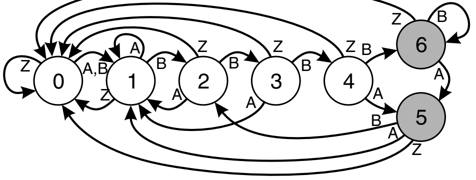


Figure 8.51 Separable finite state machines for erosion with a non-separable window.

The extension to dilation is trivial; the pattern is made up of 0s rather than 1s. It may also be extended to binary template matching (Waltz, 1994a), where the row patterns consist of a mixture of 0s and 1s (and don't cares). Binary correlation is implemented in a similar manner (Waltz, 1995), although it counts the number of pixels that match the template. The state machines for correlation are more complex because they must also account for mismatched patterns with appropriate scores.

8.6.2 Greyscale Morphology

Binary morphology requires that the image be thresholded before filtering. The concepts of binary morphology may be extended to greyscale images by using threshold decomposition. Reconstructing the greyscale image leads to selecting the minimum pixel value within the structuring element for erosion and the maximum pixel value for dilation. An alternative viewpoint is to treat the pixel value as a fuzzy representation between 0 and 1, and use fuzzy AND and OR in Equations 8.27 and 8.28 (Goetcherian, 1980). Either way, the resulting representations are:

$$I \ominus S = \min_{i,j \in \mathbf{S}} \{I[x+i, y+j]\} \tag{8.35}$$

and

$$I \oplus S = \max_{i,j \in \mathbf{S}} \{I[x-i, y-j]\} \tag{8.36}$$

with opening and closing defined as combinations of erosion and dilation as before.

Examples of greyscale morphological operations are illustrated in Figure 8.52. Opening removes lighter blobs from the image that are smaller than the structuring element, whereas closing removes darker blobs.

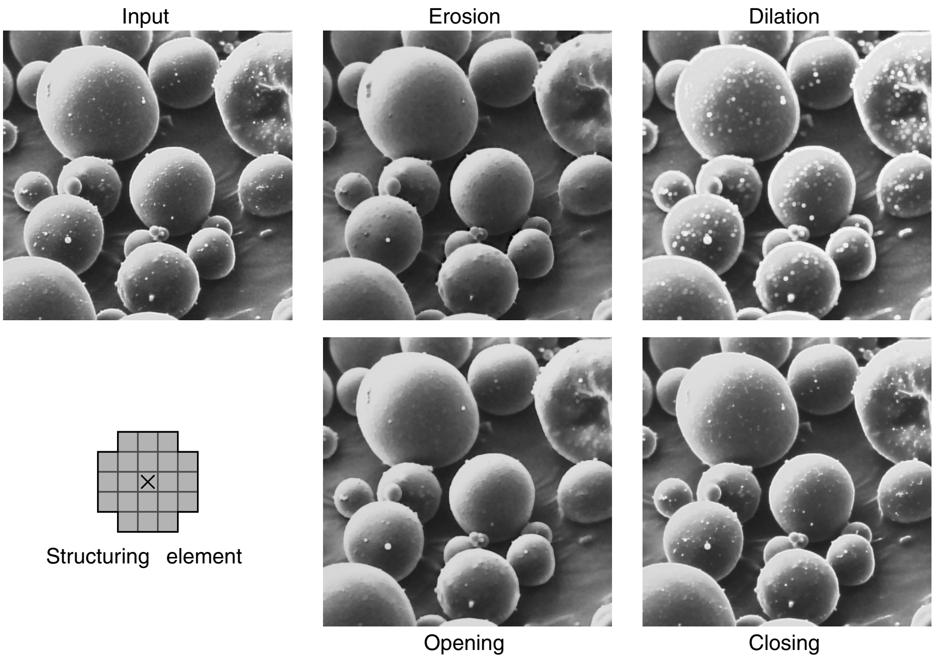


Figure 8.52 Greyscale morphological filtering.

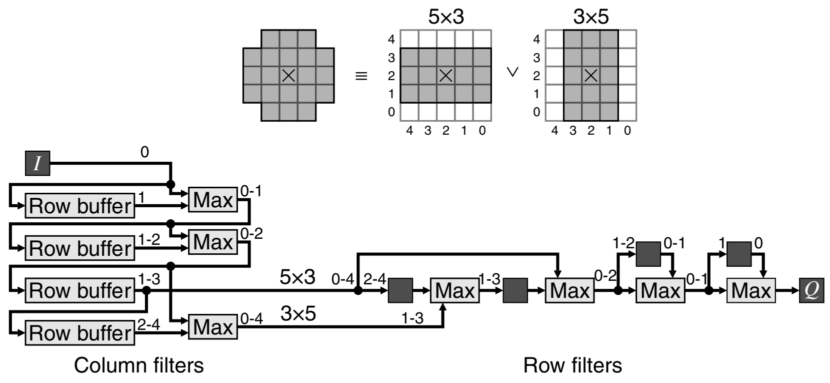


Figure 8.53 Efficient circular window implementation. Top: OR decomposition of the structuring element; bottom: the separable implementation of both components. A transposed filter structure is used for the row filter. The numbers with the column filter represent the number of rows delay from the input. The numbers in the row filter represent the number of pixel delays to the output.

Any of the decompositions described for binary morphological filters also apply to their greyscale counterparts. The finite state machine implementation may also be extended to greyscale images (Waltz, 1994b), although significantly more state information is required to represent the greyscale pixel values. For rectangular structuring elements, efficient separable implementations as illustrated in Figure 8.39 are applicable.

For circular structuring elements, the structuring element may be decomposed using OR decomposition, with the corresponding rectangular windows implemented separably (Bailey, 2010b). Figure 8.53 demonstrates this for the 5×5 circular structuring element. The row buffers and delays are reused for both filters by merging the two filters. This requires that a transpose structure be used for the row filter. If necessary, additional pipeline registers may be added to reduce the delay through a chain of maximum operations, although on modern FPGAs the maximum processing rate is more likely to be limited by the timing for the row buffers.

Equations 8.35 and 8.36 represent morphological filters with a binary structuring element. They may be further extended to use greyscale structuring elements:

$$I \ominus S = \min_{i,j \in \mathbf{S}} \{I[x+i, y+j] - S[i,j]\} \tag{8.37}$$

and

$$I \oplus S = \max_{i,j \in \mathbf{S}} \{I[x-i, y-j] + S[i,j]\} \tag{8.38}$$

The introduction of the extra component makes efficient implementation more difficult, because it is difficult to reuse results of previous calculations in all but special cases of the structuring element.

8.6.3 Colour Morphology

Extending morphological filters to colour images is more complex, because the equivalents of minimum and maximum do not exist for vectors. Simply applying the corresponding greyscale filters to each channel can give colour fringing where the relative contrast in different channels is reversed (for example on a border between red and green). The alternative (Comer and Delp, 1999) is to define a vector to scalar transformation that may be used for sorting the pixel values to enable a minimum or maximum pixel to be selected.

8.7 Adaptive Thresholding

Adaptive thresholding is a form of nonlinear filter where the input is a greyscale image and output is binary, usually object and background. It is used when a single global threshold is unable to adequately separate the objects within the image from the background. There are two ways of looking at the operation. The first considers adaptive thresholding as a filter which calculates a suitable threshold based on the local context. The other is to consider the filter as a preprocessing step that adjusts the image enabling a global threshold be effective.

Since the goal of adaptive thresholding is to distinguish between object and background, the threshold level should be somewhere between the two distributions. The simplest estimate, therefore, is an average of the pixel values within the window, provided the window is larger than the object size. (A Gaussian filter gives a smoother threshold, although the unweighted box average is simpler to calculate.) This approach works best around the edges of the object, where thresholding is most critical, but for larger objects or empty background regions may result in misclassifications. These may be reduced by offsetting the average. This is illustrated in Figure 8.54 for the image of Figure 6.10 where global thresholding did not work. The delay in the main path is to account for the latency of the Gaussian filter.

Another approach is to use morphological filtering to estimate the background level, which can then be subtracted from the image enabling a global threshold to be used. This approach is demonstrated in Figure 8.55.

8.7.1 *Error Diffusion*

When producing images for human output, one problem of thresholding is a loss of detail. A similar problem occurs when quantising an image to a few grey levels, where the spatial correlation of the

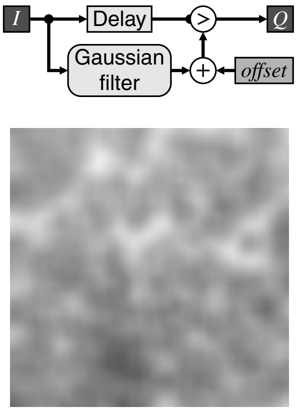


Figure 8.54 Adaptive thresholding. Top: implementation; left: input image; centre: threshold level calculated using a Gaussian filter with $\sigma = 8$ and *offset* $= 16$; right: thresholded image.

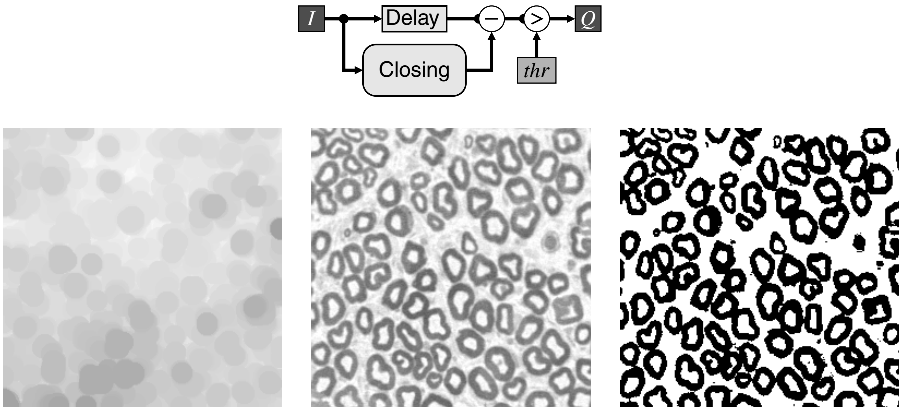


Figure 8.55 Adaptive thresholding. Top: implementation; left: estimated background after using a morphological closing with a 19 × 19 circular structuring element to remove the objects; centre: after removing the background; right: thresholded image.

quantisation error results in contouring artefacts. What is desired is for the local average level to be maintained, while reducing the number of quantisation levels.

Floyd and Steinberg (1975) devised such an algorithm for displaying greyscale images on a binary display. The basic principle is that, after thresholding or quantisation, the quantisation error (the difference between the input and the output) is propagated on to the neighbouring four pixels that have not yet been processed. A possible implementation of this is shown in Figure 8.56. A row buffer holds the accumulated error for the next row, where it is added to the incoming pixel. The small integer multiplications can be implemented with a shift and add.

The threshold is normally set at mid-grey. The thresholding is then equivalent to selecting the most significant bit of the input (and repeating it to give the equivalent output grey level as either black or white). The threshold level is actually arbitrary (Knuth, 1987); regardless of the threshold level, the errors will average out to zero, giving the desired average grey level on the output. Modulating the threshold level (making it dependent on the input) can be used to enhance edges (Eschbach and Knox, 1991), with the recommended level given as:

$$thr = \overline{I[x, y]} \tag{8.39}$$

Although described here for a binary output, the same principles can be applied when any number of quantisation levels is used, by propagating the quantisation error to reduce contouring effects.

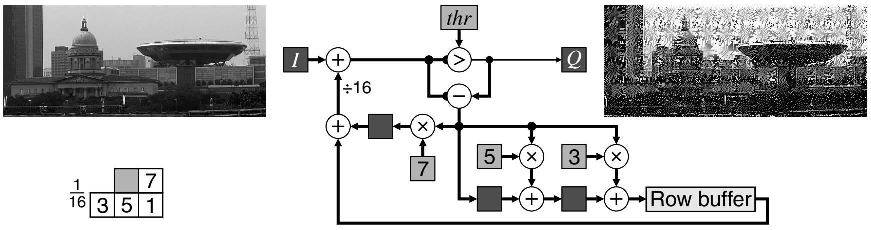


Figure 8.56 Binary error diffusion. (Photo courtesy of Robyn Bailey.)

When applied to colour images, error diffusion may be applied to each component independently. With a binary output for each component, the outputs are in general uncorrelated between the channels. Threshold modulation may be used to either force correlation or anti-correlation between the channels (Bailey, 1997a). A simple modulation based on intensity (sum of RGB components) is sufficient to force resynchronisation. If the intensity is closer to black, the threshold level in all three channels can be increased by about 15% to encourage all three channels to simultaneously produce a 0 output. Similarly, if the intensity is closer to white, the threshold can be decreased to encourage all three channels to produce a 1. If the image is coloured, there will be a different number of 0s and 1s in each channel, but they will tend to cluster together as a result of the threshold modulation. To force anti-correlation, the opposite modulation can be used.

8.8 Summary

Local filters extend point operations by making the output depend not only in the corresponding input pixel value, but also its local context (a window into the input image surrounding the corresponding input pixel). To implement filters efficiently on an FPGA, it is necessary to cache the input values as they are loaded so that each pixel is only loaded once. The regular access pattern of filters enables the cache to be implemented with relatively simple row buffers, built from the block RAMs within the FPGA. When combined with pipelining of the filter function, such caching allows one pixel to be processed every clock cycle, making local filters ideal for stream processing.

Linear filters are arguably the most widely used class of local filters. The output of a linear filter is a linear combination of the input pixels within the window. Linearity enables such filters also to be considered in terms of their effect on different spatial frequencies; this aspect is explored more fully in Chapter 10.

One of the disadvantages of linear filters is their limited ability to distinguish between signals of interest and noise. A wide range of nonlinear filters has been developed to address this problem, of which only a small selection has been reviewed in this chapter. Two important classes of nonlinear filters considered in some detail are rank filters and morphological filters. A range of efficient structures for implementing these filters has been described in some detail.

Filtering colour images poses its own problems. Linear filters can be applied separately to each component, but for nonlinear filters it is important to treat the colour vector as a single entity. This is particularly so for rank and morphological filters, where edges within the image can be shifted as a result of filtering. To prevent colour artefacts, the edges within all components must be moved similarly.

Local filters are an essential part of any image processing application. Therefore, the range of techniques and principles described in this chapter is indispensible for accelerating any embedded image processing application.

9

Geometric Transformations

The next class of operations to be considered is that of geometric transformations. These redefine the geometric arrangement of the pixels within an image (Wolberg, 1990). Examples of geometric transformations include zooming, rotating and perspective transformation. Such transformations are typically used to correct spatial distortions resulting from the imaging process, or to normalise an image by registering it to a predefined coordinate system (for example, registering an aerial image to a particular map projection or warping a stereo pair of images so that the rows corresponds to epipolar lines).

Unlike point operations and local filters, the output pixel does not, in general, come from the same input pixel location. This means that some form of buffering is required to manage the delays resulting from the changed geometry. The simplest approach is to hold the input image or the output image (or both) in a frame buffer. Most geometric transformations cannot easily be implemented with streaming for both the input and output.

The mapping between input and output pixels may be defined in two different ways, as illustrated in Figure 9.1. The forward mapping defines the output pixel coordinates, (u, v), as a function, m_f, of the input coordinates:

$$Q[u,v] = Q\left[m_{fu}(x,y), m_{fv}(x,y)\right] = I[x,y] \tag{9.1}$$

The forward mapping is suitable for processing a streamed input, for example from a camera, because it specifies for each input pixel, where the pixel value goes to in the output image. Therefore, it is able to process sequential addresses in the input image. Note that as a result of the mapping, sequential input addresses do not necessarily correspond to sequential output addresses.

Conversely, the reverse mapping defines the input pixel coordinates as a function, m_r, of the output coordinates:

$$Q[u,v] = I\left[m_{rx}(u,v), m_{ry}(u,v)\right] = I[x,y] \tag{9.2}$$

It is, therefore, more suited for producing a streamed output, for example images being streamed to a display, because for each output pixel the reverse mapping specifies where the pixel value comes from in the input image. It can therefore process sequential output addresses, but will in general require non-sequential access of the input image.

The basic architectures for implementing forward and reverse mappings are compared in Figure 9.2. For streamed access of the input (forward mapping) or output (reverse mapping), the address will be provided by appropriate counters synchronised with the corresponding stream. While the input to the

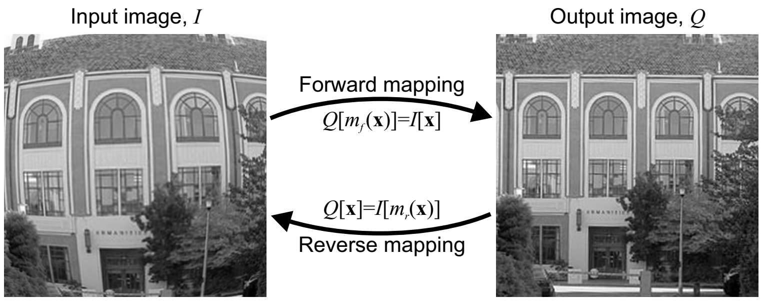

Input image, I
Output image, Q
Forward mapping
$Q[m_f(\mathbf{x})]=I[\mathbf{x}]$
$Q[\mathbf{x}]=I[m_r(\mathbf{x})]$
Reverse mapping

Figure 9.1 Forward and reverse mappings for geometric transformation.

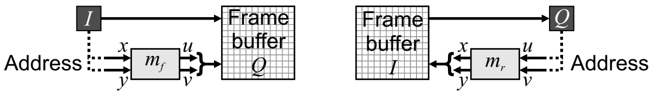


Figure 9.2 Basic architectures for geometric transformation. Left: forward mapping; right: reverse mapping.

mappings will usually be integer image coordinates, the mapped addresses will not, in general, be integer locations. Consequently, the implementation will be more complex than that implied by Figure 9.2. The main issues associated with the forward and reverse mappings are different, so are discussed here in more detail separately.

9.1 Forward Mapping

With a forward mapping, simply rounding the output coordinates to the nearest pixel results in two problems. Firstly, if the transform magnification is greater than one (zooming in), there can be holes in the output image, where no input pixels map to. Such output pixels will not be assigned a value. Secondly, if the magnification is less than one (zooming out), several input pixels can map onto a single output pixel. If the input pixel value is simply written to the frame buffer, then the value may be overwritten by subsequent pixels. Even if the magnification is exactly one, if there is a rotation then both problems can occur, as demonstrated in the left panel of Figure 9.3.

To overcome both problems, it is necessary to map the whole rectangular region associated with each input pixel to the output image. Each output pixel is then given by the weighted sum of all the input pixels which overlap it. The weight associated with each pixel is given by the proportion of the output pixel that is contributed to by the corresponding input pixel:

$$Q[u,v] = \sum_{x,y} w_{x,y,u,v} I[x,y] \tag{9.3}$$

where

$$w_{x,y,u,v} = \int Q[u,v] \cap I[x,y] du dv \tag{9.4}$$

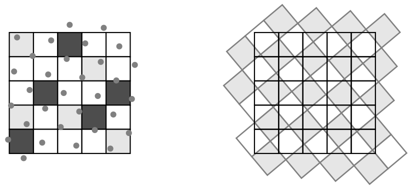


Figure 9.3 Problems with forward mapping. Left: the input pixels (dots) are rotated anti-clockwise by 40 degrees. Output pixels shaded dark have no input pixels mapping to them. Output pixels shaded light are mapped to by two input pixels. Right: it is necessary to map the whole pixel to avoid such problems.

For this, it is easier to map the corners of the pixel, because these define the edges, which in turn define the region occupied by the pixel. On the output, it is necessary to maintain an accumulator image to which fractions of each input pixel are added as they are streamed in. There are two difficulties with implementing this approach on an FPGA for a streamed input. Firstly, each input pixel can intersect many output pixels, especially if the magnification of the transform is greater than one. Therefore, many (depending on the position of the pixel and the magnification) clock cycles are required to adjust all of the output pixels affected by each input pixel. Since each output pixel may be contributed to by several input pixels, a complex caching arrangement is required to overcome the associated memory bandwidth bottleneck. A second problem is that, even for relatively simple mappings, determining the intersection as required by Equation 9.4 is both complex and time consuming.

9.1.1 Separable Mapping

Both of these problems may be significantly simplified by separating the transformation into two stages or passes (Catmull and Smith, 1980). The key is to operate on each dimension separately:

$$Q[u,v] = Q\left[u, m_{fv}(u,y)\right] = T\left[m_{fu}(x,y), y\right] = I[x,y] \tag{9.5}$$

The first pass through the image transforms each row independently to get each pixel on that row into the correct column. This forms the temporary intermediate image, $T[u,y]$. The second pass then operates on each column independently, getting each pixel into the correct row. This process is illustrated with an example in Figure 9.4. The result of the two transformations is that each pixel ends up in the desired output location. Note that the transformation for the second pass is in terms of u and y rather than x and y to reflect the results of the first pass.

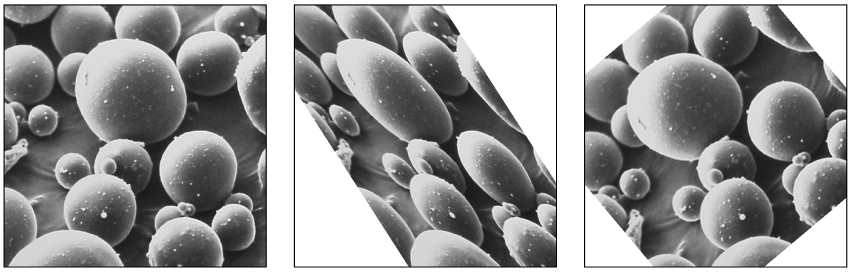


Figure 9.4 Two-pass transformation to rotate an image by 40 degrees. Left: input image; centre: intermediate image after transforming each row; right: transforming each column.

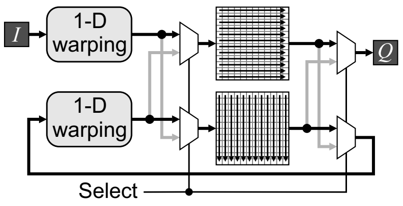


Figure 9.5 Implementation of two-pass warping with forward mapping using bank switching.

Reducing the transformation from two dimensions to one dimension significantly reduces the difficulty in finding the pixel boundaries. In one dimension, each pixel has only two boundaries: with the previous pixel and with the next pixel. The boundary with the previous pixel can be cached from the previous input pixel, so only one boundary needs to be calculated for each new pixel.

Both the input and output from each pass are produced in sequential order, enabling stream processing to be used. Although the second pass scans down the columns of the image, stream processing can still be used. However, a frame buffer is required between the two passes to hold the intermediate image. If the input is streamed from memory, and the bandwidth enables, the image can be processed in place, with the output written back into the same frame buffer. Bank switching is still required to process every frame, with an implementation is shown in Figure 9.5. One frame buffer bank is used for each pass, with the banks switched each frame to maintain the throughput. The processing latency is a little over two frame periods.

Since the same algorithm is applied to both the rows and columns (apart from calculating the mapping), a single one-dimensional processor may be used for both passes, as shown in Figure 9.6. This halves the resource requirements at the expense of only processing every second input frame.

Since each row and column is processed independently, the algorithm may be accelerated by processing multiple rows and columns in parallel. The main limitation on the number of rows that may be processed in parallel is the bandwidth available to the frame buffer.

Before looking at the implementation of the one-dimensional warping, the limitations of the two-pass approach will be discussed. Firstly, consider rotating an image by 90 degrees – all of the pixels on a row in the input will end up in the same column in the output. Therefore, the first pass maps all of the points on a row to the same point in the intermediate image. The second pass is then supposed to map each of these points onto a separate row in the output image, but is unable to because all the data has been collapsed to a point. This is called the bottleneck problem and occurs where the local rotation is close to 90 degrees. Even for other angles, there is a reduction in resolution as a result of the compression of data in the first pass, followed by an expansion in the second pass (Figure 9.4 shows an example – the data in the intermediate image occupies a smaller area than either the input or output images). The worst cases may be overcome by rotating the image 90 degrees first, and then performing the mapping. This is equivalent to treating the incoming row-wise scanned data as a column scan and performing the column mapping first. This works fine for global rotations. However, many warps may locally have a range of rotation angles, in which case some parts of the image should be rotated first and others should not. Wolberg and Boult (1989) overcame this by performing both operations in parallel and selected the output locally based on which had the least

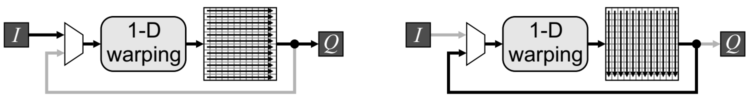


Figure 9.6 Two-pass forward mapping. Left: in the first pass each row is warped and written to a frame buffer, while the previous image is streamed out; right: in the second pass, each column is processed and written back to the frame buffer.

compression in the intermediate image. This solution comes at the cost of doubling the hardware required (in particular the number of frame buffers needed).

An alternative algorithm that avoids the bottleneck problem when performing rotations is to use three passes rather than two, with each pass a shear transformation (Wolberg, 1990). While requiring an extra pass through the image, it has the advantage that each pass merely shifts the pixels along a row or column, resulting in a computationally very simple implementation for pure rotations.

A second problem occurs if the mappings on each row or column in the first pass are not either monotonically increasing or decreasing. As a result, the mapping is many-to-one, with data from one part of the image folding over that which has already been calculated. (Such fold-over can occur even if the complete mapping does not fold over; the second pass would map the different folds to different rows.) In the original proposal (Catmull and Smith, 1980), it was suggested that such fold-over data from the first pass be held in additional frame buffers, so that it is available for unfolding in the second pass. Without such additional resources, the range of geometric transformations that may be performed using the two-pass approach is reduced.

A third difficulty with the two-pass forward mapping is that the second transformation is in terms of u and y. Since the original mapping is in terms of x and y, it is necessary to be able to invert the first mapping in order to derive the second mapping. Consider, for example, an affine transformation:

$$\begin{aligned} u &= m_{fu}(x,y) = a_{ux}x + a_{uy}y + a_u \\ v &= m_{fv}(x,y) = a_{vx}x + a_{vy}y + a_v \end{aligned} \tag{9.6}$$

The first pass is straight forward, giving u in terms of x and y. However, v is also specified in terms of x and y rather than u and y. Inverting the first mapping gives:

$$x = \frac{u - a_{uy}y - a_u}{a_{ux}} \tag{9.7}$$

and then substituting into the second part of Equation 9.6 gives the required second mapping:

$$v = m_{fv}(u,y) = \frac{a_{vx}}{a_{ux}}u + \left(a_{vy} - \frac{a_{uy}a_{vx}}{a_{ux}}\right)y + \left(a_v - \frac{a_u a_{vx}}{a_{ux}}\right) \tag{9.8}$$

While inverting the first mapping in this case is straightforward, for many mappings it can be difficult (or even impossible) to derive the mapping for the second pass and, even if it can be derived, it may be computationally expensive to perform. Wolberg and Boult (1989) overcame this by saving the original x in a second frame buffer, enabling $m_{fv}(x,y)$ from Equation 9.6 to be used. They also took the process one step further and represented the transformation through a pair of lookup tables rather than directly evaluating the mapping function.

The final issue is that the two-pass separable warp is not identical to the two-dimensional warp. This is because each one-dimensional transformation is of an infinitely thin line within the image. However, this line actually consists of a row or column of pixels with a thickness of one pixel. Since each row and column is warped independently, significant differences in the transformations between adjacent rows can lead to jagged edges (Wolberg and Boult, 1989). As long as adjacent rows (and columns) are mapped to within one pixel of one another, this is not a significant problem.

Subject to these limitations, the two-pass separable approach provides an effective approach to implement the forward mapping. Each pass requires the implementation of a one-dimensional geometric transformation. While the input data can simply be interpolated to give the output, this can result in aliasing where the magnification is less than one. To reduce aliasing, it is necessary to filter the input; in general this filter will be spatially variant, defined by the magnification at each point.

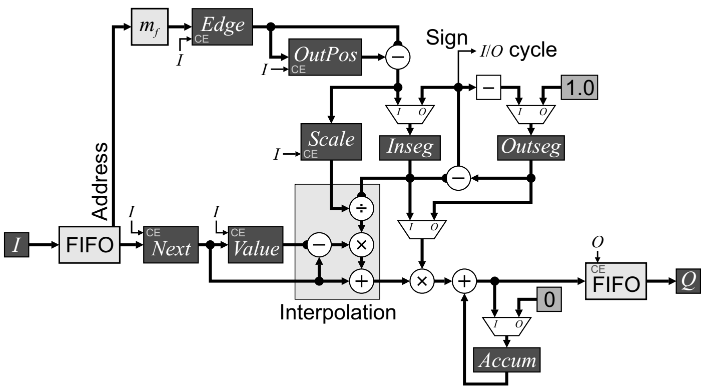


Figure 9.7 One-dimensional warping using resampling interpolation.

The filtering, interpolation and resampling can all be performed in a single operation (Fant, 1986). Fant's original algorithm used a reverse map; it was adapted by Wolberg *et al.* (2000) to use both the forward and reverse map, with the forward map presented in Figure 9.7. The basic principle is to maintain two fractional valued pointers into the stream of output pixels. *Inseg* indicates how many output pixels that the current input pixel has yet to cover, while *Outseg* indicates how much of the current output pixel remains to be filled. If *Outseg* is less than or equal to *Inseg*, an output pixel can be completed so an output cycle is triggered. The input value is weighted by *Outseg*, combined with the output value accumulated so far in *Accum*, and is output. The accumulator is reset, *Outseg* reset to one and *Inseg* decremented by *Outseg* to reflect the remaining input to be used. Otherwise *Inseg* is smaller, so the current input is insufficient to complete the output pixel. This triggers an input cycle, which accumulates the remainder of the current input pixel (weighted by the fraction of output pixel it covers), subtracts *Inseg* from *Outseg* to reflect the fraction of the output pixel remaining, and reads a new input pixel value. As each input pixel is loaded, the address is passed to the forward mapping function to get the corresponding pixel boundary in the output. The mapping may either be implemented directly or via a lookup table (Wolberg and Boult, 1989). The magnification or scale factor for the current pixel is obtained by subtracting the previous pixel boundary, *OutPos*, from the new boundary. This result is used to initialise *Inseg*.

The interpolation block is required when the magnification is greater than one, to prevent a single pixel from simply being replicated in the output, resulting in large flat areas and contouring (Fant, 1986). It performs a linear interpolation between the current input value and the next pixel value in proportion to how much of the input pixel is remaining.

Each clock cycle is either an input cycle or an output cycle, although in general these do not alternate. (If the magnification is greater than one, then there will be multiple output pixels for each input pixel, and if the magnification is less than one, then there will be multiple input pixels for each output pixel.) A FIFO buffer is therefore required on both the input and output to smooth the flow of data to allow streaming with one pixel per clock cycle. Although the one-dimensional warping has an average latency of one row, it requires two row times to complete. Therefore, to run at pixel clock rates, two such warping units must be operated in parallel.

Handling of the image boundaries is not explicitly shown in Figure 9.7. The initial and trailing edge of the input can be detected from the mapping function. If the first pixel is already within the image, a series of empty pixels need to be output. Similarly, at the end of the input row, a series of empty pixels may also be required to complete the output row.

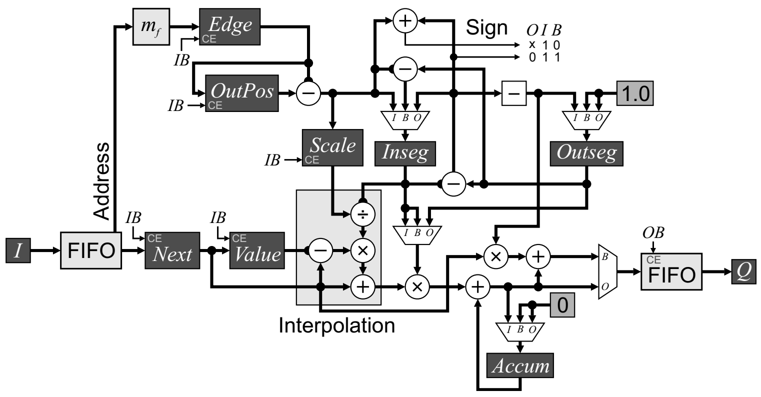


Figure 9.8 Extending Figure 9.7 to combine the input and output cycle (labelled *B* for both) if possible.

The speed can be significantly improved by combining the input and output cycles if the input pixel is sufficiently wide to also complete an output pixel (Wolberg *et al.*, 2000). This will be the case if:

$$B = (Inseg < Outseg) \wedge (Edge - Outpos + Inseg > Outseg) \tag{9.9}$$

For the combined cycle, the clock is enabled for both (*B*) the input and output circuits. The extra logic to complete this is shown in Figure 9.8. The main change is rather than add the remainder of the old input pixel to an accumulator, the sum is added to the fraction of the new pixel required to complete the output, with the result sent directly to the FIFO.

While many mappings will still take longer than one data row length, the extra time required is small in most cases. Where the magnification is greater than one, a pixel will be output every clock cycle, with an input cycle when necessary to provide more data. The converse will be true where the magnification is less than one. Assuming that the input and output are the same size, then extra clock cycles are only required when the magnification is greater than one on some parts of the row, and less than one on other parts. If the image has horizontal blanking at the end of each row, this would be sufficient to handle most mappings in a single row time.

Since the output pixel value is a sum over a fraction of pixels, Evemy *et al.* (1990) took a different approach to accelerate the mapping by completely separating the input and output cycles. As the row is input, the image is integrated to create a sum table. This enables any output pixel to be calculated by taking the difference between the two interpolated entries into the sum table corresponding to the edges of the output pixel. After the complete row has been loaded, the output phase begins. The reverse mapping (along the row) is used to determine the position of the edge of the output pixel in the sum table, with the subpixel position determined through interpolation (Figure 5.31). The difference between successive input sums gives the integral from the input corresponding to the output pixel. This is normalised by the width of the output pixel to give the output pixel value. The structure for this is shown in Figure 9.9.

As both input and output take exactly one clock cycle per pixel, regardless of the magnification, this approach can be readily streamed. By bank switching the summed row buffers, a second row is loaded while the first row is being transformed and output. The latency of this scheme is, therefore, one row length (plus any pipeline latencies). Since the block RAMs on FPGAs are dual-port, a single memory with length of two rows can be used, with the bank switching controlled by the most significant address line.

Rather than perform the two passes independently, as implied at the start of this section, the two passes can be combined together (Bailey and Bouganis, 2009a). In the second pass, rather than process each

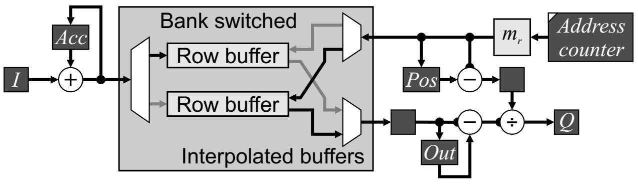


Figure 9.9 Separating the input and output phases using sum tables, and interpolating on the output.

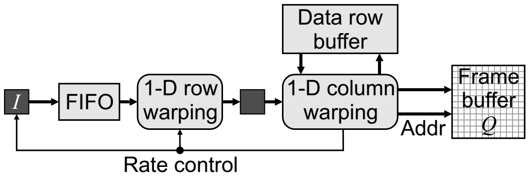


Figure 9.10 Combining the row and column warps by implementing the column warps in parallel.

column separately, they can be processed in parallel, with row buffers used to hold the data. The basic idea is illustrated in Figure 9.10.

The circuit for the first pass row warping can be any of those described above. For the second pass, using sum tables would require an image buffer because the complete column needs to be input before output begins, gaining very little. However, with the interleaved input and output methods, each time a pixel is output from the first pass, it may be passed to the appropriate column in the second pass. Any output pixel from the second pass may be saved into the corresponding location in the output frame buffer. If the magnification is less than one, then at most one pixel will be written every clock cycle. Note that, in general, pixels will be written to the frame buffer non-sequentially because the mappings for each column will be different.

If the magnification is greater than one, then multiple clock cycles may be required to ready the column for the next input pixel before moving onto the next column. This will require that the column warping controls the rate of the row warping. If the input FIFO is not blocking, then it will also control the rate of the input stream.

To summarise, a forward mapping allows data to be streamed from an input. To save having to maintain an accumulator image, with consequent bandwidth issues, the mapping can be separated into two passes. The first pass operates on each row and warps each pixel into the correct column, whereas the second pass operates on each column, getting each pixel into the correct row.

9.2 Reverse Mapping

In contrast, the reverse mapping determines for each output pixel where it comes from in the input image. This makes the reverse mapping well suited for producing a streamed output for display or passing to downstream operations. Reverse mapping is the most similar to software, where the output image is scanned and the output pixel obtained from the mapped point in the input image.

There are two factors which complicate this process. The first is the fact that the mapped coordinates do not necessarily fall on the input pixel grid. This requires some form of interpolation to calculate the pixel value to output. Interpolation methods are described in more detail in the next section. The second factor is that the when the magnification is less than one, it is necessary to filter the input image to reduce the effects of aliasing. If the magnification factor is constant (or approximately constant) then a simple filter

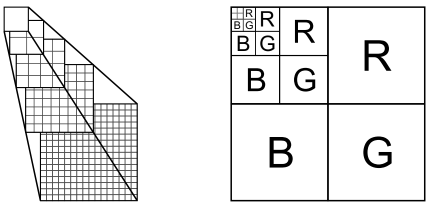


Figure 9.11 Pyramidal data structures. Left: multiresolution pyramid; right: mip-map organisation for storing colour images (Williams, 1983).

may be used. However, if the magnification varies significantly with position (for example with a perspective transformation) then the filter must be spatially variant, with potentially large windows in some places in the image. Filtering the input image with a large window is also inefficient, because a conventional filter will produce outputs corresponding to every input pixel position. The small magnification means that most of the calculated outputs will be discarded anyway.

As there is insufficient time to perform the filtering while producing the output, it is necessary to prefilter the image before the geometric transformation. There are two main approaches to filtering that allow for a range of magnifications. One is to use a pyramidal data structure with a series of filtered images at a range of resolutions; the other is to use summed area tables.

An *image pyramid* is a multiresolution data structure used to represent an image at a range of scales. Each level of the pyramid is formed by successively filtering and down-sampling the layer below it by a factor of two, as illustrated in Figure 9.11. Typically, each pixel in the pyramid is the average of the four pixels below it. A pyramid can be constructed in a single pass through the image and requires 33% more storage than the original image. A convenient memory organisation for a colour image is a *mip-map* (Williams, 1983), which divides the image memory into four quadrants, one for each of red, green, and blue, and the fourth quadrant for the remainder of the pyramid at the higher levels as shown on the right in Figure 9.11.

The pyramid can be used for filtering for geometric transformation in several ways. The simplest is to determine which level corresponds to the desired level of filtering based on the magnification, and select the nearest pixel within that level (Wolberg, 1990). The limitation of this is that the filter window is considered to be a square, with size and position determined by pyramid structure. This may be improved by interpolating within the pyramid (Williams, 1983) in the x, y, and scale directions (tri-linear interpolation). Such interpolation requires reading eight locations in the pyramid, requiring innovative mapping and caching techniques to achieve in a single clock cycle.

An alternative approach is to use *summed area tables* (Crow, 1984). A summed area image is formed in which each pixel contains the sum of all of the pixels above and to the left of it:

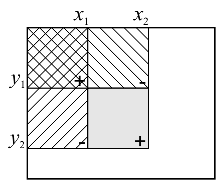
$$S[x,y] = \sum_{i\le x}\sum_{j\le y} I[i,j] \tag{9.10}$$

The sum over any arbitrary rectangular region may then be calculated in constant time (with four accesses to the summed area table, as shown in Figure 9.12) regardless of the size and position of the rectangular region:

$$\sum_{i=x_1}^{x_2}\sum_{j=y_1}^{y_2} I[i,j] = S[x_2,y_2]-S[x_1-1,y_2]-S[x_2,y_1-1]+S[x_1-1,y_1-1] \tag{9.11}$$

Summed area tables can be created recursively by inverting Equation 9.11:

$$S[x,y] = S[x-1,y]+S[x,y-1]-S[x-1,y-1]+I[x,y] \tag{9.12}$$



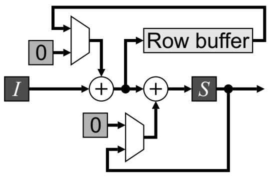


Figure 9.12 Left: using a summed area table to find the sum of pixel values within an arbitrary rectangular area; right: efficient streamed implementation to create a summed area table.

This requires a row buffer to hold the sums from the previous row. A more efficient approach, however, is to maintain a column sum and add this to the sum:

$$\begin{aligned} C[x,y] &= C[x,y-1] + I[x,y] \\ S[x,y] &= S[x-1,y] + C[x,y] \end{aligned} \tag{9.13}$$

as shown on the right in Figure 9.12. In the first row of the image, the previous column sum is initialised as 0 (rather than feeding back from the row buffer) and, in the first column, the sum table for the row is initialised to 0. Two benefits of using Equation 9.13 rather than Equation 9.12 are that it reduces the number of operations and the row buffer can be narrower, since the column sum requires fewer bits per pixel than the area sum.

To use the summed area table for filtering requires dividing the sum from Equation 9.11 by the area to give the output pixel value. This can be seen as an extension of box filtering, as described in Section 8.2, to allow filtering with a variable window size.

In the context of filtering for geometric transformation, summed area tables have an advantage over pyramid structures in that they give more flexibility to the shape and size of the window. Note that with magnifications close to and greater than one, filtering is not necessary, and the input pixel values can simply be interpolated to give the required output.

Although an arbitrary warp requires the complete input image to be available for the reverse mapping, if the transformation keeps the image mostly the same (for example, correcting lens distortion or rotating by a small angle), then the transformation can begin before the whole image is buffered. Sufficient image rows must be buffered to account for the shift in the vertical direction between the input and output images. While a new input row is being streamed in, the output is calculated and streamed from the row buffers, as shown in Figure 9.13. The use of dual-port row buffers enables direct implementation of bilinear interpolation. Address calculations (to select the appropriate row buffer) are simplified by having the number of row buffers be a power of two, although this may be overcome by using a small lookup table to map addresses to row buffers. Reducing the number of row buffers required can significantly reduce both resource requirements and the latency (Akeila and Morris, 2008).

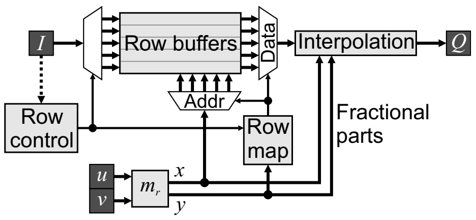


Figure 9.13 Reverse mapping with a partial frame buffer (using row buffers).

9.3 Interpolation

When transforming the image using the reverse map, the mapped coordinates are usually not on integer locations. Estimating the output pixel value requires estimating the underlying continuous intensity distribution and resampling this at the desired locations. The underlying continuous image, $C(x, y)$, is formed by convolving the input image samples with a continuous interpolation kernel, $K(x, y)$:

$$C(x,y) = \sum_{i,j} I[i,j]K(x-i, y-j) \tag{9.14}$$

which is illustrated in one dimension in Figure 9.14.

If the images are band-limited and sampled at greater than twice the maximum frequency, then the continuous image may be reconstructed exactly using a sinc kernel. Unfortunately, the sinc kernel has infinite extent and only decays slowly. Instead, simpler kernels with limited extent (or *region of support*) that approximate the sinc are used. This is important because it reduces the bandwidth required to access the input pixels.

Sampling the continuous image is equivalent to weighting a sampled interpolation kernel by the corresponding pixel value:

$$\begin{aligned} I[u, v] &= C(x,y)|_{x=u,y=v} \\ &= \sum_{i,j} I[i,j]K[u-i, v-j] \end{aligned} \tag{9.15}$$

Resampling can, therefore, be seen as filtering the input image with the weights given by the sampled interpolation kernel. There are two important differences with the filtering described in Section 8.2. Firstly, the fractional part of the input location changes from one output pixel to the next, so too do the filter weights. This means that some of the optimisations based on filtering with constant weights can no longer be applied. Secondly, the path through the input image corresponding to a row of pixels in the output will not, in general, follow a raster scan, but will follow an arbitrary curve. This makes the simple, regular, caching arrangements for local filters unsuited for interpolation, and more complex caching arrangements must be devised to manage the oblique or curved path through the input image.

Nearest neighbour interpolation is the simplest form of interpolation; it simply selects the nearest pixel to the desired location. This requires rounding the coordinates into the input image to the nearest integer,

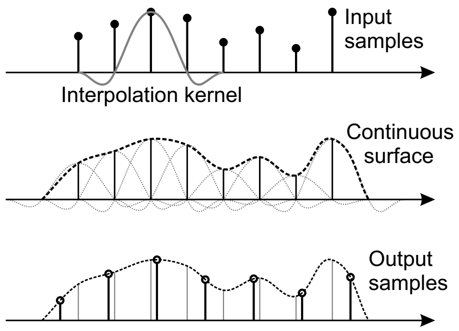


Figure 9.14 Interpolation process. Top: the input samples are convolved with a continuous interpolation kernel; middle: the convolution gives a continuous image; bottom: this is then resampled at the desired locations to give the output samples.

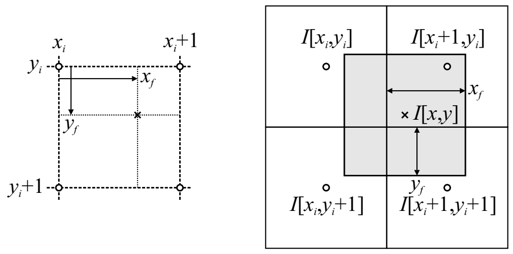


Figure 9.15 Left: coordinate definitions for interpolation; subscripts *i* and *f* denote integer and fractional parts of the respective coordinates. Right: bilinear interpolation.

which is achieved by adding 0.5 and selecting the integer component. If this addition can be incorporated into the mapping function, then the fractional component can simply be truncated from the calculated coordinates. The corresponding pixel in the input image can then be read directly from the frame buffer.

Other interpolation schemes combine the values of neighbouring pixels to estimate the value at a fractional position. The input pixel is split into its integer and fractional components

$$(x,y) = (x_i,y_i) + (x_f,y_f) \tag{9.16}$$

The integer parts of the desired location, (x_i,y_i), are used to select the input pixel values, which are weighted by values derived from the fractional components, (x_f,y_f), as defined in Figure 9.15.

As with image filtering, the logic also needs to appropriately handle boundary conditions. There are two specific cases that need to be handled. The first is when the output pixel does not fall within the input image. In this case, the interpolation is not required and an empty pixel (however this is defined for the application) is provided to the output. The second case is when the output pixel comes from near the edge of the input image and one or more of the neighbouring pixels are outside the image. It is necessary to provide appropriate values as input to the interpolation depending on the image extension scheme used.

9.3.1 Bilinear Interpolation

Perhaps the most commonly used interpolation method is bilinear interpolation. It combines the values of the four nearest pixels using separable linear interpolation:

$$\begin{aligned} I[x,y] = {} & I[x_i,y_i](1-x_f)(1-y_f) + I[x_i+1,y_i]x_f(1-y_f) + \\ & I[x_i,y_i+1](1-x_f)y_f + I[x_i+1,y_i+1]x_f y_f \end{aligned} \tag{9.17}$$

An alternative way of interpreting Equation 9.17 is demonstrated on the right in Figure 9.15. Each pixel location is represented by a square, with the weights given by the area of overlap between the desired output pixel and the available input pixels. Although Equation 9.17 appears to require eight multiplications, careful factorisation to exploit separability can reduce this to three:

$$\begin{aligned} I_{y_i} &= I[x_i,y_i] + x_f(I[x_i+1,y_i] - I[x_i,y_i]) \\ I_{y_{i+1}} &= I[x_i,y_i+1] + x_f(I[x_i+1,y_i+1] - I[x_i,y_i+1]) \\ I[x,y] &= I_{y_i} + y_f\left(I_{y_{i+1}} - I_{y_i}\right) \end{aligned} \tag{9.18}$$

While a random access implementation (for example such as performed in software) is trivial, accessing all four input pixels within a single clock cycle requires careful cache design.

If the whole image is available on chip, it will be stored in block RAMs. By ensuring that successive rows are stored in separate RAMs, a dual-port memory can simultaneously provide both input pixels on each line, hence all four pixels, in a single clock cycle. If only a single port is available, the data must be partitioned with odd and even addresses in separate banks so that all four input pixels may be accessed simultaneously. In most applications, however, there is insufficient on-chip memory available to hold the whole image and the image is held in an external frame buffer with only one access per clock cycle.

In this case, memory bandwidth becomes the bottleneck. Each pixel should only be read from external memory once, and held in an on-chip cache for use in subsequent windows. There are two main approaches that can be used. One is to load the input pixels as necessary; the other is to preload the cache and always access pixels directly from there. Since the path through the input image does not, in general, follow a raster scan, it is necessary to store not only the pixel values but also their locations within the cache. However, to avoid the complexity of associative memory, the row address can be implicit in which buffer the data is stored in. Consider a B row cache consisting of B separate block RAMs. The particular block RAM used to cache a pixel may be determined by taking the row address modulo B and using this to index the cache blocks. This is obviously simplified if B is a power of two, where the least significant bits of the row address give the cache block.

Loading input pixels on demand requires that successive output rows and pixels be mapped sufficiently closely in the input image that the reuse from one location to the next is such that the required input pixels are available (Gribbon and Bailey, 2004). This generally requires that the magnification be greater than one. Otherwise multiple external memory accesses are required to obtain the input data. If the output stream has horizontal blanking periods at the end of each row, this requirement may be partially relaxed through using a FIFO buffer on the output to smooth the flow. For a bilinear interpolation, a new frame may be initialised by loading the first row twice; the first time loads data into the cache but does not produce output, while the second pass produces the first row of output. This is similar to directly priming the row buffers for a regular filter.

The requirements are relaxed somewhat in a multiphase design, because more clock cycles are available to gather the required input pixel values.

Preloading the cache is a little more complex. It may require calculating in advance which pixels are going to be required. This will mean either performing the calculation twice, once for loading the cache and once for generating the output, or buffering the results of the calculation for use when all of the pixels are available in the cache. These two approaches are compared in Figure 9.16. Another alternative is for the cache control to monitor when data is no longer required and automatically replace the expired values with new data. This relies on the progression of the scan through the image in successive rows, so that

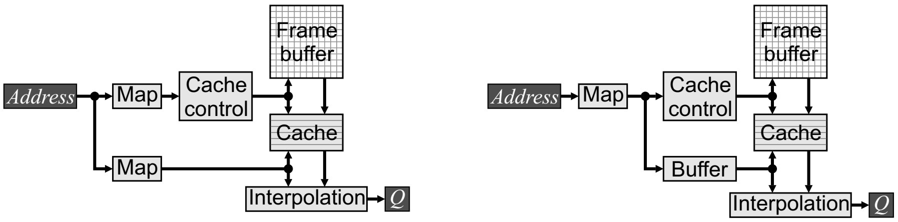


Figure 9.16 Preloading the cache. Left: calculating the mapping twice – once for loading the cache and once for performing the interpolation; right: buffering the calculated mapping for when the data is available.

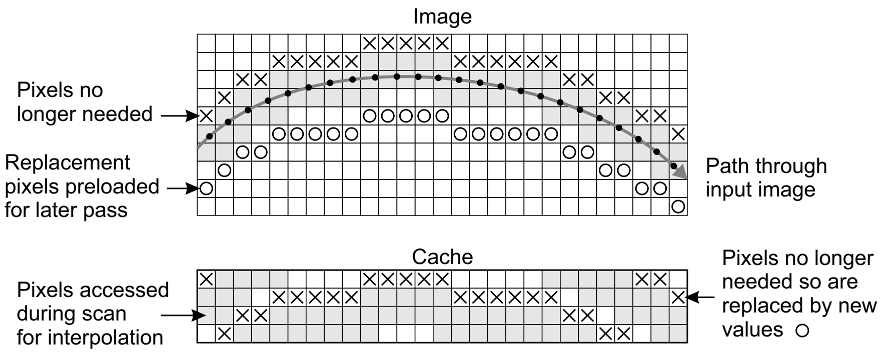


Image
Pixels no longer needed
Replacement pixels preloaded for later pass
Path through input image
Cache
Pixels accessed during scan for interpolation
Pixels no longer needed so are replaced by new values ○

Figure 9.17 Preloading pixels which are no longer required.

when a pixel is not used in a column it is replaced by the next corresponding pixel from the frame buffer. This process is illustrated in Figure 9.17.

9.3.2 Bicubic Interpolation

While bilinear interpolation gives reasonable results in many applications, a smoother result may be obtained through bicubic interpolation. Bicubic interpolation uses a separable piecewise cubic interpolation kernel with the constraints that it is continuous and has a continuous first derivative. These constraints result in the following kernel (Keys, 1981):

$$K_{Keys}(x) = \begin{cases} (a+2)|x|^3-(a+3)|x|^2+1, & 0 \le |x| < 1 \\ a|x|^3-5a|x|^2+8a|x|-4a, & 1 \le |x| < 2 \\ 0, & 2 \le |x| \end{cases} \tag{9.19}$$

where a is a free parameter. Selecting $a = -0.5$ makes the interpolated function agree with the Taylor series approximation to a sinc in as many terms as possible (Keys, 1981). The resulting one-dimensional interpolation weights two samples on either side of the desired data point, giving a region of support of four samples:

$$I[x] = \sum_{k=-1}^{2} w_k I[x_i + k] \tag{9.20}$$

where the weights may be derived by substituting the fractional part of the desired pixel location into Equation 9.19:

$$\begin{aligned} w_{-1} &= -\frac{1}{2}x_f^3 + x_f^2 - \frac{1}{2}x_f \\ w_0 &= \frac{3}{2}x_f^3 - \frac{5}{2}x_f^2 + 1 \\ w_1 &= -\frac{3}{2}x_f^3 + 2x_f^2 + \frac{1}{2}x_f \\ w_2 &= \frac{1}{2}x_f^3 - \frac{1}{2}x_f^2 \end{aligned} \tag{9.21}$$

These weights may either be calculated directly from the fractional component or determined through a set of small lookup tables. Since the weights sum to one, Equation 9.20 may be simplified to reduce the number of multiplications:

$$\begin{aligned} I[x] &= I[x_i+2]\left(1-\sum_{k=-1}^{1} w_k\right)+\sum_{k=-1}^{1} w_k I[x_i+k] \\ &= I[x_i+2]+\sum_{k=-1}^{1} w_k(I[x_i+k]-I[x_i+2]) \end{aligned} \tag{9.22}$$

The implementation of the separable bicubic interpolation requires a total of 15 multiplications, as demonstrated in Figure 9.18. Here the filter weights are provided by lookup tables on the fractional component of the pixel.

The larger region of support makes the design of the cache more complex for bicubic interpolation than for bilinear interpolation. In principle, the methods described above for bilinear interpolation may be extended, although memory partitioning is generally required to obtain the four pixels on each row within the window. The added complexity of memory management combined with the extra logic required to perform the filtering for a larger window make bilinear interpolation an acceptable compromise between accuracy and computational complexity in many applications.

Nuño-Maganda and Arias-Estrada (2005) use bicubic interpolation for image zooming. Since there is no rotation, the regular access pattern enables conventional filter structures to be used, although the coefficients change from one pixel to the next. Bellas *et al.* (2009) used bicubic interpolation to correct fisheye lens distortion, which requires a more complex access pattern within the image. They overcame the memory bandwidth by dividing the input image into overlapping tiles and preloading the image data for a complete tile into a set of block RAMs on the FPGA before processing that tile. The block RAMs then provide the required bandwidth to access all of the pixels needed to interpolate each output pixel.

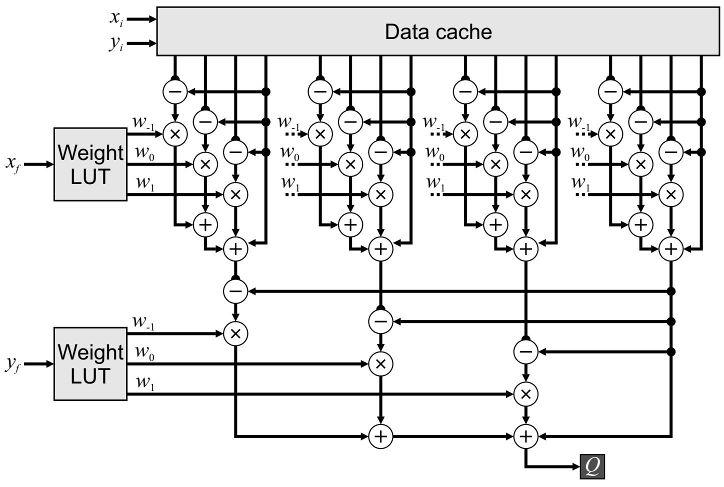


Figure 9.18 Implementation of bicubic interpolation.

9.3.3 Splines

A cubic spline is the next most commonly used interpolation technique. A cubic spline also defines a continuous signal that passes through the sample points as a series of piece-wise cubic segments, with the condition that both the first and second derivatives are continuous at each of the samples. This has the effect that changing the value of any point will affect the segments over quite a wide range. Consequently, interpolating with splines is not a simple convolution with a kernel, because of their wide region of support.

B-splines, on the other hand, do have compact support. They are formed from the successive convolution of a rectangular function, as illustrated in Figure 9.19. B_0 corresponds to nearest neighbour interpolation, while B_1 gives linear interpolation, as described above.

The cubic B-spline kernel is given by:

$$B_3(x) = \begin{cases} \frac{1}{2}|x|^3 - |x|^2 + \frac{2}{3}, & 0 \le |x| < 1 \\ -\frac{1}{6}|x|^3 + |x|^2 - 2|x| + \frac{4}{3}, & 1 \le |x| < 2 \\ 0, & 2 \le |x| \end{cases} \tag{9.23}$$

Its main limitation is that it does not go to zero at $|x| = 1$. This means that Equation 9.14 cannot be used directly for interpolation, since the continuous surface produced by convolving the image samples with the cubic B-spline kernel will not pass through the data points. Therefore, rather than weight the kernels with the image samples as in Equation 9.14, it is necessary to derive a set of coefficients based on the image samples:

$$C(x,y) = \sum_{i,j} c_{i,j} B_3(x-i, y-j) \tag{9.24}$$

These coefficients may be derived by solving the inverse problem, by setting $C[x,y] = I[x,y]$ in Equation 9.24. Solving directly, this requires a matrix inversion, although this can be implemented using infinite impulse response filters in both the forward and backward direction on each row and column of the image (Unser *et al.*, 1991). Spline interpolation therefore requires two stages, as shown in Figure 9.20: prefiltering the image to obtain the coefficients and then interpolation using the kernel. When implementing spline interpolation with an arbitrary (non-regular) sampling pattern, the prefiltering may be performed in the regularly sampled input image, with the result stored in the frame buffer. Then the interpolation stage may proceed as before, using the sampled kernel functions as filter weights to generate the output image.

Direct implementation of the infinite impulse response prefiltering presents a problem for a streamed implementation, because each one-dimensional filter requires two passes, once in the

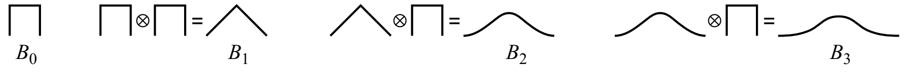


Figure 9.19 B-splines.

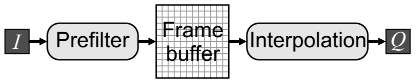
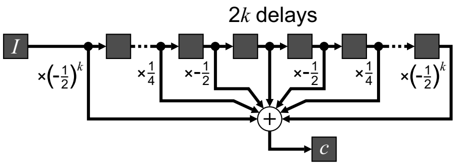


Figure 9.20 Structure for spline interpolation.



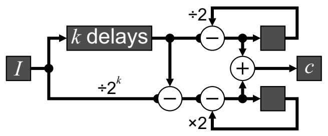


Figure 9.21 Efficient spline prefilter. Left: implementation as proposed in Ferrari and Park (1997); right: recursive implementation. (Reproduced with permission from L.A. Ferrari and J.H. Park, "An efficient spline basis for multi-dimensional applications: Image interpolation," *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, 578, 1997. © 1997 IEEE.)

forward direction and once in the reverse direction. However, since the impulse responses of these filters decay relatively quickly, they may be truncated and implemented using a finite impulse response filter (Ferrari and Park, 1997).

Rather than use B-splines, the kernel can be selected so that the coefficients of the prefilter are successively smaller powers of two (Ferrari and Park, 1997) (this has been called a 2-5-2 spline). This enables the filter to be implemented efficiently in hardware, using only shifts for the multiplications, as demonstrated on the left in Figure 9.21. Since each coefficient is shifted successively by one bit, shifting by more than k bits, where k is the number of bits used in the fixed-point representation of $c_{i,j}$, will make the term go to zero. This provides a natural truncation point for the window.

Since each half of the filter is a power series, they may be implemented recursively, as shown in the right hand panel of Figure 9.21. The bottom accumulator sums the terms in the first half of the window and needs to be $2k$ bits wide to avoid round-off errors. The upper accumulator sums the terms in the second half of the window. It does not need to subtract out the remainder after a further k delays, because that decays to zero (this is using an infinite impulse response filter).

The kernel for the 2-5-2 spline is:

$$K_{2-5-2}(x)=\begin{cases}\frac{1}{4}|x|^3-\frac{7}{12}|x|^2+\frac{5}{9}, & 0\le|x|<1\\ \frac{1}{36}|x|^3+\frac{1}{12}|x|^2-\frac{2}{3}|x|+\frac{7}{9}, & 1\le|x|<2\\ 0, & 2\le|x|\end{cases} \tag{9.25}$$

with the corresponding filter weights given as (Ferrari *et al.*, 1999):

$$\begin{aligned} w_{-1}&=\frac{1}{36}x_f^3+\frac{1}{6}x_f^2-\frac{5}{12}x_f+\frac{2}{9}\\ w_0&=\frac{1}{4}x_f^3-\frac{7}{12}x_f^2+\frac{5}{9}\\ w_1&=-\frac{1}{4}x_f^3+\frac{1}{6}x_f^2+\frac{5}{12}x_f+\frac{2}{9}\\ w_2&=-\frac{1}{36}x_f^3+\frac{1}{4}x_f^2\end{aligned} \tag{9.26}$$

These are used in the same way as for the bicubic interpolation, although the awkwardness of these coefficients makes lookup table implementation better suited than direct calculation.

The filters given here are one dimensional. For two-dimensional filtering, the two filters can be used separably on the rows and columns, replacing the delays with row buffers. If the input is coming from a frame buffer, and the memory bandwidth allows, then the k row buffer delay may be replaced by a second access to the frame buffer in a similar manner to the box filter shown in Figure 8.20.

An FPGA implementation of this cubic spline filter was used to zoom an image by an integer factor by Hudson *et al.* (1998). The prefilter shown in the left of Figure 9.21 was used. The interpolation stage was simplified by exploiting the fact that a pure zoom results in a regular access pattern. Since this was fairly early work, the whole algorithm did not fit on a single FPGA. One-dimensional filters were implemented, with the data fed through separately for the horizontal and vertical filters. Dynamic reconfiguration was also used to separate the prefiltering and interpolation stages.

A wide range of other interpolation kernels is available; a review of the more commonly used kernels for image processing is given elsewhere (Wolberg, 1990). Very few of these have been used for FPGA implementation because of their higher computational complexity than those described in this section.

9.3.4 Interpolating Compressed Data

Interpolation can also be used to reduce the data volume when using non-parametric methods of representing geometric distortion, uneven illumination or vignetting. Rather than representing the map (either geometric transformation or intensity map) by a parametric model, the map is stored explicitly for each pixel. For a geometric transformation, this map represents the source location (for a reverse mapping) for each pixel in the output image (Wang *et al.*, 2005). For modelling uneven illumination, the map contains the illumination level or correction scale factor directly for each pixel. This has two advantages over model-based approaches. Firstly, evaluating the map is simply a table lookup, so is generally much faster than evaluating a complex parametric model. Secondly, a non-parametric map is able to capture distortions and variations that are difficult or impossible to model with a simple parametric model (Bailey *et al.*, 2005). However, these advantages come at the cost of the large memory required to represent the map for every pixel. If the map is smooth and slowly varying, then it can readily be compressed. The simplest form of compression from a computational point of view is to down-sample the map and use interpolation to reconstruct the map for a desired pixel (Akeila and Morris, 2008).

The resolution required to represent the down-sampled map obviously depends on the smoothness of the map, and also the accuracy with which it needs to be recreated. There is a trade-off here. Simple bilinear interpolation will require more samples to represent the map to a given degree of accuracy than a bicubic or spline interpolation requires, but require less computational logic to evaluate the map for a given location.

For stream processing, the required points will be aligned with the scan axis, simplifying the access pattern. The relatively small number of weights used can be stored in small RAMs (fabric RAM is ideal for this). With a separable mapping, the vertical mapping can be performed first, since each set of vertical points will be reused for several row calculations.

9.4 Mapping Optimisations

With stream processing, the input to the mapping function is a raster scan, regardless of whether a forward or reverse mapping is used (refer to Figure 9.2 to see that this is so). Therefore, incremental calculation can be used to simplify the computation by exploiting the fact that the y (or v) address is constant and the x (or u) address increments with successive pixels.

For an affine transformation, Equation 9.6 can be represented in matrix form as:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_{ux} & a_{uy} & a_u \\ a_{vx} & a_{vy} & a_v \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{9.27}$$

Implemented directly, this transformation would require six multiplications and four additions. With incremental calculation, the only change moving from one pixel to the next along a row is $x \leftarrow x+1$. Substituting this into Equation 9.27 gives:

$$\begin{bmatrix} u \\ v \end{bmatrix} \leftarrow \begin{bmatrix} a_{ux} & a_{uy} & a_u \\ a_{vx} & a_{vy} & a_v \end{bmatrix} \begin{bmatrix} x+1 \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} a_{ux} \\ a_{vx} \end{bmatrix} \tag{9.28}$$

requiring only two additions. A similar simplification can be used for moving on to the next row. When implementing these, it is convenient to have a temporary register to hold the position at the start of the row. This may be incremented when moving from one row to the next and is used to initialise the main register, which is updated for stepping along the rows, as shown in Figure 9.22. It is necessary that these registers maintain sufficient guard bits to prevent the accumulation of round-off errors.

Equation 9.27 can be extended to a perspective transformation using homogenous coordinates:

$$\begin{bmatrix} mu \\ mv \\ m \end{bmatrix} = \begin{bmatrix} a_{ux} & a_{uy} & a_u \\ a_{vx} & a_{vy} & a_v \\ a_{kx} & a_{ky} & a_k \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{9.29}$$

where the third row gives a position dependent magnification, m. The mu and mv terms are then divided by m to give the transformed coordinates. The same incremental architecture from Figure 9.22 may be used for the perspective transformation, with the addition of a division at each clock cycle to perform the magnification scaling.

The mappings shown here for the affine and perspective transformations are for forward mappings. The reverse mappings will have the same form and may be determined simply by inverting the corresponding transformations.

Using incremental mapping for polynomial warping is a little more complex. Consider just the u component of a second order polynomial forward mapping:

$$u_{x,y} = a_{uxx}x^2 + a_{uxy}xy + a_{uyy}y^2 + a_{ux}x + a_{uy}y + a_u \tag{9.30}$$

First consider the initialisation of each row:

$$u_{0,y} = a_{uyy}y^2 + a_{uy}y + a_u \tag{9.31}$$

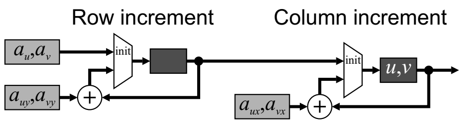


Figure 9.22 Incremental mapping for affine transformation.

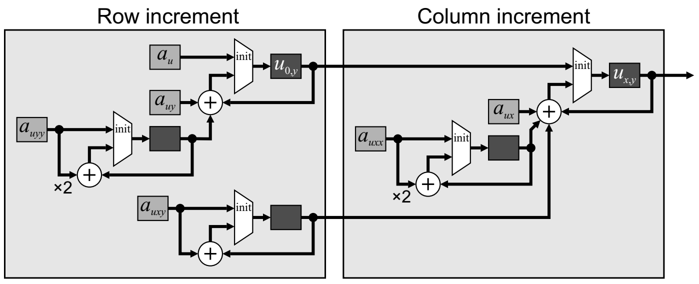


Figure 9.23 Incremental calculation for a second order polynomial warp.

For the row increment, it is necessary to know the value from the previous row:

$$\begin{aligned} u_{0,y-1} &= a_{uyy}(y-1)^2 + a_{uy}(y-1) + a_u \\ &= a_{uyy}y^2 + a_{uy}y + a_u - a_{uyy}(2y-1) - a_{uy} \end{aligned} \tag{9.32}$$

Therefore:

$$u_{0,y} = u_{0,y-1} + a_{uyy}(2y-1) + a_{uy} \tag{9.33}$$

with the second term produced by another accumulator, as shown in Figure 9.23.

For the column increment along the row, it is necessary to know the previous value:

$$\begin{aligned} u_{x-1,y} &= a_{uxx}(x-1)^2 + a_{uxy}(x-1)y + a_{uyy}y^2 + a_{ux}(x-1) + a_{uy}y + a_u \\ &= a_{uxx}x^2 + a_{uxy}xy + a_{uyy}y^2 + a_{ux}x + a_{uy}y + a_u - a_{uxx}(2x-1) - a_{uxy}y - a_{ux} \end{aligned} \tag{9.34}$$

Therefore:

$$u_{x,y} = u_{x-1,y} + a_{uxx}(2x-1) + a_{uxy}y + a_{ux} \tag{9.35}$$

where the second term is produced by a secondary accumulator incremented with each column and the third term is provided by a secondary accumulator incremented each row.

A parallel circuit is also required for the v component. A similar approach may be taken for higher order polynomial mappings, with additional layers of accumulators required for the higher order terms.

Unlike the affine and perspective mappings, the inverse of a polynomial mapping is not another polynomial mapping. Therefore, it is important to determine the mapping coefficients for the form that is being used. It is not a simple matter to convert between the forward and reverse mapping.

9.5 Image Registration

In the previous sections, it was assumed that the geometric transformation was known. Image registration is the process of determining the relationship between the pixels within two images, or between parts of images. There is a wide variety of image registration methods (Brown, 1992; Zitova and Flusser, 2003), of which only a selection of the main approaches are outlined here. General image registration involves four steps (Zitova and Flusser, 2003): feature detection, feature matching, transform model estimation and image transformation and resampling. The last step has been discussed in some detail earlier in this chapter. This section will therefore focus on the other steps. There are two broad classes of registration: those based on matching sets of feature points and those based on matching extended areas within the image. Since these are quite different in their approach, they are described separately. This section concludes with some of the applications of image registration.

9.5.1 Feature-Based Methods

A common approach to registration is to find corresponding points within the images to be registered. While such points can be found manually, this precludes automatic processing. Therefore, a feature extraction step is required that locates salient and distinctive points within each image. Ideally, these should be well distributed throughout the image and be easy to detect. The features should also be invariant to the expected transformation between the images. Once a set of features is found, it is then necessary to match these between images. Not all feature points found in one image may be found in the other, so the matching method must be robust against this.

9.5.1.1 Feature Detection

If the scene is contrived (as is often the case when performing calibration), for example consisting of a array of dots, a checkerboard pattern or a grid, then the detection method can be tailored specifically to those expected features. For an array of dots, adaptive thresholding can reliably segment the dots, with the centre of gravity used to define the feature point location. With a checkerboard pattern, either an edge detector is used to detect the edges, from which the corners are inferred, or a corner detector is used directly. A corner detector is just a particular filter that is tuned to locating corners (Noble, 1988). Consequently, corner detectors can be readily implemented on FPGAs (Claus *et al.*, 2009). For grid targets, a line detector can be used, with the location of the intersections identified as the feature point locations. An example of this is considered in more detail in Section 14.2.

The regular structure of a contrived target makes correspondence matching relatively easy because the structure is known in advance. The danger with using a periodic pattern is that the matched locations may be offset by one or more periods between images. This may be overcome by adding a distinctive feature within the image that can act as a key for alignment.

Edge and corner features can also be used with natural scenes, but often do not work as well because such points are usually not as well defined in terms of contrast, and there may be many features detected in one image but not the other. To overcome this, it is necessary to find only the dominant features which are likely to appear in both images. While there are several approaches that could be taken, one that has gained prominence in recent years is the scale invariant feature transform (SIFT).

SIFT (Lowe, 1999; 2004) filters the images with a series of Gaussian filters to obtain a representation of the image at a range of scales. The difference between successive scales (a difference of Gaussian filter) is a form of band-pass filter that responds most strongly to objects or features of that scale. The local maxima and minima within each scale are then located and compared with adjacent scales, both larger and smaller. The local extremes both spatially and in scale that are greater than a preset threshold (3% of the pixel range), are considered as possible feature points and are located to subpixel accuracy. Since the difference of Gaussian filter also responds strongly to edges, there is a danger that a point will be poorly located along the edge and, therefore, be sensitive to small amounts of noise. Therefore, it is necessary to eliminate peaks which are poorly defined along the edge direction. Let D be the response of the difference of Gaussian filter. Then only the points which satisfy

$$\frac{(D_{xx}+D_{yy})^2}{D_{xx}D_{yy}-D_{xy}^2} < Thr \tag{9.36}$$

are kept, where D_{xx}, D_{yy} and D_{xy} are second derivatives and the threshold $Thr = 12.1$ (Lowe, 2004).

The feature orientation is determined by taking the derivatives of the Gaussian filtered image at the appropriate scale. The orientations within a window are examined and the dominant direction determined through weighted averaging. Finally, local gradient information is used to derive a descriptor of the feature point. The feature orientation is used as the reference to give orientation independence, with the descriptor built from a set of histograms of gradients weighted by edge strength. Basing the descriptor on gradients reduces the sensitivity to absolute light levels, and normalising the final feature to unit length reduces sensitivity to contrast, improving the robustness.

In implementing a feature detection algorithm such as SIFT, most of the computation is spent in filtering image. Typically less than 1% of the pixels within the image are feature points. Several factors can be exploited to filter the images efficiently. The first is that the Gaussian window is separable, so the two-dimensional filter can be decomposed into a one-dimensional filter along the rows and a one-dimensional filter down the columns. Secondly, a Gaussian of a larger scale may be formed by cascading another Gaussian filter over the already processed image. Thirdly, after blurring the image sufficiently, it may be safely down-sampled, reducing the volume of processing in the subsequent stages. These stages are illustrated in Figure 9.24. Each of the blocks (apart from the feature point processing) was a filter described in Chapter 8, and the feature processing is outlined above.

Such processing has been implemented on an FPGA. Pettersson and Petersson (2005) used five element Gaussian filters and four iterations of CORDIC to calculate the magnitude and direction of the gradient. They also simplified the gradient calculation, with only one filter used per octave rather than one per scale. Chang and Hernandez-Palancar (2009) made the observation that after down-sampling the data volume is reduced by a factor of four. This allows all of the processing for the second octave and below to share a single Gaussian filter block and peak detect block.

Yao *et al.* (2009) took the processing one step further and also implemented the descriptor extraction in hardware. The gradient magnitude used the simplified formulation of Equation 8.18 rather than the square root, and a small lookup table was used to calculate the orientation. They used a modified descriptor that

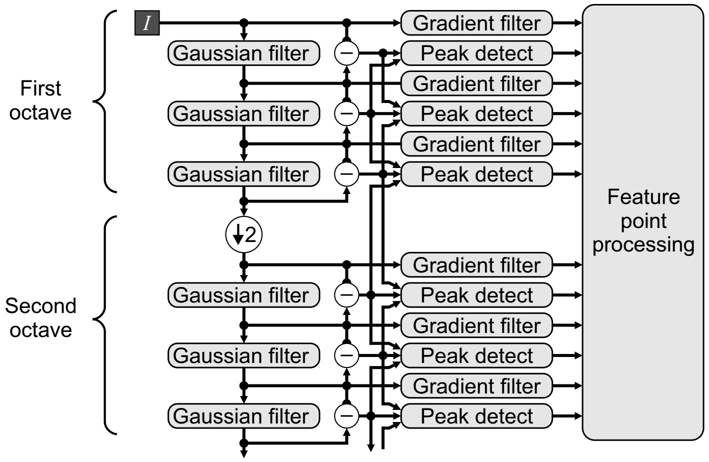


Figure 9.24 SIFT feature point extraction.

had fewer parameters, but was easier to implement in hardware. The feature matching, however, was performed in software using an embedded MicroBlaze processor.

9.5.1.2 Feature Matching

The matching step aims to find the correspondence between features detected in the two images. Again, there is a wide range of matching techniques (Zitova and Flusser, 2003), with a full description beyond the scope of this work. The method based on invariant descriptors used by SIFT will be discussed as an example of one matching algorithm.

Features descriptors which are invariant to the expected transformation may be matched by finding the feature in the reference image which has the most similar descriptor. The descriptor match is unlikely to be exact because of noise and differences resulting from local distortions. With a large number of descriptors and a large number of components within each descriptor, a brute force search for the closest match is prohibitive. Therefore, efficient search techniques are required to reduce the search cost.

If the reference is within a database, for example when searching for predefined objects within an image, then two approaches to reduce the search are hashing and tree-based methods. A locality sensitive hash function (Gionis *et al.*, 1999) compresses the dimensionality, enabling approximate closest points to be found efficiently. Alternatively, the points within the database may be encoded in a high-dimensional tree-based structure and, using a best bin first heuristic, the tree may be searched efficiently (Beis and Lowe, 1997). Such searches for a particular point are generally sequential. While the search for multiple points may be performed independently, implying that they may be searched in parallel, for large databases this is also impractical. Serial search methods with complex data structures and heuristics may be best performed in software, although it may be possible to accelerate key steps using appropriate hardware.

For matching between images, one approach is to build a database of points from the reference image and use the approaches described in the previous paragraph. An alternative is to combine the matching and model extraction steps. The first few points are matched using a full search. These are then used to create a model of the transformation, which is used to guide the search for correspondences for the remaining points. New matches can then be used to refine the transformation. Again, the search process is largely sequential, although hardware may be used to accelerate the transformation to speed up the search for new matches.

9.5.1.3 Model Extraction

Given a set of points, the transformation model defines the best mapping of the points from one image to the other. For a given model, determining the transformation can be considered as an optimisation problem, where the model parameters are optimised to find the best match.

If the model is sufficiently simple (for example an affine model, which is linear in the model parameters) it may be solved directly using least squares minimisation. Consider for example the affine mapping of Equation 9.6; this may be represented in matrix form in terms of the transform coefficients (compare this with Equation 9.27) as:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{bmatrix} \begin{bmatrix} a_{ux} \\ a_{uy} \\ a_u \\ a_{vx} \\ a_{vy} \\ a_v \end{bmatrix} \tag{9.37}$$

Each pair of corresponding points provides two constraints. Therefore, the set of equations for N sets of control points has $2N$ constraints:

$$\begin{bmatrix} u_1 \\ v_1 \\ \vdots \\ u_N \\ v_N \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N & y_N & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_N & y_N & 1 \end{bmatrix} \begin{bmatrix} a_{ux} \\ a_{uy} \\ a_u \\ a_{vx} \\ a_{vy} \\ a_v \end{bmatrix} \tag{9.38}$$

or

$$\mathbf{u} = \mathbf{Ma} \tag{9.39}$$

Usually this set of equations is over-determined and with noise it is impossible to find model parameters which exactly match the constraints. In this case, the transform parameters can be chosen such that the total squared error is minimised:

$$\begin{aligned} E^2 &= (\mathbf{u}-\mathbf{Ma})^{\mathrm{T}}(\mathbf{u}-\mathbf{Ma}) \\ &= \mathbf{u}^{\mathrm{T}}\mathbf{u}-2\mathbf{a}^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}\mathbf{u}+\mathbf{a}^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}\mathbf{Ma} \end{aligned} \tag{9.40}$$

This is minimised by taking the partial derivative with respect to each parameter and setting the result to zero:

$$\frac{\partial E^2}{\partial \mathbf{a}} = -2\mathbf{M}^{\mathrm{T}}\mathbf{u}+2\mathbf{M}^{\mathrm{T}}\mathbf{Ma} = 0 \tag{9.41}$$

so solving

$$\mathbf{M}^{\mathrm{T}}\mathbf{Ma} = \mathbf{M}^{\mathrm{T}}\mathbf{u} \tag{9.42}$$

to give

$$\mathbf{a} = \left(\mathbf{M}^{\mathrm{T}}\mathbf{M}\right)^{-1}\mathbf{M}^{\mathrm{T}}\mathbf{u} \tag{9.43}$$

yields the least squares solution. The regular nature of matrix operations and inversion makes them amenable to FPGA implementation (Irturk *et al.*, 2008; Burdeniuk *et al.*, 2010), although if care is not taken with scaling this can result in significant quantisation errors if fixed-point arithmetic is used. Least squares optimisation is quite sensitive to outliers, so robust fitting methods can be used, if necessary, to identify outliers, and remove their contribution from the solution.

If the corresponding points may come from one of several models, then the mixture can cause even robust fitting methods to fail. One approach suggested by Lowe (2004) was to have each pair of corresponding points vote for candidate model parameters using a coarse Hough transform (the principles of the Hough transform are described in Section 11.7). This effectively sorts the pairs of corresponding points based on the particular model, filtering the outliers from each set of points.

With more complex or nonlinear models, a closed form solution of the parameters is not possible. In this case, it is common to use an iterative technique as outlined in Figure 9.25. An initial estimate of the model is used to transform the reference points. The error can then directly be measured against the target points,

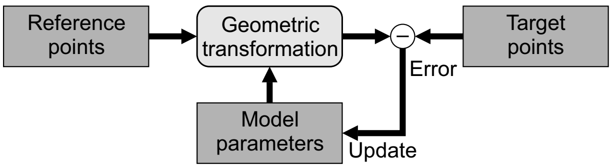


Figure 9.25 Iterative update of registration model.

with the error used to update the model using steepest descent or Levenberg–Marquardt (Press *et al.*, 1993) error minimisation.

9.5.2 Area-Based Methods

The alternative to feature-based image registration is to match the images themselves, without explicitly detecting objects within the image. Such area matching may be performed on the image as whole, or on small patches extracted from the image.

Area-based matching has a number of limitations compared with feature-based matching (Zitova and Flusser, 2003). Any distortions within the image affect the shape and size of objects within the image and, consequently, the quality of the match. Similarly, since area matching is generally based on matching pixel values, the matching process can be confounded by any change between the images, for example by varying illumination or using different sensors. When using small image patches, if the patch is textureless without any significant features, it can be matched with just about any other smooth area within the other image, and often the matching is based more on noise or changes in shading rather than a true match. In spite of these limitations, area-based matching is in common use.

9.5.2.1 Correlation Methods

The classical area matching method is cross-correlation. This extends Equation 6.40 by removing the mean from the two images:

$$CC_N[i,j] = \frac{\sum\limits_{x,y}\left(f[x+i,y+j]-\mu_f\right)\left(g[x,y]-\mu_g\right)}{\sqrt{\sum\limits_{x,y}\left(f[x+i,y+j]-\mu_f\right)^2}\sqrt{\sum\limits_{x,y}\left(g[x,y]-\mu_g\right)^2}} \tag{9.44}$$

Although, for computational simplicity, the simple sum of absolute differences (Equation 6.24) or sum of squared differences (Equation 6.25) is often substituted for the correlation as a measure of similarity. The offset that gives the maximum correlation or minimum difference provides the best registration. When looking for a smaller target within a larger image, this process is often called *template matching*. If the target is sufficiently small, this can be implemented directly using filtering to produce an image of match scores, where the local maxima above a threshold (or minima for difference methods) are the detected locations of the target.

There are several limitations with correlation for image matching. It is generally restricted to a simple translational model, although for correlating small patches it can tolerate small image imperfections. It is also computationally expensive, especially when matching one image to another. While measuring the similarity is straightforward, it is necessary to repeat this measurement over all possible offsets to find the maximum. Since the correlation surface is usually not of particular interest, the computation may be reduced by restricting the number of different offsets that are measured. Such techniques use search methods to find the optimum offset. This is complicated, however, by the fact that there may be several local maxima so there is a danger of finding the wrong offset.

One technique commonly used to reduce the search complexity is to use a pyramidal approach to the search. An image pyramid is produced by filtering and down-sampling the images to produce a series of progressively lower resolution versions. These enable a wide range of offsets to be searched relatively quickly for two reasons: the lower resolution images are significantly smaller, so measuring the similarity for each offset is faster, and a large step size is used because each pixel is much larger. This enables the global maximum to be found efficiently, even using an exhaustive search at the lowest resolution if necessary. The estimated offset obtained from the low resolution images may not be particularly accurate, but can be refined by finding the maximum at progressively higher resolutions. These latter searches are made more efficient by restricting the search region to the global maximum and can safely perform restricted searches without worry of locking onto the wrong peak.

It is straightforward to perform the correlation for multiple offsets in parallel. One way of achieving this is shown in Figure 9.26, where the 16 correlations within a 4×4 search window are performed in parallel in a single scan through the pair of images. One of the input images is offset relative to the other by a series of shift registers and row buffers, with a multiply and accumulate (MAC) unit at each offset of interest. The multiplication could just as easily be replaced by an absolute difference or squared difference. With a pyramidal search, a block such as this can be reused for each level of the pyramid.

The correlation peak is found to the nearest pixel. To estimate the offset to subpixel accuracy, it is necessary to interpolate between adjacent correlation values. Commonly a parabola is fitted between the maximum pixel, $CC[i_0,j_0]$, and those on either side. The horizontal subpixel offset is then:

$$x_0 = i_0 + \frac{CC[i_0+1,j_0]-CC[i_0-1,j_0]}{4CC[i_0,j_0]-2(CC[i_0+1,j_0]+CC[i_0-1,j_0])} \tag{9.45}$$

and similarly for the vertical offset (Tian and Huhns, 1986). However, fitting a pyramid has been found to be better (the autocorrelation of a piece-wise constant image has a pyramidal peak) (Bailey and Lill, 1999; Bailey, 2003). This gives the subpixel offset as:

$$x_0 = i_0 + \frac{CC[i_0+1,j_0]-CC[i_0-1,j_0]}{2(CC[i_0,j_0]-\min(CC[i_0+1,j_0],CC[i_0-1,j_0]))} \tag{9.46}$$

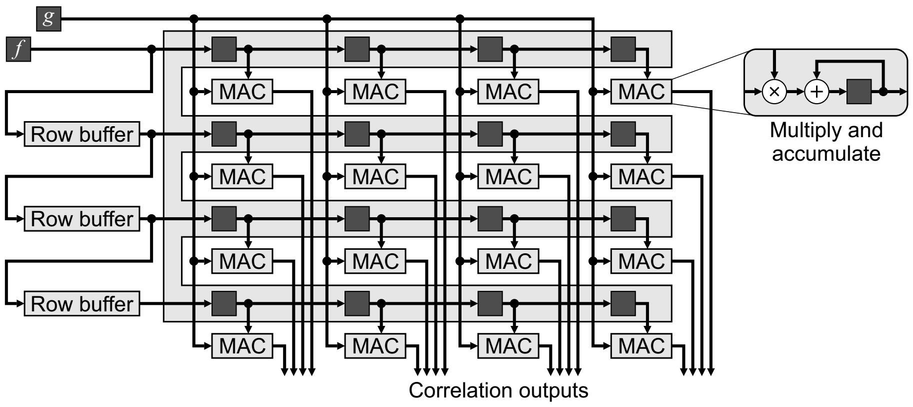


Figure 9.26 Calculating a 4×4 block of cross correlations in parallel.

Extending correlation-based matching to transformations more complex than simple translation quickly becomes computationally intractable. Each transformation parameter adds an additional dimension to the search space and effectively requires the image to be transformed for each iteration of the comparison.

9.5.2.2 Fourier Methods

Another approach to accelerating correlation is to perform the computation in the frequency domain. Consider unnormalised correlation:

$$COR[i,j] = \sum_{x,y} f[x+i, y+j]g[x,y] \tag{9.47}$$

Taking the two-dimensional discrete Fourier transform of Equation 9.47:

$$COR[u,v] = F^*[u,v]G[u,v] \tag{9.48}$$

converts the correlation to a simple product in the frequency domain, where u and v are the spatial frequencies in the x and y directions respectively, and F^* is the complex conjugate of F. Therefore, the computation to perform an exhaustive search may be significantly reduced by taking the Fourier transform of each image, calculating the product and then taking the inverse Fourier transform. (See Section 10.1 for implementing the Fourier transform on an FPGA.) The limitations of using a simple unnormalised correlation apply to using the Fourier transform to implement the correlation. It is not easy to incorporate normalisation without destroying the simplicity of Equation 9.48.

Phase correlation reduces this problem by considering only the phase. If f is the offset version of g:

$$g[x,y] = f[x+x_0, y+y_0] \tag{9.49}$$

then from the shift theorem of the Fourier transform (Bracewell, 2000), in the frequency domain this becomes:

$$G[u,v] = F[u,v]e^{j2\pi(ux_0+vy_0)} \tag{9.50}$$

Taking the phase angle of the ratio of the two spectra gives:

$$\begin{aligned} ux_0 + vy_0 &= \frac{1}{2\pi}\angle\frac{G[u,v]}{F[u,v]} = \frac{1}{2\pi}(\angle G[u,v] - \angle F[u,v]) \\ &= \frac{1}{2\pi}\angle\left(F^*[u,v]G[u,v]\right) \end{aligned} \tag{9.51}$$

There are two methods by which the offset may be estimated. The first is to normalise the magnitudes to one and take the inverse Fourier transform:

$$\frac{F^*[u,v]G[u,v]}{\|F^*[u,v]G[u,v]\|} \Leftrightarrow \delta[x+x_0, y+y_0] \tag{9.52}$$

where $\delta[x+x_0, y+y_0]$ is a delta function at $(-x_0, -y_0)$. Phase noise, combined with the fact that the offset is not likely to be exactly an integer number of pixels, means that the output is not a delta function. However, a search of the output for the maximum value will usually give the offset to the nearest pixel.

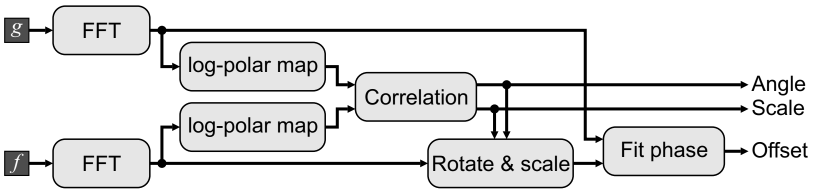


Figure 9.27 Fourier-based registration, including a rotation and scale factor.

The second approach is to fit a plane to the phase difference in Equation 9.51 using least squares. This first requires unwrapping the phase, because the arctangent reduces the phase to within $\pm\pi$. In practise, the quality of the phase estimate at a given frequency will depend on the amplitude. Therefore, the fit may be formed by weighting the phase samples by their amplitudes, or masking out those frequencies with low amplitude or those that differ significantly in amplitude between the two images (Stone *et al.*, 1999; 2001).

Just using the Fourier transform is limited to pure translation, although the offset may be estimated to subpixel accuracy. To extend the registration to include rotation and scaling requires additional processing. Rotating an image will rotate its Fourier transform and scaling an image will result in an inverse scale within the Fourier transform. Therefore, by converting the Fourier transform from rectangular coordinates to log-polar coordinates (the radial axis is the logarithm of the frequency), rotating the image corresponds to an angular shift and scaling the image results in a radial shift (Reddy and Chatterji, 1996). The angle and scale factor may be found by correlation (or even using a Fourier-based method to estimate the offset. The structure of the algorithm is shown in Figure 9.27.

9.5.2.3 Gradient Methods

Rather than perform an exhaustive search in correlation space, gradient-based methods effectively perform a hill climbing search using an estimate of the offset based on derivatives of the image gradient. Expanding an offset image, $f(x+x_0, y+y_0)$, about $f(x,y)$ using a two-dimensional Taylor series gives:

$$g(x,y) = f(x,y) + x_0\frac{\partial f}{\partial x} + y_0\frac{\partial f}{\partial y} + \frac{1}{2!}\left(x_0^2\frac{\partial^2 f}{\partial x^2} + 2x_0y_0\frac{\partial^2 f}{\partial x \partial y} + y_0^2\frac{\partial^2 f}{\partial y^2}\right) + \cdots \tag{9.53}$$

For small offsets, the Taylor series can be truncated (the second order and higher terms are considered negligible and contribute to noise). Therefore, given estimates of the gradients, the offset may be estimated from:

$$g[i,j] = f[i,j] + x_0\frac{\partial f}{\partial x} + y_0\frac{\partial f}{\partial y} \tag{9.54}$$

where the gradients may be obtained from a central difference filter. Each pixel then provides a constraint; since this over-determined system is linear in the offsets x_0 and y_0, it may be solved by least squares minimisation (Lucas and Kanade, 1981).

Table 9.1 Filter coefficients for optimal prefilters and derivative filters. (Reproduced with permission from H. Farid and E.P. Simoncelli, "Differentiation of discrete multidimensional signals," *IEEE Transactions on Image Processing*, **13**, 4, 496–508, 2004. © 2004 IEEE.)

Filter	−3	−2	−1	0	1	2	3
Prefilter			0.229 879	0.540 242	0.229 879		
Derivative			−0.425 287	0.000 000	0.425 287		
Prefilter		0.037 659	0.249 153	0.426 375	0.249 153	0.037 659	
Derivative		−0.104 550	−0.292 315	0.000 000	0.292 315	0.104 550	
Prefilter	0.004 711	0.069 321	0.245 410	0.361 117	0.245 410	0.069 321	0.004 711
Derivative	−0.018 708	−0.125 376	−0.193 091	0.000 000	0.192 091	0.125 376	0.018 708

$$\begin{bmatrix} \sum_{i,j}\left(\frac{\partial f[i,j]}{\partial x}\right)^2 & \sum_{i,j}\frac{\partial f[i,j]}{\partial x}\frac{\partial f[i,j]}{\partial y} \\ \sum_{i,j}\frac{\partial f[i,j]}{\partial x}\frac{\partial f[i,j]}{\partial y} & \sum_{i,j}\left(\frac{\partial f[i,j]}{\partial y}\right)^2 \end{bmatrix}\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} \sum_{i,j}\frac{\partial f[i,j]}{\partial x}(g[i,j]-f[i,j]) \\ \sum_{i,j}\frac{\partial f[i,j]}{\partial y}(g[i,j]-f[i,j]) \end{bmatrix} \tag{9.55}$$

The Taylor series approximation may be improved by prefiltering the images with a low pass filter. The derivatives can then be obtained by using a matching the derivative filter to the derivative of the prefilter (Farid and Simoncelli, 2004). The optimal low order filter pairs are listed in Table 9.1. In two dimensions, the filters are separable, resulting in a pipelined implementation similar to Figure 9.28.

Since the offset is only approximate (it neglects the high order series expansion terms) the expansion may be repeated about the approximation. After the first approximation, the offset will be closer, so the truncated Taylor series will be more accurate, enabling a more accurate estimate of the offset to be obtained. Iterating the formulation will converge to the minimum error (Lucas and Kanade, 1981).

Variations on this theme can manage more complex geometric transformations or use different optimisation techniques to minimise the error (Baker and Matthews, 2004). More complex transformations follow the general iterative pattern of Figure 9.25. These, too, can be based on a pyramidal coarse-to-fine search strategy to reduce the chances of becoming locked onto a local minimum (Thevenaz *et al.*, 1998). The initial match is based on large scale features, with the successive refinements at finer scales making small corrections for finer details.

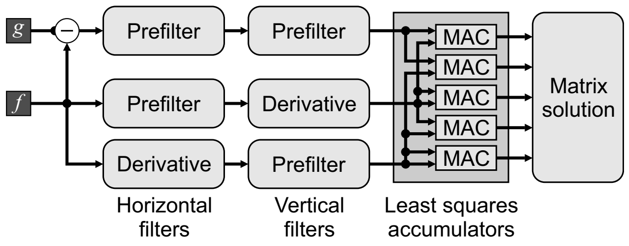


Figure 9.28 Basic approach of gradient-based registration.

Registration is also used to combine data from difference sensor types. This is common in both remote sensing and medical imaging. The basic image matching metrics of correlation or sum of absolute differences do not perform very well at registering images where there are significant differences in contrast. In these instances, is more effective to find the transformation that maximises mutual information (Viola and Wells, 1997).

9.5.2.4 Optimal Filtering

When looking at subpixel registration involving pure translation, the gradient-based methods effectively create a continuous surface through interpolation, which is then resampled with the desired offset. This is then compared with the target image, as in Figure 9.25, to minimise the error. Expressing the interpolation as a convolution in the form of Equation 9.15 gives:

$$(x_0, y_0) = \arg\min_{(x,y)} \left\| g[i,j] - \sum_{m,n} h_{(x,y)}[m,n] f[i-m, j-m] \right\|^2 \tag{9.56}$$

where $h_{(x,y)}[m,n]$ is the interpolation kernel for an offset of (x,y). Finding the offset that minimises this error requires a search because the filter coefficients, and hence the relationship between the two images, depends nonlinearly on the offset.

Predictive interpolation (Bailey and Lill, 1999) avoids the search by turning the problem around. Rather than use an interpolation kernel that makes arbitrary assumptions about the image (such as being band-limited or smooth), the optimal interpolation kernel is derived by predicting the pixel values of the target image from the reference (Bailey and Gilman, 2007; Gilman and Bailey, 2007). This solves:

$$g[i,j] - f[i,j] = \sum_{m,n \neq 0,0} h_{(x_0,y_0)}[m,n] \left(f[i-m, j-m] - f[i,j] \right) \tag{9.57}$$

where the subtraction of $f[i,j]$ enforces the partition of unity constraint (Unser, 2000)

$$\sum_{m,n} h[m,n] = 1 \tag{9.58}$$

Since Equation 9.57 is linear in the filter coefficients, it may be solved by least squares. Representing Equation 9.57 in matrix form:

$$\mathbf{g} = \mathbf{Fh} \tag{9.59}$$

where $\mathbf{h}$ are the filter coefficients arranged as a vector, and each line in $\mathbf{F}$ and $\mathbf{g}$ corresponds to one pixel in the image. The least squares solution becomes:

$$\mathbf{h} = \left(\mathbf{F}^{\mathrm{T}} \mathbf{F} \right)^{-1} \mathbf{F}^{\mathrm{T}} \mathbf{g} \tag{9.60}$$

Then, given the optimal interpolation filter, the problem becomes one of estimating the offset from the filter coefficients. This may be obtained from a weighted sum of the filter coefficients

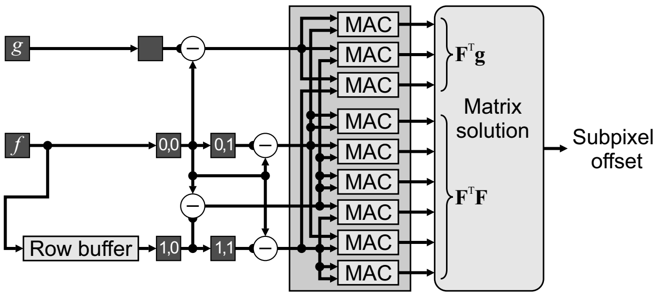


Figure 9.29 Predictive interpolation with a 2×2 window.

(Bailey and Gilman, 2007; Gilman and Bailey, 2007) with the weights dependent only on the coefficient index:

$$(x_0, y_0) = \sum_{m,n} \left(mh_{(x_0,y_0)}[m,n], nh_{(x_0,y_0)}[m,n] \right) \tag{9.61}$$

or

$$\mathbf{x}_0 = \mathbf{wh} = \mathbf{w}\left(\mathbf{F}^{\mathrm{T}}\mathbf{F}\right)^{-1}\mathbf{F}^{\mathrm{T}}\mathbf{g} \tag{9.62}$$

where $\mathbf{w}$ is the matrix of constant weights. Predictive interpolation therefore gives a direct (non-iterative) solution to estimating the offset between images to subpixel accuracy.

The cost of this approach is as follows: assume that the prediction window is $k \times k$ pixels, requiring k^2-1 filter coefficients to be calculated ($h[0,0]$ is not required). $\mathbf{F}^{\mathrm{T}}\mathbf{g}$ requires k^2-1 MACs and $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ requires $\frac{1}{2}k^2(k^2-1)$ (because of symmetry). This grows very quickly with window size, especially as windows with odd k do not perform as well (Bailey and Gilman, 2007). While only nine MAC units are required in total for a 2×2 window (Figure 9.29), this grows to 135 for 4×4, and 665 for 6×6. Most of the hardware is required for building $\mathbf{F}^{\mathrm{T}}\mathbf{F}$. To reduce the hardware, it is necessary to reduce the pixel clock and share the MACs. However, when registering multiple images, $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ and its inverse only need to be calculated once.

While optimal filtering provides the offset to subpixel accuracy, the images must already be registered to the nearest pixel (this can be relaxed slightly with larger windows). Again, a pyramidal coarse-to-fine search or other registration methods may be used to give the initial approximation. The optimal filtering approach is restricted to translation only, and cannot easily be extended to include more general transformations.

9.5.3 Applications

There are many applications of image registration or related techniques. The applications are quite diverse, with variations on some of the techniques described above used to address issues unique to that application. While it is not possible to describe all of the applications in any detail, a brief overview of some of the main applications is given.

Perhaps the most common is camera calibration, where it is desired to determine the geometric transformation which describes the imaging model of the camera, including its pose. To define the mapping between points within the real world, and pixels within the image, it is usually necessary to have a target object with distinctive features in known physical locations. This simplifies feature detection and the process becomes one of determining the camera model. Examples of a range of different models and methods of solving them are given elsewhere (Brown, 1971; Tsai, 1987; Heikkila and Silven, 1997; Devernay and Faugeras, 2001; Kannala and Brandt, 2004).

Image and video compression rely on reducing the redundancy inherent within an image or image sequence. When encoding a video sequence, there is significant correlation from one frame to the next that can be exploited to reduce the data volume. Therefore, rather than encode each image independently, considerable compression may be obtained by encoding the difference between the current and previous frame. This works well with static objects, but any motion creates a misalignment, reducing the gains. To achieve good compression, it is necessary to compensate for such motion. This is most commonly achieved by dividing the current frame into small blocks (typically 8×8 or 16×16) and finding the best offset between each block and the corresponding data in the previous frame. FPGAs have been the target for performing the matching at a rate that enables real-time compression. For example, Wong *et al.* (2002) developed a system that could perform the sum of absolute differences of a row or column of pixels in parallel. For each pair of pixels, this determined which of the corresponding pixels was smaller, so it could be inverted before being passed into an adder. All of the pixels were added, along with a correction term to convert one's complement to two's complement, by a single large adder tree. To reduce propagation delay, carry save adders were used, configured as a 33 to 2 reduction, followed by a final addition. To accelerate finding the best matched block, Dubois *et al.* (2005) built a system that could perform a random search with programmable block size. Yu *et al.* (2004) used a restricted two-step search to reduce the search space by a factor of 15 over a full search, enabling a reduced number of processing elements to be used.

Optical flow involves the estimation of motion fields within an image. This generally requires matching small patches from one frame with the previous frame. In this way it is similar to motion compensation for video coding. The difference is that with optical flow it is desired to have a coherent flow field, rather than treat each patch independently. FPGA implementations of optical flow have been reported by Diaz *et al.* (2004) and Wei *et al.* (2008).

Stereo imaging processes separate images from two spatially separated cameras to extract the disparity or differences in relative position of objects between the two images. The disparity is dependent on the range to the object of interest, enabling range data to be extracted. One of the motivating applications is the real-time extraction of range data for driver assistance and collision avoidance. The search for matches may be simplified with stereo imaging by exploiting epipolar geometry. An object point, when combined with the two camera centres, defines a plane in three-dimensional space. All points on that plane will be imaged to a single line in each of the captured images. Therefore, the search for corresponding points may be restricted to a one-dimensional search along these epipolar lines. This search process may be simplified by first rectifying the images to map the epipolar lines onto the image scanlines.

A wide range of stereo matching algorithms can be used – here is a sampling of recent papers which describe an FPGA implementation. Masrani and MacLean (2006) used correlation, with a multiresolution search centred on the disparity detected in the previous frame, working on the assumption that the disparity should not change significantly in one frame period. Longfield and Chang (2009) used a census transform to reduce the cost of finding a match. The census transform thresholds a local window with the central value, giving a binary template which may be matched more easily. Rather than find a local match, Morris *et al.* (2009) used dynamic programming to find a consistent set of matches across the whole row. Each potential match has a penalty given by the absolute difference in pixel values, with additional penalty for occlusion (a point visible in only one of the images).

Image super-resolution combines several low resolution images of a scene to create a single image with higher spatial resolution. Super-resolution reconstruction typically requires three steps: registration of the input images to subpixel accuracy; resampling the ensemble at the higher resolution; and inverse filtering to reduce the blur resulting from the low resolution imaging process. Again, there is a wide range of techniques used for image super-resolution. Many are iterative and not amenable to real-time implementation, although the algorithms may be accelerated using an FPGA as the computation engine. The optimal filtering techniques have potential, not only for image registration, but also for determining efficient resampling (interpolation) filters (Gilman, 2009; Gilman *et al.*, 2010). Little has been implemented so far on FPGAs. Perhaps one exception is the work of Bowen and Bouganis (2008), although their work assumed that the images were already registered and only considered the image combination stage.

10

Linear Transforms

Linear transformations are often used to rearrange the data into a particular form. In particular they can be used to separate different components of an image, for example separating signal from interference or noise. In general, with a linear transform, each output value is a linear combination of all of the input pixel values:

$$Q[u,v] = \sum_{x,y} w[u,v,x,y]I[x,y] \tag{10.1}$$

It extends and generalises local linear filtering of Equation 8.4 by removing the restriction of the window and allowing a more general selection of weights.

Direct implementation of Equation 10.1 is very expensive. If the input image is $N \times N$, then each output value requires N^2 multiplication and additions. For an $N \times N$ output, there are therefore N^4 operations. Many useful transforms are separable, in that they can be decomposed into separate, independent transforms on the rows and columns. In this case, Equation 10.1 simplifies to:

$$Q[u,v] = \sum_{y} w[v,y]\left(\sum_{x} w[u,x]I[x,y]\right) \tag{10.2}$$

reducing the number of operations to $2N^3$. Separable transforms can be represented in matrix form as:

$$\mathbf{Q} = \mathbf{wIw}^{\mathrm{T}} \tag{10.3}$$

where **w**, **I**, and **Q** are the weights and two-dimensional input and output images represented directly as matrices. All of the transforms considered in this chapter are separable, so only the detailed implementation of the one-dimensional transform will be considered. To implement the two dimensional transform, the architectures of Figures 9.5 and 9.6 can be used.

In many cases, the transform matrix, **w**, may be further factorised into a cascade of simpler operations. Such factorisations allow so-called fast transforms reducing the number of operations to order $N^2 \log N$.

If the rows of **w** are orthogonal and normalised so that the length is one:

$$\sum_{x} w[u_i,x]w[u_j,x] = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases} \tag{10.4}$$

then the transformation matrix is unitary and the transformation can be considered in terms of projecting the input image onto a new set of coordinate axes. The new axes may be thought of as basis images or component images, with the transformed values indicating how much of each basis image, $B_{u,v}[x, y]$, is in the original image. Thus:

$$I[x, y] = \sum_{u,v} B_{u,v}[x, y] Q[u, v] \tag{10.5}$$

In this context, Equation 10.1 is sometimes called the *analysis equation*, because it is analysing how much of each basis image is present, and Equation 10.5 is called the *synthesis equation*, because it is synthesising the image from the basis images.

10.1 Fourier Transform

The Fourier transform is one of the most commonly used linear transforms in image and signal processing. It transforms the input image from a function of position (x, y) to a function of spatial frequency (u, v) in the frequency domain:

$$\begin{aligned} w[u, v, x, y] &= \frac{1}{N} e^{-j2\pi(ux+vy)/N} \\ &= \left(\frac{1}{\sqrt{N}} e^{-j2\pi ux/N}\right)\left(\frac{1}{\sqrt{N}} e^{-j2\pi vy/N}\right) \\ &= w[u, x] w[v, y] \end{aligned} \tag{10.6}$$

The factor of $\frac{1}{\sqrt{N}}$ is often left out of the forward transform and $\frac{1}{N}$ is applied to the inverse transform. The Fourier transform is clearly separable, and is complex valued in the frequency domain. The periodic nature of the basis functions implies that the image is periodic in both space and frequency. That is:

$$\begin{aligned} w[-u, x] &= w[N-u, x] \\ &= w^*[u, x] \end{aligned} \tag{10.7}$$

This is a consequence of both the image and frequency domains being sampled (Bracewell, 2000). Although the origin in the frequency domain (corresponding to DC) is in the corner of the image, it is often shifted to the centre for display purposes by scrolling the image by $N/2$ pixels both horizontally and vertically. Equation 10.7 also implies that the Fourier transform of a real image is conjugate symmetric.

Two example images and their Fourier transforms are shown in Figure 10.1. On the top is a sinusoidal intensity pattern. Its Fourier transform has three peaks, one for the DC or average pixel value, and one for each of the positive and negative frequency components. This symmetry results from the Euler identity:

$$\begin{aligned} e^{j2\pi ux/N} &= \cos(j2\pi ux/N) + j\sin(j2\pi ux/N) \\ \cos(j2\pi ux/N) &= \frac{1}{2}\left(e^{j2\pi ux/N} + e^{-j2\pi ux/N}\right) \end{aligned} \tag{10.8}$$

In general, an image with a periodic spatial pattern will have a set of peaks in the frequency domain, corresponding to the frequency of the pattern along with its harmonics (Bailey, 1993; 1997b). The position of a peak in the frequency domain indicates the spatial frequency of the corresponding pattern in the image, with the frequency proportional to the distance from the origin and the direction of the peak from the origin corresponding to the direction of the sinusoidal variation. Rotating an object spatially will result in its Fourier transform rotating by the same angle. The magnitude of the peaks corresponds to the

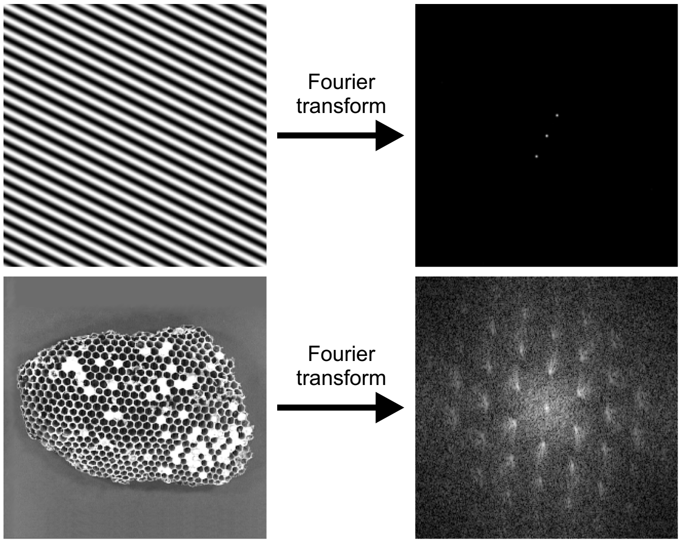


Figure 10.1 Example Fourier transforms: Top: sinusoid; bottom: semi-regular pattern, the magnitude in the frequency domain has been logarithmically transformed to show the detail more clearly.

amplitude of the pattern, and the phase of the frequency components contains information about the position of the pattern in the image (shifting an object will only change the phase of the corresponding frequency terms, not their amplitude). The shape of the peaks contains information about the shape and size of patterns in the image, and the distribution of the peaks relates to the details of the object. For example, sharp edges within the pattern will result in harmonics with amplitudes that drop in inverse proportion with frequency. In general, the amplitude of natural images in the frequency domain drops approximately in inverse proportion with frequency, primarily as a result of edges and other discontinuities within the images (Millane *et al.*, 2003). In contrast to this, spatially uncorrelated random noise has a uniform frequency distribution.

Some imaging modalities (for example magnetic resonance imaging and X-ray crystallography) capture their data in the frequency domain and it must be converted to the spatial domain for visualisation. In other applications, the desired image processing operation may be simpler to perform in the frequency domain, requiring the use of a forward and inverse Fourier transform to switch between the spatial and frequency domains.

10.1.1 Fast Fourier Transform

A *fast Fourier transform* (FFT) is an efficient implementation of the Fourier transform that results from the factorisation of the transformation. First define:

$$W_N = e^{-j2\pi/N} \tag{10.9}$$

Then, by splitting the input into the odd and even samples:

$$\begin{aligned} f_e[x_2] &= f[2x_2] \\ f_o[x_2] &= f[2x_2+1] \end{aligned} \tag{10.10}$$

the Fourier transform can be expanded as a sum of the Fourier transforms of the odd and even samples:

$$\begin{aligned}
F[u] &= \sum_{x=0}^{N-1} f[x] W_N^{ux} \\
&= \sum_{x_2=0}^{\frac{N}{2}-1} f[2x_2] W_N^{2ux_2} + \sum_{x_2=0}^{\frac{N}{2}-1} f[2x_2+1] W_N^{u(2x_2+1)} \\
&= \sum_{x_2=0}^{\frac{N}{2}-1} f_e[x_2] W_{\frac{N}{2}}^{ux_2} + W_N^u \sum_{x_2=0}^{\frac{N}{2}-1} f_o[x_2] W_{\frac{N}{2}}^{ux_2} \\
&= F_e[u] + W_N^u F_o[u]
\end{aligned} \tag{10.11}$$

A further simplification may be arrived at by considering periodicity and symmetry. Since the Fourier transform is periodic, $F_e\left[u+\frac{N}{2}\right] = F_e[u]$ and by symmetry $W_n^{u+\frac{N}{2}} = -W_n^u$. Therefore, the second half of the frequency samples become:

$$\begin{aligned}
F\left[u+\frac{N}{2}\right] &= F_e\left[u+\frac{N}{2}\right] + W_N^{u+\frac{N}{2}} F_o\left[u+\frac{N}{2}\right] \\
&= F_e[u] - W_N^u F_o[u]
\end{aligned} \tag{10.12}$$

which reuses the Fourier transforms of the first half, but with a change in sign. The W_N^u factors are sometimes called *twiddle factors*. By recursively applying this same decomposition to F_e and F_o, the $\log_2 N$ levels of the radix-2 decimation in time fast Fourier transform results. This is illustrated for $N = 8$ in Figure 10.2. The computation at each level is split into a series of 'butterfly' computations, which can be performed in place in a memory-based system.

The limitation of the decimation in time decomposition is that the input samples are in a bit-reversed order if applied in place. The operations can be rearranged so that the input operations are sequential, in which case the outputs are in a bit-reversed order.

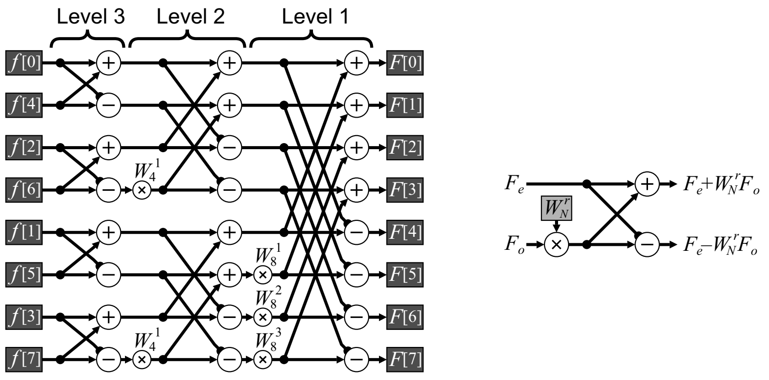


Figure 10.2 Radix-2 decimation in time FFT. Right: basic butterfly operation.

An alternative decomposition is decimation in frequency. Calculating the even and even frequency components separately gives:

$$\begin{aligned}F[2u] &= \sum_{x=0}^{N-1} f[x]W_N^{2ux} \\ &= \sum_{x=0}^{\frac{N}{2}-1} f[x]W_N^{2ux} + \sum_{x=0}^{\frac{N}{2}-1} f\left[x+\frac{N}{2}\right] W_N^{2u\left(x+\frac{N}{2}\right)} \\ &= \sum_{x=0}^{\frac{N}{2}-1}\left(f[x]+f\left[x+\frac{N}{2}\right]\right)W_{\frac{N}{2}}^{ux}\end{aligned} \tag{10.13}$$

and

$$\begin{aligned}F[2u+1] &= \sum_{x=0}^{N-1} f[x]W_N^{(2u+1)x} \\ &= \sum_{x=0}^{\frac{N}{2}-1} f[x]W_N^{2ux}W_N^{x} - \sum_{x=0}^{\frac{N}{2}-1} f\left[x+\frac{N}{2}\right] W_N^{2ux}W_N^{x} \\ &= \sum_{x=0}^{\frac{N}{2}-1}\left(f[x]-f\left[x+\frac{N}{2}\right]\right)W_N^{x}W_{\frac{N}{2}}^{ux}\end{aligned} \tag{10.14}$$

These are Fourier transforms of sequences which have half the length of the original sequence. Repeating this recursively gives the decimation in frequency algorithm illustrated in Figure 10.3. The input samples are ordered naturally, and the frequency samples use bit-reversed addressing as a consequence of separating the outputs in terms of odd and even components. The main differences in the calculation are the order of the groupings and that the twiddle factor is on the output of the butterfly rather than the input.

In many applications, the scrambling of the frequency samples is of little consequence. If an FFT is followed by a point operation in the frequency domain and then an inverse FFT, the frequency scrambling

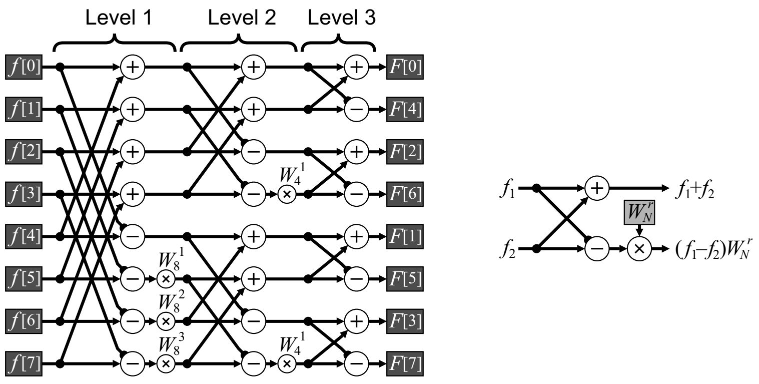


Figure 10.3 Radix-2 decimation in frequency FFT. Right: basic butterfly operation.

is of no consequence. Alternatively, if the transform is not performed in place, but goes from one memory to another, then the order of the samples may be rearranged during the process.

Note that all of the operations illustrated in Figures 10.2 and 10.3 are on complex numbers. In particular, each complex multiplication requires four real multiplications and two additions. However, through factorisation, the number of multiplications can be reduced to three at the cost of three more additions:

$$\begin{aligned}(R_1+jI_1)(R_2+jI_2) &= (R_1R_2-I_1I_2)+j(R_1I_2+I_1R_2)\\ &= (R_1(R_2-I_2)+(R_1-I_1)I_2)+j((R_1-I_1)I_2+I_1(R_2+I_2))\end{aligned} \tag{10.15}$$

Within the FFT, two of these additions are between constants, effectively trading one multiplication for an addition (see Figure 10.8).

Since each multiplication is effectively a rotation in the complex plane ($|W_N|=1$), the complete multiplication can be performed efficiently using CORDIC arithmetic (Despain, 1974; Sansaloni *et al.*, 2003). Efficiencies can be gained because the rotation angles are constants, so the decisions at each step of the CORDIC algorithm may be made in advance rather than having to wait for the signal to propagate through to the sign bit (the z_k register and logic are also eliminated, Section 5.4.3). However, compensated CORDIC is required to prevent the magnitude from changing as the complex number is rotated.

While the radix-2 algorithms (decomposing the transformation to a series of two point Fourier transforms) are the simplest to understand, the number of multiplications may be reduced by using a radix-4 algorithm. A four-point Fourier transform is:

$$\mathbf{F}=\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -j & -1 & j\\ 1 & -1 & 1 & -1\\ 1 & j & -1 & -j\end{bmatrix}\mathbf{f} \tag{10.16}$$

and the multiplication by j is trivial. The radix-4 decimation in frequency algorithm extends Equations 10.13 and 10.14 by splitting the input into four sections, combining them using Equation 10.16 and taking the $\frac{N}{4}$-point transforms of the results to get the frequency samples decimated by four:

$$\begin{aligned}F[4u] &= \sum_{x=0}^{\frac{N}{4}-1}\left(f[x]+f\left[x+\frac{N}{4}\right]+f\left[x+\frac{N}{2}\right]+f\left[x+\frac{3N}{4}\right]\right)W_{\frac{N}{4}}^{ux}\\ F[4u+1] &= \sum_{x=0}^{\frac{N}{4}-1}\left(f[x]-jf\left[x+\frac{N}{4}\right]-f\left[x+\frac{N}{2}\right]+jf\left[x+\frac{3N}{4}\right]\right)W_N^{x}W_{\frac{N}{4}}^{ux}\\ F[4u+2] &= \sum_{x=0}^{\frac{N}{4}-1}\left(f[x]-f\left[x+\frac{N}{4}\right]+f\left[x+\frac{N}{2}\right]-f\left[x+\frac{3N}{4}\right]\right)W_N^{2x}W_{\frac{N}{4}}^{ux}\\ F[4u+3] &= \sum_{x=0}^{\frac{N}{4}-1}\left(f[x]+jf\left[x+\frac{N}{4}\right]-f\left[x+\frac{N}{2}\right]-jf\left[x+\frac{3N}{4}\right]\right)W_N^{3x}W_{\frac{N}{4}}^{ux}\end{aligned} \tag{10.17}$$

The corresponding ‘butterfly’ is shown in Figure 10.4. Although such decimation can be extended to higher radixes, little is gained beyond radix-4.

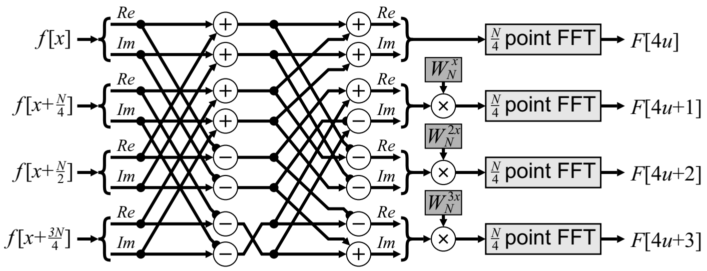


Figure 10.4 Radix-4 decimation in frequency butterfly with the real and imaginary components shown explicitly.

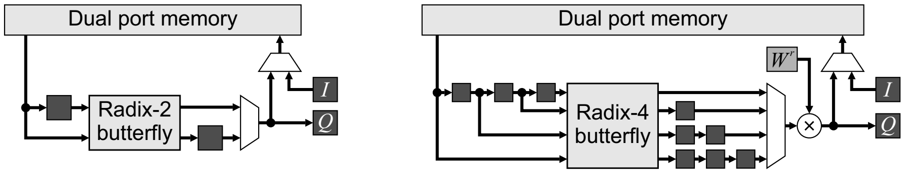


Figure 10.5 FFT with a single butterfly. Left: radix-2; right: radix-4 FFT.

There are several ways in which the FFT may be implemented given the above building blocks. If resources are scarce, a single butterfly unit may be built, with the transformation performed in place in memory (Figure 10.5). The data is first streamed into memory (performing any permutation as necessary). The data is then accessed from memory as needed for each butterfly operation, with the results written back into memory. Bandwidth limitations limit the speed to one butterfly every two clock cycles for the radix-2 and one butterfly every four clock cycles for the radix-4 operation. With the radix-4 butterfly, the multiplier can be shared by moving it from within the butterfly to after the multiplexer, as shown in Figure 10.5 (Sansaloni *et al.*, 2003). With the radix-4 butterfly, the number of adders may also be reduced by multiplexing the input layer of Figure 10.4. The throughput (and latency) may be improved by splitting the memory into multiple blocks (two for radix-2 and four for radix-4) to enable one operation to be performed every clock cycle. The memory addressing becomes considerably more complex because to ensure that the data will be available when it is needed, the transformation cannot be performed in place.

The memory addressing required by the scheme in Figure 10.5 is relatively straightforward. A standard counter may be used, with the address bits permuted depending on the FFT level. The read address for the radix-2 decimation in frequency FFT is shown in Figure 10.6. The write counter is the same, only delayed by a few clock cycles depending on the latency of the butterfly and associated registers. The address counters are similar for the other FFT implementations based on sharing a single butterfly.

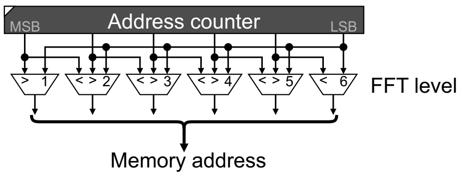


Figure 10.6 Memory read address counter for radix-2 decimation in frequency FFT.

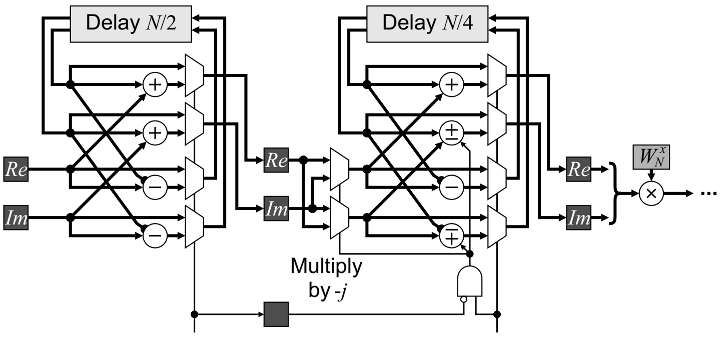


Figure 10.7 Pipelined radix-2^2 implementation. Shown are the first two stages. This structure is repeated with successive stages alternating.

Uzun *et al.* (2003; 2008) implemented several different butterfly implementations within a generic framework using Handel-C. The speed and resources were compared between the implementations for a number of butterfly units. Not surprisingly, the radix-4 algorithm was the fastest, but also consumed the most resources.

If more resources are available, multiple butterflies may be implemented and the FFT pipelined to increase the throughput. A range of radix-2 and radix-4 pipelined FFT approaches have been reviewed by He and Torkelson (1996). One of the more efficient pipelined implementations is the radix-2^2 scheme illustrated in Figure 10.7. This uses a radix-2 butterfly with the stages in pairs effectively forming a radix-4 transform. It retains the simplicity of the reduced number of multipliers as the radix-4 algorithm, with a single complex multiplier only required after every second stage. At each stage, the first half of the clock cycles feed the data through into the delay register without processing. The second half combines the incoming data with the delayed data, performing the butterfly. One of the butterfly outputs is fed directly out to the next stage, while the other is fed back into the delay. It is finally output while the next block is being loaded. Successive stages reduce the delay by a factor of two to give the interleaving as illustrated in Figure 10.3. Control consists of the successive bits of a binary counter (delayed as necessary to account for any pipeline delays). The pipelined FFT takes streamed data as input and produces a streamed output (in bit-reversed order). The throughput of these arrangements is one clock cycle per sample, with a latency of one row. FPGA-based implementations of this scheme have been described by Sukhsawas and Benkrid (2004) and Saeed *et al.* (2009).

To perform an inverse FFT, the inverse transform matrix is the complex conjugate of the forward transform. This means that the same hardware may be used, with the exception that the complex conjugate of the twiddle factors is used. Alternatively, the inverse FFT may be calculated by taking the complex conjugate of the frequency domain signal and using a standard forward FFT. With either case the result needs to be divided by N to obtain the original signal.

While an FFT is mathematically invertible, with inevitable round-off errors in twiddle factors and truncating the results of the multiplication this mean that the result is only an approximation of the Fourier transform. Consequently, taking the FFT and then the inverse FFT will not necessarily give exactly the same result. The transform may be made invertible by using a lifting scheme to implement the twiddle factor multiplications (Oraintara *et al.*, 2002). The lifting scheme is compared with conventional complex multiplication and the factorised multiplication of Equation 10.15 in Figure 10.8. The two alternatives for the lifting scheme enable the absolute value of the coefficients to be kept less than one (the final multiplication is a trivial sign change). One disadvantage of using the lifting scheme is that the multiplications are performed in series rather than in parallel as with the other two methods, increasing the latency.

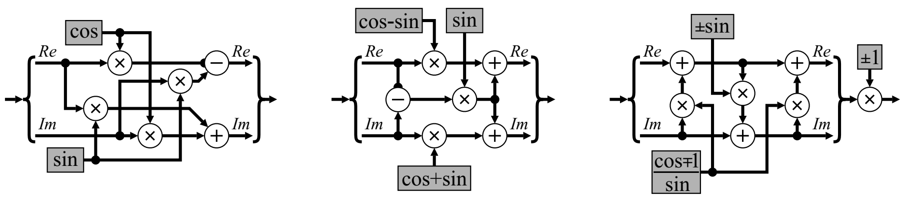


Figure 10.8 Complex multiplication. Left: conventional multiplication; centre: factorised implementation using Equation 10.15; right: implemented using a lifting scheme.

Although the lifting scheme still suffers from round-off and truncation errors, the errors are reversible if the result of the multiplication is quantised before the associated addition. Running the system in reverse, replacing the additions with subtractions, will give the original result provided that an identical quantisation is performed.

To calculate the FFT of a two-dimensional image the Fourier transform can be taken of each of the rows, and then of the columns of the result. Since each of the row transforms is of independent data, the two-dimensional transform may be accelerated by performing multiple row transforms in parallel through hardware duplication. Similarly, the column transforms may also be performed in parallel.

With real data, additional efficiencies may be gained by performing two FFTs at once. The Fourier transform of a real signal is conjugate symmetric, that is the real parts of the Fourier transform are even and the imaginary parts are odd. This allows a single FFT to take the Fourier transform of two sequences simultaneously. Let two separate rows be $f_1[x]$ and $f_2[x]$. A composite signal is formed by combining one signal in the real part and the other in the imaginary:

$$g[x] = f_1[x] + jf_2[x] \tag{10.18}$$

The FFT is taken of the composite signal and the FFT of the individual components may be extracted as:

$$\begin{aligned} F_1[u] &= \frac{G[u] + G^*[-u]}{2} = \frac{G[u] + G^*[N-u]}{2} \\ F_2[u] &= \frac{G[u] - G^*[-u]}{2j} = \frac{G[u] - G^*[N-u]}{2j} \end{aligned} \tag{10.19}$$

This is effectively another level of butterfly operations, although with a different pattern. Similarly, when performing the inverse FFT of a real image, two conjugate symmetric Fourier transforms may be combined as:

$$G[u] = F_1[u] + jF_2[u] \tag{10.20}$$

before calculating the inverse.

When calculating the Fourier transform of a two-dimensional image, symmetry may also be exploited when taking the Fourier transform of the columns. Columns 0 and $\frac{N}{2}$ will only have real data, so may be transformed together with a single FFT. Columns u and $N-u$ will be complex conjugates of one another, so will give related Fourier transforms:

$$F[u, v] = F^*[-u, -v] = F^*[N-u, N-v] \tag{10.21}$$

Therefore, only one of these needs to be calculated. Exploiting the symmetries of both row and column transforms reduces the overall computation for the two-dimensional FFT of a real image by a factor of four.

When performing a two-dimensional FFT, a one-dimensional FFT must be performed on all of the rows (or columns) before repeating the FFT in the other direction. This requires storing the frequency domain image into a frame buffer before performing the column FFTs. Processing a streamed input, and assuming that only half of the frequency domain needs to be calculated, the latency of performing a two-dimensional FFT is approximately one half of a frame time (from the time the last pixel is input to when the last pixel is output). Many operations (filtering for example) require only a point operation in the frequency domain. Therefore, taking the inverse FFT of the columns can be pipelined with the forward FFT. Again, all of the columns must be completed before taking the inverse FFT of the rows. Using Equation 10.20 to take the inverse of two rows at once will give an overall latency of frequency domain processing of just over one frame time.

The assumption throughout this section has been that N is a power of two. While other factorisations of N can also result in fast algorithms, they lack the elegance and computational simplicity of those described above for powers of two. Therefore, unless there is a particular need for a special size, the complexity of developing the circuitry for such factorisations is generally not worthwhile. An alternative is to express the Fourier transform in terms of a convolution, which can be calculated using a power of two FFT (Bergland, 1969). In one dimension:

$$\begin{aligned} F[u] &= \sum_{x=0}^{N-1} f[x] W_N^{ux} \\ &= \sum_{x=0}^{N-1} f[x] W_N^{ux+(u^2-u^2+x^2-x^2)/2} \\ &= W_N^{u^2/2} \sum_{x=0}^{N-1} \left(W_N^{x^2/2} f[x] \right) W_N^{-(u-x)^2/2} \end{aligned} \tag{10.22}$$

The summation is now a convolution, which may be performed by using an FFT of length $\tilde{N} \geq 2N$ of each of the two sequences, taking their product and calculating the inverse FFT of the result. While this requires calculating three longer FFTs, in many circumstances this is still preferable to calculating the Fourier transform directly (especially if N is prime and cannot be factorised).

10.1.2 Filtering

Linear filters are implemented as a convolution in the image domain, using Equation 8.4. In the frequency domain, this convolution becomes a product:

$$Q[u,v] = W[u,v] I[u,v] \tag{10.23}$$

where $W[u,v]$ is the Fourier transform of the flipped filter kernel. Filtering may therefore be performed in the frequency domain. For large kernel sizes, it is less expensive computationally to take the Fourier transform (using an FFT), multiply by $W[u,v]$ and take the inverse Fourier transform of the result.

The weighting in the frequency domain of Equation 10.23 implies that a linear filter may, therefore, be considered in terms of its *frequency response*, as a weighting of the different frequency components within the image. In this context, a low pass filter will tend to smooth noise, because it will pass low frequencies with little attenuation, and many images have most of their energy in the low frequency. Uncorrelated noise has uniform frequency content, so attenuating the high frequencies will have only a small effect on the image, but a more significant effect on the noise. The loss of high frequencies in the image will result in a loss of fine detail which will, in general, blur the image. Edges and fine detail may be enhanced by boosting the amplitudes of the higher frequencies, but this will also tend to amplify the noise within the image.

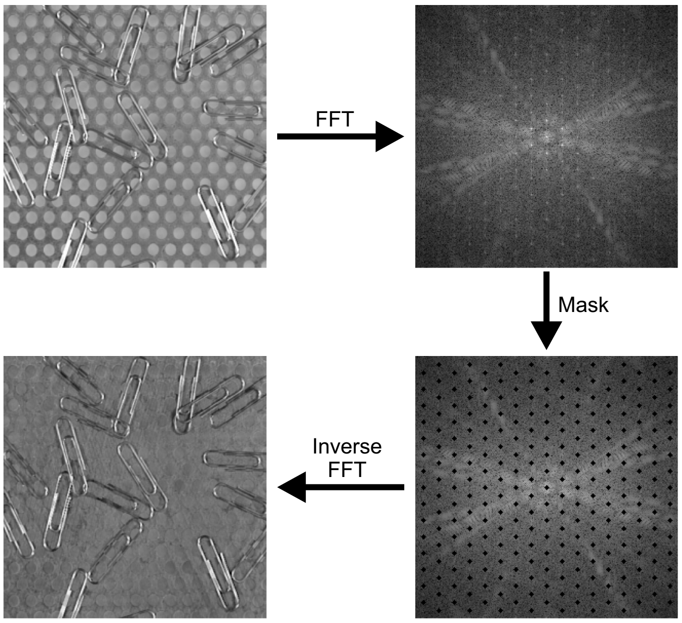


Figure 10.9 Filtering to remove pattern noise. The frequency domain is shown logarithmically scaled for clarity.

One particular application of filtering in the frequency domain is to remove pattern noise from an image. As described earlier, a regular pattern will exhibit a series of peaks within the frequency domain. These peaks may be detected directly in the frequency domain and masked out, as demonstrated in Figure 10.9. If the pattern is added within the image, then filtering can remove the pattern. Here the pattern is part of the background, so removing the pattern will also affect the object itself.

The opposite of removing pattern noise is filtering to detect regular patterns within the image. In this case, it is the periodic peaks within the frequency domain that contain the information of interest. One approach to enhancing the regular patterns is to use the image itself as the filter (Bailey, 1993; 1997b). Multiplication by the magnitude in the frequency domain:

$$W[u, v] = |I[u, v]| \tag{10.24}$$

is a zero phase filter, so objects are not shifted. Self-filtering is demonstrated in Figure 10.10. The basic self-filter will blur sharp edges, which have frequency content that rolls off inversely proportional

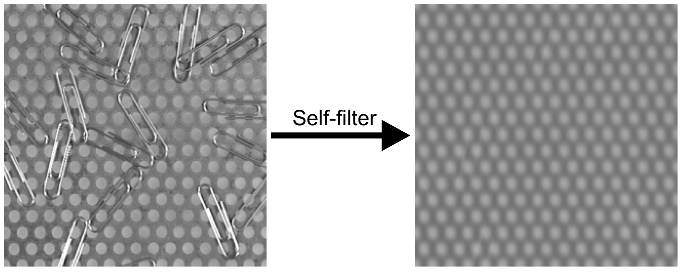


Figure 10.10 Self-filtering in the frequency domain using Equation 10.24.

to frequency. This may be compensated by weighting the filter with frequency to reduce the attenuation of the harmonics:

$$W[u,v] = \sqrt{u^2+v^2}|I[u,v]| \tag{10.25}$$

In the example in Figure 10.9, the frequency domain was simply masked. This will also remove those frequency components from the objects of interest. An alternative is to design the filter such that it minimises the error after filtering. Consider an image, $f[x,y]$, which has been corrupted by the addition of noise, $n[x,y]$ (uncorrelated with the input image). The *Wiener filter* is the optimal filter (in the least squares sense) for recovering the original signal. The analysis is easiest to perform in the frequency domain: the goal is to determine the filter $W[u,v]$ that minimises the error:

$$\begin{aligned} E^2 &= \sum_{u,v} \|(F[u,v]+N[u,v])W[u,v]-F[u,v]\|^2 = \sum_{u,v} \|(F+N)W-F\|^2 \\ &= \sum_{u,v} \|F(W-1)-NW\|^2 \\ &= \sum_{u,v} (F(W-1)-NW)(F(W-1)-NW)^* \end{aligned} \tag{10.26}$$

This can be simplified by making the assumption that the filter is symmetric so that it does not shift the image. Therefore, W is real, resulting in:

$$E^2 = \sum_{u,v} \left(FF^*(W-1)^2 + NN^*W^2\right) + \sum_{u,v} \left(FN^* + NF^*\right)W(W-1) \tag{10.27}$$

The expected value of the second summation is zero if the image and signal are uncorrelated. Therefore:

$$E^2 = \sum_{u,v} \|F\|^2\left(W^2-2W+1\right) + \|N\|^2W^2 \tag{10.28}$$

The error may be minimised by taking the derivative with respect to the filter:

$$\frac{dE^2}{dW} = \sum_{u,v} \left(2W\left(\|F\|^2 + \|N\|^2\right) - 2\|F\|^2\right) = 0 \tag{10.29}$$

The corresponding optimal filter is, therefore:

$$W[u,v] = \frac{\|F[u,v]\|^2}{\|F[u,v]\|^2 + \|N[u,v]\|^2} \tag{10.30}$$

Given estimates of both the signal and noise spectra, the frequency response of the optimal filter can be derived. If the noise is uncorrelated, then it will have a flat spectrum and will be a constant.

Equation 10.30 may be interpreted as follows: at the frequencies where the signal is much larger than the noise, the frequency response of the filter will approach unity, passing the signal through. However, the frequencies at which the signal is smaller than the noise will be attenuated, effectively reducing the noise power while having little effect on the total signal. The Wiener filter is effective are removing both random noise as well as pattern noise.

10.1.3 Inverse Filtering

Closely related to filtering is inverse filtering. Here the goal is to remove the effects of filtering from an image. A common application is removing blur, whether from the lens point spread function, or motion blur from a moving object or camera. Let $b[x,y]$ be the blur point spread function, with corresponding Fourier transform, $B[u,v]$. Also, let the desired unblurred image be $f[x,y]$. In the frequency domain the blurred image is given by:

$$G[u,v] = F[u,v]B[u,v] \tag{10.31}$$

Assuming that the blur function is known, then the deblurred image should be able to be recovered as:

$$\hat{F}[u,v] = \frac{G[u,v]}{B[u,v]} \tag{10.32}$$

Unfortunately, this does not work. The output image from Equation 10.32 appears to contain only random noise. The reason is that $B[u,v]$ goes to zero at some frequencies. This division by zero is undefined. Even if B does not actually go to zero at any of the samples, the scaling of the corresponding frequencies becomes arbitrarily large. For noiseless, infinite precision arithmetic, this would not pose a problem. However, $G[u,v]$ will inevitably have some level of noise added. Applying the inverse filter according to Equation 10.32 will cause the noise to dominate the output.

One solution to this problem is to manipulate Equation 10.32 to avoid division by zero:

$$\hat{F}[u,v] = G[u,v]\frac{1}{B[u,v]} = G\frac{B^*}{\|B\|^2} \approx G\frac{B^*}{\|B\|^2+k^2} \tag{10.33}$$

where the first transformation is to make the denominator positive real and the second adds a positive constant, k^2, in the denominator to prevent division by zero. This effectively limits the gain of the noise. The Wiener filter, which minimises the mean square error, gives the inverse filter as:

$$W[u,v] = \frac{B^*[u,v]\|F[u,v]\|^2}{\|B[u,v]\|^2\|F[u,v]\|^2 + \|N[u,v]\|^2} = \frac{B^*[u,v]}{\|B[u,v]\|^2 + \dfrac{\|N[u,v]\|^2}{\|F[u,v]\|^2}} \tag{10.34}$$

which sets k^2 optimally according to the signal-to-noise ratio at each frequency.

More complex techniques (usually iterative) are required when the blur function is not known. These are beyond the scope of this book.

10.1.4 Interpolation

Images may be interpolated by using the frequency domain. If it is assumed that the image is band-limited (and sampled according to the Nyquist sampling criterion), then extending the image in the frequency domain is trivial because the additional higher frequency samples will be zero. Therefore, an image may be interpolated by taking its Fourier transform, padding with zeros and taking the inverse Fourier transform.

Since padding will make the frequency domain image larger, taking the inverse Fourier transform will take longer. This approach is equivalent to sinc interpolation of the input image. However, in many imaging applications, the input image is not band-limited, therefore sinc interpolation will not necessarily give the best results. Interpolating directly in the image domain (Section 9.3) is usually more appropriate.

If multiple, slightly offset, images are available, then a simple approach to image super-resolution is to use the information gleaned from the multiple images to resolve the aliasing, giving an increased resolution (Tsai and Huang, 1984). As a result of aliasing, the value at each frequency sample will result from a mixture of frequencies:

$$F[u,v] = \sum_{k,l} \tilde{F}[u \pm kN, v \pm lN] \tag{10.35}$$

where $\tilde{F}$ are the samples of the continuous frequency distribution of the original unsampled image. Normally, once the frequency information has been aliased, it is impossible to distinguish between the different frequency components that have been added together. However, when an image is offset, the different aliased components will have a different phase shift (since the phase shift is proportional to frequency). Therefore, given multiple offset images, it is possible to untangle the aliasing. Consider enhancing the resolution by a factor of two. Then each image may be considered as:

$$\begin{aligned} F_i[u,v] \approx c\tilde{F}[u,v]e^{j\theta_{i,0,0}} + \tilde{F}[N-u,v]e^{j\theta_{i,1,0}} \\ +\tilde{F}[u,N-v]e^{j\theta_{i,0,1}} + \tilde{F}[N-u,N-v]e^{j\theta_{i,1,1}} \end{aligned} \tag{10.36}$$

where it is assumed that the additional higher order alias terms are negligible. The phase terms are determined directly from the offsets of each of the images (for example, using any of the techniques of Section 9.5) and are effectively known (or can be estimated). Given four such images, then in matrix form:

$$\begin{bmatrix} F_1[u,v] \\ F_2[u,v] \\ F_3[u,v] \\ F_4[u,v] \end{bmatrix} = \begin{bmatrix} e^{j\theta_{1,0,0}} & e^{j\theta_{1,1,0}} & e^{j\theta_{1,0,1}} & e^{j\theta_{1,1,1}} \\ e^{j\theta_{2,0,0}} & e^{j\theta_{2,1,0}} & e^{j\theta_{2,0,1}} & e^{j\theta_{2,1,1}} \\ e^{j\theta_{3,0,0}} & e^{j\theta_{3,1,0}} & e^{j\theta_{3,0,1}} & e^{j\theta_{3,1,1}} \\ e^{j\theta_{4,0,0}} & e^{j\theta_{4,1,0}} & e^{j\theta_{4,0,1}} & e^{j\theta_{4,1,1}} \end{bmatrix} \begin{bmatrix} \tilde{F}[u,v] \\ \tilde{F}[N-u,v] \\ \tilde{F}[u,N-v] \\ \tilde{F}[N-u,N-v] \end{bmatrix} \tag{10.37}$$

This equation may be inverted to resolve the aliased higher frequency components. (If more images are available, a least squares inverse may be used to improve the estimate.) By padding the image with these components and taking the inverse Fourier transform, the resolution may be improved.

10.1.5 Registration

As outlined in the previous chapter, images may be registered in the frequency domain. This requires taking the Fourier transform of the two input images being registered. Using Equations 10.18 and 10.19, these may be calculated in parallel using a single FFT. CORDIC arithmetic can then be used to obtain the magnitude and phase angle of each frequency component. To estimate the offset, it is necessary to determine the slope of the phase difference. The problem here is that the arctangent reduces the phase to within $\pm\pi$, requiring the phase to be unwrapped.

In many applications it can be assumed that the phase varies slowly with frequency. The simplest approach to phase unwrapping is to consider the phase of adjacent samples. Multiples of 2π are then added to one of the samples until the absolute difference is less than π. In two dimensions, this may be applied both horizontally and vertically, providing additional constraints.

One limitation with this approach is that, at some frequencies, the amplitude changes sign (consider for example a sinc function). The phase shift between such points should be close to π rather than zero. A second limitation is that when the magnitude is low, the phase angle becomes dominated by noise. Consequently, only the phases for the low frequencies are reliably unwrapped unless more complex methods are used.

In the case of image registration, the phase difference should be planar. One approach to this is to fit a plane to the phase difference. An alternative is to consider the phase shear (Stowers *et al.*, 2010). From Equation 9.51:

$$ux_0 + vy_0 = \frac{1}{2\pi}\angle\big(F^*[u,v]G[u,v]\big) \tag{10.38}$$

Let $H[u,v] = F^*G$. Then the phase difference between horizontally adjacent frequencies should be:

$$\begin{aligned} x_0 &= \frac{1}{2\pi}(\angle H[u+1,v] - \angle H[u,v]) \\ &= \frac{1}{2\pi}\angle(H[u+1,v]H^*[u,v]) \end{aligned} \tag{10.39}$$

An amplitude weighted average of these is given by (Stowers *et al.*, 2010):

$$x_0 = \frac{1}{2\pi}\angle\sum_{u,v} H[u+1,v]H^*[u,v] \tag{10.40}$$

and similarly for the vertical offset. The amplitude weighting will automatically give low weight to those differences where the phase shifts by π, and the problem of phase wrapping is avoided by deferring the arctangent to after the averaging.

Alternatively, histograms of the horizontal and vertical offsets estimated by Equation 10.39 may be built to enable outliers to be eliminated before averaging.

10.1.6 Feature Extraction

The frequency domain can provide a rich source of features, particularly in representing the texture within an image. The example in Figure 10.11 shows an electron micrograph of the surface of a pollen grain. From the frequency domain, the dominant frequency and angle variation may be easily measured (Currie, 1995).

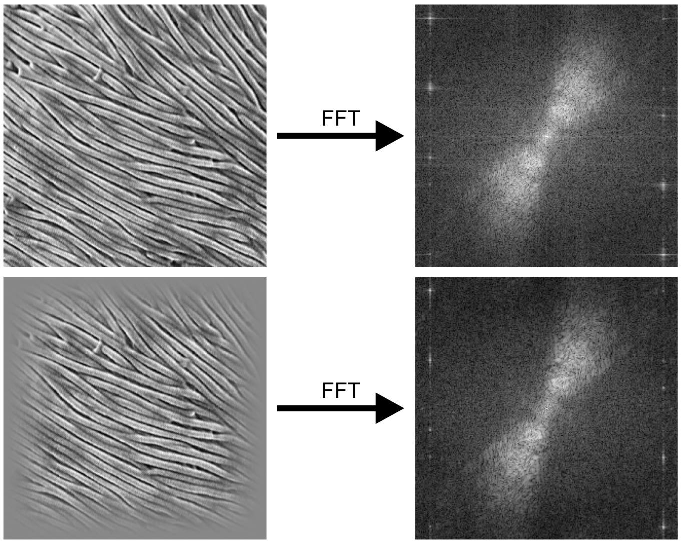


Figure 10.11 Frequency domain features: dominant frequency and angle distribution. Top: taking the Fourier transform directly of the image; bottom: windowing the image first.

The top row of Figure 10.11 shows one of the limitations of using the discrete Fourier transform (calculated via the FFT). Sampling in the frequency domain implicitly assumes that the image is periodic. However, the pattern on the left and right edges of the image do not line up. These discontinuities result in *spectral leakage*, the manifestation of which is the streaked appearance within the frequency domain.

Spectral leakage effects may be reduced by multiplying the input image by a window or *apodization function*, which smoothly tapers to zero at the edges of the image:

$$\widehat{f}[x,y] = f[x,y]w[x,y] \tag{10.41}$$

This product becomes a convolution of Fourier transforms, so the default rectangular window will convolve the true frequency content with a sinc function. It is the relatively high sidelobes of the sinc function that results in spectral leakage, and the streaking and smearing in the frequency domain. The consequences of using a windowing function are a reduction in the level of the sidelobes at the expense of an increased main lobe. The reduction in side lobes directly reduces the spectral leakage, but the wider main lobe results in reduced frequency resolution from the blurring within the frequency domain (Harris, 1978).

The window function in the lower image in Figure 10.11 was a Tukey or tapered cosine window with $\alpha = 0.25$:

$$\begin{aligned} w[x,y] &= w[x]w[y] \\ w[x] &= \begin{cases} \frac{1}{2}\left(1+\cos\pi\left(\frac{2x}{\alpha N}-1\right)\right), & 0 \le x < \frac{\alpha N}{2} \\ 1, & \frac{\alpha N}{2} \le x \le \frac{(2-\alpha)N}{2} \\ \frac{1}{2}\left(1+\cos\pi\left(\frac{2x}{\alpha N}-\frac{2-\alpha}{\alpha}\right)\right), & \frac{(2-\alpha)N}{2} < x < N \end{cases} \end{aligned} \tag{10.42}$$

This window provides a compromise between maintaining good frequency resolution while reducing spectral leakage.

For an FPGA implementation, the window function may be stored in a lookup table, and multiplied by the incoming image as the first step before calculating the FFT. Due to symmetry, only $N/2$ entries are required in the table. For the 25% Tukey window, this is reduced to $N/8$ entries.

10.1.7 Goertzel's Algorithm

In some applications, only one or a few frequencies are of interest. In this case, Goertzel's algorithm (Goertzel, 1958) can require considerably fewer calculations than performing the full Fourier transform. Consider a single frequency component of the Fourier transform:

$$\begin{aligned} F_u &= \sum_{x=0}^{N-1} f[x]W_N^{ux} = \sum_{x=0}^{N-1} f[x]\left(W_N^{-u}\right)^{(N-x)} \\ &= \left(\sum_{x=0}^{N-2} f[x]\left(W_N^{-u}\right)^{(N-x)} + f[N-1]\right)W_N^{-u} \end{aligned} \tag{10.43}$$

This is effectively a recursive calculation:

$$F_u[n] = (F_u[n-1] + f[n])W_N^{-u} \tag{10.44}$$

It still requires one complex multiplication per sample, so does not gain anything computationally. Looking at this recursion in the z-domain:

$$\begin{aligned}\frac{F_u(z)}{f(z)} &= \frac{W_N^{-u}}{1-W_N^{-u}z^{-1}}\\ &= \frac{W_N^{-u}}{1-W_N^{-u}z^{-1}}\left(\frac{1-W_N^{u}z^{-1}}{1-W_N^{u}z^{-1}}\right)\\ &= \frac{1}{1-2\cos\frac{2\pi u}{N}z^{-1}+z^{-2}}\left(W_N^{-u}-z^{-1}\right)\end{aligned} \tag{10.45}$$

The multiplication of the top and bottom by the complex conjugate of the denominator makes the denominator coefficients real. Therefore, the first term can be implemented as a second order recursion with real coefficients:

$$S_u[n] = f[n] + 2\cos\frac{2\pi u}{N}S_u[n-1] - S_u[n-2] \tag{10.46}$$

and after the N samples have accumulated, the frequency term may be calculated from the second term of Equation 10.45 as:

$$F_u = \left(S_u[N-1]\cos\frac{2\pi u}{N} - S_u[N-2]\right) + jS_u[N-2]\sin\frac{2\pi u}{N} \tag{10.47}$$

An implementation of this is shown in Figure 10.12. Note that the intermediate registers need to be reset to zero before the start of the calculation. The filter processes streamed data and produces the frequency value after N input samples. N is not restricted to a power of two – the implementation of the algorithm is independent of the number of samples (apart from the constant multipliers). If multiple frequencies must be calculated, then this circuit can be duplicated for each frequency.

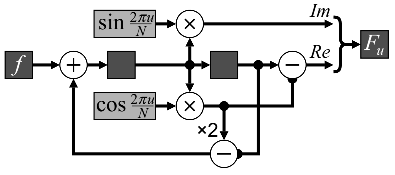


Figure 10.12 Implementation of Goertzel's algorithm.

10.2 Discrete Cosine Transform

Closely related to the Fourier transform is the discrete cosine transform (DCT). Whereas the Fourier transform has both sine and cosine components, the DCT is made purely from cosine terms. This is achieved by enforcing even symmetry and enables the transform to be calculated with real rather than complex numbers.

Mathematically, in one dimension the discrete cosine transform is:

$$F[u] = \sum_{x=0}^{N-1}\sqrt{\frac{\alpha}{N}}f[x]\cos\left(\frac{\pi u\left(x+\frac{1}{2}\right)}{N}\right), \quad \alpha = \begin{cases}1, & u=0\\ 2, & \text{otherwise}\end{cases} \tag{10.48}$$

or in terms of Equation 10.2:

$$w[u,x] = \sqrt{\frac{\alpha}{N}}\cos\left(\frac{\pi u\left(x+\frac{1}{2}\right)}{N}\right) \tag{10.49}$$

The inverse is simply $\mathbf{w}^T$ or:

$$f[x] = \sum_{u=0}^{N-1}\sqrt{\frac{\alpha}{N}}F[u]\cos\left(\frac{\pi u\left(x+\frac{1}{2}\right)}{N}\right) \tag{10.50}$$

Being separable, a two-dimensional DCT may be calculated by taking the one-dimensional transform of the rows followed by the one-dimensional transform of the columns. While algorithms for direct computation of the two-dimensional transform can be developed that require fewer arithmetic operations than the separable transform (Feig and Winograd, 1992), the separable algorithm allows hardware to be reused and results in a simpler implementation for streamed data.

Since it is closely related to the Fourier transform, there are also fast algorithms for calculating the DCT, especially for powers of two. In fact, the DCT can be calculated using an FFT of length $4N$, with the data in the odd terms (mirroring the data to maintain even symmetry) and setting the even terms to zero. The appropriate simplifications of the resulting FFT (removing unnecessary calculations) lead to efficient algorithms for the DCT. Algorithms are discussed here for $N = 8$, since this is the size most commonly used for image and video coding. The most efficient algorithm for an eight-point DCT requires 11 multiplications (Loeffler *et al.*, 1989) and is shown in Figure 10.13.

If a scaled DCT is acceptable, the number of multiplications can be reduced to five (Kovac and Ranganathan, 1995) (Figure 10.14). With a scaled DCT, each frequency coefficient is scaled by an arbitrary factor. When using the DCT for image compression, the coefficients are quantised. The scale factor can be incorporated into the quantisation step without incurring any additional operations simply by scaling the quantisation step size by scale factor. Agostini *et al.* (2001; 2005) have implemented this on an FPGA for JPEG image compression. Their implementation had a throughput of one pixel per clock cycle but with a latency of 48 clock cycles because they divided the algorithm into six blocks to share hardware. The structure of the DCT is less regular than the FFT, making an elegant pipeline less practical. However, one attempt at pipelining the scheme of Figure 10.14 is shown in Figure 10.15. Attempts have been made to reuse hardware where practical, particularly the multiplier. This design maintains a throughput of one pixel per clock cycle and has a latency of only nine clock cycles.

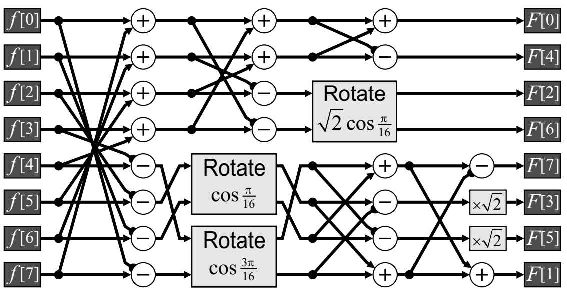


Figure 10.13 DCT with 11 multiplications. The Rotate box is a scaled rotation requiring three multiplications and additions using Equation 10.15.

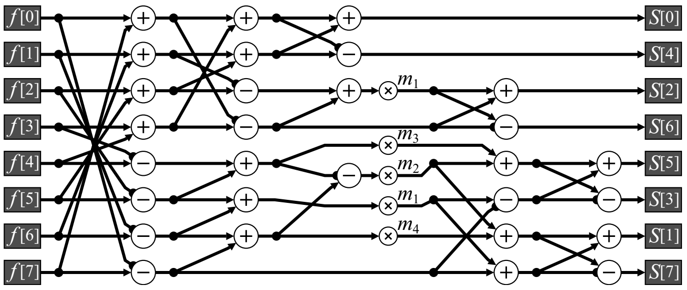


Figure 10.14 Scaled DCT (Kovac and Ranganathan, 1995), where $S[u] = 4\cos(u\pi/16) \times F[u]$. The fixed multipliers are $m_1 = \cos(2\pi/8)$, $m_2 = \cos(3\pi/8)$, $m_3 = \cos(\pi/8) - \cos(3\pi/8)$, $m_4 = \cos(\pi/8) + \cos(3\pi/8)$.

The first four samples are stored in a fabric RAM and are read out in the next four samples to be combined with the input. Using a RAM reduces the need for explicit multiplexing. Similarly, the constant multipliers can also be stored in a fabric RAM, to enable sequential access. The assumption made here is that the multiplication can be completed in a single clock cycle. If not, the multiplier will need to be pipelined and additional delays inserted in the corresponding parallel paths. The scaled outputs are not in natural order; for JPEG compression, this does not matter because the data will be reordered later using a zigzag buffer.

Modern FPGAs have plentiful multipliers. This enables a far simpler and more elegant pipelined design, as demonstrated in Figure 10.16. It simply multiplies the incoming pixel values by the appropriate DCT matrix coefficients and accumulates the results (Woods *et al.*, 1998). One level of factorisation is used to reduce the number of multiplications from eight to four. Each multiply and accumulate unit is reset every four clock cycles and calculates a separate output frequency. The factorisation means that the even and odd samples are calculated separately.

With block processing, all published systems operate on a block at a time. That is they process each of the rows of a block, saving the results into a transpose buffer, and then use a second processor take the DCT of the columns, as illustrated in Figure 10.17. The transpose buffer can be readily implemented using a dual-port block RAM. One port is used to write the results of the row transform, while the second port is used to read the values in column order. The block RAM in most FPGAs is also sufficiently large to hold two blocks of data, enabling bank switching to be incorporated directly within the single memory simply by controlling the most significant address bit.

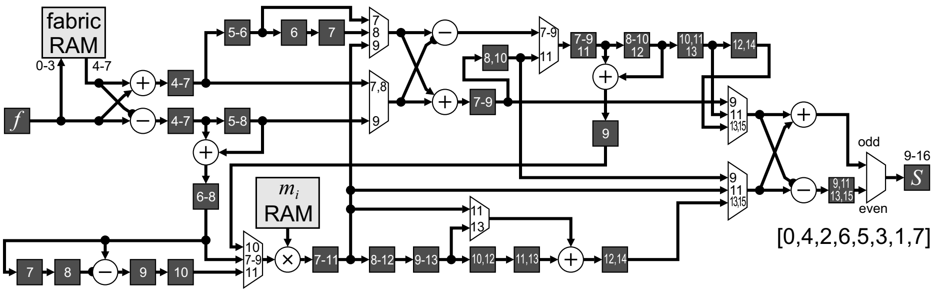


Figure 10.15 A pipelined implementation of Figure 10.14. Numbers on registers and multiplexers refer to the clock cycle they are active relative to the input samples entering on clocks 0 to 7. The registers should be clocked and the corresponding multiplexer inputs selected on these cycles.

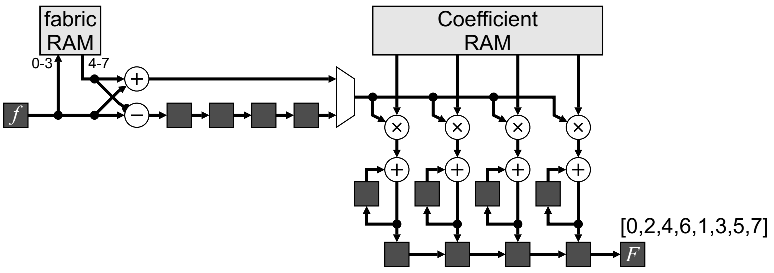


Figure 10.16 Using parallel multipliers to perform an eight-point DCT.

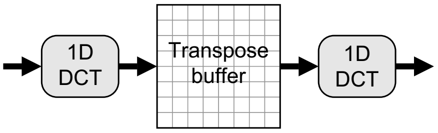


Figure 10.17 Two-dimensional discrete cosine transform of a block.

An alternative approach with data streamed from the camera is to calculate the row DCTs as the data is streamed from the camera, storing the intermediate results in an on-chip cache. Once the eighth row arrives, the data may be read from the cache in column order to perform the column DCTs. With either approach, the cache needs to hold 14 rows of data (effectively seven row buffers for each of the row and column transforms). With the latter approach, a transpose buffer is not required, but the cache may need to be a few bits wider to match the required precision of the intermediate result.

Similar circuits may be used for calculating the inverse DCT. Since the inverse uses the transposed coefficient matrix, similar factorisations to the forward transform may also be used.

The application of the DCT to image and video coding is discussed in more detail in Section 10.4.

10.3 Wavelet Transform

One of the limitations of frequency domain transformations, such as the Fourier transform and cosine transform, is that the resolution in the frequency domain is inversely proportional to that in the image domain. This is illustrated in one dimension in Figure 10.18. Features which are narrow in space have a wide frequency bandwidth, and a narrow frequency (for example, a sine wave) has a wide spatial extent.

One implication of this is that to obtain good frequency resolution, it is necessary to have a large number of samples in the image. Increasing the frequency resolution requires extending the width of the image. However, in taking the Fourier transform, the direct location of features and objects within the image is lost. Good spatial resolution of an object requires looking within a narrow window within the image, which results in poor frequency resolution.

At low frequencies, the spatial resolution is inevitably poor anyway because of the long period. However, the shorter periods of high frequencies makes higher spatial resolution more desirable. This is achieved by dividing the position–frequency space as shown in the right hand panel of Figure 10.18. This is achieved by *wavelet analysis* as follows. Firstly, the image is filtered, separating the low and high frequency components. As a result of the reduction in bandwidth, these components can be down-sampled by a factor of two by discarding every second sample. This gives the position–frequency resolution aspect ratio seen in the figure for the high frequency components. This process is then recursively repeated on the low frequency components giving the desired position–frequency resolution. The schematic of this

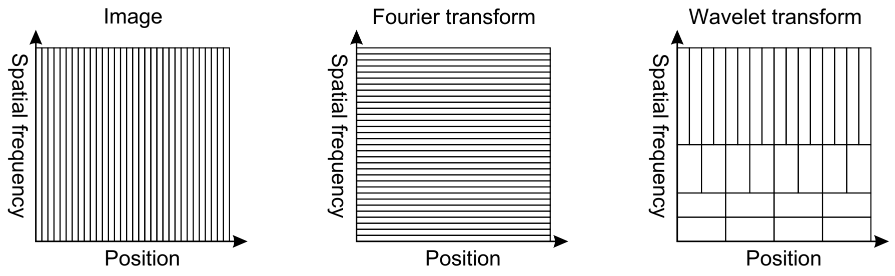


Figure 10.18 Trade-off between spatial and frequency resolution, illustrated in one dimension. Left: image samples; centre: frequency samples from Fourier transform; right: trade-off from the wavelet transform.

process in one dimension is given in Figure 10.19. The wavelet transform effectively looks at the image at a range of scales. For each level of the wavelet transform, the low pass filter outputs are often called the *approximation components*, whereas the high pass filter outputs are the *detail components*.

With the Fourier transform, every frequency component depends on and is contributed to by every pixel in the image. Conversely, the wavelet filters are usually local, finite impulse response filters. The filters are therefore local, enabling spatial location to be maintained, while still separating the frequency components.

The inverse wavelet transform (or *wavelet synthesis*) reconstructs the data by reversing the process (Figure 10.19). Firstly, the coefficients are up-sampled by inserting zeros between the samples. These zeros replace the samples discarded in the down-sampling process. The missing samples are then effectively interpolated by the following low pass or high pass filter, depending on which band the signal came from. The components are then added to give the output signal. This is repeated, going back through the levels until the final output signal is reconstructed.

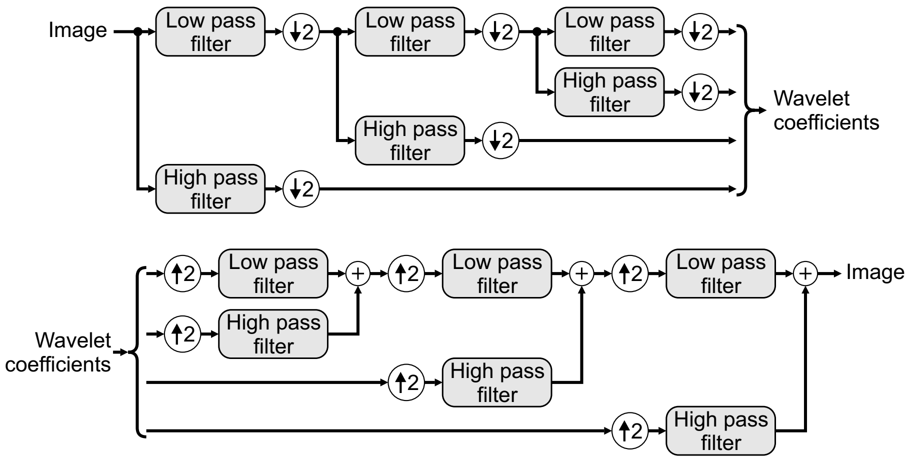


Figure 10.19 Wavelet transform in one dimension, with three decomposition levels shown. Top: wavelet analysis or decomposition; bottom: reconstruction of the original signal, or synthesis.

Let the analysis filters in the z-domain be:

$$\begin{aligned} a_{LP}(z) &= \sum_i a_{LP}[i]z^{-i} \\ a_{HP}(z) &= \sum_i a_{HP}[i]z^{-i} \end{aligned} \tag{10.51}$$

where $a[i]$ are the filter coefficients and z^{-1} corresponds to a one sample delay. Similarly, let the synthesis filters be $s_{LP}(z)$ and $s_{HP}(z)$. It is necessary that the analysis and synthesis filters be pair-wise complementary, so that information is not lost when filtering. Since the filters are not ideal low pass and high pass filters, down-sampling will introduce aliasing within the resultant components. However, given appropriately matched filters, the aliasing introduced by the each filter is exactly cancelled by the reconstruction filters. These two conditions can be expressed mathematically as (Daubechies and Sweldens, 1998):

$$\begin{aligned} a_{LP}(z)s_{LP}(z) + a_{HP}(z)s_{HP}(z) &= 2 \\ a_{LP}(z)s_{LP}(-z) + a_{HP}(z)s_{HP}(-z) &= 0 \end{aligned} \tag{10.52}$$

These imply that the analysis and synthesis filters be related by (Daubechies and Sweldens, 1998):

$$\begin{aligned} a_{LP}(z) &= -z\, s_{HP}(-z) \\ a_{HP}(z) &= z\, s_{LP}(-z) \end{aligned} \tag{10.53}$$

While it is possible for the analysis and synthesis filters to be the same (apart from reversing the order of the filter coefficients), and indeed many wavelet families do use the same filters for analysis and synthesis, such filters cannot be symmetrical. With image processing, use of symmetrical filters is preferred because they prevent movement of features, enabling coefficients to be directly related to the position of features within the image. Therefore, the symmetric biorthogonal family of wavelet filters is often used in image processing applications.

Wavelets, unlike the Fourier transform, do not assume anything beyond the end of the data. They are localised, so any edge effects will be localised to the few pixels on the edge of the image. However, since the finite impulse response filter implementing the wavelet transform is a convolution, the output is longer than the input. Consequently, the data size grows with each iteration. Since the biorthogonal filters are symmetric, a symmetric extension of the input image would result in a symmetric response. Therefore, the data beyond the edge of the image does not need to be recorded. This makes the output sequence the same length as the input. The techniques outlined in Section 8.1 can be used to provide the mirroring.

Wavelets can be readily extended to two dimensions by applying the filter separably to the rows and columns. Each level divides the image into four bands: the LL approximation band, and HL, LH and HH detail bands. The two-dimensional transform is applied recursively to the to the LL band, as illustrated in Figure 10.20 for a three level decomposition. In practise, the high and low pass filters are not separated as shown here. Since the filters are implemented in parallel, the outputs are more often interleaved, with the even samples corresponding to the low pass filter outputs and the odd samples corresponding to the high pass outputs.

10.3.1 Filter Implementations

10.3.1.1 Direct

The obvious direct approach of implementing the wavelet filters is shown in Figure 10.21. Separate filters are used for analysis and synthesis (they are shown together here for illustration). Down-sampling is

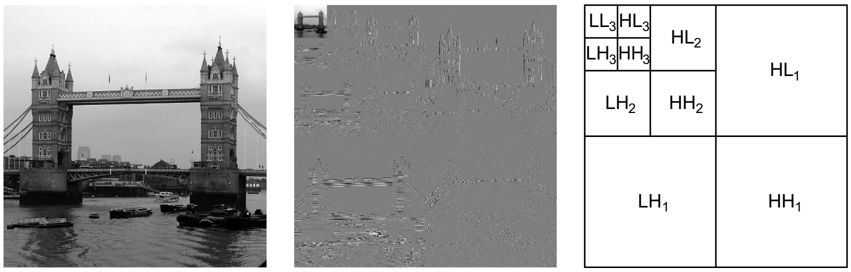


Figure 10.20 Two-dimensional wavelet transform of an image. Left: input image; centre: wavelet transform (using the Haar wavelet); right: position of the decomposition components for each level. (Photo courtesy of Robyn Bailey).

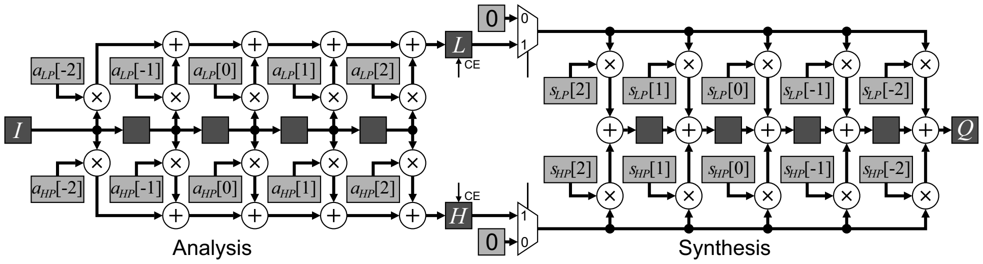


Figure 10.21 Direct implementation of the analysis and synthesis wavelet filters.

achieved by enabling the clock on the filter output registers, L and H, only every second clock cycle. The input is, therefore, one sample per clock cycle, with the output two samples (one from each of the low pass and high pass filters) every two clock cycles. The corresponding synthesis filter up-samples the wavelet signals by inserting zeros between the samples. The synthesis filter shown here uses the transpose structure so that only a single set of registers is required.

With the direct implementation, quantisation of the filter coefficients will mean that the coefficients no longer satisfy Equation 10.52. Consequently, the reconstructed output will not necessarily match the input exactly.

10.3.1.2 Polyphase Filters

The main inefficiency of the direct filters is that every second sample calculated by the analysis filter is discarded, and every second sample into the synthesis filter is a zero, so the corresponding operation is redundant. Consider rearranging the filter equation of Equation 10.51 to separate the odd and even components:

$$\begin{aligned} a(z) &= \sum_i a[i]z^{-i} \\ &= \sum_i a[2i]z^{-2i} + \sum_i a[2i+1]z^{-(2i+1)} \\ &= a_{even}(z^2) + z^{-1}a_{odd}(z^2) \end{aligned} \tag{10.54}$$

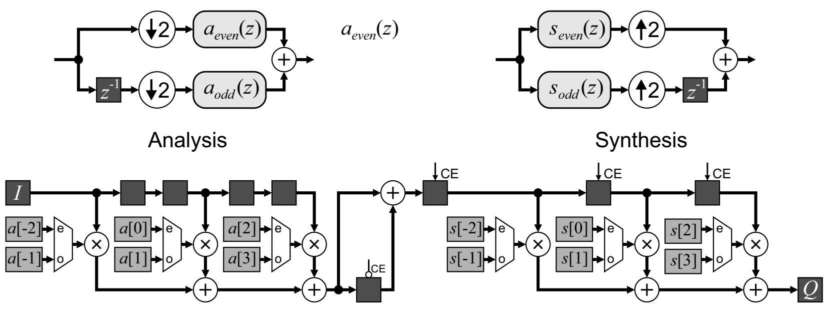


Figure 10.22 Polyphase filter architecture. Top: basic idea of polyphase filters; bottom: hardware implementation.

The two-component filters now operate at the lower sample rate, allowing the number of operations to be halved. The basic idea is shown schematically on the top of Figure 10.22.

For the analysis filter, this still means that all of the multiplications are performed in a single clock cycle, but only need to be evaluated every second cycle. The hardware may be reduced by calculating the even and odd filters in alternate clock cycles. The result of the even filter is held for one cycle and added to that from the odd filter.

For the synthesis filter, the incoming data is only clocked every second cycle. On the output, the results are produced alternately by the even and odd filters. This enables a single filter to be used, by just multiplexing the coefficients.

For a given wavelet, the filter coefficients are constant. Therefore, the multiplexers selecting the filter coefficients can be optimised away, resulting in a very efficient implementation. Only a single side of the wavelet filtering is shown in Figure 10.22; two such circuits are required, one of the low pass filter and one for the high pass filter.

10.3.1.3 Lifting Scheme

The filtering can be optimised further. The designs so far assume that the low and high pass filters are independent. However, these filters are closely related because they are pair-wise complementary. The polyphase filters for both the low and high pass filtering may be written in matrix form:

$$\begin{bmatrix} L \\ H \end{bmatrix} = \begin{bmatrix} a_{LPeven}(z) & a_{LPodd}(z) \\ a_{HPeven}(z) & a_{HPodd}(z) \end{bmatrix} \begin{bmatrix} I_{even} \\ I_{odd} \end{bmatrix} \tag{10.55}$$

The relationship between the filters allows this filter matrix to be factorised (Daubechies and Sweldens, 1998):

$$\begin{bmatrix} a_{LPeven}(z) & a_{LPodd}(z) \\ a_{HPeven}(z) & a_{HPodd}(z) \end{bmatrix} = \begin{bmatrix} k & 0 \\ 0 & k^{-1} \end{bmatrix} \prod_i \begin{bmatrix} a_i(z) & 1 \\ 1 & 0 \end{bmatrix} \tag{10.56}$$

where each of the $a_i(z)$ is usually a constant or first order filter. Each of the terms on the right is a lifting step, resulting in the implementation of Figure 10.23. In general, this will reduce the number of multiplications by a further factor of two.

A second advantage of using lifting steps is that even with quantisation of the filter coefficients and rounding of the outputs of the $a_i(z)$ filters, the output is perfectly recoverable. This is because the synthesis filter just undoes the steps of the analysis filter, subtracting off the terms that were added in.

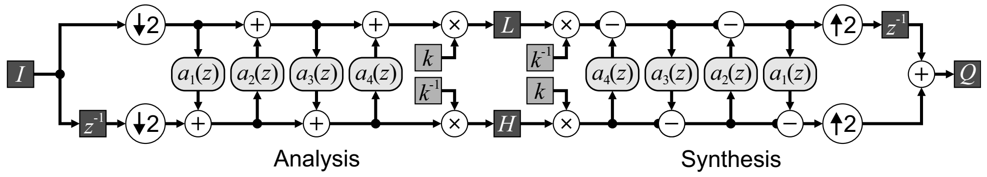


Figure 10.23 Lifting implementation of wavelet filters.

Any quantisation within the synthesis filter will be exactly the same as that in the analysis filter, recreating the data back along the filter chain.

In this context, the scaling terms (by k and k^{-1}) can also be implemented using lifting steps as well, to enable rounding errors from the scaling to also be recovered exactly (Daubechies and Sweldens, 1998):

$$\begin{aligned}\begin{bmatrix} k & 0 \\ 0 & k^{-1}\end{bmatrix} &= \begin{bmatrix} 1 & k-k^2 \\ 0 & 1\end{bmatrix}\begin{bmatrix} 1 & 0 \\ -k^{-1} & 1\end{bmatrix}\begin{bmatrix} 1 & k-1 \\ 0 & 1\end{bmatrix}\begin{bmatrix} 1 & 0 \\ 1 & 1\end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ -1 & 1\end{bmatrix}\begin{bmatrix} 1 & 1-k^{-1} \\ 0 & 1\end{bmatrix}\begin{bmatrix} 1 & 0 \\ k & 1\end{bmatrix}\begin{bmatrix} 1 & k^{-2}-k^{-1} \\ 0 & 1\end{bmatrix}\end{aligned} \tag{10.57}$$

This requires three extra lifting steps, since the first step is in parallel with the last filtering step, so can be combined with it. If wavelet coding is being used for lossy image compression, the L and H outputs will be quantised. In this case, the scaling terms can be incorporated into the quantisation stage without any additional operations.

One limitation of using lifting for implementing the wavelet transform is that all of the operations are now in series, rather than in parallel as with the direct and polyphase implementations. Consequently, the latency is increased relative to the direct or polyphase implementations, although the filters can readily be pipelined to maintain the required throughput.

An implementation of the 9/7 biorthogonal wavelet analysis filter is shown in the centre panel of Figure 10.24, which shows five multiplications and eight additions in series on the critical path. (Note that all of the registers are clocked only every second cycle.) Huang *et al.* (2004) address the problem of increased latency by identifying the critical path, and replace multiplications (which tend to be the most time consuming operation) on the critical path with a multiplication by the inverse on the corresponding parallel path. One implementation which removes all but two of the multiplications and half of the additions from the critical path is shown in the bottom of Figure 10.24. Note that this implementation loses the prefect reconstruction property of the standard lifting scheme.

Rather than rearrange the circuit, the latency can be accepted, with pipelining used to maintain the throughput. All of the filtering is performed after down-sampling, so the filters only process one sample every second clock cycle. Since each stage of the filter has the same architecture, this allows the filter to be half the length, with the second half using the second clock cycle. Alternatively, the full filter could be implemented with two rows (or columns) transformed in parallel if the input bandwidth allows. Another possibility is to feed the low pass output back to the input and use the spare cycles for subsequent levels within the decomposition (Liao *et al.*, 2004).

10.3.1.4 Two-Dimensional Implementations

The descriptions above were for one-dimensional filters. When extending to two dimensions, three approaches are commonly used. The simplest approach is to alternate the row and column processing. This requires processing all of the rows first, saving the intermediate results in off-chip RAM.

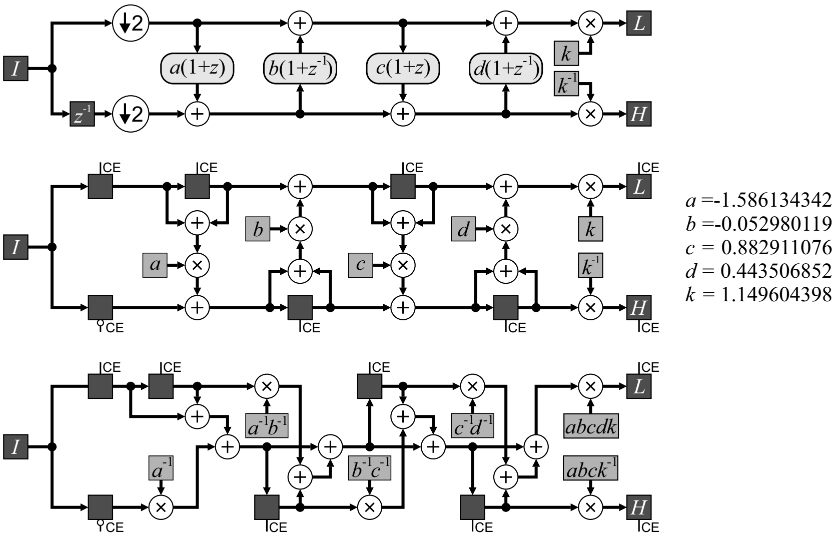


Figure 10.24 Implementation of the 9/7 biorthogonal analysis wavelet filter. Top: basic lifting schematic; centre: hardware implementation; bottom: removing multipliers from the critical path.

The columns are then processed, returning the results to off-chip memory. This procedure is then repeated for the LL band for each level of decomposition. Since each row and column is processed independently, the direct separable approach is easy to accelerate through multiple parallel processors, subject of course to memory bandwidth limitations.

A second approach is to begin the column processing as the processed row data becomes available. This requires replacing delays within the filter with on-chip row buffers, enabling all of the columns to be processed in parallel (Chrysafis and Ortega, 2000). This fits the stream processing model and completely transforms the image as it is being streamed into the FPGA. If the wavelet transform is the end of the processing, the wavelet coefficients can be streamed to off-chip memory as they are calculated. However, if further processing is required, additional caching is required to hold the results from the lower decomposition levels while the transformation is being performed on the higher levels. The amount of synchronisation memory grows rapidly with the number of decomposition levels (Chrysafis and Ortega, 2000). The line processing approach is also a little harder to parallelise. Like the separable approach, multiple rows may be processed in parallel. Processing rows in pairs will also provide the data for column processing two samples at a time. This makes full use of the available clock cycles on the down-sampled data. Alternatively, the spare cycles may be used for processing subsequent decomposition levels (Liao *et al.*, 2004). One aspect to consider is that, rather than multiplexing individual registers, using dual-port fabric RAM builds in the multiplexers for free and may reduce the resources required.

The on-chip memory requirements of the line processing approach may be reduced by dividing the image into blocks and transforming the image a block at a time. The input blocks need to overlap because data from adjacent blocks is required to correctly calculate the pixels around the block boundaries. The block processing approach is also readily parallelisable, although coordination is required when accessing those pixels which are shared between blocks. However, if the resources allow, it is simpler to use line processing rather than process multiple blocks in parallel.

Angelopoulou *et al.* (2006; 2008) compared these three methods for FPGA implementation. The direct separable approach used the fewest resources because it was the simplest architecture, but also took the longest because the data must be accessed multiple times. The best in terms of total time and power was the line buffered approach. Block-based processing has smallest memory requirement, but has the most complex logic because of the need to manage the block boundaries.

10.3.2 Applications of the Wavelet Transform

As with the Fourier transform, wavelets have a wide range of applications within image processing. While a full discussion is beyond the scope of this book, some of the main applications are introduced in this section.

One of the most significant applications of the wavelet transform is in image compression. The steps involved in compressing images are described more fully in Section 10.4. The wavelet transform is used primarily to decorrelate the data. Adjacent pixels in many images are similar and this is exploited by separating the approximation and detail components. As demonstrated in Figure 10.20, the image is sparse within the wavelet domain, with many detail coefficients zero or close to zero. Truncating these coefficients provides an effective means of lossy compression that has a low impact on the perceived image quality.

If an image has noise added, this noise will be most noticeable in the detail coefficients. Noise will affect all of the coefficients, but since the wavelet representation is generally sparse, setting the coefficients below a threshold to zero will remove the noise from those terms. When the image is reconstructed with an inverse wavelet transform, the noise will be reduced. This is called hard thresholding and has the characteristic that it preserves features such as peak heights (Donoho and Johnstone, 1994). An alternative approach is soft thresholding, where all of the wavelet coefficients are shrunk towards zero (the two approaches are compared in Figure 10.25). Soft thresholding tends to give a smoother fit and does not result in the same level of lumpiness that normally results from smoothing noise (Donoho, 1995). The threshold for both methods is proportional to the noise standard deviation in the image. These simple approaches assume that all coefficients are independent. However, there is a strong correlation between detail coefficients at different levels of decomposition. More sophisticated denoising techniques take into consideration these correlations (Sendur and Selesnick, 2002), giving better noise reduction.

Multiscale processing has been mentioned in several contexts throughout this book. Wavelets provide a natural framework for multiscale object detection (Strickland and Hahn, 1997), although relatively little work has been reported on FPGA implementations.

A final application considered here is the fusion of images from different sources. It will be assumed that the images have already been registered; this topic has been covered in some detail in the previous chapter. Image fusion is therefore concerned primarily with combining the data from each of the images in a way that provides a higher quality (in some sense of the word) output image. This can range from simple averaging to reduce noise, through to combining data from disparate imaging modalities (for example positron emission tomography and magnetic resonance images, or multiple spectral bands) and image super-resolution, where the output data has higher resolution than the input. In the context of wavelet processing, each of the registered input images is decomposed through the wavelet transform, the

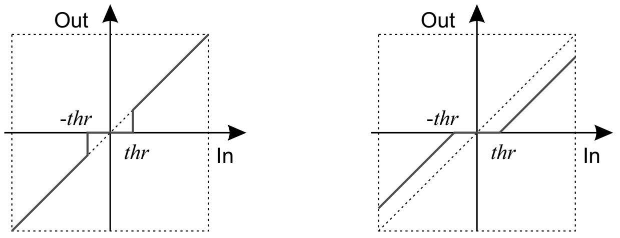


Figure 10.25 Thresholding of wavelet coefficients for denoising. Left: hard thresholding; right: soft thresholding.

wavelet coefficients are then merged in some way and the result inverse wavelet transformed (Pajares and de la Cruz, 2004).

If the coefficients are simply averaged, then the wavelet transform (being linear) is not necessary unless the images are of different resolution. One method of merging the wavelet coefficients that is surprisingly effective is to simply select the coefficient with the maximum absolute value from the input images. This can readily be extended to filtering within the wavelet domain before selecting, or using consistency verification to ensure that groups of adjacent coefficients are consistently selected from the same image (Pajares and de la Cruz, 2004). A classic example of image fusion is focus extension in microscopy of extended objects. With a thick object, the complete object cannot be brought into focus simultaneously – as the focus is adjusted, different parts of the object come into and out of focus. The in-focus components of each image will have higher local contrast, resulting in a larger magnitude within the detail coefficients. Selecting the components with the maximum absolute values will select the parts of the image that are most in focus within the set, with the fused output image having an extended focus range (Forster *et al.*, 2004).

10.4 Image and Video Coding

The information content within images and video may be compressed to reduce the volume of data for storage, or to reduce the bandwidth required to transmit data from one point to another (for example over a network). Images are able to be compressed because they contain significant redundancy.

There are at least four types of redundancy within images. Spatial redundancy results from the high correlation between adjacent pixels. Since adjacent pixels are likely to come from the same object, knowing the value of one pixel can enable the value of adjacent pixels to be predicted. Similarly, there is temporal redundancy between the frames in a video sequence. Successive frames are likely to be very similar, especially if there is limited movement within the sequence. If any movement can be estimated, then the correlation can be increased further through motion compensation. Spectral redundancy reflects the correlation between the colour components of a standard RGB image, especially when looking at natural scenes. Finally, psycho-visual redundancy results from the fact that the human visual system has limited resolution, both spatially and in terms of intensity, and is tolerant of errors or noise within textured regions.

All of these factors enable image data to be compressed by reducing the redundancy. The six basic steps commonly used in compressing an image or video sequence are listed in Figure 10.26. These are outlined briefly in the following paragraphs.

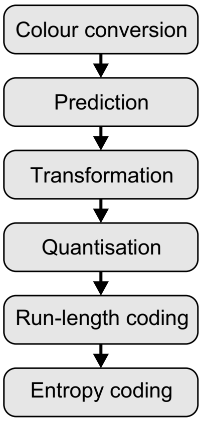


Figure 10.26 Steps within image or video compression.

10.4.1.1 Colour Conversion

Colour images are usually converted from RGB to YCbCr for two reasons. Firstly, most of the variation within images is in intensity. Therefore, the conversion will concentrate most of the signal energy into the luminance component. The weights used for the RGB to YCbCr conversion (Section 6.3) correspond to the relative sensitivities of the human visual system, and this transformation is used with many codecs. Alternatively, the reversible colour transform of Equation 6.57 can also be used by JPEG 2000 (ISO, 2000).

The lower colour spatial resolution of the human visual system enables the chrominance components to down-sampled by a factor of two without any significant loss of subjective quality. This down-sampling is optional and is automatically handled with wavelet-based compression by the first wavelet scale.

10.4.1.2 Prediction

Prediction uses the pixels already available to predict the value of the next pixel (or block of pixels). The residual, or prediction error, has a tighter distribution enabling it to be coded with fewer bits. It is most commonly used with lossless coding methods, where prediction can typically give 2:1 compression. However, it is less often used directly with lossy image coding. Lossy JPEG coding only uses prediction of the DC component between horizontally adjacent blocks. Prediction is used in intra-frame coding within H.264 with a number of different prediction modes.

One of the most common uses of prediction is in inter-frame coding of video. A video sequence is usually divided into groups of pictures (Figure 10.27). The first frame within the group is coded independently of the other frames using intra-frame coding, and is denoted an I-frame. Then, motion compensated forward prediction is used to predict each P-frame from the previous I- or P-frame. Finally, bi-directional (both forward and backward) prediction is then used to predict the intervening frames, denoted B-frames. The B-frames usually require the least data because the backward prediction is effective at predicting the background when objects have moved away. The number of P- and B-frames in each group of pictures depends on the particular codec and application.

Motion compensated prediction divides the image into 8×8 or 16×16 blocks and usually uses the sum of absolute differences to find the offset that minimises the prediction error (as described in Section 9.5.2). This is an effective method of reducing the redundancy when the motion is within the search window. Since the B-frames require the following P-frame (or I-frame), the frames are not transmitted in their natural order, but the order in which they are required to decode the sequence. This means that the encoder must be able to hold several uncompressed B-frames until the following P-frame (or I-frame) is available. The decoder is a little simpler, in that it always only needs to hold the two frames used for providing motion compensated source data.

To be most effective, it is necessary for the encoder to decode the frame before using it to make a prediction so that the data will match that available at the receiver. Otherwise, any quantisation errors in

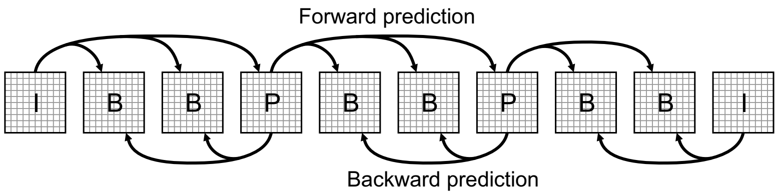


Figure 10.27 Prediction within a group of pictures, showing the relationships between I, P and B frames.

the prediction will not be taken into account and can propagate from one P-frame to the next. This decoding requires an inverse transform to convert the quantised coefficients back to pixel values. Unfortunately, the encoder has little control over the implementation of the inverse transform in the receiver. With a single image codec, this does not matter, but with video the differences in round-off error can accumulate from frame to frame, causing the prediction made at the receiver to drift. This requires care with the design of the inverse transform used at both the encoder and receiver (Reznik *et al.*, 2007).

10.4.1.3 Transformation

The primary purpose of image transformation within an image coding context is to concentrate the energy into as few components as possible. This exploits the spatial correlation within images. Different compression methods use different transformations. Two of the most commonly used transforms are the discrete cosine transform, used with JPEG, MPEG, and the wavelet transform, used with JPEG 2000.

The discrete cosine transform is close to the optimal linear transform for decorrelating the pixel values. This makes it an obvious choice for image compression. However, with global transforms every coefficient depends on every input pixel. Therefore, the image is usually divided into smaller blocks, which are transformed independently. JPEG performs a discrete cosine transform on 8×8 blocks. One limitation of dividing the image into blocks is that, since each block is encoded independently, there is a loss of coherence at block boundaries. After quantisation, this can lead to blocking artefacts – the appearance of false edges at block boundaries, especially at high compression rates. These can be reduced to some extent by filtering the image after decoding (Vinh and Kim, 2010).

The wavelet transform operates locally rather than globally, reducing the need to split the image into blocks. JPEG 2000 allows large images to be split into large blocks or tiles to reduce the memory requirements when coding or decoding. Wavelets compact most of the energy into the low frequency coefficients, giving good energy compaction. The high frequency components are concentrated on edges, with a significant number of small or zero coefficients, enabling effective compression.

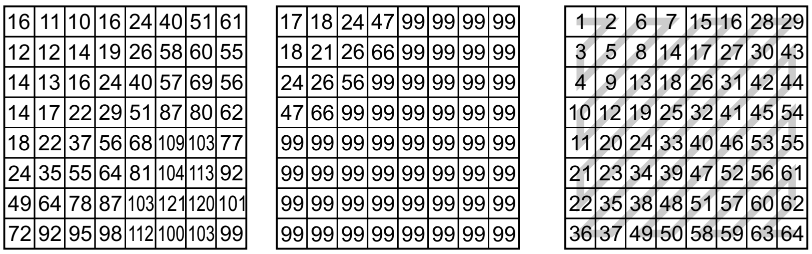
Another advantage of using the wavelet transform is that it is easy to reconstruct an output image at a range of different scales simply by choosing which level of wavelet coefficients are used in the reconstruction.

10.4.1.4 Quantisation

Transformation just rearranges the data, which is completely recoverable via the corresponding inverse transformation (in practise, if the coefficients are not integers, then rounding errors can make the transformation irreversible). The next step in the coding process, quantisation, is lossy in that the reconstructed image will only be an approximation to the original. However, it is the quantisation of the coefficients that can enable the volume of data used to represent the image to be compressed significantly. With coarser quantisation steps, fewer values are used and the greater the compression. However, more information will be lost and this can reduce the quality of the reconstructed image or video.

Quantisation divides the range of input values into a number of discrete bins by specifying a set of increasing threshold levels. It is this many-to-one mapping that makes quantisation irreversible. When reconstructing the value, the centre value for the bin is usually used. The difference between the original value and the reconstructed output is the error or noise introduced by the quantisation process. Since it is usually the transform coefficients that are quantised rather than the input pixel values, the errors do not normally appear as random noise within the image. Typically, quantisation errors show in the image as blurring of fine detail and ringing around sharp edges.

The optimal quantiser adjusts the quantisation thresholds to minimise the mean square error for a given number of quantisation levels. This makes the bins smaller where there are many similar values, reducing the error, at the expense of increasing the error where there are fewer values. This tends to make the bin



16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

1	2	6	7	15	16	28	29
3	5	8	14	17	27	30	43
4	9	13	18	26	31	42	44
10	12	19	25	32	41	45	54
11	20	24	33	40	46	53	55
21	23	34	39	47	52	56	61
22	35	38	48	51	57	60	62
36	37	49	50	58	59	63	64

Figure 10.28 JPEG compression (ISO, 1992). Left: standard luminance and chrominance quantisation tables; right: zigzag coding order.

occupancy more uniform, reducing the gains of entropy coding. The optimal quantiser is data dependent, so requires the levels to be transmitted with the compressed data, reducing compression efficiency. A similar effect may be achieved without the overhead by using a predefined nonlinear mapping followed by uniform quantisation (where the threshold levels are equally spaced). For image compression, however, the gains from using optimal quantisation are not sufficiently significant to warrant the additional complexity; it is easier to simply use uniform quantisation with a finer step size and rely on entropy coding to exploit the fact that some bins are relatively empty.

For image compression, different transformed coefficients have different visual significance, so different quantisation step sizes are used for each coefficient. JPEG has tables specifying the quantisation step sizes for each coefficient; 'standard' tables are given in Figure 10.28 (ISO, 1992; Wallace, 1992). If the DCT coefficients are $C[u, v]$ and the quantisation table is $Q[u, v]$, then the quantised coefficients are:

$$\hat{C}[u, v] = \text{round}\left(\frac{C[u, v]}{Q[u, v]}\right) \tag{10.58}$$

Any scale factors associated with the DCT can be directly included into the quantisation step. This requires two tables to be used: one for actually performing the quantisation and the other which is saved with the data as part of the JPEG output file.

A single quality factor parameter can be used to scale the quantisation step size to control the compression (and consequently the image quality). The implementation within the open source IJG JPEG codec uses a quality factor, QF, that varies from 1 to 100 to scale the standard quantisation tables as follows:

$$\hat{Q}[u, v] = \begin{cases} \text{round}\left(\dfrac{50Q[u, v]}{QF}\right), & QF < 50 \\ \text{round}\left(Q[u, v]\left(2 - \dfrac{QF}{50}\right)\right), & 50 \leq QF \end{cases} \tag{10.59}$$

with any values that go to zero set to 1.

JPEG2000 uses a bit-plane coding approach to control quality. In the output data stream, the data is encoded in bit-plane order with the most significant bit encoded first, through to the least significant bit last. The EBCOT algorithm (Taubman, 2000) divides the tree into blocks and performs coding in multiple passes at each wavelet layer to ensure that the bits are in order of their significance in terms of contributing to the quality of the final image. This enables the coding for each block to be truncated optimally to minimise the distortion while controlling the bit-rate of the final output stream. One FPGA implementation of EBCOT algorithm has been described by Gangadhar and Bhatia (2003).

10.4.1.5 Run-Length Coding

Line drawings, and other related graphics, have a limited number of pixel values, and these often occur in long runs. This makes run-length encoding of such images an effective compression technique.

Many transform techniques have a large proportion of the coefficients close to zero. Having a dead band – a wider bin for zero – gives more zero coefficients and enables run-length coding to be used to compress successive runs of zeros. This is used in two ways: one is true run-length coding, where a symbol and count are output; the other is to have a special zero-to-end symbol, which indicates that all of the remaining symbols to the end of the block are zero. This latter approach is effective both with the block-based coding used by JPEG and also with wavelet coding, eliminating whole trees of zero coefficients, especially for the more significant bits when bit-plane coding is used.

With block-based coding, many of the high frequency components are zero. These can be placed at the end of the block by storing the coefficients in the zigzag order shown in Figure 10.28. This enables many coefficients to be compressed by a single zero-to-end symbol. Zigzag reordering may be implemented efficiently by having a lookup table on the address lines, as shown in Figure 10.29. If the coefficients are not in their natural order, for example if the DCT produces a permuted output, then the zigzag lookup table can also take this into account. A similar lookup table based addressing technique can be used when decoding the images.

10.4.1.6 Entropy Coding

The final stage within the coding process is entropy coding. This assigns a variable length code to each symbol in the output stream based on frequency of occurrence. Symbols which appear frequently are given short codes, while the less common symbols are given longer codes. The entropy of a stream is:

$$E = -\sum_{x} p(x) \log_2 p(x) \tag{10.60}$$

where $p(x)$ is the probability of the xth symbol. The entropy gives a lower bound on the average number of bits used to represent the symbols. For N different symbols, the entropy will be between zero and $\log_2 N$, with the maximum occurring when all symbols are equally probable.

One of the most common forms of entropy coding is *Huffman coding* (Huffman, 1952). This uses the optimum number of bits for each symbol. The Huffman code is built by sorting the symbols in order of probability and building a tree by successively combining the two symbols with the least probability. The length of each branch is the number of bits needed for the corresponding symbol. The main limitation of Huffman coding is that an integer number of bits is required for each symbol. Consequently, the average symbol length will be greater than the entropy. In particular, symbols with probability greater than 50% will always require one bit, even though the entropy is lower. This may be improved by grouping multiple symbols together and encoding symbol groups. A second limitation is that an overhead is incurred in representing the coding table (the mapping between a symbol and its code) since this is data dependent. Although there is no default Huffman table in JPEG, a typical table is provided with the standard based on a large number of images. Using this table saves having to gather the statistics from the image before coding. There are techniques for efficient compression of the coding table (Body and Bailey, 1998), although these are not currently used by any standard.

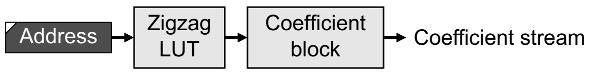


Figure 10.29 Implementation of zigzag reordering.

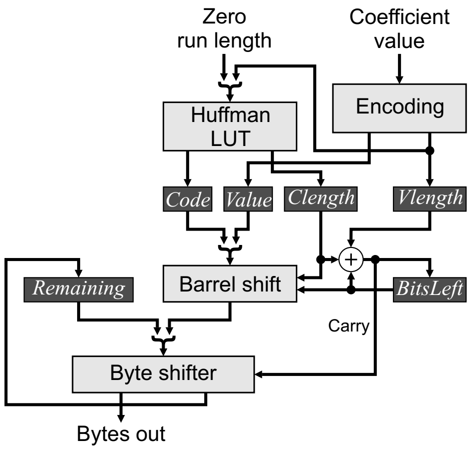


Figure 10.30 Huffman coding for JPEG.

JPEG combines the run length with the size of the coefficient to form an 8-bit symbol which is Huffman coded, followed by the actual coefficient. A block diagram of a possible Huffman coding module for JPEG compression is shown in Figure 10.30. The run length of zeros is provided with each non-zero coefficient as input. The encoding block determines the length of the value, *Vlength*, and encodes it in the available bits, *Value*. The value length is combined with the zero run length, giving an 8-bit symbol for Huffman coding. The coded symbol is restricted to 16 bits length, so a lookup table based approach therefore requires a $256 \times (16 + 4)$ lookup table to provide the *Code* and associated length, *Clength*. The *Code* and *Value* are concatenated and barrel shifted to align with the *Remaining* bits. The lengths are added to the *BitsLeft* with the carry used to determine how many bytes are shifted out.

Using a custom table with JPEG requires performing the initial coding steps without the final Huffman coding. The symbols and quantised coefficients need to be saved in a frame buffer while a histogram is used to gather the statistics on the particular symbols used. The histogram then needs to be sorted and the Huffman tree built. The final step is then to Huffman encode the data from the frame buffer.

A table approach can also be used for Huffman decoding. Brute force decoding would require 2^{16} entries to perform the decoding. However, by segmenting the address space (decoding fewer bits at a time), smaller tables can be used, although at the expense of more clock cycles (Body and Bailey, 1998). This is practical because run-length coding of the zeros means that a value need not be decoded every clock cycle. An alternative approach has been described by Sun and Lee (2003) based on counting the number of leading ones and using this to select the table for decoding. This works because the longer code words tend to begin with a relatively long sequence of ones.

Arithmetic coding removes the restriction of having an integer number of bits per symbol. It encodes the whole stream as an extended binary number, representing a proportion between 0 and 1 (Rissanen, 1976; Langdon, 1984; Witten *et al.*, 1987). Each symbol successively divides the remaining space in proportion to its probability. The key to an efficient implementation is to work with finite precision and output the leading bits once they can no longer be affected. Using finite precision (and rounding) introduces a small overhead over the theoretical minimum length. There is also an overhead associated with specifying the symbol probabilities.

The basic structure of an arithmetic coder is shown in Figure 10.31. The *Start* and *Width* registers hold the start and width of the current range. At the start of coding they are initialised to 0 and 1 respectively. The symbol to be coded is looked up in a table to determine the start and width of the symbol. These are scaled by the current *Width* to give the new *Start* and *Width* remaining. The renormalisation block

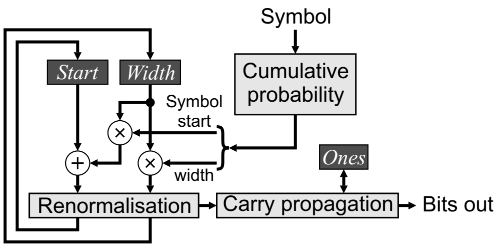


Figure 10.31 Basic structure of arithmetic coding.

renormalises the variables and outputs completed bits. The carry propagation records the number of successive *Ones* output, because a carry can cause these to switch to zeros (Witten *et al.*, 1987).

The overhead associated with initially specifying the initial probabilities may be overcome by adaptive coding. This builds the table of probabilities as the data arrives. While adaptive coding can be used with either Huffman or arithmetic coding, it is less convenient with Huffman coding because changing the symbol probabilities requires completely rebuilding the code tree. Since arithmetic coding works directly with the probabilities, this is less of a problem. There are two approaches to adaptive coding. The first is to assume each symbol is equally likely (with a count of one), with the count incremented and probabilities adjusted as the symbols arrive. There is an overhead resulting from overweighting rare symbols at the start. The second approach is to have an empty table, except for an escape symbol, which is used to introduce each new symbol.

Adaptive coding is good for very short sequences where the overhead of the table can be larger than the actual sequence itself, especially if there is a large number of symbols compared to the length of sequence. It also works well for long sequences because the statistics approach the optimum for the data, although there is an overhead at the start while the table is adapting to the true probabilities. If the statistics are not stationary, then periodic rescaling of the counts (for example dividing them all by a power of two) enables the probabilities to track that of the more recent data.

One final class of compression methods that will be mentioned briefly is that of dynamic dictionary based methods. These allow sequences of symbols to be combined and represented as a single symbol. The original method in this class is LZ77 (Ziv and Lempel, 1977), which uses as its dictionary a window into the past samples. A sequence of symbols that has already been encountered is coded as an offset plus length. LZ77 is combined with Huffman coding to give the DEFLATE algorithm (Deutsch, 1996), used to encode PNG image files. The search through past entries can be accelerated through the use of a hash table to locate potential candidate strings. Alternatively, a systolic array may be used to perform the search (Abd Elghany *et al.*, 2007). A related variation is the LZW algorithm (Welch, 1984), which explicitly builds a dictionary on-the-fly by extending existing sequences each time a new combination is encountered. LZW is used to compress GIF image files.

11

Blob Detection and Labelling

Segmentation divides an image into regions which have a common property. One common approach to segmentation is to enhance the common property through filtering, followed by thresholding to detect the pixels which have the enhanced property. Filtering has been covered in Chapter 8 and various forms of thresholding in Sections 6.1.2, 6.2.3, 7.2.3 and 8.7. However, these operations have processed the pixels individually (using local context in the case of filters). To analyse the image it is necessary to associate groups of related pixels to one another. This enables data on complete objects to be extracted from the groups of pixels.

Image processing operations that transform the image from individual pixels to objects are, therefore, intermediate level operations. Since the input data is still in terms of pixels, it is desirable where possible to use stream-based processing. The output consists of a set of blobs or blob descriptions, and may not necessarily be in the form of pixels.

Of the many approaches for blob detection and labelling, only the bounding box, run-length coding, chain coding and connected components analysis are considered here. FPGA implementation of other region-based analysis techniques, such as the distance transform, watershed transform and Hough transform, are considered at the end of this chapter.

11.1 Bounding Box

The bounding box of a set of pixels is the smallest rectangular box aligned with the pixel axes that contains all of the detected points. It is assumed that the input pixel stream has been thresholded or segmented by preprocessing operations. Let C_i be the set of pixels associated with object i. Then the bounding box of C_i is:

$$\begin{aligned} x_{\min,i} &= \min\{x_p \mid (x_p, y_p) \in C_i\} \\ x_{\max,i} &= \max\{x_p \mid (x_p, y_p) \in C_i\} \\ y_{\min,i} &= \min\{y_p \mid (x_p, y_p) \in C_i\} \\ y_{\max,i} &= \max\{y_p \mid (x_p, y_p) \in C_i\} \end{aligned} \tag{11.1}$$

Initially assume that the input image is binary; the case of multiple labels is considered after presenting the basic circuit. Figure 11.1 shows the basic stream processing implementation. In the frame blanking period

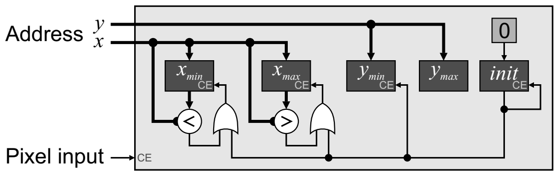


Figure 11.1 Basic bounding box implementation.

before the image is processed, *init* is set to one. The shaded block is only clocked for detected pixels, so that background pixels are effectively ignored. The *init* register provides initialisation when the first pixel is detected, by forcing the address of that pixel to be loaded into the $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$ registers. Since the image is scanned in raster order, the first pixel detected will provide the $y_{\min}$ value. The y value of every detected pixel is clocked into the $y_{\max}$ register because the last pixel detected in the frame will have the largest y. After the first pixel detected, the x value is compared with $x_{\min}$ and $x_{\max}$ with the corresponding value adjusted if extended. At the end of the frame, the four registers indicate the extent of the object pixels within the image.

The basic implementation can be readily extended to process multiple input pixels per clock cycle (Figure 11.2). It is easiest if the number of pixels processed simultaneously is a power of two. The x address then contains only the most significant bits of the address, and two decoders are required to determine the least significant bits of the leftmost and rightmost of the pixels coming in simultaneously. These decoders can easily be built from the lookup tables within the FPGA. A similar approach can be used if multiple rows are processed simultaneously.

If there are multiple input classes, they can all be processed in parallel. Rather than build separate hardware for each class, the fact that each input pixel can only have one label implies that the update logic can be reused. The registers can then be multiplexed by the label of the incoming pixel. Unfortunately, the multiplexers will use more resources than the actual update logic. The alternative is to replace the registers with RAM and use the built-in RAM addressing to perform the multiplexing. The RAMs for $x_{\min}$ and $x_{\max}$ need to be dual-port, because the value needs to be read for the comparison. Since there is typically only a small number of labels, using fabric RAM gives a resource efficient implementation. Figure 11.3 assumes that a late write is not available, so the label and x value need to be buffered for writing in the following clock cycle. Otherwise, the hardware follows much the same form as the basic implementation in Figure 11.1.

Using memory-based processing makes it difficult to process multiple input pixels simultaneously, because only one label can be read or written in each clock cycle. Therefore, to process multiple pixels per clock cycle it is necessary to have multiple copies of the hardware and to combine the results at the end of the frame.

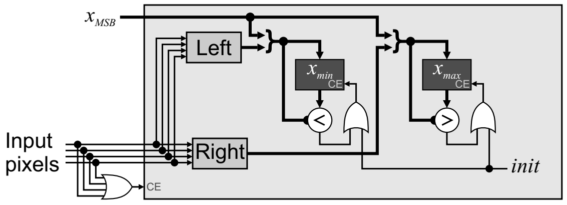


Figure 11.2 Processing multiple horizontal pixels simultaneously.

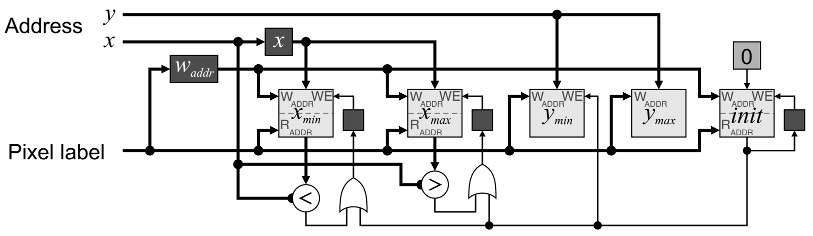


Figure 11.3 Bounding box of multiple labels in parallel.

From the bounding box, it is easy to derive the centre position, size and aspect ratio of the object:

$$centre = \left(\frac{x_{\max} + x_{\min}}{2}, \frac{y_{\max} + y_{\min}}{2}\right) \tag{11.2}$$

$$size = (x_{\max} - x_{\min} + 1, y_{\max} - y_{\min} + 1) \tag{11.3}$$

$$aspect\ ratio = \frac{y_{\max} - y_{\min} + 1}{x_{\max} - x_{\min} + 1} \tag{11.4}$$

However, the basic bounding box is unable to distinguish the orientation of long thin objects, as demonstrated in Figure 11.4. The basic system of Figure 11.1 can be extended to give orientation if required. Consider an object orientated top-right to bottom-left. The register $x_{\min}$ is adjusted on most rows, whereas $x_{\max}$ is only adjusted on the few rows at the top. The converse is true for an object orientated top-left to bottom-right. One method of discriminating between the two is to maintain an accumulator, *acc*, and increment it whenever $x_{\min}$ is adjusted and decrement it whenever $x_{\max}$ is adjusted. At the end of the frame, the sign of the accumulator indicates the orientation of the object. For long thin objects, this enables the angle to be estimated as:

$$angle \approx \text{sign}(acc)*\text{arctan}\left(\frac{y_{\max} - y_{\min}}{x_{\max} - x_{\min}}\right) \tag{11.5}$$

The main limitation of simply using the bounding box to extract the extent of an object is that the measurements are sensitive to noise. Any noise on the boundary at the extreme points will affect the limits and, consequently, the measurements derived in Equations 11.2 to 11.5. An isolated noise point away from the object can result in a spurious bounding box, giving meaningless results. Therefore, it is essential

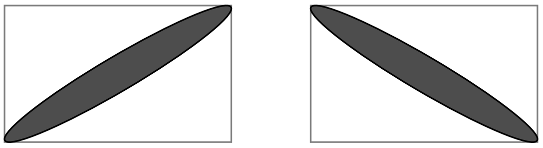


Figure 11.4 The bounding box cannot directly distinguish the orientation of long thin objects.

to apply appropriate filtering to remove noise before using the bounding box. Note that if a morphological opening with a circular or rectangular structuring element is used to remove isolated points, it is sufficient to just perform the erosion, because the effects of the dilation can be taken into account by adjusting Equations 11.2 to 11.5.

The biggest advantage of a simple bounding box is the low processing cost. Therefore, if the noise can be successfully filtered, it provides an efficient method of extracting some basic object parameters.

11.2 Run-Length Coding

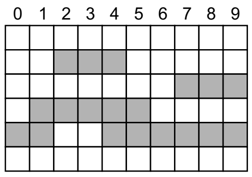
Many intermediate level image processing algorithms require more than one pass through the image. In this case, memory bandwidth is often the bottleneck on the speed of the processing. For some image processing operations, this may be overcome by processing runs of pixels as a unit, rather than processing individual pixels. In all but a few pathological cases, there are significantly fewer runs than there are pixels, allowing a significant acceleration of processing.

There are a number of different ways in which the runs can be encoded, as demonstrated in Figure 11.5. The standard approach used with image compression is to record each run as a (class, length) pair. However, if subsequent processing considers the connectivity between adjacent rows, then the absolute position is only implicit with this representation. An alternative is to record the absolute position of the transitions with a (class, start) pair, with the length given implicitly by the difference between adjacent starts. In the case that the input image is binary, it is only necessary to record the start and end of each run of object pixels and not code the background. In this case, it is also necessary to have an additional symbol to indicate the end of a line when it is a background pixel. This helps to identify blank lines and is necessary to prevent the second and third lines from Figure 11.5 from being considered as part of the same line.

Since each row is coded independently, multiple rows may be processed in parallel by building multiple processors.

The application of run-length coding for blob detection and labelling is considered briefly in the following two sections. Run-length coding can also be used for texture characterisation; in particular the distribution of run-lengths can be used to discriminate different textures. Directly run-length encoding a grey-level image is not as powerful as other texture analysis methods (Conners and Harlow, 1980). However, when applied to a binary image, it can be useful for providing statistical information on object sizes.

Another common application of run-length coding is for image compression (Brown and Shepherd, 1995). When an image has only a few distinct greyscale values or colours, it may be applied directly on the pixels. For general images, run-length coding is not usually effective on its own because adjacent pixels seldom have identical values. However, when the image is transformed, many of the coefficients are truncated to zero. Here run-length coding of the zeros can significantly reduce the number of coefficients that need to be stored.



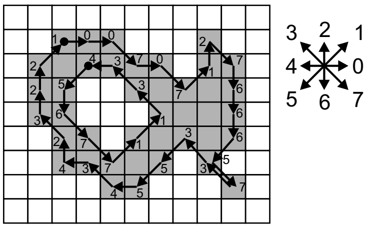
(class,length)	(class,start)	(start,end)
(0,10)	(0,0)	e
(0,2)(1,3)(0,5)	(0,0)(1,2)(0,5)	(2,4)e
(0,7)(1,3)	(0,0)(1,7)	(7,9)
(0,1)(1,5)(0,4)	(0,0)(1,1)(0,6)	(1,5)e
(1,2)(0,2)(1,6)	(1,0)(0,2)(1,4)	(0,1)(4,9)
(0,10)	(0,0)	e

Figure 11.5 Different schemes for run-length encoding.

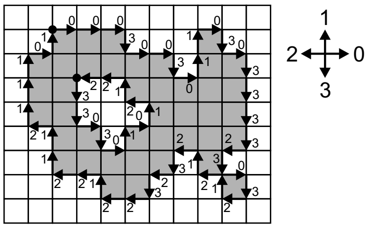
11.3 Chain Coding

One common intermediate level representation of image regions is to encode each region by its boundary. This reduces each two-dimensional shape to a one-dimensional boundary encoded within the two-dimensional space. This has advantages over a conventional image representation. The volume of data is significantly reduced in many applications, especially where the regions are large and have relatively smooth boundaries. This reduction in data can lead to efficient processing algorithms for extracting region features.

There are several different boundary representations (Wilson, 1997), of which the most common is the Freeman chain code (Freeman, 1961). Each pixel is coded with the direction of the next pixel in the sequence, as demonstrated in Figure 11.6. The chain code follows the object pixels on the boundary, keeping the background on the left. Consequently, the outside boundary is traversed clockwise and holes are coded anti-clockwise. A main alternative code is the crack code, which follows the cracks between the object and background pixels, as shown on the right in Figure 11.6.



(2,1) 0070712766657335543423221
(3,2) 567711334



(2,1) 0003003011030333323032121
23232212211211101
(3,3) 3300012122

Figure 11.6 Boundary representations. Left: chain code; right: crack code. The filled circle represents the start point of the boundary.

11.3.1 Sequential Implementation

A standard software algorithm for chain coding is to perform a raster scan of the image until an unprocessed boundary is encountered. Then scanning is suspended while the object boundary is traced and the chain code for the object built. This proceeds from a current boundary pixel looking at the neighbours sequentially in a clockwise order to find the next boundary pixel, and adding the corresponding link to the chain code. The traced pixels are also marked as processed to prevent them from being detected again later. Boundary following continues until the first pixel is reached and the complete boundary has been traced. At this point, the raster scan is resumed looking for another unmarked boundary pixel.

Such an algorithm does not fit within a stream processing implementation because it requires random access to the image. Although part of the algorithm involves a raster scan through the image, this is suspended whenever an object is encountered. The time taken to process the image also depends on image complexity; each pixel is loaded once during the raster scan, while approximately two pixels are read and one pixel is written for each boundary pixel detected during the boundary scan. An approach similar to this was implemented on an FPGA by Hedberg *et al.* (2007). Note that in some applications, the chain code does not actually have to be saved; features such as area, perimeter, centre of gravity and the bounding box can be calculated directly as the boundary is scanned.

The scanning process may be accelerated by using run-length codes as an intermediary (Kim *et al.*, 1988). In the first pass, the image is run-length encoded and the run adjacency determined. The boundary scan can then be deferred to a second pass, where it is now potentially faster because runs can be processed rather than individual pixels.

11.3.2 Single Pass Algorithms

Even better is to perform the whole process in one pass during the raster scan. The basic principle is to maintain a set of chain code fragments and grow both ends of each chain fragment as each row is scanned. In this way, all of the chains are built up in parallel as the image is streamed in.

This has significant implications in terms of memory requirements. When only one chain is encoded at a time and the chain is built in one direction only (by tracking around the edge of the blob), the chain links can simply be stored sequentially in an array. Linking from one chain link to the next is implicit in the sequential addressing. However, when multiple fragments are being built simultaneously, with both the start and the end being extended, it is necessary to use some form of linked list. The storage space required for the linking is often significantly larger than the storage required for the chain code itself.

The basic principle is illustrated in Figure 11.7. As the image is scanned, each edge is detected and the chain code built within the context of the previous row. Consider the scan of the second line in this example. Two separate objects are encountered and chains begun. Even though these belong to the same object, this is not known at this stage. In general, there may be multiple starts for each boundary, occurring at the local tops of the object and background (Cederberg, 1979). As the next line is scanned, the chains are extended on both the left and right ends. On the third line, a new edge is started for the concavity that will eventually become the hole. The two fragments from the original two chains also merge and the corresponding chains linked. If the two fragments belong to two different start points, one of the start points must be deleted.

Several software approaches using this basic technique are described in the literature. Cederberg (1979) scanned a 3 × 3 window through the image to generate the chain links and only coded the left and right fragments, as illustrated by the arrows on the left in Figure 11.7. He did not consider linking the fragments. Chakravarty (1981) also scanned a 3 × 3 window through the image and added segments built to a list. The algorithm involved a search to match corresponding segments as they are constructed. Shih and Wong (1992) used a similar approach to extend mid-crack codes, using a lookup table of the pixels within a 3 × 3 window to derive the segments to be added. Mandler and Oberlander (1990) used a 2 × 2 window to detect the corners in a crack code, which were then linked together in a single scan through the image. Their technique did not require thresholding the image first and derived the boundaries of each connected region of the same pixel value. This meant that each boundary is coded twice: once for the pixel value on each side of the crack. Zingaretti *et al.* (1998) performed a similar encoding of greyscale images. Their technique

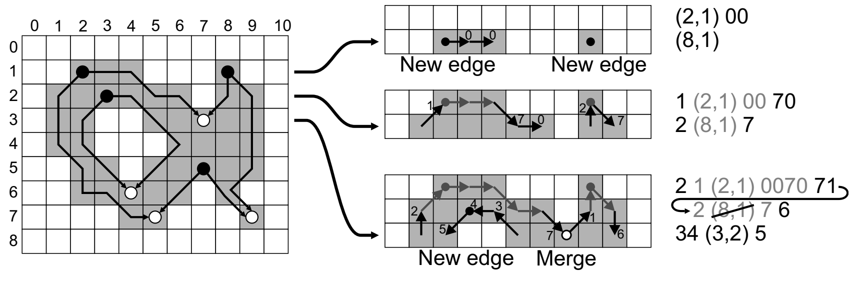


Figure 11.7 Growing a chain code with stream processing. Left: solid circles represent the starting points of new edges, with the lines showing the direction that the fragments are extended from these starting points, until the paths merge at the hollow circles. Right: how the boundary is built for the first three rows of the object.

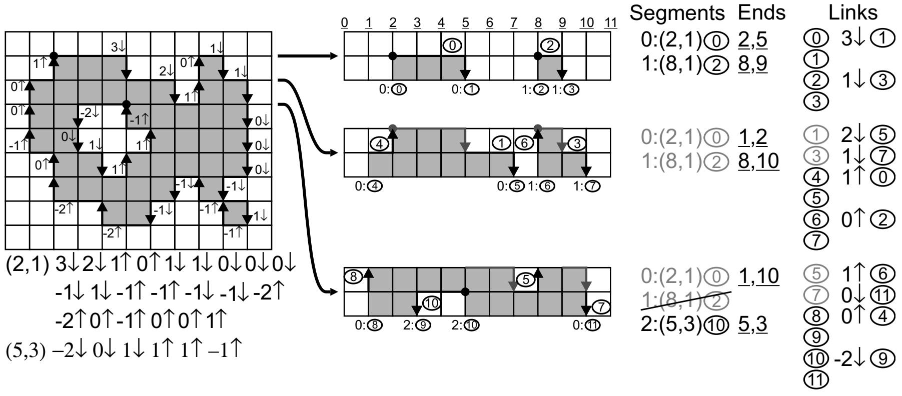


Figure 11.8 Raster-based crack run-length coding. Left: the coding of the regions. Right: building the linked lists representing the boundary, showing the first three lines of cracks.

used a form of run length coding, looking up a 3 × 2 block of run lengths to create temporary chain fragments and linking these together as the image was scanned.

In a stream processing environment, it is unnecessary to actually perform the run-length coding first. This can be achieved by performing run-length coding of the cracks between pixels rather than the pixels themselves, with a simple linking mechanism from one row to the next (Bailey, 2010a). Since the linking primarily takes place at corners, this can be considered as a variation of the method described by Mandler and Oberlander, (1990). This hybrid crack run-length code is demonstrated on the left in Figure 11.8. It consists of the length of a horizontal run, plus one bit indicating whether the crack at the end of the run links up to the previous row, or down to the next row.

The crack run-length chain can be built by scanning a 2 × 2 window through the image, as shown on the left in Figure 11.9. The pattern of pixels within the window provides the control signals for the linking circuitry on the right. With a binary image, the four bits within the window can conveniently be decoded from an arbitrary combination of patterns to produce each control signal with a single LUT.

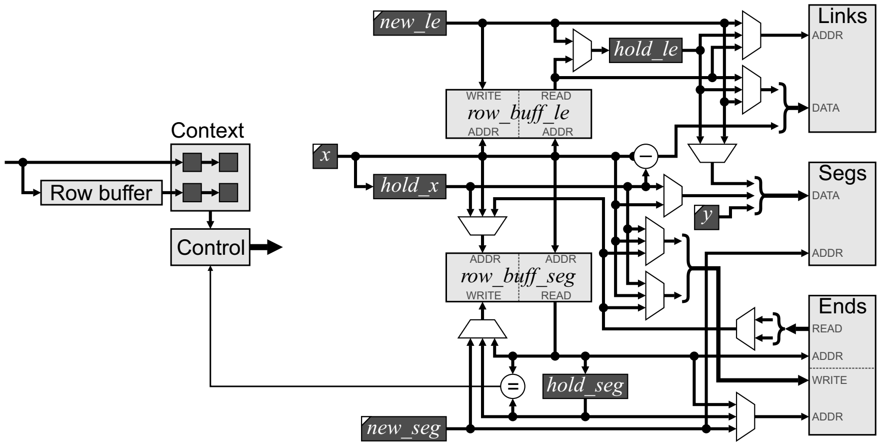


Figure 11.9 Block diagram of raster-based crack code follower.

In addition to the row buffer required to form the input window, two other row buffers are also required to record the segment and link numbers at the ends of each fragment formed, as indicated along the bottom of each scan in Figure 11.8. The link entry row buffer (*row_buff_le*) holds the ends of the links from the previous row and the segment row buffer (*row_buff_seg*) allows the mergers to be detected and processed appropriately.

Looking at the linking first, at the start of a crack run-length, the *x* position is latched into *hold_x*. If the link comes from the previous row, the *le* from the row buffer is held, otherwise a new link entry is allocated and saved in the row buffer and held in *hold_le*. At the end of the run, either the link from the previous line is read from the row buffer or another new link is created, and the run-length data written to the Links table.

If a new segment was created (both ends of the run link down to the next row) it is also necessary to allocate a new start pointer and save the position (either *x* or *hold_x* depending on the direction) and link entry into the Segments table. If a merge occurs (both ends of the run link to the previous row) it is necessary to check if the segments that are linked belong to the same segment or different segments. At the start of the run, the segment number is read from the row buffer and held in *hold_seg*. This is compared with the segment number at the end of the run. If they are the same, the boundary is completed, otherwise the segments are merged and one of the segment pointers is discarded. This requires updating the segment number in the row buffer corresponding to the opposite end of the discarded segment. The Ends table, indexed by the segment number, is used to keep track of the column numbers of the current ends of each segment.

With an early read (the memory is read in the middle of the clock cycle) and late write, the whole process can operate at one pixel per clock cycle (Bailey, 2010a). With a little extra logic, the segment entries that are discarded through merging can be recycled reducing the size of the Segments and Ends tables. The Links table requires one entry for each vertical crack within each object (the horizontal cracks are represented by the run lengths). In the worst case, this is the same as the number of pixels in the image (for alternating black and white lines), although for most realistic images of interest, the size of the table can be considerably smaller.

If necessary, the crack run-length code can be converted to a conventional crack code or Freeman chain code for subsequent processing. However, much of the processing may be able to be performed directly on the crack run-length code.

An image of the region may be reconstructed from the boundary code by drawing the boundary within an empty image and filling between the boundary pixels using a raster scan, producing a streamed output image.

11.3.3 Feature Extraction

Since the boundary contains information about the shape of a region, many features or parameters may be derived directly from the chain code (or other boundary code). The most obvious measurement from the chain code is the perimeter of the object. Unfortunately, simply counting the number of steps within the chain or crack code will overestimate the perimeter length because of quantisation effects. The error in the estimate also depends on the angle of the boundary within the image. To overcome this problem, it is necessary to smooth the boundary by measuring the distance between points *k* steps on either side, and assigning the average as the step size at each link (Proffitt and Rosen, 1979):

$$Perimeter = \frac{1}{k}\sum_i \sqrt{(y_{i+k}-y_i)^2 + (x_{i+k}-x_i)^2} \tag{11.6}$$

For long straight boundaries, increasing *k* can reduce the error in the estimate of the perimeter to an arbitrarily low level. However, this will result in cutting corners. Consequently, making *k* too large will underestimate the perimeter. For objects with rough boundaries, the perimeter will depend on the scale of the measurement, enabling a fractal dimension to be calculated as a feature.

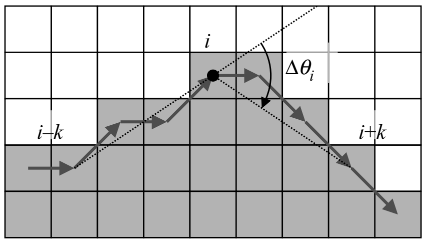


Figure 11.10 Measuring curvature for detecting corners.

Corners can be defined as regions of extreme curvature. Again, the quantisation effects on the boundary mean that the local curvature is quite noisy and can be smoothed by considering points k links apart. A simple corner measure at a point is the difference in angle between the points k links before the current point and k links after the current point (Figure 11.10).

$$\Delta\theta_i = \tan^{-1}\frac{y_{i+k}-y_i}{x_{i+k}-x_i} - \tan^{-1}\frac{y_i-y_{i-k}}{x_i-x_{i-k}} \tag{11.7}$$

Since k is a small integer and the steps are integers, the angles may be determined by table lookup. (For example, with $k = 3$, $-3 \le y_{i+3}-y_i, x_{i+3}-x_i \le 3$, requiring only four bits from each of Δx and Δy, or eight input bits total for the lookup table to implement $\tan^{-1}$.) The peaks in $\Delta\theta_i$ that are over a predefined angle are defined as corners. A more sophisticated corner measure that takes into account the length over which the angle changes is given by Freeman and Davis (1977).

Measuring the area of a region consists of counting the number of pixels within the boundary. This may be obtained directly from the boundary by using contour integration (Freeman, 1961). It is important to note that the chain code is on one side of the boundary, so it will underestimate the area of regions and overestimate the area of holes unless the part pixels outside the chain are taken into account. The crack code does not have the same problem. Expressing it in terms of the crack codes:

$$Area = \oint_{chain} 1dxdy = \sum_j \left(\sum_i^j \Delta x_i\right)\Delta y_j \tag{11.8}$$

This is amenable to direct calculation from the crack run-length codes. The area is generally much less sensitive than the perimeter to digitisation errors and noise.

Note that blobs will have a positive area, while holes will have a negative area. This enables external and internal boundaries to be distinguished, enabling the number of holes, and the Euler number to be calculated.

The centre of gravity may be determined from the contour integral for the first moment:

$$\begin{aligned} x_{COG} &= \frac{S_x}{Area}, \text{ where } S_x = \oint_{chain} xdxdy = \frac{1}{2}\sum_i x_i^2 \Delta y_i \\ y_{COG} &= \frac{S_y}{Area}, \text{ where } S_y = \oint_{chain} ydxdy = \frac{1}{2}\sum_i x_i(y_{i-1} + y_i)\Delta y_i \end{aligned} \tag{11.9}$$

Higher order moments may be calculated in a similar way.

The convex hull may also be determined from the boundary code. Consider each link consisting of an arbitrary offset, $(\Delta x_i, \Delta y_i)$. Point i on the boundary is concave if:

$$\Delta x_i \Delta y_{i-1} \leq \Delta x_{i-1} \Delta y_i \tag{11.10}$$

The point may then be eliminated by combining it with the previous point:

$$(\Delta x_{i-1}, \Delta y_{i-1}) \Leftarrow (\Delta x_{i-1}, \Delta y_{i-1}) + (\Delta x_i, \Delta y_i) \tag{11.11}$$

If this is performed for every pair of adjacent points until no further points can be eliminated, then the remaining points are the vertices of the minimum convex polygon containing the region. (Note that holes will reduce to nothing since they are concave. To obtain the convex hull of a hole, the left and right hand sides of Equation 11.10 need to be swapped.)

From the convex hull, the depth of the bays may be determined, enabling a concavity analysis approach to separating touching convex objects (Bailey, 1992). The principle is that a convex object should have no significant concavities, so a concavity indicates multiple touching objects. From the deepest concavity, an opposite concavity is found and the object separated along the line connecting them. This is repeated until no significant concavities remain.

From the convex hull, the minimum area enclosing rectangle may be determined (Freeman and Shapira, 1975). The minimum area enclosing rectangle has one side coincident with the minimum convex polygon, so each edge can be tested in turn to find the extent of the region both parallel to and perpendicular to that edge. The extents that give the smallest area correspond to the minimum area enclosing rectangle.

The bounding box (aligned with the pixel axes) can be found from a simple scan around the boundary, finding the extreme points in each direction.

Chain codes may also be used for boundary or shape matching (Freeman, 1974). Although a full description is beyond the scope of this work, the basic principles are briefly outlined. If the complete boundary is being compared, a series of shape features may be extracted from the boundary and also the boundaries of candidate matches. Those with wildly different feature vectors can be eliminated. Those with similar feature vectors may be compared in more detail using correlation to determine the quality of fit. If the match is on partial boundaries, a chain code can be broken into multiple segments based on prior knowledge of the problem. For example, in a jigsaw-type problem, appropriate segmentation points would be sharp corners (Freeman, 1974). These fragments are then matched as before, using simple features to eliminate obvious mismatches and to find approximate matches that can be investigated further.

One set of shape features that may be extracted from boundary codes are Fourier descriptors. For a closed boundary, the shape is periodic, enabling analysis by Fourier series (Zhang and Lu, 2002). While any function of the boundary points could be used, the simplest is the radius from the centre of gravity as a function of angle. The main limitation of this is that the shape must be fairly simple for the function to be single-valued. A arbitrary shape can be represented directly by its x and y coordinates, as a complex number: $x(t) + iy(t)$, as a function of length along the perimeter, t. Alternatively, the angle of the tangent as a function of length can be used, normalised to make it periodic (Zahn and Roskies, 1972). Whichever features are used, the shape may be represented by the low order Fourier series coefficients. The method is also relatively insensitive to quantisation noise and with normalisation can be used for rotation and scale invariant matching. The simple and complex number-based Fourier descriptors also allow the smoothed boundary to be reconstructed from a few coefficients.

11.4 Connected Component Labelling

While chain coding concentrates on the boundaries, an alternative approach labels the pixels within the connected components. Each set of connected pixels is assigned a unique label, enabling the pixels associated with a region to be isolated and features of that region to be extracted.

11.4.1 Random Access Algorithms

There are several distinct approaches to connected components labelling. One is based on boundary scanning (Chang *et al.*, 2004), effectively determining the boundary and then filling in the object pixels inside the boundary with a unique label. In this approach, like chain coding, the image is scanned in a raster fashion until an unlabelled object pixel is encountered. A new label is assigned to this object and the boundary is scanned. Rather than extract chain codes of the boundary, the boundary pixels are assigned the new label. When the raster scan is resumed, these boundary labels are propagated internally to label the pixels associated with the object. Hedberg *et al.* (2007) implemented this algorithm on an FPGA. The main limitation of this approach is that the boundary scan requires random access to the image, requiring the complete image to be available in a frame buffer. The advantage of such an implementation is that the memory required by the labelling process is smaller than conventional connected components labelling because fewer bits are required to store the labels. In a typical case, the processing is also faster than standard two-pass algorithms (described shortly) because all of the processing is performed in a single pass plus the time associated with the boundary scanning. Some features can also be extracted directly during the boundary scan phase (for example any features that may be extracted from the boundary codes can be extracted at this stage).

An alternative to scanning around the boundary of an object is to perform a flood–fill operation when an unlabelled pixel is encountered during the raster scan (AbuBaker *et al.*, 2007). A flood fill suspends the raster scan and defines the unlabelled pixel as a seed pixel for the new region. It then fills all pixels connected to the seed pixel with the allocated label. A row can be filled easily, with connected pixels in the rows immediately above and below added to a stack. Then, when no more unlabelled pixels are found adjacent to the current row, the seed is replaced by the top of stack and flooding continues. By carefully choosing which pixels are pushed onto the stack, an efficient algorithm can be obtained (AbuBaker *et al.*, 2007). Again, the limitation is that the flood–fill phase requires random access to the image, requiring the complete image to be available in a frame buffer.

11.4.2 Multiple-Pass Algorithms

To avoid the need for random access processing, the alternative is to propagate labels from previously labelled connected pixels. The basic idea is illustrated in Figure 11.11. When an object pixel is encountered during the initial raster scan, its neighbours to the left and above are examined. If all are background pixels, a new label is assigned. Otherwise, the minimum non-zero (non-background) label is propagated to the current pixel. This is accomplished by having the input multiplexer select the *Label* during the first pass. If *Label* is the minimum non-zero value, then the new label is used and *Label* is incremented. For **U** shaped objects, multiple labels are assigned to a single connected component. At the bottom of the **U**, the minimum of the two labels is selected and propagated. However, the pixels that have already been labelled with the non-minimum label must be relabelled. The simplest method of accomplishing this is to perform a second, reverse raster scan through the image, this time from the bottom-right corner back up through the image. In the second pass (and subsequent passes) the input

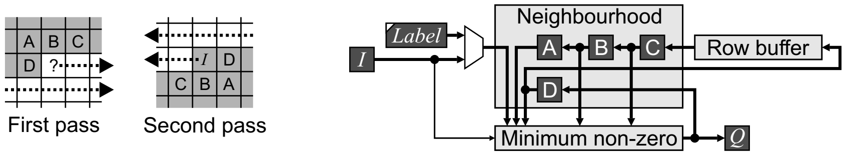


Figure 11.11 Multipass connected components labelling. Left: neighbourhood context, with the dotted lines showing the scan direction; right: label propagation.

multiplexer is switched to the input pixel, effectively including it within the neighbourhood. The second pass propagates the minimum non-zero label back up the branch, replacing the larger initial label. For **W** shaped objects and spirals, additional forward and reverse passes through the image may be required to propagate the minimum label to all pixels within the connected component. Therefore, the process is iterated until there are no further changes within the image.

This iterative, multipass approach to connected component labelling has been implemented on an FPGA (Crookes and Benkrid, 1999; Benkrid *et al.*, 2003b). The main advantage of this approach is that the circuitry is very simple and quite small. It also requires no intermediate storage (apart from the frame buffer) – all necessary information is stored in the intermediate images. However, the main problem is that the number of passes through the image depends on the complexity of the connected components and is indeterminate at the start of processing. Therefore, there is potentially a large latency as a result of processing. The image must also be stored in a frame buffer, although the second and subsequent passes may be performed in place.

11.4.3 Two-Pass Algorithms

This label propagation may be reduced to two passes by explicitly keeping track of the pairs of equivalent labels whenever two parts of a component with a single label merge. The classic software algorithm (Rosenfeld and Pfaltz, 1966) does just this and is illustrated with a simple example in Figure 11.12. The first pass is similar to the first pass of the multipass algorithm as described above: as the image is scanned, if none of the previously processed neighbours has a label, a new label is assigned as before. If the neighbourhood contains only one label, that label is propagated to the current pixel. At the bottom of **U** shaped regions, however, there will be two different labels within the neighbourhood. These labels are equivalent in that they are associated with the same connected component. One of the labels (usually the smaller) will be retained and all instances of the redundant label must be converted to the current label. Rather than use multiple additional passes to propagate the change through the image, the fact that the two labels are equivalent is recorded in an equivalence table. At the end of the first pass, all of the equivalences are resolved and a lookup table created which maps the initial temporary labels to the final label. The issue of equivalence resolution has received much attention in the literature and is the main area of difference between many related algorithms (Wu *et al.*, 2005; He *et al.*, 2007). A wide range of techniques has been used to reduce the memory requirements of the data structures used to represent the equivalence table and to speed the search for equivalent labels. A second raster scan through the image then uses the lookup table to assign the final label to each object within the image.

An early implementation of the standard two-pass algorithm (Rachakonda *et al.*, 1995) used a bank-switched algorithm structure to enable every frame to be processed. As the initial labelling pass was performed on one image, the relabelling pass was performed on the previous image. An implementation of the standard two-pass algorithm using Handel-C is described by Jablonski and Gorgon (2004), where they show how the software algorithm is ported and refined to exploit parallelism.

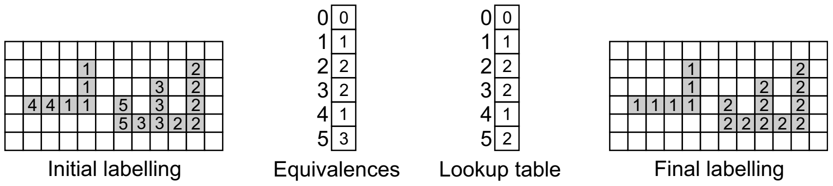


Figure 11.12 Connected component labelling process. From the left: the initial labelling after the first pass; the equivalence or merger table; the lookup table after resolving equivalences; the final labelling after the second pass.

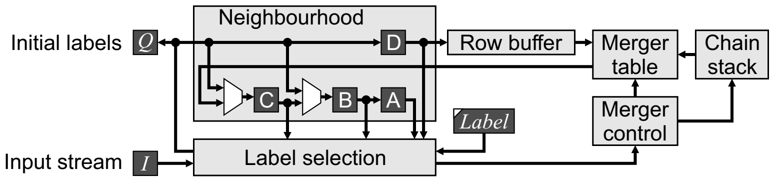


Figure 11.13 First-pass logic required to efficiently implement merging.

The key to an efficient implementation is to have an efficient approach to resolving equivalences. When a new label is created, its entry in the equivalence table is set to point to itself. Then whenever a merger occurs, the larger label is set to point to the smaller label. While this is adequate in the simple example shown in Figure 11.12, the one label may merge with several others, making it necessary to store the labels as sets. One approach is to scan through the table every time two regions merge to update the labels to the new equivalent label. However, this cannot be accomplished in a single clock cycle, preventing its use within a streamed implementation.

One approach is to partially resolve the mergers with each row (Bailey and Johnston, 2007). This is outlined in Figure 11.13. The basic idea is to prevent the redundant label from propagating further and resulting in additional mergers. To accomplish this, it is necessary to use the merger table to replace any instances of the redundant label that have been saved into the row buffer with the current label being used for that region. The merger control block initialises the entry in the merger table when a new label is allocated. A merger can only occur when pixel B is background and C has a different label to A or D. In this case, the merger table is updated so that the larger label is translated to the smaller label which is propagated. (The merger table must be dual-port to allow this update while the row buffer output is being translated.) The multiplexers on the input of B and C ensure that the redundant label within the neighbourhood is replaced by the minimum label.

One minor problem remains – if there is a sequence of mergers with the larger label on the left, as shown in the second region in Figure 11.12, then the merger table contains a chain of links from one label to the next. In this case, a single lookup in the merger table is insufficient to return the current label. (It can be shown that if the smaller label is on the left, this problem does not occur; Bailey and Johnston, 2007). The links may be unchained by a second backwards scan along each row. However, the data stored in the row buffer does not need to be changed as long as the entries in the merger table are unchained to all point to the smallest label for each region. This may also be accomplished by scanning through the merger table to unchain the links. The affected links may be accessed quickly by pushing the pair of merged labels onto a stack whenever the larger label is on the left (Bailey and Johnston, 2007). Then, at the end of the row, during the horizontal blanking period, the pairs are popped off the stack (effectively jumping to the key locations back along the row) and the merger table updated. With pipelining, unchaining requires one clock cycle per stack entry (Bailey and Johnston, 2007).

At the end of the first pass, it is necessary to reconcile chains of mergers that have taken place on different rows. A single pass through the table from the smallest label through to the largest is sufficient. If the entry in the table is not pointing to itself, only a single additional read is required to obtain the final label. If desired, the labels may also be renumbered consecutively during this pass. Each initial label will therefore require one or two reads and one write to the merger table.

The second pass through the image simply uses the merger table as a lookup table to translate the initial labels to the final label used for each connected component.

In software, the second pass can be accelerated by using run-length coding (He *et al.*, 2008), because in general there will be fewer runs than pixels. In hardware though, little is gained unless the image is already run-length encoded by a previous operation. Otherwise the pixels still have to be read one by one, and all of the processing for each pixel takes place in a single clock cycle (although several rows can be run-length encoded in parallel; Trein *et al.*, 2007). If a streamed output image is required, the only advantage of run-length encoding is to reduce the size of the intermediate frame buffer. However, if the output is an image in a frame buffer rather than streamed, then run-length coding can provide additional acceleration (Appiah and Hunter, 2005). During the first pass, in addition to storing the temporary labels in a frame buffer, the image is run-length coded. Then, during the second relabelling pass, only the runs with labels that need to be changed need processing. All of the background pixels and runs with the correct label can be skipped.

The algorithms considered so far are largely sequential. This is because labelling is not a local operation – distant pixels could potentially be connected, requiring the same label. This has led to a range of parallel architectures to speed up the label propagation (Alnuweiri and Prasanna, 1992). The major limitation of many of these approaches is that they require a large number of processors, making their implementation on an FPGA very resource intensive. It is also usually assumed that the image data has been preloaded onto the processors. When processing streamed images, much of the system must remain idle while the image is loaded in. The bandwidth bottleneck destroys the gains obtained by exploiting parallelism. One technique that may be exploited by an FPGA implementation is to divide the image into non-overlapping blocks. A separate processor is then used to label each of the blocks in parallel. In the merger resolution step, it is also necessary to consider links between adjacent blocks. This requires local communication between the processors for adjacent blocks. The second pass, relabelling each block with the final label can also be performed on each block in parallel.

11.4.4 Single-Pass Algorithms

The global nature of connected components labelling requires a minimum of two passes through the image to produce a labelled image. Frequently, connected components labelling is followed by feature extraction from each region, with the labelling operation used to distinguish between separate regions. In such cases, it is not actually necessary to produce the labelled image as long as the correct feature data for each region can be determined.

The basic principle behind single-pass approaches is to extract the data required to calculate the features during the first pass (Bailey, 1991). This requires a set of data tables to accumulate the data. When two regions merge, the data associated with each of the regions is also merged. Therefore, at the end of the image scan feature data has been accumulated for each of the connected component.

The data that needs to be maintained depends on the features that are being extracted from each connected component. The restriction on which features can be accumulated on-the-fly is governed by the fact that there must be a simple way to combine the raw data for each region when a merger occurs.

- If only the number of regions is required, this can be determined with a single global counter. Each time a new label is assigned, the counter is incremented, and when two regions merge the counter is decremented.
- For measuring the area of each region, a separate counter is maintained for each label. When two regions are merged, the corresponding counts are combined.
- For determining the centre of gravity of the region, or any other moment-based feature, the sums of the x and y coordinates are maintained. (For higher order moments, sums of powers of x and y coordinates are kept.) The corresponding sums are simply added on merging to give the new sum for the combined region. At the end of the image, centre of gravity may be obtained by normalising the sums by the blob area:

$$(x_{cog}, y_{cog}) = \left(\frac{\sum x}{\sum 1}, \frac{\sum y}{\sum 1}\right) \tag{11.12}$$

- Similarly, higher order moments may be obtained by appropriately combining the accumulated sums. This includes features such as the orientation and size of the best fit ellipse.
- To obtain the bounding box of each region, the extreme coordinates of the pixels added are recorded. On merging, the combined bounds may be obtained by using the minimum or maximum as necessary of the subcomponents.
- The average colour or pixel value may be calculated by adding the pixel values associated with each region into an accumulator and normalising by the area at the end of the scan. Similarly higher order statistics, such as variance (or covariance for colour images), skew and kurtosis, may be accumulated by summing powers of each pixel value and deriving the feature value from the accumulated data at the end of the image.
- Other, more complex, features such as perimeter may require additional operations (such as edge detection) prior to accumulating data.

Obviously, some features are not able to be calculated incrementally. In this case, it is necessary derive the labelled image prior to feature extraction.

When single-pass processing is possible, the architecture of Figure 11.13 can be extended to give Figure 11.14 (Bailey and Johnston, 2007; Bailey *et al.*, 2008). The data table can be implemented with single-port memory by reading the data associated with a label into a register and performing the updates in the register. On a merger, the merged entry can be read from the data table and combined with the data in the register. Then, on the first background pixel after the blob, the data may be written back to the data table (Bailey and Johnston, 2007). The object data is then available from the data table at the end of the frame.

The main limitation with this approach is that the size of the data and merger tables depends on the number of initial labels within the image. In the worst case, this is proportional to the area of the image, where the maximum number of labels is one quarter of the number of pixels.

Since the initial labels are not actually being saved (apart from the previous row in the row buffer), the number of labels in active use at any one time is proportional to the width of the image not the area. Therefore, the memory used by the data table and merger tables is being used ineffectively. The memory requirements of the system can be optimised by recycling labels that are no longer used (Lumia *et al.*, 1983; Khanna *et al.*, 2002).

The approach taken in Bailey *et al.* (2008) and Ma *et al.* (2008) is to completely relabel each row. Several changes are required to the architecture of Figure 11.14. The equivalence table is split into two tables (Figure 11.15), with the merger table reflecting equivalences between labels on the same row and a translation table to convert the labels used in previous row to those used in the current row. Two merger and data tables are also required, for reasons that will be outlined shortly. There are also some significant changes to the logic and operation of the system.

Firstly, the label selection code is a little more complex. A new label is assigned each time a region is first encountered on a row. Pixels from the previous row (in registers A, B and C) all need to be updated with the current label and the translation is recorded in the translation table. Secondly, the merger table is

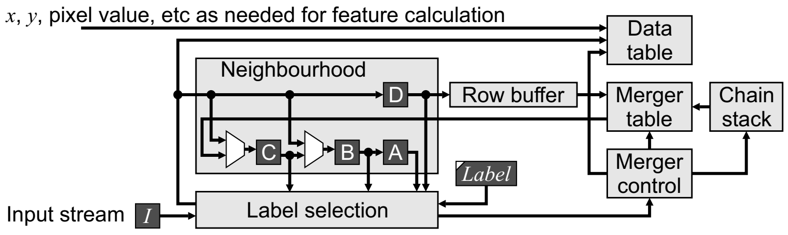


Figure 11.14 Single-pass connected components analysis.

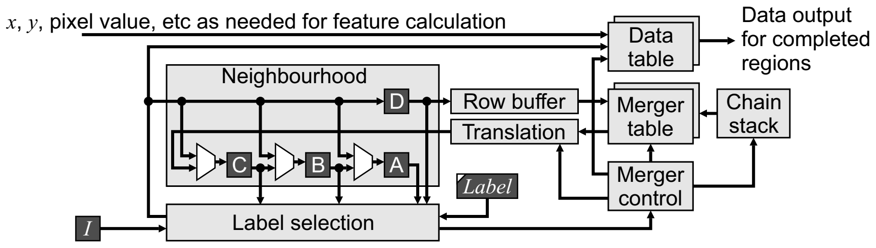


Figure 11.15 Single-pass connected components analysis with label reuse.

only updated when two regions which have different current row labels merge; many region mergers result in translations rather than requiring the merger table to be updated. It can be shown that the larger label will always be on the left for those mergers that require updating the merger table (Ma *et al.*, 2008). Therefore, all mergers must be pushed onto the chain stack for resolution at the end of the row. Thirdly, the merger table is only applicable for a row, rather than the whole image. Two tables are therefore required: one for mapping mergers from the previous row (out of the row buffer) and one for recording mergers on the current row. At the end of the row (after unchaining), these tables are then swapped. The translation table, since it is specific to the row being translated, is discarded and rebuilt each row. Fourthly, two sets of data tables are required because the indexes change with every row. Whenever an entry is made within the translation table, the corresponding data is taken from the previous row's data table and combined with the data for the current row. Any data not transferred has no pixels on the current row and therefore represents a completed objects. Such data is available for subsequent feature processing at the end of each row.

Trein *et al.* (2007; 2008) took a slightly different approach. On the input, they run-length code up to 32 pixels in parallel, enabling a slower clock speed to be used to sequentially process the run-length encoded data. When regions merge, a pointer from the redundant label is linked to the label that is kept; however, chaining is not discussed. To keep the data table small, a garbage collector detects when a region has been completely scanned, outputs the resulting region data and adds the released label to a queue for reuse.

11.4.5 Multiple Input Labels

In all of the discussion above, it has been assumed that the input image is binary. If, instead, the input was a labelled image, for example from multilevel thresholding or colour segmentation, then the processing outlined above needs to be extended. A naïve approach would be to build a separate processor for each label. However, since each incoming pixel can only have one label, the hardware of a single processor may be augmented to handle multiple input labels.

The main change is that the labelling process should only propagate labels within regions with the same input label. Any different input label is considered as background. This requires that the input label must also be cached in the row buffer. Equality comparisons with the input label are required to create a foreground and background based on the incoming pixel. Since both foreground and background are being labelled, potentially twice as many labels are required. Consequently, the merger table and data tables need to be twice as large. Otherwise, the processing is the same as the single label case.

11.4.6 Further Optimisations

If the regions being labelled or extracted are convex, then further simplifications are possible. Since there are no **U** shapes (this would be non-convex), a relatively simple label propagation is sufficient to completely label the image in a single pass. The merger logic associated with the previous techniques is no longer required. The only slight complication arises with ⌋ shaped regions, such as the left hand object in Figure 11.12. This can occur along the top left edge of convex objects and it is necessary to prevent a new label from being assigned. One solution to overcoming this problem is to defer labelling the pixels in a row until the first background pixel is encountered. A simple approach is to run-length encode the pixels stored in the row buffer and output the streamed labelled image as it is being reconstructed from the row buffer.

11.5 Distance Transform

Another form of labelling object pixels is provided by the distance transform. This labels each object pixel with its distance to the nearest background pixel. Different distance metrics result in different distance transforms. The Euclidean distance corresponds with how distances are measured in the real world. It is based on the L_2 norm and is given by:

$$||\Delta x, \Delta y||_{L_2} = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{11.13}$$

One difficulty with using the Euclidean distance is that it is difficult to determine which pixel is actually the closest background pixel without testing them all. This is complicated by the fact that images are sampled with a rectangular sampling grid. Two other distance metrics which are easier to calculate are the city block or Manhattan distance metric and the chessboard metric. The city block distance is based on the L_1 norm:

$$||\Delta x, \Delta y||_{L_1} = |\Delta x| + |\Delta y| \tag{11.14}$$

This counts the number of horizontal and vertical pixel steps, considering neighbouring pixels which are four-connected. As a result, diagonal distances are overestimated because a diagonal step counts as two pixels. The chessboard metric is based on the L_∞ norm:

$$||\Delta x, \Delta y||_{L_\infty} = \max(|\Delta x|, |\Delta y|) \tag{11.15}$$

This is so called because it counts the number of steps a king would take on a chessboard. This considers pixels to be eight-connected, so underestimates diagonal distances because a diagonal connection only counts as one step.

These three metrics are compared in Figure 11.16. It can be clearly seen that the non-Euclidean metrics are anisotropic. This has led to a wide range of other metrics that give better approximation to the Euclidean distance, while retaining the ease of calculation of the other distance metrics.

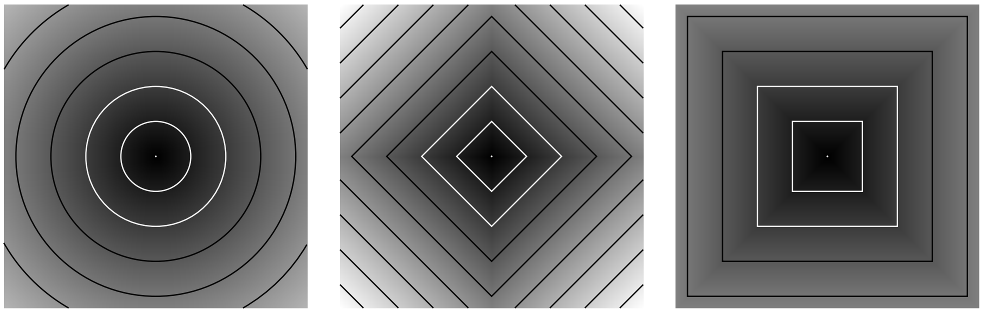


Figure 11.16 Comparison of distance metrics (contour lines have been added for illustration). Left: Euclidean, L_2; centre: city block, L_1; right: chessboard, L_∞.

Obviously, direct calculation of the distance transform by measuring the distance of every object pixel from every background pixel and selecting the minimum is impractical. Therefore, several different methods have been developed to efficiently calculate the distance transform of a binary image.

11.5.1 Morphological Approaches

One approach is to successively erode the object using a series of morphological filters. If each pass removes one layer of pixels, then the distance of a pixel from the boundary is the number of passes required for an object pixel to become part of the background. The chessboard distance is obtained by eroding with a 3×3 square structuring element, whereas the city block distance is obtained by eroding with a five element $+$ shaped structuring element. A closer approximation to Euclidean distance may be obtained by alternating the square and cross elements with successive iterations (Russ, 2002), which results in an octagonal shaped pattern. The main limitation of such algorithms is that many iterations are required to completely label an image. On an FPGA, the iterations may be implemented as a pipeline up to a certain distance (determined by the number of processors which are constructed). Beyond that distance, it is necessary to save the result as an intermediate image and feed the image back through.

To improve the approximation to a Euclidean distance, a larger structuring element needs to be used. This may be accomplished by having a series of different sized circular structuring elements applied in parallel (Waltz and Garnaoui, 1994b). Such a stack of structuring elements is equivalent to using greyscale morphology with a conical shaped structuring element. While using a conical structuring element directly is expensive, by decomposing it into a sequence of smaller structuring elements, it can be made computationally more feasible (Huang and Mitchell, 1994). However, for arbitrary sized objects, it is still necessary to iterate using finite sized structuring element.

11.5.2 Chamfer Distance

The iterative approach of morphological filters may be simplified by making the observation that successive layers will be adjacent. This allows the distance to be calculated through local propagations using stream processing (Borgefors, 1986; Butt and Maragos, 1998). The result is the chamfer distance transform.

Two raster scanned passes through the image are required. The first pass is a normal raster scan and propagates the distances from the top and left boundaries into the object.

$$Q_1[x,y] = \begin{cases} 0, & I[x,y] = 0 \\ \min\begin{pmatrix} Q_1[x,y-1] + a, Q_1[x-1,y] + a, \\ Q_1[x-1,y-1] + b, Q_1[x+1,y-1] + b \end{pmatrix}, & \text{otherwise} \end{cases} \tag{11.16}$$

The second pass is a reverse raster scan, propagating the distance from the bottom and right boundaries.

$$Q_2[x,y] = \min\begin{pmatrix} Q_1[x,y], Q_2[x,y+1] + a, Q_2[x+1,y] + a, \\ Q_2[x+1,y+1] + b, Q_2[x-1,y+1] + b \end{pmatrix} \tag{11.17}$$

An implementation of the chamfer distance transform is given in Figure 11.17. A frame buffer is required to hold the distances between passes. Note that the streamed output is in a reverse raster scan format. An FPGA-based implementation of such a distance transform is given in (Hezel *et al.*, 2002).

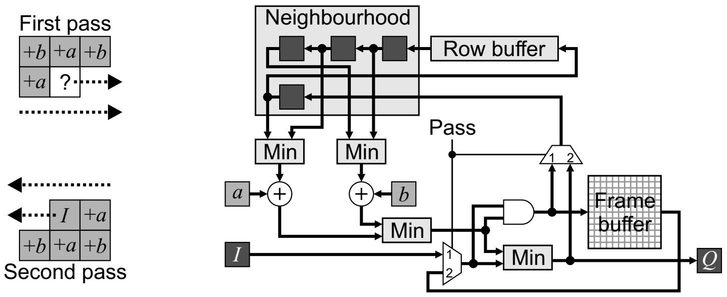


Figure 11.17 3 × 3 chamfer distance transform. Left: propagation increments with each pass; right: hardware implementation reusing the circuitry between the passes.

Higher throughput can be obtained by using separate hardware for each pass, with a bank switched frame buffer in between.

Different values of a and b will result in different distance metrics. The chessboard distance results when $a = b = 1$. Using only four-connected points ($a = 1, b = \infty$) gives the city block distance. Closer approximation to the Euclidean distance is given with $a = 1$ and $b = \sqrt{2}$. To maintain integer arithmetic, the best approximation for small integers is given with $a = 3$ and $b = 4$, and dividing the result by three. Since the division by three is not easy in hardware, the result can either be left undivided, or using $a = 2$ and $b = 3$, dividing the result by two (a right shift by one bit). The chamfer distance metric is given by:

$$\begin{aligned} ||\Delta x, \Delta y||_{C_{a,b}} &= b\min(\Delta x, \Delta y) + a|\Delta x - \Delta y| \\ &= (2a + b)L_\infty - (a + b)L_1 \end{aligned} \tag{11.18}$$

A closer approximation to the Euclidean distance may be obtained by using a larger window size. A 5 × 5 window, with weights as shown in Figure 11.18, with $a = 5$, $b = 7$ and $c = 11$, gives the best approximation with small integer weights (Borgefors, 1986; Butt and Maragos, 1998). (Weights are not required for the blank spaces because these are multiples of smaller increments.)

The two-pass chamfer distance algorithm may be adapted to measure Euclidean distance by propagating vectors $(\Delta x, \Delta y)$ instead of scalar distances. However, to correctly propagate the distance to all points requires three passes through the image (Danielsson, 1980; Ragnemalm, 1993; Bailey, 2004). The intermediate frame buffer also needs to be wider because it needs to store the vector. A further limitation is that because of the discrete nature of images, local propagation of vectors can leave some points with a small error relative (less than one pixel) to the Euclidean distance (Cuisenaire and Macq, 1999).

The chamfer distance transform may be further accelerated by processing multiple lines in parallel (Trieu and Maruyama, 2006). This requires sufficient memory bandwidth to read data for several lines simultaneously. (Horizontal data packing is more usual, but by staggering the horizontal accesses and

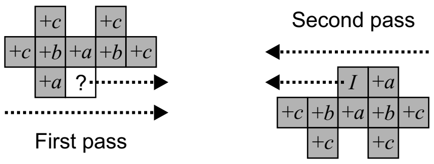


Figure 11.18 Masks for a 5 × 5 chamfer distance transform.

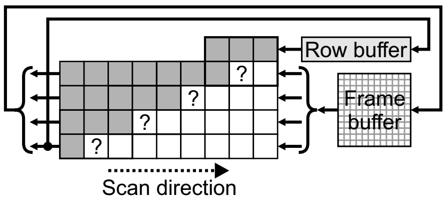


Figure 11.19 Processing multiple lines in parallel with the chamfer distance.

using short FIFOs, the same effect can be achieved.) The processor for each successive line must be offset, as shown in Figure 11.19, so that it is working with available data on the previous line. Depending on the frame buffer access pattern, short FIFOs may be required to delay the data for successive rows and to assemble the processed data for saving back to memory. Only the single row that overlaps between the strips needs to be cached in a row buffer. The same approach can also be applied to the reverse scan through the image.

11.5.3 Separable Transform

In most applications, the chamfer distance is sufficiently close to the Euclidean distance to be acceptable. If, however, the true Euclidean distance transform is required, it may be obtained by using a separable algorithm (Hirata, 1996; Maurer *et al.*, 2003; Bailey, 2004). This relies on the fact that squaring Equation 11.13 gives:

$$||\Delta x, \Delta y||_{L_2}^2 = (\Delta x)^2 + (\Delta y)^2 \tag{11.19}$$

which is separable. This allows the one-dimensional distance squared transform to be obtained of each row first, and then of each column of the result. The processing also only requires integer arithmetic, since Δx and Δy are integers. If the actual distance is required (rather than distance squared), then a square root operation may be pipelined on the output of the column processing. Since each row (and then column) is processed independently, the algorithm is readily parallelisable, and is also easily extendable to three or higher dimensional images.

The first stage, processing along the rows, requires two passes, one in the forward direction and once in the reverse direction. The pass in the forward direction determines the distance from the left edge of the object and the reverse pass updates with the distance from the right edge. The processing for each of the two passes is:

$$\begin{aligned} Q_1[x,y] &= \begin{cases} 0, & I[x,y]=0 \\ Q_1[x-1,y]+1, & \text{otherwise} \end{cases} \\ Q_2[x,y] &= \min(Q_1[x,y], Q_1[x+1,y]+1) \end{aligned} \tag{11.20}$$

At the start of each pass Q_1 and Q_2 need to be initialised based on the boundary condition. If it is assumed that outside the image is background, they need to be initialised to zero, otherwise they need to be initialised to the maximum value. The circuit for implementing the scanning is given in Figure 11.20. Since the input, I, is binary, the multiplexer reduces to a set of AND gates. The addition of one in the feedback loop needs to be performed with saturation, especially if the outside of the image is considered to be object. Note that the data is read from the row buffer in reverse order during the

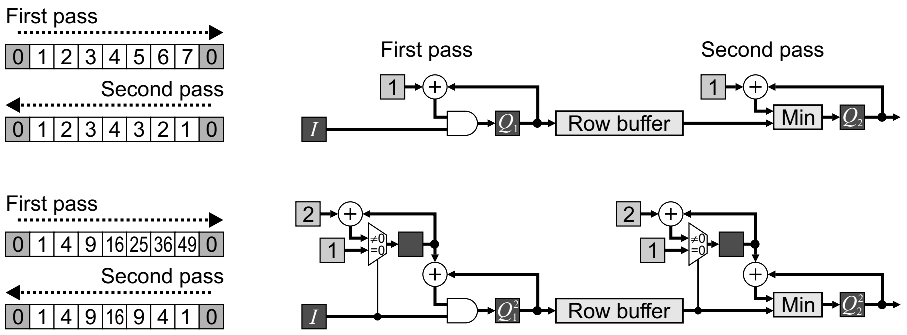


Figure 11.20 Row transform for the separable Euclidean distance transform. Top: performing the distance transform on each row; bottom: the distance squared transform using incremental update; left: example showing the result for a simple example.

second pass. Therefore, if directly processing streamed data, the row buffer needs to be able to hold two rows, or a pair of row buffers with bank switching can be used. Since the column processing stage requires the distances squared, these can be calculated directly using incremental processing techniques to avoid the multiplication (Section 5.4.5) as shown in the bottom of Figure 11.20.

The column processing is a little more complex. Finding the minimum distance at a point requires evaluating Equation 11.19 for each row and determining the minimum:

$$D^2[x,y] = \min_{y_i}\left(Q_2^2[x,y_i] + (y-y_i)^2\right) \tag{11.21}$$

Rather than test every possible row within the column to find the minimum, it can be observed that the contributions from any two rows (say y_1 and y_2) are ordered. Let $D_1^2 = Q_2^2[x,y_1]$ and $D_2^2 = Q_2^2[x,y_2]$. Then, as shown in Figure 11.21, the distances from these two points divide the column into two at $y'_{1,2}$ where:

$$D_1^2 + \left(y_1-y'_{1,2}\right)^2 = D_2^2 + \left(y_2-y'_{1,2}\right)^2 \tag{11.22}$$

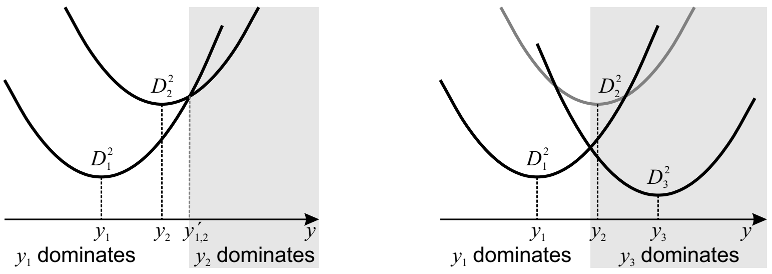


Figure 11.21 Regions of influence. Left: two rows on a column; right: adding a third row that results in row y_2 having no influence. (With kind permission from Springer Science + Business Media: *Lecture Notes in Computer Science*, "An efficient Euclidean distance transform", **3322**, © 2004, 394–408, D.G. Bailey, Figure 7).

Solving for the boundary of influence gives:

$$\begin{aligned} y'_{1,2} &= \frac{D_2^2 - D_1^2 + y_2^2 - y_1^2}{2(y_2 - y_1)} \\ &= \frac{1}{2}\left(\frac{D_2^2 - D_1^2}{y_2 - y_1} + y_2 + y_1\right) \end{aligned} \tag{11.23}$$

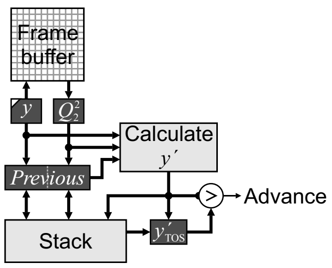
The logic for implementing the column process calculating the distance transform is shown in Figure 11.22. The row number is used to read the row distance squared from the frame buffer. These are combined with the *Previous* values to give the boundary of influence using Equation 11.23. Note that the division only needs to be an integer division because the row numbers are only integers and it is only necessary to determine which two rows the boundary falls between. If the boundary is greater than that on the top of the stack (cached in the y'_{TOS} register) then the *Previous* values are pushed onto the stack, y is incremented and the next value read from memory. If less than or equal, then the *Previous* values have no influence, so the top of the stack is popped off into the *Previous* and y'_{TOS} registers and the comparison repeated.

Although y is not necessarily incremented with every clock cycle, the number of clock cycles required for the first pass through the column is at most twice the number of rows. Therefore, to process one pixel per clock cycle, two such units must operate in parallel. The sporadic nature of frame buffer reads can be smoothed by reading from the frame buffer into a FIFO and accessing the distance values from the FIFO as necessary.

At the end of the first pass down the column, the stack contains a list of all of the regions of influence in order. It is then simply a case of treating the stack as a queue, with each section used to directly calculate the distance. When the end of the region of influence is reached, the next section is simply pulled from the queue until the bottom of the column is reached. The result is a streamed output image, with a vertical scan pattern rather than the conventional raster scan.

Row processing is actually the same operation as column processing. The simpler algorithm results from the prior distances all being zero for the background and infinite for the object. Therefore, the only pixels that have influence are the background points at each end of the groups of object pixels on a row, with the boundary of influence at the midpoint of each connected sets of pixels.

As described here, the row processing has a latency of two row lengths as a result of the combination of the forward and reverse pass. The latency may be reduced to one half a row time by also performing the second pass in the forward direction. During the first pass, the correct distance is generated for pixels up to



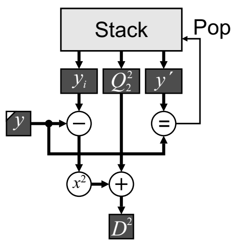


Figure 11.22 Column transform for the separable Euclidean distance transform. Left: first pass determining the regions of influence; right: second pass, expanding the regions of influence to give the distance.

the midpoint. Therefore, these do not need to be tested again. However, the midpoint is not determined until the first background pixel is reached at the end of the object. At this point, the second pass can go back to the midpoint and relabel the last half of the object, counting down rather than up.

The latency of processing a column, as described above, is two column times (the worst case time it takes for the first pass down the column). This may be reduced slightly by beginning the second pass through the image before the first pass has been completed.

The separability of the transform implies that all of the rows must be processed before beginning any of the columns. However, this is not necessarily the case. As with separable warping (Section 9.1.1), the two stages can be combined. Rather than process each column separately, they can be processed in parallel, with the stacks built in parallel as the data is streamed from the row processing. This is more complex with the distance transform because the volume of data required to hold the stacks would almost certainly require use of external memory. If the outside of the image can be considered background, the worst case latency would be just over half a frame, with conventional raster scanned input and output streams. The improved latency comes at the cost of more complex memory management. However, if outside the image is considered object, then it is necessary to process the complete frame in the worst case before any pixels can be output (for example if the only background pixel is in the bottom right corner in the image).

Since each row and each column are processed independently, an approach to accelerating the algorithm is by operating several processors in parallel. The main limitation to processing speed is governed by memory bandwidth for the input, intermediate storage and output streams.

11.5.4 Applications

The distance transform has a number of applications in image processing (Fabbri *et al.*, 2008). One use is to derive features or shape descriptors of objects. Statistics of the distances within an object, such as mean and standard deviation, are effective size and shape measures. Ridges within the distance transform correspond to the medial axis or skeleton of the object (Arcelli and Di Baja, 1985). The distance to the medial axis is therefore half the width or thickness of the object at that point. Extending this further, local maxima correspond to centres of largest inscribed ‘circles’.

If objects within the image each have only a single local maximum in their distance transform, then the distance transform can be used to provide a count of the number of objects within the image even if adjacent objects are touching. This concept may be extended to the separation of touching objects through the watershed transform (see the next section).

Fast binary morphological filtering can be accomplished by thresholding the distance transformed image. Rather than perform several iterations with a small structuring element, the distance transform effectively performs these all in one step, with the thresholding selecting the particular iteration.

The distance transform can be used for robust template matching (Hezel *et al.*, 2002) or shape matching (Arias-Estrada and Rodriguez-Palacios, 2002). The basic principle here is to compare the distance transformed images rather than simple binary images. This makes the matching less sensitive to small differences in shape, resulting for example from minor errors in segmentation.

11.5.5 Geodesic Distance Transform

A variation on the distance transform is the geodesic distance. The geodesic distance between two points within a region is defined as the shortest path between those two points that lies completely within the region. Any of the previously defined distance metrics may be used, but rather than measure the distance from the boundary, the distance from an arbitrary set of points is used. The previously described methods need to be modified to measure the geodesic distance.

Morphological methods use a series of constrained dilations beginning with the starting or seed pixels. The input is dilated from the seed pixels, with each iteration ANDed with the shape of the region to constrain the path to lie within the object. The distance from a seed point to an arbitrary point is the number of dilations required to include that point within the growing region.

Using larger conical structuring elements to perform multiple iterations in a single step does not work with this approach because a larger step may violate the constraint of remaining within the region.

The chamfer distance transform can also be used to calculate the geodesic distance. In general, two passes will not be sufficient for the distances to propagate around concavities in the region. The back and forth passes must be iterated until there is no further change.

Unfortunately, the constrained distance propagation cannot be easily achieved using stream processing. An approach that requires random access to the image can be built on a variation of Dijkstra's shortest path algorithm (Dijkstra, 1959). The seed pixels are marked within the image and are inserted into a FIFO queue. Then, pixels are extracted from the queue and their unmarked neighbours determined. These neighbours are marked with the incremented distance and their locations added to the end of the FIFO queue. The queue therefore implements a breadth-first search, which ensures that all pixels of one distance are processed and labelled before any of a greater distance. A Euclidean distance transform may be approximated by selecting four-neighbour or eight-neighbour propagation depending on the level. A potentially large FIFO queue is required to correctly process an arbitrary image. During the second phase, pixels are accessed in a random order. Searching for the unmarked neighbours requires multiple accesses to memory. An FPGA implementation of this algorithm was described by Trieu and Maruyama (2008).

Rather than use a FIFO queue, it is also possible to use chain codes as an intermediate step (Vincent, 1991). Propagating chain codes reduces the number of memory accesses because the location of unmarked pixels can come directly from the chain code, and only a single access is required to verify that the pixel is unmarked and apply the distance label.

One application of the geodesic distance is for navigation and path planning for robotics (Sudha and Mohan, 2008). The geodesic distance transform enables both obstacle avoidance and determining the shortest or lowest cost path from source to destination.

11.6 Watershed Transform

The watershed transform is closely related to connected components labelling. It considers the pixel values of an image as a topographical map and segments an image based on the topographical watersheds. There are two main approaches to determining the watersheds. The first is to consider a droplet of water falling on each pixel within the image. The droplet will flow downhill until it reaches a basin where it will stop. The image is segmented based on grouping together all of the pixels that flow into the same basin. The second approach reverses this process and starts with the basins, gradually raising the water level creating lakes of pixels associated with each basin. When two separate lakes meet, a barrier is built between them; this barrier is the watershed (Vincent and Soille, 1991).

11.6.1 Flow Algorithms

The first approach can be implemented in much the same way as connected components analysis (Bailey, 1991). The structure of the implementation is illustrated in Figure 11.23. The input stream of pixel values is augmented with an initial label of zero (unlabelled). With eight-connectivity, the neighbourhood must be extended to a 3×3 window to enable the direction of the minimum neighbour to be determined. If ambiguous, one of the neighbours can be chosen arbitrarily (or the gradient within the neighbourhood could be used to resolve the ambiguity; Bailey, 1991). If no pixels are less than the central pixel, it is a new local minimum (basin) and is assigned a label if it does not already have one. Otherwise a pixel is connected to its neighbourhood minimum. If neither is labelled, a new label is assigned to both pixels. If one is labelled, the label is propagated to the other pixel. If both are labelled, then a merger occurs; this is recorded in a merger table, in the same way as with connected components labelling.

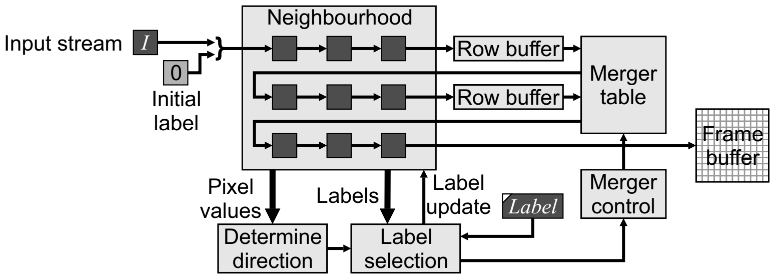


Figure 11.23 Stream-based implementation of watershed segmentation.

The merger table is a little more complicated than with connected component labelling, because potentially two accesses need to be made to update the labels emerging from the row buffer. This can be handled by running the merger table at twice the clock speed, or by using two parallel merger tables with identical entries. A further possibility is a lazy caching arrangement, where a label is translated only if necessary (if it is the centre pixel or the neighbourhood minimum).

If a labelled image is required, then two passes through the image are necessary. The first pass performs an initial labelling and determines the connectivity. The initial labels are then saved in a frame buffer, as shown in Figure 11.23, and the merger table used to provide a final consistently labelled image in the second pass. Alternatively, if the data for each region can be extracted in the first pass (as with single pass connected components analysis in Section 11.4), then the frame buffer is not required but can be replaced with a data table (Bailey, 1991).

One limitation of the above algorithm is that it does not divide plateaus which lie on a watershed evenly. Usually the whole of a plateau is assigned to a single label. Generally it is better to divide plateaus down the ‘middle’ with half being assigned to each watershed. This can be accomplished by distance transforming any plateaus, assigning pixels to the minimum that is the closest. This is a geodesic distance transform, which requires two or more passes through the image. This approach was taken by Trieu and Maruyama (Trieu and Maruyama, 2006; 2007), where multiple backward and forward scans are taken to calculate the geodesic distance transform of the plateaus where necessary. Because of the large volume of data that needs to be maintained, all of the data structures were kept in external memory. To accelerate their algorithm, they had multiple memory banks and processed multiple rows in parallel using the scheme introduced in Figure 11.19. This approach also has its own limitations; in particular, the number of back and forward scans through the image.

Trieu and Maruyama (Trieu and Maruyama, 2008) adapted the distance transform described in the previous section. The new scheme requires four processing phases. The first phase is a raster scan to identify the plateau boundaries with smaller pixel values. The locations of these boundary pixels are placed in a FIFO queue. In the second phase, the FIFO is used to propagate the boundary pixels distance transforming the plateaus. The third phase is another raster scan, which locates local minima and unlabelled plateaus, which must correspond to the basins. When a basin is found, the raster scan is suspended and the FIFO queue is used again to propagate the labels for each region (this is the fourth phase). Once the region is completed, the all pixels which drain into that basin have been segmented, so the raster scan of phase three is resumed.

11.6.2 Immersion Algorithms

The ‘traditional’ immersion approach to determining the watershed begins with the basins and increases the water level sequentially throughout the whole image, adding pixels to adjacent connected regions.

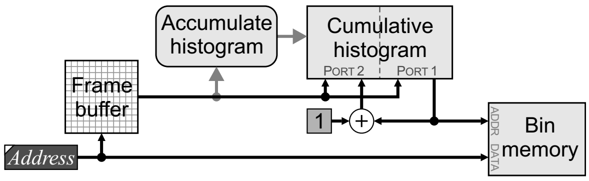


Figure 11.24 Bin sort, using the cumulative histogram for bin addressing.

This process considers the pixels within the image in strict pixel value order. Approached naïvely, this would require one pass through the image for each pixel value (requiring 2^N passes for an N-bit image). Since these successive passes through the image are in the same order, they can be pipelined, although this comes at the cost of one processor for each different pixel value within the image.

The alternative is to sort the pixels in the image into increasing order. Traditional sorting algorithms do not scale well with the size of the image, but, since the pixel values are discrete, fast methods that are linear with the number of pixels are possible. A bin sort can sort all of the pixels in a single pass through the image. This requires having 2^N bins. As the image is scanned, the coordinates of each pixel are added to the corresponding bin. Potentially, each bin could be the size of the image, although the total number in all of the bins is, of course, identical to the size of the image. The standard approach to avoid this overhead is to obtain a histogram of the image first, so that it is known in advance how many pixels occupy each bin. The cumulative histogram then gives the address in bin memory of the end of each bin. If the cumulative histogram is offset by one pixel value, so the count for pixel 0 is address 1, and so on, then the cumulative histogram will act as address translation for the incoming pixel values. As shown in Figure 11.24, the incoming pixel value is looked up in the cumulative histogram and used to store the pixel address in the bin memory. The bin memory address is incremented and written back to the cumulative histogram so that it will point to the next location. At the end of the image, the cumulative translation memory again contains the cumulative histogram and all of the pixels are sorted.

While the histogram-based bin sort is memory efficient, a disadvantage is that it requires two passes through the image. The first pass is to accumulate the histogram, which can be performed in parallel with the data being streamed into the frame buffer. The second pass does the actual pixel sorting. An alternative approach that avoids the need to pre-allocate the bin memory is to use an array of linked lists, one for each bin (De Smet, 2010). With a linked list, it is not necessary to know the bin size in advance.

A circuit for the construction of the linked lists is shown in Figure 11.25. Prior to processing the input stream, the entries in the *Tail* array need to be set to zero. (If the size of the input stream is not a power of two, they could be set to an invalid address after the end of the stream, for example all ones.) When the first pixel value of a particular value comes in, the corresponding *Tail* entry will be zero, so the pixel address stored in the corresponding *Head* entry. Otherwise the address is stored in the linked list at the entry

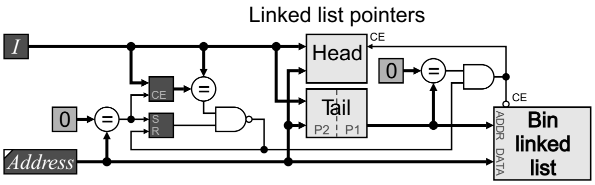


Figure 11.25 Bin sort using an array of linked lists.

pointed to by the *Tail*. In both cases, *Tail* is updated with the new address. If the *Tail* is initialised to zero, a little bit of additional logic is required to distinguish the case when a true zero address is written into the *Tail* register. This may be done as illustrated in Figure 11.25, or alternatively an additional bit could be added to the width of *Tail* to indicate this.

Since only one *Head* or *Tail* register is accessed at a time, they can be stored in either fabric RAM or block RAM. *Head* only needs to be single port, while *Tail* will need to be dual-port. The image addresses do not need to be stored explicitly because the linked list is structured such that the bin address corresponds to the image address. The pixel values also do not need to be saved within the bins because this is implicit with the linked list structure as each different pixel value has a separate linked list. However, since the neighbours need to be examined, this will require storing the image within a frame buffer. Construction of the linked list is easily parallelisable if memory bandwidth allows. Since bin addressing corresponds to image addressing, the image may be split vertically into a number of blocks, which are processed in parallel. At the end, it is then necessary to add the links between the blocks for each pixel value.

When performing the immersion, pixels are accessed in the order of pixel value from the sorted list or queue. Pixels are taken from the queue and added to the existing boundaries if an adjacent pixel has a lower level. The label is propagated from adjacent labelled pixels. If no smaller labelled pixels are in the neighbourhood, the pixels are added to a second queue to perform the geodesic distance transform of the plateau. Any unlabelled pixels remaining after this process are new basins, so are assigned a new label. An FPGA implementation of this process is described by Rambabu *et al.* (2002; Rambabu and Chakrabarti, 2007).

11.6.3 Applications

The primary use of the watershed transform is in segmentation. After applying an edge detection operator, the edges become ridges in the image which become the watersheds separating the regions. The watershed will tend to divide the image along the edges between regions. One limitation is that any noise within the image will create false edges, with consequent over-segmentation. This can be partially addressed by smoothing before segmentation and ignoring edges below a threshold height. When using the immersion method, the threshold may be implemented by allowing adjacent regions to join if the basin depth is less than the threshold below the current immersion level. This process is illustrated for a noisy image in Figure 11.26.

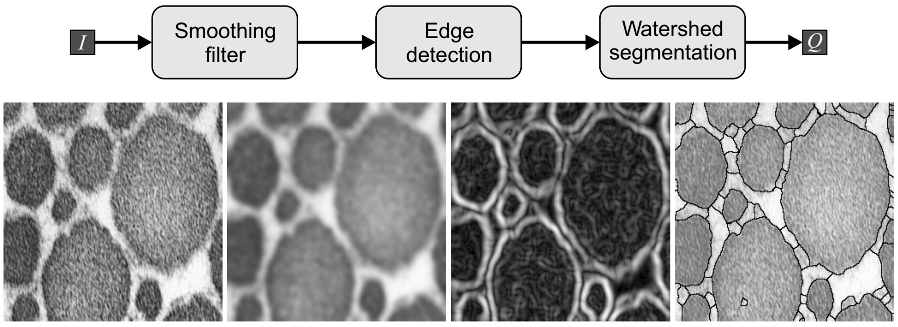


Figure 11.26 Watershed segmentation. Images from the left: original input image; Gaussian filtered with $\sigma = 2.3$; edge detection using the range within a 3×3 window; watershed segmentation merging adjacent regions with a threshold of five.

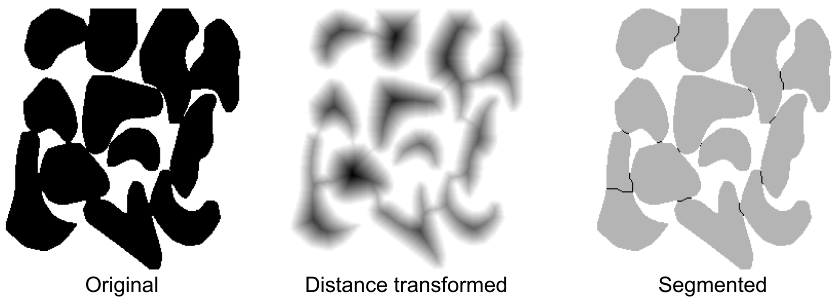


Figure 11.27 Watershed segmentation for separating touching objects. Left: original image; centre: applying the 3,4 chamfer distance transform; right: original with watershed boundaries overlaid.

Another application of the watershed transform is to separate touching objects, as illustrated in Figure 11.27. The basic principle is that convex objects (and some non-convex objects) will have a single peak when distance transformed. Therefore, if multiple objects are touching, then a blob will have multiple peaks within its distance transform. Inverting the distance transform will convert the peaks to basins, where the watershed transform will associate each pixel within a blob with the basin into which it flows. The one failure in Figure 11.27 is for the non-convex object, which has two peaks in its distance transform. Again, it is necessary to use a small threshold because with discrete images the distance transform can have small peaks along a diagonal ridge. However, when applying the watershed transform after a distance transform, there will be no plateaus. This can simplify the algorithm considerably.

The watershed transform can be used in other applications where detecting peaks is of interest. One example of this is detecting the peaks following a Hough transform, described in the following section. Inverting the image will convert peaks to basins, allowing them to be individually labelled.

11.7 Hough Transform

The Hough transform is a technique for detecting objects or features from detected edges within an image (Illingworth and Kittler, 1988; Leavers, 1993). While originally proposed for lines (Hough, 1962), it can readily be extended to any shape that can be parameterised. The basic idea behind the Hough transform is represented in Figure 11.28. Firstly, the image is processed to detect edges within the image. Each edge pixel then votes for all of the sets of parameters with which it is compatible, that is an object with those

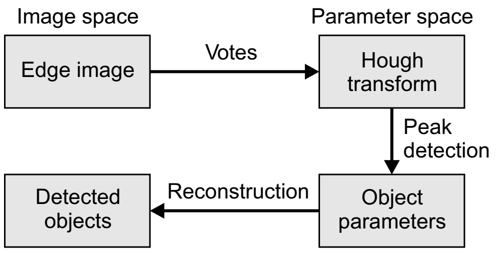


Figure 11.28 The principle behind using the Hough transform to detect objects.

parameters would have an edge pixel at that point. This voting process is effectively accumulating a multidimensional histogram within parameter space. Sets of parameters which have a large number of votes have significant support for the corresponding objects within the image. Therefore, detecting peaks within parameter space determines potential objects within the image. Candidate objects can then be reconstructed from their parameters, and verified.

The idea of voting means that the complete object does not need to be present within the image. There only needs to be enough of the object to create a significant peak. The Hough transform therefore copes well with occlusion. Noise and other false edges with the image will vote for random sets of parameters, which will overall receive relatively little support. Therefore, the Hough transform is also insensitive to noise. This makes it a powerful technique for object or feature detection.

11.7.1 Line Hough Transform

Lines are arguably the simplest image feature, being described by only two parameters. The standard equation for a line is:

$$y = mx + c \tag{11.24}$$

with parameters m and c. A detected edge point, (x, y) will, therefore, vote for points along the line:

$$c = y - mx \tag{11.25}$$

in $\{m, c\}$ parameter space. Such a parameterisation is suitable for lines which are approximately horizontal, but vertical lines have large (potentially infinite) values of m and c. Such non-uniformity makes both vote recording and peak detection expensive and impractical. A more usual parameterisation of lines (Duda and Hart, 1972) is:

$$x \cos\theta + y \sin\theta = \rho \tag{11.26}$$

where θ is the angle of the line and ρ represents the closest distance between the line and the image origin. A point (x, y) votes for points along sinusoids in $\{\theta, \rho\}$ parameter space. This process is illustrated with an example in Figure 11.29.

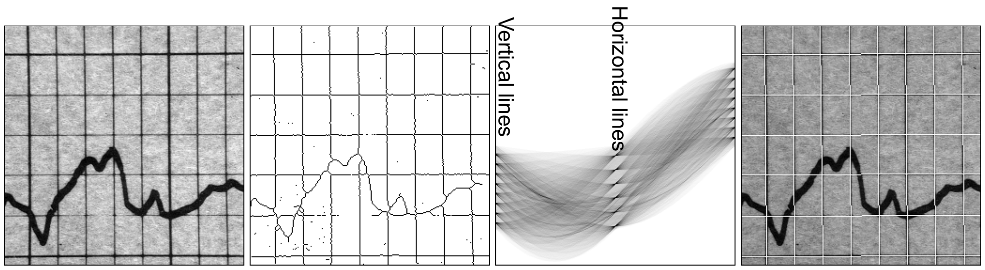


Figure 11.29 Using the Hough transform to detect the gridlines of a graph. From the left: input image; after thresholding and thinning the lines; Hough transform; detected gridlines overlaid on the original image.

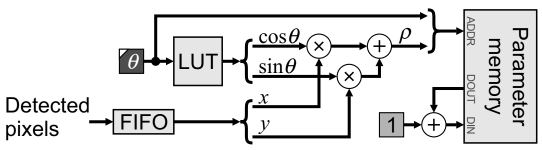


Figure 11.30 Vote accumulator for line Hough transform.

11.7.1.1 Parameter Calculation

The main problem with the Hough transform is its computational expense. This comes from two sources. The first is the calculation of the sine and cosine to determine the sets of parameters where each point votes. Normally θ is incremented from 0 to π and the corresponding ρ calculated from Equation 11.26. Since the values of θ are fixed by the bin spacing, the values of $\sin\theta$ and $\cos\theta$ can be obtained from a lookup table. These then need to be multiplied by the detected pixel location, as shown in Figure 11.30. The size of the lookup tables can be reduced by a factor of four (for θ in the range from 0 to $\frac{\pi}{4}$) with a little extra logic and exploiting trigonometric identities (Cucchiara *et al.*, 1998).

Mayasandra *et al.* (2005) used a bit-serial distributed arithmetic implementation that enabled the addition to be combined with the lookup table, with a shift and add accumulator to perform the multiplication. This significantly reduces the hardware at the cost of taking several clock cycles to calculate each point. (This was addressed by having multiple processors to calculate all angles in parallel.)

Lee and Evagelos (2008) used the architecture of Figure 11.30 using the logarithmic number system to avoid multiplications. They then converted from the logarithmic to standard binary for performing the final addition.

Alternatively, a CORDIC rotator of Equation 5.35 could be used to perform the whole computation of Equation 11.26. This option was used by Karabernou *et al.* (2005).

The parameter space could also be modified to simplify the calculation of the accumulator points. This is one of the advantages of the linear parameterisation of Equation 11.25, as incremental update can be used to calculate the intercepts for successive slopes. Tagzout *et al.* (2001) used a modified version of Equation 11.26 and used the small angle approximation of sine and cosine to enable incremental calculations to calculate the parameters for a series of angles. However, they still needed to reinitialise the calculation periodically, about every 10 degrees, to prevent accumulation of errors.

11.7.1.2 Vote Accumulation

The second problem with the Hough transform is the memory access bandwidth associated with incrementing the bins in parameter space as the votes are accumulated. Each detected pixel in the image votes for many bins within parameter space, with each vote requiring two accesses to memory (a read and a write). The parameter memory will usually be off-chip, which contributes to the bandwidth problem. If parameter space is coarsely quantised, then it may fit within distributed on-chip block RAMs. This would allow multiple bins to be updated in parallel, at the expense of multiple processors to calculate the line parameters.

The adaptive Hough transform reduces the memory requirements by performing the Hough transform in two or more passes (Illingworth and Kittler, 1987). The first pass uses a relatively coarse quantisation to find the significant peaks, but gives the parameters but with low accuracy. A second pass uses a higher resolution quantisation within those specific detected areas to refine the parameters. The adaptive approach significantly reduces both the total computational burden and the memory requirements, enabling it to be implemented with on-chip memory. Obviously the edge data needs to be buffered between passes.

The memory bandwidth issue may be addressed by not accumulating votes for every angle. Edge pixels are usually detected using an edge detection scheme that can also give the edge orientation. Knowing the edge orientation restricts the accumulation to a single angle (or small range of angles). Not only does this reduce the number of calculations, but it also reduces the clutter within parameter space, making peak detection more reliable. This approach has been used to accelerate FPGA implementations (Cucchiara *et al.*, 1998; Karabernou *et al.*, 2005).

Figure 11.30 also shows a FIFO buffer on the incoming detected pixel locations. Although the proportion of detected pixels is relatively low for normal images, these pixels tend to be clustered rather than evenly distributed. A FIFO buffer allows the Hough transform accumulation to run relatively independently of the pixel detection process.

11.7.1.3 Finding Peaks and Object Features

A single scan through the parameter space with a window is suitable for finding the peaks. A peak detection filter can be implemented with a small (3×3) window. If necessary, adjacent counts can be combined to give the centre of gravity of the peak with increased resolution. During the peak scan pass, it is also useful to clear the accumulator memory to prepare it for the next image. This is easy to do with dual-port on-chip memory, but is a little harder with off-chip memory.

While the location of a peak gives the parameters of the corresponding line, often this corresponds with only a line segment within the image. It is therefore necessary to search along the length of each line detected to find the start and end points. These extended operations are not discussed in many FPGA implementations. One exception is that of Nagata and Maruyama (2004). To give good processing throughput, the stages of edge detection, parameter accumulation, peak detection and line endpoint detection were pipelined.

11.7.2 Circle Hough Transform

Detecting circles (or circular arcs) within an image requires a parameterisation of the circle (Yuen *et al.*, 1990). The commonly used parameterisation is in terms of the circle centre and radius $\{x_c, y_c, r\}$:

$$(x-x_c)^2 + (y-y_c)^2 = r^2 \tag{11.27}$$

With three parameters, the parameter space needs to be three-dimensional, with each detected point in the image voting for parameters on the surface of a cone. The increased dimensionality exacerbates the computational and bandwidth problems of the Hough transform.

Since two of the parameters relate directly to the position of the circle, the three-dimensional Hough transform may be recast as a three-dimensional convolution (Hollitt, 2009). The conventional approach of accumulating votes is equivalent to convolving the two-dimensional image of detected points with a three-dimensional conical surface. The convolution may be implemented in the frequency domain by taking the two-dimensional Fourier transform of the input image, multiplying it by the precalculated three-dimensional Fourier transform of the cone, and then calculating the inverse Fourier transform.

As with the line Hough transform, detecting the orientation of the edge helps to reduce the computational dimensionality. The direction of steepest gradient will point either towards or away from the centre of the circle. Rather than accumulate on the surface of a cone, this reduces to accumulating points along a ray in three-dimensional space. Incremental update can be used to simplify the calculation of successive points along this ray. The reduction in number of spurious points accumulated will also reduce the clutter within parameter space. However, small errors in estimating the edge orientation will mean that the peak is a little less focussed, particularly with larger radii.

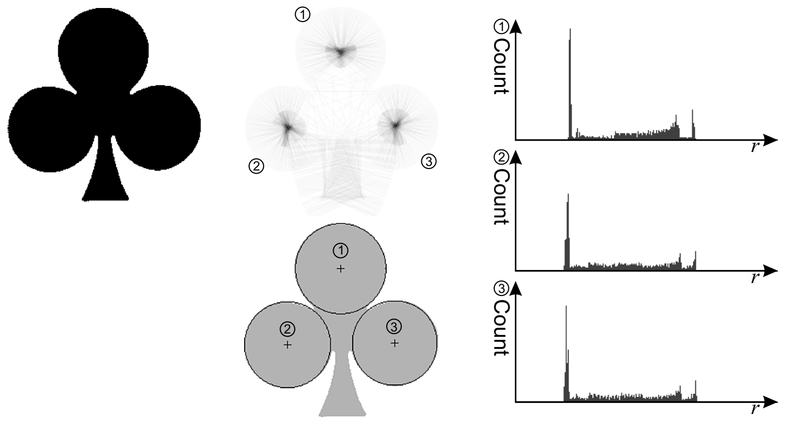


Figure 11.31 Two-step circle Hough transform. Left: original image; centre top: two-dimensional transform to detect circle centres; right: radius histograms about the detected centres; centre bottom: detected circles overlaid on the original image.

However, the reduction in clutter allows a further reduction in dimensionality. Rather than draw the rays in three-dimensional space, the parameter space can be reduced to two dimensions by projecting along the radius axis (Bailey, 1992). A two-step process can then be used to determine the three parameters (Illingworth and Kittler, 1987; Davies, 1988); this is illustrated with an example in Figure 11.31. The first step is the two-dimensional projected Hough transform accumulating the rays towards the circle centre. These will reinforce at the circle centre, enabling two of the parameters to be determined. The radius may then be determined from a one-dimensional Hough transform (histogram) of the distances of each point from the detected centres.

The two-pass approach requires that the list of detected points be buffered between passes. However, this is typically only a small fraction of the whole image, so may be stored efficiently (using chain codes is one possibility). Projecting the lines results in a fuzzy blob around the circle centres. When the peaks are found, these need to be smoothed (for example by calculating the centre of gravity) to obtain a better estimate of the peak location. In the second pass, several radius histograms may be accumulated in parallel, with the peak and the few radii on either side averaged to obtain the circle radius.

11.7.3 Generalised Hough Transform

The techniques described above can be generalised to arbitrarily shaped objects (Ballard, 1981). The object being detected is represented by a template, which is then parameterised by position and maybe also by scale and orientation. The generalised Hough transform is, therefore, a form of edge-based template matching.

Maruyama (2004) describes an FPGA implementation of a generalised Hough transform. The system was able to detect over 100 shapes per image using a two-stage process. The first stage uses reduced resolution (by a factor of 16×16) to locate and identify a shape approximately. The second stage uses full resolution, but focussed on the particular area and shape. This enabled frame rates between 13 and 26 frames per second for VGA resolution images (640×480).

Geninatti *et al.* (2009) used a variation of the generalised Hough transform to estimate the relative rotation between successive frames within a video sequence. To reduce the processing, only the DC components of MPEG compressed images were used, making the effective resolution 44×36 pixels.

11.8 Summary

The processing required for blob detection and labelling falls within the intermediate processing level within the image processing pyramid. The input is all in the form of image data, with the output in terms of segmented regions or even data extracted from those regions. While there are many possible algorithms for all of these operations, the primary focus in this chapter has been on stream processing. However, the non-local nature of labelling means that stream processing is not always possible.

The bounding box focuses purely on extracting size and shape information from each object based on simple pixel labelling. Consequently non-connected pixels are considered to belong to the same object if they have the same label. Full connected components labelling is required to take into consideration the connectivity. Producing a labelled image requires at least two passes, although if all of the required region data can be extracted in the first pass then the second pass is unnecessary.

Chain coding is an alternative region labelling approach, which defines regions by their boundaries, rather than explicitly associating pixels with regions. Chain coding can be accomplished in a single streamed pass, although this requires some quite complex memory management. Subsequent data extraction then works with the one-dimensional chains rather than the two-dimensional image.

Related intermediate level operations developed in this chapter were the distance transform, the watershed transform and the Hough transform. All can be used as part of segmentation or to extract data from an image. These require either multiple streamed passes through the image, or random access. (With the Hough transform, the random access is within parameter space.)

While other intermediate and high level image processing operations can also be implemented in hardware on an FPGA, these operations were selected as representative operations where significant benefits can be gained by exploiting parallelism. This does not mean that other image processing operations cannot be implemented in hardware. However, with many higher level operations, the processing often becomes data dependent. If implemented in hardware, separate hardware must be developed to handle each of the cases. As a result, the hardware is often sitting idle, waiting for the specific cases required to trigger its operation. There comes a point when the acceleration of the algorithm through hardware implementation becomes too expensive and complex relative to a software implementation.

12

Interfacing

An image processing system never stands alone. This is particularly the case for an embedded vision system, which is usually designed for one specific purpose or task. Therefore, there are usually one or more peripheral devices connected to the FPGA implementing the system.

Each device connected to the FPGA must have an appropriate interface within the FPGA. This is responsible for passing any data to or from the rest of the logic, including any format conversions required between the peripheral and the processor implementing the algorithm within the FPGA. It is also responsible for providing any control signals required by the peripheral device. This may range from a clock signal and simple handshaking through to complex protocol management. Many peripheral devices require initialisation or configuration through the setting of control registers.

The interface logic may be thought of as a device driver, shielding the image processing hardware from much of the lower level complexities of the physical interface (Bailey *et al.*, 2006). The primary focus of this chapter is on some of the techniques associated with the design of such interfaces. The term device driver may be a little misleading in this context, as it is referring primarily to interface hardware. If the system is made of a mix of programmable hardware and software, then the interface between them will also be in the form of a software device driver (Williams, 2009).

It is also useful within this context to consider the interaction between the FPGA system and the user. On a hosted system, the host computer will perform many of these tasks, using the host operating system. On a stand-alone system, however, all of these functions must be performed on the FPGA. One advantage of including an embedded processor is to provide a high level software environment for managing user interaction. Such tasks tend to be control-centric, making them more difficult to implement directly within the FPGA fabric. Most user interaction is not time critical, so hardware acceleration is usually not necessary. However, in some applications, particularly in the context of real-time debugging, it may be desirable to implement some of these functions directly in hardware (Bailey *et al.*, 2006; Buhler, 2007). This is a secondary focus of this chapter.

Many embedded applications are designed for performing a single task. Once configured, it is usually not necessary to change the configuration (apart from upgrading the application). Where dynamic configuration is used, it is most likely to be between predefined variants of an algorithm that are designed for specific conditions. Through reconfigurability, the algorithm is adapted to the current conditions. However, another use of FPGAs is for computation acceleration. In such cases, reconfigurability is more important depending on the particular task or task mix that is currently being executed. In this case it is important for the host system to be able to allocate a task or mix of tasks to an FPGA. This usually requires operating system support (Wigley and Kearney, 2001; So, 2007). Such techniques are beyond the scope of this book.

Before getting into details, there is one further general consideration that warrants discussion. Many external devices have their own natural clock frequency. It makes sense to interface with the device at its natural frequency. This will usually require that the low level driver span clock domains. The part of the interface that interacts directly with the hardware will be in the hardware clock domain. However, the part that connects to the core of the system will be in the main system clock domain. Since the domains are relatively asynchronous, it is important to consider the reliable communication between the domains, as discussed in more detail in Section 5.1.3. Buffer design is also important to ensure data is not lost in the transfer between domains.

12.1 Camera Input

In an embedded vision system, it is usually necessary to connect to a camera or other video input. There is quite a wide range of different camera formats and communication protocols, and each has its own interfacing requirements. While it is impossible to describe them all in any detail, the broad characteristics of most of them are introduced in the next section. Subsequent sections discuss particular interface issues: combining data from interlaced frames in the presence of motion; correcting the distortion associated with rolling shutter; and recovering colour data from a raw image captured with a Bayer pattern.

12.1.1 Camera Interface Standards

12.1.1.1 Analogue Video

A monochrome analogue camera encodes the video signal as an analogue voltage level that is proportional to the light intensity at a point in a scene. The output follows an interlaced raster scan, with the horizontal position coded with time, as illustrated in Figure 12.1. Sync pulses at the end of every scan line and every field are used to synchronise the system.

There are three common techniques for representing colour information. Component video has three separate signals, one for each of the red, green and blue components. S-video (also called Y/C) has two signals, one for the luminance and one for the chrominance. The two components of the chrominance are typically encoded by modulating a colour subcarrier with quadrature modulation (the SECAM standard uses frequency modulation instead). Composite video combines the luminance and chrominance together as a single signal.

There is also a range of different video standards. For NTSC, the signal consists of 525 scan lines (480 with actual video data) with a frame rate of 60 Hz. However, in both PAL and SECAM the signal has 625 lines (576 active) at a frame rate of 50 Hz.

Figure 12.2 is a block diagram showing the steps in the conversion of a composite video signal to digital before processing on an FPGA. The conversion to digital is shown here as the last step, but this can be moved to anywhere within the processing chain. Technically, it is possible to directly convert the incoming composite video signal to digital and perform all of synchronisation and colour separation

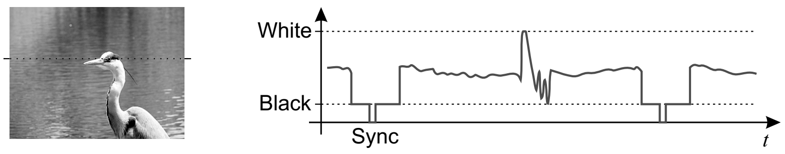


Figure 12.1 Analogue video signal. Left: input scene; right: video signal for the marked line. (Photo courtesy of Robyn Bailey.)

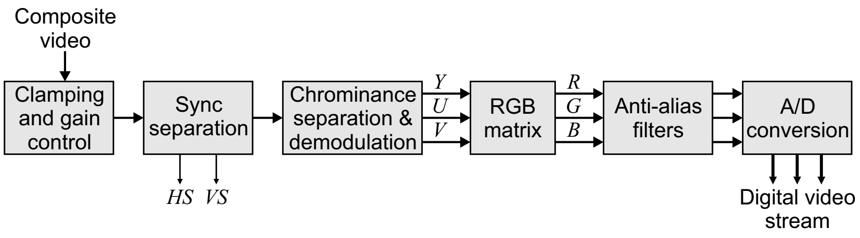


Figure 12.2 Typical functions performed by an analogue video codec.

within the FPGA. However, this is not recommended because the processing is quite complex and would use considerable resources on the FPGA. The required processing is also different depending on the particular video standard that is input. Instead, using an analogue video decoder chip will perform all of these functions, and most will automatically detect the signal standard and perform the appropriate processing.

Communication with the codec usually has two forms. The digital video stream, along with the pixel clock, horizontal and vertical sync signals are available in parallel and can be connected directly to the FPGA. Codec configuration (setting the gain, saturation control, signal standard, output format) and status query is often through a separate interface. Since this is much lower speed, a serial communication protocol such as I^2C is often used. Interfacing to I^2C is described in more detail in Section 12.3.2.

12.1.1.2 Direct Interface

Analogue cameras are increasingly being displaced by digital video cameras. These allow a much wider range of resolutions and directly provide a digital signal. In an embedded device, the best approach is to interface directly to the digital sensor chip. Direct interface is usually much the same as connecting to an analogue codec. The digital pixels are streamed from the sensor, with line and frame valid signals. Sensor configuration (shutter speed, triggering, gain control, windowing, subsampling mode) is controlled through a set of registers. Again a lower speed serial communication protocol is often used for this.

In a CMOS sensor each of the pixels are potentially individually addressable. Although such in interface is not usually provided to the user, several operating modes exploit this ability. Some of these modes commonly made available to the user are:

- **Windowing.** Rather than read out the entire image each frame, windowing allows a rectangular region of interest to be selected and only those pixels are read out. This allows the frame rate to be increased by reducing the area of the sensor that is read out. One application of this is object tracking in situations where the object being tracked only occupies a small fraction of the image area.
- **Skipping.** An alternative to windowing is skipping, where the number of pixels read out is reduced by skipping entire rows or columns. This reduces the pixel count without affecting the field of view.
- **Binning.** Rather than skipping pixels, adjacent pixels can be combined together. This has two advantages over skipping. Firstly, it reduces aliasing around sharp edges because adding adjacent bins is a form of low pass filtering. Secondly, since each pixel contributes to the output, binning can improve the sensitivity in low light conditions. Binning often has a lower frame rate because internally multiple pixels must be addressed to produce each output pixel.

A disadvantage of direct connection to the sensor is that the signal path between the sensor and FPGA must be kept short to maintain close control over timing and signal integrity. The alternative to direct connection is to use off-the-shelf cameras with standard interfaces.

12.1.1.3 Firewire and USB

Many consumer cameras have a USB or Firewire interface. These can be more difficult to interface directly to an FPGA. This is particularly so for USB, which relies on a host processor to manage the low level protocol. Firewire, on the other hand, is a peer-to-peer communication link and does not require a host processor for data transfer.

With both Firewire and USB, direct connection and signalling by the FPGA are impractical and will make inefficient use of resources. The best approach is to use an external chip that manages the physical signalling (a PHY chip). Such chips manage the high speed electrical signalling and convert the serial data into parallel data. The FPGA interfaces to the PHY chip at a speed that is easier to handle. However, this is still complex to interface because of the complexity of the protocols.

This is particularly so for USB, where the FPGA has to act as the host controller for the USB camera. It is also necessary to build considerable logic for managing the communication protocols. This ranges from sending packets to set up the image capture, to receiving and interpreting the video data packets from the camera. A further complexity is that some inexpensive webcams use proprietary protocols and compression – interfacing to these requires reverse engineering the USB protocols, not a trivial task! To connect a USB camera to an FPGA, the simplest approach is to have an embedded processor and interface to it through the appropriate software drivers. If it is necessary to connect directly from the FPGA logic to the USB, it is best to obtain an intellectual property (IP) block that manages all of the USB protocol details.

Interfacing with a Firewire camera is a little easier. Again it is necessary to use the PHY chip. The interface is made easier by a link layer controller (LLC) chip, which manages the communications protocol, CRC checking and provides a FIFO buffer for data. Some manufacturers provide integrated PHY and LLC, which provide a set of control registers through a microprocessor type of interface, and provide direct access to the streamed data through a DMA style of interface These can be connected directly to an FPGA, with a relatively simple controller to manage the application data built within the FPGA logic.

Firewire, as defined by the IEEE 1394 standard, divides all activity on the Firewire bus into 125 µs cycles. Each cycle is initiated by one of the nodes designated as the cycle master, which broadcasts a start packet allowing all other devices to synchronise with the bus. The remainder of the cycle is split amongst two types of data transfer. Isochronous transfers occur first; these are broadcast transfers designed for transferring large blocks of data. Up to 80% of the cycle time can be spent on isochronous transfers, shared amongst all the devices using the bus. The remaining time is for asynchronous transfers, which are point-to-point, with error recovery available.

Within this structure, there are two commonly used video standards or protocols that are used to control the device and transfer of video data. Industrial or machine vision cameras generally use the DCAM specification (1394 Trade Association, 2004), for the transfer of raw video data without audio. Many consumer devices use the AV/C (audio video control) protocol, also specified by the 1394 Trade Association. This defines the behaviour of digital audio-visual devices. The two protocols are quite independent and are not interoperable. The DCAM protocol is described here; the greater complexity of AV/C (because it must handle a wider range of devices) is beyond the scope of this section.

The DCAM protocol specifies a set of enquiry and command registers within the camera. The enquiry registers are queried and command registers set through asynchronous Firewire transfers. These allow the capabilities of the particular camera to be determined, and the camera to be initialised for the desired video mode, frame rate and other camera parameters (gain, white balance, shutter control, iris and focus control, and any other controls that may be offered by the camera). It is also necessary to set the isochronous channel number and bandwidth for transferring the video data and initiate the image capture process. The camera then streams the requested video data isochronously until instructed to stop. Since the asynchronous and isochronous transfers are independent, it is possible to communicate with the camera while it is streaming video data.

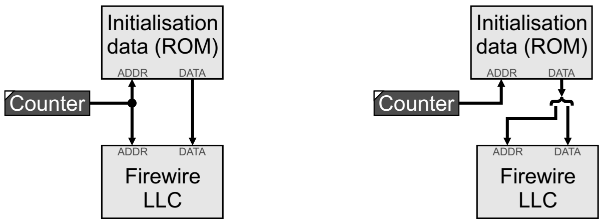


Figure 12.3 A simple controller for setting the link layer controller registers through a microprocessor interface. Left: sequential register addresses; right: register addresses in random order.

Overall, there are two initialisation phases in connecting to a Firewire camera. The first is to set up the Firewire LLC and the second is to set up the camera. With a microprocessor interface, unless an embedded processor is being used, it is necessary to build a small controller on the FPGA that provides the register address and corresponding data to set up the Firewire chipset. Figure 12.3 shows basic structure of a simple controller. The initialisation data is held in a ROM (implemented using fabric RAM). If the registers can be programmed sequentially, the counter steps through the ROM providing the data associated with each address. If the registers must be accessed multiple times, or in a random order, then the alternative is to store both the LLC address and data within the ROM and use the counter to access these sequentially.

A similar type of structure can be used to initialise the camera. The difference is that this data is sent via asynchronous Firewire transfers. The process is easiest if a specific camera with known properties is being used. Otherwise, it is necessary to send a set of queries to the camera to first determine whether or not the camera supports the desired modes. The more complex logic required to manage a range of different cameras and to send the appropriate setup commands is probably best handled through a serial processor rather than within the fabric of the FPGA.

One example of a description of a dual camera system interfaced with an FPGA is described by Akeila and Morris (2008). They connected two Firewire cameras to an FPGA as part of a stereo matching system.

12.1.1.4 Camera Link

While USB, and to a lesser extent Firewire, are targeted primarily at the consumer market, when connecting to a digital frame grabber within an industrial vision system, the Camera Link interface (AIA, 2004) is more common.

Camera Link is based on National Semiconductor's Channel Link technology. A single Channel Link connection provides one-way transmission of 28 data signals and an associated clock over five LVDS pairs. One pair is used for the clock, while the 28 data signals are multiplexed over the remaining four LVDS pairs. This requires a 7 : 1 serialisation of the data presented on the inputs.

The base configuration for Camera Link camera is shown in Figure 12.4. It has (AIA, 2004) one Channel Link connection, with the 28 bits allocated as 24 bits for pixel data (three 8-bit pixels or two 12-bit pixels) and 4 bits containing frame, line and pixel data valid signals. The pixel clock has a maximum of rate of 85 MHz. In addition, there are four LVDS pairs from the frame grabber to the camera for general purpose camera control. The use of these is defined by the particular camera manufacturer. Finally, two LVDS pairs are provided for asynchronous serial communication between the camera and frame grabber. The serial communication is relatively low speed – the specifications say that a minimum rate of 9600 baud must be supported.

For higher bandwidth, the medium configuration adds an additional Channel Link connection for an additional 24 bits of pixel data. The full configuration adds a third Channel Link, giving a total of 64 bits of

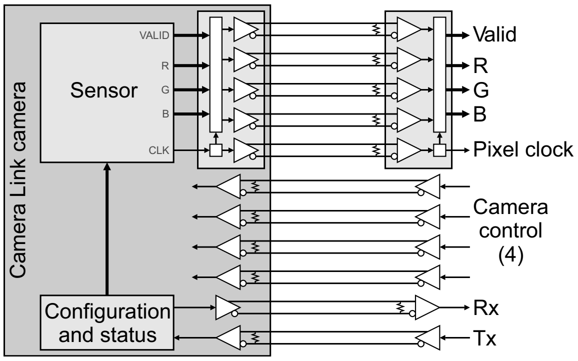


Figure 12.4 Base configuration for a Camera Link camera.

pixel data. This may allow, for example, up to eight pixels to be transmitted in parallel from the camera with each pixel clock cycle.

Modern FPGAs provide direct support for LVDS signalling. For the high speed pixel data, there are two choices. One is to use a Channel Link receiver chip to deserialise the high speed data and provide the pixel data to the FPGA in parallel. Alternatively, since there is no complex protocol involved, the multiplexed signals could be connected directly to the high speed inputs of the FPGA and the built-in SERDES blocks used to perform the 7:1 deserialisation and demultiplexing.

12.1.1.5 GigE Vision

Higher performance devices are tending towards using gigabit Ethernet for data communication. High speed or larger area image sensors are no exception. GigE Vision is a camera interface standard developed by the Automated Imaging Association that uses gigabit Ethernet for the transport of video data and sending control information to the camera. Because it builds on the back of the gigabit Ethernet standard, it is able to make use of low cost standard cables and connectors. These allow data transfer at rates of 100 Mpixels per second over distances up to 100 metres.

The GigE Vision standard has four elements to it. A control protocol defines the communication required to control or configure the camera. A stream protocol defines how the image data is transferred from the camera to the host system. Both of these protocols run over UDP. A device discovery mechanism provides a way of identifying which GigE Vision devices are connected and obtaining their Internet addresses. Finally, an XML description file provides an online data sheet that specifies the camera controls and image formats available from the device. The structure of this XML data sheet is defined by the GenICam standard (EMVA, 2009).

The design of a GigE Vision based system is an advanced topic and a detailed discussion of all of the issues is beyond the scope of this book. Only a brief overview is provided in the following paragraphs.

The complexities of the Ethernet protocol make a pure FPGA implementation difficult. To connect a GigE Vision camera to an FPGA would use a significant fraction of the FPGA resources just to receive the data. A reference design is available from Sensor to Image (2009c) for a GigE Vision receiver that streams data in from a camera and displays the resultant video on a VGA output. In most embedded vision applications, it would be more practical to build the FPGA system within the camera and perform any processing before sending the results to a host system using GigE Vision. Sensor to Image also provides reference designs for GigE Vision transmitters for both Xilinx and Altera FPGAs (Sensor to Image, 2009a; 2009b).

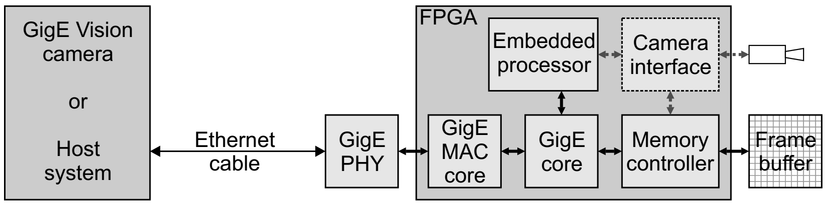


Figure 12.5 Structure of a GigE Vision based system.

The basic structure of a GigE Vision system is shown in Figure 12.5. Unless the FPGA has a built-in gigabit Ethernet PHY this must be provided as an external chip. The MAC core may either be internal or external to the FPGA. The GigE core manages the low level networking features, routing the video stream to or from the frame buffer memory. The control protocol is routed to the embedded processor, which manages the camera configuration in software. When acting as a camera, the system also needs to interface with the sensor. This also provides an interface to the embedded processor for control of the actual camera and to the memory controller for saving the images to the frame buffer. In such a system, the embedded processor also makes available to the GigE Vision side the GenICam-based XML data sheet describing the capabilities of the camera. Not shown in Figure 12.5 is the image processing that adds value to the system!

Real-time operation is achieved on the host system by using a dedicated device driver. This bypasses the standard TCP/IP protocol stack for the video stream, instead transferring the data directly to the application using DMA transfers. This eliminates any CPU overhead from the handling the actual video data as it is transferred.

12.1.2 Deinterlacing

Most analogue cameras use an interlaced scan. The frame is split into two fields, with the even rows scanned in the first field and the odd rows scanned in the second. For a stationary object or camera, this presents no problem, but if either the camera or objects within the scene are moving, then there is a displacement between the two fields.

There are two problems associated with interlacing. The first is the interlace artefacts associated with object motion relative to the camera. The second is associated with scan-rate conversion, for example when displaying video from a 25 frame per second PAL camera on a 60 Hz non-interlaced VGA display.

Consider first the conversion from an interlaced scan to progressive scan, with the output frame rate being double that of the input (so each field is converted to a full frame). Each input field only has half of the lines available. Let the nth field be $I_n(x,y)$, where the missing lines are $I_n(x,y)=0$ for $y \bmod 2 \neq n \bmod 2$. The problem is then to estimate the data within the missing lines. This is an interpolation problem, with solutions involving both spatial and temporal interpolation.

There is a wide range of methods that may be applied to this problem (de Haan and Bellers, 1998), with the more complex methods detecting the motion of objects within the scene and using motion compensated interpolation. One of the simplest methods that gives reasonable results on many images is a simple three-point median filter:

$$Q_n[x,y] = \begin{cases} I_n[x,y], & y \bmod 2 = n \bmod 2 \\ \operatorname{median}(I_n[x,y-1], I_{n-1}[x,y], I_n[x,y+1]), & y \bmod 2 \neq n \bmod 2 \end{cases} \tag{12.1}$$

For static scenes, this is effectively just providing a median filter on the missing rows. This will have little effect on edges, but may cause a small reduction in fine detail. Where there is motion, the previous field may be incompatible with the current field as a result of the motion of edges. In this case, the pixel from the previous field will be quite different from the adjacent pixels in the current field. The median will therefore select either the pixel above or the pixel below to replace the incompatible pixel.

When performing frame rate conversion as well, then the best approach is to use full temporal interpolation. This detects the motion of objects and renders the object in an appropriate position according to the current frame time and object motion. If, however, a small amount of inter-frame judder can be tolerated, then the three-point median approach of Equation 12.1 can also be applied. To avoid tearing artefacts, it is necessary to work with complete fields. Equation 12.1 is simply applied to the two most recent complete fields at the start of the output frame. Incoming fields are simply buffered while the output is being produced. For example, to convert from 50 interlaced fields per second to 60 progressive frames per second, it is necessary to buffer four fields: the two that are currently being interlaced for the display and two to allow for an incoming field to be completed while the output frame is still being produced.

12.1.3 Global and Rolling Shutter Correction

The exposure time within a CMOS sensor is from the time that the pixel reset is released until the pixel value is read out. With global shuttering, all pixels are released simultaneously. The consequence is that pixels read out earlier will have a shorter exposure than those read out later. For short exposure times, there can be a significant difference in exposure between the first and last row read out, resulting in the top of the image being under-exposed and the bottom of the image being over-exposed.

This problem may be overcome in two ways. One is to use an external mechanical shutter to expose the image for a constant time. The alternative is to use an electronic rolling shutter. The reset for each row is successively released so that the time between release and readout is constant for each row. While this solves one problem, it introduces another.

If any objects within the scene are moving (or if the camera is moving or even panning) then since the different rows are exposed at different times, the output image will appear distorted. The nature of this distortion will depend on the relative motion between objects in the scene. For a stationary camera, any stationary background is unaffected. Any downwards motion will result in objects appearing stretched vertically, while upwards motion will result in vertical compression. Sideways motion will result in a shearing of the object. More complex motions, for example resulting from the panning of the camera, can result in additional rotation of objects within the image as a result of the change in perspective. Additional effects can be observed with light variations, for example when the scene is illuminated by fluorescent lights (Nicklin *et al.*, 2007) where the periodic variation of light intensity appears as a distinct banding within the image.

Compensation for the rolling shutter requires modelling both the image capture process and the relative motion between objects and the camera (Geyer *et al.*, 2005). A global correction is only applicable if the scene is stationary and the camera motion can be modelled (Nicklin *et al.*, 2007; Liang *et al.*, 2008). Otherwise, the motion of individual objects must be determined with the position of each object in each frame corrected accordingly. This is still an area of active research, with detailed algorithms not available at present apart for simple cases such as global motion.

12.1.4 Bayer Pattern Processing

Most single-chip cameras obtain a colour image by filtering the light falling on each pixel using a colour filter array – essentially a mosaic of filters. The most common pattern is the Bayer pattern (Bayer, 1976), as shown in Figure 12.6. Each pixel receives light only of a single colour, with half of the pixels green, one quarter red and one quarter blue. Therefore, to form a full colour image, it is necessary to interpolate the

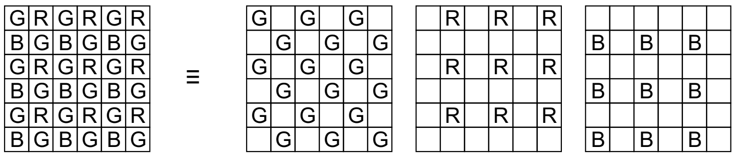


Figure 12.6 Bayer pattern. Left: raw image captured; right: component colour images.

missing values in each of the component images so that there is an RGB value for each pixel. This interpolation process is sometimes called *demosaicing*, because it is removing the mosaic of the colour filter array. The interpolation filters are spatially dependent because the particular filter function depends on the position within the mosaic.

The simplest form of filtering is nearest neighbour interpolation. For the red and blue components, this is accomplished by duplicating the available component within each 2×2 block. Within the green component, the missing pixels may be obtained either from above or to the left. While nearest neighbour interpolation is simple, it does not really improve the resolution, with the image appearing blocky, particularly along edges. This requires a 2×2 window, with multiplexers to route the buffered pixels to the appropriate colour channels, as shown on the left in Figure 12.7.

An improvement may be gained by using linear interpolation. The missing red and blue components with two horizontal or vertical neighbours are obtained by averaging the adjacent pixels vertically. Those in the middle of a block are obtained by averaging the four neighbouring values. This requires a 3×3 window, as shown in Figure 12.7. The vertical average may be reused to reduce the computation required. The divide by two from the averaging is free in hardware.

Using linear interpolation can result in blurring of edges and fine detail. Since a different filter is applied to each colour channel, this can result in colour fringes around sharp edges and fine lines. To reduce the blurring and consequent colour artefacts, it is necessary to detect the orientation of any features within the image, and perform any interpolation in the direction along the edges rather than perpendicular to them. This is complicated by the fact that only one colour component is available for each pixel.

To obtain an accurate estimate of the position of the edge, a much larger window and significantly more computation is required. There are many more complex algorithms described in the literature; a few examples are mentioned here. The ratio between the colour components at each pixel can be iteratively adjusted to make them consistent on each side of an edge (Kimmel, 1999). While this method handles edges reasonably well, the underlying assumption is that there are several consistent samples on each side of an edge. This is not the case with textured regions and fine detail results in colour fringing effects. This approach of enforcing consistency between colour channels is taken further by Gunturk *et al.*, (2002), where the image is first decomposed into low and high frequency bands using a form of

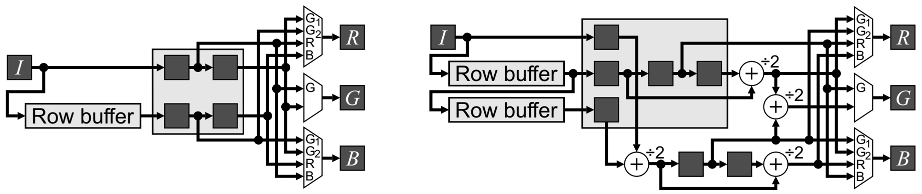


Figure 12.7 Basic Bayer pattern processing. Left: nearest neighbour interpolation; right: linear interpolation. The letters within the multiplexers refer to the current output position, where G_1 is the green pixel on the red row and G_2 is the green pixel on the blue row.

wavelet transform before applying consistency constraints within each band. An alternative to interpolation is extrapolation. The missing value is estimated by extrapolating in each of the four directions and using a classifier to select the best result (Randhawa and Li, 2005). A limitation of this approach is that it requires a very large window, making it less practical for FPGA implementation.

The final result will inevitably be a compromise between interpolation quality and the computation and memory resources required. One relatively simple algorithm that provides significantly better results than simple interpolation with only modest increases in computation is that by Hsia (2004). An edge directed weighting is used for the green channel, with simple interpolation used for the red and blue channels. However, the results are then enhanced using a local contrast-based gain to reduce blurring across edges.

Firstly, the edge directed weighting requires the horizontal and vertical edge strengths to be estimated:

$$\Delta H = |G[x-1,y] - G[x+1,y]| \tag{12.2}$$

$$\Delta V = |G[x,y-1] - G[x,y+1]| \tag{12.3}$$

Then, these are used to weight the horizontal and vertical averages:

$$\hat{G}[x,y] = \frac{\Delta V}{\Delta V + \Delta H}\frac{(G[x-1,y] + G[x+1,y])}{2} + \frac{\Delta H}{\Delta V + \Delta H}\frac{(G[x,y-1] + G[x,y+1])}{2} \tag{12.4}$$

If the denominator is zero, then the two weights are made equal at 0.5.

The interpolated output values are scaled by the local contrast based gain, which is calculated horizontally to reduce the number of row buffers required (Hsia, 2004). The gain term, K, is:

$$K[x,y] = \frac{4I[x,y]}{I[x-2,y] + Q[x-1,y] + I[x,y] + I[x+2,y]} \tag{12.5}$$

where $Q[x-1,y]$ is the previously calculated output for the colour channel corresponding to the input pixels. The final value is the product of Equations 12.4 and 12.5. A similar gain term is used for the interpolated red and blue channels.

A direct implementation of this scheme is shown in Figure 12.8. The outputs are fed back through a multiplexer to select $Q[x-1,y]$ for the channel that determines the gain term. The effective window size is 5×3, requiring only two row buffers as with the simple bilinear interpolation.

One limitation of this circuit from a real-time implementation is the feedback term used to calculate the local gain. The propagation delay through the division, multiplication and two multiplexers cannot be pipelined, and will therefore limit the maximum pixel clock rate. Potential alternatives for K are:

$$K_3[x,y] = \frac{3I[x,y]}{I[x-2,y] + I[x,y] + I[x+2,y]} \tag{12.6}$$

or

$$K_5[x,y] = \frac{5I[x,y]}{I[x-2,y] + I[x,y-2] + I[x,y] + I[x+2,y] + I[x,y+2]} \tag{12.7}$$

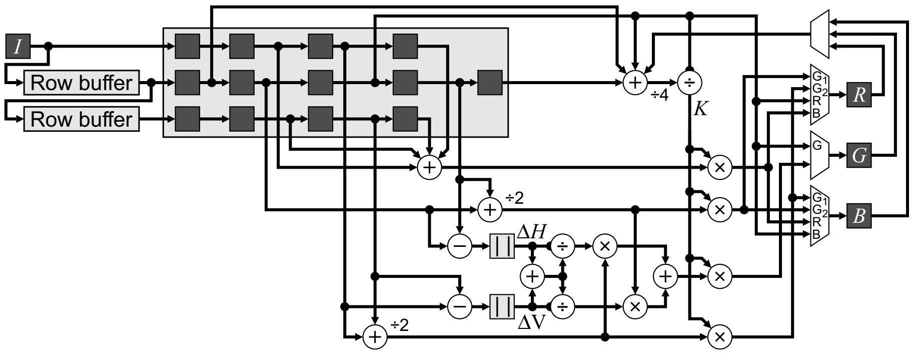


Figure 12.8 Direct implementation of the demosaicing scheme of Equations 12.4 and 12.5.

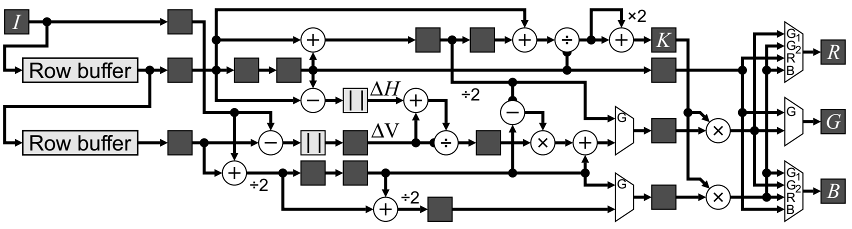


Figure 12.9 Simplified logic for colour filter array demosaicing.

although this latter gain term will require an additional two row buffers. The multiplication by three or five may be performed by a single addition either before or after the division.

This circuit of Figure 12.8 may be simplified by reusing parts of the filter and factorising Equation 12.4 to give Figure 12.9. The local gain term uses Equation 12.6 to remove the feedback and the number of multiplications by K was reduced from four to two by observing that only two of the results will be used for any particular output. An additional stage of pipelining has been added to perform the division and multiplications in separate clock cycles.

12.2 Display Output

Just as image capture is important for an embedded image processing system, most imaging systems also need to be able to display images, even if just for debugging. There are two components to displaying an image, or indeed any other graphics, on an output from an FPGA. The first is the display driver that that controls the timing and generates the synchronisation signals for the display device. The second is generating the content, that is controlling what gets displayed where and when.

12.2.1 Display Driver

The basic timing for video signals was introduced in Figure 1.3. The image is sent to a display in a raster scanned format. At the end of each line of active video data there is a blanking period (Figure 12.10). During this blanking period, a synchronisation pulse is used to indicate to the monitor to begin the next line. For CRT monitors, the blanking period is typically 20% of the scan time for each row. This allows

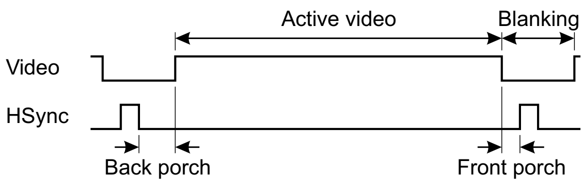


Figure 12.10 Horizontal video timing.

time for the scanning electron beam to be repositioned for the next line. A similar timing structure is used at the end of each frame to synchronise the monitor for displaying the next frame.

The Video Electronics Standards Association has defined a set of formulae which define the detailed timing for a given resolution and refresh rate known as the coordinated video timing (CVT) standard (VESA, 2003). (A number of legacy industry standard timings are also listed in (VESA, 2007).) Within the CVT standard, there are two sets of timings: one for traditional CRT type monitors and one for more modern flat panel displays that do not require the long blanking period. The latter timings significantly reduce the horizontal blanking period, increasing the available bandwidth for active video and significantly reducing the pixel clock frequency. The polarity of the horizontal and vertical sync pulses indicate whether the CRT timing, the reduced blanking interval timing or a legacy timing mode is used. The CVT standard also indicates to the monitor the target aspect ratio by the duration of the vertical sync pulse.

A number of common formats are listed in Table 12.1, along with the timing parameters for a refresh rate of 60 Hz. Parameters for other refresh rates may be obtained from standard tables or from a spreadsheet associated with the CVT standard.

Table 12.1 Timing data for several common screen resolutions for a refresh rate of 60 Hz (VESA, 2003; 2007)

Pixel clock	Visible width	Front porch	Horiz. sync	Back porch	Whole line	Visible height	Front porch	Vert. sync	Back porch	Whole frame	Sync polarity	
MHz	Pixels					Lines					Horiz.	Vert.
25.175	**640**	16	96	48	800	**480**	10	2	33	525	Neg	Neg
40.00	**800**	40	128	88	1056	**600**	1	4	23	628	Pos	Pos
65.00	**1024**	24	136	160	1344	**768**	3	6	29	806	Neg	Neg
81.75	**1152**	64	120	184	1520	**864**	3	4	26	897	Neg	Pos
108.00	**1280**	96	112	312	1800	**960**	1	3	36	1000	Pos	Pos
108.00	**1280**	38	112	248	1688	**1024**	1	3	38	1066	Pos	Pos
85.50	**1360**	64	112	256	1792	**768**	3	6	18	795	Pos	Pos
121.75	**1400**	88	144	232	1864	**1050**	3	4	32	1089	Neg	Pos
101.00	**1400**	48	32	80	1560	**1050**	3	4	23	1080	Pos	Neg
106.50	**1440**	80	152	232	1904	**900**	3	6	25	934	Neg	Pos
162.00	**1600**	64	192	304	2160	**1200**	1	3	46	1250	Pos	Pos
119.00	**1680**	48	32	80	1840	**1050**	3	6	21	1080	Pos	Neg
138.50	**1920**	48	32	80	2080	**1080**	3	5	23	1111	Pos	Neg
154.00	**1920**	48	32	80	2080	**1200**	3	6	26	1235	Pos	Neg
184.75	**1920**	48	32	80	2080	**1440**	3	4	34	1481	Pos	Neg
156.75	**2048**	48	32	80	2208	**1152**	3	5	25	1185	Pos	Neg
241.50	**2560**	48	32	80	2720	**1440**	3	5	33	1481	Pos	Neg
268.50	**2560**	48	32	80	2720	**1600**	3	6	37	1646	Pos	Neg

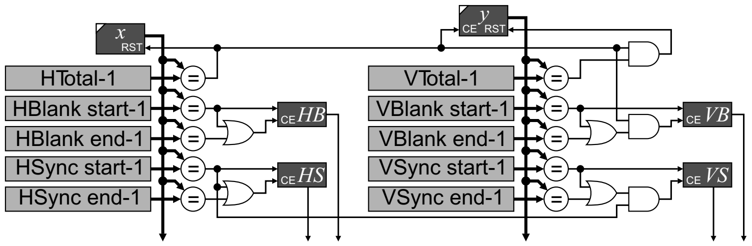


Figure 12.11 Circuit for producing the timing signals for a video display.

The timing can easily be produced by two counters, one for the horizontal timing along each row and one for the vertical timing, as shown in Figure 12.11. The count is simply compared with the times that the key events are required to generate the appropriate timing signals. Rather than use less than and greater than to compare when the counter is in a particular range, tests for equality reduce the logic. This requires an additional single bit register which is set at the start of the signal and cleared at the end. The vertical counter is enabled only when the horizontal counter is reset at the end of each line. The vertical sync pulse is also synchronised with the leading edge of the horizontal sync pulse (VESA, 2007). It is usual for the counters to begin at zero at the start of the addressable active region. In this case, the end of the blanking will correspond to the total, so one of the comparisons may be eliminated. However, if any stream processing pipelines require priming it may be convenient to advance the origin to simplify the programming logic.

12.2.1.1 VGA Output

The VGA output connector provides the display with analogue signal video signals for each of the red, green, and blue components. These require a high speed D/A converter for each channel. For simple displays, with only one or two bits per channel, these signals may be provided reasonably simply using a resistor divider chain. However, for image display the simplest approach is to use a video digital to analogue chip, which is available from several manufacturers.

In addition to the pixel data, separate horizontal and vertical sync signals also need to be sent. These need to be of appropriate polarity, according to the resolution and display mode used. CVT standard timings have positive sync pulses on the horizontal sync and negative pulses (normally high) on the vertical sync. Reduced blanking timings reverse these. Legacy modes may have both horizontal and vertical sync pulses high (or low). The sync pulses may be driven directly by the FPGA using 3.3 V TTL level signals.

The VGA connector also provides an I^2C channel for communication between the host and the display. This is normally used for plug and play functionality, to enable the host to determine which resolutions and refresh rates are supported and for setting various display parameters. Generally it is not necessary to use this channel for driving the display from the FPGA provided that the particular resolution and refresh rate are supported.

12.2.1.2 DVI Output

One limitation of the VGA signal is that it is analogue, and therefore subject to noise. This becomes more acute at higher resolutions, which requires a higher pixel clock. The digital visual interface (DVI) standard (DDWG, 1999) overcomes this limitation by transmitting the signal digitally.

The red, green and blue pixel values are transmitted serialised using a transition minimised digital signalling (TDMS) scheme. This converts each eight bits of input data into a 10 bit sequence using a code that minimises the number of transitions and also averages out the DC level over the long term. The encoded signals are transmitted using a single ended differential pair. As a result of serialisation, the bit clock is 10 times higher than the pixel clock. Since there can be a wide range of clock frequencies (depending on the resolution and refresh rate), a pixel clock is also transmitted on a separate channel.

Four additional 10 bit code words are reserved for synchronisation. These are transmitted during the blanking period with the particular code word dependent on a pair of control signals. These control signals are normally zeros, however, on the blue channel, the control signal is made up of the horizontal and vertical sync signals.

The DVI connector comes in a number of configurations that differ in which signals are connected. The single-link digital connector contains the three data signals described above plus an I^2C-based data channel for communication as described within Section 12.2.1.1. The maximum pixel clock frequency is 165 MHz. This is where the reduced blanking timings come in useful – they increase the useful bandwidth enabling a lower clock frequency to be used. Where the pixel clock exceeds the maximum, additional pins are used to provide a dual link, with two pixels transmitted in parallel each clock cycle. For backwards compatibility, the analogue video signals can also be provided, enabling a simple adaptor to be used with both digital and analogue monitors.

While the 8/10 bit encoding could potentially be performed on the FPGA (for example using a lookup table), a SERDES block would be required by each channel to meet the required data rates. DVI transmitter chips are also available which remove this logic from the FPGA with just a parallel data output for each channel.

12.2.1.3 Television Output

An analogue television signal combines the red, green and blue signals together, producing luminance and chrominance components, with the chrominance signals used to modulate a subcarrier. The processing for this is best performed using an external composite video signal codec.

For digital television connections, the HDMI standard interface using RGB mode is compatible with DVI and just requires a suitable adaptor.

12.2.2 Display Content

The second part of displaying any output is to produce the content that is required at the appropriate time. In particular, this requires producing each output pixel value at the correct time for the corresponding object or image to appear on the display.

The conventional approach used in software-based systems is to represent the content that is to be displayed as an array of data within memory (a frame buffer). This is then streamed to the display as required. The limitation of this approach is that the data to be displayed is calculated and written to a memory. Then, the contents of that memory is periodically read and streamed out to the display. To reduce flicker effects when updating the contents, a double buffering scheme is often used. A copy is made of the display image, which is then updated independently of what is being displayed, and then rapidly written back to the frame buffer to change the display smoothly. The cost of such an approach is that a significant fraction of the time may be spent reading from and writing to memory. Arbitration logic is also required to avoid access conflicts with the process that is reading from the frame buffer to actually display the image.

Using random access processing on an FPGA, like its software counterpart, requires a frame buffer to hold the image being displayed. Generally, double buffering, or bank switching, is also used to give a clean display and meet the bandwidth requirements of both image generation and display driver processes.

However, using stream processing on an FPGA provides an alternative approach. By synchronising the image processing with the display driver, it is possible to perform a much of the processing on the data as it is being displayed. To guarantee that the data is always available when it is needed it is necessary that the processing has a fixed latency. The processing is then started in advance of when it is required. This is made possible since the display driver produces regular, predictable signals, enabling the processing to be begun at precisely the right time. The fixed latency requirement may also be relaxed if necessary by introducing a FIFO buffer to take up any slack. In such cases the blanking periods at the end of each line and each frame may be used to catch up and refill the FIFO buffer.

Even within such a rigid processing regime, considerable flexibility is possible. For example, the display can be segmented, with different images or stages within the process displayed in each section.

12.2.2.1 Windows

This concept may be extended further to display each image within a separate window on the display. Dynamic windows allow the content to be resized and repositioned, giving greater flexibility over fixed image locations (Bailey *et al.*, 2006). Each window requires a separate data structure containing the window properties, including a z-index which controls the order they appear on the display (Buhler, 2007).

The basic principle of on-the-fly display may be extended to the display of window contents if the processing latency is kept small. If two windows are overlapping, it is necessary switch the generation from one process to another as the scanline moves from one window to the next. Since the processing for each window will be sharing the hardware, the timing of such a context switch must take place within the width of the window border. Since the images may be offset vertically relative to one another between the windows, this may require additional caching to retain the context of both windows for the next row.

12.2.2.2 Buttons and Other Widgets

In a windowing environment, it is also necessary to be able to display buttons and other widgets to control the processes. Each widget is contained within a window, and within each window the widgets are not overlapping. This allows a single instance of the display hardware for each type of widget to be multiplexed amongst all instances of that widget. This allows the widget to be produced on-the-fly as it is required for display.

From a hardware perspective, this is best managed by using a memory to hold the data structure for each type of widget (Buhler, 2007). In that way, the multiplexing is performed efficiently through memory addressing.

In addition to displaying widgets, it is also necessary for the user to interact with them – that is what they are there for! In a conventional frame buffer-based system, it is necessary to perform a hierarchical search through the display objects to determine which widget is at the current cursor location. With on-the-fly processing, as the cursor is displayed, the widget at the cursor location is also being indexed and displayed. This facilitates a very simple widget processing regime, because the widget at the location of the cursor can automatically receive any associated user interactions. If the widget control is implemented in hardware, such interactions can take place directly with the widget, again multiplexing a single hardware controller between all of the instances of a particular widget. Alternatively, the widget index and relative cursor position may be passed to a software process for performing more complex interactions.

12.2.2.3 Character Generation

It may be useful to have textual or other annotations associated with a window. This may be useful for labelling objects or regions of interest, or displaying textual debugging information. While text could be displayed in a frame buffer, this is inefficient in terms of memory usage and makes dynamic update more difficult.

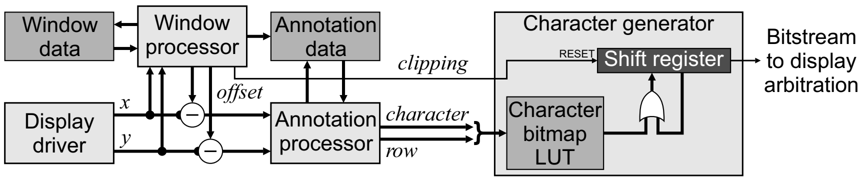


Figure 12.12 Character generation process.

A more efficient approach is to represent the text directly as ASCII codes and interpret these on-the-fly to produce the bitmaps that need to be displayed. Character generation logic is simplified if character locations are restricted to a grid, but more general positioning can be accommodated. The process is outlined in Figure 12.12. The display driver produces the position of the current pixel being displayed. The window processor receives this and, based on which window is currently visible at that location (if any), selects the annotation data associated with the window and then provides the offset based on window position and character generation latency for the annotation processor. The annotation processor then detects the leading edge of each character and outputs the ASCII code and row within the character to the character generator. The line of pixels is read from the character generator bitmap and is ORed into the output shift register, combining it with any other characters that are also being produced. The shift register shifts the bits out sequentially, producing the character glyph on the display. The window processor detects the trailing edge of the window and resets the shift register at the appropriate time to clip the character to within the window borders. Of course, there are many more details that have been skipped within this simplified explanation (Buhler, 2007).

12.2.2.4 Arbitration Issues

With several processes producing output for the display, it is essential to combine these together appropriately. A relatively simple priority stack can be used to determine which process produces the pixel for the display, as illustrated in Figure 12.13. The window processing determines which window and associated components are being displayed at any location. If no window is at the current pixel, then the background is displayed by default. Within a window, the background or image if it is an image window is the next layer. On top of this comes any textual annotation and widgets. At the highest priority is the mouse cursor layer. Any object on a higher priority layer will either mask (or modify) the underlying layers.

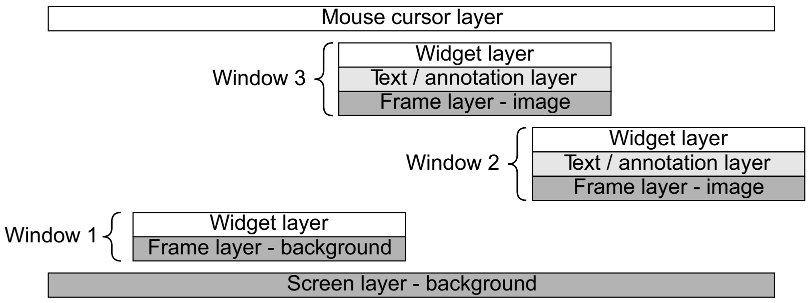


Figure 12.13 Window layering showing the arbitration priority. (Reproduced with permission from D.G. Bailey *et al.*, "GATEOS: A windowing operating system for FPGAs," *IEEE International Workshop on Electronic Design Test and Applications*, 405–409, 2006. © 2006 IEEE.)

12.3 Serial Communication

Many peripheral devices communicate using some form of serial communications. This section briefly describes the most common of these, along with related FPGA implementation issues.

12.3.1 PS2 Interface

The PS2 is a low speed serial connection designed primarily for communication between a computer and keyboard or mouse. Since the PS2 protocol is much easier to work with than the more modern USB standard, it has become the method of choice for connecting a mouse or keyboard directly to an FPGA. Since the underlying PS2 protocol is the same for both the keyboard and mouse (Chapweske, 2003), it is described first. The specific communication is then described for interfacing with a mouse and keyboard.

The PS2 connector provides a 5 Volt supply to power the keyboard or mouse and also has data and clock signals. The data and clock signals use an open collector interface. These can be implemented on the FPGA by using tristate output control to drive the output pin low when needed and use an internal pull-up resistor on the input to pull the signal to its default high state (Figure 12.14). If the pull-up voltage is too low, or the pull-up resistor is too strong, then an external pull-up resistor may be used.

Both the clock and data signals are only driven when transferring data; in the idle state, both lines are at their default high state. The clock signal (typically 10–16 kHz) is always generated by the PS2 device, so if the FPGA wants to send a command to the device it must signal with a request to send (RTS) pattern. This consists of pulling the clock signal low for at least 100 microseconds, then pulling the data signal low and releasing the clock. This instructs the device to begin transmitting a clock enabling the FPGA to transmit data.

Each packet, whether transmitting or receiving, consists of 11 clock cycles from the device (Figure 12.15):

- a start bit, which is always low (the RTS from the FPGA provides the initial start bit for FPGA to device communication);
- eight data bits, sent least significant bit first;
- a parity bit, using odd parity (the number of ones in the data bits plus parity bit is odd);
- a stop bit, which is always high;
- and a final acknowledge bit, sent by the device only when the FPGA is transmitting data to the device.

When the FPGA is transmitting data to the keyboard or mouse, the data line should be changed while the clock signal is low and be held while the clock is high. When receiving data from the device, the opposite occurs: the data changes while the clock is high and can be read on the falling edge of the clock.

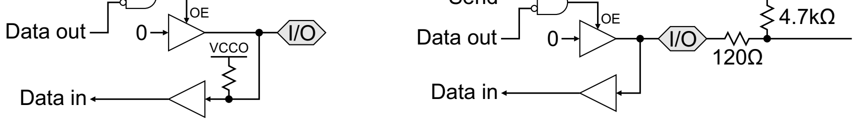


Figure 12.14 Implementation of an open collector or passive pull-up data bus. Left: using an internal pull-up resistor; right: with external pull-up resistor.

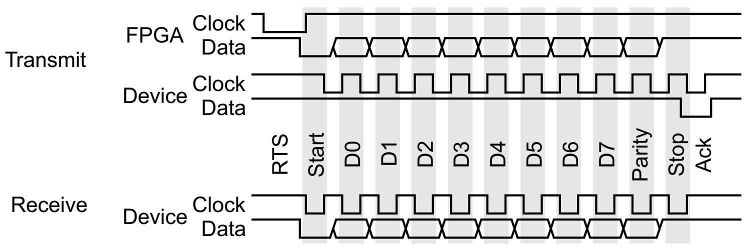


Figure 12.15 PS2 communication sequences. Top: transmission from FPGA to device; bottom: receiving from the device.

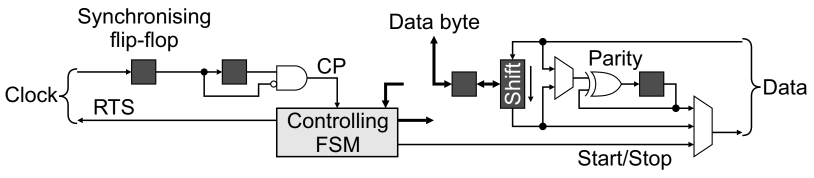


Figure 12.16 Skeleton implementation for a PS2 driver.

On the FPGA, the communication can be managed by a simple state machine; a skeleton implementation is given in Figure 12.16. The main control comes from the clock input from the device. A synchronising flip-flop is required to prevent metastability problems on the input (it is not needed on the data line because it is synchronous with the clock). The AND gate then detects the falling edge and produces a single pulse used to enable the finite state machine and the rest of the circuit through the clock enable pins on appropriate registers. This enables the driver to be operated in the main clock domain that is directly using the data. For transmit, the length of the RTS pulse may be determined by a counter. On the data side, a single shift register can be used to clock data in and out, with a single parity generator/checker switched between the input or the output. The output multiplexer selects the source of the current output bit. The state machine also has control inputs from the input data line (for checking start and stop bits) and from the parity checker.

12.3.1.1 Mouse

To use the mouse, the FPGA must first confirm that a mouse is connected to the PS2 port. This consists of sending a sequence of commands to the mouse and checking the response. A selection of the main commands is listed in Table 12.2. The mouse may be operated in one of several modes, the most useful of which is probably the streaming mode, where the mouse reports any changes in position or button status. If the mouse has a scroll wheel, it may also be enabled by sending the appropriate sequence.

When in streaming mode, whenever the mouse is moved or a button is pressed or released the mouse streams three byte packets, as listed in Table 12.3. If the mouse has been successfully put into scroll wheel mode (device type reports as 0x03), the data packet is four bytes long. The movements are all relative to the previously reported position, with the X and Y movements represented using 9-bit two's complement offsets. If the offset exceeds ± 255 then the corresponding overflow bit is set. The scroll wheel movement is a 4-bit two's complement offset, contained in the fourth byte.

Table 12.2 Main mouse commands and responses

FPGA		Mouse	
Reset	0xFF	0xFA	Acknowledge
		0xAA	Passed self test or 0xFC (error)
		0x00	Device type of standard mouse
Set scroll wheel mode	0xF3	0xFA	
	0xC8	0xFA	
	0xF3	0xFA	
	0x64	0xFA	
	0xF3	0xFA	
	0x50	0xFA	
Read device type	0xF2	0xFA	Acknowledge
		0x03	If scrolling is supported, otherwise 0x00
Enable reporting	0xF4	0xFA	Acknowledge

12.3.1.2 Keyboard

A keyboard is detected by sending a reset command to the PS2 port. After the self test response is received, sending a read device type command (Table 12.2) should result in a two-byte device type of 0xAB, 0x83.

Whenever a key is pressed, the corresponding scan code (listed in Figure 12.17) is transmitted by the keyboard to the host. While the key is held down, the keyboard will periodically resend the scan code. When the key is released, a key-up scan code is sent, which for one-byte codes consists of 0xF0 followed by the scan code, and for two-byte codes, the 0xF0 comes after the 0xE0. The keyboard does not take into account whether or not the Ctrl, Shift or Alt keys are down – it is the responsibility of the host to account for these when interpreting the scan codes. If ASCII input is required, these scan codes may be converted to ASCII using a lookup table, combined with the shift status.

The indicator LEDs on the keyboard may be set by sending the command 0xED followed by a data byte with Scroll Lock in bit 0, Num Lock in bit 1 and Caps Lock in bit 2.

12.3.2 I^{2}C

Many devices and interface chips are configured using an Inter IC (I^{2}C) bus (NXP, 2007) or related communications protocol (for example, SMBus and PMBus used within computer systems). This is a two wire serial bus, with a data line (SDA) and clock line (SCL). Many devices may be connected on the single bus, including multiple bus masters, with each device on the bus individually addressable. The bus supports multiple data rates, from 100 kbit/s up to 3.4 Mbit/s, with the speed limited by the slowest device on the bus. When connecting to peripherals from an FPGA, it is assumed here that there is only a single bus

Table 12.3 Mouse data packet in streaming mode

	MSB	6	5	4	3	2	1	LSB
Byte 1	Y overflow	X overflow	Y sign	X sign	1	Middle button	Right button	Left button
Byte 2	X movement							
Byte 3	Y movement							
Byte 4 (scroll only)					Z movement			

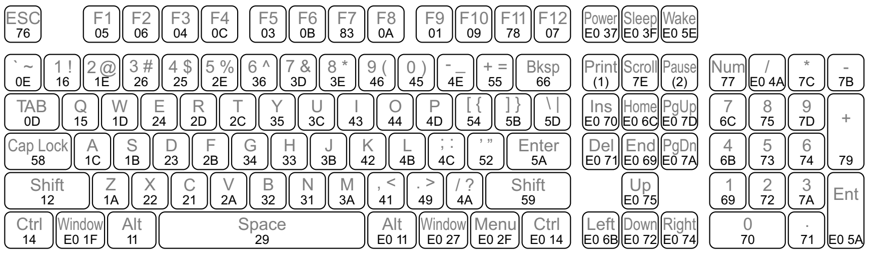


Figure 12.17 Keyboard scan-code numbers in hexadecimal. (1) E0 12 E0 7C; (2) E1 14 77.

master – the FPGA, and that it will be programmed to run at the speed of the slowest peripheral. (To handle the more complex case of multiple bus masters, consult the I^2C specifications; NXP, 2007.)

The bus lines normally use a passive pull-up (like the PS2 port; Figure 12.14), although active pull-up may be used when operating with a single master and none of the devices will stretch the clock. The clock signal is always generated by the master, which also initiates and terminates any data transfer. Normally, the data signal is only allowed to change while the clock signal is low. The exception to this is indicating a start condition (S) or a stop condition (P). Each byte transferred requires nine clock cycles: eight to transfer data bits (MSB first) and one to acknowledge.

A transfer is initiated by the master (the FPGA) signalling a start condition: pulling SDA low while the clock is high. The first byte in any data transfer is always sent by the bus master and consists of the 7-bit address of the slave device and a $R/\overline{W}$ bit. This is then acknowledged by the addressed slave pulling SDA low for the ninth clock cycle. The remainder of the transfer depends on whether the master signalled a read or a write (Figure 12.18). When the FPGA is writing, each byte transferred is acknowledged by the slave, unless there is some error condition (for example it was not able to understand a command that was sent). At the end of the transfer, the master signals a stop condition (a rising edge on SDA after the clock has gone high) or another start condition if it wishes to make another transfer. When reading from the device, after the first address byte the slave begins transmitting data which is acknowledged by the FPGA. After the last byte, the FPGA sends a negative acknowledgement followed by the stop or repeated start condition.

There are four conditions that can lead to a negative acknowledgement (NXP, 2007), indicating that the transfer should be aborted:

- There was no slave on the bus with the specified address. Since there will be no response, the default will be for the bus to stay high.
- The slave device is not ready to communicate or cannot receive any more data from the master.

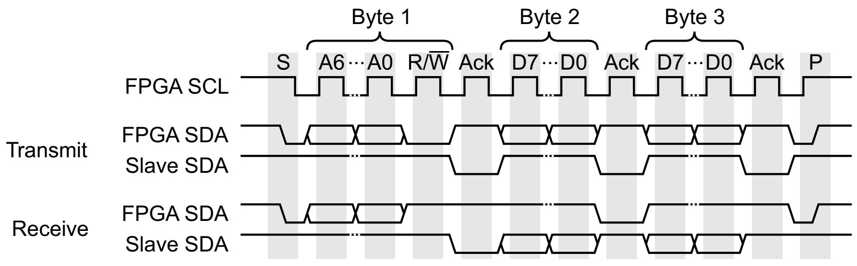


Figure 12.18 I^2C communication sequence. Top: writing to the device; bottom: reading.

- The data sent to the slave was invalid, out of range or otherwise not understood by the slave.
- When the master is receiving data from a slave, to indicate that no more data should be transmitted.

Many I^2C devices are register based, in that data is written to, or read from, a numbered register on the device. A write is straightforward – the FPGA sends the write to the slave, followed by the register address, then the data written to that register. Reading from a register is a little more involved. Firstly the FPGA needs to send a write to the slave and then write the register address. Then the FPGA sends a repeated start condition to terminate the first transaction and begin a new one. This consists of sending a read to the slave, and then the slave returns the contents of the register addressed in the first message.

An FPGA implementation of I^2C is made easier by the FPGA being the bus master and providing the clock signal. The clock signal can be produced simply by dividing down the clock within the main clock domain. A finite state machine may be used to control the driver, using a circuit similar to Figure 12.16 (obviously the parity generator and checker are not required). The control is a little more complex as a result of slave addressing. However, the driver can be kept reasonably simple by separating the driver from the data interpretation.

12.3.3 SPI

Another serial bus architecture is the serial peripheral interface (SPI). It is sometimes used with sensors and is commonly used as an interface to serial memories. It is both simpler and faster than I^2C, although it has no formal protocols and there are several variations. The basic structure of the interface is as shown in Figure 12.19. The master selects the slave device and then data is communicated in synchronisation with the clock. To access multiple slaves, each slave requires its own separate slave select ($\overline{SS}$) pin, although the clock and data lines can be shared. The data transfer is full duplex, with MoSi being the master output/ slave input and SoMi the slave output/master input, although both lines do not always carry meaningful data simultaneously. The meaning of the data transferred, and even the size of the transfer, is dependent on the size of the device. Therefore, it is necessary to consult with the manufacturer's documentation on the specific commands and protocols used in order to design an appropriate driver for the peripheral.

Within an FPGA-based system, an SPI interface can be used to provide the configuration data for the FPGA. The application may require access to the SPI bus if the application has the ability to reprogram the configuration memory. It may also provide other non-volatile data storage as required by the particular application.

12.3.4 RS-232

Another common serial interface is RS-232. While in many applications it has been superseded by USB, it still remains popular because of its simplicity and is a standard peripheral in many microcontroller chips. While technically RS-232 refers to the particular electrical signalling standard, it is often uses synonymously for asynchronous serial communication, and this is primarily the sense used here.

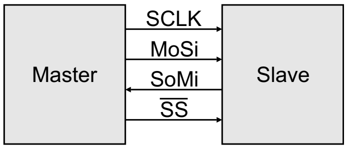


Figure 12.19 Serial peripheral interface.

To convert from normal logic levels to RS-232 levels, an appropriate interface chip is used, for example the MAX232 from Maxim Integrated Products. The need for a common ground limits the length of an RS-232 cable. In an industrial setting, the differential signalling of RS-422 (or RS-485 for multidrop) are more robust.

Asynchronous communication is so-named because it does not use a clock signal. Before transmission, the data line is held in an idle state, (a logical 1). Transmission is started by sending a start bit (logic 0); the falling edge is used to synchronise the receiver to the transmitter. Following the start bit, five to eight data bits are sent, followed by an optional parity bit, and finally one to two stop bits (logic 1) which are used to synchronise with the next word transmitted. The following timing and data framing parameters must be agreed between the transmitter and receiver before transmission:

- the clock frequency;
- the number of data bits and the order in which they are sent (MSB or LSB first);
- whether to use a parity bit and, if so, whether odd or even parity is used;
- the number of stop bits (the minimum time before a new transmission may begin).

A half-duplex connection uses a single data line for both transmitting and receiving. This requires hardware handshaking to control the direction of the data transfer. The handshaking is asymmetric, using separate signal lines. One end is designated as a data terminal and it signals its desire to transmit by setting the request to send (RTS) signal low. This is acknowledged by a reply on the clear to send (CTS) line. Full-duplex uses separate independent data lines for transmitting and receiving.

The implementation of asynchronous communication with a predefined set of parameters is a straight forward extension of the circuit of Figure 12.16. However, being able to change the parameters at run-time makes the system significantly more complex.

12.3.5 USB

As mentioned earlier, interfacing to devices using USB is quite complex. For simple serial communication, the simplest approach is to connect to an integrated USB chip that enables asynchronous serial data to be streamed directly to it. Many such devices have a built-in microcontroller which manages the details of the USB protocols. The alternative is to use a soft-core processor on the FPGA to manage the communication and protocols. It is generally not worth the effort interfacing hardware directly to a USB connection.

12.3.6 Ethernet

The use of high speed Ethernet was introduced earlier in this chapter with the GigE Vision camera interface protocol. To recap, the communication interface requires a PHY for managing the physical signalling and a MAC core for encapsulating the data with the required Ethernet headers and controlling data transmission and reception. The use of Ethernet for point-to-point communication is relatively straightforward, although writing a driver for interfacing between the rest of the design and the Ethernet is not for the faint-hearted. Fortunately, logic cores are available to do most of the hard work.

Using an Ethernet connection to connect to the Internet adds a whole new layer of complexity. Communication on the Internet is based on the TCP/IP protocol. This is, in fact, a suite or stack of protocols (Roberts, 1996), where the Internet protocol (IP) encapsulates data transmitted with a header that describes the particular protocol, which is then transmitted (as payload) over Ethernet. Several of the core protocols of the Internet protocol suite are:

- The transmission control protocol (TCP) is one of the most commonly used IP protocols for the transmission of data on the Internet. It establishes and maintains a virtual stream connection between the

two hosts, on which reliable bidirectional transfer of data may take place. The protocol has mechanisms for detecting dropped or corrupted packets and automatically retransmitting the data until it is received correctly.
- The user datagram protocol (UDP) provides a lightweight mechanism for the transmission of data packets from one host to another. While there is a checksum for checking data integrity, there is no mechanism for ensuring reliable delivery. UDP is used for time sensitive applications where the overhead of retransmitting lost or missing data is unacceptable, for example streamed media.
- The Internet control message protocol (ICMP) works behind the scenes to communicate status and error messages between hosts and routers in an Internet-based network.
- The address resolution protocol (ARP) maps an IP address to an Ethernet address to enable the packet to be sent to the correct destination on an Ethernet-based network.

On top of UDP and TCP are the application layer protocols, which specify how the actual data content is transferred for a particular application, such as file transfer, mail, news, domain name resolution, Web (HTTP), and so on.

While all of these component protocols could be implemented in hardware (Dollas *et al.*, 2005), the complexity of the layered architecture can quickly become unwieldy. Consequently, the Internet protocol is probably best managed by a software-based protocol stack, with the Ethernet MAC implemented in hardware using the FPGA logic.

As outlined in Section 4.3.3, one advantage of equipping an embedded system with a full TCP/IP stack is that it provides a generic mechanism for remote communication. A Web interface allows remote control and setting of algorithm parameters. The results, including images, can be readily displayed directly through a Web interface. An Internet connection also allows the FPGA configuration to be remotely updated, providing a ready path for upgrades.

12.3.7 PCI Express

PCI express is the latest standard for connecting to peripherals within a high speed computer system (Wilen *et al.*, 2003). Previous generations of the peripheral component interconnect (PCI) bus were parallel, with 32 or 64 data lines. However, as the bus speed is increased, two problems are encountered: increased cross-talk from the coupling between adjacent signal lines and increased skew between data lines resulting from small differences in path length. In switching to a serial connection, PCI express solves the skew problem (although a much higher clock rate is required to maintain the bandwidth) and uses differential signalling to significantly reduce the effects of cross-talk. The other significant change is that while PCI is a bus, shared by all peripherals, PCI express is a point-to-point connection with separate transmit and receive, avoiding overheads associated with bus arbitration and contention.

Each differential pair operates at 2.525 GBit/s, with a transmit and receive pair called a lane. Clocking is embedded within the signal by using 10 bits to encode each eight bits of data. This limits the length of consecutive zeros or ones to provide edges for a phase locked loop to recover the clock. (The new PCI express version 3 achieves the same effect with reduced overhead by scrambling the data first and using 130 bits to encode each 128 bits of data.) Higher bandwidth connections can run multiple lanes in parallel, with successive bytes sent down successive lanes.

At the link layer, additional overheads are introduced for CRC-based error detection and packet acknowledgements. Each data packet is kept in the transmit buffer until acknowledgement is received. This allows packets signalled as corrupt to be automatically resent.

Higher layers of the protocol create virtual links between the components that share the underlying physical link. This is implemented through a transaction layer, where each transaction consists of a request followed by a response. Between the request and response, the link is available for other traffic. Flow control uses a credit-based scheme where the receiver advertises the space available within the receive buffer as credit. The transmitter keeps track of its remaining credit and must stop transmitting before the buffer is full. As the receiver processes data and removes it from the buffer, it restores credit to the transmitting device.

PCI express is most commonly used to interface between an FPGA board and its host computer system, enabling the rapid transfer of data in high performance computing applications. Recent FPGAs have built in PCI express ports, with logic cores available for interfacing between the rest of the logic of the application.

12.4 Memory

In many applications, it is necessary to use memory external to the FPGA for frame buffers and other large memory blocks. External memory is also required when running an embedded processor with an operating system such as Linux. There are two main types of memory: static and dynamic. The issues involved in interfacing with these are described in turn.

A common issue, particularly with high speed memories (in particular DDR memories) is ensuring that any signal skew in the parallel data and address lines does not violate setup and hold times.

12.4.1 Static RAM

Static memory stores each bit using a latch. Each memory bit therefore requires a minimum of four transistors, with an additional two transistors used for accessing the latch for reading and writing. This is considerably larger than the one transistor used by dynamic memory, making it more expensive to manufacture. Its advantage is its speed, making it commonly used for intermediate memory sizes (in the low megabyte range), such as commonly used by cache memories in high end computer systems.

Almost all high speed synchronous memories are pipelined. Therefore, when reading, the data is available several clock cycles after the address is provided, although the memory has a throughput of one memory access per clock cycle. This access latency must be built into the design of the application. This makes external memories a little more difficult to use than internal block RAMs.

Usually when writing, both the address and data must be applied at the same time. Consequently, when following a read by a write, there is a dead time to allow the read to complete before the data pins are available for providing the data to be written. If using such memories in a design, this dead time must be taken into account in the timing budget. While such memories may be suitable for reading or writing large blocks, such as might be encountered when streaming an image to or from a frame buffer, they are unsuitable for two-phase designs, where reads and writes may be alternated.

Zero bus turnaround (ZBT) memories have no dead period between writes and reads. This is accomplished by providing the data to be written after the address, as shown in Figure 12.20. The ability to switch between reads and writes from one cycle to the next makes ZBT memories significantly

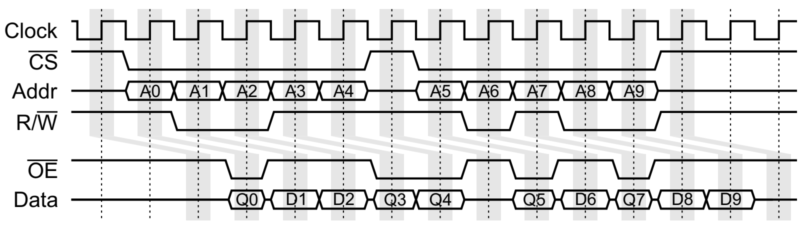


Figure 12.20 Typical timing for a pipelined ZBT static memory with two clock cycles latency.

easier to interface to an application. The data to be written may be delayed within the FPGA if necessary by using a pipeline.

12.4.2 Dynamic RAM

Dynamic memory, because of its lower cost, is best suited for larger volumes of memory. It uses a single transistor per memory cell, with the data stored as charge on a capacitor. Since the charge leaks off the capacitor, it must be refreshed at least once every 64 ms to prevent the data from being lost. To achieve this with large memories, dynamic memories are structured so that a complete row (within the two dimensional memory array) is refreshed at a time. The availability of a whole row at a time leads to a paged memory structure, where a row is selected first, and then the column within that row is read from or written to.

Interface to a chip is controlled through a sequence of commands which are provided through a set of control pins. A typical command sequence is as follows:

- An activate command selects the memory row to use. The address lines specify which row is selected. Activating a row reads the row into the column sense amplifiers, enabling the data within that row to be accessed. It also has the side effect of refreshing the charge on the capacitors within that row. Row activation typically takes several clock cycles.
- Once activated, read and write commands may be made to that row. With a read or write command, the address lines specify the column that is to be read or written. A read or write has several clock cycles latency, however these are pipelined, enabling one access per clock cycle to data on that row. When writing, the data must be provided with the address, so there is a bus turnaround delay when switching from reads to writes. Operating in burst mode enables several successive memory locations to be read or written with a single command. This is exploited with DDR memories, giving two reads or writes per clock cycle, one on each edge of the clock.
- Before selecting another memory row, it is necessary to close the row with a precharge command. This returns the sense amplifiers to an idle state ready to sense the next row.
- A refresh command refreshes one row of memory. An internal counter is maintained so that the rows are refreshed in sequence. The refresh command internally activates a row, waits for sufficient time for the charge on the capacitors to be refreshed, and then returns the line to the idle state. While it is possible to do this manually, having a refresh command simplifies the memory controller.

There is considerable latency, especially when moving from one row to another. To help alleviate this, most DRAM chips have multiple banks of memory internally, so that one bank may be used while waiting for row activation or precharging within another bank.

Reading or writing from a dynamic memory is, therefore, not a simple matter of providing the address and data. The complex sequence of operations to access a particular memory and ensure that the memory is kept refreshed requires a memory controller. Some new FPGAs incorporate dedicated hardware memory controllers to simplify access (Xilinx, 2010c). Otherwise it is necessary to build the memory controller from the fabric of the FPGA.

If using dynamic memory for a frame buffer, it works best with streaming mode because there will be many sequential accesses to the same row. Streamed access by columns (for example with the FFT) is generally not practical because of the large latencies to switch from one memory row to another. This may be overcome to some extent by creating a mapping between logical memory and physical memory to increase the number of sequential accesses to the same page. However, such techniques complicate the system design. A similar problem is encountered with random access processing. In such systems, it may be necessary to maintain a static memory as a frame buffer in addition to the dynamic memory. If the access pattern is known in advance, then a smaller cache may be built on the FPGA.

DIMMs (dual in line memory modules) consist of a number of dynamic RAM chips mounted on a printed circuit board to give a wider data width. These are commonly used with conventional computer systems, but can also be used with FPGAs.

12.4.3 Flash Memory

Flash memory is non-volatile, making it useful for storing data required by an application while the FPGA is switched off, or when switching from one configuration file to another. The basic mechanism is to have charge stored on a floating gate transistor, which results in a change in transistor threshold voltage, enabling the different states to be identified (Pavan *et al.*, 1997). Programming consists of injecting charge onto the floating gate, while erasing removes the charge. There are two main types of flash memory based on how these transistors are arranged: NOR flash and NAND flash.

NOR flash is similar to regular memory, with address and data lines allowing individual transistors to be read or written using a random access memory pattern. Read access times are reasonably fast, although writes are very slow in comparison with static or dynamic memories, and this must be taken into account by the application. Erasure is on a block by block basis, and is also very slow. Erasing a block sets all the bits to ones, and writing can only change ones to zeros. To change a zero to a one requires erasing the complete block, losing any data contained within it.

NAND flash arranges the transistors differently to enable a higher packing density (resulting in lower cost). The consequence is that NAND flash has page or sector-based structure, making it better suited for sequential access than random access. Programming individual bytes and random access are difficult, but programming and erasing blocks is significantly faster than NOR flash (Micron, 2010). Internally, reading copies the whole block from non-volatile storage into buffer register (typically 25 μs) from which the data can then be streamed out (typically 25 ns per byte or word). Similarly, writing streams the data into the buffer register (25 ns per transfer) from which the whole block is programmed at once (200–500 μs). Since NAND flash memories are orientated to sequential access, the data pins are usually multiplexed to carry commands and the page address.

NAND flash requires error checking and correction to ensure data integrity. Each block usually has a number of spare bytes that can be used for storing error correction codes. However, it is up to the controller to implement any error checking and correction. It is also important to check the status after a program or erase operation to confirm that the operation has completed successfully. If not successful, such blocks should be marked as bad and not be used. Micron provides VHDL code that may be used for the basis of a NAND flash controller (Micron, 2007).

12.5 Summary

This chapter has reviewed some basic interfacing issues particularly related to embedded vision applications.

A camera or input sensor is perhaps the key peripheral in a vision system. The most straightforward approach is to interface directly with the sensor chip. This gives the lowest latency and avoids many of the complications of interfacing with the USB or Firewire connection of consumer cameras. An issue with CMOS sensors is characterising and correcting the distortions introduced by rolling shutter-based electronic shuttering. This is a complex problem that depends on the motion of both object and camera. When interfacing directly with a colour sensor, it is also necessary for the FPGA to recreate the full colour image by interpolating the values of the missing colour channels.

Image display is another important capability, even if it is just used for debugging and system tuning. A simple display is relatively easy to generate, although an analogue output will require external digital to analogue converters. A DVI output keeps the system all digital, although it will require more logic on the

FPGA and high speed SERDES interfaces to produce the serial signals. Alternatively, a DVI transmitter chip offloads the logic at the expense of increased component count.

Some issues with display content generation have been discussed briefly. In particular, techniques for window and character generation have been outlined.

A range of serial communication protocols have been outlined. The control channel for many devices (including cameras and displays) is usually over a relatively low speed serial link. The slower speed makes the design easier, with simple interfaces able to be built with relatively few resources. Higher speed connections such as gigabit Ethernet and PCI express are significantly more complex and should be built using intellectual property blocks. While both will be less common in embedded vision applications, PCI express is commonly used to interface between a host computer system and an FPGA in a high performance reconfigurable computing platform.

Finally, issues with interfacing to external memories have been considered briefly. The main complication is the memory latency with pipelined access. This is particularly acute with dynamic memory, which requires a page be selected before it is accessed. Careful design of the system, and in particular the memory controller, is required to prevent the acceleration gained elsewhere in the algorithm from being lost due to memory bandwidth limitations.

13

Testing, Tuning and Debugging

In any application, it is the fortunate developer who can get the system to function as desired on its first attempt. Inevitably, there are errors in either the design or implementation that require the algorithm to be tuned or debugged.

There are four main causes for an algorithm not behaving in the intended manner that are addressed briefly in this chapter. These are:

- **Design errors.** This encompasses a range of issues including faulty logic and the algorithm being unsuitable or not sufficiently robust for the task. A rigorous design process should eliminate such errors before reaching the FPGA implementation stage. However, many development applications follow a less formal design-as-you-go process, where such design errors can be expected!
- **Implementation errors.** Syntax errors such as spelling mistakes, incorrect operations and misplaced parentheses will usually (but not always) fail to compile. More subtle errors result from the implementation not matching the design. These include errors in the mapping of operations onto the FPGA, such as not correctly taking into account the latency. Particular problems can relate to conflicts between subsystems working in parallel.
- **Tuning errors.** In this class, the algorithm is basically correct, but the parameters are such that it does not behave in the intended manner. Tuning involves finding the set of parameters that gives the best performance.
- **Timing errors.** These result from the algorithm being logically correct, but being clocked too fast for the propagation delays of the logic. Systems run at the limit of their performance may fail intermittently as setup and hold times of registers are violated.

A range of techniques were discussed from a design perspective in Section 4.5. These will not be examined again in this chapter in any detail, although they will be reviewed as appropriate under each of the categories above.

13.1 Design

As outlined in Chapter 4, the design of an image processing algorithm for a particular task is a heuristic process. It requires considerable experimentation to develop the sequence of image processing operations to achieve the desired result. A software-based image processing system provides the level of user interaction required to try different operations and examine the results.

The main issue with algorithm design is how to go about testing a complex algorithm. In all but trivial cases, it is impossible to test an algorithm exhaustively. No algorithm is perfect; all will have flaws or limitations. The primary role of testing is, therefore, to determine the useable limits of the algorithm and determine its robustness.

The conventional approach to testing digital logic is to develop a test bench which exercises all of the components of the design and validates that the design is correct. With image and video processing, however, this approach is impractical because of the large input space. A single low resolution, 256×256 greyscale (8 bit) image has a total of 524 288 bits. This gives 2^{524288} different images. While only a small fraction of these are 'useful' in any sense of the word, and, of those, even fewer images may be relevant to a particular task, the number of images likely to be encountered in any application is still impractically large to test exhaustively. By today's standards, a 256×256 image is small. Larger images and image sequences (video) contain even more raw data, compounding the problem.

Testing an image processing algorithm therefore only considers a selection of sample images. In addition to a range of typical cases, it is important to use images which test the boundaries of correct operation. When using image processing to detect defects, it is particularly useful to have a large proportion of the test images near the classification boundaries. If necessary, the available images may be modified artificially to simulate the effects of changes in lighting, to adjust the contrast or modify the noise characteristics.

Test patterns and noise sources may be used to create test images or modify existing sample images to simulate effects which may occur rarely. For example, uneven lighting may be simulated by adding an intensity wedge to an image. The noise sensitivity of an algorithm may be tested by adding increasing levels of random noise to an image to determine the level at which the algorithm becomes too unreliable to be usable.

While some testing may be performed in software, in some applications the system must be tested on-line to achieve a suitably wide range of inputs. However, before getting this far, the developer should be fairly certain that the algorithm is not going to change significantly. Most of the adjustments should be parameter tuning rather than wholesale algorithm changes (unless of course the testing determines that the algorithm is not sufficiently robust or is otherwise deficient in some way).

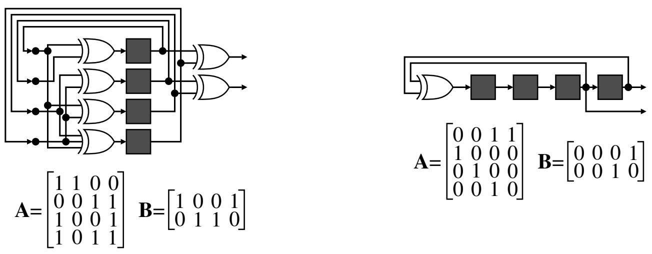
13.1.1 Random Noise Sources

When adding noise to an image, it is necessary to have a source of random numbers. Two main types of random number generators may be implemented on an FPGA: true random number generators and pseudorandom number generators. Two techniques for generating true random numbers (Danger *et al.*, 2007; Tsoi *et al.*, 2007) are to rely on metastability (deliberately violating setup and hold times) so that the output is effectively random, and to sample a high speed clock signal with a slow but unstable clock. The jitter of the latter makes the sampled clock random. Two limitations of true random number generators are that the bit generation frequency is relatively low and that the sequence cannot be regenerated without storing it.

Pseudorandom number generators produce deterministic sequences which pass many of the tests of true random numbers. These work by maintaining a Boolean state vector, **x**, with the next state determined as some function of the current state. The output vector, **y**, is also some function of the current state. The most common form of hardware-based random sequence generator is a *linear recurrence generator*, where the function is given by the matrix:

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{A}\mathbf{x}_n \\ \mathbf{y}_n &= \mathbf{B}\mathbf{x}_n\end{aligned} \qquad (13.1)$$

with all of the operations performed using modulo-2 arithmetic. With this multiplication becomes an AND gate and addition becomes exclusive OR. Since matrices **A** and **B** are constant, this is equivalent to



$$\mathbf{A}=\begin{bmatrix}1&1&0&0\\0&0&1&1\\1&0&0&1\\1&0&1&1\end{bmatrix}\quad \mathbf{B}=\begin{bmatrix}1&0&0&1\\0&1&1&0\end{bmatrix}$$

$$\mathbf{A}=\begin{bmatrix}0&0&1&1\\1&0&0&0\\0&1&0&0\\0&0&1&0\end{bmatrix}\quad \mathbf{B}=\begin{bmatrix}0&0&0&1\\0&0&1&0\end{bmatrix}$$

Figure 13.1 Linear recurrence random number generators. Left: general case with $N = 4$; right: linear feedback shift register with two output bits.

each state variable being the exclusive OR of some combination of the state variables, as shown on the left in Figure 13.1. With appropriate selection of **A**, the length of the sequence is $2^N - 1$, where N is the number of bits in the state vector. Such sequences are called *maximal length* sequences. The -1 comes from the fact that a state vector of zero will always produce a zero result using such a linear combination. Therefore, it is essential to initialise the state vector to be other than all zeros, otherwise a random sequence will not be generated. An implementation of Equation 13.1 optimised for FPGA implementation is described by Thomas and Luk (2005). The multiplication of each row of **A** was implemented using a single 4-LUT on the FPGA to combine up to four state variable bits to produce each output bit, giving a resource efficient implementation.

One class of linear recurrence generator is the *linear feedback shift register* (LFSR), which uses the feedback for only one state variable, with the others obtained by shifting the bits along. Although multiple bits may be obtained from an LFSR, as shown in the right in Figure 13.1, the problem is that the bits are related by a simple shift and are not independent. This problem may be addressed in a number of ways.

- Use several such generators (with different feedback combinations), each producing one bit output. If the generators have coprime maximal lengths, then the length of the output sequence will be the product of the lengths.
- Combine multiple bits together (using a **B** matrix) to improve the independence. The bits combined should be random rather than in a sequence.
- Taking k bits output every k clock cycles. This ensures that an independent set of bits is provided each time. A variation on this is to shift the data by k positions each clock cycle. This effectively uses $\mathbf{A}^k$ rather than **A** as the state transition matrix. An example is shown in Figure 13.2 for $k = 2$.
- A combination of the above can be used to improve the statistical properties of the output.

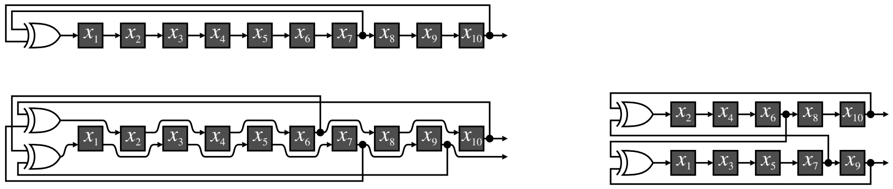

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
x_2 x_4 x_6 x_8 x_{10}
x_1 x_3 x_5 x_7 x_9

Figure 13.2 Producing multiple output bits by shifting several positions each clock cycle. Top: original 10-stage LFSR shifted by one bit; bottom left: shifted by two bits; bottom right: rearranged to show coupled LFSR implementation.

Looking in more detail at Figure 13.2, one 10-stage maximal length LFSR is represented by the feedback:

$$x_1 \leftarrow x_0 = x_{10} \oplus x_7 \tag{13.2}$$

which on the next clock cycle is clocked into x_1. When shifting everything by two places it is necessary to have two sets of feedback, representing successive shifts:

$$\begin{aligned} x_2 \leftarrow x_0 &= x_{10} \oplus x_7 \\ x_1 \leftarrow x_{-1} &= x_9 \oplus x_6 \end{aligned} \tag{13.3}$$

This may be readily extended to shifting any number of places. When shifting multiple places, the state bits fall into distinct groups, each of which is a standard shift register but with cross-coupled feedback, as demonstrated on the bottom right in Figure 13.2. As long as k is not a factor of 2^N-1, then the length before the sequence repeats is unchanged. Obviously for image processing $N = 10$ is far too small. For practical use, the sequence length should be significantly longer than the number of pixels in the image. (For $N = 31$ the following all give maximal length sequences: $x_{31} \oplus x_3$, $x_{31} \oplus x_6$, $x_{31} \oplus x_7$, $x_{31} \oplus x_{13}$, $x_{31} \oplus x_{18}$, $x_{31} \oplus x_{24}$, $x_{31} \oplus x_{25}$, and $x_{31} \oplus x_{28}$. For $N = 32$, there are no two feedback maximal length combinations, but there are many four feedback combinations including $x_{32} \oplus x_{19} \oplus x_{18} \oplus x_{13}$, $x_{32} \oplus x_{27} \oplus x_{24} \oplus x_{12}$, $x_{32} \oplus x_{29} \oplus x_{17} \oplus x_{15}$, and $x_{32} \oplus x_{31} \oplus x_{23} \oplus x_{10}$.)

The numbers produced by most pseudorandom number generators have a uniform distribution. To produce numbers with a different distribution (for example Gaussian) then a lookup table containing the inverse cumulative distribution can be used to transform the input to the corresponding output. This follows the same procedure as histogram shaping illustrated in Figure 7.14. If necessary, to maintain precision, an interpolated lookup table (Section 5.4.2) can be used to map wider data words.

For a Gaussian distributed sequence of random numbers, with programmable mean and variance, an alternative is to use a single lookup table to generate a normal distribution ($\mu = 0$, $\sigma^2 = 1$), then scale this by the standard deviation and offset by the mean:

$$g = \sigma r + \mu, \quad r \in N(0,1) \tag{13.4}$$

Since the transformation from uniform to normal distribution is symmetric, the resolution can be doubled for the same size lookup table, with the MSB used to select addition or subtraction, as shown in Figure 13.3.

If multiple correlated numbers need to be generated, then Equation 13.4 may be adapted by scaling vectors:

$$\mathbf{g} = \mathbf{Sr} + \boldsymbol{\mu} \tag{13.5}$$

where

$$\mathbf{SS}^{\mathrm{T}} = \Sigma \tag{13.6}$$

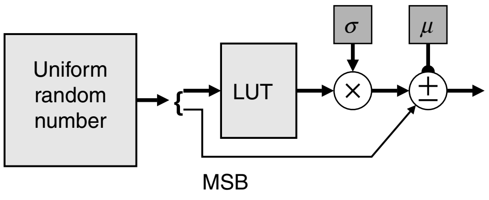


Figure 13.3 Gaussian random number generator.

and **Σ** is the covariance matrix. For a given covariance matrix, **S** is not unique and may be derived by many techniques. To generate n correlated output numbers, the multiplication by **S** requires n^2 individual multiplications. Alternatively, n^2 lookup tables can be used to produce the scaled Gaussians (Thomas and Luk, 2009), which are then added with the mean vectors. That is, Equation 13.5 is expanded to:

$$g_i = \sum_j S_{i,j} r_j + m_i \tag{13.7}$$

where each term within the summation is generated by a separate lookup table.

As well as noise images, other test patterns can also be generated and used for testing algorithms or for modifying images to test a wider range of circumstances.

13.2 Implementation

The mapping from the initial software representation to a hardware implementation can introduce errors within the algorithm. Rather than test the complete application level algorithm as a unit, it is far easier to test each image processing operation and verify that it is behaving as designed.

Verifying each operation level algorithm is simpler than verifying the complete application level algorithm in one step. This is because it is easier to test each operation more thoroughly when directly providing input to that operation. When testing as a sequence, some states or combinations may be difficult or impossible to test given the preceding operations. If each operation behaves the same as its software counterpart, then the complete algorithm should also operate correctly. Testing consists of providing one or more images which exercise each aspect of the algorithm and validate that the correct output is being produced.

While the algorithm can be validated through simulation, this is often quite slow for complex designs. This is another reason for testing each operation individually before combining them to create the application level algorithm. Conventional simulators operate from a hardware perspective and usually provide the waveforms of selected signals. Although this can be useful for validating timing and synchronisation, it is less useful for algorithm debugging, which is best achieved with a software style debugger.

A structure for testing the operation of an algorithm on an FPGA is given in Figure 13.4. It consists of two main sections. The communication with the host is used to load test images into the frame buffer and retrieve the output images. If resources on the FPGA are tight, this may be implemented as a separate configuration from the actual test operation. The second section is the stream interface, which is used to interface between the test image(s) in the frame buffer and the operation under test. It is responsible for reading the data from the frame buffer and converting it into a pixel stream as required for testing the operation. It simulates image capture and any necessary processing steps prior to the operation under test. It also takes the output stream and stores the results back into the frame buffer (or another frame buffer depending on resources available and memory bandwidth). The output is then retrieved from the frame buffer by the host and compared with the expected output (produced for example by running the software

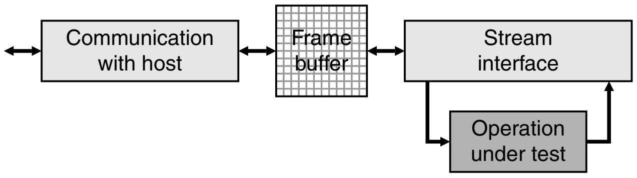


Figure 13.4 Configuration for testing each image processing operation.

version of the algorithm on the host). Any differences indicate potential errors in the implementation. The lack of differences is only significant if the input image exercises all parts of the algorithm.

Testing the algorithm with a low clock frequency will also remove potential timing issues from the test. This enables the test for functionality to be separated from potential clock rate and propagation delay issues.

While such a system can provide a pass or fail indication, it is of limited value in debugging the algorithm. Several methods of debugging were outlined in Section 4.5.2. To review, these are:

- Route key signals to unused I/O ports. This makes them observable from outside the FPGA, enabling monitoring by an external oscilloscope or logic analyser.
- Embedding a logic analyser within the design. Both Xilinx (ChipScope; Xilinx, 2008a) and Altera (SignalTap; Altera, 2008e) provide embedded logic analysers with their development toolsets. These can monitor selected signals, recording their history in block memory enabling later readout and analysis with associated tools.
- Stop the system and read back the configuration to analyse the logic state at the time the system was stopped. When using a hosted system, one of the main limitations is the latency in transferring data between the FPGA and the host system.
- This can be augmented with a small amount of additional logic to enable hardware single-stepping by gating the clock (Tombs *et al.*, 2004). Using logic to detect key events can provide the system with hardware breakpoints. Iskander *et al.* (2010) extended this idea by controlling the process with an on-chip microprocessor that uses a software style debugging interface for debugging the hardware in place. They used the internal ICAP port of Xilinx FPGAs to read back selected variables to provide true hardware debugging.

Of course, any combination of these techniques can be used, combined with simulation of the operation within the development environment.

13.2.1 Common Implementation Bugs

Experience has shown that there are a number of common bugs which occur during the implementation phase. This list is not exhaustive, but serves as a guide for investigating possible causes of malfunctions.

Hardware allows the bit widths of variables to be adjusted to minimise the hardware resources required. A common cause of errors is to underestimate the number of bits required by an application. There are two main consequences of having too few bits: overflow and underflow. Overflow occurs when the numbers exceed the available range. With most arithmetic operations, the effect of overflow is for the numbers to wrap around, becoming completely meaningless. During debugging and development, additional logic can be added to detect overflow. This requires adding an additional bit, x_{MSB+1}, to a register. Overflow is then detected as:

$$Overflow = \begin{cases} x_{MSB+1}, & \text{for unsigned data} \\ x_{MSB+1} \oplus x_{MSB}, & \text{for signed data} \end{cases} \tag{13.8}$$

After thoroughly testing the algorithm, the extra test hardware can be removed in the final implementation. Overflow can be corrected by adding extra bits, or by rescaling the numbers.

At the other extreme, underflow occurs when too many bits have been truncated from the least significant end. The effect of underflow is a loss of precision. Underflow occurs primarily when working with fixed-point designs, or when using multiplication and division. It also shows when reducing the size of lookup tables to reduce resources. Underflow is harder to test for with hardware. During the development stages (in software) a range of designs can be compared with different numbers of least significant bits to check that the errors that arise through truncation are acceptable. Underflow can also be affected by the order in which arithmetic operations are performed, especially when using a floating-point representation. Some applications may require one or more additional least significant guard bits during intermediate steps to reduce the loss of precision.

Both overflow and underflow reflect incomplete or inadequate exploration during the design stages of the project.

Closely related to overflow is the wraparound of array indices (unless this is intentional). A similar problem occurs in C programming, especially when using pointers and pointer arithmetic. Apart from the fact that such errors are errors whether in hardware or software, in a hardware system the effects are usually less severe. In software, with its monolithic memory structure, extending past the end of an array will go into following memory and can corrupt completely unrelated variables. In hardware, the effects are usually local. If the array is not a power of two, then there may be no effect if the hardware past the end of the array does not exist. Otherwise extending past the end of an array will usually wrap back to the start. The localisation of such errors can make them easier to track and locate in hardware than software. If necessary, additional hardware may be built which detects array bounds errors (and overflow of the index variable).

Another error relating to data word length is misalignment of data words when performing fixed-point operations. Most hardware description languages have integers as a native type (even if implemented as a bit array), but do not have direct support for fixed-point numbers. It is over to the user to remember the implicit location of the binary point and align the operands appropriately when performing arithmetic operations. The manual processing required can make alignment tedious to check and misalignment bugs difficult to locate. Errors are easily introduced when changing the size or precision of a variable and not updating all instances of its use. This may be overcome to some extent by using appropriate parameterised macros to maintain the representation and manage the alignment. The design of an appropriate test bench should detect the presence of any data misalignment errors.

The parallel nature of FPGA-based designs can result in additional bugs not commonly encountered in software-based systems. One of the most common of these is to write to a register (or access to any other shared resource) simultaneously from multiple parallel threads. The synthesiser will normally build a multiplexer to select the inputs, and if multiple inputs are selected the result may be a combination of them. Hardware can be added to a design to detect when multiple inputs are selected simultaneously when the multiplexer has been added explicitly to the design (for example using a structural programming style). However, when using a behavioural programming style, such multiplexers are added implicitly and it will be difficult, if not impossible, to detect the selection of multiple inputs.

Multiplexing errors result from bugs within the control logic. These can be caused by the parallel streams not being synchronised properly or by failure to build appropriate arbitration logic. Explicitly building in arbitration (Section 5.3.2) can provide error signals when a parallel thread is blocked from using a resource.

Pipelines with multiple parallel data paths can also suffer from synchronisation problems. Temporal misalignment can occur not only within an operation level algorithm but also within the higher application level algorithm, making such errors hard to detect. Most hardware description languages treat pipelining as simply another case of parallelism and rely on the developer to implement the correct pipeline control (in particular priming and flushing). The higher level, C-based languages, where the compiler automatically detects parallelism from the dataflow and implements pipelining, are better in this regard.

Synchronisation between clock domains is another complex issue prone to error. If throughput is not critical, a channel may be used, but for higher throughput (for example sending pixel data) use of a FIFO is essential. The resulting non-uniform data rate on the receiving side of the transfer can cause a wide range of synchronisation and control issues.

To summarise this section, if the software version of the algorithm behaves correctly then there are two primary sources of error within the corresponding hardware implementation. The first relates to customisation of data word widths and the second relates to the use of parallelism with consequent synchronisation and conflict issues.

13.3 Tuning

Most algorithms require some tuning of parameter values such as threshold levels to optimise their performance. The requirements for tuning an algorithm were covered in some detail in Section 4.5.1. The basic structure for algorithm tuning is shown in Figure 13.5.

Firstly, a mechanism must be provided for dynamically changing the algorithm parameters. Recompiling the application each time a parameter is changed is not particularly practical and methods of modifying the parameter through directly manipulating the configuration file are not particularly transparent or portable. Altera provides a mechanism where selected register and memory contents may be changed or edited manually within the system (Altera, 2008e). A standard approach used by many ASICs is to provide a set of addressable control and status registers. These provide a flexible mechanism for directly setting algorithm parameters and can also be used for monitoring the algorithm status. These registers are usually accessed either via a low speed serial interface or a microprocessor style interface with parallel address and control busses.

The second key requirement is some form of user interface for manipulating the parameter and visualising the results of the changes. On a hosted system, the host computer can readily provide such a user interface. If the two are connected via a high speed interface such as PCI express, then the host may also display resultant image data as well. Otherwise it may be beneficial to provide some form of image display directly from the FPGA. On a stand-alone system, such a user interface may be implemented through an embedded processor. Otherwise the complete interface will need to be implemented directly in hardware (Buhler, 2007).

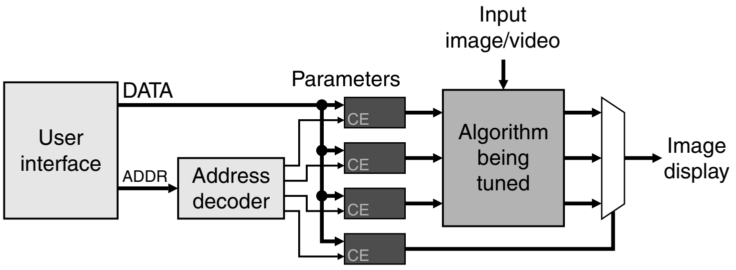


Figure 13.5 Generic structure for algorithm tuning.

13.4 Timing Closure

Timing is a critical issue in any high performance design. *Timing closure* is the step of taking a functionally correct design and getting a working implementation that meets the desired operating speed and timing constraints.

Timing closure has become more difficult with each successive technology generation. Older FPGAs were relatively small, so both the complexity and the speed of a design were limited. Modern FPGAs are significantly larger and capable of being clocked at a much higher speed. Consequently, the scale of applications being implemented on FPGAs has grown, not only in complexity but also in target operating speed. Consider for example image processing. Clock speeds from video cameras are relatively low and many early implementations of image processing on FPGA focussed on simple image processing operations and small local filters. However, the resolution of digital cameras has increased significantly in recent years. To maintain video frame rates, this has required a corresponding increase in pixel clock frequency by almost an order of magnitude. The complexity of image processing algorithms implemented on FPGAs has also grown. As a result, bigger and faster designs are being implemented on bigger and faster chips.

There are two aspects or components to timing closure: one is ensuring that the design on the FPGA can be clocked at the desired rate and the other is meeting the timing constraints of devices external to the FPGA. As device scales have shrunk, an increasing proportion of the propagation delay between registers on the FPGA is coming from the routing (Altera, 2009b). With high clock speeds and large FPGAs, the propagation delay from one side of the chip to the other can exceed the clock period. Therefore, it is essential to keep related logic together to minimise propagation delay. When interfacing to peripherals external to the FPGA, it is important to have the logic placed relatively close to the associated I/O pins. I/O drivers also introduce a significant propagation delay, as do the relatively long tracks on the printed circuit board (PCB) between the FPGA and peripheral. For high speed devices, such as DDR memory, even the skew between parallel signal lines becomes significant (this is why many high speed peripherals are going to even higher speed, serial connections with the clock embedded with the signal). Fortunately, many of these problems, at least within the FPGA, are managed by the FPGA vendor's tools, which can optimise the placement and routing of designs based on timing constraints specified by the developer.

The tools also provide a timing report which details the timing characteristics of the design. Such a report is based on a *static timing analysis* – after the design has been placed and routed on the FPGA, the propagation delay through each component and routing wire can be modelled and analysed against the timing constraints. The report will identify any timing violations, the associated critical paths and determine the maximum operating frequency of a design. However, there are three limitations of such an analysis.

The timing analysis is usually based on the worst case conditions. This usually corresponds with the performance of the slowest device, operating at the maximum temperature and minimum power supply voltage. Consequently, passing under these conditions will guarantee that the propagation delay will be shorter than the clock period, taking into account register setup times and any clock skew. With modern high speed devices it is also necessary to repeat the analysis under best case conditions (Simpson, 2010) (fastest device, operating at the minimum temperature and maximum power supply voltage) to ensure that register hold times are also met, given the worst case clock uncertainty or skew.

The second limitation is that the timing analysis is only as good as the constraints provided by the user. If a path has no timing constraint, it will not be analysed and any timing violation will not be reported. This can cause a design to fail. Timing constraints are used not only for timing analysis, but are also used by the place and route engine to determine where to focus its effort and optimisations during fitting. Changing any of the source files will change the placement seed, resulting in a different initial placement. Changing the timing constraints in general will change the order in which paths are optimised, resulting in a different layout and different timing. It is important to specify realistic timing constraints. If too tight, then the compile times can increase significantly as the place and route tools try to satisfy the constraints. Timing constraints are also essential when interfacing to external devices. In setting such constraints, it is also important to take into account the propagation delay of the PCB tracks and clock skew.

The third limitation of static timing analysis is that it can only identify errors within a synchronous design. With multiple clock domains, the clocks in each domain are usually relatively asynchronous. Even when operating at the same frequency, unless one clock is locked to the other, the phase relationship between them is unknown and cannot be relied on when directly transferring data from one domain to the other. When the frequencies are different, the relative phase is always changing, so specifying timing constraints between domains is virtually impossible. Therefore, it is essential that appropriate synchronisation logic be built whenever signals traverse clock domains. Suitable synchronisation logic was described in Section 5.1.3.

Designs that pass timing analysis but have timing errors typically do so for two reasons. The first is that an appropriate time constraint was not set for the path. With multiphase designs, the setting of suitable constraints is complicated by the fact that the clock might not be enabled on every clock cycle. In such cases it may be perfectly legitimate for the propagation delay to take up to two or three clock cycles before the register is enabled. Specifying such paths as multicycle paths in the constraints file aligns the constraints with reality. (It also avoids unnecessary effort on the part of the place and route tools trying to reduce the propagation delay to smaller than it needs to be.) The second problem is metastability resulting from asynchronous signals. This is caused by inadequate synchronisation of signals between clock domains. Problems caused by asynchronous signals can be much harder to detect and debug because they are often intermittent.

If a design is failing timing constraints, consider the following checklist:

- Use realistic timing constraints. Specifying a higher clock speed will result in a faster design up to a certain point. This comes at the cost of increased optimisation effort giving larger compile times. Setting the desired clock frequency beyond that which can realistically be achieved will drastically increase the compile times, without achieving timing closure. As the device use increases, the achievable clock speeds tend to reduce because there is less freedom available in optimising the critical paths.
- Identify multicycle or false paths. These can give false failure warnings. They also cause the compiler to spend unnecessary effort trying to make these paths conform to the timing constraint.
- Use low level pipelining. Deep logic has a long propagation delay and may need to be pipelined to meet performance targets. Also consider pipelining the control path, as this can also add to the propagation delay, especially with a series of nested conditional statements. Assigning to a register from multiple sources will build a multiplexer on the register input increasing the number of layers of logic. Propagation delay can be decreased if separate registers are used for each instance if the design allows.
- Reduce high fanout. A signal that is used in a large number of places may be subjected to excessive delay, especially if the signal is distributed over a wide area of the chip. Duplicating the driving registers in different parts of the design can reduce the fanout and increase the locality of the design.
- Register I/O signals. When working with high speed signals, both inputs and outputs should be registered within the I/O blocks. Related signals should use the same I/O banks where possible to reduce differences in propagation delay. Some FPGAs have programmable delays in the I/O blocks to balance small differences in delay.
- Use modular design with incremental compilation. If the compile times are excessively long, split the design into modules and use incremental compilation. For this, each module is compiled separately and the layout within modules is not changed when bringing the modules together under the top layer. The reduced compilation times come at the expense of poorer performance for signals spanning between modules (Simpson, 2010). Where possible, module inputs and outputs should also be registered so that the only delay between them is the inter-module routing. Incremental compilation can also result in reduced flexibility in optimising the placement and routing of the top layer of the design. Such techniques can therefore become less effective as the design approaches the capacity of the FPGA.

To conclude, timing and timing closure are not just considerations at the end of the implementation, but need to be considered at all aspects of a design, from the initial specification all the way through design and implementation.

14

Example Applications

In the earlier chapters, much of the focus in describing FPGA implementation was on individual image processing operations, rather than complete applications. This final chapter shows how the individual operations tie together within an application. Of particular interest are some of the optimisations used to reduce hardware or processing time.

14.1 Coloured Region Tracking

In this application, it was desired to create a gesture-based user input. Rather than operate in a completely unstructured environment, it was decided to use coloured paddles to provide the input gestures (Johnston *et al.*, 2005b). One of the goals was to minimise resource use to enable the remainder of the FPGA to be used for the application being controlled (Johnston *et al.*, 2005a).

The basic structure of the algorithm is shown in Figure 14.1. The basic steps within the application were to use the distinctive colours of the paddles to segment them from the background. A simple bounding box was then used to determine the paddle locations. The remainder of this section details the algorithm and the optimisations made to reduce the size of the implementation.

This application was implemented on an RC100 board from Celoxica Ltd. The board uses a Xilinx XC2S200 Spartan II FPGA and has a number of peripheral devices on the board, including a video codec chip. The codec digitised the composite video signal from the camera and provided a stream of 16-bit colour (RGB565) pixels to the FPGA. The codec provided a 27 MHz pixel clock, with one pixel every two clock cycles.

Two counters were implemented to keep track of the input, one to count the pixels on the row, x, and the other to count the row number, y. Both were reset at the end of the respective blanking intervals. The control signals for driving the rest of the application were derived from these counters.

The first stage is to identify the colour of the pixels in the incoming pixel stream. As described in Section 6.3.3, the RGB colour space is not the most suitable for colour segmentation. The RGB pixel values were therefore converted to a form of YCbCr to separate the luminance from the chrominance. Since the output is not intended for human viewing, the exact YCbCr matrix conversion is not necessary.

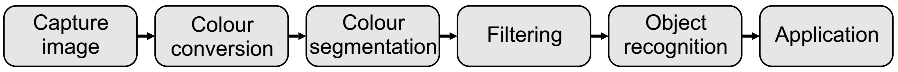


Figure 14.1 Steps within a colour tracking algorithm.

Instead, the simplified YUV colour space of Equation 6.61 was used:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 0 & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{14.1}$$

which can implement the conversion with four additions, as shown in Figure 6.35. The fact that G has one more bit that R and B does not matter as all three values are treated at binary fractions. The resultant YUV components each have seven bits. However, since the transformation is linear, scaling the intensity will scale all three RGB components and, consequently, also the YUV components. Therefore, dividing U and V by Y would provide true intensity normalisation:

$$\begin{aligned} U' &= \frac{U}{Y} \\ V' &= \frac{V}{Y} \end{aligned} \tag{14.2}$$

Unfortunately, with the definition in Equation 14.1, V can exceed Y for some colours, and V' goes outside the range -1 to 1. Therefore, an alternative definition was used for the luminance:

$$Y' = \max(R, G, B) \tag{14.3}$$

This normalises U' and V' to the range $-\frac{1}{2}$ to $\frac{1}{2}$. While such intensity scaling removes the intensity dependence, it also has a side effect of reducing colour specificity. After normalisation, all colours with a similar hue are mapped to the same range. Normalisation therefore introduces a trade-off between insensitivity to intensity and colour selectivity.

The next step in the process is colour segmentation. This detects pixels within a box within $Y'U'V'$ colour space. Rather than build a set of comparators for each colour detected, a variation on the lookup table approach of Figure 6.42 is used. The divisions of Equation 14.2 are relatively expensive (although with the two-phase input clock, a single divider could be shared between the two channels, as shown in Figure 5.22). The divisions are avoided by using the lookup table to perform the division. This is accomplished by concatenating the Y' with the U and V, respectively, as shown on the left in Figure 14.2, and setting the table contents based on Equation 14.2. Of course, during tuning when the colour thresholds are set, the division is required to initialise the LUTs.

To minimise the resource use, it was decided to fit both lookup tables in a single dual-port block RAM (Johnston *et al.*, 2005a). On the target FPGA, each block RAM is 4 Kbits in size. To detect four colours in parallel, the memory needs to be four bits wide. Therefore, the 4 Kbit RAM allows 512 entries for each table, or nine address bits to be divided between the Y' and UV components. Since colour resolution is more important than precise intensity normalisation, six bits were used for each of U and V and three bits were allocated to Y'. While this gives only an approximate result, the three bits are sufficient to give some degree of intensity normalisation.

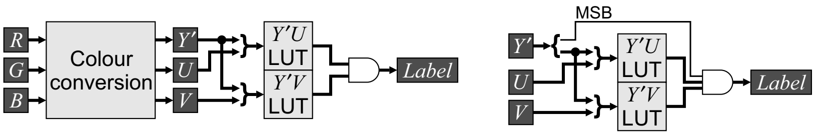


Figure 14.2 Colour segmentation and labelling. Left: basic idea of joint lookup tables; right: gaining an extra bit of precision for Y'.

As the paddles are always lighter coloured, the minimum threshold for Y' is always above the 50% level. This allows four bits to be used from Y', with the three least significant bits passed to the lookup table and the most significant by used directly to control the AND gate. This optimisation is shown on the right in Figure 14.2.

Each coloured object was recognised by finding the bounding box of the corresponding label. However, since the bounding box is sensitive to noise, a 3×3 erosion filter was used to remove isolated noise pixels. These commonly occur on the boundaries between colours, or around specular highlights where the pixel values saturate. The 3×3 filter is separable, decomposing into a 3×1 horizontal filter followed by a 1×3 vertical filter. The row buffers are indexed directly by the pixel counter, x, rather than needing an additional counter. Although erosion shrinks the size of each region, it is not necessary to dilate the image again – it does not affect the centre of the bounding box, and if necessary this can be taken into account when interpreting the bounding box dimensions.

The bounding box implementation of Figure 11.3 is used to determine the bounding box of all the labels in parallel. Fabric RAM is used to avoid the explicit need for a multiplexer. With a two-phase clock, only a single-port RAM is required, reducing the resource requirements. Each LUT gives a 16-bit deep RAM. With four bits used to represent the label, one bit for each colour, it is unnecessary to explicitly convert this to binary. The raw four-bit label is used to directly address the RAM. As described in Section 11.1, each bounding box is augmented with a counter to determine the orientation of long thin objects.

The timing for the coloured region tracking pipeline is shown in Figure 14.3. Signals from the codec are valid on the rising edge of the 27 MHz codec produced clock. The CREF signal produced by the codec is high when pixel data is available (every second cycle). The x register is also incremented in this phase, so that it remains constant between phase 1 when the row buffer and bounding box are read and when they are written in phase 2.

The pipeline timing shows the pixel processing. Additional processing is performed at the end of each line and frame. After each row, the pixel counter is reset and the row counter incremented. A state variable used for determining the orientation is also reset. At the end of each field, the data for each bounding box is transmitted to the main clock domain over a channel (all of the image processing is performed directly in the codec's clock domain). After transmission, the bounding box data is reset for the next field.

The tracking algorithm locates up to four programmed colours in each field at either 50 or 60 fields per second (depending on whether a PAL or NTSC camera is connected). All of the processing is performed

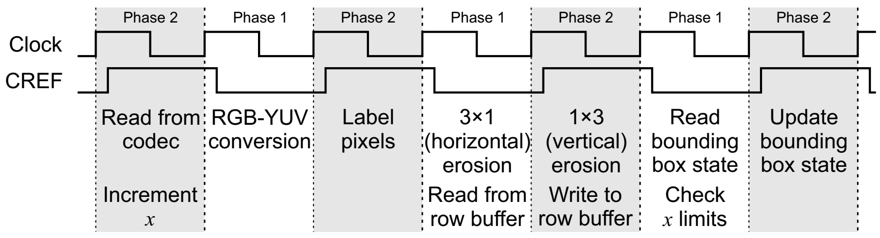


Figure 14.3 Pixel timing for coloured region tracking.

on-the-fly as the data is streamed in. The only pixel buffering is two block RAMs used as row buffers for the vertical erosion filter. The complete tracking system used less than 10% of the logic resources of the FPGA (and 3/14 of the block RAMs), leaving the remainder of the system to be used for the application.

14.2 Lens Distortion Correction

Many low cost lenses used in a lot of image processing applications suffer from lens distortion. Such distortion results from the magnification of the lens not being uniform across the field of view; this is particularly prevalent with wide angle lenses. In many imaging applications, inexpensive wide angle lenses are used because of space and cost constraints. An example of the form of distortion is shown in Figure 14.4 (it has been exaggerated here for effect). This next application looks at characterising and correcting the lens distortion in images being displayed.

Lens distortion is usually modelled using a radially dependent magnification:

$$r_q = r_i M_f(r_i) \tag{14.4}$$

where

$$r_i = \sqrt{x^2 + y^2} \tag{14.5}$$

is the radius about the centre of the distortion in the input image,

$$r_q = \sqrt{u^2 + v^2} \tag{14.6}$$

is the radius of the corresponding point in the output image and M_f is the magnification of the forward mapping. The magnification function is often modelled as a generic radial Taylor series:

$$M_f(r_i) = 1 + \kappa_1 r_i^2 + \kappa_2 r_i^4 + \cdots \tag{14.7}$$

There are two points of note. Firstly, the centre of distortion (the origin in these equations) is not necessarily in the centre of the image (Willson and Shafer, 1994). Two parameters are therefore required to represent the distortion centre, (x_0, y_0). Secondly, for many lenses, a simple first order model is sufficient to account for much of the distortion (Li and Lavest, 1996) even when the distortion is quite severe. With the truncated series, the subscript will be dropped from the κ parameter.

Equation 14.4 represents the forward mapping. From this equation, the inverse (the reverse mapping) may be obtained as another radially dependent magnification:

$$r_i = r_q M_r(\kappa, r_q) \tag{14.8}$$

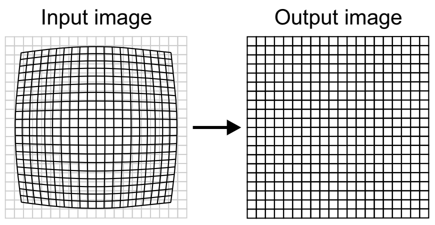


Figure 14.4 Lens distortion correction.

Lens distortion correction can be broken into two phases: the first is calibration, which determines the distortion parameters (x_0, y_0, and κ), and the second is to use the distortion model and calibration parameters to correct the distortion within an image.

14.2.1 Characterising the Distortion

Since the calibration process is not time critical, it is best performed using a software-based system. However, out of interest, an FPGA implementation is presented here. Calibration can use a separate configuration file to distortion correction since both processes do not need to operate concurrently.

While there are a number of methods that can be used to calibrate lens distortion, a variation of the 'plumb line' originally proposed by (Brown, 1971) is used here. It is based on the premise that without lens distortion straight lines should be straight under projection. Any deviation from straightness is, therefore, an indication of distortion (Fryer *et al.*, 1994; Devernay and Faugeras, 2001). Many algorithms for determining the distortion parameters are iterative. Here a direct, non-iterative method will be used.

The basic principle (Bailey, 2002) is to capture an image of a rectangular grid and fit a parabola to each of the gridlines. The quadratic term of the parabola indicates the degree of curvature. Lines through the centre of distortion will be straight, allowing the centre to be estimated from where the sign of the quadratic term changes. The distortion parameter is then derived to make the curved lines straight.

Assuming that the gridlines are approximately one pixel wide, the algorithm for detecting both sets of gridlines is outlined in Figure 14.5. While thresholding can detect the gridlines, the resulting lines are often thicker than a single pixel. Two sets of filters in parallel are used to solve this problem. The gridlines are separated by using two small one-dimensional erosion filters: a vertical 1×3 erosion will remove the horizontal gridlines and a horizontal 3×1 erosion will remove the vertical gridlines. A second pair of filters (one horizontal and one vertical) operates on the original image to detect the local minimum pixel positions. This is masked with the detected gridlines to select the best pixel. With both sets of filters, the horizontal filters follow the first row buffer to offset them by one line, giving correct temporal alignment of the outputs. Similarly, the output of the vertical filters is delayed by one clock cycle for alignment.

In the output, there may still be the occasional double pixel, or the occasional missing pixel where the gridlines cross. To correct for this, a small 3×3 cleaning filter is used. For horizontal lines, to bridge narrow gaps if there is a pixel detected in the left and right of the window, but not in the centre column, a linking pixel is added. To reduce the thickness of double wide lines, if two pixels are adjacent vertically then one is removed. A similar filter is used for cleaning vertical lines. The filters are given in Figure 14.6.

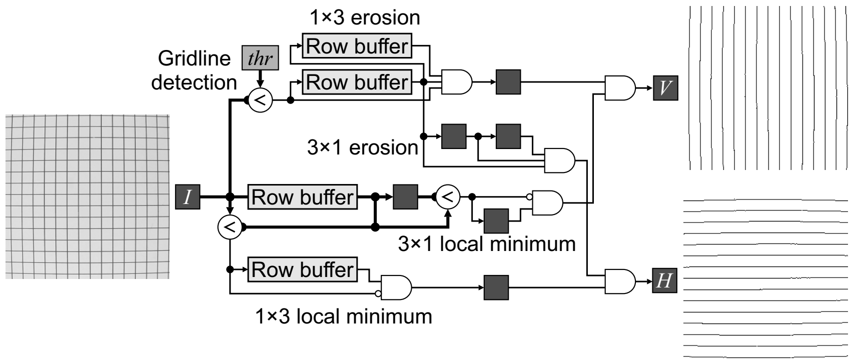
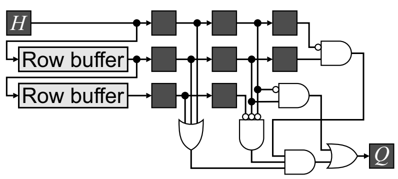


Figure 14.5 Algorithm for grid detection.



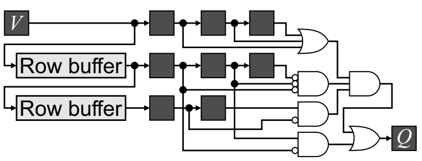


Figure 14.6 Filters for filling in small gaps and thinning double wide lines. Left: for horizontal gridlines; right: for vertical gridlines.

A version of connected components analysis is then used to label each gridline and extract the data required to determine the parabola coefficients. The single pass algorithm given in Section 11.4.4 can be simplified, because the number of labels is relatively small and unchaining is not necessary.

Firstly consider the horizontal gridlines. The parabola fitted to each gridline is:

$$y = ax^2 + bx + c \tag{14.9}$$

Using a least squares fit to determine the coefficients requires solving:

$$\begin{bmatrix} \sum x^4 & \sum x^3 & \sum x^2 \\ \sum x^3 & \sum x^2 & \sum x \\ \sum x^2 & \sum x & \sum 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum x^2 y \\ \sum xy \\ \sum y \end{bmatrix} \tag{14.10}$$

This poses a number of problems from an implementation perspective. Consider a 1024 × 1024 image with 1024 pixels across each row. The sum of x^4 requires 48 bits to represent. Defining the origin in the centre of the image improves this, reducing the representation to 44 bits. A second problem is the wide range in scales between the terms. Both problems may be solved by scaling x and y by a power of two, so that the absolute value at the edges of the image is between one and two. In this case, scaling by 2^8 allows a 20-bit representation for all numbers (12 integer bits and eight fraction bits).

If each horizontal line has a pixel in every column, then the matrix on the left of Equation 14.10 is actually a constant and does not need to be accumulated. The inverse can also be calculated off-line and hardwired into the implementation, enabling the coefficients to be calculated as a weighted combination of the terms on the right hand side. Incremental update can be used calculate $\sum xy$, $\sum x^2y$, and $\sum xy^2$ without multiplications.

The centre of distortion is determined from where the quadratic term of the set of parabolas goes to zero. Therefore, the horizontal gridlines can be used to calculate y_0 and the vertical gridlines to calculate x_0. Solving for where the curvature is zero from the horizontal gridlines gives:

$$y_0 = \frac{\sum ca \sum c - \sum a \sum c^2}{\sum ca \sum 1 - \sum a \sum c} \tag{14.11}$$

and similarly for the vertical gridlines. After determining the centre, each of the parabola coefficients needs to be adjusted to the new origin:

$$\begin{aligned} \hat{x} &= x - x_0 \\ \hat{y} &= y - y_0 \end{aligned} \tag{14.12}$$

Substituting into Equation 14.9 for the horizontal gridlines adjusts the parabola coefficients for the new origin:

$$\begin{aligned}\hat{y} &= a(\hat{x} + x_0)^2 + b(\hat{x} + x_0) + c - y_0 \\ &= a\hat{x}^2 + (2ax_0 + b)\hat{x} + (ax_0^2 + bx_0 + c - y_0)\end{aligned} \tag{14.13}$$

Each gridline can then be used to estimate the distortion parameter (Bailey, 2002):

$$\kappa = \frac{-a}{c(3ac + 3b^2 + 1)} \tag{14.14}$$

Since each gridline (both horizontal and vertical) provides a separate estimate of the distortion parameter, the estimates can be combined by averaging. A weighted average should be used because the division by c in the denominator makes the estimate less accurate near the centre of the image. Therefore:

$$\kappa = \frac{\sum \frac{-a}{3ac + 3b^2 + 1}}{\sum c} \tag{14.15}$$

If necessary, additional steps can be taken to also calibrate the perspective distortion associated with the camera not being perpendicular to the grid (Bailey, 2002; Bailey and Sen Gupta, 2010).

Apart from the initial image processing and connected components labelling, much of the computation in this application is probably best performed in software, since it is largely sequential in nature. This could either be through an embedded processor or through a custom processor with an instruction set designed specifically for the types of computation used.

14.2.2 Correcting the Distortion

A number of papers have been written on the correction of radial distortion using FPGAs. Three different approaches have been taken to this problem. The first (Oh and Kim, 2008) transforms the incoming distorted image on-the-fly, resulting in a corrected image being stored in a frame buffer on the output. The second (Eadie *et al.*, 2002; Gribbon *et al.*, 2003) transforms the output image on-the-fly, taking the input from the frame buffer. The third approach transforms the image from one buffer to another. This allows the technique to exploit symmetry to reuse computation (Ngo and Asari, 2005) or divide the image into tiles to reuse pixel values for interpolation without running into memory bandwidth problems (Bellas *et al.*, 2009).

A range of different distortion models have also been used, from a two-dimensional polynomial warp (Eadie *et al.*, 2002), to a high order radial distortion model (Ngo and Asari, 2005; Oh and Kim, 2008), to the simple first order model described in the previous section (Gribbon *et al.*, 2003) to a dedicated model of fish-eye distortion (Bellas *et al.*, 2009). Most implementations used bilinear interpolation, although Bellas *et al.* (2009) used the more demanding bicubic interpolation.

The approach taken here is to correct a distorted image on-the-fly as it is being displayed from a frame buffer. This requires the reverse mapping to determine where each output pixel is to come from in the input image. Substituting Equations 14.4 and 14.7 into Equation 14.8 gives:

$$\begin{aligned}M_r(\kappa, r_q) &= \frac{r_i}{r_q} = \frac{1}{M_f} = \frac{1}{1 + \kappa r_i^2} \\ &= \frac{1}{1 + \kappa M_r^2 r_q^2}\end{aligned} \tag{14.16}$$

therefore

$$M_r + \kappa r_q^2 M_r^3 = 1 \tag{14.17}$$

The reverse magnification function depends on the product κr_q^2 rather than each of the terms separately. Therefore, a single magnification function can be implemented and used for different values of κ. Secondly, it is not necessary to take the square root to calculate r_q as in Equation 14.6; this simplifies the computational requirements of determining the mapping.

Incremental calculation can be used to calculate r_q^2. From Equation 14.6:

$$r_q^2 = u^2 + v^2 \tag{14.18}$$

Moving from one pixel to the next:

$$(u+1)^2 = u^2 + 2u + 1 \tag{14.19}$$

just requires a single addition (since the 1 can be placed in the least significant bit when u is shifted). A similar calculation can be used when moving from one line to the next during the horizontal blanking interval.

Solving Equation 14.17 for the reverse magnification function gives:

$$M_r(\kappa, r_q) = M_r(\kappa r_q^2) = \frac{1}{3\kappa r_q^2 W} - W \tag{14.20}$$

where

$$W = \sqrt[3]{\sqrt{\frac{1}{4\left(\kappa r_q^2\right)^2} + \frac{1}{27\left(\kappa r_q^2\right)^3}} - \frac{1}{2\kappa r_q^2}} \tag{14.21}$$

Clearly this is clearly impractical to calculate on-the-fly. This function is plotted in Figure 14.7. Since it is quite smooth, it may be implemented efficiently using either an interpolated lookup table or multipartite tables (Section 5.4.2).

The accuracy of the representation depends on the desired accuracy in the corrected image. For a 1024×1024 image, the maximum radius is approximately 725 pixels in the corner of the image. Therefore, to achieve better than one pixel accuracy at the corners of the image, the magnification needs to have at least 10 bits accuracy. To maintain good accuracy, it is desired to represent the magnitude with at

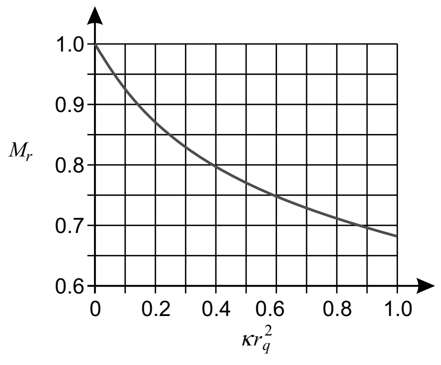


Figure 14.7 Reverse magnification function.

least 15 bits accuracy. For κr_q^2 in the range of 0 to 1, direct lookup table implementation would require 2^{15} entries. An interpolated lookup table only requires 2^8 entries, but needs a multiplication to perform the interpolation.

The next step is to use the magnification function to scale the radius. One approach for achieving this (Ngo and Asari, 2005) is to use a CORDIC engine to convert the output Cartesian coordinates to polar coordinates, scale the radius by the magnification function and then use a second CORDIC engine to convert polar coordinates back to Cartesian coordinates in the input image. However, such transformations are unnecessary because the angle does not change. From similar triangles, scaling the radius can be achieved simply by scaling the u and v components directly:

$$\begin{aligned} x &= uM_r \\ y &= vM_r \end{aligned} \tag{14.22}$$

The resulting pipeline is shown in Figure 14.8. At the start of the frame, the v and v^2 registers are initialised based on the location of the origin. Similarly, u and u^2 are initialised at the start of each line, with v and v^2 updated for the next line. The interpolated lookup table was implemented using a single 256×16 dual-port block RAM as described in Section 5.4.2. The -3 in the u line is to account for the pipeline latency; u is incremented three times while determining the magnification factor. An alternative to the subtraction is to add three pipeline registers to the u line before multiplying by the magnification factor.

The last stage is to use the calculated position in the input image to derive the corresponding output pixel value. For this, bilinear interpolation was used (Gribbon and Bailey, 2004). Any of the techniques described in Section 9.3.1 could be used. Caching is simplified by the fact that the magnification factor is less than one. This means that of the four pixel values required for bilinear interpolation, at least two (and usually three) will be in common with the previous position or previous row. Therefore, with appropriate caching, only one or two new pixel values will need to be loaded to calculate each output pixel value. This can be achieved using caching with preload, using the circuit in Figure 14.9. The operation is as follows.

The row caches store data based on the two least significant bits of the corresponding row address (y_i). The cache row records the two LSBs of the row that was most recently loaded for each column, to enable the preload to load the next free row. As each data is extracted from the cache, the cache control determines which new location to fetch from memory based on the current location, and next row to load.

Having multiple parallel cache rows enables all the required pixels for the interpolation window to be accessed in parallel in a similar manner to window filters. Firstly, the incoming pixel coordinates are split into their integer and fractional components. To access the pixels from the cache, x_i is compared with the previous value and, if different, the values from column $x_i + 1$ are read from all four cache rows.

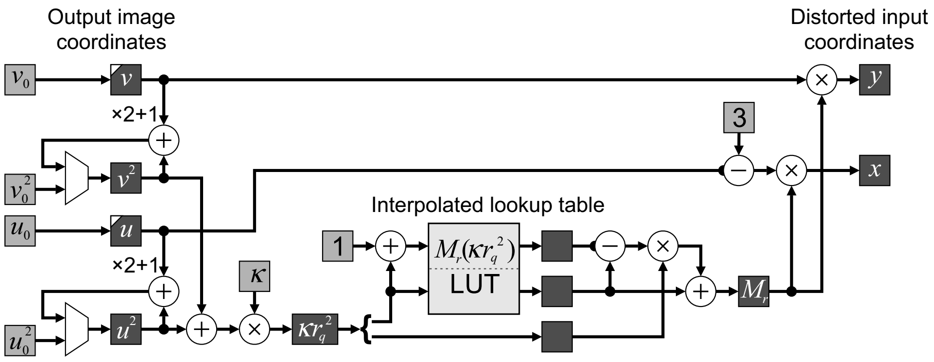


Figure 14.8 Pipeline for correcting lens distortion.

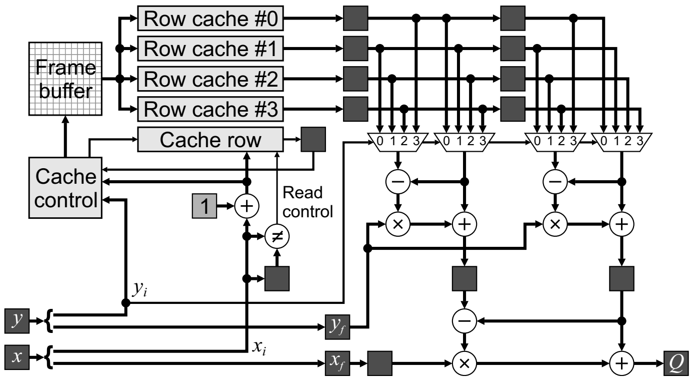


Figure 14.9 Bilinear interpolation for lens distortion correction.

Values from the previous column are shifted along, giving a 2×4 window into the input image. The two least significant bits of y_i are used to select the required rows for the 2×2 window. These values are then weighted according to the fractional parts to produce the output pixel value. The pipeline from the input coordinates to the output pixel value has three stages.

To prime the row caches will require the first row to be loaded three times. This may be accomplished during the vertical blanking period before the first row is output. Similarly, it is necessary to prime each row by loading the first pixel twice. The extra control logic required to achieve the priming is not shown here.

This application has shown how an image with lens distortion may be corrected on-the-fly as it is being displayed. To correct the image on-the-fly as it is being loaded from the camera will require quite a different transformation. A related example, that of introducing a predefined distortion into an image being captured is shown in the next section.

14.3 Foveal Sensor

Low cost high resolution sensors (up to 10 megapixels and larger) are now becoming commonplace. This presents both a bonus and a problem for image processing. The increased resolution means that more accurate and higher quality results can be produced. However, there is also a significantly increased computational cost to achieve these results, especially when real-time constraints have to be met. Therefore, there is often a trade-off between quality of results and processing requirements.

Single high resolution images may be readily processed in real time on an FPGA if stream processing can be used. However, any algorithm that requires multiple frames will require significant off-chip memory and large associated memory bandwidth. Simply reducing the data volume by reducing the resolution can result in a critical loss of information. In many applications (for example tracking and pattern recognition), high resolution is only required in a small region of the image, although a wide field of view is important to maintain context. One solution to this dilemma is to use a foveated window. Inspired by the human visual system, this maintains high resolution in the fovea, with resolution decreasing in the periphery. It has been shown that such a variable spatial resolution can reduce the volume of data in tracking applications by a factor of 22 (Martinez and Altamirano, 2006) to 64 (Bailey and Bouganis, 2009b) without a severely affecting the tracking performance. Several researchers have investigated foveal mappings using FPGAs.

Traditionally, a log-polar mapping has been used for foveal vision in software to mimic the change of resolution within the human visual system (Wilson and Hodgson, 1992; Traver and Pla, 2003). The log-polar map is both rotation and scale invariant once the fovea is positioned on a common key point. Arribas and Macia (1999) implemented a log-polar mapping using an FPGA that was able to

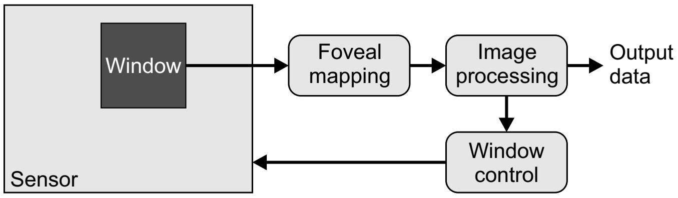


Figure 14.10 Active vision architecture using a foveal mapping.

achieve real-time performance. They represented the mapping as a full lookup table from each input pixel to the corresponding output pixel. The image was mapped from a frame buffer rather than directly as it was streamed from the camera.

The complexities of processing a log-polar image have led many researchers to work with Cartesian approximations. Camacho *et al.* (1998) used a pyramidal-based fovea with each successive level decreasing the resolution by a factor of two towards the periphery, giving a form of stepped log-Cartesian topology. Ovod *et al.* (2005) implemented multiple fovea, allowing multiple objects to be tracked simultaneously, although their system only allowed two levels of resolution: fovea and periphery. Martinez and Altamirano (2006) used high resolution rectangular sampling within the fovea and approximated a log-polar mapping in the periphery by mapping multiple square rings to a rectangular output image using simple subsampling. Bailey and Bouganis (2008) introduced a family of Cartesian foveal mappings with continuously variable resolution. To minimise latency, the foveal image was mapped on-the-fly as the pixels were streamed from the camera.

An important part of foveal vision systems is the use of active vision techniques (mechanical or electronic-based systems) to ensure that the high resolution part of the sensor corresponds to the region of the scene where it can be most effective. Active vision requires minimising the latency between image capture and repositioning the fovea, because delays may degrade the performance of the application and limit the stability of this control loop. The lower resolution of the foveal images can significantly increase their processing rate, enabling real-time camera control.

A CMOS sensor with a wide angle lens enables a wide field of view without a loss of resolution compared to that of a standard camera. CMOS sensors also allow an arbitrary rectangular window of the sensor to be selectively read out; this window may be positioned anywhere within the active area of the camera. Repositioning the window from one frame to the next provides the digital equivalent of pan and tilt, without the physical latency and motion blur associated with physically moving the camera. An active foveal vision system based on this mechanism (Bailey and Bouganis, 2008; 2009a; 2009c) is illustrated in Figure 14.10. The pixels within the window undergo a foveal mapping to give a reduced resolution image. This image is then processed to extract the required data and to determine the best position to relocate the window to within the CMOS sensor for the next frame.

14.3.1 Foveal Mapping

A separable foveal mapping (Bailey and Bouganis, 2009a) will be used because it is easier to implement and it has been demonstrated through simulations (Bailey and Bouganis, 2009b) that its performance in tracking applications is similar to more complex radial mappings. The mapping here will map a 512 × 512 window to a 64 × 64 foveal image. An example of the foveal mapping for:

$$\begin{aligned} x &= u + \frac{7}{32}u^2\operatorname{sign}(u) \\ y &= v + \frac{7}{32}v^2\operatorname{sign}(v) \end{aligned} \tag{14.23}$$

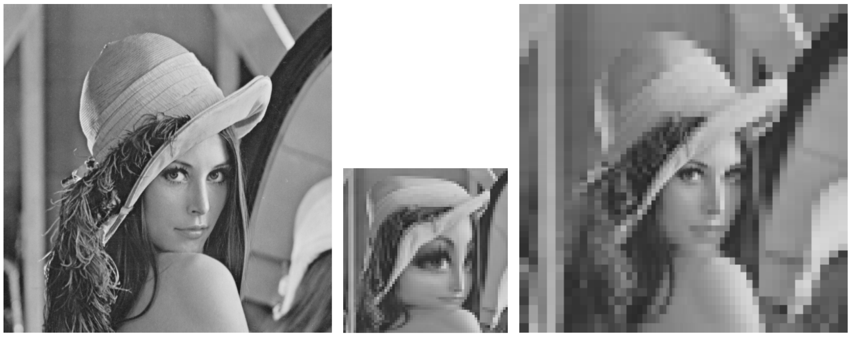


Figure 14.11 Example foveal mapping. Left: original 512 × 512 image; centre: after the foveal map to 64 × 64; right: reconstructed to the original size, showing the reduction in resolution in the periphery. (Reproduced with permission from D.G. Bailey and C.S. Bouganis, "Reconfigurable foveated active vision system," *International Conference on Sensing Technology*, 162–169, 2008. © 2008 IEEE.)

is illustrated in Figure 14.11. Note that due to the symmetry of the mapping, signed coordinates are used, with the origin in the centre of the images.

In the previous section, a reverse mapping was used to correct for lens distortion. Here it is desirable to use a forward mapping for several reasons. Firstly, it saves the need for a frame buffer to hold the much larger input image. Secondly, the forward mapping enables the image to be transformed on-the-fly as it is streamed from the camera, significantly reducing the latency. Thirdly, the reduction in resolution in the periphery requires a spatially variant anti-aliasing filter. This can easily be incorporated into the forward mapping as described in Section 9.1.

The basic architecture of the forward mapping is shown in Figure 14.12. In the first stage, the incoming pixels are mapped into their correct column and accumulated until the output pixel is completed horizontally. It is then passed to the Y mapping, which adds it to the appropriate column accumulator (stored in the row buffer). Once each row is completed vertically, the completed output pixels are normalised (the accumulated total is divided by the area) and saved to the foveal image buffer.

Since each output pixel is contributed to by many input pixels, it is necessary for each pixel accumulator, *Acc*, to consist of two component registers: one to accumulate the pixel value, *Acc.V*, and one to accumulate the area (or number of pixels), *Acc.A*. It can therefore be considered as a vector:

$$Acc = \begin{bmatrix} Acc.V \\ Acc.A \end{bmatrix} \tag{14.24}$$

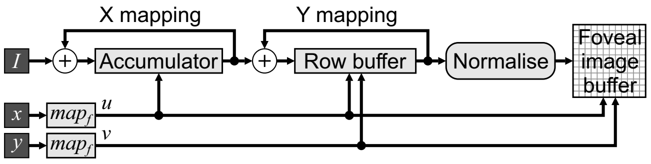


Figure 14.12 Architecture of the separable foveal mapping. (Reproduced with permission from D.G. Bailey and C.S. Bouganis, "Implementation of a foveal vision mapping," *International Conference on Field Programmable Technology*, 22–29, 2009. © 2009 IEEE.)

For a continuously variable resolution function, some input pixels will need to be split between adjacent output pixels. Although Figure 14.12 indicates a forward mapping, it is actually more convenient to implement this by using a reverse mapping function. The index of the current output pixel being accumulated is looked up in a table to obtain the position of its edge in the input image (this is a reverse mapping). The input pixels can then be accumulated until this edge is reached, with the last fraction of a pixel added before the accumulated pixel is passed to the next stage. An advantage of using the reverse mapping function is that the output image has fewer pixels, so the table is significantly smaller. It also manages the splitting of the input pixel more naturally (Bailey and Bouganis, 2009a). The circuit for implementing the foveal mapping is shown in Figure 14.13.

Consider firstly the X mapping. If an incoming pixel is not split, it is simply added to the accumulator. A split pixel is detected if the input pixel position matches the integer part of the edge of the output pixel. In that case the fractional part of the edge, w_x, is used to weight the incoming pixel, which is combined with the accumulator and passed to the output register P_x. The remainder of the pixel is used to update the accumulator:

$$Acc = \begin{cases} Acc + \begin{bmatrix} I \\ 1 \end{bmatrix}, & \text{if not split} \\ (1-w_x)\begin{bmatrix} I \\ 1 \end{bmatrix}, & \text{if split} \end{cases} \tag{14.25}$$

and

$$P_x = Acc + w_x\begin{bmatrix} I \\ 1 \end{bmatrix} \tag{14.26}$$

Since w_x is typically only a few bits, it can be implemented efficiently in the FPGA fabric using adders rather than dedicated multipliers. The incoming pixel area is always one, so a multiplier for that is not required.

A token passing mechanism is used to indicate that valid data is available. Therefore, since a pixel is completed whenever a split occurs, T_x is asserted and the current value of u is registered with it.

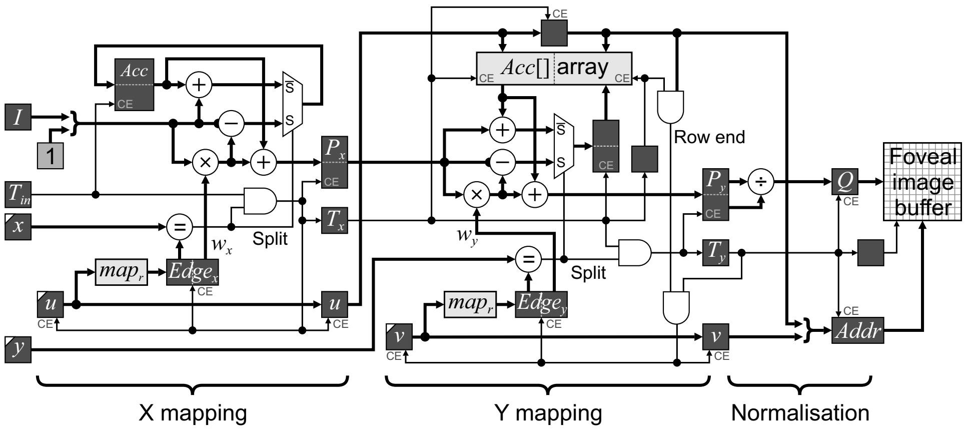


Figure 14.13 Detailed foveal mapping circuit. (Adapted from Bailey and Bouganis, 2009a.)

The Y mapping is very similar to the X mapping logic. The difference is that the pixels are accumulated in columns, although the data is still streamed horizontally. Therefore all columns are mapped in parallel, with the accumulators maintained in a memory-based array. The incoming *u* selects the accumulator to add the new data to, with the accumulated data written back to the array in the following clock cycle:

$$Acc[u] = \begin{cases} Acc[u] + P_x, & \text{if not split} \\ (1-w_y)P_x, & \text{if split} \end{cases} \tag{14.27}$$

and

$$P_y = Acc[u] + w_y P_x \tag{14.28}$$

with corresponding output valid token, T_y. Since the mapping is separable, on a split row, all of the pixels will be split. The *v* counter is only incremented at the end of split rows, with an AND gate over the bits of the *u* address used to detect the end of row.

The final processing stage is to normalise the accumulated pixel value by the area to obtain the average input value within region spanned by each output pixel:

$$Q[u,v] = \frac{P_y.V}{P_y.A} \tag{14.29}$$

The division may be implemented efficiently using eight iterations (for eight 8 bits output) of a non-restoring division algorithm (Bailey, 2006). If necessary, the division can be pipelined to achieve the desired clock frequency.

If desired, the circuit can be optimised further (Bailey and Bouganis, 2009a). Since the Y mapping only uses the map lookup table at the end of each row, a single table can readily be shared between the X and Y mapping sections, as demonstrated on the left in Figure 14.14. The mapping is also symmetrical, therefore only one half needs to be stored in the table. This requires additional logic to mirror the input and foveal coordinates. The fraction also needs to be subtracted from one for the first half of the table to reflect this mirroring. With mirroring, the end of the row is no longer detected by the mapping, so this must be detected explicitly. The mirroring logic is shown on the right in Figure 14.14.

These optimisations reduce the memory required by the lookup table by a factor of four. An additional benefit is that it the foveal mapping itself is changed dynamically, it only needs to be changed in one place.

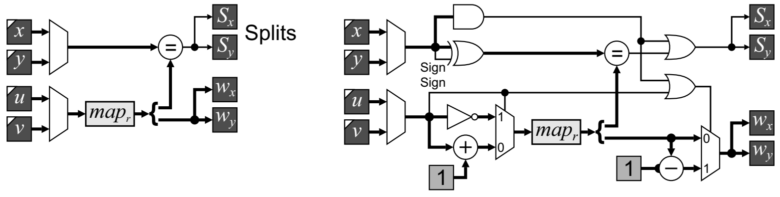


Figure 14.14 Sharing the mapping between the X and Y mapping sections. Left: simple sharing; right: reducing the map LUT size by mirroring.

14.3.2 Using the Sensor

The primary applications of a foveal sensor are in pattern recognition and tracking. In this section, the use of the sensor for object tracking is described briefly. When tracking an object, it is desired to centre the fovea on the target as it moves through the scene. The following stages are required within any tracking system (Bailey and Bouganis, 2009b):

1. The high resolution fovea is positioned on the expected position of the target within the image. Precise positioning is not essential because the lower resolution periphery enables a small image to be used while still maintaining a wide field of view.
2. The target is detected within the foveal image. Any standard object detection technique, such as filtering, thresholding, or colour detection can be used. Processing is relatively quick because of the small image size.
3. The true location of the target is estimated, based on the fact that the image is distorted. Note that standard correlation and registration-based techniques will give a biased result because of the distortion introduced in the mapping. However, for a given mapping, this bias may be estimated and compensated for. If using the centre of gravity, it is necessary to take into account the size and position of each pixel and weight accordingly (Bailey and Bouganis, 2009b).
4. The detected target is fed into a tracking system (for example a Kalman filter; Lee and Salcic, 1997; Liu *et al.*, 2007) and the next location of the target is predicted. This is then used to reposition the window for the next image captured.

Using a foveal sensor has a number of advantages over a conventional sensor in this application. It allows a lower resolution image to be processed, which allows more complex processing with reduced latency. For tracking, the latency requirements make it important that the input image is converted to a foveal image on-the-fly. Multiple targets can be tracked by alternating the fovea between the targets with successive images captured (this reduces the effective frame rate for each target).

14.4 Range Imaging

Standard imaging techniques only obtain two-dimensional data of a scene. To capture a three-dimensional image, it is also necessary to estimate the range of each pixel. Two commonly used techniques for obtaining the range are stereo imaging, which captures two images of the scene from slightly different viewpoints and uses the disparity between the views to estimate the range, and time of flight imaging, which transmits a light signal and measures the round trip propagation time of the light from the source to each point in the scene and back to the sensor. Time of flight sensing is considered in this example.

The distance to each point on an object is:

$$d = \frac{ct}{2} \tag{14.30}$$

where t is the round trip propagation time and c is the speed of light. Since it is difficult to measure the time of flight for a large number of pixels simultaneously, it is more usual to modulate the amplitude of the light source with a sinusoid and measure the phase shift of the envelope of the returned light (for example using synchronous detection). The distance from Equation 14.30 then becomes:

$$d = \frac{c}{2f}\left(k + \frac{\theta}{2\pi}\right) \tag{14.31}$$

where f is the modulation frequency, θ is the measured phase shift and k is an integer number of complete cycles (to account for the phase ambiguity). Since k is unknown, it is usually assumed to be zero, with a maximum unambiguous range of:

$$d_{\max} = \frac{c}{4\pi f} \tag{14.32}$$

Two image capture modalities are commonly used. Heterodyne imaging (Jongenelen *et al.*, 2008) uses slightly different modulation frequencies at the transmitter and sensor. The result is a beat frequency as the phase changes slowly with time. Capturing a sequence of images spaced exactly over one cycle of the beat waveform enables the phase at each pixel to be determined using a single bin Fourier transform. The alternative, homodyne imaging (Jongenelen *et al.*, 2010), uses identical frequencies for the transmitter and sensor. The phase of the transmitter is offset from that of the receiver successively by a fixed amount each frame, again enabling a single bin Fourier transform to be used to measure the phase delay. In both cases, the frequency synthesis features of the FPGA can be used to provide phase locked clocks for both the light source and the sensor.

Given a sequence of images, $I_i[x,y]$, taking the temporal Fourier transform at each pixel:

$$\begin{aligned} F[x,y] &= \sum_{i=0}^{N-1} I_i[x,y] e^{-j2\pi i/N} \\ &= \sum_{i=0}^{N-1} I_i \cos\left(\frac{2\pi i}{N}\right) - j\sum_{i=0}^{N-1} I_i \sin\left(\frac{2\pi i}{N}\right) \end{aligned} \tag{14.33}$$

gives the phase shift:

$$\theta[x,y] = \tan^{-1}\frac{\mathrm{Im}\{F[x,y]\}}{\mathrm{Re}\{F[x,y]\}} = \tan^{-1}\frac{-\sum_{i=0}^{N-1} I_i \sin\left(\frac{2\pi i}{N}\right)}{\sum_{i=0}^{N-1} I_i \cos\left(\frac{2\pi i}{N}\right)} \tag{14.34}$$

and amplitude:

$$A[x,y] = |F[x,y]| = \sqrt{\left(\sum_{i=0}^{N-1} I_i \cos\left(\frac{2\pi i}{N}\right)\right)^2 + \left(\sum_{i=0}^{N-1} I_i \sin\left(\frac{2\pi i}{N}\right)\right)^2} \tag{14.35}$$

The phase shift is proportional to the range, and the amplitude provides a standard intensity image.

A direct implementation of this is given on the left in Figure 14.15. The sine and cosine values can be obtained from a small table (only N entries are needed) or, since they are constant for each whole frame,

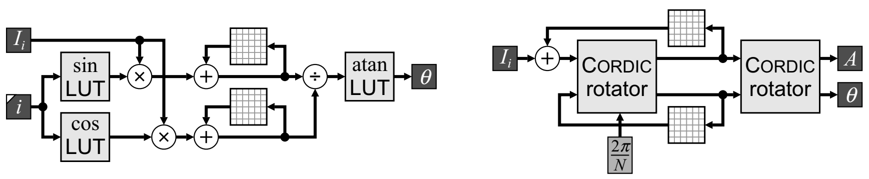


Figure 14.15 Calculating the phase shift at each pixel. Left: direct implementation of Equation 14.34; right: recursive implementation using CORDIC.

may be loaded into registers at the start of each successive frame. Obviously, if $N = 4$ everything is simplified since the sine and cosine terms are -1, 0 or 1 and no multiplication is needed. However, the general case is considered here.

As each frame is streamed from the sensor, it is scaled by the sine and cosine terms and added into a complex accumulator image. For small images, the accumulator image may be held on-chip in block RAM. However, for larger images, it will need to be off-chip in a frame buffer. On the Nth frame, the imaginary component is divided by the real component and the arctangent taken. This can be through a suitable lookup table, which can also include the scale factor required to directly give the range. Additional logic is also required to obtain the amplitude from Equation 14.35. Alternatively, the division, arctangent and amplitude calculation may be performed by a single CORDIC unit.

The processing may be simplified by rearranging Equation 14.33 into recursive form. Firstly define:

$$\begin{aligned} F_n &= \sum_{i=0}^{n} I_i e^{j2\pi(n+1-i)/N} \\ &= (F_{n-1} + I)e^{j2\pi/N} \end{aligned} \tag{14.36}$$

then

$$F = F_{N-1} = \sum_{i=0}^{N-1} I_i e^{j2\pi(N-i)/N} = \sum_{i=0}^{N-1} I_i e^{-j2\pi i/N} \tag{14.37}$$

The recursive multiplication by $e^{j2\pi/N}$ corresponds to a rotation and may be implemented either using a complex multiplication (Figure 10.8) or by using a compensated CORDIC rotator operating in rotation mode (Section 5.4.3). Either way, since the multiplication is by a constant, it can be optimised to reduce the required logic. To calculate both the angle and magnitude, a second CORDIC rotator may be used, this time operating in vectoring mode. This does not need to use compensated CORDIC because scaling the amplitude will scale all pixels equally. The scale factor to convert the angle to a range can also be built into the CORDIC angle tables to avoid the need for an additional multiplication.

If using a running Fourier transform to obtain an update every frame, the accumulation of Equation 14.33 should be used with $I_i - I_N$ as input. The recursive form of Equation 14.36 is only marginally stable (Duda, 2010) and any rounding or truncation errors may make it unstable with errors accumulating and growing exponentially.

14.4.1 Extending the Unambiguous Range

The unambiguous range is limited by assuming $k = 0$ in Equation 14.31. While simply reducing the modulation frequency will extend the range, this comes at the cost of reduced accuracy. However, by making two measurements with different modulation frequencies, the unambiguous range can be extended (Jongenelen *et al.*, 2010; 2011). Let φ be the phase represented as a binary fraction:

$$\varphi = \frac{\theta}{2\pi} \tag{14.38}$$

then with two measurements, Equation 14.31 is extended to:

$$d = \frac{c}{2f_1}(k_1 + \varphi_1) = \frac{c}{2f_2}(k_2 + \varphi_2) \tag{14.39}$$

where k_1 and k_2 represent the number of times the phase terms φ_1 and φ_2 have wrapped respectively. Let the ratio between the two frequencies be expressed as a ratio of co-prime integers:

$$\frac{f_1}{f_2} = \frac{M_1}{M_2} \tag{14.40}$$

To determine the range, it is necessary to solve Equation 14.39 for $0 \leq k_1 < M_1$ and $0 \leq k_2 < M_2$. Unfortunately, it is not as simple as this since, as a result of inevitable noise in the measurements, the two estimates may not match exactly.

The naïve approach is to evaluate all combinations of k_1 and k_2 to find the combination that has the smallest absolute difference. While this will find the closest match, it involves significant unnecessary computation.

This problem is closely related to the conversion from a residue number system to standard binary. Given two residues, x_1 and x_2 in co-prime bases M_1 and M_2, the goal is to calculate:

$$X = n_1 M_1 + x_1 = n_2 M_2 + x_2 \tag{14.41}$$

where n_1 and n_2 are integers. The solution given by the modified Chinese remainder theorem (Wang, 1998) can be found directly as:

$$X = x_2 + M_2((\lambda(x_1 - x_2)) \bmod M_1) \tag{14.42}$$

where λ is an integer such that:

$$(\lambda M_2) \bmod M_1 = 1 \tag{14.43}$$

Comparing Equations 14.41 and 14.42, it can be seen that the right hand term is effectively calculating the unknown multiplier, n_2, and λ is the multiplicative inverse of M_2. Since M_1 and M_2 are co-prime, such an inverse will always exist.

The difference with range disambiguation is that $M_2\varphi_1$ and $M_1\varphi_2$ are not integers and that the result is not necessarily exactly equal. However, Equation 14.42 can be applied in several steps (Jongenelen *et al.*, 2011). Firstly the difference term is calculated:

$$e = M_1\varphi_2 - M_2\varphi_1 \tag{14.44}$$

This should be an integer, but may not be because of noise. Forcing it to an integer by rounding enables k_1 to be calculated:

$$k_1 = (\lambda \text{round}(e)) \bmod M_1 \tag{14.45}$$

Next, a weighted average of the two estimates is obtained:

$$\begin{aligned} X &= (1-w)(M_2k_1 + M_2\varphi_1) + w(M_1k_2 + M_1\varphi_2) \\ &= M_2k_1 + M_2\varphi_1 + w((M_1\varphi_2 - M_2\varphi_1) + (M_1k_2 - M_2k_1)) \\ &= M_2k_1 + M_2\varphi_1 + w(e - \text{round}(e)) \end{aligned} \tag{14.46}$$

where w is chosen to reduce the variance by taking a weighted average of the two estimates. The third line saves having to explicitly calculate k_2. Finally, the range may be calculated as:

$$d = \frac{c}{2\ \text{gcd}\ (f_1, f_2)} \frac{X}{M_1 M_2} \tag{14.47}$$

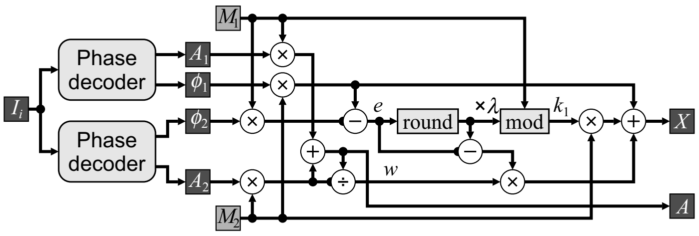


Figure 14.16 Logic for range disambiguation using two modulation frequencies.

To simplify the calculation further, f_1 and f_2 can be chosen so that λ is a power of two (Jongenelen *et al.*, 2010). For example, if $M_2 = M_1 + 1$ or $M_2 = 1$ then $\lambda = 1$. Furthermore, M_1 and M_2 are usually small integers; their size is limited by noise to ensure that Equation 14.45 gives the correct result for most pixels. Therefore, the multiplications in Equation 14.44 can be implemented directly as one or two additions. Alternatively, if the angles are produced by CORDIC units, the scale factor may be built in with the angle, giving $M_2\varphi_1$ and $M_1\varphi_2$ directly.

Finally, the noise in the phase estimate will be approximately inversely proportional to the amplitude (Frank *et al.*, 2009). Therefore, the weight, w, in Equation 14.46 is derived to favour the signal with the stronger amplitude (Jongenelen *et al.*, 2011):

$$w = \frac{M_2A_2}{M_1A_1 + M_2A_2} \tag{14.48}$$

An implementation of the extended range processing is shown in Figure 14.16. In practise, the integration time can be divided between the two modulation frequencies (Jongenelen *et al.*, 2011), with the phase offsets for each image set so that they fall in different frequency bins for the Fourier transform. The system therefore requires two phase decoder units operating in parallel. The output intensity image is formed from the weighted combination of the two component images.

The two time consuming blocks within this circuit are the division to calculate w and the following multiplication to weight the error term. These may be pipelined, if necessary, to achieve the required throughput. Best results (in terms of range accuracy) are obtained if the two frequencies are both high and with approximately the same integration time allocated to each frequency (Jongenelen *et al.*, 2011). Under these circumstances, $w \approx 0.5$ and both the division and multiplication can be removed (multiplication by 0.5 is just a right shift).

This example has shown the application of embedded image processing in enhancing a sensor. The use of an FPGA to perform the image processing has off-loaded the burden from the system using the data. Subsequent processing of the range image may be performed either by the host system or as downstream processing on the FPGA.

14.5 Real-Time Produce Grading

This final application describes a machine vision system for the grading of asparagus by weight. The goal was to estimate the weight of each spear and perform simple quality grading at up to 15 pieces per second. While the original algorithms (Bailey *et al.*, 2004; Bailey, 2011a) were written in C and were able to execute sufficiently fast on a desktop machine for real-time operation, this application would be an ideal candidate for FPGA implementation within a smart camera.

While there is significantly more to a machine vision application than just the image processing, the mechanical handling, lighting and image capture arrangement are beyond the scope of the discussion here.

The focus here will be solely on implementing the image processing algorithm on an FPGA. More details on other aspects of the project are given elsewhere (Bailey *et al.*, 2001; 2004; Mercer *et al.*, 2002; Bailey, 2011a). The basic principle behind estimating the weight of each asparagus spear is to treat each spear as a generalised cylinder and estimate its volume by measuring its projections in two perpendicular views. The weight is then estimated as:

$$Weight \propto \sum_x D_1^2[x] + D_2^2[x] \tag{14.49}$$

where x is the pixel position along the length of the spear, and D_1 and D_2 are the diameters of each view. The constant of proportionality is determined through calibration. The long, thin nature of asparagus enables multiple views to be captured in a single image by using a series of mirrors to divide the field of view.

14.5.1 Software Algorithm

The structure of the software algorithm is shown in Figure 14.17 with images at various stages of the algorithm shown in Figure 14.18. Images are captured asynchronously, when triggered by the cups carrying the asparagus passing over a sensor. Cups are identified by counting them, with cup 0 causing an indicator LED to light up. Since images are buffered with variable latency before processing, it was necessary to use a visual indication to associate each image with a particular cup. The first processing step was to check for the presence of this indicator and for the presence of a spear on the cup. The spear was detected by calculating the mean and standard deviation within the one pixel wide window indicated in Figure 14.18. If no spear was present, processing was aborted for that image.

If a spear was detected, the exposure was checked by building a histogram of the input image and determining the pixel value associated with the 99.6th percentile, essentially using the algorithm described in Section 7.1.3. This was used to automatically adjust the camera exposure by a small increment. The spear was segmented from the background by first performing a contrast enhancement using a lookup table, filtering to remove specular highlights with a 9 × 1 horizontal greyscale morphological closing. The background intensity was estimated by averaging the pixels in each column within

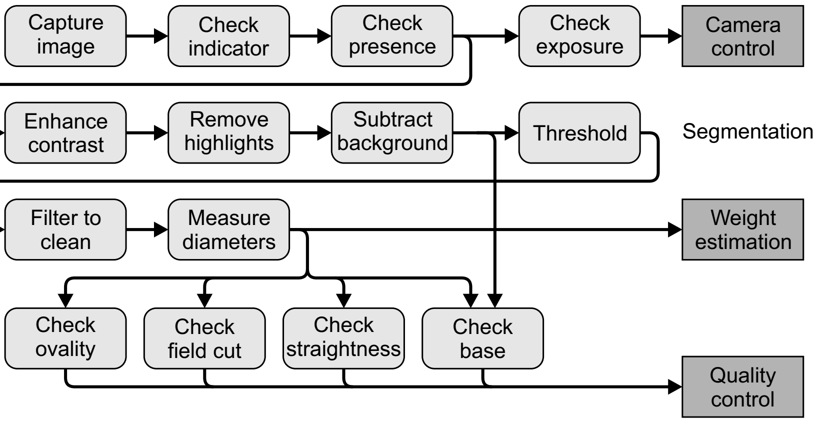


Figure 14.17 Machine vision application for grading asparagus.

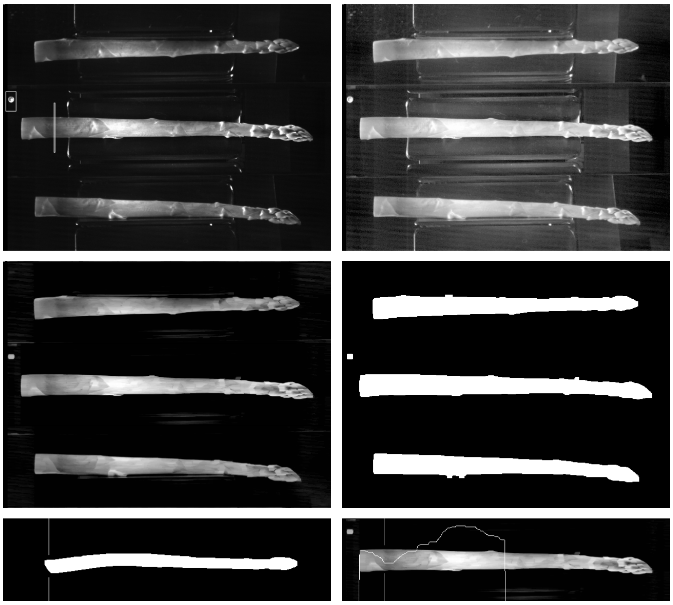


Figure 14.18 Images at various stages through the asparagus grading algorithm. Top left: captured image with indicator region and asparagus detection window overlaid; top right: after contrast enhancement; middle left: after background subtraction; middle right: after thresholding and cleaning; bottom left; side image showing field cut at base; bottom right: detecting white stalk.

each three views within the image. This background was subtracted and the contrast expanded using:

$$\hat{p}_i[x,y] = \begin{cases} 0, & p_i[x,y] \le \mu_i[x] \\ \dfrac{p_i[x,y] - \mu_i[x]}{255 - \mu_i[x]}, & p_i[x,y] > \mu_i[x] \end{cases} \tag{14.50}$$

where p_i is a pixel value in the ith view and μ_i is the corresponding column mean. This image was thresholded with a constant threshold level and then cleaned using a series of binary morphological filters: a vertical 1×9 opening was followed by a horizontal 5×1 opening and a 5×1 closing. The pixels in each column were counted for the two perpendicular views to enable the volume to be estimated using Equation 14.49.

Finally, a quality grading was performed using four simple tests. Firstly, the ovality of the spear was estimated by measuring the diameter at the base of the spear in each of the three views. The ovality ratio was the minimum diameter divided by the maximum diameter, after compensating for the change in magnification in the centre view. Secondly, the angle of the base was checked in each of the three views for the presence of a field cut, with the pixel position required to remove the cut determined. A diagonal field cut is clearly seen in the bottom left panel of Figure 14.18. Thirdly, the straightness of the spear is evaluated by measuring the variance of the spear centreline in the y coordinate. Fourthly, the presence of a white base is detected in the central view. The greyscale image is filtered horizontally with an 11×1 minimum filter before reducing the intensity profile to one dimension by calculating the median of each row. This profile is then checked for a dip, indicating the presence of a white base, as illustrated in the bottom right panel of Figure 14.18.

14.5.2 Hardware Implementation

In mapping the algorithm to hardware, there are a number of optimisations and simplifications that may be made to exploit parallelism. Only the image processing steps for the weight estimation and the filtering required for checking the base of the spear will be implemented in hardware. The output is a set of one-dimensional data, consisting of the edge positions of the spear in each view, along with the intensity profile from the central view. The quality control checks only need to be performed once per frame and are more control orientated, so can be implemented in software using the preprocessed one-dimensional data.

To minimise the memory requirements, as much as possible of the processing will be performed as the data is streamed in from the camera. However, since the processing consists of a mix of row orientated and column orientated processing, a frame buffer will be necessary. The detailed implementation of the algorithm is shown in Figure 14.19 and is described in the following paragraphs.

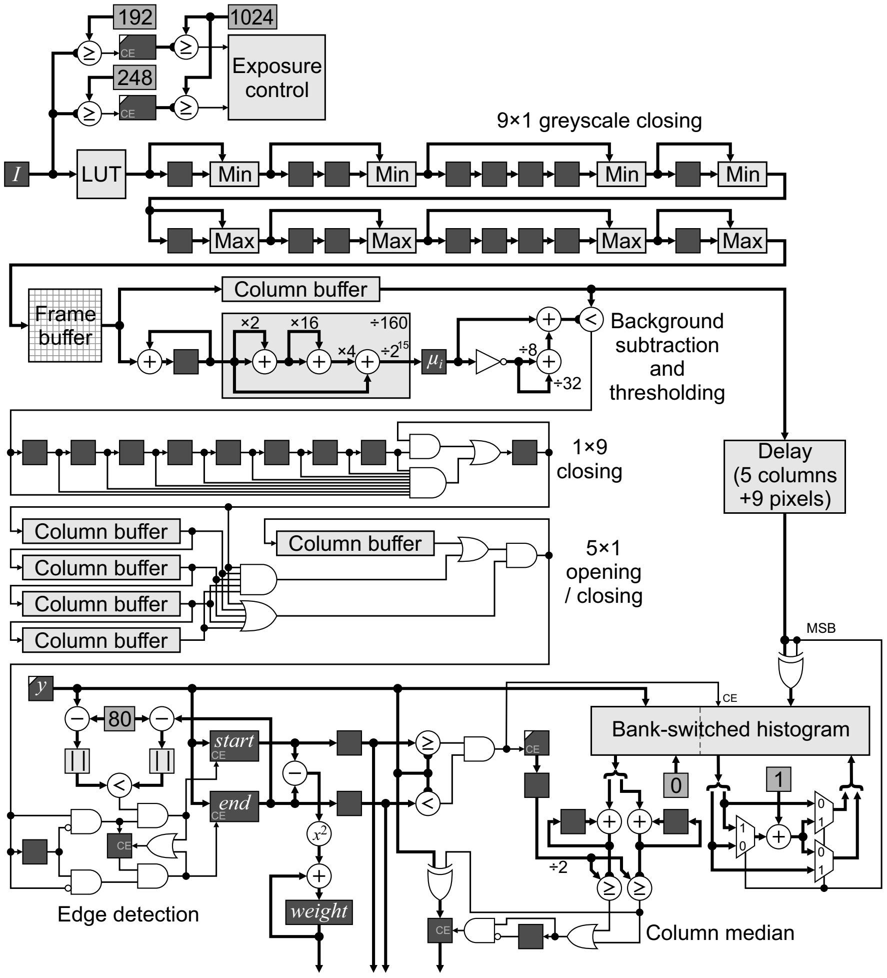


Figure 14.19 Schematic of the FPGA implementation.

Of the initial processing checks, the check for the indicator LED is no longer necessary. Since the images are processed directly as they are streamed from the camera, this signal can be fed directly to the FPGA rather than capturing it via the image. The check for the presence of an asparagus spear will need to be deferred because it relies on the middle view. Processing will need to begin assuming that a spear is present; the results are discarded if it is later not found. For the exposure check, rather than build a histogram then determine the 99.6th percentile and check to see whether it is in range, the processing can be simplified by checking whether the incoming pixels are within range and accumulating them accordingly. In software the threshold levels were 192 and 250. By adjusting the upper threshold to 248 (binary 11111000) the comparison may be simply formed by ANDing the upper five bits. Similarly, comparing with 192 (binary 11000000) reduces to a two-input AND gate. These signals are used to enable the clocks of two counters, to count the pixels above the respective thresholds. The 99.6th percentile is somewhat arbitrary; it is equivalent to two lines of pixels in a 640×480 image. Adjusting this to 1024 simplifies the checking logic; at the end of the frame, if a spear is present, the counts are compared with 1024 (a simple check of the upper bits of the counter) for exposure control.

The contrast enhancement used a fixed lookup table in software. The incoming stream may be processed in exactly the same way, with a lookup table implemented using a block RAM. The 9×1 greyscale closing may be pipelined from the output of the lookup table. The erosion and dilation are implemented using the filter structure from Figure 8.39. Subsequent processing requires a column of data to remove the average. A bank of 160 row buffers could be used, but to fit this on-chip would require a large FPGA. Therefore, the image is streamed to an off-chip frame buffer. A ZBT RAM clocked at twice the pixel clock would avoid the need for a bank switched memory. Alternatively, two pixels could be packed into each memory location, although this would require a row buffer on the input or column buffer on the output.

After each image panel has been loaded into the frame buffer (160 columns) the column processing may begin. This streams the data out of the frame buffer by column for each third of the image. The first step is to calculate the column average, which is then subtracted from each pixel in that column. The pixel data is accumulated and also stored in a column buffer. At the end of each column, the total is divided by 160 and clocked into the mean register, μ_i. This division by a constant may be implemented efficiently with three adders (multiplying by $205/2^{15}$ and eliminating a common subexpression) as shown in Figure 14.19. If only a binary image was required, the normalisation division of Equation 14.50 may be avoided by rearranging the threshold:

$$\begin{gathered} \frac{p_i[x,y]-\mu_i[x]}{255-\mu_i[x]} > Thr \\ p_i[x,y] > \mu_i[x] + (255-\mu_i[x])Thr \end{gathered} \tag{14.51}$$

and since the threshold is constant (40/256) it may be implemented by a single addition ($\times 32/256 + \times 8/256$). The subtraction from 255 is simply a one's complement. Although the greyscale values are used to derive the profile for the quality grading, the pixel values without the background subtracted would be just as good, since the primary purpose of the background subtraction is to segment the spear. The background subtraction and thresholding is effectively implementing an adaptive threshold.

The presence check may be implemented by testing μ_i in the appropriate column of the central panel. If it is above a predefined threshold, then the presence of a spear is indicated (this is not shown in Figure 14.19).

The binary morphological filters for cleaning the image may also be significantly simplified because they are one dimensional. The 1×9 vertical closing is normally implemented as an erosion followed by a dilation. The nine-input AND gate performs the erosion and sets the output register when nine consecutive ones are detected. The output is held at one until the zero propagates through to the end of the chain; this is effectively performing the dilation, using the same shift registers.

Rather than use another frame buffer to switch back to row processing, the 5×1 horizontal cleaning filters are implemented in parallel using column buffers to cache the previous columns along the row.

Rather than perform the opening and closing as two separate operations, both operations are performed in parallel: a sequence of five consecutive ones will switch the output to a one, and a sequence of five zeros will switch the output to a zero. While this filter is not identical to that used in software, it has the same intent of removing sequences of fewer than five ones or zeros.

The next step is to determine the row numbers corresponding to the top and bottom edges of the asparagus spear in each column. A binary transition detector is used to clock the row counter, y, into the *start* and *end* registers. This is complicated slightly by the fact that occasionally spears in adjacent cups are also detected near the edge of the image, as illustrated in Figure 14.20. The region detected near the centre of the panel corresponds to the correct region to be measured. A second detected region replaces the first only if it is nearer to the centre of the panel, that is if:

$$|y - 80| < |end - 80| \tag{14.52}$$

At the end of the column, the edge values are transferred to a pair of output registers. The squared difference is also added to the weight accumulator only for the side images, evaluating Equation 14.49 for the current spear. If a hardware multiplier was not available, the squaring could be performed efficiently using a sequential algorithm since it only needs to be calculated once per column.

If desired, additional accumulators could be added to calculate the column variance for each view, used to evaluate the spear straightness. This would require two accumulators and a counter, and a squarer and a division.

$$straightness = \frac{\sum (start + end)^2}{\sum 1} - \left(\frac{\sum (start + end)}{\sum 1} \right)^2 \tag{14.53}$$

Again, the squarer and division can use sequential algorithm over several clock cycles.

The final processing step is to calculate the median of the pixel values in each column of the central view. The greyscale values are delayed to compensate for the latency of the filters and the edge detection block. The most efficient way of calculating the median of a variable number of pixels is through accumulating a histogram and finding the 50th percentile. The row counter, y, is compared against spear edges and only the pixels within the spear are accumulated by enabling the clock of the histogram accumulator. A counter also counts the pixels which are accumulated. When the column is complete, the histogram is summed to find the 50th percentile. This requires two histogram banks to be used, while the median is being found from one bank, the other bank is accumulating the histogram of the next column.

The problem here is that there are only 160 clock cycles available for each column, requiring two bins to be summed every clock cycle. In the implementation in Figure 14.19, two histogram bins are stored in each memory location. Address n holds the counts for histogram bins n and $255-n$. When accumulating the histogram, if the most significant bit of the pixel value is set, the least significant bits are inverted to get

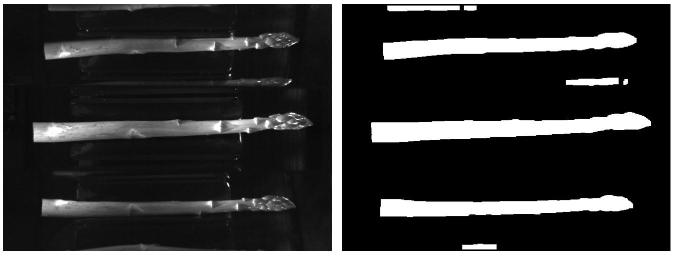


Figure 14.20 Spears visible and detected from adjacent cups.

the address. The most significant bit is also used to control the multiplexers which determine which bin is incremented. When determining the median, the two bins are accumulated separately, using the *y* counter as the address until half the total count is reached. This is effectively scanning from both ends of the histogram in parallel until the 50th percentile is located. If the upper bin is reached first, *y* is inverted to obtain the correct median value. While scanning for the median, the write port is used to reset the histogram count to zero to ready it for the next column.

Not shown in Figure 14.19 is much of the control logic. Various registers need to be reset at the start of the row (or column or image) and others are only latched at the end of a column. In controlling these, allowance must also be made for the latencies of the preceding stages within the pipeline. It is also necessary to account for the horizontal and vertical blanking periods when controlling the various modules and determining the latencies. Also not shown are how the outputs are sent to the serial processor. This depends on whether the processor is implemented on the FPGA or is external. However, because of the relatively low data rate (three numbers per column), this is not particularly difficult.

While quite a complex algorithm, it can still readily be implemented on an FPGA by successively designing each image processing component and piecing these together. A hardware implementation has allowed a number of optimisations that were not available in software, including the simplification of a number of the filters. This example demonstrates that even quite complex machine vision algorithms can be targeted to an FPGA implementation.

14.6 Summary

This chapter has illustrated the design of a number of embedded image processing systems implemented using FPGAs. They have varied from the relatively simple to the complex. There has not been sufficient space available to go through the complete design process, including the false starts and mistakes. What is presented here is an overview of some of the aspects of the completed design. It is hoped that these have demonstrated a range of the techniques covered in this book.

Overall, I trust that this book has provided both a solid foundation and will continue to provide a useful reference in your design. It is now over to you to build on this foundation, in the development of your own embedded image processing applications based on field programmable gate arrays.

References

1394 Trade Association (2004) 1394-based Digital Camera Specification, Vol. Version 1.31, 1394 Trade Association.

Abd Elghany, M.A., Salama, A.E. and Khalil, A.H. (2007) Design and implementation of FPGA-based systolic array for LZ data compression. IEEE International Symposium on Circuits and Systems (ISCAS 2007), New Orleans, Louisiana, USA (27–30 May, 2007), pp. 3691–3695. doi: 10.1109/ISCAS.2007.378644

Abdou, I.E. and Pratt, W.K. (1979) Quantitative design and evaluation of edge enhancement/thresholding edge detectors. *Proceedings of the IEEE*, **67** (5), 753–763. doi: 10.1109/PROC.1979.11325

AbuBaker, A., Qahwaji, R., Ipson, S. and Saleh, M. (2007) One scan connected component labeling technique. IEEE International Conference on Signal Processing and Communications (ICSPC 2007), Dubai, United Arab Emirates (24–27 November, 2007), pp. 1283–1286. doi: 10.1109/ICSPC.2007.4728561

Achronix (2009) Speedster FPGA Family, Vol. DS001 Rev 1.1, Achronix Semiconductor Corporation.

Actel (2008a) IGLOO Low-Power Flash FPGAs, vol. v1.3, Actel Corporation.

Actel (2008b) ProASIC3 Flash Family FPGAs, vol. v1.0, Actel Corporation.

Actel (2009) Axcelerator Family FPGAs, vol. c2.8, Actel Corporation.

Actel (2010) Actel's SmartFusion Intelligent Mixed-Signal FPGAs, Vol. Rev 1, Actel Corporation.

Aeroflex (2008) UT6325 RadTol Eclipse FPGA Data Sheet, Aeroflex Incorporated.

Agostini, L.V., Silva, I.S. and Bampi, S. (2001) Pipelined fast 2D DCT architecture for JPEG image compression. 14th Symposium on Integrated Circuits and Systems Design, Pirenopolis, Brazil (10–15 September, 2001), pp. 226–231. doi: 10.1109/SBCCI.2001.953032

Agostini, L.V., Porto, R.C., Bampi, S. and Silva, I.S. (2005) A FPGA based design of a multiplierless and fully pipelined JPEG compressor. 8th Euromicro Conference on Digital System Design, Porto, Portugal (30 August–3 September, 2005), pp. 210–213. doi: 10.1109/DSD.2005.6

Ahmed, H.M. (1982) Signal processing algorithms and architectures. PhD Thesis, Electrical Engineering Department, Stanford University, California, USA.

Ahmed, E. and Rose, J. (2000) The effect of LUT and cluster size on deep-submicron FPGA performance and density. International Symposium on Field Programmable Gate Arrays, Monterey, California, USA (10–11 February, 2000), pp. 3–12. doi: 10.1145/329166.329171

AIA (2004) Camera Link Specifications, Vol. Version 1.1., Automated Imaging Association.

Aikens, R.S., Agard, D.A. and Sedat, J.W. (1989) Solid-state imagers for microscopy, in *Methods in Cell Biology*, vol. 29 (eds Y.L. Wangand D.L. Taylor), Academic Press, New York, pp. 291–313.

Akeila, H. and Morris, J. (2008) High resolution stereo in real time, in *Robot Vision, Second International Workshop*, Auckland, New Zealand (18–20 February, 2008). Lecture Notes in Computer Science, vol. LNCS 4931, Springer, pp. 72–84. doi: 10.1007/978-3-540-78157-8_6

Al-Hasani, F., Bainbridge-Smith, A. and Hayes, M. (2011) A new sub-expression elimination algorithm using zero dominant representations. 6th International Symposium on Electronic Design, Test and Applications, Queenstown, New Zealand (17–19 January, 2011), pp. 45–50. doi: 10.1109/DELTA.2001.18

Alnuweiri, H.M. and Prasanna, V.K. (1992) Parallel architectures and algorithms for image component labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **14** (10), 1014–1034. doi: 10.1109/34.159904

Alston, I. and Madahar, B. (2002) From C to netlists: hardware engineering for software engineers? *IEE Electronics & Communication Engineering Journal*, **14** (4), 165–173. doi: 10.1049/ecej:20020404

Altera (2002) Excalibur Device Overview, vol. DS-EXCARM-2.0, Altera Corporation.

Altera (2006) Stratix Device Handbook, vol. S5V1-3.4, Altera Corporation.

Altera (2007) Stratix II Device Handbook, vol. SII5V1-4.3, Altera Corporation.

Altera (2008a) Arria GX Device Handbook, vol. AGX5V1-1.3, Altera Corporation.

Altera (2008b) Cyclone Device Handbook, vol. C5V1-2.4, Altera Corporation.

Altera (2008c) Cyclone II Device Handbook, vol. CII5V1-3.3, Altera Corporation.

Altera (2008d) Cyclone III Device Handbook, vol. CIII5V1-2.1, Altera Corporation.

Altera (2008e) Quartus II Development Software Handbook, Version 8.1, vol. 3, Altera Corporation.

Altera (2008f) Stratix III Device Handbook, vol. SIII5V1-1.6, Altera Corporation.

Altera (2009a) Generating Functionally Equivalent FPGAs and ASICs with a Single Set of RTL and Synthesis/Timing Constraints, White Paper, Altera Corporation.

Altera (2009b) Timing Closure Methodology for Advanced FPGA Designs, Application note AN-584-1.0., Altera Corporation.

Altera (2010a) Arria II GX Device Handbook, vol. IIAGX5V1-2.0, Altera Corporation.

Altera (2010b) Cyclone IV Device Handbook, vol. CYIV-5V1-1.1, Altera Corporation.

Altera (2010c) DSP Builder Handbook Volume 1: Introduction to DSP Builder, vol. HB_DSPB_INTRO_1.0, Altera Corporation.

Altera (2010d) Stratix IV Device Handbook, vol. SIV5V1-4.0, Altera Corporation.

Altera (2010e) Stratix V Device Handbook, vol. STX5 1.1, Altera Corporation.

Altera (2010f) Video and Image Processing Suite User Guide, vol. UG-VIPSUITE-10.0, Altera Corporation.

Amdahl, G.M. (1967) Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Spring Joint Computer Conference, Atlantic City, New Jersey, USA, vol. 30 (18–20 April, 1967) pp. 483–485. doi: 10.1145/1465482.1465560

Andraka, R. (1996) Building a high performance bit serial processor in an FPGA. 1996 On-Chip System Design Conference (Design SuperCon), Santa Clara, USA (January, 1996), pp. 5.1–5.21.

Andraka, R. (1998) A survey of CORDIC algorithms for FPGA based computers. ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, Monterey, California, USA (22–25 February, 1998), pp. 191–200. doi: 10.1145/275107.275139

Andrews, D., Niehaus, D. and Ashenden, P. (2004) Programming models for hybrid CPU/FPGA chips. *IEEE Computer*, **37** (1), 118–120. doi: 10.1109/MC.2004.1260732

Angelopoulou, M., Masselos, K., Cheung, P. and Andreopoulos, Y. (2006) A comparison of 2-D discrete wavelet transform computation schedules on FPGAs. International Conference on Field Programmable Technology, Bangkok, Thailand (13–15 December, 2006), pp. 181–188. doi: 10.1109/FPT.2006.270310

Angelopoulou, M.E., Cheung, P.Y.K., Masselos, K. and Andreopoulos, Y. (2008) Implementation and comparison of the 5/3 lifting 2D discrete wavelet transform computation schedules on FPGAs. *Journal of Signal Processing Systems*, **51** (1), 3–21. doi: 10.1007/s11265-007-0139-5

Appiah, K. and Hunter, A. (2005) A single-chip FPGA implementation of real-time adaptive background model. IEEE International Conference on Field-Programmable Technology, Singapore (11–14 December, 2005), pp. 95–102. doi: 10.1109/FPT.2005.1568531

Appiah, K., Hunter, A., Dickenson, P. and Owens, J. (2008) A run-length based connected component algorithm for FPGA implementation. International Conference on Field Programmable Technology, Taipei, Taiwan (7–10 December, 2008), pp. 177–184. doi: 10.1109/FPT.2008.4762381

Arcelli, C. and Di Baja, G.S. (1985) A width independent fast thinning algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **7** (4), 463–474. doi: 10.1109/TPAMI.1985.4767685

Arias-Estrada, M. and Rodríguez-Palacios, E. (2002) An FPGA co-processor for real-time visual tracking, in *International Conference on Field Programmable Logic and Applications*, Montpellier, France (2–4 September, 2002), Lecture Notes in Computer Science, vol. LNCS 2438, Springer, pp. 710–719. doi: 10.1007/3-540-46117-5_73

Arno, S. and Wheeler, F.S. (1993) Signed digit representations of minimal Hamming weight. *IEEE Transactions on Computers*, **42** (8), 1007–1010. doi: 10.1109/12.238495

Arnold, M. and Corporaal, H. (2001) Designing domain-specific processors. Ninth International Symposium on Hardware Software Codesign, Copenhagen, Denmark (25–27 April, 2001), pp. 61–66. doi: 10.1145/371636.371677

Arribas, P.C. and Maciá, F.M.H. (1999) FPGA implementation of a log-polar algorithm for real time applications. Conference on Design of Circuits and Integrated Systems, Mallorca, Spain (16–19 November, 1999), pp. 63–68.

Ashenden, P.J. (2008) *The Designer's Guide to VHDL*, 3rd edn, Morgan Kaufmann Publishers, Burlington, Massachusetts, USA.

Askitis, N. (2009) Fast and compact hash tables for integer keys. Thirty-Second Australasian Computer Science Conference (ACSC 2009), Wellington, New Zealand, vol. CRPIT 91 (January 19–23, 2009), pp. 101–110.

Astola, J., Haavisto, P., Heinonen, P. and Neuvo, Y. (1988) Median type filters for color signals. IEEE International Symposium on Circuits and Systems, Espoo, Finland, vol. 2 (7–9 June, 1988), pp. 1753–1756. doi: 10.1109/ISCAS.1988.15274

Astola, J., Haavisto, P. and Neuvo, Y. (1990) Vector median filters. *Proceedings of the IEEE*, **78** (4), 678–689. doi: 10.1109/5.54807

Ataman, E., Aatre, V.K. and Wong, K.M. (1980) A fast method for real-time median filtering. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **28** (4), 415–420. doi: 10.1109/TASSP.1980.1163426

Athanas, P.M. and Abbot, A.L. (1995) Real-time image processing on a custom computing platform. *IEEE Computer*, **28** (2), 16–25. doi: 10.1109/2.347995

Atmel (2006) AT40KAL Series FPGA, vol. 2818F-FPGA-07/06, Atmel Corporation.

Atmel (2008) AT94KAL Series Field Programmable System Level Integrated Circuit, vol. 1138I-FPLI-1/08, Atmel Corporation.

Auer, S. (1982) Imaging by dust rays: a dust ray camera. *Optica Acta*, **29** (10), 1421–1426. doi: 10.1080/713820766

Avizienis, A. (1961) Signed-digit number representation for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, **EC-10** (3), 389–400. doi: 10.1109/TEC.1961.5219227

Backus, J. (1978) Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, **21** (8), 613–641. doi: 10.1145/359576.359579

Baer, R.L., Holland, W.D., Holm, J.M. and Vora, P.L. (1999) Comparison of primary and complementary color filters for CCD-based digital photography, in *Sensors, Cameras, and Applications for Digital Photography*, San Jose, California, USA (January 27–28, 1999), vol. 3650, SPIE, pp. 16–25. doi: 10.1117/12.342859

Bailey, D.G. (1985) Hardware and software developments for applied digital image processing. PhD Thesis, Department of Electrical and Electronic Engineering, University of Canterbury: Christchurch, New Zealand.

Bailey, D.G. (1988) Machine vision: a multi-disciplinary systems engineering problem, in *Hybrid Image and Signal Processing*, Orlando, Florida, USA (7–8 April, 1988), vol. 939, SPIE, pp. 148–155.

Bailey, D.G. (1990) A rank based edge enhancement filter. 5th New Zealand Image Processing Workshop, Palmerston North, New Zealand (9–10 August, 1990), pp. 42–47.

Bailey, D.G. (1991) Raster based region growing. 6th New Zealand Image Processing Workshop, Lower Hutt, New Zealand (29–30 August, 1991), pp. 21–26.

Bailey, D.G. (1992) Segmentation of touching objects. 7th New Zealand Image Processing Workshop, Christchurch, New Zealand (26–28 August, 1992), pp. 195–200.

Bailey, D.G. (1993) Frequency domain self-filtering for pattern detection. First New Zealand Conference on Image and Vision Computing, Auckland, New Zealand (16–18 August, 1993), pp. 237–243.

Bailey, D.G. (1995) Pixel calibration techniques. New Zealand Image and Vision Computing '95 Workshop, Lincoln, New Zealand (28–29 August, 1995), pp. 37–42.

Bailey, D.G. (1997a) Colour plane synchronisation in colour error diffusion. IEEE International Conference on Image Processing, Santa Barbara, California, USA, vol. 1 (26–29 October, 1997) pp. 818–821. doi: 10.1109/ICIP.1997.648089

Bailey, D.G. (1997b) Detecting regular patterns using frequency domain self-filtering. IEEE International Conference on Image Processing, Santa Barbara, California, USA, vol. 1 (26–29 October, 1997), pp. 440–443. doi: 10.1109/ICIP.1997.647801

Bailey, D.G. (2002) A new approach to lens distortion correction. Image and Vision Computing New Zealand (IVCNZ'02), Auckland, New Zealand (26–28 November, 2002), pp. 59–64.

Bailey, D.G. (2003) Sub-pixel estimation of local extrema. Image and Vision Computing New Zealand (IVCNZ'03), Palmerston North, New Zealand (26–28 November, 2003), pp. 414–419.

Bailey, D.G. (2004) An efficient Euclidean distance transform, in *International Workshop on Combinatorial Image Analysis*, Auckland, New Zealand (1–3 December, 2004), Lecture Notes in Computer Science, vol. LNCS 3322, Springer, pp. 394–408. doi: 10.1007/b103936

Bailey, D.G. (2006) Space efficient division on FPGAs. Electronics New Zealand Conference (ENZCon'06), Christchurch, New Zealand (13–14 November, 2006), pp. 206–211.

Bailey, D.G. (2010a) Chain coding streamed images through crack run-length encoding. Image and Vision Computing New Zealand (IVCNZ 2010), Queenstown, New Zealand (8–9 November, 2010), pp. 155–160.

Bailey, D.G. (2010b) Efficient implementation of greyscale morphological filters. International Conference on Field Programmable Technology (FPT 2010), Beijing, China (8–10 December, 2010), pp. 421–424.

Bailey, D.G. (2011a) Automatic produce grading system, in *Machine Vision Handbook* (ed. B. Batchelor), Springer. doi: 10.1007/978-1-84996-169-1_29

Bailey, D.G. (2011b) Image border management for FPGA based filters. 6th International Symposium on Electronic Design, Test and Applications, Queenstown, New Zealand (17–19 January, 2011), pp. 144–149. doi: 10.1109/DELTA.2011.34

Bailey, D.G. and Bouganis, C.S. (2008) Reconfigurable foveated active vision system. in International Conference on Sensing Technology, Tainan, Taiwan (30 November–3 December, 2008), pp. 162–169. doi: 10.1109/ICSENST.2008.4757093

Bailey, D.G. and Bouganis, C.S. (2009a) Implementation of a foveal vision mapping. International Conference on Field Programmable Technology (FPT'09), Sydney, Australia (9–11 December, 2009), pp. 22–29. doi: 10.1109/FPT.2009.5377646

Bailey, D.G. and Bouganis, C.S. (2009b) Tracking performance of a foveated vision system. International Conference on Autonomous Robots and Agents (ICARA 2009), Wellington, New Zealand (10–12 February, 2009), pp. 414–419. doi: 10.1109/ICARA.2000.4804029

Bailey, D.G. and Bouganis, C.S. (2009c) Vision sensor with an active digital fovea, in *Recent Advances in Sensing Technology*, Lecture Notes in Electrical Engineering, vol. LNEE 49 (eds. S.C. Mukhopadhyay, G. Sen Gupta and R.Y.M. Huang), Springer-Verlag, pp. 91–111. doi: 10.1007/978-3-642-00578-7_6

Bailey, D.G. and Gilman, A. (2007) Bias of higher order predictive interpolation for sub-pixel registration. 6th International Conference on Information, Communications and Signal Processing, Singapore (10–13 December, 2007), pp. 1–5. doi: 10.1109/ICICS.2007.4449703

Bailey, D.G. and Hodgson, R.M. (1985) Range filters: local intensity subrange filters and their properties. *Image and Vision Computing*, **3** (3), 99–110. doi: 10.1016/0262-8856(85)90058-7

Bailey, D.G. and Hodgson, R.M. (1988) VIPS – a digital image processing algorithm development environment. *Image and Vision Computing*, **6** (3), 176–184. doi: 10.1016/0262-8856(88)90024-8

Bailey, D.G. and Johnston, C.T. (2007) Single pass connected components analysis. Image and Vision Computing New Zealand (IVCNZ), Hamilton, New Zealand (5–7 December, 2007), pp. 282–287.

Bailey, D.G. and Johnston, C.T. (2010) Algorithm transformation for FPGA implementation. 5th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2010), Ho Chi Minh City, Vietnam (13–15 January, 2010), pp. 77–81. doi: 10.1109/DELTA.2010.17

Bailey, D.G. and Lill, T.H. (1999) Image registration methods for resolution improvement. Image and Vision Computing New Zealand (IVCNZ), Christchurch, New Zealand (30–31 August, 1999) pp. 91–96.

Bailey, D.G. and Sen Gupta, G. (2010) Automated camera calibration for robot soccer, in *Robotic Soccer* (ed. V. Papic), In-Tech, Vukovar, Croatia, pp. 311–336.

Bailey, D.G. and Shand, R.D. (1996) Determining large scale sandbar behaviour. IEEE International Conference on Image Processing, Lausanne, Switzerland, vol. 2 (16–19 September, 1996), pp. 637–640. doi: 10.1109/ICIP.1996.560958

Bailey, D.G., Hodgson, R.M. and McNeill, S.J. (1984) Local filters in digital image processing. National Electronics Conference (NELCON), Christchurch, New Zealand, vol. 21 (22–24 August, 1984), pp. 95–100.

Bailey, D.G., Mercer, K.A., Plaw, C., Ball, R. and Barraclough, H. (2001) Three dimensional vision for real-time produce grading, in *Machine Vision and Three-Dimensional Imaging Systems for Inspection and Metrology II*, Boston, Massachusetts, USA, (29–30 October, 2001), vol. 4567, SPIE, pp. 171–178. doi: 10.1117/12.455254

Bailey, D.G., Mercer, K.A., Plaw, C., Ball, R. and Barraclough, H. (2004) High speed weight estimation by image analysis. 2004 New Zealand National Conference on Non Destructive Testing, Palmerston North, New Zealand (27–29 July, 2004), pp. 89–96.

Bailey, D.G., Seal, J.R. and Sen Gupta, G. (2005) Nonparametric calibration for catadioptric stereo. Image and Vision Computing New Zealand (IVCNZ'05), Dunedin, New Zealand (28–29 November, 2005), pp. 404–409.

Bailey, D.G., Gribbon, K. and Johnston, C. (2006) GATOS: a windowing operating system for FPGAs. 3rd IEEE International Workshop on Electronic Design, Test, and Applications (DELTA 2006), Kuala Lumpur, Malaysia (17–19 January, 2006), pp. 405–409. doi: 10.1109/DELTA.2006.51

Bailey, D.G., Johnston, C.T. and Ma, N. (2008) Connected components analysis of streamed images. International Conference on Field Programmable Logic and Applications (FPL 2008), Heidelberg, Germany (8–10 September, 2008), pp. 679–682. doi: 10.1109/FPL.2008.4630038

Bajard, J.C., Kla, S. and Muller, J.M. (1994) BKM: a new hardware algorithm for complex elementary functions. *IEEE Transactions on Computers*, **43** (8), 955–963. doi: 10.1109/12.295857

Baker, S. and Matthews, I. (2004) Lucas–Kanade 20 years on: a unifying framework. *International Journal of Computer Vision*, **56** (3), 221–255. doi: 10.1023/B:VISI.0000011205.11775.fd

Balakrishnan, S. and Eddington, C. (2007) Efficient DSP algorithm development for FPGA and ASIC technologies, White paper. Synopsis Incorporated.

Ballard, D.H. (1981) Generalizing the Hough transform to detect arbitrary shapes. *Pattern Recognition*, **13** (2), 111–122. doi: 10.1016/0031-3203(81)90009-1

Banerjee, P., Shenoy, N., Choudhary, A., Hauck, S., Bachmann, C., Haldar, M., Joisha, P., Jones, A., Kanhare, A., Nayak, A., Periyacheri, S., Walkden, M. and Zaretsky, D. (2000) A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, California, USA (17–19 April, 2000) pp. 39–48. doi: 10.1109/FPGA.2000.903391

Banerjee, P., Bagchi, D., Haldar, M., Nayak, A., Kim, V. and Uribe, R. (2003) Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design. 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003), Napa, California, USA (9–11 April, 2003), pp. 263–264. doi: 10.1109/FPGA.2003.1227262

Barnett, V. (1976) Ordering of multivariate data. *Journal of the Royal Statistical Society Series A - Statistics in Society*, **139** (3), 318–355. doi: 10.2307/2344839

Barni, M. (1997) A fast algorithm for 1-norm vector median filtering. *IEEE Transactions on Image Processing*, **6** (10), 1452–1455. doi: 10.1109/83.624972

Barni, M., Buti, F., Bartolini, F. and Cappellini, V. (2000) A quasi-Euclidean norm to speed up vector median filtering. *IEEE Transactions on Image Processing*, **9** (10), 1704–1709. doi: 10.1109/83.869182

Batchelor, B.G. (1979) Streak and spot detection in digital pictures. *Electronics Letters*, **15** (12), 352–353. doi: 10.1049/el:19790250

Batchelor, B.G. (1994) HyperCard lighting advisor. in *Machine Vision Applications, Architectures, and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 180–188. doi: 10.1117/12.188730

Batchelor, B.G. and Whelan, P.F. (1994) Machine vision systems: proverbs, principles, prejudices and priorities, in *Machine Vision Applications, Architectures and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 374–383. doi: 10.1117/12.188748

Batcher, K.E. (1980) Design of a massively parallel processor. *IEEE Transactions on Computers*, **C-29** (9), 836–849. doi: 10.1109/TC.1980.1675684

Bayer, B.E. (1976) *Color imaging array*, United States of America patent 3971065.

Bednar, J.B. and Watt, T.L. (1984) Alpha trimmed means and their relationship to median filters. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **32** (1), 145–153. doi: 10.1109/TASSP.1984.1164279

Beis, J.S. and Lowe, D.G. (1997) Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Juan, Puerto Rico (17–19 June, 1997), pp. 1000–1006. doi: 10.1109/CVPR.1997.609451

Bellas, N., Chai, S.M., Dwyer, M. and Linzmeier, D. (2009) Real-time fisheye lens distortion correction using automatically generated streaming accelerators. 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), Napa, California, USA (5–7 April, 2009), pp. 149–156. doi: 10.1109/FCCM.2009.16

Bellows, P. and Hutchings, B. (1998) JHDL-an HDL for reconfigurable systems. IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, California, USA (15–17 April, 1998), pp. 175–184. doi: 10.1109/FPGA.1998.707895

Bellows, P. and Hutchings, B. (2001) Designing run-time reconfigurable systems with JHDL. *Journal of VLSI Signal Processing*, **28** (1), 29–45. doi: 10.1023/A:1008107104782

Benitez, D. (2002) Performance of remote FPGA-based coprocessors for image-processing applications. Euromicro Symposium on Digital System Design, Dortmund, Germany (4–6 September, 2002) pp. 268–275. doi: 10.1109/DSD.2002.1115378

Benkrid, K. (2000) Design and implementation of a high level FPGA based coprocessor for image and video processing. PhD Thesis, Department of Computer Science, Queen's University of Belfast: Belfast.

Benkrid, A. and Benkrid, K. (2008) HIDE +: a logic based hardware development environment. *Engineering Letters*, **16** (3), 460–468.

Benkrid, K. and Crookes, D. (2003) New bit-level algorithm for general purpose median filtering. *Journal of Electronic Imaging*, **12** (2), 263–269. doi: 10.1117/1.1557153

Benkrid, K., Crookes, D., Smith, J. and Benkrid, A. (2000) High level programming for real time FPGA based video processing. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '00), Istanbul, Turkey, vol. 6 (5–9 June, 2000), pp. 3227–3230. doi: 10.1109/ICASSP.2000.860087

Benkrid, K., Crookes, D. and Benkrid, A. (2002a) Design and implementation of a novel algorithm for general purpose median filtering on FPGAs. IEEE International Symposium on Circuits and Systems (ISCAS 2002), Phoenix, Arizona, vol. 4 (26–29 May, 2002), pp. 425–428. doi: 10.1109/ISCAS.2002.1010482

Benkrid, K., Crookes, D., Benkrid, A. and Belkacemi, S. (2002b) A Prolog-based hardware development environment, in *International Conference on Field Programmable Logic and Applications*, Montpellier, France (2–4 September, 2002), Lecture Notes in Computer Science, vol. LNCS 2438, Springer, pp. 370–380. doi: 10.1007/3-540-46117-5_39

Benkrid, A., Benkrid, K. and Crookes, D. (2003a) A novel FIR filter architecture for efficient signal boundary handling on Xilinx VIRTEX FPGAs. 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003), Napa, California, USA (9–11 April, 2003), pp. 273–275. doi: 10.1109/FPGA.2003.1227267

Benkrid, K., Sukhsawas, S., Crookes, D. and Benkrid, A. (2003b) An FPGA-based image connected component labeller, in *International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisbon, Portugal (1–3 September, 2003), Lecture Notes in Computer Science, vol. LNCS 2778, Springer, pp. 1012–1015. doi: 10.1007/b12007

Benkrid, K., Belkacemi, S. and Benkrid, A. (2006) HIDE: a hardware intelligent description environment. *Microprocessors and Microsystems*, **6** (30), 283–300. doi: 10.1016/j.micpro.2006.02.005

Bergland, G.D. (1969) A guided tour of the fast Fourier transform. *IEEE Spectrum*, **6** (7), 41–52. doi: 10.1109/MSPEC.1969.5213896

Bhasker, J. (1999) *A VHDL primer*, 3rd edn, Prentice-Hall, New Jersey.

Bhasker, J. (2005) *A Verilog HDL Primer*, 3rd edn, Star Galaxy Publishing.

Blodget, B., McMillan, S. and Lysaght, P. (2003) A lightweight approach for embedded reconfiguration of FPGAs. Design, Automation and Test in Europe Conference and Exhibition (DATE'03), Munich, Germany (3–7 March, 2003), pp. 399–400. doi: 10.1109/DATE.2003.10160

Body, N.B. and Bailey, D.G. (1998) Efficient representation and decoding of static Huffman code tables in a very low bit rate environment. IEEE International Conference on Image Processing, Chicago, Illinois, USA, vol. 3 (4–7 October, 1998), pp. 90–94. doi: 10.1109/ICIP.1998.727139

Bohm, A.P.W., Draper, B., Najjar, W., Hammes, J., Rinker, R., Chawathe, M. and Ross, C. (2001) One-step compilation of image processing applications to FPGAs. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), Rohnert Park, California, USA (30 April–2 May, 2001), pp. 209–218. doi: 10.1109/FCCM.2001.32

Bohm, W., Hammes, J., Draper, B., Chawathe, M., Ross, C., Rinker, R. and Najjar, W. (2002) Mapping a single assignment programming language to reconfigurable systems. *Journal of Supercomputing*, **21** (2), 117–130. doi: 10.1023/A:1013623303037

Bollaert, T. (2008) Catapult synthesis: a practical introduction to interactive C synthesis, in *High-Level Synthesis* (eds P. Coussy and A. Morawiec), Springer, The Netherlands, pp. 29–52. doi: 10.1007/978-1-4020-8588-8_3

Booth, A.D. (1951) A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, **4** (2), 236–240. doi: 10.1093/qjmam/4.2.236

Borgefors, G. (1986) Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, **34** (3), 344–371. doi: 10.1016/S0734-189X(86)80047-0

Bouganis, C.S., Constantinides, G.A. and Cheung, P.Y.K. (2005) A novel 2D filter design methodology for heterogeneous devices. 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), Napa, California, USA (18–20 April, 2005), pp. 13–22. doi: 10.1109/FCCM.2005.10

Boullis, N., Mencer, O., Luk, W. and Styles, H. (2001) Pipelined function evaluation on FPGAs. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), Rohnert Park, California, USA (30 April–2 May, 2001), pp. 304–306. doi: 10.1109/FCCM.2001.37

Bovik, A.C., Huang, T.S. and Munson, D.C. (1983) A generalisation of median filtering using linear combinations of order statistics. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **31** (6), 1342–1349. doi: 10.1109/TASSP.1983.1164247

Bowen, O. and Bouganis, C.S. (2008) Real-time image super resolution using an FPGA. International Conference on Field Programmable Logic and Applications (FPL 2008), Heidelberg, Germany (8–10 September, 2008), pp. 89–94. doi: 10.1109/FPL.2008.4629913

Bracewell, R.N. (2000) *The Fourier Transform and its Applications*, 3 edn, McGraw Hill, New York.

Braun, G.J. and Fairchild, M.D. (1999) Image lightness rescaling using sigmoidal contrast enhancement functions. *Journal of Electronic Imaging*, **8** (4), 380–393. doi: 10.1117/1.482706

Brisebarre, N., Muller, J.M. and Tisserand, A. (2006) Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, **32** (2), 236–256. doi: 10.1145/1141885.1141890

Brown, D.C. (1971) Close range camera calibration. *Photogrammetric Engineering*, **37** (8), 855–866.

Brown, S.R. (1987) A note on the description of surface roughness using fractal dimension. *Geophysical Research Letters*, **14** (11), 1095–1098.

Brown, L.G. (1992) A survey of image registration techniques. *ACM Computing Surveys*, **24** (4), 325–376. doi: 10.1145/146370.146374

Brown, C.W. and Shepherd, B.J. (1995) *Graphics File Formats: Reference and Guide*, Manning, Greenwich, Connecticut, USA.

Brumfitt, P.J. (1984) Environments for image processing algorithm development. *Image and Vision Computing*, **2** (4), 198–203. doi: 10.1016/0262-8856(84)90023-4

Buck, J.T. and Lee, E.A. (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. IEEE International Conference on Acoustics, Speech and Signal Processing, Minneapolis, Minnesota, USA, vol. 1 (27–30 April, 1993), pp. 429–432. doi: 10.1109/ICASSP.1993.319147

Buhler, A. (2007) GateOS: a minimalist windowing environment and operating system for FPGAs. Master of Engineering Thesis, Institute of Information Sciences and Technology, Massey University, Palmerston North, New Zealand. hdl: 10179/667

Bunnik, H.M.W., Bailey, D.G. and Mawson, A.J. (2006) Objective colour measurement of tomatoes and limes. Image and Vision Computing New Zealand (IVCNZ'06), Great Barrier Island, New Zealand (27–29 November, 2006), pp. 263–268.

Burdeniuk, A., To, K.N., Lim, C.C. and Liebelt, M.J. (2010) An event-assisted sequencer to accelerate matrix algorithms. 5th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2010), Ho Chi Minh City, Vietnam (13–15 January, 2010), pp. 158–163. doi: 10.1109/DELTA.2010.12

Butt, M.A. and Maragos, P. (1998) Optimal design of chamfer distance transforms. *IEEE Transactions on Image Processing*, **7** (10), 1477–1484. doi: 10.1109/83.718487

Buyukkurt, B., Guo, Z. and Najjar, W.A. (2006) Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs, in *Second International Workshop on Reconfigurable Computing (ARC 2006)*, Delft, The Netherlands, (1–3 March, 2006), Lecture Notes in Computer Science, vol. LNCS 3985, Springer, pp. 401–412. doi: 10.1007/11802839_48

Cady, F.M., Hodgson, R.M., Pairman, D., Rodgers, M.A. and Atkinson, G.J. (1981) Interactive image processing software for a microcomputer. *IEE Proceedings E: Computers and Digital Techniques*, **128** (4), 165–171. doi: 10.1049/ip-e:19810030

Camacho, P., Arrebola, F. and Sadoval, F. (1998) Multiresolution sensors with adaptive structure. 24th Annual Conference of the IEEE Industrial Electronics Society (IECON '98), Aachen, Germany, vol. 2 (31 August–4 September, 1998), pp. 1230–1235. doi: 10.1109/IECON.1998.724279

Canny, J. (1986) A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **8** (6), 679–698. doi: 10.1109/TPAMI.1986.4767851

Carlotto, M.J. (1987) Histogram analysis using a scale-space approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **9** (1), 121–129. doi: 10.1109/TPAMI.1987.4767877

Castleman, K.R. (1979) *Digital Image Processing*, Prentice Hall, Englewood Cliffs, NJ.

Castleman, K.R. (1996) *Digital Image Processing*, 1st edn, Prentice-Hall, New Jersey.

Catmull, E. and Smith, A.R. (1980) 3-D transformations of images in scanline order. *ACM SIGGRAPH Computer Graphics*, **14** (3), 279–285. doi: 10.1145/965105.807505

Catsoulis, J. (2005) *Designing Embedded Hardware*, 2nd edn, O'Reilly.

Cederberg, R.T.L. (1979) Chain link coding and segmentation for raster scan devices. *Computer Graphics and Image Processing*, **10** (3), 224–234. doi: 10.1016/0146-664X(79)90002-9

Celebi, M.E. and Aslandogan, Y.A. (2008) Robust switching vector median filter for impulsive noise removal. *Journal of Electronic Imaging*, **17** (4), 043006-1-9. doi: 10.1117/1.2991415

Chai, D. and Bouzerdoum, A. (2000) A Bayesian approach to skin color classification in YCbCr color space. IEEE Region 10 Conference (TENCON 2000), Kuala Lumpur, Malaysia, vol. 2 (24–27 September, 2000), pp. 421–424. doi: 10.1109/TENCON.2000.888774

Chakrabarti, C. and Wang, L.Y. (1994) Novel sorting network-based architectures for rank order filters. *IEEE Transactions on VLSI Systems*, **2** (4), 502–507. doi: 10.1109/92.335027

Chakravarty, I. (1981) A single-pass, chain generating algorithm for region boundaries. *Computer Graphics and Image Processing*, **15** (2), 182–193. doi: 10.1016/0146-664X(81)90078-2

Chan, F.H.Y., Lam, F.K., Li, H.F. and Liu, J.G. (1996) An all adder systolic structure for fast computation of moments. *Journal of VLSI Signal Processing*, **12** (2), 159–175. doi: 10.1007/BF00924524

Chandrasekaran, R. and Tamir, A. (1989) Open questions concerning Weiszfeld's algorithm for the Fermat–Weber location problem. *Mathematical Programming*, **44** (1–3), 293–295. doi: 10.1007/BF01587094

Chang, L. and Hernández-Palancar, J. (2009) A hardware architecture for SIFT candidate keypoints detection, in *14th Iberoamerican Conference on Pattern Recognition*, Guadalajara, Mexico, Lecture Notes in Computer Science, vol. LNCS 5856, Springer, pp. 95–102. doi: 10.1007/978-3-642-10268-4_11

Chang, S.K., Barnett, M.M., Levialdi, S., Marriott, K., Pfeiffer, J.J. and Tanimoto, S.L. (1999) The future of visual languages. IEEE Symposium on Visual Languages, Tokyo, Japan (13–16 September, 1999), pp. 58–61. doi: 10.1109/VL.1999.795875

Chang, F., Chen, C.J. and Lu, C.J. (2004) A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, **93** (2), 206–220. doi: 10.1016/j.cviu.2003.09.002

Chanussot, J., Paindavoine, M. and Lambert, P. (1999) Real time vector median like filter: FPGA design and application to color image filtering. IEEE International Conference on Image Processing (ICIP'99), Kobe, Japan, vol. 2 (24–28 October, 1999), pp. 414–418. doi: 10.1109/ICIP.1999.822929

Chapweske, A. (2003) The PS/2 Mouse/Keyboard Protocol. Available from http://www.computer-engineering.org [cited 20 July, 2006].

Choo, C. and Verma, P. (2008) A real-time bit-serial rank filter implementation using Xilinx FPGA, in *Real-Time Image Processing 2008*, San Jose, California, USA (28–29 January, 2008) vol. 6811, SPIE, pp. 68110F-1-8. doi: 10.1117/12.765789

Chrysafis, C. and Ortega, A. (2000) Line-based, reduced memory, wavelet image compression. *IEEE Transactions on Image Processing*, **9** (3), 378–389. doi: 10.1109/83.826776

Claus, C., Huitl, R., Rausch, J. and Stechele, W. (2009) Optimizing the SUSAN corner detection algorithm for a high speed FPGA implementation. International Conference on Field Programmable Logic and Applications (FPL), Prague, Czech Republic (31 August–2 September, 2009), pp. 138–145. doi: 10.1109/FPL.2009.5272492

Clist, R.S. and Valkenburg, R.J. (1994) Close coupling of a multiple-instruction-multiple-data (MIMD) system to a MAXbus pipeline: the Kiwivision MTM multitransputer architecture, in *Machine Vision Applications, Architectures and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 106–114. doi: 10.1117/12.188723

Coffman, E.G., Elphick, M. and Shoshani, A. (1971) System deadlocks. *ACM Computing Surveys*, **3** (2), 67–78. doi: 10.1145/356586.356588

Comer, M. and Delp, E. (1999) Morphological operations for color image processing. *Journal of Electronic Imaging*, **8** (3), 279–289. doi: 10.1117/1.482677

Compton, K. and Hauck, S. (2002) Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, **34** (2), 171–210. doi: 10.1145/508352.508353

Conners, R.W. and Harlow, C.A. (1980) A theoretical comparison of texture algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **2** (3), 204–223. doi: 10.1109/TPAMI.1980.4767008

Constantinides, G.A., Cheung, P.Y.K. and Luk, W. (2001) The multiple wordlength paradigm. IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'01), Rohnert Park, California, USA (30 April–2 May, 2001), pp. 51–60. doi: 10.1109/FCCM.2001.46

Cope, B., Cheung, P.Y.K., Luk, W. and Witt, S. (2005) Have GPUs made FPGAs redundant in the field of video processing? IEEE International Conference on Field-Programmable Technology, Singapore (11–14 December, 2005), pp. 111–118. doi: 10.1109/FPT.2005.1568533

Coutinho, J.G.F. and Luk, W. (2003) Source-directed transformations for hardware compilation. IEEE International Conference on Field-Programmable Technology (FPT), Tokyo, Japan (15–17 December, 2003), pp. 278–285. doi: 10.1109/FPT.2003.1275758

Crookes, D. and Benkrid, K. (1999) An FPGA implementation of image component labelling, in *Reconfigurable Technology: FPGAs for Computing and Applications*, Boston, Massachusetts, USA (20–21 September, 1999), vol. 3844, SPIE, pp. 17–23. doi: 10.1117/12.359538

Crookes, D., Morrow, P.J. and McParland, P.J. (1989) An algebra-based language for image processing on transputers. Third International Conference on Image Processing and its Applications, Warwick, UK (18–20 July, 1989), pp. 457–461.

Crookes, D., Alotaibi, K., Bouridane, A., Donachy, P. and Benkrid, A. (1998) An environment for generating FPGA architectures for image algebra-based algorithms. 1998 International Conference on Image Processing (ICIP 98), Chicago, Illinois, USA, vol. 3 (4–7 October, 1998), pp. 990–994. doi: 10.1109/ICIP.1998.999082

Crookes, D., Benkrid, K., Bouridane, A., Alotaibi, K. and Benkrid, A. (2000) Design and implementation of a high level programming environment for FPGA-based image processing. *IEE Proceedings Vision, Image and Signal Processing*, **147** (4), 377–384. doi: 10.1049/ip-vis:20000579

Crow, F.C. (1984) Summed-area tables for texture mapping. SIGGRAPH'84 International Conference on Computer Graphics and Interactive Techniques, Minneapolis, Minnesota, USA (23–27 July, 1984), pp. 207–212. doi: 10.1145/800031.808600

Cucchiara, R., Neri, G. and Piccardi, M. (1998) A real-time hardware implementation of the Hough transform. *Journal of Systems Architecture*, **45** (1), 31–45. doi: 10.1016/S1383-7621(97)00071-4

Cucchiara, R., Onfiani, P., Prati, A. and Scarabottolo, N. (1999) Segmentation of moving objects at frame rate: a dedicated hardware solution. Seventh International Conference on Image Processing and its Applications, Manchester, UK, vol. Conf Publ 465 (13–15 July, 1999), pp. 138–142. doi: 10.1049/cp:19990297

Cuisenaire, O. and Macq, B. (1999) Fast Euclidean distance transformation by propagation using multiple neighbourhoods. *Computer Vision and Image Understanding*, **76** (2), 163–172. doi: 10.1006/cviu.1999.0783

Currie, A.J. (1995). Differentiating apple sports by pollen ultrastructure. Master of Horticultrual Science Thesis, Massey University, Palmerston North, New Zealand.

Cutler, R. and Davis, L. (1998) View-based detection and analysis of periodic motion. Fourteenth International Conference on Pattern Recognition, Brisbane, Australia, vol. 1 (16–20 August, 1998), pp. 495–500. doi: 10.1109/ICPR.1998.711189

Dadda, L. (1965) Some schemes for parallel multipliers. *Alta Frequenza*, **34** (5), 349–356.

Danger, J.L., Guilley, S. and Hoogvorst, P. (2007) Fast true random generator in FPGAs. IEEE Northeast Workshop on Circuits and Systems, Montreal, Canada (5–8 August, 2007), pp. 506–509. doi: 10.1109/NEWCAS.2007.4487970

Danielsson, P.E. (1980) Euclidean distance mapping. *Computer Graphics and Image Processing*, **14** (3), 227–248. doi: 10.1016/0146-664X(80)90054-4

Danielsson, P.E. (1981) Getting the median faster. *Computer Graphics and Image Processing*, **17** (1), 71–78. doi: 10.1016/S0146-664X(81)80010-X

Das Sarma, D. and Matula, D.W. (1995) Faithful bipartite ROM reciprocal tables. 12th Symposium on Computer Arithmetic, Bath, UK (19–21 July, 1995), pp. 17–28. doi: 10.1109/ARITH.1995.465381

DAU (2001) *Systems Engineering Fundamentals*, Defense Acquisition University Press, Fort Belvoir, Virginia, USA.

Daubechies, I. and Sweldens, W. (1998) Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, **4** (3), 247–269. doi: 10.1007/BF02476026

Davies, E.R. (1988) A modified Hough scheme for general circle location. *Pattern Recognition Letters*, **7** (1), 37–43. doi: 10.1016/0167-8655(88)90042-6

Davies, E.R. (1999) High precision discrete model of median shifts. Seventh International Conference on Image Processing and its Applications, Manchester, UK, vol. 1 (13–15 July, 1999), pp. 197–201.

Davies, E.R. (2000) Accuracy of multichannel median filter. *Electronics Letters*, **36** (25), 2068–2069. doi: 10.1049/el:20001465

Davies, E.R. (2008) Stable bi-level and multi-level thresholding of images using a new global transformation. *IET Computer Vision*, **2** (2), 60–74. doi: 10.1049/iet-cvi:20070071

Dawid, H. and Meyr, H. (1992) VLSI implementation of the CORDIC algorithm using redundant arithmetic. IEEE International Symposium on Circuits and Systems (ISCAS '92), San Diego, California, USA, vol. 3 (10–13 May, 1992), pp. 1089–1092. doi: 10.1109/ISCAS.1992.230290

DDWG (1999) Digital Visual Interface (DVI), Revision 1.0., Digital Display Working Group.

de Dinechin, F. and Tisserand, A. (2001) Some improvements on multipartite table methods. IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA (11–13 June, 2001), pp. 128–135. doi: 10.1109/ARITH.2001.930112

de Haan, G. and Bellers, E.B. (1998) Deinterlacing – an overview. *Proceedings of the IEEE*, **86** (9), 1839–1857. doi: 10.1109/5.705528

De Smet, P. (2010) Optimized high speed pixel sorting and its application in watershed based image segmentation. *Pattern Recognition*, **43** (7), 2359–2366. doi: 10.1016/j.patcog.2010.01.014

DeHon, A., Adams, J., DeLorimier, M., Kapre, N., Matsuda, Y., Naeimi, H., Vanier, M., and Wrighton, M. (2004) Design patterns for reconfigurable computing. 12th Annual IEEE Symposium on Field-Programmable Custom

Computing Machines (FCCM 2004), Napa Valley, California, USA (20–23 April, 2004), pp. 13–23. doi: 10.1109/FCCM.2004.29

Dempster, A.G. and Macleod, M.D. (1994) Constant integer multiplication using minimum adders. *IEE Proceedings – Circuits, Devices and Systems*, **141** (5), 407–413. doi: 10.1049/ip-cds:19941191

Denyer, P.B. and Renshaw, D. (1985) *VLSI Signal Processing; a Bit-Serial Approach*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Despain, A.M. (1974) Fourier transform computers using CORDIC iterations. *IEEE Transactions on Computers*, **C-23** (10), 993–1001. doi: 10.1109/T-C.1974.223800

Detrey, J. and de Dinechin, F. (2004) Second order function approximation using a single multiplication on FPGAs, in *14th International Conference on Field Programmable Logic and Application*, Antwerp, Belgium (29 August–1 September, 2004), Lecture Notes in Computer Science, vol. LNCS 3203, Springer, pp. 221–230. doi: 10.1007/b99787

Deutsch, P. (1996) DEFLATE Compressed Data Format Specification Version 1.3, Aladdin Enterprises.

Devernay, F. and Faugeras, O. (2001) Straight lines have to be straight. *Machine Vision and Applications*, **13**, 14–24. doi: 10.1007/PL00013269

Diamantaras, K.I. and Kung, S.Y. (1997) A linear systolic array for real-time morphological image processing. *Journal of VLSI Signal Processing*, **17** (1), 43–55. doi: 10.1023/A:1007996916499

Diaz, J., Ros, E., Mota, S., Carrillo, R. and Agis, R. (2004) Real time optical flow processing system, in *14th International Conference on Field Programmable Logic and Application*, Antwerp, Belgium (29 August–1 September, 2004), Lecture Notes in Computer Science, vol. LNCS 3203, Springer, pp. 617–626. doi: 10.1007/b99787

Dijkstra, E.W. (1959) A note on two problems in connexion with graphs. *Numerische Mathematik*, **1** (1), 269–271. doi: 10.1007/BF01386390

Dike, C. and Burton, E. (1999) Miller and noise effects in a synchronizing flip-flop. *IEEE Journal of Solid-State Circuits*, **34** (6), 849–855. doi: 10.1109/4.766819

Dollas, A., Ermis, I., Koidis, I., Zisis, I. and Kachris, C. (2005) An open TCP/IP core for reconfigurable logic. 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), Napa, California, USA (18–20 April, 2005), pp. 297–298. doi: 10.1109/FCCM.2005.20

Donoho, D.L. (1995) De-noising by soft-thresholding. *IEEE Transactions on Information Theory*, **41** (3), 613–627. doi: 10.1109/18.382009

Donoho, D.L. and Johnstone, I.M. (1994) Threshold selection for wavelet shrinkage of noisy data. 16th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, Baltimore, Maryland, USA, vol. 1 (3–6 November, 1994), pp. A24–A25. doi: 10.1109/IEMBS.1994.412133

Dougherty, E.R. and Laplante, P.A. (1985) *Introduction to Real Time Imaging*, The International Society for Optical Engineering, Washington.

Downton, A. and Crookes, D. (1998) Parallel architectures for image processing. *IEE Electronics & Communication Engineering Journal*, **10** (3), 139–151. doi: 10.1049/ecej:19980307

Doyle, W. (1962) Operations useful for similarity invariant pattern recognition. *Journal of the Association for Computing Machinery*, **9** (2), 259–267. doi: 10.1145/321119.321123

Draper, B., Najjar, W., Bohm, W., Hammes, J., Rinker, B., Ross, C., Chawathe, M. and Bins, J. (2000) Compiling and optimizing image processing algorithms for FPGAs. Fifth IEEE International Workshop on Computer Architectures for Machine Perception, Padova, Italy (11–13 September, 2000), pp. 222–231. doi: 10.1109/CAMP.2000.875981

Draper, B.A., Bohm, A.P.W., Hammes, J., Najjar, W.A., Beveridge, J.R., Ross, C., Chawathe, M., Desai, M. and Bins, J. (2001) Compiling SA-C programs to FPGAs: performance results, in *Second International Workshop on Computer Vision Systems*, Vancouver, Canada (7–8 July, 2001), Lecture Notes in Computer Science, vol. LNCS 2095, Springer, pp. 220–235. doi: 10.1007/3-540-48222-9_15

Draper, B.A., Beveridge, J.R., Bohm, A.P.W., Ross, C. and Chawathe, M. (2002) Implementing image applications on FPGAs. 16th International Conference on Pattern Recognition, Quebec, Canada, vol. 3 (11–15 August, 2002), pp. 265–268. doi: 10.1109/ICPR.2002.1047845

Draper, B.A., Beveridge, J.R., Bohm, A.P.W., Ross, C. and Chawathe, M. (2003) Accelerated image processing on FPGAs. *IEEE Transactions on Image Processing*, **12** (12), 1543–1551. doi: 10.1109/TIP.2003.819226

Dubois, J., Mattavelli, M., Pierrefeu, L. and Miteran, J. (2005) Configurable motion-estimation hardware accelerator module for the MPEG-4 reference hardware description platform. IEEE International Conference on Image Processing (ICIP 2005), Genoa, Italy, vol. 3 (11–14 September, 2005), pp. 1040–1043. doi: 10.1109/ICIP.2005.1530573

Duda, K. (2010) Accurate, guaranteed stable, sliding discrete Fourier transform. *IEEE Signal Processing Magazine*, **27** (10), 124–127. doi: 10.1109/MSP.2010.938088

Duda, R.O. and Hart, P.E. (1972) Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, **15** (1), 11–15. doi: 10.1145/361237.361242

Duff, M.J.B. (2000) Thirty years of parallel image processing, in *Vector and Parallel Processing (VECPAR 2000)*, Porto, Portugal (21–23 June, 2000), Lecture Notes in Computer Science, vol. LNCS 1981, Springer, pp. 419–438. doi: 10.1007/3-540-44942-6_35

Duff, M.J.B., Watson, D.M., Fountain, T.J. and Shaw, G.K. (1973) A cellular logic array for image processing. *Pattern Recognition*, **5** (3), 229–240. doi: 10.1016/0031-3203(73)90045-9

Dumitriu, V., Marcantonio, D. and Kirischian, L. (2009) Run-time component relocation in partially-reconfigurable FPGAs. International Conference on Computational Science and Engineering (CSE'09), Vancouver, Canada, vol. 2 (29–31 August, 2009), pp. 909–914. doi: 10.1109/CSE.2009.493

Duncan, D.B. and Leeson, G. (1999) Cost effective real-time multi-spectral digital video imaging, in *Sensors, Cameras, and Systems for Scientific/Industrial Applications*, San Jose, California, USA (25–26 January, 1999), vol. 3649, SPIE, pp. 100–108. doi: 10.1117/12.347065

Eadie, D., Shevlin, F.O. and Nisbet, A. (2002) Correction of geometric image distortion using FPGAs, in *Opto-Ireland 2002: Optical Metrology, Imaging, and Machine Vision*, Galway, Ireland (5–6 September, 2002), vol. 4877, SPIE, pp. 28–37. doi: 10.1117/12.463765

Economakos, G., Oikonomakos, P., Panagopoulos, I., Poulakis, I. and Papakonstantinou, G. (2001) Behavioral synthesis with SystemC. Design, Automation and Test in Europe, Conference and Exhibition, Munich, Germany (13–16 March, 2001), pp. 21–25. doi: 10.1109/DATE.2001.914995

Edwards, S.A. (2005) The challenges of hardware synthesis from C-like languages. Design, Automation and Test in Europe, Munich, Germany, vol. 1 (7–11 March, 2005), pp. 66–67. doi: 10.1109/DATE.2005.307

Edwards, S.A. (2006) The challenges of synthesizing hardware from C-Like languages. *IEEE Design & Test of Computers*, **23** (5), 375–383. doi: 10.1109/MDT.2006.134

Ehlers, M. (1991) Multisensor image fusion techniques in remote sensing. *ISPRS Journal of Photogrammetry and Remote Sensing*, **46** (1), 19–30. doi: 10.1016/0924-2716(91)90003-E

Elzinga, S., Lin, J. and Singhal, V. (2000) Design tips for HDL implementation of arithmetic functions, Application note XAPP215 (v1.0). Xilinx Inc.

EMVA (2009) GenICam Standard, Vol. 2.0., European Machine Vision Association.

Eschbach, R. and Knox, K.T. (1991) Error-diffusion algorithm with edge enhancement. *Journal of the Optical Society of America A*, **8** (12), 1844–1850. doi: 10.1364/JOSAA.8.001844

Evemy, J.D., Allerton, D.J. and Zaluska, E.J. (1990) Stream processing architecture for real-time implementation of perspective spatial transformations. *IEE Proceedings I: Communications, Speech and Vision*, **137** (3), 123–128.

Ewe, C.T., Cheung, P.Y.K. and Constantinides, G.A. (2004) Dual fixed-point: an efficient alternative to floating-point computation, in *14th International Conference on Field Programmable Logic and Applications*, Antwerp, Belgium (29 August–1 September, 2004), Lecture Notes in Computer Science, vol. LNCS 3203, Springer, pp. 200–208. doi: 10.1007/b99787

Ewe, C.T., Cheung, P.Y.K. and Constantinides, G.A. (2005) Error modelling of dual fixed-point arithmetic and its application in field programmable logic. International Conference on Field Programmable Logic and Applications, Tampere, Finland (24–26 August, 2005), pp. 124–129. doi: 10.1109/FPL.2005.1515710

Fabbri, R., Costa, L.D.F., Torelli, J.C. and Bruno, O.M. (2008) 2D Euclidean distance transform algorithms: A comparative survey, *ACM Computing Surveys*, **40** (1), Article #2 (44 pages). doi: 10.1145/1322432.1322434

Fahmy, S.A., Cheung, P.Y.K. and Luk, W. (2005) Novel FPGA-based implementation of median and weighted median filters for image processing. International Conference on Field Programmable Logic and Applications, Tampere, Finland (24–26 August, 2005), pp. 142–147. doi: 10.1109/FPL.2005.1515713

Fant, K.M. (1986) A nonaliasing, real-time spatial transform technique. *IEEE Computer Graphics and Applications*, **6** (1), 71–80. doi: 10.1109/MCG.1986.276613

Farid, H. and Simoncelli, E.P. (2004) Differentiation of discrete multidimensional signals. *IEEE Transactions on Image Processing*, **13** (4), 496–508. doi: 10.1109/TIP.2004.823819

Feig, E. and Winograd, S. (1992) Fast algorithms for the discrete cosine transform. *IEEE Transactions on Signal Processing*, **40** (9), 2174–2193. doi: 10.1109/78.157218

Ferrari, L.A. and Park, J.H. (1997) An efficient spline basis for multi-dimensional applications: image interpolation. IEEE International Symposium on Circuits and Systems, Hong Kong, vol. 1 (9–12 June, 1997), pp. 757–760. doi: 10.1109/ISCAS.1997.609003

Ferrari, L.A., Park, J.H., Healey, A. and Leeman, S. (1999) Interpolation using a fast spline transform (FST). *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, **46** (8), 891–906. doi: 10.1109/81.780371

Finlayson, G.D., Drew, M.S. and Funt, B.V. (1994) Color constancy: generalized diagonal transforms suffice. *Journal of the Optical Society of America A*, **11** (11), 3011–3019. doi: 10.1364/JOSAA.11.003011

Finlayson, G.D., Hordley, S.D. and Hubel, P.M. (2001) Color by correlation: a simple, unifying framework for color constancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **20** (11), 1209–1221. doi: 10.1109/34.969113

Fitch, J.P., Coyle, E.J. and Gallagher, N.C. (1984) Median filtering by threshold decomposition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **32** (6), 1183–1188. doi: 10.1109/TASSP.1984.1164468

Fitch, J.P., Coyle, E.J. and Gallagher, N.C. (1985) Threshold decomposition of multidimensional ranked order operations. *IEEE Transactions on Circuits and Systems*, **32** (5), 445–450. doi: 10.1109/TCS.1985.1085740

Floyd, R.W. and Steinberg, L. (1975) An adaptive algorithm for spatial grey scale, in *Society for Information Display Symposium Digest of Technical Papers*, Society for Information Display, Campbell, California, pp. 36–37.

Flynn, M. (1972) Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, **C-21** (9), 948–960. doi: 10.1109/TC.1972.5009071

Foley, J.D. and Van Dam, A. (1982) *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts.

Forster, B., Van De Ville, D., Berent, J., Sage, D. and Unser, M. (2004) Complex wavelets for extended depth-of-field: A new method for the fusion of multichannel microscopy images. *Microscopy Research and Technique*, **65** (1–2), 33–42. doi: 10.1002/jemt.20092

Fossum, E.R. (1993) Active pixel sensors: are CCDs dinosaurs? in *Charge-Coupled Devices and Solid State Optical Sensors III*, San Jose, California, USA (2–3 February, 1993), vol. 1900, SPIE, pp. 2–14. doi: 10.1117/12.148585

Frank, M., Plaue, M., Rapp, H., Köthe, U., Jähne, B. and Hamprecht, F.A. (2009) Theoretical and experimental error analysis of continuous-wave time-of-flight range cameras. *Optical Engineering*, **48** (1), 013601- 1-16. doi: 10.1117/1.3070634

Freeman, H. (1961) On the encoding of arbitrary geometric configurations. *IRE Transactions on Electronic Computers*, **10** (2), 260–268. doi: 10.1109/TEC.1961.5219197

Freeman, H. (1974) Computer processing of line-drawing images. *ACM Computing Surveys*, **6** (1), 57–97. doi: 10.1145/356625.356627

Freeman, H. and Davis, L.S. (1977) A corner finding algorithm for chain coded curves. *IEEE Transactions on Computers*, **26** (3), 297–303. doi: 10.1109/TC.1977.1674825

Freeman, H. and Shapira, R. (1975) Determining the minimum area encasing rectangle for an arbitrary enclosed curve. *Communications of the ACM*, **18** (7), 409–413. doi: 10.1145/360881.360919

Frei, W. (1977) Image enhancement by histogram hyperbolisation. *Computer Graphics and Image Processing*, **6** (3), 286–294. doi: 10.1016/S0146-664X(77)80030-0

Friedman, E.G. (2001) Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, **89** (5), 665–692. doi: 10.1109/5.929649

Fryer, J.G., Clarke, T.A. and Chen, J. (1994) Lens distortion for simple C-mount lenses. *International Archives of Photogrammetry and Remote Sensing*, **30** (5), 97–101.

Fu, H., Mencer, O. and Luk, W. (2006) Comparing floating-point and logarithmic number representations for reconfigurable acceleration. International Conference on Field Programmable Technology, Bangkok, Thailand (13–15 December, 2006), pp. 337–340. doi: 10.1109/FPT.2006.270342

Fu, H., Mencer, O. and Luk, W. (2008) Optimizing residue arithmetic on FPGAs. International Conference on Field Programmable Technology, Taipei, Taiwan (7–10 December, 2008), pp. 41–48. doi: 10.1109/FPT.2008.4762364

Funt, B., Barnard, K. and Martin, L. (1998) Is machine colour constancy good enough? in *5th European Conference on Computer Vision (ECCV'98)*, Freiburg, Germany (2–6 June, 1998), Lecture Notes in Computer Science, vol. LNCS 1406, Springer, pp. 445–459. doi: 10.1007/BFb0055655

Galloway, D. (1995) The Transmogrifier C hardware description language and compiler for FPGAs. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, USA (19–21 April, 1995), pp. 136–144. doi: 10.1109/FPGA.1995.477419

Gangadhar, M. and Bhatia, D. (2003) FPGA based EBCOT architecture for JPEG 2000. IEEE International Conference on Field-Programmable Technology (FPT), Tokyo, Japan (15–17 December, 2003), pp. 228–233. doi: 10.1109/FPT.2003.1275752

Garibotto, G. and Lambarelli, L. (1979) Fast online implementation of two-dimensional median filtering. *Electronics Letters*, **15** (1), 24–25. doi: 10.1049/el:19790018

Gasteratos, I., Gasteratos, A. and Andreadis, I. (2006) An algorithm for adaptive mean filtering and its hardware implementation. *Journal of VLSI Signal Processing*, **44** (1–2), 63–78. doi: 10.1007/s11265-006-5920-3

Geer, D. (2005) Chip makers turn to multicore processors. *Computer*, **38** (5), 11–13. doi: 10.1109/MC.2005.160

Genest, G., Chamberlain, R. and Bruce, R. (2007) Programming an FPGA-based super computer using a C-to-VHDL compiler: DIME-C. Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), Edinburgh, UK (5–8 August, 2007), pp. 280–286. doi: 10.1109/AHS.2007.89

Geninatti, S.R., Benitez, J.I.B., Calvio, M.H., Mata, N.G. and Luna, J.G. (2009) FPGA implementation of the generalized Hough transform. International Conference on Reconfigurable Computing and FPGAs (ReConFig '09), Cancun, Quintana Roo, Mexico (9–11 December, 2009), pp. 172–177. doi: 10.1109/ReConFig.2009.78

Gershon, R., Jepson, A.D. and Tsotsos, J.K. (1987) From [R,G,B] to surface reflectance: computing color constant descriptors in images. International Joint Conference on Artificial Intelligence, Milan, Italy, vol. 2 (23–29 August, 1987), pp. 755–758.

Geyer, C., Meingast, M. and Sastry, S. (2005) Geometric models of rolling-shutter cameras. 6th Workshop on Omnidirectional Vision, Camera Networks and Non-classical Cameras (OMNIVIS05), Beijing, China (21 October, 2005), pp. 12–19.

Gigliotti, P. (2004) Implementing Barrel Shifters Using Multipliers, Application note XAPP195 (v1.1), Xilinx Inc.

Gilblom, D.L., Yoo, S.K. and Ventura, P. (2003) Real-time color imaging with a CMOS sensor having stacked photodiodes, in *Ultrahigh- and High-Speed Photography, Photonics, and Videography*, San Diego, California, USA (7 August, 2003), vol. 5210, SPIE, pp. 105–115. doi: 10.1117/12.506206

Gilman, A. (2009) Least-squares optimal interpolation for direct image super-resolution. PhD Thesis, School of Engineering and Advanced Technology, Massey University, Palmerston North. hdl: 10179/893

Gilman, A. and Bailey, D.G. (2007) Noise characteristics of higher order predictive interpolation for sub-pixel registration. IEEE Symposium on Signal Processing and Information Technology, Cairo, Egypt (15–18 December, 2007), pp. 269–274. doi: 10.1109/ISSPIT.2007.4458153

Gilman, A., Bailey, D. and Marsland, S. (2010) Least-squares optimal interpolation for fast image super-resolution. 5th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2010), Ho Chi Minh City, Vietnam (13–15 January, 2010), pp. 29–34. doi: 10.1109/DELTA.2010.59

Ginosar, R. (2003) Fourteen ways to fool your synchronizer. Ninth International Symposium on Asynchronous Circuits and Systems, Vancouver, Canada (12–15 May, 2003), pp. 89–96. doi: 10.1109/ASYNC.2003.1199169

Gionis, A., Indyk, P. and Motwani, R. (1999) Similarity search in high dimensions via hashing. 25th International Conference on Very Large Data Bases, Edinburgh, UK (7–10 September, 1999), pp. 518–529.

Goertzel, G. (1958) An algorithm for the evaluation of finite trigonometric series. *The American Mathematical Monthly*, **65** (1), 34–35.

Goetcherian, V. (1980) From binary to grey tone imaging processing using fuzzy logic concepts. *Pattern Recognition*, **12** (1), 7–15. doi: 10.1016/0031-3203(80)90049-7

Gokhale, M.B. and Stone, J.M. (1998) NAPA C: compiling for a hybrid RISC/FPGA architecture. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, USA (15–17 April, 1998), pp. 126–135. doi: 10.1109/FPGA.1998.707890

Gokhale, M., Stone, J., Arnold, J. and Kalinowski, M. (2000) Stream-oriented FPGA computing in the Streams-C high level language. IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, California, USA (17–19 April, 2000), pp. 49–56. doi: 10.1109/FPGA.2000.903392

Golay, M.J.E. (1969) Hexagonal parallel pattern transformations. *IEEE Transactions on Computers*, **C-18** (8), 733–740.

Goldberg, D. (1991) What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, **23** (1), 6–48. doi: 10.1145/103162.103163

Goldman, D.B. and Chen, J.H. (2005) Vignette and exposure calibration and compensation. Tenth IEEE International Conference on Computer Vision (ICCV 2005), Beijing, China, vol. 1 (17–21 October, 2005), pp. 899–906. doi: 10.1109/ICCV.2005.249

Golin, E.J., Feng, A.C., Huang, L. and Hughes, E. (1993) A visual design environment. 1993 IEEE/ACM International Computer-Aided Design (ICCAD-93), Santa Clara, California, USA (7–11 November, 1993), pp. 364–367. doi: 10.1109/ICCAD.1993.580082

Gonzalez, R.C. and Woods, R.E. (2004) *Digital Image Processing, Using MATLAB*, Pearson Prentice Hall.

Gonzalez, R.C. and Woods, R.E. (2008) *Digital Image Processing*, 3rd edn, Prentice-Hall, New Jersey.

Graham, P.S. (2001) Logical hardware debuggers for FPGA-based systems. PhD Thesis, Department of Electrical and Computer Engineering, Brigham Young University.

Graham, P., Nelson, B. and Hutchings, B. (2001) Instrumenting bitstreams for debugging FPGA circuits. IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'01), Rohnert Park, California, USA (30 April–2 May, 2001), pp. 41–50. doi: 10.1109/FCCM.2001.26

Gregori, S. (2009) Full Linux on FPGA, in *Embedded Computing Conference*, Winterthur, Switzerland, pp 4B2 (26 May, 2009).

Gribbon, K.T. and Bailey, D.G. (2004) A novel approach to real time bilinear interpolation. 2nd IEEE International Workshop on Electronic Design, Test, and Applications (DELTA 2004), Perth, Australia (28–30 January, 2004), pp. 126–131. doi: 10.1109/DELTA.2004.10055

Gribbon, K.T., Johnston, C.T. and Bailey, D.G. (2003) A real-time FPGA implementation of a barrel distortion correction algorithm with bilinear interpolation. Image and Vision Computing New Zealand (IVCNZ'03), Palmerston North, New Zealand (26–28 November, 2003), pp. 408–413.

Gribbon, K.T., Bailey, D.G. and Johnston, C.T. (2004) Colour edge enhancement. Image and Vision Computing New Zealand (IVCNZ'04), Akaroa, New Zealand (21–23 November, 2004), pp. 291–296.

Gribbon, K.T., Johnston, C.T. and Bailey, D.G. (2005) Design patterns for image processing algorithm development on FPGAs. IEEE Region 10 Conference (IEEE Tencon'05), Melbourne, Australia (21–24 November, 2005). doi: 10.1109/TENCON.2005.301109

Gribbon, K.T., Johnston, C.T. and Bailey, D.G. (2006) Using design patterns to overcome image processing constraints on FPGAs. 3rd IEEE International Workshop on Electronic Design, Test, and Applications (DELTA 2006), Kuala Lumpur, Malaysia (17–19 January, 2006), pp. 47–53. doi: 10.1109/DELTA.2006.93

Gribbon, K.T., Bailey, D.G. and Bainbridge-Smith, A. (2007) Development issues in using FPGAs for image processing. Image and Vision Computing New Zealand (IVCNZ), Hamilton, New Zealand (5–7 December, 2007), pp. 217–222.

Guccione, S., Levi, D. and Sundararajan, P. (1999) JBits: Java based interface for reconfigurable computing. Military and Aerospace Applications of Programmable Devices and Technologies International Conference, Laurel, Maryland, USA (28–30 September, 1999).

Gunturk, B.K., Altunbasak, Y. and Mersereau, R.M. (2002) Color plane interpolation using alternating projections. *IEEE Transactions on Image Processing*, **11** (9), 997–1013. doi: 10.1109/TIP.2002.801121

Habegger, A., Stahel, A., Goette, J. and Jacomet, M. (2010) An efficient hardware implementation of a reciprocal unit. 5th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2010), Ho Chi Minh City, Vietnam (13–15 January, 2010), pp. 183–187. doi: 10.1109/DELTA.2010.65

Hadley, J.D. and Hutchings, B.L. (1995) Design methodologies for partially reconfigured systems. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, USA (19–21 April, 1995), pp. 78–84. doi: 10.1109/FPGA.1995.477412

Hagemeyer, J., Kettelhoit, B., Koestner, K. and Porrmann, M. (2007) Design of homogeneous communication infrastructures for partially reconfigurable FPGAs. 2007 International Conference on Engineering of Reconfigurable Systems (ERSA'07), Las Vegas, USA (25–28 June, 2007).

Haldar, M., Nayak, A., Choudhary, A. and Banerjee, P. (2001a) Automated synthesis of pipelined designs on FPGAs for signal and image processing applications described in MATLAB(R). Asia and South Pacific Design Automation Conference (ASP-DAC), Yokohama, Japan (30 January–2 February, 2001), pp. 645–648. doi: 10.1145/370155.370572

Haldar, M., Nayak, A., Shenoy, N., Choudhary, A. and Banerjee, P. (2001b) FPGA hardware synthesis from MATLAB. in Fourteenth International Conference on VLSI Design, Bangalore, India (3–7 January, 2001), pp. 299–304. doi: 10.1109/ICVD.2001.902676

Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.W., Bugnion, E. and Lam, M.S. (1996) Maximizing multiprocessor performance with the SUIF compiler. *Computer*, **29** (12), 84–89. doi: 10.1109/2.546613

Hammes, J.P., Draper, B.A. and Böhm, A.P.W. (1999) Sassy: a language and optimizing compiler for image processing on reconfigurable computing systems, in *First International Conference on Computer Vision Systems (ICVS'99)*, Las Palmas, Spain (13–15 January, 1999), Lecture Notes in Computer Science, vol. LNCS 1542, Springer, pp. 83–97. doi: 10.1007/3-540-49256-9_6

Haralick, R.M. and Shapiro, L.G. (1991) Glossary of computer vision terms. *Pattern Recognition*, **24** (1), 69–93. doi: 10.1016/0031-3203(91)90117-N

Haralick, R.M., Shanmugam, K. and Dinstein, I.H. (1973) Textural features for image classification. *IEEE Transactions on Systems, Man and Cybernetics*, **3** (6), 610–621. doi: 10.1109/TSMC.1973.4309314

Harber, R. and Neudeck, S.B.G. (1985) VLSI implementation of a fast rank order filtering algorithm. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '85), Tampa, Florida, USA (March 26–29, 1985) pp. 1396–1399.

Harris, F.J. (1978) On the use of windows for harmonic analysis with the discrete Fourier transform. *Proceedings of the IEEE* **66** (1), 51–83. doi: 10.1109/PROC.1978.10837

Harriss, T., Walke, R., Kienhuis, B. and Deprettere, E. (2002) Compilation from Matlab to process networks realized in FPGA. *Design Automation for Embedded Systems*, **7** (4), 385–403. doi: 10.1023/A:1020367508848

Hartley, R. (1991) Optimization of canonic signed digit multipliers for filter design. IEEE International Symposium on Circuits and Systems, Singapore, vol. 4 (11–14 June, 1991) pp. 1992–1995. doi: 10.1109/ISCAS.1991.176054

Haselman, M. (2005) A comparison of floating point and logarithmic number systems for FPGAs. Master of Science Thesis, Department of Electrical Engineering, University of Washington.

Haselman, M., Beauchamp, M., Wood, A., Hauck, S., Underwood, K. and Hemmert, K.S. (2005) A comparison of floating point and logarithmic number systems for FPGAs. 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), Napa, California, USA (18–20 April, 2005), pp. 181–190. doi: 10.1109/FCCM.2005.6

Hassler, H. and Takagi, N. (1995) Function evaluation by table look-up and addition. 12th Symposium on Computer Arithmetic, Bath, UK (19–21 July, 1995), pp. 10–16. doi: 10.1109/ARITH.1995.465382

Hauck, S., Hosler, M.M. and Fry, T.W. (2000) High-performance carry chains for FPGAs. *IEEE Transactions on VLSI Systems*, **8** (2), 138–147. doi: 10.1109/92.831434

Hauck, S., Fry, T.W., Hosler, M.M. and Kao, J.P. (2004) The Chimaera reconfigurable functional unit. *IEEE Transactions on VLSI Systems*, **12** (2), 206–217. doi: 10.1109/TVLSI.2003.821545

Haviland, G.L. and Tuszynski, A.A. (1980) A CORDIC arithmetic processor chip. *IEEE Journal of Solid-State Circuits*, **15** (1), 4–15. doi: 10.1109/JSSC.1980.1051332

He, S. and Torkelson, M. (1996) A new approach to pipeline FFT processor. 10th International Parallel Processing Symposium (IPPS'96), Honolulu, Hawaii, USA (15–19 April, 1996), pp. 766–770. doi: 10.1109/IPPS.1996.508145

He, L., Chao, Y. and Suzuki, K. (2007) A linear-time two-scan labelling algorithm. IEEE International Conference on Image Processing (ICIP 2007), San Antonio, Texas, USA, vol. 5 (16–19 September, 2007), pp. 241–244. doi: 10.1109/ICIP.2007.4379810

He, L., Chao, Y. and Suzuki, K. (2008) A run-based two-scan labeling algorithm. *IEEE Transactions on Image Processing*, **17** (5), 749–756. doi: 10.1109/TIP.2008.919369

Hedberg, H., Kristensen, F. and Owall, V. (2007) Implementation of a labeling algorithm based on contour tracing with feature extraction. IEEE International Symposium on Circuits and Systems (ISCAS 2007), New Orleans, Louisiana, USA (27–30 May, 2007), pp. 1101–1104. doi: 10.1109/ISCAS.2007.378202

Heikkila, J. and Silven, O. (1997) A four-step camera calibration procedure with implicit correction. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Juan, Puerto Rico (17–19 June, 1997), pp. 1106–1112. doi: 10.1109/CVPR.1997.609468

Heikkila, J. and Silven, O. (1999) A real-time system for monitoring of cyclists and pedestrians. Second IEEE Workshop on Visual Surveillance, Fort Collins, Colorado, USA (26 June, 1999), pp. 74–81. doi: 10.1109/VS.1999.780271

Herbordt, M.C., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J. and DiSabello, D (2007) Achieving high performance with FPGA-based computing. *IEEE Computer*, **40** (3), 50–57. doi: 10.1109/MC.2007.79

Heygster, G. (1982) Rank filters in digital image processing. *Computer Graphics and Image Processing*, **19** (2), 148–164. doi: 10.1016/0146-664X(82)90105-8

Hezel, S., Kugel, A., Manner, R. and Gavrila, D.M. (2002) FPGA-based template matching using distance transforms. Symposium on Field-Programmable Custom Computing Machines, Napa, California, USA (22–24 April, 2002), pp. 89–97. doi: 10.1109/FPGA.2002.1106664

Hirata, T. (1996) A unified linear-time algorithm for computing distance maps. *Information Processing Letters*, **58** (3), 129–133. doi: 10.1016/0020-0190(96)00049-X

Hoare, C.A.R. (1985) *Communicating Sequential Processes*, Prentice-Hall International, London, UK.

Hodgson, R.M., Bailey, D.G., Naylor, M.J., Ng, A.L.M. and McNeill, S.J. (1985) Properties, implementations and applications of rank filters. *Image and Vision Computing*, **3** (1), 3–14. doi: 10.1016/0262-8856(85)90037-X

Hoffmann, G. (2000) CIE colour space. Available from http://www.fho-emden.de/~hoffmann/ciexyz29082000.pdf [cited 24 January, 2010].

Hoisko, S., Hakkarainen, H., Vihavainen, K. and Isoaho, J. (1996) Specification, hardware implementation and prototyping environment for image processing algorithms. IEEE International Symposium on Circuits and Systems (ISCAS '96), Atlanta, Georgia, USA, vol. 4 (12–15 May, 1996), pp. 834–837. doi: 10.1109/ISCAS.1996.542154

Hollitt, C. (2009) Reduction of computational complexity of Hough transforms using a convolution approach. 24th International Conference Image and Vision Computing New Zealand (IVCNZ '09), Wellington, New Zealand (23–25 November, 2009), pp. 373–378. doi: 10.1109/IVCNZ.2009.5378379

Horta, E.L., Lockwood, J.W., Taylor, D.E. and Parlour, D. (2002) Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. ACM IEEE Design Automation Conference, New Orleans, Louisiana, USA (10–14 June, 2002), pp. 343–348. doi: 10.1145/513918.514007

Hough, P.V.C. (1962) *Method and means for recognizing complex patterns*, United States of America patent 3069654.

Hsia, S.C. (2004) Fast high-quality color-filter-array interpolation method for digital camera systems. *Journal of Electronic Imaging*, **13** (1), 244–247. doi: 10.1117/1.1631443

Huang, C.T. and Mitchell, O.R. (1994) A Euclidean distance transform using grayscale morphology decomposition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **16** (4), 443–448. doi: 10.1109/34.277600

Huang, T.S., Yang, G.Y. and Tang, G.Y. (1979) A fast two dimensional median filtering algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **27** (1), 13–18. doi: 10.1109/TASSP.1979.1163188

Huang, C.T., Tseng, P.C. and Chen, L.G. (2004) Flipping structure: an efficient VLSI architecture for lifting-based discrete wavelet transform. *IEEE Transactions on Signal Processing*, **52** (4), 1080–1089. doi: 10.1109/TSP.2004.823509

Hudson, R.D., Lehn, D.I. and Athanas, P.M. (1998) A run-time reconfigurable engine for image interpolation. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, USA (15–17 April, 1998), pp. 88–95. doi: 10.1109/FPGA.1998.707886

Huffman, D.A. (1952) A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* **40** (9), 1098–1101. doi: 10.1109/JRPROC.1952.273898

Hummel, R.A. (1975) Histogram modification techniques. *Computer Graphics and Image Processing*, **4** (3), 209–224. doi: 10.1016/0146-664X(75)90009-X

Hummel, R.A. (1977) Image enhancement by histogram transformation. *Computer Graphics and Image Processing*, **6** (2), 184–195. doi: 10.1016/S0146-664X(77)80011-7

Hunt, B.R. (1983) Digital image processing. *Advances in Electronics and Electron Physics*, **60**, 161–221. doi: 10.1016/S0065-2539(08)60890-2

Hutchings, B., Bellows, P., Hawkins, J., Hemmert, S., Nelson, B. and Rytting, M. (1999) A CAD suite for high-performance FPGA design. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99), Napa Valley, California, USA (21–23 April, 1999), pp. 12–24. doi: 10.1109/FPGA.1999.803663

Hutton, M., Schleicher, J., Lewis, D., Pedersen, B., Yuan, R., Kaptanoglu, S., Baeckler, G., Ratchev, B., Padalia, K., Bourgeault, M., Lee, A., Kim, H. and Saini, R. (2004) Improving FPGA performance and area using an adaptive logic module, in *14th International Conference on Field Programmable Logic and Applications*, Antwerp, Belgium (29 August–1 September, 2004), Lecture Notes in Computer Science, vol. LNCS 3203, Springer, pp. 135–144. doi: 10.1007/b99787

Hwang, J.N. and Jong, J.M. (1990) Systolic architecture for 2-D rank order filtering. International Conference on Application Specific Array Processors, Princeton, New Jersey, USA (5–7 September, 1990), pp. 90–99. doi: 10.1109/ASAP.1990.145446

Hwang, J., Milne, B., Shirazi, N. and Stroomer, J.D. (2001) System level tools for DSP in FPGAs, in *Field-Programmable Logic and Applications*, Belfast, UK (27–29 August, 2001), Lecture Notes in Computer Science, vol. LNCS 2147, Springer, pp. 534–543. doi: 10.1007/3-540-44687-7_55

IEEE (2004) IEC 62050-2005 First edition 2005-07 IEEE Std 1076 6 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. IEEE/IEC. doi: 10.1109/IEEESTD.2004.94802

IEEE (2005) IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1 Verilog Register Transfer Level Synthesis. IEEE/IEC. doi: 10.1109/IEEESTD.2005.339572

IEEE (2006a) 1364-2005 IEEE Standard for Verilog Hardware Description Language. IEEE. doi: 10.1109/IEEESTD.2006.99495

IEEE (2006b) 1666-2005 IEEE Standard System C Language Reference Manual. IEEE. doi: 10.1109/IEEESTD.2006.99475

IEEE (2008) 754-2008 IEEE Standard for Floating-Point Arithmetic. IEEE. doi: 10.1109/IEEESTD.2008.4610935

IEEE (2009) 1076-2008 IEEE Standard VHDL Language Reference Manual. IEEE. doi: 10.1109/IEEESTD.2009.4772740

Illingworth, J. and Kittler, J. (1987) The adaptive Hough transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **9** (5), 690–698. doi: 10.1109/TPAMI.1987.4767964

Illingworth, J. and Kittler, J. (1988) A survey of the Hough transform. *Computer Vision, Graphics, and Image Processing*, **44** (1), 87–116. doi: 10.1016/S0734-189X(88)80033-1

Irturk, A., Benson, B. and Kastner, R. (2008) Automatic generation of decomposition based matrix inversion architectures. International Conference on Field Programmable Technology, Taipei, Taiwan (7–10 December, 2008), pp. 373–376. doi: 10.1109/FPT.2008.4762421

Iskander, Y., Craven, S., Chandrasekharan, A., Rajagopalan, S., Subbarayan, G. Frangieh, T. and Patterson, C. (2010) Using partial reconfiguration and high-level models to accelerate FPGA design validation. International Conference on Field Programmable Technology (FPT 2010), Beijing, China (8–10 December, 2010), pp. 341–344.

ISO (1992) Digital compression and coding of continuous-tone still images – requirements and guidelines, ISO/IEC 10918-1.

ISO (2000) JPEG 2000 image coding system – part 1: core coding system, ISO/IEC 15444-1:2000.

Isshiki, T. and Dai, W.W.M. (1995) High-level bit-serial datapath synthesis for multi-FPGA systems. International Symposium on Field Programmable Gate Arrays, Monterey, California, USA (12–14 February, 1995), pp. 167–173. doi: 10.1145/201310.201336

Jablonski, M. and Gorgon, M. (2004) Handel-C implementation of classical component labelling algorithm. 2004 Euromicro Symposium on Digital System Design (DSD 2004), Rennes, France (31 August–3 September, 2004), pp. 387–393. doi: 10.1109/DSD.2004.1333301

Jain, A.K. (1989) *Fundamentals of Image Processing*, Prentice Hall, Englewood Cliffs, New Jersey.

Jarvis, R.A. (1983) A perspective on range finding techniques for computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **5** (2), 122–139. doi: 10.1109/TPAMI.1983.4767365

Jiulun, F. and Winxin, X. (1997) Minimum error thresholding: a note. *Pattern Recognition Letters*, **18** (8), 705–709. doi: 10.1016/S0167-8655(97)00059-7

Johnson, D. and Defossez, M. (1999) Programming a Xilinx FPGA in “C”. *Xcell Journal* (34), 26–30.

Johnson, D. (2000) Architectural synthesis from behavioral code to implementation in a Xilinx FPGA. *Xcell Journal* (36), 23–25.

Johnston, C.T. (2009) VERTIPH: a visual environment for real-time image processing on hardware. PhD Thesis, School of Engineering and Advanced Technology, Massey University: Palmerston North. hdl: 10179/1219

Johnston, C.T., Bailey, D.G., Lyons, P. and Gribbon, K.T. (2004) Formalisation of a visual environment for real time image processing in hardware (VERTIPH). Image and Vision Computing New Zealand (IVCNZ’04), Akaroa, New Zealand (21–23 November, 2004), pp. 297–302.

Johnston, C.T., Bailey, D.G. and Gribbon, K.T. (2005a) Optimisation of a colour segmentation and tracking algorithm for real-time FPGA implementation. Image and Vision Computing New Zealand (IVCNZ’05), Dunedin, New Zealand (28–29 November, 2005), pp. 422–427.

Johnston, C.T., Gribbon, K.T. and Bailey, D.G. (2005b) FPGA based remote object tracking for real-time control. International Conference on Sensing Technology, Palmerston North, New Zealand (21–23 November, 2005), pp. 66–71.

Johnston, C.T., Bailey, D.G. and Lyons, P. (2006a) A visual environment for real time image processing in hardware (VERTIPH). *EURASIP Journal on Embedded Systems*, **2006**, no. Article ID 72962. doi: 10.1155/ES/2006/72962

Johnston, C.T., Bailey, D.G. and Lyons, P. (2006b) Towards a visual notation for pipelining in a visual programming language for programming FPGAs. 7th International Conference of the NZ chapter of the ACM’s Special Interest Group on Human-Computer Interaction (CHINZ 2006), Christchurch, New Zealand (6–7 July, 2006), ACM International Conference Proceeding Series, vol. 158, pp. 1–9. doi: 10.1145/1152760.1152761

Johnston, C.T., Lyons, P. and Bailey, D.G. (2008) A visual notation for processor and resource scheduling. IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008), Hong Kong (23–25 January, 2008), pp. 296–301. doi: 10.1109/DELTA.2008.76

Johnston, C.T., Bailey, D.G. and Lyons, P. (2010) Notations for multiphase pipelines. 5th IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2010), Ho Chi Minh City, Vietnam (13–15 January, 2010), pp. 212–216. doi: 10.1109/DELTA.2010.29

Jongenelen, A.P.P., Carnegie, D.A., Dorrington, A.A. and Payne, A.D. (2008) Heterodyne range imaging in real-time. International Conference on Sensing Technology, Tainan, Taiwan (30 November–3 December, 2008), pp. 57–62. doi: 10.1109/ICSENST.2008.4757073

Jongenelen, A.P.P., Bailey, D.G., Payne, A.D., Carnegie, D.A. and Dorrington, A.A. (2010) Efficient FPGA implementation of homodyne-based time of flight range imaging. *Journal of Real-Time Image Processing*. doi: 10.1007/s11554-010-0173-6

Jongenelen, A.P.P., Bailey, D.G., Payne, A.D., Dorrington, A.A. and Carnegie, D.A. (2011) Analysis of errors in ToF range imaging with dual-frequency modulation. *IEEE Transactions on Instrumentation and Measurement*, **60** (5), 1861–1868. doi: 10.1109/TIM.2010.2089190

Justusson, B.I. (1981) Median filtering: statistical properties, in *Two Dimensional Digital Signal Processing II*, Topics in Applied Physics, vol. 43 (ed. T.S. Huang), Springer-Verlag, Berlin, pp. 161–196. doi: 10.1007/BFb0057597

Kakumanua, P., Makrogiannisa, S. and Bourbakis, N. (2007) A survey of skin-color modeling and detection methods. *Pattern Recognition*, **40** (3), 1106–1122. doi: 10.1016/j.patcog.2006.06.010

Kalman, R.E. (1960) A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, **82** (1), 35–45.

Kambe, T., Yamada, A., Nishida, K., Okada, K., Ohnishi, M., Kay, A., Boca, P., Zammit, V. and Nomura, T. (2001) A C-based synthesis system, Bach, and its application. 2001 Asia and South Pacific Design Automation Conference, Yokohama, Japan (2001), pp. 151–155. doi: 10.1145/370155.370309

Kameda, Y. and Minoh, M. (1996) A human motion estimation method using 3-successive video frames. International Conference on Virtual Systems and Multimedia (VSMM'96), Gifu, Japan (18–20 September, 1996), pp. 135–140.

Kamp, W.H.M., McLoughlin, I.V. and Bainbridge-Smith, A. (2006) An exploration of redundant number representations in FPGA. Electronics New Zealand Conference (ENZCon'06), Christchurch, New Zealand (27–29 November, 2006), pp. 63–68.

Kannala, J. and Brandt, S. (2004) A generic camera calibration method for fish-eye lenses. 17th International Conference on Pattern Recognition (ICPR 2004), Surrey, UK, vol. 1 (23–26 August, 2004) pp. 10–13. doi: 10.1109/ICPR.2004.1333993

Kao, J., Narendra, S. and Chandrakasan, A. (2002) Subthreshold leakage modeling and reduction techniques. IEEE/ACM International Conference on Computer Aided Design, San Jose, California, USA (10–14 November, 2002), pp. 141–148. doi: 10.1145/774572.774593

Kapura, J.N., Sahoob, P.K. and Wong, A.K.C. (1985) A new method for gray-level picture thresholding using the entropy of the histogram. *Computer Vision, Graphics, and Image Processing*, **29** (3), 273–285. doi: 10.1016/0734-189X(85)90125-2

Karabernou, S.M., Kessal, L. and Terranti, F. (2005) Real-time FPGA implementation of Hough transform using gradient and CORDIC algorithm. *Image and Vision Computing*, **23** (11), 1009–1017. doi: 10.1016/j.imavis.2005.07.004

Kawada, S. and Maruyama, T. (2007) An approach for applying large filters on large images using FPGA. International Conference on Field Programmable Technology, Kitakyushu, Japan (12–14 December, 2007), pp. 201–208. doi: 10.1109/FPT.2007.4439250

Kehtarnavaz, N. and Gamadia, M. (2006) Real-time image and video processing: from research to reality. Synthesis Lectures on Image, Video and Multimedia Processing. Morgan & Claypool. doi: 10.2200/S00021ED1V01Y200604IVM005

Keys, R.G. (1981) Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **29** (6), 1153–1160. doi: 10.1109/TASSP.1981.1163711

Khan, S., Bailey, D. and Sen Gupta, G. (2009) Simulation of triple buffer scheme (comparison with double buffering scheme). 2nd International Conference on Computer and Electrical Engineering (ICCEE 2009), Dubai, UAE, vol. 2 (28–30 December, 2009), pp. 403–407. doi: 10.1109/ICCEE.2009.226

Khanna, V., Gupta, P. and Hwang, C.J. (2002) Finding connected components in digital images by aggressive reuse of labels. *Image and Vision Computing*, **20** (8), 557–568. doi: 10.1016/S0262-8856(02)00044-6

Kim, K. and Kumar, V.K.P. (1989) Parallel memory systems for image processing. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Diego, California, USA (4–8 June, 1989), pp. 654–659. doi: 10.1109/CVPR.1989.37915

Kim, S.D., Lee, J.H. and Kim, J.K. (1988) A new chain-coding algorithm for binary images using run-length codes. *Computer Vision, Graphics, and Image Processing*, **41** (1), 114–128. doi: 10.1016/0734-189X(88)90121-1

Kimmel, R. (1999) Demosaicing: image reconstruction from color CCD samples. *IEEE Transactions on Image Processing*, **8** (9), 1221–1228. doi: 10.1109/83.784434

Kingsbury, N.G. and Rayner, P.J.W. (1971) Digital filtering using logarithmic arithmetic. *Electronics Letters*, **7** (2), 56–58. doi: 10.1049/el:19710039

Kirsch, R.A. (1998) SEAC and the start of image processing at the National Bureau of Standards. *IEEE Annals of the History of Computing*, **20** (2), 7–13. doi: 10.1109/85.667290

Kittler, J. and Illingworth, J. (1986) Minimum error thresholding. *Pattern Recognition*, **19** (1), 41–47. doi: 10.1016/0031-3203(86)90030-0

Knuth, D.E. (1987) Digital halftones by dot diffusion. *ACM Transactions on Graphics*, **6** (4), 245–273. doi: 10.1145/35039.35040

Koc, C.K. and Johnson, S. (1994) Multiplication of signed-digit numbers. *Electronics Letters*, **30** (11), 840–841. doi: 10.1049/el:19940623

Koester, M., Kalte, H., Porrmann, M. and Rückert, U. (2007) Defragmentation algorithms for partially reconfigurable hardware. Thirteenth International Conference on Very Large Scale Integration of System on Chip, Perth, Australia, *IFIP Advances in Information and Communication Technology,* vol. IFIP 240 (17–19 October, 2007), pp. 41–53. doi: 10.1007/978-0-387-73661-7_4

Kokufuta, K. and Maruyama, T. (2009) Real-time processing of local contrast enhancement on FPGA. International Conference on Field Programmable Logic and Applications (FPL), Prague, Czech Republic (31 August–2 September, 2009), pp. 288–293. doi: 10.1109/FPL.2009.5272284

Koren, I. and Zinaty, O. (1990) Evaluating elementary functions in a numerical coprocessor based on rational approximations. *IEEE Transactions on Computers*, **39** (8), 1030–1037. doi: 10.1109/12.57042

Kota, K. and Cavallaro, J.R. (1993) Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special-purpose processors, *IEEE Transactions on Computers*, vol. 42, no. 7, pp. 769–779. doi:10.1109/12.237718

Kouloheris, J.L. and El Gamal, A. (1991) FPGA performance versus cell granularity. IEEE Custom Integrated Circuits Conference, San Diego, California, USA, pp. 6.2/1-6.2/4 (12–15 May, 1991). doi: 10.1109/CICC.1991.164048

Kovac, M. and Ranganathan, N. (1995) JAGUAR: a fully pipelined VLSI architecture for JPEG image compression standard. *Proceedings of the IEEE*, **83** (2), 247–258. doi: 10.1109/5.364464

Krishnan, G. (2005) Flexibility with EasyPath FPGAs. *Xcell Journal* (55), 96–99.

Kung, S. (1985) VLSI array processors. *IEEE ASSP Magazine*, **2** (3), 4–22.

Kung, H.T. and Leiserson, C.E. (1978) Systolic Arrays (for VLSI). Symposium on Sparse Matrix Computations, Knoxville, Tennessee, USA (2–3 November, 1978), pp. 256–282.

Kung, H.T. and Webb, J.A. (1986) Mapping image processing operations onto a linear systolic machine. *Distributed Computing*, **1** (4), 246–257. doi: 10.1007/BF01660036

Kuon, I. and Rose, J. (2006) Measuring the gap between FPGAs and ASICs. International Symposium on Field Programmable Gate Arrays, Monterey, California, USA (22–24 February, 2006), pp. 21–30. doi: 10.1145/1117201.1117205

Kurak, C.W. (1991) Adaptive histogram equalization: a parallel implementation. Fourth Annual IEEE Symposium Computer-Based Medical Systems, Baltimore, Maryland, USA (12–14 May, 1991), pp. 192–199. doi: 10.1109/CBMS.1991.128965

Lachowicz, S. and Pfleiderer, H.J. (2008) Fast evaluation of the square root and other nonlinear functions in FPGA. IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008), Hong Kong (23–25 January, 2008), pp. 474–477. doi: 10.1109/DELTA.2008.119

Lai, V. and Diessel, O. (2009) ICAP-I: a reusable interface for the internal reconfiguration of Xilinx FPGAs. International Conference on Field Programmable Technology (FPT'09), Sydney, Australia (9–11 December, 2009), pp. 357–360. doi: 10.1109/FPT.2009.5377616

Lamoureux, J. and Wilton, S.J.E. (2006) FPGA clock network architecture: flexibility vs. area and power, in. International Symposium on Field Programmable Gate Arrays, Monterey, California, USA (22–24 February, 2006), pp. 101–108. doi: 10.1145/1117201.1117216

Landsman, R.M., Scott, L.B. and Golay, M.J.E. (1965) *Apparatus for counting bi-nucleate lymphocytes in blood*, United States of America patent 3214574.

Langdon, G.G. (1984) An introduction to arithmetic coding. *IBM Journal of Research and Development*, **28** (2), 135–149. doi: 10.1147/rd.282.0135

Lattice (2007) LatticeXP Family Data Sheet, Vol. DS1001 V05.1., Lattice Semiconductor Corporation.

Lattice (2008a) LatticeECP/EC Family Data Sheet, Vol. DS1000 V02.7., Lattice Semiconductor Corporation.

Lattice (2008b) LatticeECP2/M Family Data Sheet, Vol. DS1006 V03.3., Lattice Semiconductor Corporation.

Lattice (2008c) LatticeSC/M Family Data Sheet, Vol. DS1004 V02.2., Lattice Semiconductor Corporation.

Lattice (2008d) LatticeXP2 Family Data Sheet, Vol. DS1009 V01.6., Lattice Semiconductor Corporation.

Lattice (2010) LatticeECP3 Family Data Sheet, Vol. DS1021 V01.6., Lattice Semiconductor Corporation.

Lavin, C., Padilla, M., Lundrigan, P., Nelson, B. and Hutchings, B. (2010) Rapid prototyping tools for FPGA designs: RapidSmith, in *International Conference on Field Programmable Technology (FPT 2010)*, Beijing, China, pp 353–356 (8–10 December, 2010). doi:10.1109/FPT.2010.5681429

Leavers, V.F. (1993) Which Hough transform? *Computer Vision, Graphics, and Image Processing: Image Understanding*, **58** (2), 250–264. doi: 10.1006/ciun.1993.1041

Lee, D. (1988) Scrambled storage for parallel memory systems. 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, USA (30 May–2 June, 1988), pp. 232–239. doi: 10.1109/ISCA.1988.5233

Lee, P. and Evagelos, A. (2008) An implementation of a multiplierless Hough transform on an FPGA platform using hybrid-log arithmetic, in Real-Time Image Processing 2008, San Jose, California, USA (28–29 January, 2008), vol. 6811, SPIE, pp. 68110G-1-10. doi: 10.1117/12.766459

Lee, H. and Park, R.H. (1990) Comments on An optimal multiple threshold scheme for image segmentation. *IEEE Transactions on Systems, Man and Cybernetics*, **20** (3), 741–742. doi: 10.1109/21.57290

Lee, C.R. and Salcic, Z. (1997) High-performance FPGA-based implementation of Kalman filter. *Microprocessors and Microsystems*, **21** (4), 257–265. doi: 10.1016/S0141-9331(97)00040-9

Lee, D.U., Luk, W., Villasenor, J. and Cheung, P.Y.K. (2003a) Hierarchical segmentation schemes for function evaluation. in IEEE International Conference on Field-Programmable Technology (FPT), Tokyo, Japan (15–17 December, 2003), pp. 92–99. doi: 10.1109/FPT.2003.1275736

Lee, D.U., Luk, W., Villasenor, J. and Cheung, P.Y.K. (2003b) Non-uniform segmentation for hardware function evaluation, in *International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisbon, Portugal (September 1–3, 2003), Lecture Notes in Computer Science, vol. LNCS 2778, Springer, pp. 796–807. doi: 10.1007/b12007

Leeser, M., Miller, S. and Yu, H. (2004) Smart camera based on reconfigurable hardware enables diverse real-time applications. 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004), Napa, California, USA (20–23 April, 2004), pp. 147–155. doi: 10.1109/FCCM.2004.53

Leeser, M., Theiler, J., Estlick, M., and Szymanski, J.J. (2000) Design tradeoffs in a hardware implementation of the K-means clustering algorithm. 2000 IEEE Sensor Array and Multichannel Signal Processing Workshop, Cambridge, Massachusetts (16–17 March, 2000), pp 520–524. doi: 10.1109/SAM.2000.878063

Leiserson, C.E. and Saxe, J.B. (1991) Retiming synchronous circuitry. *Algorithmica*, **6** (1–6), 5–35. doi: 10.1007/BF01759032

Leong, P.H.W. (2008) Recent trends in FPGA architectures and applications. IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008), Hong Kong (23–25 January, 2008), pp. 137–141. doi: 10.1109/DELTA.2008.14

Levine, B., Natarajan, S., Tan, C., Newport, D. and Bouldin, D. (1999) Mapping of an automated target recognition application from a graphical software environment to FPGA-based reconfigurable hardware. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99), Napa Valley, California, USA (21–23 April, 1999), pp. 292–293. doi: 10.1109/FPGA.1999.803702

Lewis, D.M. (1990) An architecture for addition and subtraction of long word length numbers in the logarithmic number system. *IEEE Transactions on Computers*, **39** (11), 1325–1336. doi: 10.1109/12.61042

Li, Y. and Chu, W. (1996) A new non-restoring square root algorithm and its VLSI implementations. IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD '96 Austin, Texas, USA (7–9 October, 1996), pp. 538–544. doi: 10.1109/ICCD.1996.563604

Li, M. and Lavest, J.M. (1996) Some aspects of zoom lens camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **18** (11), 1105–1110. doi: 10.1109/34.544080

Liang, C.K., Chang, L.W. and Chen, H.H. (2008) Analysis and compensation of rolling shutter effect. *IEEE Transactions on Image Processing*, **17** (8), 1323–1330. doi: 10.1109/TIP.2008.925384

Liao, H., Mandal, M.K. and Cockburn, B.F. (2004) Efficient architectures for 1-D and 2-D lifting-based wavelet transforms. *IEEE Transactions on Signal Processing*, **52** (5), 1315–1326. doi: 10.1109/TSP.2004.826175

Lin, K.C. (2005) On improvement of the computation speed of Otsu's image thresholding. *Journal of Electronic Imaging*, **14** (2), 023011- 1-12. doi: 10.1117/1.1902997

Lindeberg, T. (1994) Scale-space theory: a basic tool for analysing structures at different scales. *Journal of Applied Statistics*, **21** (2), 225–270. doi: 10.1080/757582976

Litwiller, D. (2005) CMOS vs. CCD: maturing technologies, maturing markets. *Photonics Spectra*, **2005** (8), 54–59.

Liu, Y., Bouganis, C.S. and Cheung, P.Y.K. (2007) Efficient mapping of a Kalman filter into an FPGA using Taylor expansion. International Conference on Field Programmable Logic and Applications (FPL 2007), Amsterdam, The Netherlands (27–29 August, 2007), pp. 345–350. doi: 10.1109/FPL.2007.4380670

Liu, Q., Constantinides, G.A., Masselos, K. and Cheung, P.Y.K. (2008) Combining data reuse exploitation with data-level parallelization for FPGA targeted hardware compilation: a geometric programming framework. International Conference on Field Programmable Logic and Applications (FPL 2008), Heidelberg, Germany (8–10 September, 2008), pp. 179–184. doi: 10.1109/FPL.2008.4629928

Loeffler, C., Ligtenberg, A. and Moschytz, G.S. (1989) Practical fast 1-D DCT algorithms with 11 multiplications. 1989 International Conference on Acoustics, Speech, and Signal Processing (ICASSP-89), Glasgow, UK, vol. 2 (23–26 May, 1989) pp. 988–991. doi: 10.1109/ICASSP.1989.266596

Longfield, S. and Chang, M.L. (2009) A parameterized stereo vision core for FPGAs. 17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), Napa, California, USA (5–7 April, 2009) pp. 263–265. doi: 10.1109/FCCM.2009.32

Lowe, D.G. (1999) Object recognition from local scale-invariant features. Seventh IEEE International Conference on Computer Vision, Corfu, Greece, vol. 2 (20–27 September, 1999), pp. 1150–1157. doi: 10.1109/ICCV.1999.790410

Lowe, D.G. (2004) Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, **60** (2), 91–110. doi: 10.1023/B:VISI.0000029664.99615.94

Lu, W.S., Wang, H.P. and Antoniou, A. (1990) Design of two-dimensional FIR digital filters by using the singular-value decomposition. *IEEE Transactions on Circuits and Systems*, **37** (1), 35–46. doi: 10.1109/31.45689

Lubbers, E. and Platzner, M. (2007) ReconOS: an RTOS supporting hard- and software threads. International Conference on Field Programmable Logic and Applications (FPL 2007), Amsterdam, The Netherlands (27–29 August, 2007), pp. 441–446. doi: 10.1109/FPL.2007.4380686

Lucas, B.D. and Kanade, T. (1981) An iterative image registration technique with an application to stereo vision. 7th International Joint Conference on Artificial Intelligence, Vancouver, British Columbia, Canada (24–28 August, 1981), pp. 674–679.

Lucke, L.E. and Parhi, K.K. (1992) Parallel structures for rank order and stack filters. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-92), San Francisco, California, USA, vol. 5 (23–26 March, 1992) pp. 645–648. doi: 10.1109/ICASSP.1992.226513

Lumia, R., Shapiro, L. and Zuniga, O. (1983) A new connected components algorithm for virtual memory computers. *Computer Vision, Graphics and Image Processing*, **22** (2), 287–300. doi: 10.1016/0734-189X(83)90071-3

Lyon, R.F. and Hubel, P.M. (2002) Eyeing the camera: into the next century. Tenth Color Imaging Conference: Color Science and Engineering Systems, Technologies, Applications, Scottsdale, Arizona, USA (November 12–15, 2002), pp. 349–355.

Ma, N., Bailey, D. and Johnston, C. (2008) Optimised single pass connected components analysis. International Conference on Field Programmable Technology, Taipei, Taiwan (8–10 December, 2008), pp. 185–192. doi: 10.1109/FPT.2008.4762382

Mahalanobis, P.C. (1936) On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India*, **2** (1), 49–55.

Mandler, E. and Oberlander, M.F. (1990) One-pass encoding of connected components in multivalued images. 10th International Conference on Pattern Recognition, Atlantic City, New Jersey, USA, vol. 2 (16–21 June, 1990), pp. 64–69. doi: 10.1109/ICPR.1990.119331

Manocha, D. (2005) General-purpose computations using graphics processors. *Computer*, **38** (8), 85–88. doi: 10.1109/MC.2005.261

Marr, D. and Hildreth, E. (1980) Theory of edge detection. *Proceedings of the Royal Society of London, Series B, Biological Sciences*, **207** (1167), 187–217. doi: 10.1098/rspb.1980.0020

Martin, G. and Smith, G. (2009) High-level synthesis: past, present, and future. *IEEE Design & Test of Computers*, **26** (4), 18–25. doi: 10.1109/MDT.2009.83

Martinez, J. and Altamirano, L. (2006) FPGA-based pipeline architecture to transform Cartesian images into foveal images by using a new foveation approach. IEEE International Conference on Reconfigurable Computing and FPGA's, San Luis Potosi, Mexico (20–22 September, 2006), pp. 227–236. doi: 10.1109/RECONF.2006.307774

Martinez-Peiro, M., Boemo, E.I. and Wanhammar, L. (2002) Design of high-speed multiplierless filters using a nonrecursive signed common subexpression algorithm. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, **49** (3), 196–203. doi: 10.1109/TCSII.2002.1013866

Maruyama, T. (2004) Real-time computation of the generalized Hough transform, in *14th International Conference on Field Programmable Logic and Application*, Antwerp, Belgium (29 August–1 September, 2004), Lecture Notes in Computer Science, vol. LNCS 3203, Springer, pp. 980–985. doi: 10.1007/b99787

Masrani, D.K. and MacLean, W.J. (2006) A real-time large disparity range stereo-system using FPGAs, in *7th Asian Conference on Computer Vision (ACCV 2006)*, Hyderabad, India (13–16 January, 2006), Lecture Notes in Computer Science, vol. LNCS 3852, Springer, pp. 42–51. doi: 10.1007/11612704_5

MathStar (2007) ArrixTM Family FPOATM Architecture Guide, vol. V1.02., MathStar Incorporated.

MathWorks (2010) *Simulink HDL Coder 2 User's Guide*. The MathWorks Inc.

Maurer, C.R., Qi, R. and Raghavan, V. (2003) A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **25** (2), 265–270. doi: 10.1109/TPAMI.2003.1177156

Mayasandra, K., Salehi, S., Wang, W. and Ladak, H.M. (2005) A distributed arithmetic hardware architecture for real-time Hough-transform-based segmentation. *Canadian Journal of Electrical and Computer Engineering*, **30** (4), 201–205. doi: 10.1109/CJECE.2005.1541752

McCollum, A.J., Bowman, C.C., Daniels, P.A. and Batchelor, B.G. (1988) A histogram modification unit for real-time image enhancement. *Computer Vision, Graphics and Image Processing*, **42** (3), 387–398. doi: 10.1016/S0734-189X(88)80047-1

McDonnell, M.J. (1981) Box filtering techniques. *Computer Graphics and Image Processing*, **17** (1), 65–70. doi: 10.1016/S0146-664X(81)80009-3

McFarlane, M.D. (1972) Digital pictures fifty years ago. *Proceedings of the IEEE*, **60** (7), 768–770.

McIvor, A., Zang, Q. and Klette, R. (2001) The background subtraction problem for video surveillance systems, in *International Workshop on Robot Vision (RobVis 2001)*, Auckland, New Zealand (16–18 February, 2001), Lecture Notes in Computer Science, vol. LNCS 1998, Springer, pp. 176–183. doi: 10.1007/3-540-44690-7_22

McLaughlin, J. (2000) The development of a Java image processing framework. Master of Technology Thesis, Institute of Information Sciences and Technology, Massey University: Palmerston North, New Zealand.

Mencer, O. (2006) ASC: a stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **25** (9), 1603–1617. doi: 10.1109/TCAD.2005.857377

Mencer, O. and Luk, W. (2004) Parameterized high throughput function evaluation for FPGAs. *Journal of VLSI Signal Processing*, **36** (1), 17–25. doi: 10.1023/B:VLSI.0000008067.31043.35

Mencer, O., Pearce, D.J., Howes, L.W. and Luk, W. (2003) Design space exploration with A Stream Compiler. 2003 IEEE International Conference on Field-Programmable Technology (FPT 2003), Tokyo, Japan (15–17 December, 2003), pp. 270–277. doi: 10.1109/FPT.2003.1275757

Mentor (2010a) DK User Manual, Vol. Release v5.3_2., Mentor Graphics Corporation.

Mentor (2010b) Handel-C Language Reference Manual, Vol. Release v5.3_2., Mentor Graphics Corporation.

Mentor (2010c) PixelStreams User Manual, Vol. Release v5.3_2., Mentor Graphics Corporation.

Mercer, K., Bailey, D.G., Plaw, C., Ball, R. and Barraclough, H. (2002) Intelligent actuators for a high speed grading system. Ninth Electronics New Zealand Conference (ENZCon'02), Dunedin, New Zealand (14–15 November, 2002), pp. 61–65.

Mesquita, D., Moraes, F., Palma, J., Möller, L. and Calazans, N. (2003) Remote and partial reconfiguration of FPGAs: tools and trends. International Parallel and Distributed Processing Symposium, Nice, France (22–26 April, 2003), p. 8. doi: 10.1109/IPDPS.2003.1213326

Micron (2007) Micron NAND flash controller via Xilinx Spartan-3 FPGA, Vol. TN-29-06 Revision B., Micron Technologies Inc.

Micron (2010) NAND Flash 101: an introduction to NAND flash and how to design it into your next product, Vol. TN-29-19 Revision B., Micron Technology, Inc.

Mignolet, J.Y., Nollet, V., Coene, P., Verkest, D., Vernalde, S. and Lauwereins, R. (2003) Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. Design, Automation and Test in Europe (DATE'03), Munich, Germany (3–7 March, 2003), pp. 986–991. doi: 10.1109/DATE.2003.10020

Millane, R.P., Alzaidi, S. and Hsiao, W.H. (2003) Scaling and power spectra of natural images. Image and Vision Computing New Zealand (IVCNZ'03), Palmerston North, New Zealand (26–28 November, 2003), pp. 148–153.

Miller-Karlow, D.L. and Golin, E.J. (1992) vVHDL: a visual hardware description language. 1992 IEEE Workshop on Visual Languages, Seattle, Washington, USA (15–18 September, 1992), pp. 133–139. doi: 10.1109/WVL.1992.275773

Mitra, S.K. (1998) *Digital signal processing: a computer-based approach*, McGraw-Hill, Singapore.

Miyamori, T. and Olukotun, U. (1998) A quantitative analysis of reconfigurable coprocessors for multimedia applications. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, USA (15–17 April, 1998), pp. 2–11. doi: 10.1109/FPGA.1998.707876

Mohl, S. (2006) The Mitrion-C programming language, Vol. 1.3.0-001., Mitrionics.

Morris, J., Jawed, K., Gimel'farb, G. and Khan, T. (2009) Breaking the 'ton': achieving 1% depth accuracy from stereo in real time. 24th International Conference Image and Vision Computing New Zealand (IVCNZ '09), Wellington, New Zealand (23–25 November, 2009), pp. 142–147. doi: 10.1109/IVCNZ.2009.5378423

Mosqueron, R., Dubois, J. and Paindavoine, M. (2007) High-speed smart camera with high resolution. *EURASIP Journal on Embedded Systems*, **2007**, no. Article ID 24163. doi: 10.1155/2007/24163

Muller, J.M. (1985) Discrete basis and computation of elementary functions. *IEEE Transactions on Computers*, **C-34** (9), 857–862. doi: 10.1109/TC.1985.1676643

Nagata, N. and Maruyama, T. (2004) Real-time detection of line segments using the line Hough transform. IEEE International Conference on Field-Programmable Technology, Brisbane, Australia (6–8 December, 2004), pp. 89–96. doi: 10.1109/FPT.2004.1393255

Nagayama, S., Sasao, T. and Butler, J.T. (2008) Numerical function generators using bilinear interpolation. International Conference on Field Programmable Logic and Applications (FPL 2008), Heidelberg, Germany (8–10 September, 2008), pp. 463–466. doi: 10.1109/FPL.2008.4629984

Najjar, W.A., Böhm, W., Draper, B.A., Hammes, J., Rinker, R., Beveridge, J.R., Chawathe, M. and Ross, C. (2003) High-level language abstraction for reconfigurable computing. *IEEE Computer*, **36** (8), 63–69. doi: 10.1109/MC.2003.1220583

Nakagawa, Y. and Rosenfeld, A. (1978) A note on the use of local MIN and MAX operations in digital picture processing. *IEEE Transactions on Systems, Man and Cybernetics*, **8** (8), 632–635. doi: 10.1109/TSMC.1978.4310040

Narendra, P.M. (1981) A separable median filter for image noise smoothing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **3** (1), 20–29. doi: 10.1109/TPAMI.1981.4767047

Nash, J.G. (2005) Systolic architecture for computing the discrete Fourier transform on FPGAs. 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), Napa, California, USA (18–20 April, 2005), pp. 305–306. doi: 10.1109/FCCM.2005.60

Nassi, I. and Shneiderman, B. (1973) Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, **8** (8), 12–26. doi: 10.1145/953349.953350

National Instruments (2005) Creating Custom Hardware with LabVIEW: NI LabVIEW FPGA Module, National Instruments.

Nayak, A., Haldar, M., Choudhary, A. and Banerjee, P. (2001) Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. Design, Automation and Test in Europe, Munich, Germany (13–16 April, 2001), pp. 722–728. doi: 10.1109/DATE.2001.915108

NEC (2009) Gate Arrays and Embedded Arrays, vol. A18258EJ7V0PF, NEC Electronics Corporation.

Ngan, P.M. (1992) The development of a visual language for image processing applications. PhD Thesis, Computer Science, Massey University: Palmerston North, New Zealand.

Ngo, H.T. and Asari, V.K. (2005) A pipelined architecture for real-time correction of barrel distortion in wide-angle camera images. *IEEE Transactions on Circuits and Systems for Video Technology*, **15** (3), 436–444. doi: 10.1109/TCSVT.2004.842609

Nicklin, S.P., Fisher, R.D. and Middleton, R.H. (2007) Rolling shutter image compensation, in *RoboCup 2006: Robot Soccer World Cup X, Lecture Notes in Artificial Intelligence*, vol. LNAI 4434, Springer, pp. 402–409. doi: 10.1007/978-3-540-74024-7_39

Nicol, C.J. (1995) A systolic approach for real time connected component labeling. *Computer Vision and Image Understanding*, **61** (1), 17–31. doi: 10.1006/cviu.1995.1002

Nielsen, A.M. and Muller, J.M. (1996) On-line algorithms for computing exponentials and logarithms, in *Second International Euro-Par Conference*, Lyon, France (26–29 August, 1996), Lecture Notes in Computer Science, vol. LNCS 1124, Springer, pp. 165–174. doi: 10.1007/BFb0024699

Nilsson, M., Dahl, M. and Claesson, I. (2005a) Gray-scale image enhancement using the SMQT. IEEE International Conference on Image Processing (ICIP '05), Genoa, Italy, vol. 1 (11–14 September, 2005), pp. 933–936. doi: 10.1109/ICIP.2005.1529905

Nilsson, M., Dahl, M. and Claesson, I. (2005b) The successive mean quantization transform. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '05), Philadelphia, Pennsylvania, USA, vol. 4 (18–23 March, 2005), pp. 429–432. doi: 10.1109/ICASSP.2005.1416037

Nilsson, M., Sattar, F., Chng, H.K. and Claesson, I. (2005c) Automatic enhancement and subjective evaluation of dental X-ray images using the SMQT. 5th International Conference on Information, Communications and Signal Processing, Bangkok, Thailand (6–9 December, 2005), pp. 1448–1451. doi: 10.1109/ICICS.2005.1689298

Noble, J.A. (1988) Finding corners. *Image and Vision Computing*, **6** (2), 121–128. doi: 10.1016/0262-8856(88)90007-8

Nodes, T.A. and Gallagher, N.C. (1982) Median filters: some modifications and their properties. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **30** (5), 739–746. doi: 10.1109/TASSP.1982.1163951

Nuño-Maganda, M. and Arias-Estrada, M. (2005) Real-time FPGA-based architecture for bicubic interpolation: an application for digital image scaling. International Conference on Reconfigurable Computing and FPGAs, Puebla City, Mexico (28–30 September, 2005), p. 8. doi: 10.1109/ReConFig.2005.34

NVIDIA (2006) Technical Brief – NVIDIA GeForce 8800 GPU Architecture Overview, NVIDIA Corporation.

NXP (2007) I2C-Bus Specification and User Manual, Vol. UM10204 Rev. 03., NXP Semiconductors.

Offen, R.J. (1985) *VLSI Image Processing*, Collins, London.

Oh, S. and Kim, G. (2008) An architecture for on-the-fly correction of radial distortion using FPGA, in *Real-Time Image Processing 2008*, San Jose, California, USA (28–29 January, 2008), vol. 6811, SPIE, pp. 68110X-1-9. doi: 10.1117/12.767125

Ong, S.W., Kerkiz, N., Srijanto, B., Langston, M.C.T., Newport, D. and Bouldin, D. (2001) Automatic mapping of multiple applications to multiple adaptive computing systems. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01), Rohnert Park, California, USA (30 April–2 May, 2001), pp. 10–20. doi: 10.1109/FCCM.2001.15

Oraintara, S., Chen, Y.J. and Nguyen, T.Q. (2002) Integer fast Fourier transform. *IEEE Transactions on Signal Processing*, **50** (3), 607–618. doi: 10.1109/78.984749

Orwell, J., Remagnino, P. and Jones, G.A. (1999) Multi-camera colour tracking. Second IEEE Workshop on Visual Surveillance, Fort Collins, Colorado, USA (26 June, 1999), pp. 14–21. doi: 10.1109/VS.1999.780264

Ostresh, L.M. (1978) On the convergence of a class of iterative methods for solving the Weber location problem. *Operations Research*, **26** (4), 597–609. doi: 10.1287/opre.26.4.597

Otsu, N. (1979) A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, **9** (1), 62–66. doi: 10.1109/TSMC.1979.4310076

Ovod, V.I., Baxter, C.R., Massie, M.A. and McCarley, P.L. (2005) Advanced image processing package for FPGA-based re-programmable miniature electronics, in *Infrared Technology and Applications XXXI*, Orlando, Florida, USA (28 March–1 April, 2005), vol. 5783, SPIE, pp. 304–315. doi: 10.1117/12.603019

Page, I. (1996) Closing the gap between hardware and software: hardware-software cosynthesis at Oxford. IEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems (Digest No: 1996/036), Bristol, UK (22 February, 1996), pp. 2/1—12 doi: 10.1049/ic:19960221

Page, I. and Luk, W. (1991) Compiling Occam into field-programmable gate arrays. Field Programmable Logic and Applications, Oxford, UK (4–6 September, 1991), pp. 271–283.

Pajares, G. and de la Cruz, J.M. (2004) A wavelet-based image fusion tutorial. *Pattern Recognition*, **37** (9), 1855–1872. doi: 10.1016/j.patcog.2004.03.010

Parhami, B. (2000) *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, New York.

Parhi, K.K. (1991) A systematic approach for design of digit-serial signal processing architectures. *IEEE Transactions on Circuits and Systems*, **38** (4), 358–375. doi: 10.1109/31.75394

Park, J.W. (1986) An efficient memory system for image processing. *IEEE Transactions on Computers*, **35** (7), 669–674. doi: 10.1109/TC.1986.1676813

Parulski, K.A. (1985) Color filters and processing alternatives for one-chip cameras. *IEEE Transactions on Electron Devices*, **32** (8), 1381–1389. doi: 10.1109/T-ED.1985.22133

Pavan, P., Bez, R., Olivo, P. and Zanoni, E. (1997) Flash memory cells-an overview. *Proceedings of the IEEE*, **85** (8), 1248–1271. doi: 10.1109/5.622505

Peleg, A., Wilkie, S. and Weiser, U. (1997) Intel MMX for multimedia PCs. *Communications of the ACM*, **40** (1), 24–38. doi: 10.1145/242857.242865

Pell, O. and Luk, W. (2005) Quartz: a framework for correct and efficient reconfigurable design. International Conference on Reconfigurable Computing and FPGAs (ReConFig 2005), Puebla City, Mexico (28–30 September, 2005), p. 8. doi: 10.1109/RECONFIG.2005.32

Pellerin, D. and Thibault, S. (2005) *Practical FPGA programming in C*, Parson Education, Upper Saddle River, New Jersey, USA.

Pellerin, D., Edvenson, G., Shenoy, K. and Isaacs, D. (2005) Accelerating PowerPC software applications. *Xcell Journal* (55), 82–87.

Peterson, W.W. (1957) Addressing for random-access storage. *IBM Journal of Research and Development*, **1** (2), 130–146.

Pettersson, N. and Petersson, L. (2005) Online stereo calibration using FPGAs. IEEE Intelligent Vehicles Symposium, Las Vegas, Nevada, USA (6–8 June, 2005), pp. 55–60. doi: 10.1109/IVS.2005.1505077

Piccardi, M. (2004) Background subtraction techniques: a review. IEEE International Conference on Systems, Man and Cybernetics, The Hague, The Netherlands, vol. 4 (10–13 October, 2004), pp. 3099–3104. doi: 10.1109/ICSMC.2004.1400815

Pizer, S.M., Amburn, E.P., Austin, J.D., Cromartie, R., Geselowitz, A., Greer, T., Romeny, B.t.H., Zimmerman, J.B. and Zuiderveld, K. (1987) Adaptive histogram equalization and its variations. *Computer Vision, Graphics, and Image Processing*, **39** (3), 355–368. doi: 10.1016/S0734-189X(87)80186-X

Porikli, F. (2008) Reshuffling: a fast algorithm for filtering with arbitrary kernels, in *Real-Time Image Processing 2008*, San Jose, California, USA (28–29 January, 2008), vol. 6811, SPIE, pp. 68110M-1-10. doi: 10.1117/12.772114

Potkonjak, M., Srivastava, M.B. and Chandrakasan, A.P. (1996) Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **15** (2), 151–165. doi: 10.1109/43.486662

Pratt, W.K. (1978) *Digital Image Processing*, John Wiley & Sons, Inc., New York.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. (1993) *Numerical Recipes in C: The Art Of Scientific Computing*, Cambridge University Press.

Preston, K., Duff, M.J.B., Levialdi, S., Norgren, P.E. and Toriwaki, J. (1979) Basics of cellular logic with some applications in medical image processing. *Proceedings of the IEEE*, **67** (5), 826–856.

Prewitt, J.M.S. and Mendelsohn, M.L. (1966) The analysis of cell images. *Annals of the New York Academy of Sciences*, **128** (3), 1035–1053. doi: 10.1111/j.1749-6632.1965.tb11715.x

Proffitt, D. and Rosen, D. (1979) Metrication errors and coding efficiency of chain encoding schemes for the representation of lines and edges. *Computer Graphics and Image Processing*, **10** (4), 318–332. doi: 10.1016/S0146-664X(79)80041-6

Punchihewa, A., Bailey, D.G. and Hodgson, R.M. (2005) Colour reproduction performance of JPEG and JPEG2000 codecs. 8th International Symposium on DSP and Communication Systems, (DSPCS'2005) and 4th Workshop on the Internet, Telecommunications and Signal Processing, (WITSP'2005), Noosa Heads, Australia (19–21 December, 2005), pp. 312–317.

QuickLogic (2007a) Eclipse II Family Data Sheet, Vol. Rev R., QuickLogic Corporation.

QuickLogic (2007b) Eclipse Family Data Sheet, Vol. Rev F., QuickLogic Corporation.

Rachakonda, R.V., Athanas, P.M. and Abbott, A.L. (1995) High-speed region detection and labeling using an FPGA-based custom computing platform, in *Field Programmable Logic and Applications*, Oxford, UK (29 August–1 September, 1995), Lecture Notes in Computer Science, vol. LNCS 975, Springer, pp. 86–93. doi: 10.1007/3-540-60294-1_101

Ragnemalm, I. (1993) The Euclidean distance transformation in arbitrary dimensions. *Pattern Recognition Letters*, **14** (11), 883–888. doi: 10.1016/0167-8655(93)90152-4

Rajagopalan, K. and Sutton, P. (2001) A flexible multiplication unit for an FPGA logic block. IEEE International Symposium on Circuits and Systems (ISCAS 2001), Sydney, Australia, vol. 4 (6–9 May, 2001), pp. 546–549. doi: 10.1109/ISCAS.2001.922295

Rambabu, C. and Chakrabarti, I. (2007) An efficient immersion-based watershed transform method and its prototype architecture. *Journal of Systems Architecture*, **53** (4), 210–226. doi: 10.1016/j.sysarc.2005.12.005

Rambabu, C., Chakrabarti, L. and Mahanta, A. (2002) An efficient architecture for an improved watershed algorithm and its FPGA implementation. IEEE International Conference on Field-Programmable Technology (FPT), Hong Kong (16–18 December, 2002), pp. 370–373. doi: 10.1109/FPT.2002.1188713

Randen, T. and Husoy, J.H. (1999) Filtering for texture classification: a comparative study. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **21** (4), 291–310. doi: 10.1109/34.761261

Randhawa, S. and Li, J.S. (2005) CFA demosaicking with improved colour edge preservation. IEEE Region 10 Conference (IEEE Tencon'05), Melbourne, Australia (21–24 November, 2005). doi: 10.1109/TENCON.2005.301070

Ranganathan, N., Mehrotra, R. and Subramanian, S. (1995) A high speed systolic architecture for labeling connected components in an image. *IEEE Transactions on Systems, Man and Cybernetics*, **25** (3), 415–423. doi: 10.1109/21.364855

Ratha, N.K. and Jain, A.K. (1999) Computer vision algorithms on reconfigurable logic arrays. *IEEE Transactions on Parallel and Distributed Systems*, **10** (1), 29–43. doi: 10.1109/71.744833

Reddy, B.S. and Chatterji, B.N. (1996) An FFT-based technique for translation, rotation, and scale-invariant image registration. *IEEE Transactions on Image Processing*, **5** (8), 1266–1271. doi: 10.1109/83.506761

Reulke, R., Meysel, F. and Bauer, S. (2008) Situation analysis and atypical event detection with multiple cameras and multi-object tracking. *Robot Vision*, Auckland, New Zealand (18–20 February, 2008), Lecture Notes in Computer Science, vol. LNCS 4931, Springer, pp. 234–247. doi: 10.1007/978-3-540-78157-8_18

Reznik, Y.A., Hinds, A.T., Zhang, C., Yu, L. and Ni, Z. (2007) Efficient fixed-point approximations of the 8×8 inverse discrete cosine transform. Applications of Digital Image Processing XXX, San Diego, California, USA, vol. 6696, SPIE, p. 669617. doi: 10.1117/12.740228

Ridler, T.W. and Calvard, S. (1978) Picture thresholding using an iterative selection method. *IEEE Transactions on Systems, Man and Cybernetics*, **8** (8), 630–632. doi: 10.1109/TSMC.1978.4310039

Rissanen, J.J. (1976) Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, **20** (3), 198–203. doi: 10.1147/rd.203.0198

Roberts, D. (1996) *Internet Protocols Handbook*, Coriolis Group Books.

Robertson, J.E. (1958) A new class of digital division methods. *IRE Transactions on Electronic Computers*, **EC-7** (3), 218–222. doi: 10.1109/TEC.1958.5222579

Rosenfeld, A. and de la Torre, P. (1983) Histogram concavity analysis as an aid in threshold selection. *IEEE Transactions on Systems, Man and Cybernetics*, **13** (3), 231–235.

Rosenfeld, A. and Pfaltz, J. (1966) Sequential operations in digital picture processing. *Journal of the Association for Computing Machinery*, **13** (4), 471–494. doi: 10.1145/321356.321357

Russ, J.C. (2002) *The Image Processing Handbook*, 4th edn, CRC Press, Boca Raton, Florida.

Rutenbar, R.A., Baron, M., Daniel, T., Jayaraman, R., Or-Bach, Z., Rose, J. and Sechen, C. (2001) (When) will FPGAs kill ASICs? 38th annual Design Automation Conference, Las Vegas, Nevada, USA (18–22 June, 2001), pp. 321–322. doi: 10.1145/378239.378499

Saeed, A., Elbably, M., Abdelfadeel, G. and Eladawy, M.I. (2009) Efficient FPGA implementation of FFT/IFFT processor. *International Journal of Circuits, Systems and Signal Processing*, **3** (3), 103–110.

Sam, H. and Gupta, A. (1990) A generalized multibit recoding of two's complement binary numbers and its proof with application in multiplier implementations. *IEEE Transactions on Computers*, **39** (8), 1006–1015. doi: 10.1109/12.57039

Sanderson, C. (2004) Simplify FPGA application design with DIMEtalk. *Xcell Journal*, **51**, 104–107.

Sansaloni, T., Perez-Pascual, A. and Valls, J. (2003) Area-efficient FPGA-based FFT processor. *Electronics Letters*, **39** (19), 1369–1370. doi: 10.1049/el:20030892

Sarwar, A. (1997) CMOS power consumption and Cpd calculation, Application Note: SCAA035B. Texas Instruments.

Sawchuk, A.A. (1977) Real-time correction of intensity nonlinearities in imaging systems. *IEEE Transactions on Computers*, **26** (1), 34–39. doi: 10.1109/TC.1977.5009271

Schaffer, G. (1984) Machine vision: a sense for computer integrated manufacturing. *American Machinist*, **128** (6), 101–129.

Schoonees, J.A. and Palmer, T. (2009) Camera shading calibration using a spatially modulated field. Image and Vision Computing New Zealand (IVCNZ 2009), Wellington, New Zealand (23–25 November, 2009), pp. 191–196. doi: 10.1109/IVCNZ.2009.5378412

Schulte, M.J. and Stine, J.E. (1997) Symmetric bipartite tables for accurate function approximation. 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA (6–9 July, 1997), pp. 175–183. doi: 10.1109/ARITH.1997.614893

Schwarz, E.M. and Flynn, M.J. (1993) Hardware starting approximation for the square root operation. 11th Symposium on Computer Arithmetic, Windsor, Ontario, Canada (29 June–2 July, 1993), pp. 103–111. doi: 10.1109/ARITH.1993.378103

Sedcole, P. (2006) Reconfigurable platform-based design in FPGAs for video image processing. PhD Thesis, Department of Electrical and Electronic Engineering, Imperial College, London, UK.

Sedcole, N.P., Cheung, P.Y.K., Constantinides, G.A. and Luk, W. (2003) A reconfigurable platform for real-time embedded video image processing. *International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisbon, Portugal (1–3 September, 2003), Lecture Notes in Computer Science, vol. LNCS 2778, Springer, pp. 606–615. doi: 10.1007/b12007

Sedcole, P., Cheung, P.Y.K., Constantinides, G.A. and Luk, W. (2007) Run-time integration of reconfigurable video processing systems. *IEEE Transactions on VLSI Systems*, **15** (9), 1003–1016. doi: 10.1109/TVLSI.2007.902203

Sen, M., Corretjer, I., Haim, F., Saha, S., Schlessman, J., Lv, T., Bhattacharyya, S.S. and Wolf, W. (2007) Dataflow-based mapping of computer vision algorithms onto FPGAs. *EURASIP Journal on Embedded Systems*, **2007**, no. Article ID 49236. doi: 10.1155/2007/49236

Sen Gupta, G., Win, T.A., Messom, C., Demidenko, S. and Mukhopadhyay, S. (2003) Defect analysis of grit-blasted or spray painted surface using vision sensing techniques. Image and Vision Computing New Zealand (IVCNZ'03), Palmerston North, New Zealand (26–28 November, 2003), pp. 18–23.

Sen Gupta, G., Bailey, D. and Messom, C. (2004) A new colour-space for efficient and robust segmentation. Image and Vision Computing New Zealand (IVCNZ'04), Akaroa, New Zealand (21–23 November, 2004), pp. 315–320.

Sen Gupta, G. and Bailey, D. (2008) Discrete YUV look-up tables for fast colour segmentation for robotic applications. IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2008), Niagara Falls, Canada (4–7 May, 2008), pp. 963–968. doi: 10.1109/CCECE.2008.4564679

Sendur, L. and Selesnick, I.W. (2002) Bivariate shrinkage functions for wavelet-based denoising exploiting interscale dependency. *IEEE Transactions on Signal Processing*, **50** (11), 2744–2756. doi: 10.1109/TSP.2002.804091

Sensor to Image (2009a) GigE Receiver Reference Design Description, vol. X-1.0.2, Sensor to Image GmbH.

Sensor to Image (2009b) GigE Vision Reference Design, vol. A-0.2.3, Sensor to Image GmbH.

Sensor to Image (2009c) GigE Vision Reference Design Description, vol. X-1.0.4, Sensor to Image GmbH.

Serra, J. (1986) Introduction to mathematical morphology. *Computer Vision, Graphics and Image Processing*, **35** (3), 283–305. doi: 10.1016/0734-189X(86)90002-2

Sezgin, M. and Sankur, B. (2004) Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, **13** (1), 146–165. doi: 10.1117/1.1631315

Shamos, M.I. (1978) Robust picture processing operators and their implementation as circuits. ARPA Image Understanding Workshop, Pittsburgh, Pennsylvania, USA (November, 1978), pp. 127–129.

Shih, F.Y. and Wong, W.T. (1992) A new single-pass algorithm for extracting the mid-crack codes of multiple regions. *Journal of Visual Communication and Image Representation*, **3** (3), 217–224. doi: 10.1016/1047-3203(92)90018-O

Shilton, A. and Bailey, D. (2006) Drogue tracking by image processing for study of laboratory scale pond hydraulics. *Journal of Flow Measurement and Instrumentation*, **17** (1), 69–74. doi: 10.1016/j.flowmeasinst.2005.04.002

Sidahao, N., Constantinides, G.A. and Cheung, P.Y.K. (2003) Architectures for function evaluation on FPGAs. International Symposium on Circuits and Systems (ISCAS '03), Bangkok, Thailand, vol. 2 (25–28 May, 2003), pp. 804–807. doi: 10.1109/ISCAS.2003.1206096

SiliconBlue (2009) iCE65 Ultra Low-Power mobileFPGA Family, vol. v2.0.1, SiliconBlue Technologies Corporation.

Simmler, H., Levinson, L. and Männer, R. (2000) Multitasking on FPGA coprocessors. *10th International Conference on Field Programmable Logic and Applications*, Villach, Austria (27–30 August, 2000), Lecture Notes in Computer Science, vol. LNCS 1896, Springer, pp. 121–130. doi: 10.1007/3-540-44614-1_13

Simpson, P. (2010) *FPGA Design: Best Practices for Team-Based Design*, Springer. doi: 10.1007/978-1-4419-6339-0_1

Singh, S., Rose, J., Chow, P. and Lewis, D. (1992) The effect of logic block architecture on FPGA performance. *IEEE Journal of Solid-State Circuits*, **27** (3), 281–287. doi: 10.1109/4.121549

Skliarova, I. and Sklyarov, V. (2009) Recursion in reconfigurable computing: A survey of implementation approaches. International Conference on Field Programmable Logic and Applications (FPL), Prague, Czech Republic (31 August–2 September, 2009), pp. 224–229. doi: 10.1109/FPL.2009.5272304

Sklyarov, V. (2002) Reconfigurable models of finite state machines and their implementation in FPGAs. *Journal of Systems Architecture*, **47** (14–15), 1043–1064. doi: 10.1016/S1383-7621(02)00067-X

Sklyarov, V. and Skliarova, I. (2008) Design and implementation of parallel hierarchical finite state machines. Second International Conference on Communications and Electronics (ICCE 2008), Hoi an, Vietnam (4–6 June, 2008), pp. 33–38. doi: 10.1109/CCE.2008.4578929

So, H.K.H. (2007) BORPH: An operating system for FPGA-based reconfigurable computers. PhD Thesis, Electrical Engineering and Computer Sciences, University of California, Berkley.

Solomon, C. and Breckon, T. (2011) *Fundamentals of Digital Image Processing: A practical approach with examples in Matlab*. Wiley-Blackwell.

Specker, W.H. (1965) A class of algorithms for ln(x), exp(x), sin(x), cos(x), $\tan^{-1}$(x) and $\cot^{-1}$(x). *IEEE Transactions on Electronic Computers*, **EC-14** (1), 85–86. doi: 10.1109/PGEC.1965.264066

Stauffer, C. and Grimson, W.E.L. (1999) Adaptive background mixture models for real-time tracking. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Fort Collins, Colorado, USA, vol. 2 (23–25 June, 1999), pp. 246–252. doi: 10.1109/CVPR.1999.784637

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B. and Deprette, E. (2004) System design using Khan process networks: the Compaan/Laura approach. Design, Automation and Test in Europe Conference and Exhibition, Paris, France, vol. 1 (16–20 February, 2004), pp. 340–345. doi: 10.1109/DATE.2004.1268870

Steiger, C., Walder, H. and Platzner, M. (2004) Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, **53** (11), 1393–1407. doi: 10.1109/TC.2004.99

Stine, J.E. and Schulte, M.J. (1999) The symmetric table addition method for accurate function approximation. *Journal of VLSI Signal Processing*, **21** (2), 167–177. doi: 10.1023/A:1008004523235

Stokes, M., Anderson, M., Chandrasekar, S. and Motta, R. (1996) A standard default color space for the internet – sRGB. Available from http://www.w3.org/Graphics/Color/sRGB [cited 12 January, 2010].

Stone, H., Orchard, M. and Chang, E.C. (1999) Subpixel registration of images. Thirty-Third Asilomar Conference on Signals, Systems, and Computers, Monterey, California, USA, vol. 2 (24–27 October, 1999), pp. 1446–1452. doi: 10.1109/ACSSC.1999.831945

Stone, H.S., Orchard, M.T., Chang, E.C. and Martucci, S.A. (2001) A fast direct Fourier-based algorithm for subpixel registration of images. *IEEE Transactions on Geoscience and Remote Sensing*, **39** (10), 2235–2243. doi: 10.1109/36.957286

Stowers, J., Hayes, M. and Bainbridge-Smith, A. (2010) Phase correlation using shear average for image registration. Image and Vision Computing New Zealand, Queenstown, New Zealand (8–9 November, 2010).

Strickland, R.N. and Hahn, H.I. (1997) Wavelet transform methods for object detection and recovery. *IEEE Transactions on Image Processing*, **6** (5), 724–735. doi: 10.1109/83.568929

Suciu, R.E. and Reeves, A.P. (1982) A comparison of differential and moment based edge detectors. IEEE Computer Society Conference on Pattern Recognition and Image Processing, Las Vegas, Nevada, USA (14–17 June, 1982), pp. 97–102.

Sudha, N. and Mohan, A.R. (2008) Design of a hardware accelerator for path planning on the Euclidean distance transform. *Journal of Systems Architecture*, **54** (1–2), pp. 253–264. doi: 10.1016/j.sysarc.2007.06.003

Sukhsawas, S. and Benkrid, K. (2004) A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs. IEEE Computer society Annual Symposium on VLSI, Lafayette, Louisiana, USA (19–20 February, 2004), pp. 229–232. doi: 10.1109/ISVLSI.2004.1339538

Sun, S.H. and Lee, S.J. (2003) A JPEG chip for image compression and decompression. *The Journal of VLSI Signal Processing*, **35** (1), 43–60. doi: 10.1023/A:1023383820503

Swain, M.J. and Ballard, D.H. (1991) Colour indexing. *International Journal of Computer Vision*, **7** (1), 11–32. doi: 10.1007/BF00130487

Swan, S. (2006) SystemC transaction level models and RTL verification. 43rd ACM/IEEE Design Automation Conference, San Francisco, California, USA (24–28 July, 2006), pp. 90–92. doi: 10.1109/DAC.2006.229170

Swenson, R.L. and Dimond, K.R. (1999) A hardware FPGA implementation of a 2D median filter using a novel rank adjustment technique. Seventh International Conference on Image Processing And Its Applications, Manchester, UK, vol. Conf Publ 465 (13–15 July, 1999), pp. 103–106. doi: 10.1049/cp:19990290

Tabula (2009) Abax™ Family Overview. Tabula Inc.

Tabula (2010a) Abax™ Product Family Overview. Tabula Inc.

Tabula (2010b) Tabula SpaceTime™ architecture White paper. Tabula Inc.

Tagzout, S., Achour, K. and Djekoune, O. (2001) Hough transform algorithm for FPGA implementation. *Signal Processing*, **81** (6), 1295–1301. doi: 10.1016/S0165-1684(00)00248-6

Tahir, M.A., Bouridane, A., Kurugollu, F. and Amira, A. (2003a) An FPGA based coprocessor for calculating grey level co-occurrence matrix. IEEE International Symposium on Micro-NanoMechatronics and Human Science, Cairo, Egypt, vol. 2 (27–30 December, 2003), pp. 868–871. doi: 10.1109/MWSCAS.2003.1562424

Tahir, M.A., Roula, M.A., Bouridane, A., Kurugollu, F. and Amira, A. (2003b) An FPGA based co-processor for GLCM texture features measurement. 10th IEEE International Conference on Electronics, Circuits and Systems, Sharjah, United Arab Emirates, vol. 3 (14–17 December, 2003), pp. 1006–1009. doi: 10.1109/ICECS.2003.1301679

Tang, K., Astola, J. and Neuvo, Y. (1994) Multichannel edge enhancement in color image processing. *IEEE Transactions on Circuits and Systems for Video Technology*, **4** (5), 468–479. doi: 10.1109/76.322994

Tatas, K., Soudris, D.J., Siomos, D., Dasygenis, M. and Thanailakis, A. (2002) A novel division algorithm for parallel and sequential processing. 9th International Conference on Electronics, Circuits and Systems, Dubrovnik, Croatia, vol. 2 (15–18 September, 2002), pp. 553–556. doi: 10.1109/ICECS.2002.1046225

Taubman, D. (2000) High performance scalable image compression with EBCOT. *IEEE Transactions on Image Processing*, **9** (7), 1158–1170. doi: 10.1109/83.847830

Therrien, C.W., Quatieri, T.F. and Dudgeon, D.E. (1986) Statistical model-based algorithms for image analysis. *Proceedings of the IEEE* **74** (4), 532–551.

Thevenaz, P., Ruttimann, U.E. and Unser, M. (1998) A pyramid approach to subpixel registration based on intensity. *IEEE Transactions on Image Processing*, **7** (1), 27–41. doi: 10.1109/83.650848

Thomas, D.B. and Luk, W. (2005) High quality uniform random number generation through LUT optimised linear recurrences. IEEE International Conference on Field-Programmable Technology, Singapore (11–14 December, 2005), pp. 61–68. doi: 10.1109/FPT.2005.1568526

Thomas, D.B. and Luk, W. (2009) Using FPGA resources for direct generation of multivariate Gaussian random numbers. International Conference on Field Programmable Technology (FPT'09), Sydney, Australia (9–11 December, 2009), pp. 344–347. doi: 10.1109/FPT.2009.5377680

Tian, Q. and Huhns, M.N. (1986) Algorithms for sub-pixel registration. *Computer Vision, Graphics and Image Processing*, **35** (2), 220–233. doi: 10.1016/0734-189X(86)90028-9

Timmermann, D., Hahn, H. and Hosticka, B.J. (1992) Low latency time CORDIC algorithms. *IEEE Transactions on Computers*, **41** (8), 1010–1015. doi: 10.1109/12.156543

Tocher, K.D. (1958) Techniques of multiplication and division for automatic binary divider. *Quarterly Journal of Mechanics and Applied Mathematics*, **11** (3), 364–384. doi: 10.1093/qjmam/11.3.364

Todman, T.J., Constantinides, G.A., Wilton, S.J.E., Mencer, O., Luk, W. and Cheung, P.Y.K. (2005) Reconfigurable computing: architectures and design methods. IEE Proceedings Computers and Digital Techniques, 152 (2), 193–207. doi: 10.1049/ip-cdt:20045086

Toledo, F., Martinez, J.J., Garrigos, J., Ferrandez, J. and Rodellar, V. (2006) Skin color detection for real time mobile applications. International Conference on Field Programmable Logic and Applications (FPL'06), Madrid, Spain (28–30 August, 2006), pp. 721–724. doi: 10.1109/FPL.2006.311299

Tombs, J., Aguirre Echanove, M.A., Munoz, F., Baena, V., Torralba, A., Fernandez-Leon, A. and Tortosa, F. (2004) The implementation of an FPGA hardware debugger system with minimal system overhead. *14th International Conference on Field Programmable Logic and Application*, Antwerp, Belgium (29 August–1 September, 2004), Lecture Notes in Computer Science, vol. LNCS 3203, Springer, pp. 1062–1066. doi: 10.1007/b99787

Tomczak, T. (2006) Residue arithmetic in FPGA matrices. International Conference on Dependability of Computer Systems, Szklarska Poreba, Poland (25–27 May, 2006), pp. 297–305. doi: 10.1109/DEPCOS-RELCOMEX.2006.43

Torres-Huitzil, C. and Arias-Estrada, M. (2004) Real-time image processing with a compact FPGA-based systolic architecture. *Real-Time Imaging*, **10** (3), 177–187. doi: 10.1016/j.rti.2004.06.001

Traver, V.J. and Pla, F. (2003) The log-polar image representation in pattern recognition tasks. First Iberian Conference on Pattern Recognition and Image Analysis, Mallorca, Spain (4–6 June, 2003), pp. 1032–1040. doi: 10.1007/b12122

Trein, J., Schwarzbacher, A.T., Hoppe, B., Noffz, K.H. and Trenschel, T. (2007) Development of a FPGA based real-time blob analysis circuit. Irish Signals and Systems Conference, Derry, UK (13–14 September, 2007), pp. 121–126.

Trein, J., Schwarzbacher, A.T. and Hoppe, B. (2008) FPGA implementation of a single pass real-time blob analysis using run length encoding. in MPC-Workshop, Ravensburg-Weingarten, Germany (1 February, 2008), pp. 71–77.

Trieu, D.B.K. and Maruyama, T. (2006) Implementation of a parallel and pipelined watershed algorithm on FPGA. International Conference on Field Programmable Logic and Applications (FPL'06), Madrid, Spain (28–30 August, 2006), pp. 561–566. doi: 10.1109/FPL.2006.311267

Trieu, D.B.K. and Maruyama, T. (2007) A pipeline implementation of a watershed algorithm on FPGA. International Conference on Field Programmable Logic and Applications (FPL 2007), Amsterdam, The Netherlands (27–29 August, 2007), pp. 714–717. doi: 10.1109/FPL.2007.4380752

Trieu, D.B.K. and Maruyama, T. (2008) An implementation of a watershed algorithm based on connected components on FPGA. International Conference on Field Programmable Technology, Taipei, Taiwan (7–10 December, 2008), pp. 253-256. doi: 10.1109/FPT.2008.4762391

Trummer, R.K.L. (2005) A high-performance data-dependent hardware integer divider. Masters Thesis, University of Salzburg, Salzburg, Austria.

Trussell, H.J. (1979) Comments on "Picture thresholding using an iterative selection method". *IEEE Transactions on Systems, Man and Cybernetics*, **9** (5), 311–1311 doi: 10.1109/TSMC.1979.4310204

Tsai, R.Y. (1987) A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses. *IEEE Journal of Robotics and Automation*, **3** (4), 323–344. doi: 10.1109/JRA.1987.1087109

Tsai, R.Y. and Huang, T.S. (1984) Multiframe image restoration and registration, *Advances in Computer Vision and Image Processing*, vol. 1, JAI Press, pp. 317–339.

Tsoi, K.H., Leung, K.H. and Leong, P.H.W. (2007) High performance physical random number generator. *IET Computers and Digital Techniques*, **1** (4), 349–352. doi: 10.1049/iet-cdt:20050173

Turk, M.A. and Pentland, A.P. (1991) Face recognition using eigenfaces. IEEE Conference on Computer Vision and Pattern Recognition (CVPR'91), Maui, Hawaii, USA (3–6 June, 1991), pp. 586–591. doi: 10.1109/CVPR.1991.139758

Uber, G.T. (1986) Illumination methods for machine vision. *Optics, Illumination, and Image Sensing for Machine Vision*, Cambridge, Massachusetts, USA (30–31 October, 1986), vol. 728, SPIE, pp. 93–102.

Unger, S.H. (1958) A computer oriented toward spatial problems. *Proceedings of the IRE*, **46** (10), 1744–1750. doi: 10.1109/JRPROC.1958.286755

Unsal, O.S. and Koren, I. (2003) System-level power-aware design techniques in real-time systems. *Proceedings of the IEEE*, **91** (7), 1055–1069. doi: 10.1109/JPROC.2003.814617

Unser, M. (2000) Sampling - 50 years after Shannon. *Proceedings of the IEEE*, **88** (4), 569–587. doi: 10.1109/5.843002

Unser, M., Aldroubi, A. and Eden, M. (1991) Fast B-spline transforms for continuous image representation and interpolation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **13** (3), 277–285. doi: 10.1109/34.75515

Uzun, I.S. and Bouridane, A.A.A. (2003) FPGA implementations of fast Fourier transforms for real-time signal and image processing. IEEE International Conference on Field-Programmable Technology (FPT), Tokyo, Japan (15–17 December, 2003), pp. 102–109. doi: 10.1109/FPT.2003.1275737

Uzun, I.S., Amira, A. and Bouridane, A. (2005) FPGA implementations of fast Fourier transforms for real-time signal and image processing. *IEE Proceedings – Vision, Image and Signal Processing*, **152** (3), 283–296. doi: 10.1049/ip-vis:20041114

Vanmeerbeeck, G., Schaumont, P., Vernalde, S., Engels, M. and Bolsens, I. (2001) Hardware/software partitioning of embedded system in OCAPI-xl. International Conference on Hardware Software Codesign, Copenhagen, Denmark (25–27 April, 2001), pp. 30–35. doi: 10.1145/371636.371665

Velten, J. and Kummert, A. (2002) FPGA-based implementation of variable sized structuring elements for 2D binary morphological operations. IEEE International Workshop on Electronic Design, Test, and Applications (DELTA), Christchurch, New Zealand (29–31 January, 2002), pp. 309–312. doi: 10.1109/DELTA.2002.994636

VESA (2003) Coordinated Video Timings Standard, Version 1.1., Video Electronics Standards Association.

VESA (2007) VESA and Industry Standards and Guidelines for Computer Display Monitor Timing (DMT), Version 1.0, Revision 11, Video Electronics Standards Association.

Vezhnevets, V., Sazonov, V. and Andreeva, A. (2003) A survey on pixel-based skin color detection techniques. Graphicon-2003, Moscow, pp. 85–92.

Villalba, J., Hidalgo, J.A., Zapata, E.L., Antelo, E. and Bruguera, J.D. (1995) CORDIC architectures with parallel compensation of the scale factor. International Conference on Application Specific Array Processors, Strasbourg, France (24–26 July, 1995), pp. 258–269. doi: 10.1109/ASAP.1995.522930

Villasenor, J., Jones, C. and Schoner, B. (1995) Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, **5** (6), 565–567. doi: 10.1109/76.475899

Villasenor, J., Schoner, B., Chia, K.N., Zapata. C., Kim, H.J., Jones, C., Lansing, S. and Mangione-Smith, B. (1996) Configurable computing solutions for automatic target recognition. IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, USA (17–19 April, 1996), pp. 70–79. doi: 10.1109/FPGA.1996.564749

Vincent, L. (1991) Exact Euclidean distance function by chain propagations. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Maui, Hawaii, USA (3–6 June, 1991), pp. 520–525. doi: 10.1109/CVPR.1991.139746

Vincent, L. and Soille, P. (1991) Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **13** (6), 583–598. doi: 10.1109/34.87344

Vinh, T.Q. and Kim, Y.C. (2010) Edge-preserving algorithm for block artifact reduction and its pipelined architecture. *ETRI Journal*, **32** (3), 380–389. doi: 10.4218/etrij.10.0109.0290

Viola, P. and Wells, W.M. (1997) Alignment by maximization of mutual information. *International Journal of Computer Vision*, **24** (2), 137–154. doi: 10.1023/A:1007958904918

Vogt, R.C. (1986) Formalised approaches to image algorithm development using mathematical morphology. Vision'86, Detroit, Michigan, USA, pp. 5/17–5/37 (3–5 June, 1986).

Volder, J.E. (1959) The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, **EC-8** (3), 330–334. doi: 10.1109/TEC.1959.5222693

Vuillemin, J.E., Bertin, P., Roncin, D., Shand, M., Touati, H.H. and Boucard, P. (1996) Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, **4** (1), 56–69. doi: 10.1109/92.486081

Wallace, C.S. (1964) A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, **13** (1), 14–17. doi: 10.1109/PGEC.1964.263830

Wallace, G.K. (1992) The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, **38** (1), 17–34. doi: 10.1109/30.125072

Walther, J.S. (1971) A unified algorithm for elementary functions. in Spring Joint Computer Conference, Atlantic City, New Jersey, USA (18–20 May, 1971), pp. 379–385.

Walther, J.S. (2000) The story of unified CORDIC. *Journal of VLSI Signal Processing*, **25** (2), 107–112. doi: 10.1023/A:1008162721424

Waltz, F.M. (1994a) Application of SKIPSM to binary template matching, in *Machine Vision Applications, Architectures, and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 417–427. doi: 10.1117/12.188752

Waltz, F.M. (1994b) Application of SKIPSM to grey-level morphology, in *Machine Vision Applications, Architectures, and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 428–435. doi: 10.1117/12.188754

Waltz, F.M. (1994c) Separated-kernel image processing using finite-state machines (SKIPSM), in *Machine Vision Applications, Architectures, and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 386–395. doi: 10.1117/12.188749

Waltz, F.M. (1995) Application of SKIPSM to binary correlation, in *Machine Vision Applications, Architectures, and Systems Integration IV*, Philadelphia, USA (23–24 October, 1995), vol. 2597, SPIE, pp. 82–91. doi: 10.1117/12.223967

Waltz, F.M. and Garnaoui, H.H. (1994a) Application of SKIPSM to binary morphology, in *Machine Vision Applications, Architectures, and Systems Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 396–407. doi: 10.1117/12.188750

Waltz, F.M. and Garnaoui, H.H. (1994b) Fast computation of the grassfire transform using SKIPSM, in *Machine Vision Applications, Architectures and System Integration III*, Boston, Massachusetts, USA (31 October–2 November, 1994), vol. 2347, SPIE, pp. 408–416. doi: 10.1117/12.188751

Waltz, F.M., Hack, R. and Batchelor, B.G. (1998) Fast efficient algorithms for 3x3 ranked filters using finite-state machines, in *Machine Vision Systems for Inspection and Metrology VII*, Boston, Massachusetts, USA (4–5 November, 1998), vol. 3521, SPIE, pp. 278–287. doi: 10.1117/12.326970

Wang, Y. (1998) New Chinese remainder theorems. Thirty-Second Asilomar Conference on Signals, Systems & Computers, Pacific Grove, California, USA, vol. 1 (1–4 November, 1998), pp. 165–171. doi: 10.1109/ACSSC.1998.750847

Wang, S., Piuri, V. and Swartzlander, E.E. (1996) A unified view of CORDIC processor design. IEEE 39th Midwest Symposium on Circuits and Systems, Ames, Iowa, USA, vol. 2 (18–21 August, 1996), pp. 852–855. doi: 10.1109/MWSCAS.1996.588050

Wang, Y., Song, X., Aboulhamid, M. and Shen, H. (2002) Adder based residue to binary number converters for (2^n-1, 2^n, 2^n+1). *IEEE Transactions on Signal Processing*, **50** (7), 1772–1779. doi: 10.1109/TSP.2002.1011216

Wang, J., Hall-Holt, O., Konecny, P. and Kaufman, A.E. (2005) Per pixel camera calibration for 3D range scanning, in *Videometrics VIII*, San Jose, California, USA (18–20 January, 2005), vol. 5665, SPIE, pp. 342–352. doi: 10.1117/12.586209

Weems, C.C. (1991) Architectural requirements of image understanding with respect to parallel processing. *Proceedings of the IEEE* **79** (4), 537–547. doi: 10.1109/5.92046

Wei, Z., Lee, D.J., Nelson, B.E., Archibald, J.K. and Edwards, B.B. (2008) FPGA-based embedded motion estimation sensor. *International Journal of Reconfigurable Computing*, **2008**, no. Article ID 636145. doi: 10.1155/2008/636145

Weinhardt, M. and Luk, W. (2001a) Memory access optimisation for reconfigurable systems. *IEE Proceedings Computers and Digital Techniques*, **148** (3), 105–112. doi: 10.1049/ip-cdt:20010514

Weinhardt, M. and Luk, W. (2001b) Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **20** (2), 234–248. doi: 10.1109/43.908452

Welch, T.A. (1984) A technique for high-performance data compression. *IEEE Computer*, **17** (6), 8–19. doi: 10.1109/MC.1984.1659158

Wendt, P.D., Coyle, E.J. and Gallagher, N.C. (1986) Stack filters. *IEEE Transactions on Acoustics, Speech and Signal Processing*, **34** (4), 898–911. doi: 10.1109/TASSP.1986.1164871

Weszka, J.S., Nagel, R.N. and Rosenfeld, A. (1974) A threshold selection technique. *IEEE Transactions on Computers*, **23** (12), 1322–1326. doi: 10.1109/T-C.1974.223858

Wigley, G. and Kearney, D. (2001) The development of an operating system for reconfigurable computing. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), Rohnert Park, California, USA (30 April–2 May, 2001), pp. 249–250. doi: 10.1109/FCCM.2001.43

Wilen, A., Schade, J.P. and Thornburg, R. (2003) *Introduction to PCI Express: A Hardware and Software Developer's Guide*, Intel Press.

Will, P.M. and Pennington, K.S. (1972) Grid coding: a novel technique for image processing. *Proceedings of the IEEE*, **60** (6), 669–680. doi: 10.1109/PROC.1972.8726

Williams, L. (1983) Pyramidal parametrics. *ACM SIGGRAPH Computer Graphics*, **17** (3), 1–11. doi: 10.1145/964967.801126

Williams, J. (2009) Embedded Linux on Xilinx MicroBlaze. Xilinx University Program and PetaLogix Professor's Workshop Series. PetaLogix Qld Pty Ltd.

Williams, C.S. and Rasure, J.R. (1990) A visual language for image processing. IEEE Workshop on Visual Languages, Skokie, Illinois, USA (4–6 October, 1990), pp. 86–91. doi: 10.1109/WVL.1990.128387

Willson, R.G. and Shafer, S.A. (1994) What is the center of the image? *Journal of the Optical Society of America A*, **11** (11), pp. 2946–2955. doi: 10.1364/JOSAA.11.002946

Wilson, G.R. (1997) Properties of contour codes. *IEE Proceedings Vision, Image and Signal Processing*, **144** (3), 145–149. doi: 10.1049/ip-vis:19971159

Wilson, H.R. and Giese, S.C. (1977) Threshold visibility of frequency gradient patterns. *Vision Research*, **17** (10), 1177–1190. doi: 10.1016/0042-6989(77)90152-3

Wilson, J.C. and Hodgson, R.M. (1992) Log-polar mapping applied to pattern representation and recognition, *Computer Vision and Image Processing*, Academic Press, pp. 245–277.

Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S. and Hennessy, J.L. (1994) SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, **29** (12), 31–37. doi: 10.1145/193209.193217

Wirthlin, M.J., Hutchings, B.L. and Worth, C. (2001) Synthesizing RTL hardware from Java byte codes, *in Field-Programmable Logic and Applications (FPL 2001)*, Belfast, UK (27–29 August, 2001), Lecture Notes in Computer Science, vol. LNCS 2147, Springer, pp. 123–132. doi: 10.1007/3-540-44687-7_13

Witten, I.H., Neal, R.M. and Cleary, J.G. (1987) Arithmetic coding for data compression. *Communications of the ACM*, **30** (6), 520–540. doi: 10.1145/214762.214771

Wolberg, G. (1990) *Digital Image Warping*, IEEE Computer Society Press, Los Alamitos, California.

Wolberg, G. and Boult, T.E. (1989) Separable image warping with spatial lookup tables. *ACM SIGGRAPH Computer Graphics*, **23** (3), 369–378. doi: 10.1145/74334.74371

Wolberg, G., Sueyllam, H.M., Ismail, M.A. and Ahmed, K.M. (2000) One-dimensional resampling with inverse and forward mapping functions. *Journal of Graphics Tools*, **5**, 11–33.

Wong, S., Vassiliadis, S. and Cotofana, S. (2002) A sum of absolute differences implementation in FPGA hardware. 28th Euromicro Conference, Dortmund, Germany (4–6 September, 2002), pp. 183–188. doi: 10.1109/EURMIC.2002.1046155

Woods, R., Trainor, D. and Heron, J.P. (1998) Applying an XC6200 to real-time image processing. *IEEE Design & Test of Computers*, **15** (1), 30–38. doi: 10.1109/54.655180

Wu, K., Otoo, E. and Shoshani, A. (2005) Optimizing connected component labelling algorithms, in *Medical Imaging 2005: Image Processing*, San Diego, California, USA (15–17 February, 2005), vol. 5747, SPIE, pp. 1965–1976. doi: 10.1117/12.596105

Xilinx (2001) VirtexTM 2.5V Field Programmable Gate Arrays, Vol. DS003 (v2.5), Xilinx Inc.

Xilinx (2007a) Virtex-II Platform FPGAs: Complete Data Sheet, Vol. DS031 (v3.5), Xilinx Inc.

Xilinx (2007b) Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, Vol. DS083 (v4.7), Xilinx Inc.

Xilinx (2008a) Spartan-3 FPGA Family Data Sheet, Vol. DS099 (v2.4), Xilinx Inc.

Xilinx (2008b) Spartan-II FPGA Family Data Sheet, Vol. DS001 (v2.8), Xilinx Inc.

Xilinx (2008c) Virtex-4 FPGA User Guide, Vol. UG070 (v2.6), Xilinx Inc.

Xilinx (2008d) Virtex-5 FPGA User Guide, Vol. UG190 (v4.4), Xilinx Inc.

Xilinx (2008e) Spartan-3A DSP FPGA Family Data Sheet, Vol. DS610 (v2.1), Xilinx Inc.

Xilinx (2008f) ChipScope Pro 10.1 Software and Cores User Guide, Vol. UG029, Xilinx Inc.

Xilinx (2009a) Spartan-6 Family Overview, Vol. DS160 (V1.0), Xilinx Inc.

Xilinx (2009b) Spartan-3AN FPGA In-System Flash User Guide, Vol. UG333 (v2.1), Xilinx Inc.

Xilinx (2009c) AccelDSP Synthesis Tool User Guide, Vol. UG634 (v11.4), Xilinx Inc.

Xilinx (2010a) 7 Series Overview, Vol. DS150 (V1.0), Xilinx Inc.

Xilinx (2010b) Spartan-6 FPGA CLB User Guide, Vol. UG384 (v1.1), Xilinx Inc.

Xilinx (2010c) Spartan-6 FPGA Memory Controller, vol. UG388 (v2. 1), Xilinx Inc.

Xilinx (2010d) Virtex-6 Family Overview, Vol. DS150 (V2.2), Xilinx Inc.

Yamada, A., Nishida, K., Sakurai, R., Kay, A., Nomura, T. and Kambe, T. (1999) Hardware synthesis with the Bach system. 1999 IEEE International Symposium on Circuits and Systems (ISCAS '99), Orlando, Florida, USA, vol. 6 (30 May–2 June, 1999), pp. 366–369. doi: 10.1109/ISCAS.1999.780171

Yao, L., Feng, H., Zhu, Y., Jiang, Z., Zhao, D., and Feng, W. (2009) An architecture of optimised SIFT feature detection for an FPGA implementation of an image matcher. International Conference on Field-Programmable Technology (FPT 2009), Sydney, Australia (9–11 December, 2009), pp. 30–37. doi: 10.1109/FPT.2009.5377651

Yu, N., Kim, K. and Salcic, Z. (2004) A new motion estimation algorithm for mobile real-time video and its FPGA implementation. IEEE Region 10 Conference (TENCON), Chiang Mai, Thailand, vol. 1 (21–24 November, 2004), pp. 383–386. doi: 10.1109/TENCON.2004.1414437

Yuen, H.K., Princen, J., Illingworth, J. and Kittler, J. (1990) Comparative study of Hough transform methods for circle finding. *Image and Vision Computing*, **8** (1), 71–77. doi: 10.1016/0262-8856(90)90059-E

Zahn, C.T. and Roskies, R.Z. (1972) Fourier descriptors for plane closed curves. *IEEE Transactions on Computers*, **21** (3), 269–281. doi: 10.1109/TC.1972.5008949

Zhang, D. and Lu, G. (2002) A comparative study of Fourier descriptors for shape representation and retrieval. 5th Asian Conference on Computer Vision, Melbourne, Australia (23–25 January, 2002), pp. 646–651.

Zingaretti, P., Gasparroni, M. and Vecci, L. (1998) Fast chain coding of region boundaries. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **20** (4), 407–415. doi: 10.1109/34.677272

Zitova, B. and Flusser, J. (2003) Image registration methods: a survey. *Image and Vision Computing*, **21** (11), 977–1000. doi: 10.1145/146370.146374

Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **23** (3), 337–343. doi: 10.1109/TIT.1977.1055714

Zoss, R., Habegger, A., Bandi, V., Goette, J. and Jacomet, M. (2011) Comparing signal processing hardware-synthesis methods based on the Matlab tool-chain, in *6th International Symposium on Electronic Design, Test and Applications*, Queenstown, New Zealand, pp 281–286 (17–19 January, 2011). doi: 10.1109/DELTA.2011.58

Index

Page numbers appearing in bold refer to definitions or significant sections within the document where the topic is covered in some depth (for example section headings).