به نام خدا

مؤسسه فرهنگی هنری
دیباگران تهران

آموزش VHDL

مترجم
مهندس فرزان گیتی

RWTÜV
TÜV CERT
EN ISO 9001

دارنده گواهینامه ISO 9001/2000
در زمینه نشر کتاب و طراحی جلد

آموزش VHDL

مترجم : مهندس فرزان گیتی

ناشر : مؤسسه فرهنگی هنری دیباگران تهران

حروفچینی و صفحه‌آرایی : مجتمع فنی تهران

طرح روی جلد : مجتمع فنی تهران

چاپ : وطن آرا

نوبت چاپ : اول

تاریخ نشر : شهریور ماه ۱۳۸۳

تیراژ : ۳۰۰۰ نسخه

قیمت : ۴۱۵۰۰ ریال

شابک : ۴- ۴۴۹-۳۵۴-۹۶۴

ISBN : 964-354-449-4

شوهولم ، اشتفان Sjoholm , Stefan

آموزش VHDL [وی.اچ.دی.ال] / اشتفان شوهولم ،لنارت‌لیند؛ مترجم فرزان گیتی.—تهران: مؤسسه‌فرهنگی‌هنری‌دیباگران‌تهران،۱۳۸۳.

۴۷۲ ص.:مصور،جدول،نمودار .

ISBN 964-354-449-4

فهرستنویسی بر اساس اطلاعات فیپا .

عنوان اصلی: VHDL For Designers , 1997.

۱.وی.اچ.دی.ال (زبان توصیفی سخت افزار) .۲.کامپیوترهای رقمی-- شبیه‌سازی.الف.لیند،لنارت Lindh ,Lennart ب.گیتی،فرزان،۱۳۵۸- ،مترجم.

TK۷۸۸۵/۷/ش۹۱۸ ۶۲۱/۳۹۲

۱۳۸۳

کتابخانه ملی ایران ۸۳-۸۱۴۱م

آدرس: سعادت آباد، میدان کاج . خ سرو شرقی. روبه‌روی خ علامه . ساختمان شماره ۹۷

تلفن : ۷- ۲۰۹۸۴۴۶ صندوق پستی: ۹۴۳/ ۱۴۳۳۵

فهرست مطالب

فصل چهارم : VHDL متوالی

فصل پنجم : کتابخانه، بسته و زیربرنامه‌ها

فصل ششم : VHDL ساختاری

فصل هفتم : RAM و ROM

فصل هشتم : محیط آزمایش

فصل نهم : ماشینهای حالت

فصل دهم : سنتز در سطح RT

فصل یازدهم : آشنایی با روشهای طراحی

فصل دوازدهم : آشنایی با روشهای آزمایش

فصل سیزدهم : نمونه‌سازی اولیه با سرعت بالا

فصل چهاردهم : خطاهای متداول در VHDL و چگونگی اجتناب از آنها

فصل پانزدهم : مثالهایی از طراحی و نکات راهنما

فصل شانزدهم : ابزارهای تولید

فصل هفدهم : سنتز رفتاری

مقدمه ناشر

حمد و سپاس ایزد منان را که با الطاف بیکران خود این توفیق را به ما ارزانی داشت تا بتوانیم در راه ارتقای دانش عمومی و فرهنگ این مرز و بوم در زمینه چاپ و نشر کتب علمی دانشگاهی، علوم پایه و به ویژه علوم کامپیوتر و انفورماتیک گامهایی هر چند کوچک برداشته و در انجام رسالتی که بر عهده داریم مؤثر واقع شویم. گستردگی علوم و توسعه روزافزون آن، شرایطی را به وجود آورده که هر روز شاهد تحولات اساسی چشمگیر در سطح جهان هستیم. این گسترش و توسعه نیاز به منابع مختلف از جمله کتاب را به عنوان اصیل‌ترین و مطمئن‌ترین راه دستیابی به اطلاعات و اطلاع‌رسانی، بیش از پیش روشن می‌نماید. در این راستا ، واحد انتشارات مؤسسه فرهنگی هنری دیباگران با همکاری جمعی از اساتید، مؤلفان، مترجمان، متخصصان، پژوهشگران و نیز پرسنل ورزیده و ماهر در زمینه امور نشر در صدد است تا با تلاشهای مستمر خود برای رفع کمبودها و نیازهای موجود، منابعی پربار، معتبر و با کیفیت بالا در اختیار علاقه‌مندان قرار دهد.

کتابی که در دست دارید به همت " آقای فرزان گیتی" و تلاش جمعی از همکاران انتشارات میسر گشته و شایسته است از یکایک این گرامیان تشکر و قدردانی کنیم.

ویراستاری ادبی : خانم فریبا امین کاظمی

ویرایش و صفحه‌آرایی کامپیوتری : خانم ملوک احمدسلطانی و آقای عباس هادی‌نیا

تهیه تصاویر : آقای مهدی علیمردانی

طراحی روی جلد : خانم محبوبه توکلی

امور چاپ و نشر : آقای حیدر شفیعی

ناظر چاپ : آقای کریم براغ

در خاتمه از عموم هموطنان عزیز و دانش پژوهان گرامی خواهشمندیم ما را با ارائه پیشنهادها و انتقادهای خود در بهبود کیفی کارهای انجام شده راهنمایی نمایند تا بتوانیم در آینده کتابهایی با کیفیت بهتر تقدیم حضورشان کنیم.

مدیر انتشارات

مؤسسه فرهنگی هنری دیباگران تهران

مقدمه مترجم

زبانهای توصیف سخت‌افزاری[1] (HDLs) برای توصیف ساختاری و عملکردی سیستمهای الکترونیکی به کار می‌روند و گسترش این زبانها و در رأس آنها VHDL بیشتر به دنبال پیچیده‌تر شدن روز به روز طراحی‌های الکترونیکی صورت می‌پذیرد. VHDL این امکان را فراهم ساخته است که مراحل مختلف یک برنامه به طور هم‌زمان و موازی در کنار اجرای مرحله به مرحله آن انجام گیرد. سادگی کار و کوتاه بودن مدت طراحی با VHDL از جمله ویژگیهایی است که باعث رواج بیشتر این زبان توصیف سخت‌افزاری به جای طراحی‌های شماتیکی متداول گردیده است.

در جهان پیشرفته امروز به VHDL به عنوان یک پدیده جدید در طراحی پروژه‌های الکترونیکی نگاه می‌شود و در اهمیت این زبان کافی است اشاره شود که IEEE در دهه ۱۹۸۰ اقدام به عرضه اولین نسخه استاندارد این زبان تحت عنوان VHDL-87 نمود. در دهه ۱۹۹۰ نیز شاهد ورود نسخه جدید این زبان (نسخه استاندارد VHDL-93) هستیم و گمان می‌رود دیرزمانی نخواهد گذشت که این زبان به دلیل کارآیی، کیفیت و سرعت بالای خود کاربردی فراگیرتر پیدا نماید. در حال حاضر یکی از کاربردهای VHDL در کشورمان استفاده از آن در طراحی‌هایی است که در داخل تراشه‌های FPGA و CPLD اجرا می‌گردند. با توجه به ساختار داخلی این تراشه‌ها، قابلیت پیاده‌سازی انواع کدهای VHDL وجود دارد. علت رونق گرفتن طراحی‌های جدید با FPGA را می‌توان در قابلیت بالای آن برای جا دادن طرحهای بزرگ، سرعت عمل بالا، نویزپذیری کم و سایر امتیازات آن جستجو کرد.

استفاده از زبان VHDL در کشور ما سابقه کوتاهی دارد و کتابهای آموزشی یا راهنما که درباره آن در سطح دانشگاهی به فارسی انتشار یافته بسیار اندک است. این کتاب یکی از جامع‌ترین کتابهای VHDL است که به عنوان کتاب درسی و مرجع در دانشکده‌های برق (الکترونیک) و کامپیوتر تدریس می‌شود و علاوه بر آن مورد استفاده وسیع طراحان و مهندسین الکترونیک و کامپیوتر نیز قرار دارد. مباحث این کتاب و سطح مطالب آن به گونه‌ای است که هم می‌تواند برای دانشجویان دوره کارشناسی و کارشناسی ارشد مورد استفاده قرار گیرد و هم برای طراحانی که می‌خواهند از پدیده‌های نوین برنامه‌نویسی در طراحی‌های خود بهره جویند مفید باشد.

این کتاب برگردان کتاب VHDL for Designers تألیف Stefan Sjoholm و Linnart Lindh است که با توجه به ارزش آموزشی آن تحت عنوان «آموزش VHDL» ترجمه شده است.

در خاتمه لازم به یادآوری است که کتابهای برنامه‌نویسی حاوی یک سلسله عبارات و اصطلاحات تخصصی است که در فارسی تا به حال معادل‌های گویا و رسایی برای آنها یافت نشده و

---

1. Hardware Description Language

همین امر ترجمه این کتب را نه تنها مشکل بلکه از لحاظ بیان مفاهیم و معانی واژگان تخصصی بعضاً با نارسایی‌هایی مواجه ساخته است. گاه معادل‌های فارسی جدید ساخته شده و گاه نیز چاره‌ای جز استفاده از اصطلاحات به زبان اصلی نبوده است. با این حال این مشکلات نباید مانع از تلاش بیشتر برای رساتر کردن زبان فارسی در بیان مفاهیم علمی گردد. امید است که ترجمه این کتاب برای دانشجویان و طراحان مفید باشد و حداقل به عنوان گام کوچکی آنان را در مسیر اهداف علمی و عملی‌شان یاری رساند. واضح است که کارهایی از این دست خالی از عیب و نقص نیستند و بنابراین خوانندگان با ارسال نظرات خود مترجم را سپاسگزار خواهند کرد.

فرزان گیتی

مقدمه نویسندگان

کتاب حاضر سومین کتاب ما درباره VHDL است. این کتاب برای آنکه بخشهای جدید VHDL را دربرگیرد (از جمله نسخه استاندارد VHDL-93) بیشتر از حد متعارف طولانی شده و به جزئیات مفصل‌تری پرداخته است. اما موجب خوشوقتی ماست که این کتاب اینک در دانشگاهها و کالج‌های مختلف مورد استفاده قرار گرفته و برنامه‌ریزان و مهندسین این صنعت آن را مطالعه کرده و مورد بهره‌برداری قرار می‌دهند.

هدف این کتاب، آموزش VHDL و طرز استفاده از آن در عمل برای طراحی سیستمهای الکترونیکی به کمک ابزارهای پیشرفته نوین و امروزی است. در این کتاب نحوه ساختن محیطهای آزمایش نیز مورد بررسی قرار گرفته است.

مثالهای فراوانی که در هر مبحث آورده شده این امکان را به کتاب حاضر می‌دهد که هم به عنوان یک کتاب درسی در سطوح مختلف و هم در مرحله بعد به عنوان یک کتاب مرجع مورد استفاده قرار گیرد. سعی ما بر این است که مباحث کتاب، کوتاه و در عین حال دقیق و گویا باشد.

اینک چند سالی است که از VHDL برای طراحی استفاده می‌شود و سعی این کتاب بر این است که امکاناتی را که این زبان تا به امروز قادر به عرضه آن شده، منعکس نماید. ابزارهای کمکی، شیوه‌ها و روشهای توسعه، ابزارهای سنتز و به موازات آنها روشهای تست و آزمایش به سرعت در حال تغییر و پیشرفت هستند. به عنوان مثال، سنتز رفتاری یکی از آن حوزه‌هایی است که در آینده اهمیت بیشتری پیدا خواهد کرد و از این رو است که این کتاب مقدمه‌ای را در موضوع سنتز رفتاری و امکانات و توانایی‌های آن ارائه می‌دهد. در نظر داریم که در چند سال آینده این کتاب را در سایه پیشرفتهای فنی جدیدی که پدید می‌آید به روز کنیم.

VHDL زبانی تقریباً جدید است و ساختار آن از بعضی لحاظ مانند سایر زبانهای معروف مثل C و پاسکال است، ولی رفتار و عمل آن کاملاً با سایر زبانها فرق دارد. زبانهای C و پاسکال با CPU، یعنی ماشین سریالی که دستورات را در هر نوبت یکی به اجرا می‌گذارد، دمساز هستند در حالی که VHDL با ساختارهای کلی سخت‌افزار هماهنگی کامل دارد. ساختارهای سخت‌افزار کلاً حالت موازی دارند و در نتیجه عملکرد VHDL در چهارچوب این ساختارها همیشه بهتر و بالاتر از زبانهای برنامه‌نویسی مرسوم برای CPUهاست. این بدان معناست که VHDL به عنوان یک زبان اجرایی کامل برای جانشینی زبانهای نرم‌افزاری روز به روز کاربرد بیشتری پیدا می‌کند. باور ما بر این است که در آینده اکثر میکروکنترلرهای کوچک به جای طراحی شدن با CPU و زبانهای ماشینی، با کد VHDL طراحی خواهند شد.

در چند سال اخیر شبیه‌سازهایی به بازار عرضه شده‌اند که می‌توانند رفتار توصیف شده توسط یک کد VHDL را به اجرا درآورند. کد VHDL می‌تواند به طور اتوماتیک در داخل FPGA (Field Programmable Gate Array) یا ماتریسهای گیتی پیاده‌سازی شود. یکی از شرطهای ضروری برای انجام این عمل آن است که ابزار سنتز باید بتواند سخت‌افزاری را به وجود آورد که همان رفتار و عملکرد کد VHDL را که در ابتدا شبیه‌سازی شده بود دارا باشد. از آنجا که توصیف کلی یک طرح در قالب یک کد VHDL کار دشواری است بنابراین می‌توانیم بخشهای مختلف طرح را با کدهای VHDL در سطوح مختلف توصیف کنیم و در نتیجه باید از ابزارهای سنتز متناسب با هر بخش استفاده نماییم. به همین دلیل کتاب حاضر مثالهایی از سه نوع ابزار مختلف سنتز را ارائه می‌دهد که عبارتند از: Synopsys، Autologic (Mentor Graphics) و ViewLogic. سیستم ViewLogic یک‌سیستم وابسته به PC و یا کارگاه (workstation) است در حالی که Synopsys و Autologic هر دو سیستمهایی وابسته به کارگاه هستند.

برای بسیاری از طراحان سخت‌افزار، توصیف رفتار با زبان VHDL به جای گیت، مستلزم ایجاد تغییراتی در سیستم به عنوان گام اول است. اگر ساخت طرحی با بهره‌وری و کیفیت بالا موردنظر باشد انجام این تغییرات بسیار ضروری است. استفاده از یک زبان سطح بالا و غیروابسته به نوع تکنولوژی مورد مصرف برای تعریف و توصیف یک پروژه ـ یعنی استفاده از زبان VHDL ـ این امکان را به طراح می‌دهد که توجه خود را روی طرحی متمرکز سازد که کارآیی و عملکرد صحیح داشته باشد. طراحی با VHDL در مقایسه با طراحی در سطح گیتی از کارآیی و استواری بسیار بالاتری برخوردار است. همچنین کاربرد VHDL همراه با یک ابزار سنتز به این معناست که تکنولوژی مورد استفاده را می‌توان به سرعت و به سادگی تغییر داد. از این امتیاز می‌توان با کمک FPGA ها برای تولید آنچه به عنوان نمونه‌سازی سریع مدارهای ***ASIC***[1] خوانده می‌شود بهره جست.

تولید طرحی دقیق، صحیح، برخوردار از روش اصولی، آزمایش‌پذیر و غیروابسته به تکنولوژی مستلزم آگاهی دقیق از زبان VHDL است که کتاب حاضر درباره آن به نگارش درآمده است. به علاوه ما اگر بخواهیم از امتیازات ناشی از توصیف طرح به زبانی سطح بالا مانند VHDL برخوردار گردیم باید با روشهای مطلوب طراحی و روشهای مطالعه شده تست به خوبی آشنا باشیم. به این جهت است که کتاب حاضر فصلهایی را نیز به متدولوژی طراحی و تست اختصاص داده است.

در این کتاب تمام واژه‌های خاص و کلیدی زبان VHDL با حروف برجسته چاپ شده است.

بعضی از شماتیکهایی که به طور اتوماتیک توسط ابزار سنتز تولید شده‌اند نشان‌دهنده اصول نتایج حاصل از سنتز هستند، در این شماتیک‌ها خوانا بودن برچسبها مدنظر نبوده است.

1- Application Specific Integrated Circuit

اگر سؤالات یا نظراتی درباره VHDL و مطالب این کتاب داشته باشید می‌توانید با نشانی سایت ما تماس حاصل نمایید :

http://www.mdh.se/avdelningar/idt/forskning/cus/books/VHDL/
Stefan Sjoholm, ABB Industrial Systems, 721 67 Vasteras, Sweden
(E-mail : stefan.j.sjoholm@seisy.mail.abb.com) or
Lennart Lindh, Malardalens hogskola, IDT, Box 883, 721 23 Vasteras, Sweden
(E-mail : lennart.lindh@mdh.se).

در پایان، لازم می‌دانیم از حمایت شرکتهای Synopsys، Mentor Graphics، Motorola، Texas Instrument، Xilinx و سایرین قدردانی کنیم.

Stefan Sjoholm
Lennart Lindh

VHDL = VHSIC Hardware Description Language
VHSIC = Very High Speed Integrated Circuit

فصل اول

مقدمه و نگاه کلی

VHDL در اوایل دهه ۱۹۹۰ به وجود آمد و در حال حاضر به عنوان یکی از مهم‌ترین زبانهای استاندارد در طراحی‌های الکترونیکی پذیرفته شده است. امروزه تعداد زیادی از شرکتهای بزرگ و معتبر برای طراحی‌های سیستمهای دیجیتالی خود منحصراً از VHDL استفاده می‌کنند. در دانشگاهها واحدهای درسی VHDL ارائه می‌شود و گروههایی برای آموزش استفاده‌کنندگان VHDL تشکیل شده است. احتمالاً شرکتهای کوچک نیز تا چند سال آینده استفاده از VHDL را شروع خواهند کرد. اینک مدتی است که VHDL و طراحی‌های ASIC رقابت شدیدی را با کنترلرهای تک چیپی شروع کرده‌اند. این زبان، طراح را از اجبار در استفاده از ساختار ***وون نیومن***[1] رها می‌سازد و به او امکان می‌دهد تا به جای مکانیسمهای تسلسلی و متوالی از برنامه‌ای هم‌زمان و موازی استفاده نماید. این برنامه امکانات و قابلیتهای کاملاً جدیدی را فراروی او قرار می‌دهد.

دو دلیل عمده استفاده از VHDL به جای طراحی‌های معمولی شماتیکی، یکی کوتاه‌تر بودن مدت زمان لازم برای تولید طرحهای الکترونیکی و دیگری آسان‌تر بودن تغییر و اصلاح آنهاست.

زبان VHDL در اواخر دهه ۱۹۸۰ با قابلیت مدل‌سازی و فقط برای بیان مشخصات سیستم مطرح شد و همراه با آن اولین شبیه‌سازها شروع به رشد کردند. چند ابزار معدود نیز که با استاندارد VHDL همخوانی داشتند در پایان دهه ساخته شدند و چند سال پس از آن بود که طراحی با VHDL

1- Von Neumann

وارد صحنه شد. امروزه کلیه سازندگان ابزارهای طراحی، استاندارد VHDL را حمایت می‌کنند. در حال حاضر VHDL زبانی است استاندارد شده و با این خاصیت که می‌توان کد نوشته شده با VHDL را بدون آنکه نیازی به تغییری داشته باشد از یک محیط طراحی به محیط دیگر جابه‌جا کرد. گرچه VHDL برای طراحی مدلهای دیجیتالی استاندارد شده است لیکن متأسفانه هنوز برای طراحی (سنتز) استاندارد نشده است. در سالهای اخیر سنتز کردن بخشهای بیشتری از ساختارهای این زبان امکان‌پذیر شده است.

خوشبختانه در سالهای اخیر در تولید ابزارهای مناسب VHDL برای PC ها، خصوصاً محیطهای شبیه‌سازی، پیشرفتهای فراوانی حاصل شده است. این بدان معناست که قیمتها شدیداً کاهش یافته و امکان استفاده شرکتهای کوچک از VHDL فراهم شده است. همچنین ابزارهای تحلیل‌کننده کامپیوتری مخصوصاً برای FPGA ها و CPLD ها پدید آمده که البته عملکرد آنها قدری محدودتر از عملکرد ابزارهای کارگاهی (workstation) است.

رویداد جدیدی که دنیای طراحی الکترونیک را دگرگون کرده است روشی است که به وسیله آن می‌توان هزاران گیت و فلیپ فلاپ را توسط یک PC در ظرف چند دقیقه در یک تراشه برنامه‌ریزی کرد بی‌آنکه نیازی به استفاده از دستگاههای گران‌قیمت باشد. به عبارت دیگر امروز این توانایی را یافته‌ایم که یک فایل را به طور اتوماتیک از VHDL به دست آورده و از آن برای برنامه‌ریزی کردن تراشه‌ها استفاده کنیم. به این روش ***نمونه‌سازی سریع اولیه***[1] می‌گویند. در حال حاضر مدارهایی هستند که دارای ۱۰۰۰۰۰ گیت در یک مدار واحد هستند و با همین مکانیسم ساخته می‌شوند. از این روش برای تولیدهای کوچک‌تر نیز می‌توان استفاده کرد.

نوشتن کد VHDL به جای به کارگیری المانهای شماتیکی (مثل گیتها) یک روش جدید طراحی است. کار با VHDL صرفاً به معنای نوشتن یک برنامه نیست بلکه تسهیلاتی است که VHDL برای تقسیم طرح به بخشهای کوچک‌تر و تقسیم بخشها به زیربخشها به صورت طراحی سلسله مراتبی و ایجاد کتابخانه‌ای از اجزا برای استفاده در طراحی ارائه می‌دهد. همچنین می‌توان از VHDL برای نوشتن کد مدارات و تراشه‌های استاندارد (مثل Motorola 680020) استفاده کرد. در حال حاضر شرکتهایی هستند که برای محیطهای مختلف شبیه‌سازی اجزای ترکیبی تولید می‌کنند و این بدان معناست که مدلهای تمام بوردها با اجزای استاندارد را می‌توان به طریقه شبیه‌سازی کامپیوتری مورد تست و تأیید قرار داد. وقتی کل بورد شبیه‌سازی می‌شود، به طور مثال زمانهای دسترسی به داده‌ها و آدرسهای غلط سریعاً کشف می‌شوند که این کار اغلب با بررسی کردن سیگنال‌های واسط بین اجزای

1- Rapid Prototyping

مختلف مدار و زمان رسیدن این سیگنال‌ها و غیره انجام می‌شود.

طراحی با VHDL مستلزم آن است که طراح کد را نوشته و سپس صحت عمل آن را در یک شبیه‌ساز تست کند و بعد از این مراحل netlist حاصل می‌شود. سنتز را می‌توان با یک کامپایلر مقایسه کرد که یک برنامه را به کد ماشین ترجمه می‌کند. در سخت‌افزار، کد VHDL به شماتیکی حاوی گیت‌ها و فلیپ فلاپ‌ها ترجمه می‌شود.

در اوایل دهه ۱۹۸۰ میلادی، نخستین بار وزارت دفاع آمریکا بود که به تولید نسخه اولیه VHDL اقدام نمود، زیرا در آن زمان ارتش آمریکا برای توصیف سیستمهای الکترونیکی خود به یک روش استاندارد نیاز داشت. در سال ۱۹۸۷ زبان VHDL توسط ***IEEE***[1] استاندارد شد و در دسامبر همان سال تحت عنوان IEEE 1076-1987 به ***تصویب رسید***[2]. لازم به یادآوری است که نسخه اولیه VHDL تنها برای بیان مشخصه‌های سیستمهای الکترونیکی و نه برای طراحی استاندارد شده بود.

VHDL شباهت فراوانی به زبان نرم‌افزاری ADA دارد، زیرا ***شرکتی***[3] که از جانب پنتاگون مأموریت تولید زبان جدید را یافته بود تجربه‌های زیادی در زمینه زبان ADA داشت. استاندارد VHDL نیز مثل دیگر استانداردهای IEEE هر پنج سال یک بار مورد تجدیدنظر واقع می‌شود. آخرین نسخه استاندارد، VHDL-93 نامیده می‌شود که قدری با تأخیر در سال ۱۹۹۳ به وجود آمد. این نسخه جدید تفاوتهای زیادی با نسخه استاندارد VHDL-87 ندارد و فقط تعدادی دستورها و مشخصه‌های جدید، خصوصاً در زمینه مدل‌سازی به آن اضافه شده است. در این کتاب اضافات جدید به نسخه استاندارد قبلی با عبارت «VHDL-93» مشخص گردیده است.

برتری عمده VHDL تا حدودی ناشی از این واقعیت است که VHDL تنها زبان سخت‌افزاری است که استاندارد گردیده است. به عنوان مثال، زبان ADA برای رقابت با دیگر زبانهای برنامه‌نویسی مثل C و C^{++} دارای اشکالات زیادی است.

۱-۱ چرا از VHDL استفاده می‌شود ؟

طراحی با VHDL از امتیازات فراوانی در مقایسه با تکنیکهای مرسوم در طراحی‌های شماتیکی برخوردار است. این بخش به بررسی نقاط قوت و ضعف این زبان می‌پردازد.

1- Institute of Electrical and Electronics Engineers

۲- کتاب مرجع : IEEE VHDL Language Reference Manual Draft Standard Version 1076/B

3- Intermetrics

VHDL محیط بسیار مناسبی برای کمک به پیشرفتهای دیجیتالی است. VHDL همچنین روشهای مختلف طراحی از قبیل روش ***بالا به پایین***[1] و روش ***پایین به بالا***[2] و یا روش آمیخته را عرضه می‌نماید. بسیاری از محصولات الکترونیکی امروزه عمری بیشتر از ده سال دارند و برای آنکه بتوانند از تکنولوژیهای جدیدتر استفاده کنند نیاز به طراحی‌های مجدد دارند. ساده‌ترین راه برای این کار استفاده از VHDL غیروابسته به تکنولوژی است. این بدان معناست که می‌توان با کمک ابزارهای اتوماتیک، تکنولوژی مورد استفاده را تغییر داد. وقتی یک محصول الکترونیکی عمری برابر ده سال دارد بدیهی است که سیستم آن هر از چند گاهی نیاز به تغییر داشته و باید وظایف جدیدی به آن اضافه شود. از آنجایی که VHDL زبانی است با خوانایی بالا، ساختار یافته و حمایت‌کننده طراحی سلسله مراتبی، این تغییرپذیری را به راحتی قبول می‌کند.

با کمک این زبان می‌توان سیستم را به فرم سلسله مراتبی (بلوک دیاگرامی) طراحی کرد و از اجزا و المان‌های کتابخانه مجدداً استفاده نمود و به سادگی طرح را مورد بازبینی و بررسی قرار داد. بلوک‌های سلسله مراتبی می‌توانند با به کارگیری VHDL ساختاری، function ها و procedure ها توصیف شوند. VHDL ساختاری را می‌توان با یک بلوک دیاگرام مقایسه کرد. بسیاری از سیستمها ورودی‌های گرافیکی را که به طور اتوماتیک قابل ترجمه به VHDL ساختاری هستند قبول می‌کنند. این زبان همچنین دستورات و ساختارهای متوالی و موازی را حمایت می‌کند.

طراحی اجزای ترکیبی با VHDL می‌تواند مستقل از تکنولوژی باشد. در VHDL می‌توان اجزا را برای استفاده مجدد در طراحی‌های مختلف در کتابخانه‌ای ذخیره‌سازی نمود. همان طور که گفته شد با کمک VHDL می‌توان تراشه‌های متداول و تجاری با اجزای استاندارد را مدل‌سازی کرد، که یک امتیاز بزرگ در طراحی‌های الکترونیکی به شمار می‌رود. در ضمن این مدلها را می‌توان به گونه‌ای طراحی کرد که به راحتی بتوان آنها را تغییر داد. به طور مثال در طراحی یک المان ***FIFO***[3] می‌توان امکان تغییر کلیه احتمالات ممکن از قبیل تعداد بیت‌ها، تعداد ردیفها و غیره را تحت پوشش قرار دارد. به این المان‌ها، ***المان‌های عمومی***[4] می‌گویند.

عملکرد کد VHDL را می‌توان توسط یک شبیه‌ساز مورد آزمایش قرار داد. شبیه‌ساز با اعمال سیگنال‌های ورودی و تولید نمودار سیگنال‌های خروجی به شبیه‌سازی کد VHDL می‌پردازد. وقتی کد

1- Top-down

2- Bottom-up

3- First In First Out

4- Generic Component

VHDL شبیه‌سازی می‌شود تست عملکرد نیز به اجرا در می‌آید. در مراحل بعدی، تست زمانی طرح نیز امکان‌پذیر می‌گردد.

در طراحی‌های مرسوم شماتیکی، طراح باید فاکتورهای ویژه هر تکنولوژی مانند زمان‌بندی، حجم و فضای مورد نیاز، قدرت درایورها، انتخاب اجزا و ***جریان‌دهی خروجی***[1] را به طور دستی چک کند. یکی از امتیازات بزرگ طراحی به زبان VHDL آن است که طراح می‌تواند توجه خود را بر روی عملکرد یا به عبارت دیگر اجرای مشخصه‌های مورد نیاز متمرکز سازد و نیازی نیست که وقت و انرژی خود را صرف فاکتورهای خاص هر تکنولوژی که ربطی به عملکرد ندارند، بنماید.

با توجه به استاندارد بودن VHDL این امکان وجود دارد که کد آن را برای مدل‌سازی (شبیه‌سازی) بین سیستمهای تولیدی متفاوت جابه‌جا نمود. اما چون این استاندارد در مورد طراحی وجود ندارد جابه‌جا کردن کد VHDL کمی مشکل‌تر است. این کتاب سعی دارد نشان دهد که کدهای قابل سنتز شدن VHDL چگونه باید برای ابزارهای سنتز ViewLogic، Mentor Graphics و Synopsys که در حال حاضر در صنایع و دانشگاهها از آنها استفاده می‌شود، نوشته شود. ViewLogic یک سیستم مبتنی بر PC (و همچنین قابل استفاده در workstation ها) است و در خصوص سنتز کارآیی و عملکرد آن دو ابزار دیگر را ندارد. Synopsys و Mentor Graphics Autologic2 مبتنی بر سیستمهای workstation هستند که خیلی گران‌تر و در عین حال خیلی قوی‌تر از ابزار ViewLogic می‌باشند. در طول دهه ۱۹۹۰ شرکت Synopsys شرکت پیشرو در سنتز VHDL بود اما امروزه در زمینه تولید ابزار سنتز با رقابت شدیدی از جانب AutoLogic2 روبه‌رو شده است. خوشبختانه هم Synopsys و هم Autologic2 با تقریباً تمام زیربخشهای VHDL (تا حدود ۹۹ درصد) سازش دارند و از این رو VHDL می‌تواند بین این دو ابزار پیشرو بدون آنکه نیاز به دستکاری و تغییر داشته باشد، جابه‌جا شود.

چون در این کتاب طرز نوشتن کد VHDL برای تمام موارد، از ابزار ساده سنتز برای کامپیوترهای شخصی گرفته تا پیشرفته‌ترین ابزارهای سنتز برای workstation شرح داده شده است خواننده در به کارگیری قالبهای سنتز ارائه شده در این کتاب در محیطی غیر از محیطهای سه گانه فوق هیچ گونه مشکلی نخواهد داشت.

تاکنون VHDL برای الکترونیک آنالوگ استاندارد نشده است. گرچه تلاشهای بسیاری برای ایجاد یک شاخه آنالوگ (AHDL) برای توصیف سیستمهای آنالوگ انجام گرفته است. این نسخه جدید شاخه آنالوگ استانداردی کاملاً بر پایه استاندارد VHDL است و دارای فاکتورها و معیارهایی اضافی

1- Fan-out

برای توصیف عملکردهای آنالوگ خواهد بود.

۲-۱ پروسه تولید

این بخش به تشریح روشهای کلی در پروسه تولید می‌پردازد. از این روشها می‌توان برای پروژه‌های کوچک دانشجویی و تجربه‌های آزمایشگاهی نیز استفاده نمود.

در این مرحله، محصول توصیف، طراحی و تست می‌شود. یک مدل معمول تولید، مدل ***آبشاری***[1] نام دارد. این مدل با ذکر مشخصه‌ها آغاز و طی مراحل تعریف شده‌ای به تولید یک نمونه اولیه منتهی می‌شود. مدل آبشاری یک مدل ایده‌آل برای بیان مشخصه‌ها و تجربه‌های آزمایشگاهی است.

جریان تولید از مرحله توصیف مشخصه‌ها تا تولید نمونه اولیه را می‌توان به نحو زیر به چندین مرحله تقسیم نمود (شکل ۱-۱ را ببینید).

فاز تولید	نتیجه	نکات
تجزیه و تحلیل و آنالیز	مشخصات	مراحل کارهایی که باید انجام شود چیست ؟
طراحی	کد VHDL	نحوه انجام کارها به چه صورتی است ؟
نگاشت با تکنولوژی	netlist	
ساخت نمونه اولیه	نمونه اولیه	نتیجه به دست آمده چیست ؟

شکل ۱-۱ نگاه کلی به پروسه تولید

- فاز آنالیز عبارت است از نوشتن مشخصه‌ها. این مشخصه‌ها را می‌توان با VHDL یا با زبان معمولی نوشت. منظور از مشخصه‌ها آن است که بی ببریم چه کارهایی باید انجام گیرد. مشخصه‌ها را می‌توان در VHDL توصیف نمود و سپس با یک شبیه‌ساز VHDL مورد آزمایش قرار داد.
- فاز طراحی یعنی تبدیل مشخصه‌ها به یک معماری و کد VHDL. انجام این کار تاکنون به طور اتوماتیک عملی نشده است. این فاز با تعریف معماری (بلوک دیاگرام) آغاز می‌شود. وقتی معماری آماده شد، کد VHDL را برای اجزای مختلف (بلوک‌ها) می‌نویسیم و یا از اجزای حاضری داخل ذخائر کتابخانه نسخه‌برداری می‌کنیم. آنگاه عملکرد طراحی در یک شبیه‌ساز

1- Waterfall

مورد آزمایش قرار می‌گیرد. وقتی نتیجه حاصل با مشخصه‌ها تطبیق داشته باشد، طراح می‌تواند به فاز بعدی برود. در این فاز مسأله اصلی نحوه طراحی معماری و اجزا می‌باشد.

- مرحله بعدی، نگاشت تکنولوژی است. عواملی که در انتخاب نوع تکنولوژی دخالت دارند عبارتند از قیمت، کیفیت کار، فراهم بودن و امثال آن. این فاز تا حد زیادی به طور اتوماتیک انجام می‌گیرد. محدودیتهای زمانی با فرمتی توصیف می‌شوند که توسط ابزار سنتز قابل خواندن باشند. اگر ابزار سنتز نتواند شروط زمانی را برآورده سازد عملیات فاز طراحی باید دوباره به طور کامل و یا در بخشهایی از آن تکرار شود. سنتز تأیید شده netlist (شماتیک) وابسته به تکنولوژی را تولید می‌کند. این شماتیک به صورت یک فایل ورودی برای سایر ابزارهاست. حال برنامه‌ریزی یک FPGA فقط چند دقیقه وقت لازم دارد. اما اگر در یک طراحی فرضاً آرایه‌ای از گیت‌ها باید به‌کار گرفته شود، تهیه یک مدار کامل تا چندین روز طول خواهد کشید. در این‌صورت بردارهای تست نیز باید در طراحی منظور گردند. مدارهای FPGA قبلاً در کارگاههای تولید تست می‌شوند.
- بعد از این مرحله، نمونه اولیه ساخته می‌شود و با مشخصه‌های طراحی مقایسه می‌گردد. اگر نتیجه با مشخصه‌ها یکسان باشد مدار آماده بهره‌برداری است.

برنامه تحریری برای یک پروژه دانشجویی به شکل زیر است :

۱) خلاصه

۲) مقدمه

۲-۱) تحریر پروژه

جدول زمانی

تاریخچه

۳) مشخصه‌های طرح

وظایف و شروط زمانی

۴) توصیف وظایف

تعریف معماری

اجزای I/O

توصیف اجزا (VHDL)

۵) نتیجه سنتز

تعداد گیت‌ها

۶) تست نمونه اولیه

طرح‌ریزی[1] پایه‌های تراشه

۷) نتیجه‌گیری

منابع و مراجع

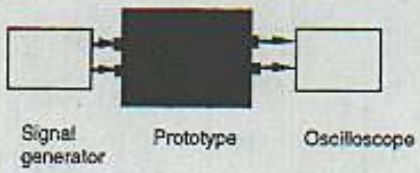
ضمائم :

- مشخصه‌های تعیین شده همراه با نتایج شبیه‌سازی
- محیط آزمایش
- توصیف اجزا با VHDL
- نتایج شبیه‌سازی
- شروط زمانی سنتز
- پروتکل تست

۳-۱ تاریخچه

تاریخچه طراحی‌های الکترونیک موضوع بسیار جالبی است. بزرگ‌ترین انقلابی که تاکنون شاهد آن بوده‌ایم از بزرگ‌تر شدن مدارهای مجتمع و ارزان‌تر شدن آنها منشأ گرفته است. البته به موازات آن ابزارهای جدیدی نیز با شتاب افزاینده‌ای تولید شده و مورد استفاده قرار گرفته‌اند.

اگر ده سال به عقب برگردیم، برای آزمایش یک طرح، یک نمونه اولیه فیزیکی درست می‌کردند (معمولاً به صورت سیم‌بندی بر روی بورد). در آن زمان برای تولید سیگنال‌های ورودی از یک ***مولد سیگنال***[2] و از یک اسیلوسکوپ به ترتیب برای تولید سیگنال‌های ورودی و مشاهده سیگنال‌های خروجی استفاده می‌شد (شکل ۲-۱ را ببینید). نمونه اصلی اولیه مانند جعبه سیاهی بود با سیگنال‌های ورودی/خروجی که بر روی صفحه اسیلوسکوپ مشاهده و کنترل می‌شد.



شکل ۲-۱ تست نمونه اولیه به شکل جعبه سیاه

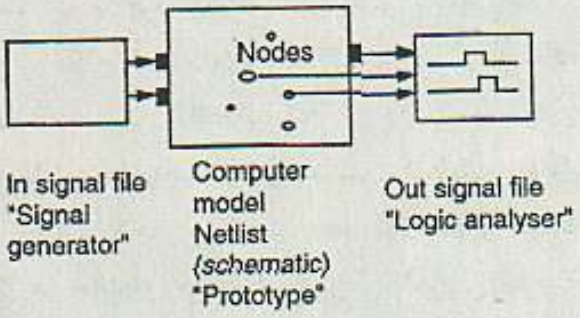
1- Layout

2- Signal Generator

اشکال در این بود که امکان مشاهده ده شکل موج غیرمتناوب بر روی اسیلوسکوپ وجود نداشت. رفع این نقص زمانی ممکن شد که تحلیل‌گرهای منطقی ساخته شدند و ذخیره‌سازی شکل موجها را از چند سیگنال خروجی امکان‌پذیر ساختند.

آنگاه پیشرفتهای بیشتر موجب پیدایش نمونه‌های اولیه پیچیده‌تر (معروف به جعبه‌های سیاه) گردید. اما این پدیده خود مشکل تست کردن گره‌های داخلی تراشه را به همراه داشت. تولید ابزارهای بعدی کامپیوتری راه حل این مشکل را نیز فراهم ساخت. در این نرم‌افزارها مدلهای مدارات تجمع مختلف، منبع تولید سیگنال و تحلیل منطقی گره‌ها پیش‌بینی شده بود. در کامپیوتر، به مولد تولید سیگنال و اسیلوسکوپ شبیه‌سازی می‌گویند (شکل ۳-۱). در ساخت مدل اولیه از علائم خاصی استفاده می‌شد. آنگاه کامپیوتر می‌توانست مدل را با تأخیرهای زمانی تعریف شده برای هر گیت شبیه‌سازی نماید.

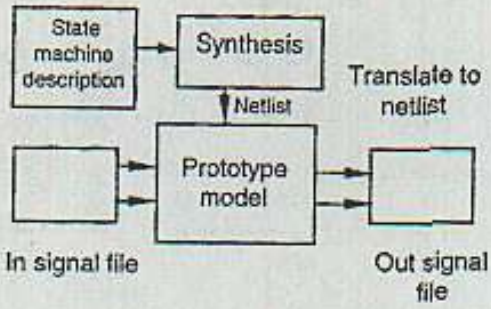
اینک این امکان به‌وجود آمده بود که تمام اطلاعات گرفته شده از هر گره را به منظور انجام یک شبیه‌سازی کامل ذخیره نمود. اطلاعات در یک پایگاه زمانی از زمان شروع (زمان صفر) تا زمان اتمام ذخیره می‌شد. ابتدا اطلاعات سیگنال فقط شامل '1' یا '0' بودند. بعداً مقادیر 'X' (نامعین) و 'Z' (امپدانس بالا) به آن اضافه شد. همچنین انجام شبیه‌سازی در دماهای مختلف و کنترل اینکه آیا طراحی در تمام گستره دمایی و برای کلیه ابزارها به خوبی عمل نموده یا خیر امکان‌پذیر شد.



شکل ۳-۱ تست شماتیک در کامپیوتر

طراحی توابع بولی به طریقه شماتیک بسیار وقت‌گیر بود. طراح باید خلاصه و ساده‌سازی شبکه‌های گیتی را به طور دستی انجام می‌داد. این مشکل از طریق تولید ابزارهایی برای ترجمه اتوماتیک یک فایل نوشتاری که حاوی معادلات بولی به صورت شماتیک گیتی (شامل الگوریتم‌های ساده‌سازی) باشد حل گردید. این ابزارها اولین نسل ابزارهای سنتز بودند و پروسه آنها تحت نام سنتز منطقی نامیده شده است (شکل ۴-۱). نخستین ابزارهای سنتز منطقی همراه با مدارهای ساده برنامه‌پذیر (فقط گیت‌ها) ارائه شدند. آنگاه مدارهای برنامه‌پذیر که دربردارنده فلیپ فلاپ‌ها بودند نیز

پدیدار گشتند. این مدارها به نام ***PLD***[1] خوانده می‌شدند. با کمک PLDها بود که پیاده‌سازی ***ماشینهای حالت***[2] امکان‌پذیر گردید. اولین برنامه‌های سنتز که مشخصه‌های حالت را به خانواده‌ای از مدارهای خاص ترجمه و تبدیل می‌کردند وارد بازار شدند (به فصل ۹، «ماشینهای حالت» مراجعه نمایید).



شکل ۴-۱ سنتز یک ماشین حالت به گیت‌ها و فلیپ فلاپ‌ها

در این زمان، به وجود آمدن و از بین رفتن سریع خانواده‌ای از مدارات، طراحان را با مشکلات جدیدی مواجه ساخت. اولین مشکل آن بود که وقتی یک مدار از بازار خارج می‌شد طراح مجبور بود به خاطر آنکه هر ابزار قواعد مخصوص خود را داشت، تمام مشخصه‌های کد را از نو بازنویسی کند. سپس ابزارهای جدیدی ظاهر شدند که با چند خانواده از مدارهای مختلف سازگاری داشتند. هر ابزار زبان خاص خود را داشت. از این رو طراح وابسته به ابزار می‌شد اما چندین خانواده از مدارهای مختلف را در اختیار داشت تا از میان آنها مناسب‌ترین را انتخاب کند.

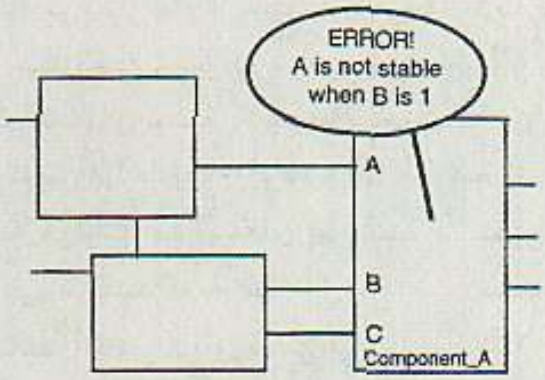
به همان نسبت که طراحی‌ها پیچیده‌تر می‌شد تست عملیات و وظایف نیز که قبلاً به سادگی با نگاه کردن به دیاگرام‌های پالس انجام می‌گرفت، به‌طور افزاینده‌ای سخت‌تر شد. چک کردن تمام زمانهای تنظیم شده، پروتکل‌ها، حالتهای خطا و غیره در مورد هزاران جزء، کاری بسیار وقت‌گیر و غالباً ناموفق بود. بدین جهت برای دستیابی به یک مدار بدون نقص، دست کم یک طراحی مجدد اجتناب ناپذیر می‌شد. امروزه، با توجه به اینکه اکثر طراحی‌ها در همان نوبت اول به طور کامل عمل می‌کنند عکس آن قضیه مصداق پیدا کرده است. از این گذشته، طراحی مجدد برای تغییر دادن ابزارها خود باعث بروز مشکلات عدیده‌ای می‌گردید.

از آنجا که با VHDL امکان سنتز کردن عناصر اصلی زبان پدید آمد، این مسائل کلاً حل گردید. از سوی دیگر هنوز مشکل جابه‌جایی کد VHDL بین تمام سیستمهای سنتز رفع نشده ولی تا

1- Programmable Logic Device

2- State Machine

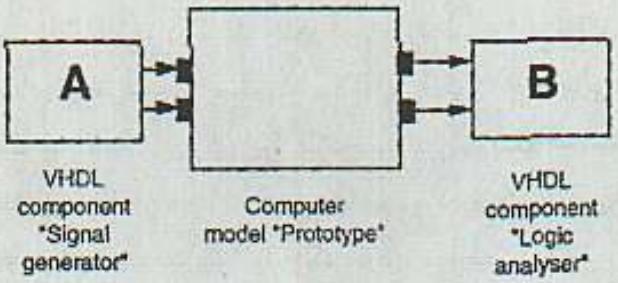
حدودی کاهش یافته است. VHDL برای سنتز هنوز استاندارد نشده است. اگر طراح بخواهد از یکی از زیربخشهای VHDL استفاده کند ممکن است کد VHDL قابلیت جابه‌جایی بین چند ابزار سنتز را داشته باشد اما باید دانست که تضمینی برای این کار وجود ندارد. تست صحت عمل با کد VHDL نیز امکان ساخت سیستم کنترل خطا در داخل اجزا را فراهم آورده است (شکل ۵-۱ را ببینید). به طور مثال یک RAM می‌تواند زمانهای دستیابی را چک کرده و مطمئن شود که دستورات READ و WRITE به طور هم‌زمان فعال نشوند. اگر سیگنالی این الزامات را بشکند یک پیام خطا بر روی شبیه‌ساز ظاهر می‌گردد. به این ترتیب مشاهده می‌کنیم که اجزا شروع به «حرف زدن» کرده‌اند و می‌توانند اعلام نمایند که چه قسمتهایی در مرحله تست و تأیید پروژه کار نمی‌کنند.



شکل ۵-۱ مدیریت خطا در یک جزء ترکیبی

وقتی زبان استاندارد به وجود آمد، کمپانی‌های جدید این امکان را یافتند که اجزایی را که می‌توانستند شبیه‌سازی شوند و توسط تولیدکنندگان عمده روز مورد استفاده قرار گیرند به بازار عرضه نمایند. این اجزا دارای کیفیت بالایی بوده و از مدیریت خطای پیشرفته‌ای نیز برخوردارند. این اجزای تجاری VHDL فقط برای شبیه‌سازی و نه برای سنتز تولید شده‌اند.

همان طور که اشاره کردیم، چندین زبان وجود دارند که مشابه VHDL هستند. تفاوت آنها در این است که VHDL استاندارد شده و به یک زبان مترقی و پیشرو، هم در عالم تجارت و هم در عالم علم تبدیل گردیده است. VHDL در ابتدا برای نوشتن مشخصه‌ها (مدل‌سازی) و ساخت مدلهای تست (شبیه‌سازی) به وجود آمد ولی به زودی معلوم شد که بخشهایی از زبان VHDL برای طراحی نیز قابلیتهای خوبی دارند.



شکل ۶-۱ بررسی صحت عمل با کمک محیط آزمایش (B و A) در کامپیوتر

در حال حاضر برای تست کردن سیستمهای فرعی، روشهایی پیدا شده که از آنچه به نام محیط آزمایش معروف است، استفاده می‌کنند. ساختار محیط آزمایش به صورت گرافیکی در بخشهای A و B شکل ۶-۱ نشان داده شده است. نیازی نیست که طراحی (نمونه اولیه) با زبان VHDL بیان شود بلکه می‌تواند به صورت یک شبکه شماتیکی توصیف گردد. اما برنامه تست با زبان VHDL نوشته می‌شود با این امتیاز که می‌توان آن را ما بین ابزارها جابه‌جا کرد (به فصل ۸، «محیط آزمایش» مراجعه نمایید).

شبیه‌سازهای آمیخته می‌توانند مشخصه‌های بیان شده در سطوح مختلف را با هم بیامیزند. شبیه‌سازها می‌توانند به طریقی کار کنند که به طور مثال برای تست مشخصه‌ها، طراحی را در بالاترین سطح رفتاری مدل‌سازی نمایند. آنگاه بخشهای معینی نیز در سطح ***RT***[1] (انتقال رجیستری) طراحی می‌شوند. این امکان وجود دارد که بر روی تمام شبیه‌سازهای VHDL، شبیه‌سازی را هم در سطح RT و هم در سطح رفتاری انجام داد. بعضی از شبیه‌سازها از جمله شبیه‌سازهای ViewLogic و MentorGraphics نیز می‌توانند شبیه‌سازی را به صورت آمیخته‌ای از سطوح رفتاری، RT و گیتی (با تأخیرهای مربوط به سیم‌بندی‌های داخلی) انجام دهند.

به موازات بزرگ‌تر شدن مداوم طراحی‌ها، زمان شبیه‌سازی نیز به شدت افزایش یافته است. راه حل این مشکل استفاده از شتاب‌دهنده‌های سخت‌افزاری است. این واحدها که برای همین منظور طراحی شده‌اند قادرند صد مرتبه سریع‌تر از ابزارهای عادی کامپیوتری شبیه‌سازی نمایند. متأسفانه قیمتهای آنها نیز صد برابر از ابزارهای معمولی گران‌تر است.

ابزارهایی برای تست و تأیید صحت عملکرد سیستمهای الکترونیکی به بازار عرضه شده است. با استفاده از این تست می‌توان پی برد که آیا سیستم در «بن‌بست» قرار گرفته است یا خیر. این عبارت از ترم‌افزار گرفته شده و اینک به همان اندازه در سخت‌افزار نیز کارآیی دارد. روشهای تست نیز می‌توانند برای تأیید اینکه آیا دو مشخصه توصیف شده عین هم هستند یا خیر مورد استفاده قرار گیرند (فصل

1- Register Transfer

۱۱ را ببینید، «آشنایی با روشهای طراحی»).

امروزه ابزارهایی به وجود آمده که با آنها می‌توان به طور اتوماتیک طرحهای قابل تست را تولید نمود. به طور مثال این ابزار می‌تواند یک شیفت رجیستر را با محتویات مشخص بر روی طرح قرار داده تا بتواند با ورودی‌های مشخص، تراشه موردنظر را تست کند. این ابزار همچنین می‌تواند یک فایل تست به وجود آورد، که بعداً بتوان آن را در کارخانه آی‌سی‌سازی بر روی یک آزمایشگر بارگذاری نمود (به فصل ۱۲، «آشنایی با روشهای آزمایش» مراجعه نمایید). متأسفانه استانداردی برای فایل‌های تست وجود ندارد.

٤-١ سنتز

استفاده از نرم‌افزارهای سنتز به این معناست که طراح مجبور نباشد شخصاً به ترجمه، خلاصه و ساده‌سازی و برآورده‌کردن شروط زمانی در کد VHDL بپردازد. سنتز برای سطوح مختلفی که در زیر ذکر شده تعریف گردیده است :

- **سنتز منطقی** : توابع بولی را ساده کرده و به گیت تبدیل می‌کند.
- **سنتز RTL** : همان کارهای سنتز منطقی را انجام می‌دهد و به علاوه ساختارهای متوالی زبان را به گیت‌ها و فلیپ فلاپ‌ها ترجمه می‌کند.
- **سنتز رفتاری** : این نوع سنتز قادر است یک جزء سخت‌افزاری را در بیش از یک ساختار متوالی- موازی زبان مجدداً مورد استفاده قرار دهد.

قبلاً سنتز منطقی را توصیف کرده‌ایم. حال به بررسی سنتز RLT می‌پردازیم. مثال ساده‌ای که در زیر آورده شده اصول اولیه کار ابزار سنتز را نشان می‌دهد.

کد VHDL زیر مثالی است از آنچه که می‌تواند سنتز شود.

```
process(del,a,b)
begin
    if sel= '1' then
        c<=b;
    else
        c<=a;
    end if;
end process;
```

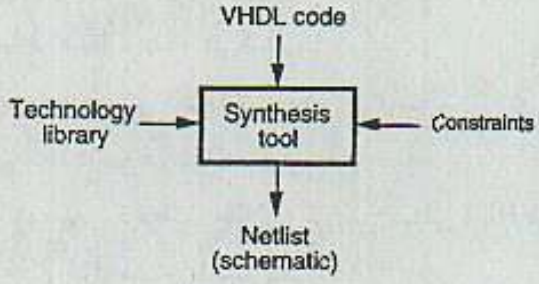
کد VHDL غیروابسته به نوع تکنولوژی مورد استفاده است. خواندن کد VHDL این مثال

خیلی ساده است : اگر sel برابر '1' باشد سیگنال c مقدار سیگنال b را می‌گیرد. در غیر این صورت سیگنال c مقدار a را خواهد گرفت. کد VHDL به عنوان داده ورودی به ابزار سنتز به کار گرفته شده است (شکل ۷-۱).

محدودیتهای زمانی در فایلی مشخص می‌گردند که در عین حال به منزله ورودی دیگر ابزار سنتز است. کتابخانه تکنولوژی، نوع تکنولوژیی که کد VHDL باید به آن ترجمه شود را توصیف می‌کند. مثال زیر یک دستور ساده برای ترجمه یک فایل VHDL به یک شماتیک حاوی گیت‌ها و فلیپ فلاپ‌ها را نشان می‌دهد (ابزار سنتز ViewLogic) :

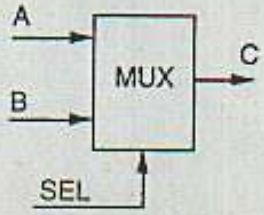
```
VHDLES VHDL _FIL -TEC=XC4000,
```

تکنولوژی XC4000 اطلاعاتی را که مورد نیاز ابزار سنتز است به آن می‌رساند. مشخصاتی از قبیل مقدار نیروی درایورها، سرعت عملکرد و اندازه حجمی آنها به میلی‌متر. تکنولوژی فوق از خانواده Xilinx 4000 است. سایر کتابخانه‌هایی که قابل استفاده هستند عبارتند از : Actel و XC3000 (خانواده Xilinx3000).



شکل ۷-۱ سنتز

نتیجه کد VHDL بعد از سنتز، یک مالتی‌پلکسر با دو ورودی خواهد بود (شکل ۸-۱). در این مثال کد VHDL ترجمه، خلاصه و ساده‌سازی شده است. طراحی نیز وابسته به تکنولوژی شده است. اگر از یک تکنولوژی دیگر استفاده می‌شد ممکن بود تعداد گیت‌ها فرق کند.



شکل ۸-۱ نتیجه سنتز

مثال دیگری که سنتز آن می‌تواند در یک دقیقه انجام گیرد چنین است :

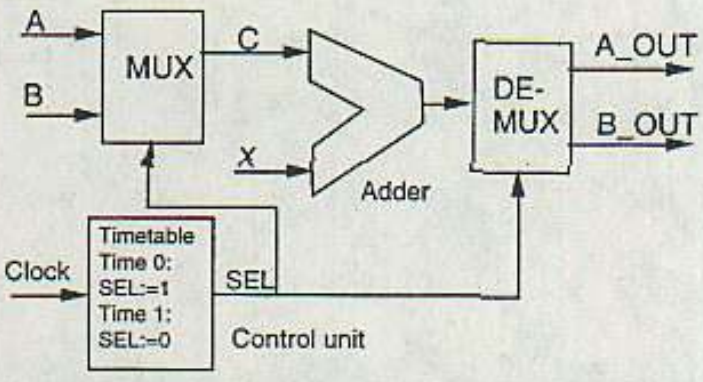
```
a:= b+c;
if a > 32 then
        count <= count +1;
end if;
```

مثال فوق بعضی امتیازات VHDL را نشان می‌دهد. این کد می‌تواند در مدت کمتر از یک دقیقه به تکنولوژی مناسب ترجمه شود. این کتاب همچنین سایر جنبه‌های زبان VHDL را که بر جذابیتهای آن خواهد افزود شرح می‌دهد.

وقتی طراحی‌های بزرگ انجام می‌شود، در بعضی جاها استفاده از توابع تکراری پیش خواهد آمد، مانند جمعگرها، ضرب‌کننده‌ها **مرتب‌کننده‌ها**[1] و الگوریتم‌ها. معمولاً در نرم‌افزارها کد را برای یک تابع بیش از یک بار نمی‌نویسند (procedure یا function) و توابع محاسباتی در کد ماشینی تنها یک واحد ALU را مصرف می‌کنند.

در سخت‌افزار عمل مشترک‌سازی توابع یکسان بین چندین کاربر قدری مشکل‌تر است. لیکن ابزارهایی در حال پیدایش هستند که این مشکل را حل خواهند کرد. امروزه کاربرها اکثراً دارای یک الگوریتم و توابع مشترک شامل جمعگرها و ضرب‌کننده‌ها هستند.

پروسه مشترک‌سازی به جای داشتن واحدهای (توابع) سخت‌افزاری تکراری، سنتز رفتاری نامیده می‌شود. سنتز رفتاری توأم با جداول زمانی دقیق کار می‌کند و این جدول‌های زمانی هستند که تعیین می‌کنند کدام کاربر باید اجزای مشترک شده را استفاده کند.



شکل ۹-۱ سنتز رفتاری

1- Sorters

شکل ۹-۱ نشان می‌دهد که چگونه یک جمعگر بین A و B مشترک شده است. کار واحد کنترل آن است که مطمئن شود که فقط یک کاربر در هر نوبت به جمعگر متصل گردیده است. حال به جای یک جمعگر ممکن است یک الگوریتم پیچیده در میان چند کاربر مختلف مشترک شود.

پیشرفتهای آتی ممکن است این امکان را فراهم آورد که بلوک‌های عملکردی بزرگی توسط چند کاربر مورد استفاده قرار گیرند، درست مانند کامپایلرهای امروزی که این عمل را برای نرم‌افزارها ممکن ساخته‌اند. وقتی قدرت عمل ابزارهای سنتز به پای قدرت عمل کمپایلرهای امروزی برسد طراح قادر خواهد بود از هم‌زمانی طبیعی در الکترونیک بهره جوید و به ابزاری بسیار قوی دسترسی یابد (به فصل ۱۷ را رجوع کنید، «سنتز رفتاری»).

شاید بتوان پیش‌بینی کرد که در آینده یک سیستم عامل سخت‌افزاری ساخته شود که قادر به حمایت و سازگاری با واحدها (برنامه‌ها)ی سخت‌افزاری باشد ـ این چالش بزرگی است. یک سوال مهم آن است که در این صورت تفاوت بین نرم‌افزار و سخت‌افزار چه خواهد بود.

۱-۵ تمرین

۱- مزایای VHDL در مقایسه با محیطهای شماتیکی مرسوم برای طراحی را بنویسید.

۲- چه عاملی باعث به وجود آمدن VHDL گردید ؟

۳- چرا زبانی مثل VHDL استاندارد شده است ؟

۴- عمل شبیه‌سازی یک طرح در کامپیوتر را توصیف کنید.

۵- کد VHDL زیر را به طور دستی سنتز کنید. فقط از گیت‌های NAND استفاده کنید.

```
(a) a_out <= not (a_in and b_in and c_in);
(b) if a_in = '1' then
        a_out <= '0';
    else
        a_out <= '1';
    end if;
```

۶- چنان‌چه به ابزارهای سنتز در سطح رفتاری، در سطح RT و در سطح گیت دسترسی داشته باشید، برای سنتز هر یک از برنامه‌های زیر از کدام ابزار استفاده می‌کنید ؟

برنامه		ابزار سنتز در سطح گیتی	ابزار سنتز در سطح RT	ابزار سنتز در سطح رفتاری
(۱)	a:=b and c;	گیت		
(۲)	if b='1' them c:='1'; else c:='0'; and if;			
(۳)	wait until clk='1' d<='1';			
(۴)	a<=(b*d)+(c*c);			
(۵)	wait for 1ms;			

راهنمایی برای تمرین ۶ :

(برنامه ۳) d مقدار '1' را در لبه بالارونده clk به خود می‌گیرد.

(برنامه ۴) * : ضرب

(برنامه ۵) برنامه برای مدت یک میلی‌ثانیه متوقف خواهد شد و سپس ادامه خواهد یافت.

"1 ms" یک مشخصه زمانی است.

فصل دوم

مقدمه‌ای بر VHDL

این فصل به شرح مفاهیم پایه‌ای زیر می‌پردازد :

- سطوح مختلف طراحی با کمک زبان VHDL
- ساختار توصیفی سلسله مراتبی
- اجزای ترکیبی و زیربخشها

VHDL یک زبان توصیف سخت‌افزاری با ساختارهای توصیفی گوناگون است و می‌توان با کمک آن کل سیستم را در پایین‌ترین سطح توصیفی یعنی در حد تعریف گیت‌های سیستم توصیف نمود. فراگیری VHDL برای طراحی‌های کوچک کار آسانی است ولی برای طراحی‌های پیچیده و بزرگ به دانش بسیار وسیع‌تری نیاز خواهد داشت.

دانستن مفاهیمی مثل سطوح مختلف توصیفی، عبارات موازی و متوالی، طراحی سلسله مراتبی، کتابخانه و غیره برای فهم کامل‌تر زبان VHDL ضروری است زبان VHDL بسیار مشابه زبان برنامه‌نویسی ADA می‌باشد، با این تفاوت که پارامترهایی مثل زمان و خصوصیاتی مثل کارکرد موازی را نیز اضافه دارد.

۱-۲ سطوح گوناگون برنامه‌نویسی با VHDL

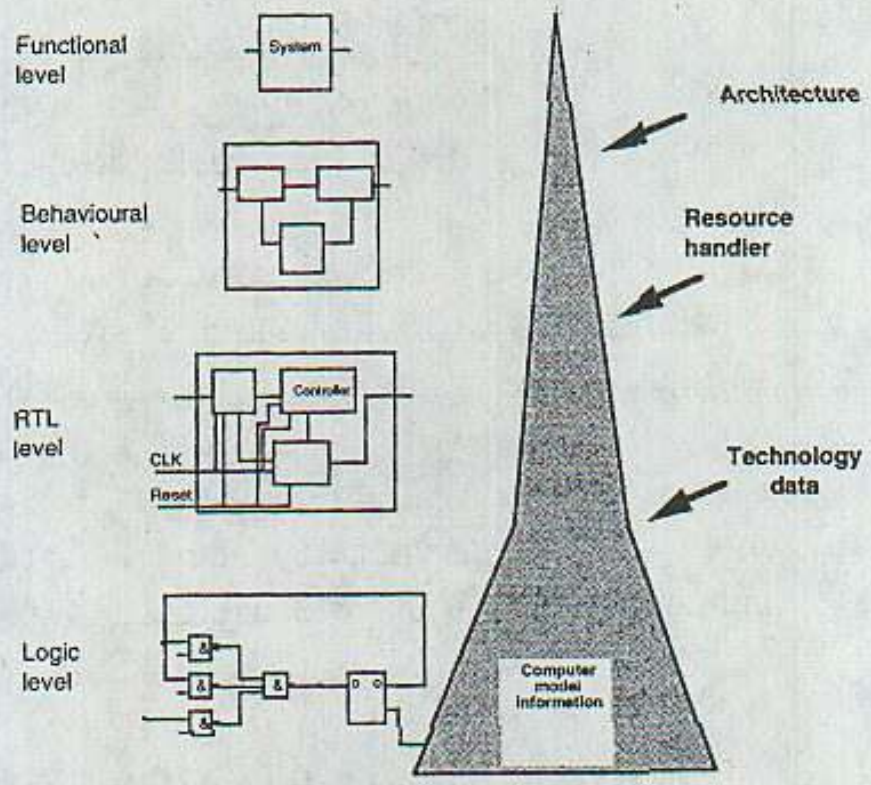
VHDL از نقطه نظر ساختاری بسیار غنی بوده و می‌توان از این زبان جهت توصیف بخشهای مختلف سخت‌افزاری در سطوح گوناگون، از تعریف توابع گرفته تا توصیف گیتی، استفاده کرد. سطوح

توصیفی متفاوتی که از جمله قابلیتهای ارزنده VHDL است وسیله‌ای برای پنهان کردن جزئیات خواهد بود. به طور مثال چنان‌چه یک طراح بخواهد دو عدد را در هم ضرب کند (A = B * C)، می‌تواند یکی از روشهای زیر را به کار برد :

- به کار بردن عملگر '*' در VHDL، a <= b * c
- طراحی ضرب‌کننده در سطح گیتی
- طراحی ضرب‌کننده در سطح لایه‌ای

سه مثال بالا نشان می‌دهند که چگونه یک تابع می‌تواند در سطوح مختلف پیاده‌سازی و اجرا شود : ***سطح انتقال رجیستری***[1]، ***سطح گیت***[2] و ***سطح لایه‌ای***[3].

شکل ۱-۲ سطوح گوناگون توصیفی سخت‌افزاری را معین می‌کند.



شکل ۱-۲ سطوح مختلف توصیفی در VHDL

1- Register Transfer Level

2- Gate Level

3- Layout Level

الگوریتم‌ها می‌توانند در ***سطح عملکردی***[1] توصیف شوند، مانند الگوریتم یک کنترلر که می‌توان آن را توسط نرم افزارهای کامپیوتری شبیه‌سازی کرد. یک الگوریتم نیازی به اطلاعات زمانی ندارد. با الگوریتم، تمام مشخصه‌های مندرج در VHDL قابل شبیه‌سازی خواهند شد.

رفتار یک سیستم و محدودیت‌ها و ویژگی‌های زمانی آن در ***سطح رفتاری***[2] توصیف می‌شود. برنامه در این سطح نیاز به معماری خاصی ندارد، به طور مثال اجرای عملگرها، پیاده‌سازی رجیسترها، ارتباط با RAM و غیره تعیین نمی‌شود. یکی از امتیازات برنامه‌نویسی در این سطح، طراحی سریع مدلهای شبیه‌سازی برای تست است. یک مدل طراحی شده در سطح رفتاری می‌تواند به صورت ***ماجولهای***[3] عملکردی و ارتباط بین آنها باشد، ماجول‌هایی که یک یا چند کار خاص را انجام می‌دهند و ویژگی‌ها و ارتباطات زمانی برای آنها معین شده است. در موارد مشخص می‌توان معماری خاصی را برای برنامه تعریف نمود، که در این حالت نباید توابع از منبع واحد استفاده کنند. در صورت استفاده از منابع مشترک خطا اعلام خواهد شد و در نتیجه ناچار به تغییر معماری برنامه خواهیم بود. در این موارد و به منظور جلوگیری از تغییر معماری برنامه می‌توان توابعی را که از منبع یکسانی استفاده می‌کنند دسته‌بندی نمود. در حال حاضر چند پروژه تحقیقاتی در حال استفاده از ابزارهای سنتز رفتاری هستند و تعدادی از این گونه ابزارها به بازار عرضه شده است.

سطح انتقال رجیستری : در این سطح امکان توصیف رفتار سنکرون و آسنکرون ماشینهای حالت، مسیرهای عبور داده‌ها، عملگرها (+، *، / ، >، ≥ و ...)، رجیسترها و غیره وجود دارد. در سطح RT کلیه رجیسترها در کد VHDL تعریف می‌شوند.

سطح منطقی یا گیتی : توصیف سیستم در این سطح با توابع و جبر بولی و یا به کمک شبکه گیتی انجام می‌شود.

سطوح دیگر : سطوح و رده‌های پایین‌تری نیز جهت توصیف یک طرح وجود دارد. سطح ترانزیستوری (یا سطح الکتریکی) و سطح لایه‌ای از این جمله‌اند، اما توسط VHDL حمایت نمی‌شوند. در سطح ترانزیستوری مدل ترانزیستورها، خازنها و مقاومتها طراحی می‌شود و در سطح لایه‌ای مدلها بر اساس فاکتورهای فیزیکی و ساختاری المان موردنظر، طراحی می‌گردند.

عمل سنتز یک طرح پس از بهینه‌سازی و کوچک کردن آن با تبدیل و ترجمه برنامه نوشته شده، در هر سطح توصیفی، به گیت‌ها و فلیپ فلاپ‌ها انجام می‌گیرد. در این مسیر فایل‌های اطلاعاتی

1- Functional Level

2- Behavioural Level

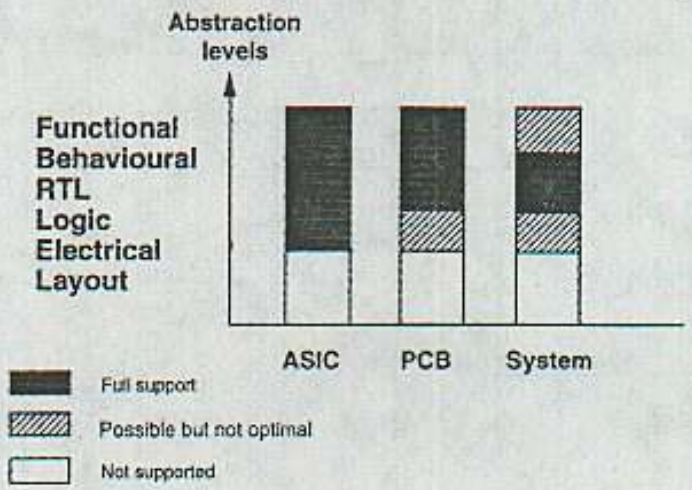
۳- ماجول : بخشی از یک سیستم که عملکرد خاص و ویژه‌ای را بر عهده دارد.

بسیاری در مورد تکنولوژی به کار رفته در سیستم و پروسه بهینه‌سازی و سنتز ایجاد می‌شود.

حجم اطلاعات در سطوح مختلف توصیفی افزایش می‌یابد، به عبارت دیگر پیچیدگی زیاد می‌شود. به منظور پیاده‌سازی یک تابع در یک ASIC، اطلاعات مربوط به تکنولوژی خاص مورد استفاده، سیم‌بندی‌ها، اطلاعات مربوط به گیت‌ها و اطلاعات مربوط به محدودیتهای زمانی و غیره مورد نیاز است. یک جمع‌کننده ۱۶ بیتی در سطح انتقال رجیستری با '+' توصیف می‌شود در حالی که برای توصیف همان جمعگر در سطح منطقی نیاز به نوشتن یک صفحه مطلب خواهد بود.

چرا از سطوح توصیفی گوناگون استفاده می‌شود ؟ اگر نگاهی به زبانهای برنامه‌نویسی CPUها بیندازیم خواهیم دید که آنها نیز دارای سطوح متفاوتی هستند، مثل میکروکدها، کدهای ماشینی، اسمبلر، C و C^{++}. چنان‌چه یک برنامه نیاز به زمان اجرای کمی داشته باشد، اسمبلر به کار برده می‌شود. از طرفی دیگر برای نوشتن یک برنامه پیچیده، از C یا ADA استفاده خواهد شد. در حالت کلی نیازهای مختلف، سطوح گوناگون برنامه‌نویسی را تعیین می‌کنند. در مورد VHDL نیز همین طور است، اگر زمان کمی برای اجرای برنامه لازم باشد باید سطح توصیفی بالاتری، انتخاب شود. در عمل سطح RT (و بخشهایی از سطح رفتاری) به طور اتوماتیک در سطح گیتی سنتز می‌شود.

توصیف یک جمع‌کننده با '+' بسیار سریع‌تر از توصیف آن در سطح گیتی است. اگر بخواهیم از طرح موردنظر به عنوان یک مدل استفاده کنیم، باید سطح رفتاری یا عملکردی انتخاب شود. چنان‌چه حجم کمی در اختیار باشد، سطح RT بهترین روش توصیف خواهد بود. البته برای کلیه سطوح، یک ابزار پیشرفته سنتز موردنیاز است. دسترسی به نتایج مطلوب در سطح گیتی بسیار مشکل می‌باشد ولی ناممکن نیست. سطح لایه‌ای در بعضی موارد برای حصول حداقل حجم و حداکثر بهره‌برداری مخصوصاً برای ASICها مؤثر است. معمولاً مصالحه‌ای بین فاکتورهای گوناگون، تعیین‌کننده سطح توصیفی و یا سطح برنامه‌نویسی خواهد بود.



شکل ۲-۲ حمایت VHDL از سطوح مختلف توصیف در موارد گوناگون

در ارتباط با شکل ۲-۲ :

- ***ASIC***[1]، معمولاً شامل آرایه‌های گیتی، سلولهای استاندارد و طراحی‌های کاملی که به سفارش مشتری انجام شده است، خواهد بود. در شکل ۲-۲، ***FPGA***[2]**ها** نیز در ستون ASICها قرار دارند.
- ***PCB***[3]، در یک بورد الکترونیکی معمولاً چندین ASIC به همراه یک میکروپروسسور وجود دارد.
- **System**، سیستم می‌تواند معانی مختلفی داشته باشد ولی در شکل ۲-۲ به معنای چند PCB و ارتباط بین آنها است.

شکل ۲-۲ نشان می‌دهد که می‌توان برای طراحی مدلهای سخت‌افزاری مورد نظر از بالاترین سطح توصیف سخت‌افزاری تا پایین‌ترین آن که شامل سطح منطقی است از VHDL استفاده کرد. VHDL برای ساختن مدلها و طراحی ASIC ها بسیار ایده‌آل است. در مورد PCB ها زمان شبیه‌سازی در سطح منطقی بسیار طولانی خواهد بود و معمولاً طراحی‌ها در سطح گیتی انجام نمی‌شود. بنابراین المان‌های PCB حداقل باید در سطح انتقال رجیستری طراحی شوند تا حداقل شبیه‌سازی سریع‌تر انجام گیرد. در سیستمهای بزرگ، توصیف عملکردهای پیچیده با VHDL بسیار مشکل خواهد بود، لذا باید سیستم را به بخشهای کوچک‌تر تقسیم کرد و کد مربوط به هر کدام و ارتباط بین آنها را با زبان VHDL نوشت.

سطوح مختلف توصیفی سخت‌افزاری در زبان VHDL را نباید با سطوح گوناگون طراحی‌های سلسله مراتبی اشتباه کرد. هر دو این روشها به طراح برای طراحی‌های پیچیده کمک می‌کنند.

۱-۱-۲ شبیه‌سازی

شبیه‌سازی، یکی از روشهای مؤثر بررسی طرح است. طرح مورد نظر در کامپیوتر به صورت زمان-گسسته تحلیل می‌شود در حالی که در واقعیت زمان-پیوسته است، با این وجود مدل کامپیوتری کم و بیش شبیه مدل واقعی است. چنان‌چه طراحی در سطوح بالا انجام شده باشد، مدل تحلیل شده کمتر شبیه واقعیت خواهد بود، در صورتی که طراحی در سطوح پایین‌تر بیشتر شبیه به واقعیت است.

1- Application Specific Integrated Circuit

2- Field Programmable Gate Array

3- Printed Circuit Board

سطوح برنامه‌نویسی	توصیف	واحد زمان
سطح رفتاری	رفتاری	میکروثانیه
سطح انتقال رجیستری	زبان RLT	عرض پالس ساعت
سطح منطقی	فرمولهای بول	نانوثانیه

جدول ۱-۲ مقایسه سطوح توصیف

مدل کامپیوتری نسبت به نمونه فیزیکی دارای مزایای بسیاری است. با تنظیم پارامترها در مقادیر مرزی و همچنین تغییر رنج دمایی می‌توان طرح را تحت شرایط متفاوت تحلیل کرد. این در حالی است که در مورد نمونه اولیه ساخته شده می‌توان تغییرات فاکتورهای فوق را بررسی نمود.

همان طور که گفته شد در شبیه‌سازی می‌توان پارامترها را در مقادیر مرزی آنها تنظیم کرد :

- **بدترین حالت** : پایین‌ترین ولتاژ (مثلاً ۴/۵ ولت)، بالاترین دما (مثلاً ۱۲۵ °C) و کندترین مد پروسه
- **حالت عادی و واقعی** : ولتاژ نرمال (۵ ولت)، دمای نرمال (۲۵ °C) و مد نرمال پروسه
- **بهترین حالت** : بالاترین ولتاژ (مثلاً ۵/۵ ولت)، پایین‌ترین دما (۵۵- °C) و سریع‌ترین مد پروسه

شبیه‌سازی مدلهایی که در سطح رفتاری طراحی شده‌اند تنها بررسی عملکردی آنها را در بر خواهد داشت. عملکرد و همچنین محدودیتهای زمانی را می‌توان با سنتز و شبیه‌سازی در سطح RT بررسی کرد، اما تنها از مدل فیزیکی است که می‌توان یک نتیجه واقعی را از رفتار زمانی مدل به دست آورد. در شبیه‌سازی می‌توان تلفات توان و غیره را نیز محاسبه کرد.

یکی از مشکلات شبیه‌سازی یک مدل پیچیده که فاکتورهای متنوعی را تحت بررسی دارد، سرعت کم و طولانی شدن زمان آن است. بنابراین شبیه‌سازی طراحی‌های عظیم و پیچیده در سطح لایه‌ای تقریباً ناممکن خواهد بود. سپس برای بررسی عملکردی یک مدل بهتر است حتی‌المقدور از سطوح بالاتر توصیفی مثل RTL استفاده شود و برای بررسی زمانی طرح در کوتاه‌ترین زمان، سطح توصیفی گیتی توصیه می‌گردد (به فصل ۱۱ رجوع کنید، «آشنایی با روشهای طراحی»).

۲-۱-۲ زبانهای دیگر توصیف‌کننده مدارات الکترونیکی

در حال حاضر چندین زبان برای توصیف مدلهای الکترونیکی به کار می‌رود. یکی از معروف‌ترین آنها VERILOG است. VERILOG برای سطوح پایین‌تر از RT به کار می‌رود.

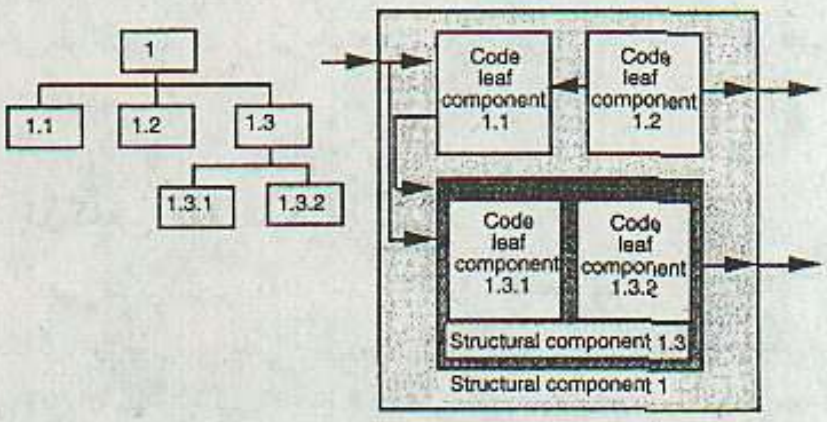
از آنجایی که در ساختار بعضی زبانها طراحی سلسله مراتبی مقدور نیست، لذا طراح در

طراحی‌های پیچیده دچار مشکل خواهد شد. مثالهایی از زبانهای توصیف سخت‌افزاری که دانشگاهها و مراکز تحقیقاتی آنها را طراحی کرده‌اند در زیر آورده شده است :

SLIDE	Structured Language for Interface Description and Evaluation (Parker and Wallace, 1981)
CONLAN	CONsensus LANguage (Piloty *et al.*, 1983)
ISPS	Instruction Set Processor Specification (Barbacci *et al.*, 1979
ADLIB	A Design Language for Indicating Behaviour [Hill *et al.*, 1979]
HILL	HIerarchical Layout Language (Lengauer and Melhorn, 1984)
OODE	Object Oriented Description Environment for computer hardware (Takeuchi, 1981)
BORIS	Block-ORiented Interacting Simulation system (Decker and Maierhofer, 1984)

۲-۲ طراحی سلسله مراتبی ـ کاهش پیچیدگی

در طراحی‌های بزرگ برای کاهش پیچیدگی نیاز به یک مکانیسم خاص کاملاً محسوس است. فهم یک مدل که دارای صدها جزء از قبیل RAM، منطق کنترل‌کننده، ماشینهای حالت و غیره می‌باشد بسیار مشکل است. طراحی سلسله مراتبی و تقسیم مدار به زیربخشهای مجزا و طراحی هر بخش به صورت جداگانه، از روشهای قدیمی برای دست و پنجه نرم کردن با پیچیدگی‌های مدار است. ممکن است با این کار مدار ظاهراً کمی پیچیده‌تر به نظر برسد اما فهم آن بسیار آسان‌تر خواهد بود.



شکل ۳-۲ طراحی سلسله مراتبی یک مدل

مکانیسمهای متنوعی جهت کاهش پیچیدگی در طراحی‌ها وجود دارد :

- سطوح توصیفی متفاوت زبان برنامه‌نویسی، که به منظور توصیف بخشهای پیچیده، بدون نیاز به توضیح در مورد جزئیات آنها به‌کار می‌روند (به بخش ۱-۲مراجعه کنید). function ها

و procedure ها نیز برای مقابله با پیچیدگی‌ها بسیار اهمیت دارند.

- طراحی سلسله مراتبی، که به منظور پنهان کردن جزئیات از اجزای ترکیبی استفاده می‌کند - قانون جعبه سیاه، جعبه سیاه به معنی زیربخشی است که تنها ورودی‌ها و خروجی‌های آن قابل رؤیت هستند (به فصل ۶ رجوع کنید، «VHDL ساختاری»).

چنان‌چه در طراحی به رده‌های بالاتر توصیفی و یا به مراحل بالاتر طراحی سلسله مراتبی برسیم، درجه دخالت جزئیات کمتر می‌شود.

طراحی سلسله مراتبی از زیربخشهایی تشکیل می‌شود. هر کدام از زیربخشها می‌توانند حاوی اجزای ترکیبی، کد VHDL و یا ترکیبی از هر دو باشند. مشابه درخت که وقتی در امتداد یک شاخه پیش رویم به شاخه‌ای دیگر می‌رسیم که بر روی آن زیربخشهای به نام برگها قرار دارند. اجزای ترکیبی می‌توانند با استفاده از عبارات و دستورات موازی و متوالی VHDL نوشته شده باشند.

شکل ۳-۲ یک طراحی سلسله مراتبی و زیربخشهای آن را نشان می‌دهد. اطلاعات مربوط به ارتباط زیربخشها با هم در توصیف ساختاری زیربخشها قرار دارد.

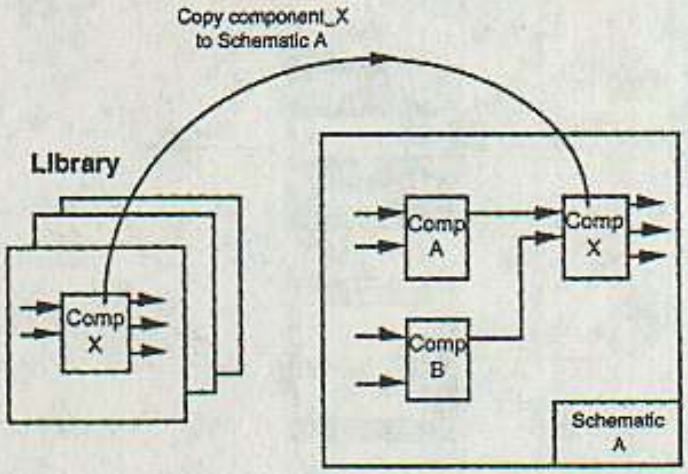
معمولاً کار طراحی با تعیین بخشهای واسط بین جزء اصلی و قسمت خارجی آغاز می‌شود و با تقسیم جزء اصلی به چند زیربخش و برقراری ارتباط بین آنها ادامه می‌یابد. هیچ ابزار خاصی انجام این تقسیم‌بندی را بر عهده نمی‌گیرد. قانون عمومی این است که عمل تقسیم‌بندی به گونه‌ای انجام شود که ارتباط بین اجزا به ساده‌ترین و کوچک‌ترین شکل ممکن باشد. در ضمن زیربخشها نباید بسیار کوچک باشند زیرا باعث نتیجه سنتز نامناسب و همچنین نوشتن کدهای VHDL غیرضروری برای توصیف بلوک‌های طراحی سلسله مراتبی خواهد شد (به فصل ۱۱ مراجعه کنید، «آشنایی با روشهای طراحی»).

۳-۲ جزء ترکیبی[1] VHDL

اجزای ترکیبی مرکزی‌ترین موضوع در VHDL هستند. آنها در کنار دیگر بخشها، سازندگان فایل‌های کتابخانه‌ای مربوط به قطعاتی چون میکروپروسسورها، مدارات ویژه و دیگر مدارات استاندارد می‌باشند. چنان‌چه یک جزء ترکیبی «خوب» طراحی شود می‌توان آن را در کتابخانه ذخیره کرده و چندین بار در طراحی‌های گوناگون به کار برد (به شکل ۴-۲ مراجعه کنید). در زبان علم کامپیوتر استفاده مجدد از المان‌های کتابخانه را «فراخوانی» جزء ترکیبی می‌گویند.

1- Component

اجزای ترکیبی عمومی و فراخوانی آنها مخصوص زبانهایی مثل VHDL است که بر پایه توصیف سخت‌افزاری یک قطعه عمل می‌کنند. اجزای ترکیبی عمومی اجزایی هستند که قبل از فراخوانی شدن می‌توان آنها را تغییر داده و اصلاح کرد. به طور مثال می‌توان به جزء ترتیبی که با سیگنال‌های ورودی و خروجی با طولهای مختلف ارتباط دارد اشاره کرد.



شکل ۴-۲ فراخوانی یک جزء ترکیبی

ساختار داخلی یک جزء ترکیبی می‌تواند مخفی باشد ـ قانون جعبه سیاه. در بعضی از موارد هیچ نیازی به دانستن چگونگی ساختار داخلی جزء ترکیبی نیست. معمولاً طراحان به ورودی‌ها و خروجی‌ها، مشخصات عملکرد بلوک و زمان‌بندی آن علاقه دارند. اکثر طراحان سخت‌افزاری باید چگونگی استفاده از جعبه‌های سیاه مثل مدارهای خانواده 74LSXX و نه ساختار داخلی آنها را بدانند.

یک جزء ترکیبی می‌تواند مشتمل بر اجزای ترکیبی دیگر باشد. همچنین امکان در برداشتن اجزای ترکیبی پیچیده و بزرگ مثل هسته مدارهای خاص مخابراتی برای کتابخانه وجود دارد.

۱-۳-۲ بخش اعلام[1] و بدنه[2] کد VHDL

اجزای ترکیبی از مهم‌ترین بخشهای زبان VHDL هستند و می‌توانند یک طرح کامل و یا بخش کوچکی از یک سیستم باشند. در این قسمت به بررسی بخشهای گوناگون یک جزء ترکیبی می‌پردازیم.

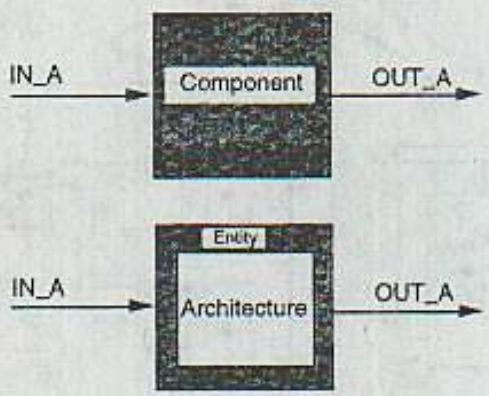
هر جزء ترکیبی از دو قسمت مجزا تشکیل می‌شود (شکل ۵-۲) :

1- Entity

2- Architecture

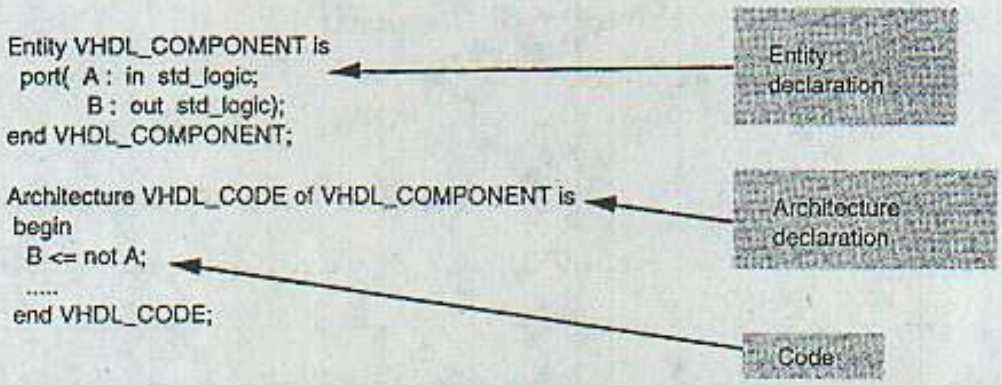
- **بخش اعلام (معرفی)** : این قسمت به معرفی درگاههای ورودی و خروجی می‌پردازد.
- **بدنه برنامه** : توصیف رفتاری و یا ساختاری بلوک در این قسمت انجام می‌شود.

توصیف رفتاری بلوک، یک عبارت کلی برای توابع، عملگرها، عملکرد و روابط بلوک با دیگر بلوک‌ها است. همچنین می‌تواند توصیف ساختاری جزء ترکیبی که مشتمل بر زیربخشهای گوناگون است، باشد.



شکل ۵-۲ بخشهای یک جزء ترکیبی : بخش اعلام و بدنه

اعلام برنامه در اصل یک جعبه سیاه با ورودی‌ها و خروجی‌های آن است. به طور مثال یک میکروپروسسور دارای یک بخش اعلام (جعبه سیاه) مشتمل بر سیگنال‌های داده، آدرس، کنترل و باس می‌باشد. رفتار و عملکرد این جعبه سیاه در بدنه برنامه آن و معمولاً با VHDL ساختاری مشخص می‌شود. به عنوان مثال می‌توان به جزء ترکیبی که شامل ALU و ***رجیسترهای وضعیت***[1] است اشاره کرد.



شکل ۶-۲ یک جزء ترکیبی

1- Status Register

شکل ۶-۲ یک جزء ترکیبی حاوی یک تابع ساده بولی را نشان می‌دهد. بخش اعلام این جزء ترکیبی vhdl_component و بدنه آن vhdl_code نام دارد.

دو اسم برای معرفی بدنه برنامه منظور شده است : یکی vhdl_component که تعیین می‌کند بدنه به چه جزء ترکیبی تعلق دارد و دیگری vhdl_code که نام بدنه برنامه است.

اعلام برنامه

این قسمت رابطه بین جزء ترکیبی و محیط اطراف آن را مشخص می‌کند. نام بخش اعلام در اصل نام جزء ترکیبی است.

```
Syntax:
entity < identifier_name> is
port([signal] <identifier>: [<mode>]< type_indication>;
     [signal] <identifier>: [<mode>]<type_indication>);
end [<identifier_name>];
```

<mode>	=	**in, out, inout, buffer, linkage**
in	=	Component only read the signal
out	=	Component only write to the signal
inout	=	Component read or write to the signal (bidirectional signals)
buffer	=	Component write and read back the signal (no bidirectional signals, the signal is going out from the component)
linkage	=	Used only in the documentation

in: جزء ترکیبی فقط سیگنال را می‌خواند

out: جزء ترکیبی فقط بر روی سیگنال می‌نویسد

inout: جزء ترکیبی یا سیگنال را می‌خواند و یا بر روی آن می‌نویسد (سیگنال دو جهته)

buffer: جزء ترکیبی بر روی آن می‌نویسد و از روی آن می‌خواند (سیگنال دو جهته نمی‌باشد و تنها به صورت سیگنال خروجی بلوک است)

linkage : تنها در متن نوشتاری به کار می‌رود

در مثال زیر یک سیگنال ورودی a_in و یک سیگنال خروجی b_out وجود دارد :

```
entity vhdl_component is
port(signal a_in:   in  std_logic;  --input
     signal b_out: out std_logic); --output
end vhdl_component;
```

تنها باید در مواردی که سیگنال‌ها دو طرفه و دو جهته هستند از مد inout استفاده کرد. چنان‌چه لازم باشد سیگنالی دوباره خوانده شود یا باید از مد buffer و یا از ***سیگنال‌های واسط داخلی***[1] استفاده شود (به فصل ۱۴ مراجعه کنید، «خطاهای متداول طراحی در VHDL و چگونگی اجتناب از آنها»). از آنجایی که کلمه signal اطلاعات خاصی را به سیگنال ورودی یا خروجی بلوک اضافه نمی‌کند، لذا می‌توان آن را حذف کرد. مد ورودی (in) و نام قسمت اعلام در انتهای آن نیز قابل حذف هستند. بنابراین دو مثال زیر معادلند :

```
Entity exl is
port (signal a,b: in  std_logic;
      signal c:   out std_logic);
end exl;

Entity exl is
port(a,b: in   std_logic;
     c:   out std_logic);
end;
```

لازم به یادآوری است که در VHDL تفاوتی بین حروف بزرگ و کوچک وجود ندارد.

به عنوان مثال بخش اعلام یک میکروپروسسور شرکت Intel به نام 8080A در زیر آورده شده است :

```
Entity uP8080A     is
 port (clk1,clk2:  in     std_logic; -- Clock inputs inputs
    RESET:         in     std_logic; -- Initiazes  processor
    HOLD:          in     std_logic; -- Suspends  processor
    INT:           in     std_logic; -- Interrupts  processor
    READY:         in     std_logic; -- Data bus
    D:             inout  byte;      -- Data bus
    A:             out    word;      -- Adress bus
    INT:           inout  std_logic; -- Interrupt  enable
    DBIN:          out    std_logic; -- Data bus in
    WR_N:          out    std_logic; -- Data bus out
    SYNC:          out    std_logic; -- Start of processor cycle
```

1- Internal Dummy Signal

```
        HLDA:        out   std_logic;  -- Hold acknowledge
        WAITs:       out   std_logic); -- Wait output
end;
```

در نسخه استاندارد VHDL-93 عبارت end entity را می‌توان به جای end (مطابق با نسخه VHDL-87) به کار برد.

```
Syntax (VHDL-93):
entity < identifier_name> is
port ([signal] <identifier>: [<mode>]< type_indication>;
      [signal] <identifier>: [<mode>]<type_indication>);
end [entity] [<identifier_name>];
```

مثال (VHDL-93) :

```
Entity ex is
  port (a,b: in   std_logic;
        c:   out  std_logic);
end entity ex;
```

بدنه برنامه

بدنه برنامه در اصل تعیین‌کننده رابطه بین ورودی‌ها و خروجی‌ها می‌باشد. نام بدنه برنامه متفاوت از نام جزء ترکیبی است، اما از آنجایی که اسم جزء ترکیبی (نام بخش اعلام برنامه) در قسمت بدنه وارد می‌شود ارتباط مستقیمی بین بدنه و جزء ترکیبی وجود دارد.

```
Syntax:
architecture <architecture_name> of <entity_identifier> is
[<architecture_declarative_part>]
begin
  <architecture_statement_part>
end [<architecture_name>];
```

قسمت معرفی بدنه باید قبل از اولین begin برنامه باشد. در این قسمت می‌توان *نوعها*[1]، زیربرنامه‌ها، اجزای ترکیبی دیگر و سیگنال‌ها را معرفی کرد. به‌عنوان مثال بدنه برنامه یک معکوس کننده در زیر آورده شده است :

1- Types

```
Architecture vhdl_ architecture  of vhdl_component is
begin
   b_out<=not a_in;
end vhdl_ architecture;
```

یک جزء ترکیبی باید دارای یک بخش اعلام و حداقل یک بدنه باشد. امکان داشتن چند بدنه در یک جزء ترکیبی وجود دارد اما همه آنها باید در ارتباط با بخش اعلام باشند. معماری‌های گوناگون نظیر توصیف در سطح رفتاری و یا در سطح انتقال رجیستری می‌توانند به عنوان بدنه‌های یک جزء ترکیبی محسوب شوند.

همان نکته‌ای که در مورد قسمت اعلام گفته شد در مورد بدنه برنامه نیز وجود دارد.

Syntax (VHDL-93):

```
architecture <architecture_name> of <entity_identifier> is
[<architecture_declarative_part>]
begin
   <architecture_statement_part>
end [architecture] [architecture_name>];
```

جزء ترکیبی NAND به عنوان یک مثال کامل در زیر آورده شده است :

```
-- ********************************
-- *
--* Filename          :NAND.VHD
--* File type         :VHDL for design (RTL)
--* Date              :14/11-96
--* Description       :NAND gate
--* Author            :Mr X
--* State             :Verified
--* Error             :None (I hope)
--* History           :A) started with the design 12/10

Entity nand_comp is
   port (a_in, b_in:   in  std_logic;    -- inputs
         a_out         out std_logic);   -- output

end;
```

```
Architecture nand_behv of nand_comp is
signal int: std_logic;                    -- Internal signal declaration
begin
      int<=a_in and b_in;
      a_out <=not int;
end;
```

به منظور به کار بردن عبارت تفسیری از علامت " - - " استفاده می‌شود. این مثال به خوبی چگونگی استفاده از عبارات توصیفی را نشان می‌دهد. به علاوه قوانین نوشتن یک کد VHDL واقعی که می‌تواند شامل دهها یا صدها خط برنامه باشد در این مثال مشخص شده است.

فراخوانی جزء ترکیبی به صورت زیر انجام می‌شود :

```
U1: nand_comp port map (a_in => givar_in_a,
                        b_in => givar_in_b
                        a_out =>out_lamp);
```

با اجرای این دستور، جزء ترکیبی nand_comp فراخوانی می‌شود و نام U1 را به خود می‌گیرد. هر اسم جزء ترکیبی فراخوانده شده باید یگانه و متفاوت از بقیه نامها باشد (به فصل ۶ مراجعه کنید، «VHDL ساختاری»).

به جای تعریف سیگنال داخلی int می‌توان مستقیماً از عملگر NAND در کد VHDL استفاده کرد. اپراتورهای منطقی زیر در استاندارد VHDL تعریف شده‌اند.

```
NOT
AND
NAND
OR
NOR
XOR
XNOR      --VHDL-93
```

مثال :

```
Architecture rtl of ex is
signal int: std_logic;
begin
  int<=not (((a nand b) nor (c or d)) xor e);
  A_OUT<=int and f;
end;
```

٤-۲ تمرین

۱- سطوح مختلف زبان VHDL را توصیف کنید.

۲- چه سطحی از زبان VHDL تنها به منظور ساختن مدلهایی برای شبیه‌سازی به کار می‌رود ؟ برای پاسخ خود دلیل بیاورید.

۳- چه سطحی از زبان VHDL به منظور طراحی و شبیه‌سازی سریع به کار می‌رود ؟ برای پاسخ خود دلیل بیاورید.

۴- یک قسمت اعلام (component_A) و یک قسمت معماری (rtl) برای انجام گرفتن عملیات زیر بنویسید.

```
d_out <= (a_in and b_in) and c_in;
```

راهنمایی : پارامترها از نوع std_logic هستند (در فصل ۳ توضیح داده شده‌اند).

۵- مفاهیم بسته، فراخوانی، پارامتر عمومی و VHDL ساختاری را توصیف کنید.

۶- علامت "--" در یک کد VHDL به چه معناست ؟

۷- چنان‌چه قرار باشد یک طرح به مدت طولانی کار کند، بخش توضیحات در کنار بخشهای دیگر آن مهم خواهد بود. فکر می‌کنید بخش ابتدایی کد VHDL که به توضیحات اختصاص داده‌اید باید چه فرمی داشته باشد ؟ مثال بزنید.

۵-۲ مراجع

Barbacci, M.R., W.B. Dietz und L.J. Szewerenko (1979) 'Specifications, evaluation and validation of computer architecture using instruction set processor descriptions', *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, IEEE.

Decker, H. and J. Maierhofer (1984) 'Very high level model description and simulation', *Proceedings of the Soc. Comput. Simulation Conference, San Diego.*

Hill, D.D. (1979) 'ADLIB: A modular, strongly-typed computer design language, *Proceedings of the 4th International Symposium on Computer Hardware Description Language*, IEEE.

Lengauer, T. and K. Melhorn (1984) 'The HILL System: A design environment for the hierarchical specification, compaction and simulation of integrated circuit layout', *Proceedings MIT Conference on Advanced Research in VLSI*, Artech House Company.

Parker, A. and J. Wallace (1981) 'SLIDE, an I/O Hardware Description Language', *IEEE Transactions on Computers*, vol. C-30.

Piloty, R., M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill and P. Skelly (1983) *CONLAN Report*, Springer-Verlag.

Takeuchi, A. (1981) 'Object-oriented description environment for computer hardware', *Proceedings of the IFIP International Conference and their Applications, Kaiserslauten.*

VERILOG, *VERILOG HDL Compiler Reference Manual*, Gateway Design Automation Corp.

فصل سوم

VHDL هم‌زمان

در زبان VHDL تعدادی ساختار موازی وجود دارد، باید دانست که این ساختارها چگونه کار می‌کنند. این فصل ساختارهای گوناگون موازی زبان VHDL را مورد بررسی قرار داده و اطلاعات اولیه درباره رفتار موازی را ارائه می‌دهد. همچنین در این فصل به تشریح نحوه کارگذاری سیستم کنترل کننده خطا در داخل اجزای ترکیبی (که غیرقابل سنتز هستند) می‌پردازیم. سیستم کنترل خطا امکانات بسیاری برای تولید کتابخانه‌ای از اجزای ترکیبی عرضه می‌دارد و چنان‌چه خطایی رخ دهد، اجزای ترکیبی سیگنال‌های ورودی را چک کرده و پیغام رخداد خطا را به شبیه‌ساز می‌فرستند. در خاتمه فصل نیز انواع مختلف داده‌ها مورد بررسی قرار خواهند گرفت.

۳-۱ مقداردهی به سیگنال

متغیرها، پارامترهای متوالی و سیگنال‌ها، پارامترهای موازی هستند. در سخت‌افزار این یک امر طبیعی است که با استفاده از جملات حاوی مقداردهی به سیگنال‌ها کار را به طور موازی به اجرا درآورد.

Syntax:
signal assignment:
<target_identifier> '<=' <selected_expression>';'

مثال :

```
Entity NAND_comp is
   port (a,b: in  std_logic;
         c:   out std_logic);
end;

Architecture rtl of NAND_comp is
begin
   c<=a nand b;
end;
```

مثال فوق هیچ گونه تأخیری را برای اجزای ترکیبی در برندارد. این یک روش معمول برای نوشتن کد VHDL به منظور سنتز کردن است. راه حل دیگر آن است که کد را با استفاده از دستور after در VHDL با تأخیری فرضاً معادل ۵ نانوثانیه برای جزء ترکیبی بنویسیم.

```
Architecture NAND_beh of NAND_comp is
begin
   c<=a nand b after 5 ns;  -- Component delay= 5 ns
end;                        -- inertial delay.
```

این امکان نیز وجود دارد که چندین مقدار را به یک سیگنال بدهیم. مقدارها را یکی بعد از دیگری لیست می‌کنیم و بعد از هر یک، علامت "," می‌گذاریم. به عنوان مثال :

```
c <=  '1',
      '0' after 10 ns,
       b after 20 ns;
```

سیگنال خروجی فوق مقادیر زیر را خواهد داشت :

زمان	مقدار	
0	'1'	
10	'0'	
20	b	مقدار سیگنال b--

عمل مقداردهی سیگنال فوق یک بار دیگر نیز هنگامی که مقدار سیگنال b تغییر کند، اجرا خواهد شد. به این ترتیب می‌توان برای سیگنال c یک شکل موج به دست آورد. این روش توصیف به وسیله ابزارهای سنتز حمایت نمی‌شوند. تأخیرهای بعد از دستور after باید به ترتیب، صعودی باشند. به این جهت مثال زیر غلط خواهد بود :

```
c <= '1',
     '1' after 10 ns,
     '0' after  5 ns;    -- 5 ns less than 10 ns-> Error
```

مثال صحیح به این شکل خواهد بود :

```
c <=  '1',
      '0' after 5 ns,
      '1' after 10 ns;
```

۲-۳ تأخیر درونی[1] و تأخیر انتقالی[2]

دو نوع تأخیر وجود دارد : تأخیر درونی و تأخیر انتقالی.

- **تأخیر درونی :**

ـ در VHDL به صورت حالت اولیه وجود دارد (می‌تواند مدل خازنی شبکه باشد).

ـ اگر از دستور after استفاده شود جهشهای نویزی به وجود نمی‌آیند.

ـ غالباً برای تأخیرهای اجزای ترکیبی به کار می‌رود.

- **تأخیر انتقالی :**

ـ پالس تولید می‌شود، صرف‌نظر از عرض آن.

ـ رویداد خوبی است برای تأخیرهای مربوط به ارتباطات درونی.

اگر از حالت تأخیر انتقالی استفاده شود، دستور باید در کد VHDL اضافه شود، اما تأخیر درونی فرض اولیه است و نیازی به ذکر جداگانه ندارد.

مثال :

```
q1 <= a after 5 ns;                -- تأخیر درونی
q2 <= a transport after 5 ns;      -- تأخیر انتقالی
```

اگر یک جهش نویزی فرضاً با طول زمانی ۲ نانوثانیه در جزء ترکیبی که تأخیری به اندازه ۱۰ نانوثانیه دارد رخ دهد این نویز به طور نرمال در سیگنال خروجی دیده نخواهد شد، به همین دلیل است که تأخیر درونی معمولاً برای مدل‌سازی تأخیرهای اجزای ترکیبی به کار می‌رود. اما مشکل وقتی بروز

1- Inertial Dalay

2- Transport Delay

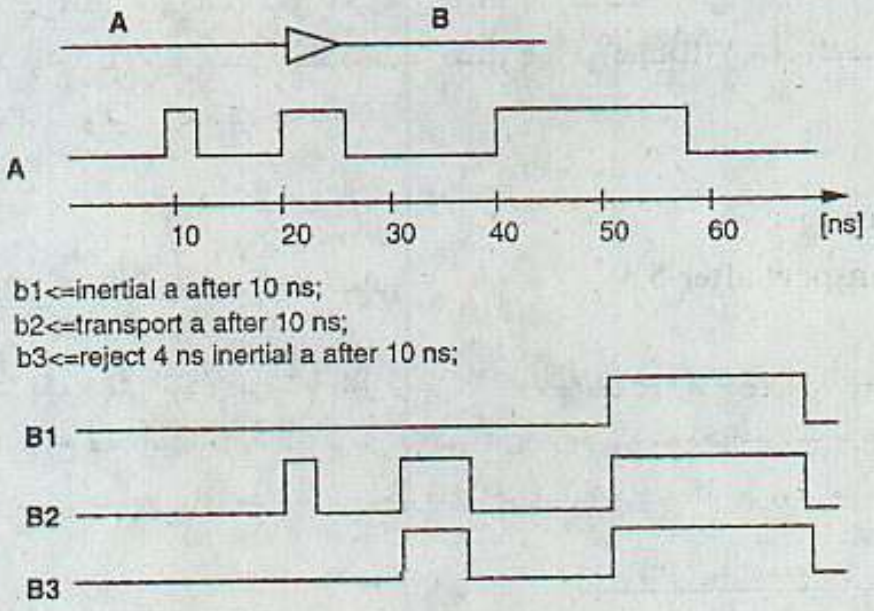
می‌کند که بخواهید مدلی از جزء ترکیبی بسازید که دارای یک تأخیر ۱۰ نانوثانیه بوده و تمام جهشهای نویز آن در سیگنال ورودی، طولی برابر یا بیشتر از فرضاً ۵ نانوثانیه داشته باشند که در سیگنال خروجی نیز دیده می‌شوند. این مشکل در VHDL-93 که دستور جدیدی با عنوان reject را معرفی می‌کند حل شده است. از دستور reject می‌توان برای تعریف طول زمانی جهشهای نویز استفاده نمود. در این فاصله زمانی است که جزء ترکیبی می‌تواند نویز را عبور دهد یا ندهد.

مثال (VHDL-93):

```
q3 <= reject 4ns inertial a after 10ns;
```

تأخیر فوق تمام نویزهای ورودی را که طولی کمتر از ۴ نانوثانیه داشته باشند نادیده می‌گیرد. پالس‌هایی که ۴ نانوثانیه یا بیشتر باشد در خروجی (q3) بعد از ۱۰ نانوثانیه دیده می‌شوند. دقت کنید که اگر از دستور reject استفاده می‌شود تأخیر درونی باید مشخص و ذکر شود.

ابزارهای سنتز هیچ یک از این مدلهای تأخیر را حمایت نمی‌کنند. بهترین استفاده آنها هنگامی است که مدلهای VHDL برای شبیه‌سازی ساخته می‌شوند. اما تأخیر ساکن می‌تواند در طراحی مورد استفاده قرار گیرد. اتفاقی که در اینجا می‌افتد این است که ابزار سنتز تأخیر را نادیده می‌گیرد. تأخیر انتقالی در اکثر ابزارهای سنتز موجب بروز خطا می‌شود. شروط زمانی باید در موقع طراحی برای ابزار سنتز معین شوند (به فصل ۱۱ رجوع کنید، «آشنایی با روشهای طراحی»). مثالی از این مورد در شکل ۱-۳ آورده شده است.



شکل ۱-۳ تأخیر انتقالی و درونی

۳-۳ ساختار موازی

سخت‌افزار به طور طبیعی موازی است و VHDL نیز همین طور است. این به آن معناست که تمام ساختارهای موازی زبان VHDL می‌توانند به طور موازی به اجرا درآیند و در نتیجه ترتیب ردیفی کد VHDL بی‌معنا می‌شود.

مثال :

```
Architecture example of ex is
begin
   a<=b;
   b<=c;
end example;
```

که با مثال زیر برابر است :

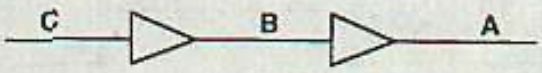
```
Architecture example2 of ex is
begin
    b<=c;
    a<=b;
end example2;
```

دو مثال گفته شده در بالا از لحاظ عملکردی مثل هم هستند زیرا دستورات موازی VHDL به وسیله رویدادها کنترل می‌شوند.

b <= c; -- زمانی اجرا می‌شود که مقدار c تغییر کند.

مثال فوق به این معنی است که وقتی مقدار سیگنال ورودی c تغییر می‌کند تمام خطهایی که c در سمت راست علامت مقداردهی قرار دارد (مانند b <= c) اجرا می‌شوند.

اگر در مثال بالا عمل سنتز انجام شود نتیجه مطابق آنچه که در شکل ۲-۳ نشان داده شده است به دست می‌آید.



شکل ۲-۳ نتیجه سنتز مثال بالا (بهینه‌سازی انجام نشده است)

همان طور که سخت‌افزار نشان می‌دهد، مقدار سیگنال a قبل از آنکه سیگنال b مقدار خود را تغییر دهد تغییر نمی‌کند. به همین نحو سیگنال b نیز تا وقتی مقدار سیگنال c تغییر نکرده، تغییر مقدار نمی‌دهد. این موضوع در کد VHDL نیز به خاطر آنکه به وسیله رویداد کنترل می‌شود همین طور است. این بدان معناست که تمام دستورات موازی و همزمان VHDL را می‌توان بدون آنکه عملکرد

دستخوش تغییر شود با هر نظم و ترتیبی نوشت.

تمام جملات موازی و هم‌زمان را می‌توان با یک برچسب (عنوان) تشخیص داد :

```
Label_name : b <= a;
```

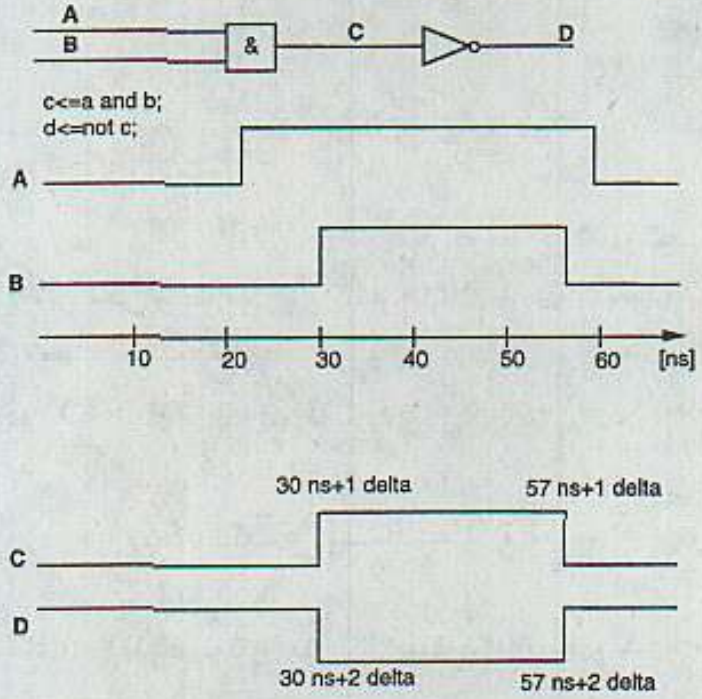
این عنوان تنها به منظور مستندسازی به کار می‌رود و هیچ گونه اهمیت عملکردی ندارد.

۳-٤ زمان دلتا[1]

از زمان دلتا در VHDL برای صف‌بندی رویدادهای متوالی استفاده می‌شود. مدت زمان بین دو رویداد متوالی را به نام تأخیر دلتا می‌شناسند. تأخیر دلتا در اجرای آنی، هیچ معادلی ندارد و زمانی اجرا می‌شود که کلاک شبیه‌سازی در حالت ساکن قرار دارد. در مقداردهی به سیگنال، مقدار مستقیماً به سیگنال داده نمی‌شود بلکه این عمل در زودترین موقع بعد از تأخیر دلتا صورت می‌گیرد :

```
b <= a;        -- سیگنال b مقدار سیگنال a را بعد از یک زمان دلتا دریافت می‌کند.
```

در شکل ۳-۳ مثالی برای این مورد آورده شده است :



شکل ۳-۳ مثالی از تأخیر دلتا

1- Delta Time

در یک بلوک منطق ترکیبی که در آنجا تمام المان‌ها تأخیر صفر نانوثانیه دارند، تمام مقداردهی‌ها بدون تأخیر (صفر نانوثانیه) روی می‌دهند، اما ممکن است چندین تأخیر دلتا داشته باشند. شبیه‌ساز VHDL تا زمانی که تمام سیگنال‌ها به حالت ثابت درآیند تعداد زمانهای دلتا را شمارش می‌کند. اگر کد VHDL ناصحیح نوشته شده باشد این ریسک وجود دارد که طراحی تا بی‌نهایت در نوسان باقی بماند. برای جلوگیری از فشردگی و گیر کردن شبیه‌سازها اکثر آنها بعد از فرضاً ۱۰۰۰ زمان دلتا متوقف می‌شوند. معمولاً امکان تنظیم این عدد وجود دارد. مثال زیر نشان‌دهنده یک کد VHDL صحیح است ولی این مثال طرحی را تولید می‌کند که تا بی‌نهایت در نوسان خواهد بود.

```
q <= not q;
```

سیگنال q می‌تواند بعد از یک زمان دلتا ارزش معکوس خود را به عنوان مقدار جدید به خود بگیرد. این تغییر باعث می‌شود که خط دوباره اجرا شود و در نتیجه سیگنال q بعد از یک زمان دلتا دوباره مقدار جدید به خود خواهد گرفت و همین طور الی آخر. این مشکل با اضافه کردن یک تأخیر مطابق زیر به سادگی حل می‌شود :

```
q <= not q after 10ns;
```

توجه کنید که اگر کد بالا سنتز شود منجر به یک فیدبک آسنکرون می‌شود که معمولاً رویداد مطلوبی در طراحی به شمار نمی‌رود (به فصل ۱۲ مراجعه کنید، «آشنایی با روشهای آزمایش»).

۳-۵ عبارت when

Syntax:

```
<target> <= <expression> [after < expression>] when <condition>
            <expression> [after < expression>]. . .;
```

امکان استفاده چندباره از خط when-else وجود دارد. سیگنال <target> صرف‌نظر از مقدار <expression> همیشه باید مقداردهی شود. این بدان معناست که دستور باید با عبارت else <expression> خاتمه‌یابد.

مثال :

```
Entity ex is
  port (a,b,c:  in  std_logic;
        data:   in  std_logic;
        q:      out std_logic);
```

```
end;

Architecture rtl of ex is
begin
  q<=  a when data="00" else
       b when data="11" else
       c;
end;
```

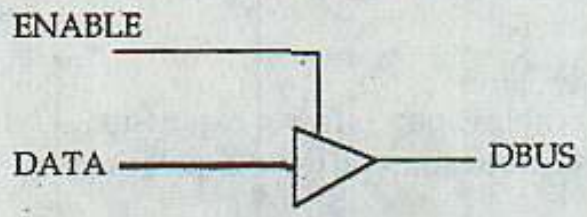
اما در استاندارد VHDL-93 آخرین عبارت <else <expression می‌تواند حذف شود (اگر چه این عمل توسط بیشتر ابزارهای سنتز حمایت نمی‌شود).

مثال (VHDL-93) :

```
Architecture rtl of ex is
begin
  q<= a when data= "00" else
      b when data= "11";   --No else < expression>, only
end;                       --valid in VHDL-93
```

دستور when دستور بسیار مفیدی است، مخصوصاً وقتی که یک بافر سه حالته طراحی می‌کنیم این دستور خیلی به کار می‌آید. مثال آن در شکل ۴-۳ آورده شده است.

```
dbus <=  data when enable = '1' else 'Z';
```



شکل ۴-۳ نتیجه سنتز

اگر به هنگام تعریف سیگنال‌های dbus و data، آنها را بردار تعریف می‌کردیم، تمام پاس به ازای هر خروجی دارای یک بافر سه حالته بود. اما باید در معماری کد آن تغییرات اندکی اعمال شود :

```
dbus <= data when enable = '1' else (others => 'Z');
```

با استفاده از مقداردهی «others => 'Z'»، تمام بردار بدون توجه به طول آن مقدار 'Z' را به

خود می‌گیرند. این نحوه مقداردهی به تمام بردار بسیار مفید و مؤثر است و خوانایی برنامه را بیشتر می‌کند. اگر لازم باشد طول بردار تغییر یابد، تنها کافی است که معرفی بردار در بخش اعلام اصلاح شود و نیازی به تغییر معماری برنامه نخواهد بود.

استفاده از چند سیگنال مختلف در عبارت <condition> مجاز بوده و باعث می‌شود که دستور انعطاف‌پذیر شده و برای طراحی مفیدتر شود.

مثال :

```
q<= a when en= '0' else
    b when data= "11" else
    c when enable= '1' else
    d;
```

توجه نمایید که شرطها با همان ترتیبی که ذکر شده‌اند چک می‌شوند. این شرطها خط به خط ارزیابی می‌گردند تا آنکه شرط صحیح یافت شود. این بدان معناست که اگر فرضاً ترتیب خط ۲ (data = "11") و خط ۳ (enable = '1') تغییر کند، چنان‌چه هر دو شرط به طور همزمان صحیح باشند، نتیجه متفاوت خواهد بود. فرض کنید en = '1'، data = '11' و enable = '1' باشد. این به آن معناست که در مثال فوق q مقدار سیگنال b را می‌گیرد. اگر ترتیب خطهای ۲ و ۳ تغییر کند به q مقدار سیگنال c داده خواهد شد. فقط در میان دستورات موازی و همزمان است که ترتیب فاقد اهمیت می‌شود. در داخل یک دستور موازی و همزمان، یعنی دستور when-else ترتیب مهم است.

VHDL بین حروف بزرگ و کوچک فرقی نمی‌گذارد. تنها استثنا در داخل علامت نقل قول (' ') یا علامت نقل قول دوبله (" ") یعنی زمانی است که مقدار 'Z' به سیگنالی از نوع std_logic یا vlbit داده شده که باید به حروف بزرگ باشد :

```
sig <= 'Z';        -- خوب
sig <= 'z';        -- بد
```

۳-۶ عبارت with

Syntax:

```
with <expression> select
    <target> <= < expression> when <chose>;
```

تمام گزینه‌های ممکن باید یک به یک و به ترتیب ذکر شوند. اگر بخواهید تمام گزینه‌های باقیمانده را در کنار هم جمع کنید، می‌توانید از دستور when others استفاده نمایید. در چنین موردی دستور others باید آخرین حالت ممکن باشد.

مثال :

```
Entity ex is
  port (a,b,c: in  std_logic;
        data:  in  std_logic_vector(1 downto 0);
        q:     out std_logic);
end;

Architecture rtl of ex is
begin
    with data select
       q<= a when "00" ,
            b when "11",
            c when others;
end;
```

دستور with-select در مقایسه با دستور when-else انعطاف‌پذیری کمتری دارد زیرا می‌تواند فقط یک عبارت یا <expression> داشته باشد. لیکن این دستور منتج به کدی می‌شود که خواندن و ساخت آن نسبتاً آسان است.

۷-۳ مثالی از مدل رفتاری یک مالتی پلکسر

دو مدل رفتاری یک مالتی پلکسر که در زیر آورده شده است با هم برابر و معادل هستند. در معماری مدل اول از عبارت when و در مدل دوم از عبارت with استفاده شده است. هر دو مدل نیز قابل سنتز می‌باشند اما ابزار سنتز تأخیر ۱۰ نانوثانیه را نادیده خواهد گرفت.

```
Entity  mux2 is
  port (sel_0,a,b: in  std_logic;
        c:         out std_logic);
end mux2;

Architecture mux2_beh of mux2 is
begin
  c<= a after 10 ns when sel_0='0' else
      b after 10 ns;
end mux2_beh;
```

```
Architecture mux2_beh2 of mux2 is
begin
  with sel_0 select
     c<= a after 10 ns when '0',
         b after 10 ns when others;
end mux2_beh2;
```

بنابراین VHDL ساختارهای گوناگونی را برای توصیف یک نوع رفتار مشابه عرضه می‌دارد. فصل ۱۵، «مثالهایی از طراحی و نکات راهنما» نگاهی دارد به اینکه کدام گزینه باید انتخاب شود.

۸-۳ پارامترهای عمومی[1]

از پارامترهای عمومی برای بیان اطلاعاتی در داخل یک مدل، فرضاً اطلاعاتی مربوط به زمان، استفاده می‌شود. این پارامترها باید در بخش معرفی یک entity و قبل از دستور port اعلام شوند. یک پارامتر عمومی می‌تواند هر نوع داده‌ای به خود بگیرد، لیکن محدودیتهایی از لحاظ اینکه ابزارهای سنتز کدام نوع را قبول خواهند کرد وجود دارد.

```
Entity generic_ex is
  generic (delay: time:=10 ns);
  port (a,b: in  std_logic;
        c:  out std_logic);
end generic_ex;

Architecture generic_beh of generic_ex is
begin
     c<=a and b after delay;
end generic_beh;
```

جزء ترکیبی فوق یک تأخیر عمومی به اندازه delay نانوثانیه دارد. این تأخیر ۱۰ نانوثانیه تعریف شده است. این مقدار می‌تواند به هنگام فراخوانی جزء ترکیبی تغییر کند. پارامترهای عمومی را همچنین می‌توان برای طراحی اجزای ترکیبی عمومی مورد استفاده قرار داد (به فصل ۶ مراجعه کنید، «VHDL ساختاری»).

1- Generic

۹-۳ دستور assert ـ کنترل خطا در VHDL

assert یک ساختار زمانی جالبی است که از طریق آن می‌توان عملکرد و شروط زمانی مدلی را در داخل یک جزء ترکیبی VHDL مورد آزمایش قرار داد.

اگر شرط مربوط به دستور assert در حین شبیه‌سازی یک کد VHDL رعایت نشود پیام حاوی یک نوع severity برای کاربر (شبیه‌ساز) ارسال خواهد شد. با به کارگیری دستور assert می‌توان موارد زیر را تست کرد :

- ترکیبهای نادرست و ممنوع سیگنال‌ها
- آیا ویژگی‌های زمانی رعایت شده‌اند یا خیر
- نوع داده
- متصل بودن ورودی‌های جزء ترکیبی

Syntax:
Assert <condition>
Report <message>
Severity <error_level>

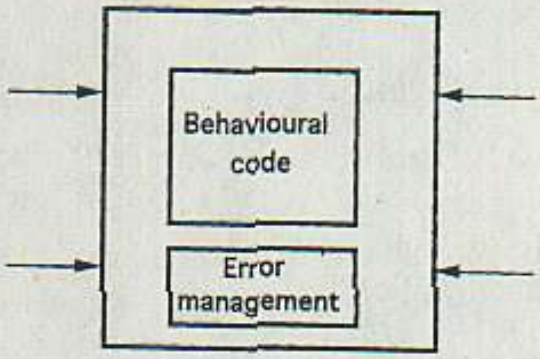
اگر شرط assert رعایت نشده باشد، پیام assert به علاوه نام واحد (پروسس) که حذف شده است به کاربر ارسال می‌گردد. چهار سطح گوناگون برای پیغام خطا وجود دارد :

- توجه (Note)
- اخطار (Warning)
- خطا (Error)
- ابطال (Failure)

پیام و سطح خطا به شبیه‌ساز گزارش شده و معمولاً به صورت متن ساده در پنجره حاوی دستور شبیه‌سازی VHDL ظاهر می‌شود. به طور هم‌زمان این امکان وجود دارد که آن سطح خطایی را که شبیه‌ساز VHDL با وقوع آن باید کار شبیه‌سازی را متوقف کند تعیین نمود. حالت اولیه برای اکثر شبیه‌سازها سطح خطا است.

از دستور assert می‌توان برای آزمایش و تأیید سیگنال‌های خارجی جزء ترکیبی یا رفتار داخلی و همچنین تأیید زمان و عملکرد استفاده کرد. از آنجایی که assert هم یک دستور متوالی و هم یک دستور موازی است، می‌توان آن را در هر کجای کد VHDL به کار گرفت (شکل ۵-۳ را ببینید).

کد کنترل خطا فقط در خلال شبیه‌سازی VHDL حضور پیدا می‌کند. این کد در سنتز منظور نخواهد شد.



شکل ۵-۳ جزء ترکیبی با بخش کنترل خطا

مثالی از کد کنترل خطا در زیر آورده شده است :

```
assert in_0 /='X' and in_1 /='X'
report "in is not connected"
severity Error;
```

کارکرد کنترل خطا، چک کردن این است که سیگنال‌های in_0 و in_1 به هم متصل هستند و یا اینکه دارای مقادیر تعریف نشده‌ای می‌باشند. اگر به هم متصل نباشند شبیه‌ساز متوقف شده و پیغام «in is not connected» را نشان می‌دهد.

متغیر now در استاندارد VHDL تعریف شده است. این متغیر حاوی زمان مطلق داخلی در شبیه‌ساز است. دستور assert این امکان را به ما می‌دهد که زمان شبیه‌ساز را چک کنیم، فرضاً بررسی کنیم که آیا از ۹۰۰ نانوثانیه بیشتر نخواهد شد :

```
Process (clk)
begin
    assert now < 900 ns
    report "stopping simulator (max simulation time 900 ns)"
severity Failure;
end process;
```

اگر دستور assert در بخش موازی کد VHDL به کار گرفته شده باشد، فقط زمانی اجرا خواهد شد که یک رویدادی بر روی لیست حساسیت رخ داده باشد. اگر دستور assert فقط به چک کردن

زمان با کمک متغیر now بپردازد هیچ رویدادی رخ نخواهد داد و دستور assert هیچ گاه به اجرا در نخواهد آمد. از این رو دستور assert در مثال بالا در یک پروسس که با هر لبه پالس ساعت به اجرا در می‌آید قرار داده شده است. درباره پروسسها، در فصل ۴ «VHDL متوالی» اطلاعات بیشتری کسب خواهید کرد.

در مواردی که دستور assert در بخش موازی کد VHDL به کار رفته باشد می‌توان آن را در یک پروسه غیرفعال قرار داد و در entity مطابق مثال زیر به کار برد :

```
Entity ex is
  port (a,b: in   std_logic;
        q:   out std_logic);
  begin
  assert  a/='1' or b/='1'
  report  "a='1' and b='1' at the same time"
  severity  warning;
end;

Architecture
  . . .
```

assert دستور مناسبی برای آزمایش و تأیید پاسخ دریافتی از مدار یک محیط آزمایشی خواهد بود. مطالب بیشتر در این باره در فصل ۸ «محیط آزمایش» آورده شده است.

در نسخه استاندارد VHDL-93 اگر دستور report در بخش متوالی کد VHDL به کار رفته باشد می‌توان دستور assert را حذف نمود.

مثال (VHDL-87) :

```
process (a,b)
begin
 if a='1' and b='1' then
      assert false
      report " a='1' and b='1'  ";
 end if;
  . . .
end process;
```

مثال (VHDL-93) :

```
process (a,b)
begin
  if a='1' and b='1' then
     report " a='1' and b='1'  " ;
```

```
    end if;
    . . .
end process;
```

۳-۱۰ طراحی در سطح رفتاری یا جریان داده‌ای

در هنگام نوشتن کد VHDL می‌توان دو روش طراحی را به کار برد : طراحی رفتاری و طراحی جریان داده‌ای.

مثال طراحی جریان داده‌ای چنین است :

```
q<= c3 and c2 and c1 or c0;
c0<= not c0;
```

یک توصیف رفتاری می‌تواند به صورت زیر باشد :

```
if sign= '0' then
  q0<=q0+1;
else
  q0<=q0-1;
end if;
```

هر دو مثال فوق قابلیت سنتز شدن را دارند. اما غالباً کد رفتاری به طریقی نوشته می‌شود که تنها می‌تواند شبیه‌سازی شود و نمی‌توان آن را برای سنتز به کار برد (به فصل ۱۱ مراجعه کنید، «آشنایی با روشهای طراحی»).

۳-۱۱ موضوع[1]، گروه[2] و نوع[3]

در مورد بعضی از محتویات این قسمت قبلاً صحبت شده است، ولی در زیر واژه‌ها به طور سیستماتیک‌تری توصیف می‌شوند. تعریف یک موضوع در VHDL یعنی به کارگیری یک ***ثابت***[4]، یک ***متغیر***[5] یا یک ***سیگنال***[6]. یک موضوع دارای مقدار و ارزشی از یک نوع بخصوص است.

1- Object

2- Class

3- Type

4- Constant

5- Variable

6- Signal

هر موضوع یک «نوع» و یک «گروه» دارد. «نوع» مشخص می‌سازد که موضوع حاوی چه نوع داده‌ای است. گروه مشخص می‌سازد که با موضوع چه کارهایی می‌توان انجام داد. کد VHDL به شدت وابسته به نوع است و این بدان معناست که نوعهای مختلف را نمی‌توان بدون تبدیل کردن نوعها با هم در آمیخت.

گروه	موضوع	نوع داده
signal	a :	Std_logic

گروه

موضوعها دارای سه گروه گوناگون هستند :

- ثابت
- متغیر
- سیگنال «یک سیم»

«ثابت» مقدارش تغییر نمی‌کند. «ثابت» را می‌توان در هر بخشی معرفی نمود و می‌تواند دارای هر نوعی باشد. مثال :

```
constant a : a_type : = "1001";
```

«متغیر» ارزش و مقدار خود را تغییر می‌دهد و می‌تواند در یک پروسس یا زیربرنامه اعلام شود و از هر نوعی باشد. مثال :

```
a : = John;
```

«سیگنال» می‌تواند مقدارش را در ارتباط با زمان تغییر دهد. مثال :

```
b <= input1;
```

نکته : یک نام نوع خوب و مناسب می‌تواند با ختم کلیه اسمهای نوع با عبارت "_type" باشد.

ex John_type

۱-۱۱-۳ نوعهای داده[1]

هر سیگنال باید یک نوع داده داشته باشد، که تعیین‌کننده مقدار و ارزشی است که سیگنال می‌تواند به خود بگیرد. متداول‌ترین نوعهای داده را می‌توان به شکل زیر توصیف نمود.

1- Data Type

نوع Boolean

سیگنال از نوع boolean می‌تواند فقط ارزشهای "درست" یا "غلط" را به خود بگیرد :

```
Architecture rtl of ex is
begin
   process(. . .)
   variable John: boolean;
   begin
      John:=a<b;
      if John then . . .
      . . .
   end process;
end;
```

ابزارهای سنتز مقدار غلط را به "0" و مقدار درست را به "1" ترجمه می‌کنند. نباید ارزشهای "درست" یا "غلط" را به یک سیگنال از نوع std_logic نسبت داد، مگر از توابع تبدیل استفاده شده باشد.

نوع Integer

استاندارد VHDL-87 طول یک integer را تعریف نمی‌کند. طول integer در اجرا کنترل می‌شود، یعنی وابسته به ابزار مورد استفاده است. لیکن بیشتر ابزارها ۳۲ بیت را در نظر می‌گیرند. این بدان معناست که یک integer می‌تواند مقادیری بین ۲۱۴۷۴۸۳۶۴۸- و ۲۱۴۷۴۸۳۶۴۷ را به خود بگیرد.

مثال ۱ :

```
constant loop_number : integer : = 345;
```

مثال ۲ :

```
signal my_int : integer range 0 to 255;
```

نوع Character (VHDL-87)

```
type character is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS, HT, LF, VT, FF, CR, SO, SI,
    DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
    CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,
    ' ', '!', '"', '#', '$', '%', '&', ''',
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', ':', ';', '<', '=', '>', '?',
    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[', '\', ']', '^', '_',
    '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
    'x', 'y', 'z', '{', '|', '}', '~', DEL);
```

در نسخه VHDL-93 تعداد کاراکترها مجاز افزایش یافته و چند کاراکتر ویژه نیز به آن اضافه شده است.

نوع Character (VHDL-93)

```
type character is (
        nul, soh, stx, etx, eot, enq, ack, bel,
        bs, ht, lf, vt, ff, cr, so, si,
        dle, dc1, dc2, dc3, dc4, nak, syn, etb,
        can, em,  sub, esc, fsp, gsp, rsp, usp,

        ' ', '!', '"', '#', '$', '%', '&', ''',
        '(', ')', '*', '+', ',', '-', '.', '/',
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', ':', ';', '<', '=', '>', '?',

        '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '[', '\', ']', '^', '_',
        '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
        'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', del,
```

```
c128, c129, c130, c131, c132, c133, c134, c135,
c136, c137, c138, c139, c140, c141, c142, c143,
c144, c145, c146, c147, c148, c149, c150, c151,
c152, c153, c154, c155, c156, c157, c158, c159,

-- the character code for 160 is there (NBSP),
-- but prints as no char

The last 64 characters are special characters
      (example national characters)...
```

در حال حاضر بیشتر ابزارهای سنتز فقط نوع character مربوط به نسخه VHDL-87 را حمایت می‌کنند و نسخه وسیع‌تر شده در VHDL-93 را نمی‌پذیرند.

نوع Bit

بیت نوعی است که در نسخه استاندارد VHDL تعریف شده است و فقط می‌تواند مقدارهای "0" یا "1" را به خود بگیرد.

نوع Vlbit

Vlbit نوعی است که توسط ViewLogic مورد استفاده قرار می‌گیرد و نسخه وسیع شده نوع BIT در استاندارد VHDL است. Vlbit دارای مقادیر "X"، "Z"، "0" و "1" می‌باشد. در شبیه‌سازی، "X" به مفهوم «ناشناخته» است. سنتز، ارزش "X" را به «don't care» ترجمه می‌کند.

مثال : عبارت

if John (6 **downto** 0)= "1XXXX0X" **then** ...

در سنتز معادل است با :

if (John (6) = '1' **and** John (1) = '0') **then** ...

'Z' به مفهوم امپدانس بالا، هم در شبیه‌سازی و هم در سنتز است. طراح باید مراقب آنچه مورد سنتز قرار می‌گیرد باشد، مثلاً بعضی از FPGA ها دارای بافر سه حالته نمی‌باشند. این مشکل را می‌توان با مالتی‌پلکس کردن سیگنال حل کرد.

در استاندارد VHDL تنها نوعهای bit و bit_vector تعریف شده‌اند. اگر از bit و bit_vector در طراحی استفاده شود با چندین محدودیت مواجه خواهیم شد :

- توصیف سه حالته ممکن نیست.
- داشتن چند درایور برای یک سیگنال امکان‌پذیر نخواهد بود.
- دادن مقدار «نامعلوم» به سیگنال ممکن نخواهد بود.
- دادن مقدار don't care به سیگنال ممکن نخواهد بود.

از آنجایی که نوع bit تنها می‌تواند مقادیر '0' و '1' را بگیرد، توصیف مفاهیم «سه حالته» و یا «نامعلوم» بودن، برای آن ناممکن است. در مورد سیگنال سه حالته غالباً چندین سیگنال برای درایو آن وجود دارند. این بدان معناست که باید روشی یافت که به مدد آن بتوان معین کرد چنان‌چه چندین درایور وجود داشته باشد، کدام مقدار باید به سیگنال داده شود. این عمل در نسخه استاندارد VHDL برای داده از نوع *bit* یا *bit_vector* تعریف نشده است. دادن مقدار don't care نیز می‌تواند در بعضی وضعیتها مؤثر باشد. این روش به ابزار سنتز امکان می‌دهد که مقدار را به نحوی انتخاب کند که هدفهای تعیین شده بهینه‌سازی، آسان‌تر به دست آیند.

در طراحی مدلهای شبیه‌سازی در VHDL استفاده از bit و bit_vector نیز به محدودیتهای زیر منجر خواهد شد :

- توصیف کردن pull up ممکن نیست ('1' ضعیف)
- توصیف کردن pull down ممکن نیست ('0' ضعیف)
- تعریف کردن اینکه سیگنالی مقداردهی اولیه نشده است ممکن نیست.

برای غلبه بر این مشکلات، تولیدکنندگان VHDL نوعهای داده‌ای خاص خود را ساختند. این عمل گرچه مشکلات یاد شده را برطرف کرد اما موجب پیدایش مشکلات جدیدی شد :

- کد VHDL وابسته به نوع ابزار مورد استفاده شد.
- ایجاد ارتباط و یکی کردن دو طرح از دو ابزار مختلف غیرممکن شد.

از آنجایی که این نوعهای داده فقط در ابزارهای مربوط به خود تعریف شده‌اند اگر کد VHDL، به محیط یک شبیه‌ساز دیگر وارد شود مشکلاتی بروز خواهد کرد. به طور عادی این امکان وجود دارد که این نوعهای داده را در شبیه‌ساز جدید تعریف کرد. لیکن این امر با اساس وجودی VHDL که عدم وابستگی آن به محیط طراحی و شبیه‌سازی بود مغایرت پیدا می‌کند. اگر از این نوعهای داده تعریف شده برای هر کاربر استفاده شود یکی کردن دو طرح یا دو زیر بلوک نیز امکان‌پذیر نخواهد بود. همان‌طور که قبلاً گفته شد، VHDL زبانی است که شدیداً وابسته به نوع داده است و از این رو ارتباط داشتن دو سیگنال (در طراحی سلسله مراتبی) چنان‌چه از نوعهای مشابه نباشند مجاز نخواهد بود.

برای غیروابسته کردن VHDL به محیط طراحی و شبیه‌ساز، نوعهای داده تعریف شده توسط IEEE معرفی شدند که عبارتند از : std_logic و std_ulogic. این نوعهای داده تبدیل به نوعهای استاندارد در صنعت طراحی شدند. std_logic و std_ulogic می‌توانند دقیقاً مقادیر مساوی هم بگیرند و تنها تفاوتشان در این است که std_logic یک ***زیر ـ نوع تعیین شده***[1] از std_ulogic است (به فصل ۵، «کتابخانه، بسته و زیربرنامه‌ها» رجوع کنید).

نوع std_ulogic

std_ulogic نوعی است که در بسته std_logic_1164 از کتابخانه ieee معرفی شده است. Std_ulogic می‌تواند مقدارهای زیر را بگیرد :

1. 'U' -- بدون مقداردهی اولیه
2. 'X' -- نامعین قوی
3. '0' -- صفر قوی
4. '1' -- یک قوی
5. 'Z' -- امپدانس بالا
6. 'W' -- نامعین ضعیف
7. 'L' -- صفر ضعیف
8. 'H' -- یک ضعیف
9. '-' -- بی‌تفاوت

نوع std_logic

این نوع نیز در بسته std_logic_1164 از کتابخانه ieee معرفی شده است. std_logic می‌تواند همان مقادیر std_ulogic را اتخاذ کند. تفاوت آنها در این است که std_logic به شکل زیر تعریف شده است :

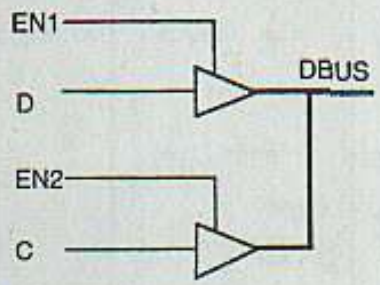
```
subtype std_logic is resolved std_ulogic;
```

اینکه گفته می‌شود std_logic "تعیین‌شده" است به آن معناست که اگر سیگنالی با چند درایور جداگانه مقداردهی شده باشد یک تابع تعیین‌کننده از پیش تعریف شده فعال می‌شود که کار آن حل

1- Resolved Subtype

تضاد و تعیین مقداری است که باید به سیگنال داده شود. این بدان معناست که به عنوان مثال std_logic می‌تواند برای باس سه حالته‌ای که در آن چندین درایور مقادیر گوناگونی را به آن می‌دهند (اما معمولاً هم زمان عمل نمی‌کنند) مورد استفاده قرار گیرد (شکل ۶-۳). اگر یک سیگنال از نوع std_logic با بیش از یک درایور مقداردهی شده باشد، این امر منجر به خطا خواهد شد (شکل ۷-۳)، زیرا VHDL اجازه نمی‌دهد که یک سیگنال غیر«تعیین‌شده» توسط بیش از یک درایور مقداردهی شود.

```
Entity ex is
  port( d,c,en1,en2:  in  std_logic;
        dbus:         out std_logic);
end;
Architecture rtl of ex is
begin
    dbus<= d when enable1='1' else 'Z';
    dbus<= c when enable2='1' else 'Z';
end ;
```



شکل ۶-۳ std_logic

```
Entity ex is
  port( d,c,en1,en2:  in  std_ulogic;
        dbus:         out std_ulogic);
end;
Architecture rtl of ex is
begin
    dbus<= d when enable='1' else 'Z';
    dbus<= c when enable2='1' else 'Z';
end ;
```

→ ERROR

شکل ۷-۳ std_ulogic

این محدودیت std_ulogic باعث شده است که نوع داده std_logic بیشتر ترجیح داده شود. آسان‌ترین کار این است که در سراسر طراحی از یک نوع داده استفاده شود زیرا با این عمل می‌توان از تبدیل نوعها اجتناب کرد. عیب std_logic این است که اگر به طور اشتباه دو درایور برای یک سیگنال واحد در کد VHDL داشته باشیم خطا اعلام نخواهد شد. در عمل شبیه‌سازی مقدار 'X' به این سیگنال داده می‌شود تا بتوان خطا را کشف کرد.

نوع Array

std_ulogic_vector is defined as:
type std_ulogic_vector **is array** (natural **range**<>) **of** std_ulogic;

std_logic_vector is defined as:
type std_logic_vector **is array** (natural **range**<>) **of** std_logic;

bit_vector is defined as:
type bit_vector **is array** (natural **range** <>) **of** bit;

vlbit_vector is defined as:
type vlbit_vector **is array** (natural **range** <>) **of** vlbit;

نوعهای vlbit_vector و vlbit_1d یکسان هستند.

معرفی و اعلام نوع

می‌توانید نوعهای خاص خود را در معماری برنامه به وجود بیاورید.

type <identifier> **is** <type_definition>;
type_definition = **array** <index_value> **of** element_type;
Example:
type A_type **is array** (3 **downto** 0) **of** std_logic;
type my_int **is** integer **range** 0 **to** 15;
signal d: A_type;
signal q: my_int;

نوع integer را می‌توان مستقیماً در بخش اعلام سیگنال تعریف کرد :

signal q : integer **range** 0 **to** 15;

ضروری است که **گستره**[1] نوع integer معلوم و اعلام شود زیرا در غیر این صورت ابزار سنتز فرض را بر این خواهد گذاشت که integer دارای گستره ۳۲ بیتی است. به طور مثال اگر پارامتر با نوع

1- Range

integer وارد یک رجیستر شود و گستره آن تعریف نشده باشد، به جای ۴ بیت که در مثال بالا برای آن منظور شده است، ۳۲ بیت به آن داده خواهد شد. همین وضع در مورد بردارها نیز صادق است.

بردارها

طول بردارها را می‌توان در بخش اعلام سیگنال مشخص کرد :

```
Entity ex is
 port (a:   in      std_logic_vector (3 downto 0);
       b:   in      bit_vector (0 to 3);
       c:   out     std_ulogic_vector (4 downto 0));
end;

Architecture rtl of ex is
signal i1: std_logic_vector (3 downto 0);
signal i2: vlbit_vector (3 downto 0);
signal i3: std_ulogic_vector (3 downto 0);
begin
  . . .
end;
```

می‌توان بردارها را با to یا downto معرفی کرد. اگر از to استفاده شود بردار باید به شکل 0 to 3 تعریف شود. اگر از downto استفاده گردد بردار باید به فرم 3 downto 0 تعریف شود. معمولاً ساده‌ترین کار در طراحی‌ها آن است که پیوسته از یک تعریف واحد استفاده شود. به علاوه دستور downto با روال عادی فکری طراح مبنی بر آنکه بیتی که دورتر از همه در سمت چپ قرار دارد ***مهم‌ترین بیت***[1] است، بیشتر سازگاری دارد. از این روست که در اکثر طراحی‌هایی که برای سنتز نوشته شده‌اند در تمام تعریفهای برداری از شکل downto استفاده شده است.

زمان

مثال :

```
Constant delay: time:=0 ns;
a<= b after delay;
```

1- Most Significant Bit (MSB)

نوع Enumerated

این امکان وجود دارد که شما نوع داده خود را در VHDL تعریف کنید. به طور اصولی، این نوعهای داده می‌توانند هر مقداری را اختیار کنند. ماشینهای حالت محیط متداولی برای استفاده از این نوعها هستند.

مثال :

```
type state_type is (start, idle, waiting, run);
signal state: state_type;
```

سیگنال حالت در شبیه‌سازی VHDL یکی از چهار مقدار تعریف شده برای آن را اختیار می‌کند. این اسمهای خود ـ گویای سیگنال عمل شبیه‌سازی طرح را آسان‌تر می‌سازند. اکثر ابزارهای سنتز نیز این گونه نوعها را حمایت می‌کنند. از آنجایی که سیگنال حالت در مثال فوق می‌تواند چهار مقدار مختلف اختیار کند به وسیله ابزار سنتز به یک بردار ۲ بیتی تبدیل می‌شود.

۳-۱۱-۲ نوعهای داده سنتزپذیر

جدول ۱-۳ نشان می‌دهد که چه نوعهایی در حال حاضر (۱۹۹۵) می‌توانند در ViewLogic، Synopsys و Autologic2 سنتز شوند. علامت "1d-" به معنای آرایه یک بعدی، و علامت "2d-" نشانه آرایه دو بعدی است. حرف V نمایانگر ViewLogic، حرف A نمایانگر Autologic2 و حرف S نمایانگر Synopsys است.

سایر ابزارهای سنتز تا حدی زیربخشهایی از جدول فوق را حمایت می‌کنند.

۳-۱۲ مقداردهی به بردار

وقتی طراحی با VHDL انجام می‌گیرد معمولاً بردارها به عنوان نوع داده استفاده می‌شوند. یک بردار می‌تواند مقداری را به طرق مختلف اختیار کند.

اگر بخواهیم یک مقدار باینری به بردار بدهیم به طریق زیر عمل می‌کنیم :

```
a_vector <= "10011";
```

دقت کنید که مقدار بردارها در داخل علامت دوبله نقل قول و بیت‌های انفرادی در داخل علامت تکی نقل قول نوشته می‌شوند.

اگر عملگرهای منطقی مانند and بر روی بردارها مورد استفاده قرار گیرند، نتیجه یک and بیتی خواهد شد.

Data types	Fully supported	Partly supported	Typically not supported	No meaning with support
Boolean	V, A, S			
boolean_1d	V, A, S			
boolean_2d	A, S		V	
String		V, A, S		
Integer	A, S	V		
integer_1d	A, S	V		
integer_2d	A, S	V		
Std_logic	A, S		V	
std_logic_1d	A, S		V	
std_logic_2d	A, S		V	
std_ulogic	A, S		V	
std_ulogic_1d	A, S		V	
std_ulogic_2d	A, S		V	
Vlbit	V	A, S		
vlbit_1d	V	A, S		
vlbit_2d		V, A, S		
Character		V, A, S		
character_1d		V, A, S		
character_2d		A, S	V	
Time				V, A, S
time_1d				V, A, S
time_2d				V, A, S
enumeration	V, A, S			
enum._array	V, A, S			
Record	A, S		V	
record_array	A, S		V	
Text				V, A, S

جدول ۱-۳ نوعهای داده سنتزپذیر با ViewLogic، Autologic2 و Synopsys

مثال :

```
Architecture rtl of ex is
signal a,b: std_logic_vector (3 downto 0);
signal c:   std_logic_vector (3 downto 0);
begin
      a<="0110";
      b<="1101";
      c<=a and b;
end;
```

بردار c مقدار "0100" را اختیار خواهد کرد.

```
  "0110"
and "1101"
  "0100"
```

شرط لازم برای عملگرهای منطقی این است که هر دو بردار باید دارای طول مساوی باشند و برداری که مقدار نتیجه در آن قرار می‌گیرد نیز باید دارای طول صحیح باشد.

۱-۱۲-۳ عدد پایه[1] در مقداردهی به رشته‌ای از بیت‌ها

برای مقداردهی به bit_vector علائم از قبل تعریف شده‌ای وجود دارند. این علائم به شرح زیر مورد استفاده قرار می‌گیرند :

	مثال	
Binary	B "11000"	
Octal	O "456"	
Hex	X "FFA5"	
Decimal	239	(تنها برای ثابت‌ها)
Real	4.6E-4	(برای سنتز حمایت نمی‌شود)

امتیاز این روش این است که اگر در کد VHDL به یک بردار به جای یک مقدار باینری مقدار هگزادیسمال داده شود، خواندن آن آسان‌تر می‌شود. لیکن از نقطه نظر سنتز هیچ تفاوتی ندارند.

اگر بخواهیم به یک بردار مقدار باینری بدهیم لزومی ندارد که حرف B را قبل از بردار بنویسیم. بنابراین دو مقداردهی زیر یکسان هستند.

```
a_vector <= "10011";
a_vector <= B "10011";
```

عدد پایه در مثال بالا فقط بر بردارهای bit_vector اثر دارد. لیکن ViewLogic استاندارد را اندکی تغییر می‌دهد، به این معنا که اعداد پایه فوق بر روی بردارهای Vlbit_vector در ابزار ViewLogic نیز اثربخش هستند.

از *زیر ـ خط*[2] (_) می‌توان برای بهتر کردن خوانایی برنامه در بردار بیتی استفاده کرد. هرگاه

1- Number Base

2- Underline

زیر ـ خط مورد استفاده قرار گیرد باید علائم و حروف تعریف شده به طور کامل نوشته شوند. همچنین می‌توان از زیر ـ خط در رشته‌های بیتی به غیر از bit_vector استفاده کرد.

مثال :

```
a_vect <=B"1100_0011_0011_1100";     -- Good
a_vect <="1100001100111100";         -- Identical to first
                                     -- assignment
a_vect <=X"C33C";                    -- Identical to first
                                     -- assignment
a_vect <=X"C3_3C";                   -- Identical to first
                                     -- assignment
a_vect <= "1100_0011_0011_1100";     -- Error, B missing
a_vect <= "C33C";                    -- Error, X missing
```

همان طور که گفته شد، VHDL زبانی است که بسیار وابسته به نوع داده است. این بدان معناست که مجاز نخواهیم بود فرضاً مقدار یک bit_vector را به یک std_logic_vector بدهیم مگر آنکه از یک تابع تبدیل استفاده کنیم (مراجعه نمایید به «بسته‌های IEEE» در پیوست ب).

مثالی از تبدیل :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex is
  port (a:  in   std_logic_vector (2 downto 0);
        b:  out  bit_vector (2 downto 0));
end;

Architecture rtl of ex is
begin
  b<= to_bitvector(a,0);
end;
```

همان طور که استفاده از بردار std_logic یا بردار std_ulogic به هنگام طراحی کاملاً طبیعی است، ضروری است که یک تابع تبدیل را نیز برای تبدیل عدد پایه به کار گیریم.

مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
```

```
Entity ex is
   port (a: out   std_logic_vector (15 downto 0);
         b: out  bit_vector (15 downto 0));
end;

Architecture rtl of ex is
begin
   a<=to_stdlogicvector(X"F6");          -- Conversion function must
                                         -- be used
   b<=X"E4";                             -- Bit_vector
end;
```

همان طور که مثال بالا نشان می‌دهد، مقدار X "F6" با کمک تابع تبدیل to_stdlogicvector به بردار std_logic_vector تبدیل شد. این تبدیل به این دلیل است که X "F6" یک بردار بیتی (bit_vector) به وجود می‌آورد و از آنجا که VHDL زبانی است که شدیداً وابسته به نوع است، بنابراین مقدار یک بردار bit_vector را نمی‌توان به یک بردار std_logic_vector داد، مگر آنکه نخست به بردار std_logic_vector تبدیل شده باشد. تابعی که برای تبدیل مورد استفاده قرار می‌گیرد به بسته‌ای که در ابتدای کد VHDL اعلام شده باشد بستگی دارد (به فصل ۵ مراجعه کنید، «کتابخانه، بسته و زیربرنامه‌ها»). در این مثال تابع مورد استفاده to_stdlogicvector بوده که در بسته ieee.std_logic_unsigned اعلام و معرفی شده است. تابع تبدیل مورد استفاده تأثیری بر روی نتیجه شبیه‌سازی یا سنتز ندارد. انتخاب بین آنها بر اساس این خواهد بود که ابزار سنتز کدام تابع را حمایت می‌کند.

اگر سیگنال a در مثال فوق مقدار X "F6" را مستقیماً اختیار کرده بود در هنگام کامپایل کردن کد VHDL خطا اعلام می‌شد. در نسخه استاندارد VHDL-93 می‌توانیم به یک بردار std_logic_vector و به یک بردار std_logic_vector به طور مستقیم مقدار بدهیم بی‌آنکه نیاز به تابع تبدیل داشته باشیم. این امر، کد VHDL را از لحاظ خواندن آسان‌تر و از لحاظ نوشتن سریع‌تر می‌کند.

```
signal a: std_logic_vector (7 downto  0);
a<=to_stdlogicvector (X"F4");          -- VHDL-87
a<=X"F6";                              -- VHDL-93
```

مقداردهی به یک بردار همان اندازه انعطاف‌پذیر است که وقتی بخواهیم به یک integer مقداردهی کنیم. هر دو نوع داده (بردار و integer) می‌تواند با هر عدد پایه مقداردهی شوند. مثلاً، می‌توان به یک integer مستقیماً عددی را بین دو علامت # و با مشخص کردن عدد پایه داد.

مثال :

```
Architecture rtl of ex is
constant myint: integer: =16#FF#;              -- myint=255
signal int1, int2,int3: integer range 0 to 1023;
begin
   int1<=16#FE#;                               -- 16#FE# = 254
   int2<=2#100110#;                            -- 2#100110# = 38
   int3<=8#17#;                                -- 8#17# = 15

end;
```

۲-۱۲-۳ برشی[1] از یک آرایه

اگر بخواهید به یک بیت و یا بخشی از یک بردار مقداری را بدهید می‌توانید به صورت زیر عمل کنید :

```
Architecture rtl of ex is
signal a_vect: std_logic_vector(4 downto 0);
signal b_vect: std_logic_vector (0 to 4);

begin
   a_vect(4) <= '1';
   a_vect(3 downto 0)<= "0110";
   b_vect(4)<= '0';
   b_vect(0 to 3)<= "1001";
end;
```

توجه نمایید که وقتی برشی از یک آرایه مقداردهی می‌شود، باید جهت برش به همان سمتی باشد که در بخش اعلام ذکر شده است، یعنی to یا downto همچنین می‌توان به یک بردار برشی از یک بردار دیگر را مقداردهی کرد.

مثال :

```
Architecture rtl of ex is
signal  a_vect:     std_logic_vector(4 downto 0);
signal  b_vect:     std_logic_vector(0 to 4);
```

1- Slice

```
signal  c:          std_logic;
begin
   a_vect<= "01101";
   b_vect(4)<=c;
   b_vect(0 to 3)<=a_vect(3 downto 0);
end;
```

همان طور که قبلاً گفته شد نوع داده بردارها در دو طرف علامت مقداردهی باید یکسان و طول بردارها نیز باید مساوی باشند، بنابراین مثال زیر غلط خواهد بود :

```
Architecture bad of ex is
signal a: std_logic_vector (2 downto 0);
signal b: std_logic_vector (3 downto 0);
begin
    a<=b:    -- Left side = 3 bits , right side = 4 bits -> error
end;
```

مقداردهی بالا باید طوری بازنویسی شود که هر دو طرف علامت مقداردهی دارای طول برابر باشند، یعنی :

```
a <= b (2 downto 0);
```

و یا به جای آن (بسته به تابع موردنظر) :

```
a <= b (3 downto 1);
```

اگر ترتیب اندیس بردارها برای هر دو بردار به طور متفاوت تعریف شده باشد، نیز می‌توان به یک بردار مقدار بردار دیگر داد. ولی باید به دقت چک شود که کدام بیت چه مقدار را دریافت می‌کند.

مثال :

```
Architecture rtl of ex is
signal a,b,c:   std_logic_vector (2 downto 0);
signal   d:     std_logic_vector (0 to 2);
  a<=d;
  b<=c;
end;
```

در مقداردهی‌های بردارهای فوق، ارزشهای زیر به بردارهای a و b داده شده است :

```
a (2) <= d (0);     b (2) <= c (2);
a (1) <= d (1);     b (1) <= c (1);
a (0) <= d (2);     b (0) <= c (0);
```

۳-۱۲-۳ پیونددهی[1]

علامت '&' به معنای پیونددهی در کد VHDL است. پیونددهی در مقداردهی به بردارها خیلی مفید واقع می‌شود.

مثال :

```
Architecture rtl of ex is
signal a:       std_logic_vector (5 downto 0);
signal b,c,d:   std_logic_vector (2 downto 0);
begin
  b<='0'  & c(1)  & d(2);
end;
```

اگر بردار c مقدار "011" و بردار d مقدار "101" را داشته باشند به بردار b مقدار '0' & '1' & '1' یعنی در واقع مقدار "011" و به بردار a مقدار "101" & "011" یعنی "011101" داده می‌شود.

پیونددهی بیشتر برای جمله‌های if دار به کار برده می‌شود. به مثال زیر توجه کنید :

```
if c & d = "001100" then
...
```

همان طور که قبلاً گفته شد، بردارها در هر دو طرف علامت مقداردهی باید دارای طول برابر باشند و نمی‌توانیم با استفاده از پیونددهی در سمت‌چپ علامت مقداردهی، چنین کاری را انجام دهیم.

مثال (ناصحیح) :

```
Architecture bad of ex is
signal a: std_logic_vector(2 downto 0);
signal b: std_logic_vector(3 downto 0);
begin
  '0' & a<=b;                --Error
end;
```

۳-۱۲-۴ متراکم کردن[2]

اگر با بردارهای قدری بزرگ‌تر طراحی می‌کنید و می‌خواهید مقدار یکسانی را به تمام بردار بدهید، می‌توانید این عمل را به شکل زیر انجام دهید :

1- Concatenation

2- Aggregate

```
Architecture rtl of ex is
signal a:  std_logic_vector(4 downto 0);
begin
  a<=(others=>'0');
end;
```

این دستور عیناً مانند ;"00000" ⇐ a است. از مزایای روش (others ⇒ '0') ⇐ a یکی این است که در مورد مقداردهی به بردارهای بزرگ به نوشتن کمتری نیاز است و دیگر آنکه مقداردهی کاملاً غیروابسته به طول بردار می‌شود. مقداردهی به بردار با کلمه others اصطلاحاً مقداردهی با مکانیسم متراکم کردن نام دارد. همچنین مقدار دادن به چند بیت از یک بردار و سپس مقداردهی بیت‌های باقیمانده با استفاده از کلمه others امکان‌پذیر است.

```
a <= (1 => '1', 3 => '1', others => '0');
```

مقداردهی بالا به این معناست که بیت‌های ۱ و ۳ در بردار a مقدار '1' را گرفته و سایر بیت‌های بردار، مقدار '0' را گرفته‌اند. همین طور با روش متراکم‌سازی این امکان وجود دارد که مقادیر سایر سیگنال‌ها را به بردار داد :

```
a <= (1 =>c(2), 3 => c (1), others => d (0));
```

در مقداردهی فوق به بردار a می‌توان از روش پیونددهی به عنوان روش عمل دیگری استفاده کرد (فرض کنید که بردار a، ۵ بیتی باشد) :

```
a <= d (0) & c (1) & d (0) & c (2) & d (0);
```

عیب این روش توصیفی این است که مقداردهی وابسته به طول بردار می‌شود و در صورتی که طول بردار a تغییر یابد باید تغییر کند. از نقطه نظر سنتز هیچ تفاوتی در نتیجه حاصل نخواهد شد. هر دو روش توصیف توسط اکثر ابزارهای سنتز حمایت می‌شوند.

۵-۱۲-۳ عبارت تکمیل‌کننده[1]

بعضی اوقات نوع داده یک پارامتر مشخص نیست. اگر کامپایلر نتواند نوع پارامتر را به طور آشکار تشخیص دهد خطا به وجود خواهد آمد. برای آنکه بتوان نوع داده را برای کامپایلر روشن نمود

1- Qualifier

باید از عبارت تکمیل‌کننده استفاده کرد. منظور از عبارت تکمیل‌کننده آن است که نوع داده به طور صریح بیان شده، به دنبال آن علامت (') آورده شود و سپس مقادیر ذکر گردند :

data type' expression or value

مثال :

ROM_type' ("01", "10", "00");

۳-۱۳ نوعهای پیشرفته داده

برای انجام طراحی‌های پیشرفته، دسترسی به نوعهای پیشرفته‌تر دیتا ضروری است. در اینجا نگاهی می‌کنیم به نوعهای داده فرعی، نوعهای چندبعدی و نوعهای گزارشی.

۱-۱۳-۳ نوعهای فرعی[1] داده

اگر تعریف زیربخشی از یک نوع داده، که قبلاً معرفی شده است مد نظر باشد باید در بخش اعلام حتماً از کلمه کلیدی subtype استفاده کنیم.

انواع فرعی داده محدود به پارامترهای مشخص‌کننده نوع اصلی هستند.

مقادیری که نوع فرعی جدید می‌تواند اختیار کند باید مشخص شود و باید یک زیربخش از مقدارهای نوع پایه، یا آنکه یک طول محدود از نوع پایه باشد. به طور مثال :

```
subtype my_int is integer range 0 to 3215;          -- Good
subtype byte is std_logic_vector (7 downto 0);      -- Good
type byte2 is array (7 downto 0) of std_logic;      -- OK
type byte3 is std_logic_vector (7 downto 0);        -- Error
subtype byte4 is array (7 downto 0) of std_logic;   -- Error
```

همچنان که در مثال فوق مشاهده می‌شود، array می‌تواند تنها برای تعریف کردن نوعهای جدید و نه نوعهای فرعی به کار رود. Std_logic_vector (7 downto 0) نمی‌تواند برای اعلام نوع جدید به کار رود. بایت نیز می‌تواند به عنوان یک نوع جدید دیتا اعلام شود (byte 2). عیب استفاده از نوع جدید به جای نوع فرعی این است که نوع اعلام شده کاملاً یک نوع جدید تلقی می‌شود، به آن معنی که دادن برشی از مقدار یک سیگنالی که از نوع اصلی دیتا بوده به سیگنالی که از نوع جدید است، بدون تبدیل کردن سیگنال، ممکن نخواهد بود.

1- Subtype

مثال (ناصحیح) :

```
Architecture bad of ex is
type byte is array (7 downto 0) of std_logic; -- So far so good
signal a: byte;
signal c: std_logic_vector (7 downto 0);
begin
        a<=c;          -- Error, a and c do not have the same data type
end;
```

اگر بایت (byte) به عنوان یک نوع فرعی تعریف شده بود مثال بالا بدون اعلام خطا از کامپایلر VHDL می‌گذشت.

در VHDL دو نوع فرعی داده از قبل تعریف شده وجود دارد :

```
subtype natural is integer range 0 to impl.defined   -- typical 2147483647
subtype positive is integer range 1 to impl.defined -- typical 2147483647
```

از نقطه نظر سنتز مهم نیست که یک دیتا با یک نوع فرعی یا یک نوع جدید و جداگانه اعلام و تعریف شده باشد.

۲-۱۳-۳ نوعهای داده چندبعدی[1]

در تئوری می‌توان برای یک نوع داده به هر تعداد، بعد تعیین کرد؛ ولی در عمل بیش از دو یا سه بعد مورد استفاده قرار نمی‌گیرد. به هنگام تعریف نوعهای چند بعدی داده باید از کلمه کلیدی array در بخش اعلام استفاده شود. به عنوان مثال :

```
type data4x8 is array (0 to 3) of std_logic_vector (7 downto 0);
type data3x4x8 is array (0 to 2) of data4x8;
```

داده از نوع data 4 × 8 یک داده دو بعدی (۸ × ۴ بیتی) می‌باشد، در حالی که داده از نوع data 3 × 4 × 8 یک دیتای سه بعدی (۸ ×۴ × ۳ بیتی) است. بخش اعلام نوع data 4 × 8 می‌تواند به صورت زیر نیز نوشته شود :

```
type byte is array (7 downto 0) std_logic;
type data4x8 is array (integer range 0 to 3) of byte;
```

1- Multidimensional

چند روش برای مقداردهی به یک بردار دو بعدی وجود دارد. می‌توان از اندیس آرایه استفاده کرد و یا کل بردارهای دو بعدی را با مکانیسم متراکم کردن در یک مرحله مقداردهی نمود.

مثال :

```
Architecture rtl of ex ix
type data4x8 is array (0 to 3) of std_logic_vector (7 downto 0);

signal d,e,f,g,h,t:        data4x8;
signal b1,b2,b3,b4:        std_logic_vector (7 downto 0);
begin
        d(0)<="01010110";
        d(1)<="10101000";
        d(2)<="01110110";
        d(3)<="10111011";
e<=(others=> (others=> '0')); -- Clear the whole 2-dim. signal

f(0)  (0)<= '1';
f(0)  (1)<= '0';
. . .

g<=(b1, b2, b3, b4);

h<=(others=> b1);
process (h)
begin
        ll: for n in 0 to 3 loop
                i(n)<=h(n);
        end loop;
end process;
. . .
end;
```

روش مقداردهی مورد انتخاب، به نوع کاربرد و خوانایی مورد نیاز بستگی دارد. اکثر ابزارهای سنتز که نوعهای داده دو بعدی را حمایت می‌کنند تمام روشهای ذکر شده در بالا را نیز می‌پذیرند. روشی که انتخاب می‌شود از نقطه نظر سنتز اهمیتی ندارد.

۳-۱۳-۳ نوعهای داده گزارشی[1]

نوع دیتای گزارشی شامل نوعهای مختلف داده است.

```
type<identifier> is record
      record definition
end record;
```

مثال :

```
Architecture beh of ex is
type data_data is record
      year:       integer  range 1996 to  2099;
      month:      integer  range 1 to  12;
      date:       integer  range 1 to  31;
      hour:       integer  range 0 to  23;
      minute:     integer  range 0 to  59;
      second:     integer  range 0 to  59;
      data:       std_logic_vector(31 downto 0);
end record;

signal d:data_date;
begin
      d.year<=1997;
      d.month<=4;
      d.date<=8;
      d.hour<=11;
      d.minute<=57;
      d.second<=22;
      d.data<=data_in;
end;
```

البته مقداردهی به داده گزارشی فوق را می‌توان با به کار بردن مکانیسم متراکم کردن نیز انجام داد :

```
d <= (1997, 4, 8, 11, 57, 22, data_in);
```

در بعضی از طراحی‌های خاص اعم از طراحی‌های ASIC و یا طراحی مدلهای VHDL برای

1- Record

شبیه‌سازی، نوعهای داده گزارشی نقش بسیار مهمی را خواهند داشت و ابزارهای پیشرفته سنتز نیز آنها را حمایت می‌کنند.

۳-۱۴ عناوین غیرواقعی[1]

Alias برای دادن نامهای دیگر به پارامترهای مختلف به کار می‌رود و باعث می‌شود که خواندن کد آسان‌تر شود.

مثال :

```
alias data : std_logic_vector (7 downto 0) is data_in (15 downto 8);
```

به کار بردن alias در نسخه استاندارد VHDL-87 تنها برای بعضی پارامترها معتبر بود اما این محدودیت در نسخه استاندارد VHDL-93 برطرف شده است و می‌توان از آن برای تعریف کردن توابع، نوعهای داده و غیره نیز استفاده کرد.

۳-۱۵ عملگرهای رابطه‌ای[2]

در VHDL چندین اپراتور رابطه‌ای تعریف شده است :

سمبل	مفهوم
=	تساوی
/=	نامساوی
<	کوچک‌تر از
>	بزرگ‌تر از
<=	کوچک‌تر یا مساوی
>=	بزرگ‌تر یا مساوی

نتیجه حاصل از اعمال این اپراتورها مقادیر صحیح (Ture) و یا غلط (False) خواهد بود. اپراتورهای فوق می‌توانند بر روی integer ها، bit_vector ها و std_logic_vector ها اعمال شوند. عملگرهای "=" و "/=" را می‌توان برای تمام نوعهای داده به کار برد.

1- Alias

2- Relational Operator

مثال :

```
Architecture rtl of ex is
type my_type is (on, off, idle, start);
signal statel, state2:     my_type;
signal a,b,c:              bit_vector (2 downto 0);
signal d,e,f,g:            std_logic_vector (2 downto 0);
signal my_int:             integer range 0 to 15;
begin
      a<=b when c= "010" else b;
      d<=e when d/= "111" else "000";
      e<=f when my_int<=12 else "001";
      f<="101" when e>=d else"110";
      g<=f when statel=state2 else "000";
      . . .
end;
```

کلیه عملگرهای رابطه‌ای توسط اکثر ابزارهای سنتز حمایت می‌شوند. ابزارهای ساده سنتز ممکن است از لحاظ اینکه کدام یک از نوعهای داده را می‌توانند به کار گیرند محدودیتهایی داشته باشند ولی ابزارهای پیشرفته سنتز عملگرهای رابطه‌ای را در مورد تمام انواع داده حمایت می‌کنند.

۳-۱۶ عملگرهای محاسباتی[1]

در نسخه استاندارد VHDL چندین اپراتور محاسباتی تعریف شده‌اند :

سمبل	مفهوم
+	جمع
-	تفریق
*	ضرب
/	تقسیم
abs	قدر مطلق
rem	باقیمانده
mod	مدول
**	توان

1- Arithmatic Operator

این عملگرهای محاسباتی برای عملیات بر روی متغیرهایی از نوع integer، real (به جز mod و rem) و time تعریف شده‌اند. از سوی دیگر این اپراتورها برای متغیرهایی از نوع std_logic_vector و std_ulogic_vector تعریف نشده‌اند. برای استفاده از این اپراتورها در محاسبات متغیرهای برداری باید از بسته‌های تعریف شده استفاده کرد. به طور مثال این کار قبلاً در بسته ieee.std_logic_unsigned انجام شده است.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;        -- Needed for "+" with
                                        -- std_logic_vectors

Entity ex is
 port ( a,b:  in   std_logic_vector (2 downto 0);
        c,d:  in   integer range 2 to 15;
        q1:   out  std_logic_vector (2 downto 0);
        q2:   out  integer range 0 to 30);
end;

Architecture rtl of ex is
begin
  q1 <=a + b;
  q2 <=c + d;
end;
```

اکثر ابزارهای سنتز استفاده از اپراتورهایی مثل '+'، '-' و همچنین '*' را برای integerها حمایت می‌کنند. با این وجود این اپراتورها برای نوعهای دیگر داده مثل real و time قابل قبول نیستند. از آنجایی که اعمال این اپراتورها بر پارامترهایی از نوع std_logic_vector مدنظر است، ابزارهای پیشرفته سنتز عملگرهای '+'، '-'، '*' و '**' را حمایت می‌کنند. آنچه ممکن است فرق کند بسته‌ای است که ابزار سنتز آن را می‌پذیرد، که در صورت تغییر یافتن ابزار سنتز ممکن است مجبور به تغییر بسته‌ای باشیم که باید در ابتدای کد VHDL برای عملگرهای محاسباتی تعریف شود (به فصل ۵ «کتابخانه، بسته و زیربرنامه‌ها» رجوع کنید).

۳-۱۷ مقدار اولیه[1]

در لحظه شروع شبیه‌سازی (زمان صفر) تمام سیگنال‌ها مقادیر اولیه خود را اختیار می‌کنند. این مقادیر بر حسب اینکه تحت نام چه نوعی از داده معرفی شده‌اند، فرق می‌کنند. اگر به نحو دیگری مشخص نشده باشد، کلیه سیگنال‌ها آن مقادیری را می‌گیرند که تحت عنوان «مقدار سمت چپ در تعریف نوع داده» شناخته شده است. نوع bit و نوع std_logic به صورت زیر تعریف می‌شوند :

```
type bit is ('0', '1');
type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

نشان[2] 'left به معنای اولین مقدار (مقدار سمت چپ) است که هنگام اعلام یک نوع داده در نظر گرفته شده است. بنابراین :

```
bit 'left = '0'
std_logic 'left = 'U'
```

با توجه به توضیحات بالا، مقدار اولیه متغیرهایی که از نوع bit تعریف می‌شوند '0' و مقدار اولیه متغیرهایی که از نوع std_logic تعریف شده‌اند 'U' خواهد بود. بنابراین اگر بخواهیم مقدار اولیه پارامتر خاصی را بدانیم، باید بفهمیم که نوع داده چگونه تعریف شده است.

چنان‌چه لازم باشد می‌توان مقدار اولیه‌ای را که برای هر نوع داده در استاندارد VHDL معین شده است با تعریف کردن یک مقدار ویژه تغییر داد. این تغییر باید یا در هنگام معرفی متغیر در entity و یا در بخش معماری برنامه انجام شود.

مثال :

```
Entity ex is
  port ( a:   in  std_logic:= '0';
         b,c: in  std_logic;
         q:   out bit);
end;

Architecture behv of ex is
signal i1:          std_logic:= '1';
```

1- Initial Value

2- Attribute

```
signal i2:      std_logic:= 'H';
signal i_vect:  std_logic_vector(3 downto 0):= "00LL";
begin
  i1<=a or b;
  . . .
end;
```

در مثال بالا، سیگنال‌های ورودی دارای مقادیر اولیه زیر خواهند بود :

سیگنال	مقدار
A	'0'
b	'U'
c	'U'
q	'0'
i1	'1'
i2	'H'
i_vect	"00LL"

در مثال بالا، اگر در معماری برنامه مقدار '0' را در لحظه 0ns + 0delta به متغیر داخلی i1 بدهیم (مثلاً به طور مستقیم آن را مقداردهی کنیم) در طول مدت زمان یک delta مقدار '1' را به خود می‌گیرد و سپس در لحظه 0ns + 1delta ارزش '0' را اختیار می‌کند.

در VHDL می‌توان ارزشهای اولیه برای سیگنال‌ها را در بخش entity با مدهای inout و out تعریف کرد. اگر بخواهیم به چندین سیگنال مقادیر اولیه یکسانی بدهیم این کار را می‌توانیم در یک خط به صورت زیر انجام دهیم :

```
signal a,b: std_logic_vector(2 downto 0):= "001";
```

ابزارهای سنتز مقادیر اولیه را حمایت نمی‌کنند. گرچه اغلب ابزارها وجود مقادیر اولیه را قبول می‌کنند اما آنها را با صدور یک اخطار نادیده می‌گیرند. بنابراین با مقداردهی اولیه یک سیگنال ورودی به 'H' نمی‌توان انتظار داشت که ابزار سنتز این سیگنال را به یک pull-up متصل کند. هر گونه pull-up باید در کد VHDL به‌طور خاص مشخص شود.

در مورد مقادیر اولیه این ریسک وجود دارد که نتیجه شبیه‌سازی و سنتز VHDL یکسان نباشد. سیگنال ورودی ممکن است در سخت‌افزار و در هنگام شروع به کار مدار مقدار متفاوتی داشته باشد و همین امر می‌تواند باعث ایجاد «ناهمخوانی» بین شبیه‌سازی کد VHDL و پیاده‌سازی آن در

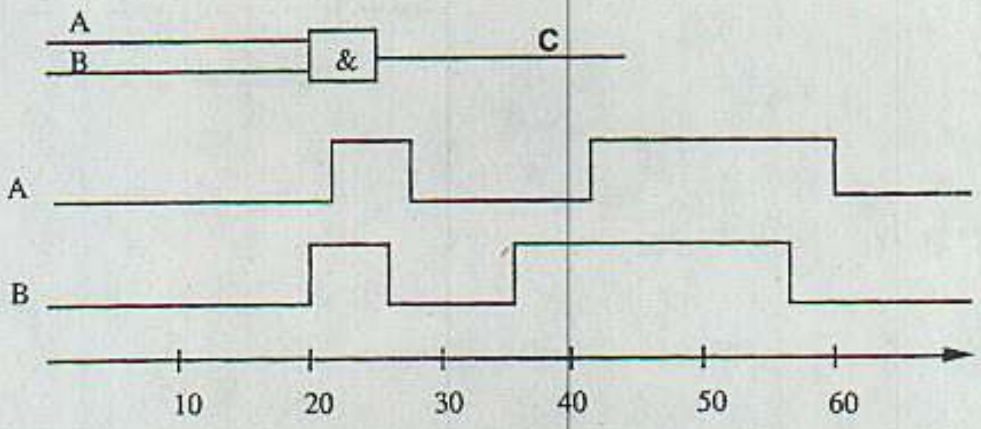
سطح گیتی شود. اگر سیگنال std_logic در طراحی به کار رفته و هیچ مقدار اولیه‌ای به آن داده نشده باشد، این سیگنال مقدار 'U' را اختیار خواهد کرد که یک مقدار اولیه مناسب است. اگر هیچ سیگنالی برای درایو کردن سیگنال std_logic وجود نداشته باشد سایر سیگنال‌هایی که وابسته به سیگنال std_logic بوده‌اند مقدار 'X' را به خود خواهند گرفت. این موضوع خطایی است که در شبیه‌سازی کد VHDL به آسانی کشف می‌شود. اما اگر مقدار اولیه‌ای که به سیگنال داده می‌شود '0' باشد، خطا محرزتر است اما کشف آن سخت‌تر خواهد بود.

۱۸-۳ تمرین

۱- علامت مقداردهی به یک سیگنال به چه شکلی است ؟

۲- کدام یک از دو مد تأخیر در استاندارد VHDL منظور شده است ؟

۳- سیگنال‌های خروجی c1 و c2 را بکشید.




```
c1 <= a and b after 10 ns;
c2 <= transport a and b after 10 ns;
```

۴- جداول زیر را کامل کنید. شکل موج سیگنال a در هر دو مورد داده شده است ؟

```
Architecture rtl of ex is
begin
 b<=a;
 c<=b;
end;
```

	0	10	10 + 1 delta	10 + 2 delta	20	20 + 1 delta	20 + 2 delta
a	0	1	1	1	0	0	0
b	0						
c	0						

```
Architecture rtl of ex is
begin
 c<=b;
 b<=a;
end;
```

	0	10	10 + 1 delta	10 + 2 delta	20	20 + 1 delta	20 + 2 delta
a	0	1	1	1	0	0	0
b	0						
c	0						

۵- (i) چرا کد VHDL زیر نادرست است ؟

```
q<=    a when sel= '0' else
       b when sel= '1';
```

(ii) کد VHDL فوق را بدون تغییر عملکرد به گونه‌ای صحیح بنویسید.

(iii) چه تغییری در ارتباط با دستور when-else در استاندارد VHDL-93 اعمال شده است ؟

(iv) آیا مثال ۵-(i) در استاندارد VHDL-93 معتبر است ؟

۶- سطوح مختلف خطا در دستور assert را بنویسید و بگویید کدام یک به صورت حالت اولیه پذیرفته شده است ؟

۷- کد VHDL زیر را با اعمال دستور assert دوباره بنویسید. دستور assert باید پیغام "too many ones" را وقتی که "111" = a را صادر کند.

```
Architecture rtl of ex is
signal a:  std_logic_vector(2 downto 0);
begin
   a<=c and b;
end;
```

۸- (i) سه گروه مختلف یک موضوع در کد VHDL را بنویسید.

(ii) چرا نوعهای داده غیر از bit و bit_vector مورد نیاز است ؟

(iii) دو نوع داده‌ای که توسط IEEE مشخص و تعریف شده‌اند کدامند و تفاوت بین آنها چیست ؟

۹- (i) عدد پایه در مقداردهی به رشته‌ای از بیت‌ها به چه منظور به کار می‌رود ؟

(ii) در استاندارد VHDL-87 عدد پایه برای مقداردهی به رشته‌ای از بیت‌ها برای چه نوع داده‌ای تعریف شده است ؟

(iii) چه تغییری در استاندارد VHDL-93 اعمال شده است ؟

۱۰- آیا یک integer می‌تواند هر مقداری را با هر عدد پایه به خود بگیرد ؟ اگر پاسخ مثبت است چگونه ؟

۱۱- برشی از یک آرایه به چه معناست ؟

۱۲- مزیت استفاده از مکانیسم متراکم‌سازی برای مقداردهی به سیگنال a در کد زیر چیست ؟

```
Architecture rtl of ex is
signal  a: std_logic_vector(31 downto 0);
begin
        a<=(others => '1' );
      . . .
end;
```

۱۳- مقدار بردار a پس از انجام مقداردهی‌های زیر چیست ؟

```
Architecture rtl of ex is
 signal a,b: std_logic_vector(4 downto 0);
 signal c:   std_logic_vector (0 to 1);
 begin
       a<=(1=> '0', 3=> '1', others=> b(2));
       b<=(1=> '1', 3=> '0', others=> c(1));
       c<= "10";
end;
```

۱۴- (i) به هنگام شروع به کار شبیه‌سازی کد زیر، چه مقدار اولیه‌ای برای سیگنال a منظور می‌شود ؟

```
type mytype is ('T' , 'R' , '0' , '1');
signal  a: mytype;
```

(ii) مقدار اولیه سیگنالی از نوع bit چیست ؟

(iii) مقدار اولیه سیگنالی از نوع std_logic چیست ؟

(iv) آیا می‌توان مقدار اولیه سیگنال را تغییر داد ؟ اگر پاسخ مثبت است، چگونه ؟

فصل چهارم

VHDL متوالی

طراحی با کمک پروسس‌های متوالی بعد جدیدی را در طراحی سخت‌افزاری برای طراحانی که در سطوح گیتی طراحی خود را انجام می‌دهند فراهم می‌کند. این فصل با نگاهی بر چگونگی عملکرد موازی و متوالی دستورات آغاز می‌شود. سپس تفاوتهای بین مقداردهی به متغیرها به طور موازی و متوالی بررسی می‌گردد. در انتها دستوارت متوالی زبان VHDL تشریح خواهد شد.

۱-٤ عملکرد موازی و متوالی دستورات

در زبان VHDL ساختارهایی وجود دارند که می‌توانند به هر دو صورت متوالی و موازی عمل کنند، اما اغلب دستورات فقط در بخشهای موازی یا متوالی VHDL عملکرد صحیح خواهند داشت. ساختارهای متداول زبان VHDL در هر بخش موازی و متوالی در زیر آورده شده است :

ساختارهای موازی VHDL

- Process statement
- When else statement
- With select statement
- Signal declaration
- Block statement

ساختارهای متوالی VHDL

- If-then-else statement
- Case statement
- Variable declaration
- Variable assignment
- Loop statement
- Return statement
- Null statement
- Wait statement

دستورات بسیاری نیز در هر دو قسمت موازی و متوالی VHDL معتبر هستند که در زیر به تعدادی از آنها اشاره شده است :

- Signal assignment
- Declaration of types and constants
- Function and procedure calls
- Assert statement
- After delay
- Signal attributes

دانستن چگونگی و محل به کارگیری دستورات و موازی یا متوالی بودن بخشهای گوناگون یک کد VHDL برای یک طراح بسیار ضروری است. در زیر مثالی از اینکه چه مکانی از یک کد VHDL به طور موازی یا متوالی عمل می‌کند آورده شده است.

به‌طور خلاصه می‌توان گفت که کد VHDL در سرتاسر بدنه برنامه، بجز در داخل process ها، function ها و producer ها به صورت موازی رفتار می‌کند.

```
Architecture rtl of ex is
concurrent declaration part
begin
 concurrent VHDL

 process(...)
 sequential declaration part
 begin
     sequential VHDL
 end process;

 concurrent VHDL
end;
```

Process, one concurrent statement

۴-۲ مقداردهی به متغیرها[1] و سیگنال‌ها[2]

متغیرها و سیگنال‌ها با نقشهایی کاملاً متفاوت در یک کد VHDL مورد استفاده قرار می‌گیرند. متغیرها برای اجرای دستورات متوالی و سیگنال‌ها برای اجرای دستورات موازی به کار می‌روند.

```
<target_identifier> := <selected_expression>;
<selected_expression>=
<identifier> [(and ! or ..! xor                    and ! nor ..)
<identifier >] ';'
```

تفاوت نماد مقداردهی به متغیرها و سیگنال‌ها با استفاده از سمبلهای ":=" و "<=" مشخص می‌شود. یک متغیر می‌تواند فقط در یک بخش متوالی VHDL معرفی و به کار برده شود، اما یک سیگنال می‌تواند در یک بخش موازی معرفی و در هر دو بخش متوالی و موازی به کار رود. چنان‌چه متغیر و سیگنال از یک 'نوع' تعریف شده باشند، می‌توان مقدار متغیر را به سیگنال نسبت داد و یا برعکس.

یک برنامه در CPU به صورت متوالی و سطر به سطر اجرا می‌شود، برنامه زیر از این نوع است :

```
variable temp_a, diff: integer_type;
temp_a := in_1;
diff:=temp_a - 2;
```

فرض کنید هر دستور برنامه در مدت زمان ΔT اجرا شود. جدول ۱-۴ چگونگی اثرپذیری متغیرها را در سه زمان نشان می‌دهد.

سیگنال	T - Δt	T	T + Δt
in_1	۲	۳	۳
Temp_a	۲	۳	۳
diff	۰	۱	۱

جدول ۱-۴ مقداردهی متوالی

1- Variable

2- Signal

در لحظه T - ΔT، متغیر in_1 مقدار ۲ را دارد و temp_a نیز مقدار ۲ را به خود می‌گیرد. سطر بعدی به diff مقدار ۰ = ۲ - ۲ را نسبت می‌دهد.

اگر برنامه مثال قبل در بخش موازی VHDL نوشته شود باید به جای متغیر از سیگنال استفاده گردد.

```
signal temp_a, diff : integer_type;
temp_a <= in_1;
diff <= temp_a - 2;
```

چنان‌چه بررسی زمانی برنامه در نظر باشد، می‌توان سطرهای برنامه فوق را به شکل زیر بازنویسی کرد:

```
temp_a(t) <= in_1(t - ΔT);
diff(t) <= temp_a(t - ΔT) - 2;
```

سیگنال in_1 جدیدترین مقدارش را در لحظه t - Δt به خود می‌گیرد (in_1 (t - Δt) = ۲). هر دو خط برنامه مثال فوق به طور هم‌زمان اجرا می‌شوند. این بدان معناست که آخرین مقدار سیگنال temp_a در سطر اول بعد از زمان Δt به آن نسبت داده خواهد شد.

سیگنال	t - Δt	t	t + Δt
in_1	۲	۳	۳
Temp_a	مقدار قدیمی	۲	۳
diff	مقدار قدیمی	مقدار قدیمی	۰

جدول ۲-۳ مقداردهی موازی

تفاوت بزرگ بین متغیرها و سیگنال‌ها این است که مقدار سیگنال پس از تأخیر «دلتا» به آن نسبت داده می‌شود در حالی که مقدار متغیر بلافاصله به آن واگذار می‌گردد.

لازم به یادآوری است که هیچ مقداری در بخش متوالی VHDL منقضی نمی‌گردد مگر هنگامی که در برنامه از دستور wait استفاده شود و یا هنگامی که کامپایلر به end یک پروسس که دارای لیست حساسیت است برسد، که در این صورت اگر تأخیری قید شده باشد رخ می‌دهد. چنان‌چه برنامه نوشته شده در بخش متوالی VHDL شامل هزاران خط باشد این اصل کماکان برقرار خواهد بود.

مثال :

```
process
variable c: std_logic_vector (1 downto 0);
begin
    if a=12 then        -- Line 1
        e<= "01";       -- Line 2
    ...
    if b<=10 then       -- Line 876
    ...
    c:= "10";           -- Line 2876
    ...
end process;
```

اگر پروسس بالا در لحظه ۲۲ns + ۴delta فعال شود، تمام خطوط در پروسس در طول زمان همان دلتا اجرا خواهند شد. مقدار "01" به سیگنال c در لحظه ۲۲ns + ۴delta و مقدار "10" به متغیر c در لحظه ۲۲ns + ۴delta نسبت داده می‌شود. سپس سیگنال c مقدار جدید خود را در لحظه ۵delta + ۲۲ns به دست می‌آورد در حالی که متغیر c فوراً و در همان لحظه ۲۲ns + ۴delta مقدار جدیدش را به خود می‌گیرد. خطوط در بخش متوالی یک کد VHDL به صورت سطر به سطر اجرا می‌شوند، به همین دلیل به آن VHDL متوالی گفته می‌شود. در VHDL موازی تنها زمانی خطوط اجرا می‌شوند که یک رویداد در لیست حساسیت پروسس رخ دهد.

مثال :

sum1 and sum2 are signals:

```
p0:process
begin
    wait for 10 ns;
    sum1<=sum1+1;
    sum2<=sum1+1;
end process;
```

sum1 and sum2 are variables:

```
p1:process
variable sum1, sum2: integer;
begin
    wait for 10 ns;
    sum1:=sum1+1;
    sum2:=sum1+1;
end process;
```

sum1, sum2 = Variables		
Time	Sum 1	Sum 2
0	0	0
10	1	2
10 + 1 delta	1	2
20	2	3
20 + 1 delta	2	3
30	3	4
30 + 1 delta	3	4

sum1, sum2 = Signals		
Time	Sum 1	Sum 2
0	0	0
10	0	0
10 + 1 delta	1	1
20	1	1
20 + 1 delta	2	2
30	2	2
30 + 1 delta	3	3

جدول ۳-۴ سیگنال‌ها و متغیرها

همان طور که مثال بالا نشان می‌دهد، عملکرد برنامه بسته به آنکه در آن متغیر استفاده شده باشد و یا سیگنال، بسیار متفاوت خواهد بود. تفاوت بین این دو برنامه در هنگام سنتز کردن آنها مشخص خواهد شد. بنابراین دانستن این نکته که چه زمانی باید دستورات متوالی را به کار برد و چه زمانی دستورات موازی، بسیار مهم است. به طور کلی نمی‌توان تعیین کرد که استفاده از متغیر یا سیگنال در برنامه مفیدتر خواهد بود. انتخاب آنها به عملکرد موردنیاز در برنامه بستگی دارد. امکان انتقال داده توسط متغیرها به خارج از بخش متوالی VHDL که در آن تعریف شده است وجود ندارد. بنابراین چنان‌چه بخواهیم در مثال قبل، به مقادیر sum1 و sum2 در پروسس P1 دسترسی داشته باشیم، به عنوان خروجی و یا استفاده از آنها در بخش دیگر برنامه، باید این متغیرها را از نوع سیگنال تعریف کرده و یا مقدار آنها را به یک سیگنال نسبت دهیم.

مثال :

```
Entity ex is
  port (sum1_sig, sum2_sig: out integer);
end;

Architecture behv of rtl is
begin
  p1: process
  variable sum1, sum2: integer;
  begin
     wait for 10 ns;
     sum1:=sum1+1;
     sum2:=sum1+1;
     sum1_sig<=sum1;
     sum2_sig<=sum2;
  end process;
end;
```

متغیرها فقط می‌توانند مقادیر موقتی داخل یک process، function و یا producer را در خود ذخیره کنند.

با توجه به دلایل فوق، در تعریف مفهوم متغیر در نسخه استاندارد VHDL-93 تغییرات اندکی صورت گرفته است. ***متغیرهای عمومی***[1] در نسخه استاندارد VHDL-93 تعریف شده‌اند که می‌توانند اطلاعات را به خارج از پروسس انتقال دهند.

مثال (VHDL-93):

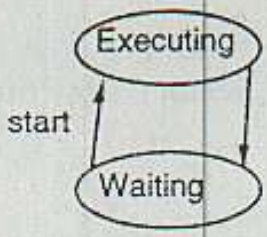
```
Architecture behv of ex is
shared variable  v: std_logic_vector (3 downto 0);
begin
 p0:process (a,b)
 begin
    v:=a & b;
 end process;

 p1: process (d,v)
  begin
   if v= "0110" then
     c<=d;
   else
     c<=(others=> '0');
   end if;
  end process;
. . .
end;
```

متغیرهای عمومی در بخش موازی کد VHDL قابل دسترسی نیستند و فقط در داخل پروسس‌ها کاربرد دارند. همچنین متغیرهای عمومی می‌توانند جزء لیست حساسیت یک پروسس قرار گیرند. با معرفی شدن این متغیرها مشکل اولیه حل نشده است و هنوز هیچ ابزار سنتزی متغیرهای عمومی را حمایت نمی‌کند. باید در هنگام استفاده از این نوع متغیرها دقت فراوانی به کار برد.

1- Global Variable

۳-٤ عبارت پروسس[1]

مفهوم پروسس از نرم‌افزار گرفته شده است و می‌تواند با یک برنامه متوالی مقایسه شود. اگر در بدنه یک برنامه VHDL چندین پروسس تعریف شده باشد همه آنها به طور موازی با هم اجرا می‌شوند.
یک پروسس می‌تواند در مرحله انتظار و یا در مرحله اجرا باشد.



شکل ۱-۴ وضعیت یک پروسس

چنان‌چه وضعیت در حالت انتظار باشد باید یک شرط برآورده شود مثلاً wati until clk = '1' . این بدان معناست که زمانی پروسس شروع به کار می‌کند که clk به سطح منطقی '1' رود. در این صورت پروسس به وضعیت اجرا می‌رود. با اجرای یک بار کد برنامه، پروسس به وضعیت انتظار می‌رود و تا بالا رفتن مجدد clk در این حالت باقی می‌ماند.

The syntax for the process is:

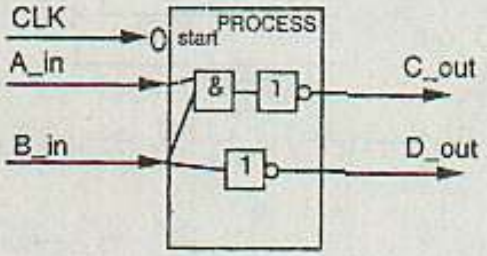
```
[<process_name>:] process [(<sensitivity_list>)]
        [<process_declarative_part>]
begin
      <process_statement_part>
end process [<process_name>];
```

Syntax (VHDL-93):

```
[<process_name>:] process [(<sensitivity_list>_] [is]
      . . .
begin
      . . .
end process [<process_name>];
```

1- Process

مدت زمان لازم برای بازگشت از مرحله اجرا به مرحله انتظار زمان دلتا (Δt) است، به این معنی که برای اجرای پروسس هیچ زمانی صرف نخواهد شد. به بیان دیگر می‌توان پروسس را به عنوان یک حلقه بی‌نهایت بین begin و end آن در نظر گرفت.



شکل ۲-۴ یک پروسس که با سطح منطقی '0' شروع به کار می‌کند.

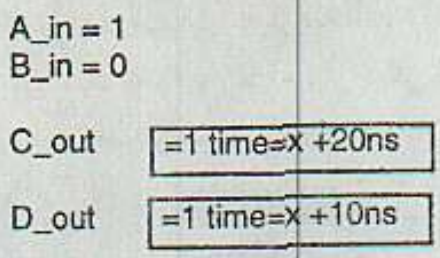
پروسس شکل ۲-۴ را می‌توان به صورت زیر توصیف کرد :

```
sync_process:process
begin
      wait until clk= '0';
      c_out <= not (a_in and b_in);
      d_out <= not b_in;
end process sync_process;
```

فعالیت پروسس با پایین رفتن سیگنال clk شروع می‌شود. دو عبارت داخل آن اجرا شده و به مرحله انتظار باز می‌گردد. در این حالت نیازی نیست که برنامه‌نویس یک حلقه را در داخل پروسس تعریف کند، زیرا پروسس با برقراری شرط اولیه از ابتدا شروع به کار می‌کند. در این مدل دو عبارت در مدت زمان Δt، که برابر با کمترین سرعت اجرایی شبیه‌ساز است، اجرا می‌شوند. برای تغییر زمان اجرای هر کدام می‌توان تأخیری اعمال کرد. اعمال تأخیر در زیر مدل شده است :

```
c_out <= not (a_in and b_in) after 20 ns;
d_out <= not  b_in after 10 ns;
```

بر اساس برنامه فوق متغیر c_out به اندازه ۲۰ نانوثانیه و متغیر d_out به اندازه ۱۰ نانوثانیه پس از آغاز فعالیت پروسس تأثیر می‌پذیرند. شبیه‌ساز نتیجه c_out و d_out را در ردیف رویدادهای زمانی هر کدام قرار می‌دهد (شکل ۳-۴).



شکل ۳-۴ ردیف رویدادهای زمانی شبیه‌ساز در زمان x

در VHDL دو نوع پروسس وجود دارد :

۱- *پروسس ترکیبی*[1]

۲- *پروسس پالسی*[2]

پروسس‌های ترکیبی جهت طراحی منطق ترکیبی در سخت‌افزار به کار می‌روند.

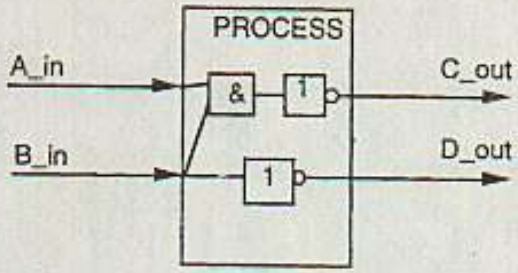
پروسس‌های پالسی منجر به استفاده از فلیپ فلاپ و در صورت لزوم استفاده از منطق ترکیبی می‌شوند.

۱-۳-۴ پروسس ترکیبی

در یک پروسس ترکیبی کلیه سیگنال‌های ورودی (تمام سیگنال‌هایی که در سمت راست علامت '⇐' و سیگنال‌هایی که داخل عبارت if/case قرار دارند) باید در لیست حساسیت پروسس وارد شوند. چنان‌چه سیگنالی داخل لیست حساسیت قرار نگیرد، پروسس مطابق با منطق ترکیبی در سخت‌افزار رفتار نخواهد کرد. در سخت‌افزار سیگنال‌های خروجی در صورتی تغییر می‌کنند که یک یا چند سیگنال ورودی به بلوک ترکیبی تغییر کند. اگر یکی از این سیگنال‌ها از لیست حساسیت حذف شود، پروسس با تغییر سیگنال حذف شده فعال نمی‌شود و به دنبال آن مقدار جدیدی به خروجی پروسس اعمال نخواهد شد. امکان حذف سیگنال از لیست حساسیت پروسس در استاندارد VHDL به منظور طراحی مدلهایی که فقط برای شبیه‌سازی به کار می‌روند و نه برای ساخت و پیاده‌سازی طرح در سخت‌افزار، وجود دارد. با حذف این سیگنال از لیست حساسیت یک کد VHDL، شبیه‌سازی و تحلیل سخت‌افزاری آن نتایج متفاوتی خواهند داشت که خطای مهمی است، زیرا هدف از به کارگیری VHDL توصیف عملکردی یک سخت‌افزار می‌باشد.

1- Combinational Process

2- Clocked Process



شکل ۴-۴ پروسس ترکیبی

پروسس شکل ۴-۴ را می‌توان به صورت زیر توصیف کرد :

```
comb_process: process (a_in,b_in)
begin
        c_out <= not (a_in and b_in) after 20 ns;
        d_out <= not b_in after 10 ns;
end process comb_process;
```

در داخل پرانتز عبارت پروسس لیست سیگنال‌هایی که با تغییر آنها پروسس شروع به کار می‌کند قرار دارد.

اگر a_in از لیست حساسیت حذف شود، c_out مقدار قبلی خودش را حفظ می‌کند و تا آن زمان که b_in تغییر کند مقدار جدید را به خودش نمی‌گیرد. این امر منجر به عدم تطبیق بین نتایج حاصل از شبیه‌سازی نرم‌افزاری و پیاده‌سازی سخت‌افزاری کد VHDL خواهد شد.

مثال (نامناسب) :

```
comb_process: process (b_in)
begin
        c_out <= not (a_in and b_in) after 20 ns;
        d_out <= not b_in after 10 ns;
end process comb_process;
```

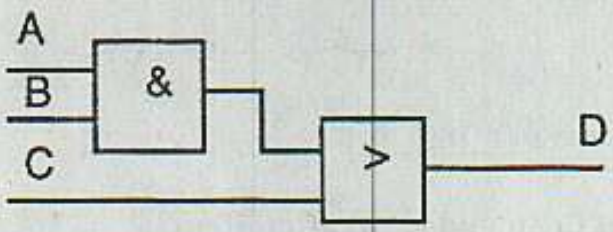
پروسس‌های ترکیبی برای طراحی در سطح ***جریان داده‌ای***[1] مثلاً مسیر عبور داده در یک CPU مناسب هستند.

1- Data Flow

مثال (پروسس ترکیبی) :

```
process (a,b,c)
begin
   d<= (a and b) or c;
end process;
```

نتیجه سنتز این پروسس در شکل ۵-۴ نشان داده شده است.



شکل ۵-۴ نتیجه سنتز

از دیدگاه طراحی با پروسس‌های ترکیبی باید با هر بار اجرا شدن پروسس به کلیه سیگنال‌های خروجی آن، مقداری داده شود. اگر این شرط محقق نشود، سیگنال مقدار قبلی خودش را نگه می‌دارد. ابزار سنتز این مسأله را با در نظر گرفتن یک لچ برای خروجیی که هیچ مقداری به آن داده نشده است حل می‌کند. از آنجایی که این سیگنال خروجی مقدار خودش را نگه می‌دارد، ارزش این لچ هیچ‌گاه تغییر نخواهد کرد. از لحاظ عملکرد، کد VHDL و سخت‌افزار مشابه یکدیگر خواهند بود. هدف از پروسس ترکیبی، تولید و پیاده‌سازی منطق ترکیبی است. اگر لچ‌ها لحاظ شوند تعداد گیت‌ها افزایش یافته و دقت زمان‌بندی از بین می‌رود. نکته دیگر این است که لچ‌ها به طور معمول قوانین آزمایش و تست به منظور تولید خودکار ***بردارهای آزمون***[1] را نقض می‌کنند (به فصل ۱۲ مراجعه کنید، «آشنایی با روشهای آزمایش»). پروسس‌هایی که منجر به ایجاد این نوع لچ‌ها می‌شوند اصطلاحاً ***پروسس‌های ترکیبی ناقص***[2] نامیده می‌شوند (به فصل ۱۴ رجوع کنید، «خطاهای متداول در VHDL و چگونگی اجتناب از آنها»). راه حل ساده است : کلیه سیگنال‌هایی که در داخل پروسس «خوانده» می‌شوند به داخل لیست حساسیت پروسس‌های ترکیبی وارد شوند.

1- Test Vector

2- Incomplete Combinational Process

۲-۳-۴ پروسس پالسی

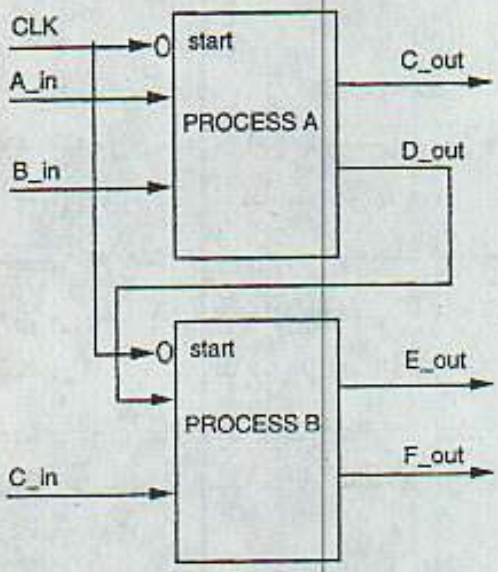
پروسس‌های پالسی *هم‌زمان*[1] هستند و می‌توان چندین نوع از این پروسس‌ها را با یک پالس ساعت یکسان به هم متصل کرد. متداول است که هیچ پروسسی شروع به کار نکند مگر با دریافت لبه پایین‌رونده پالس ساعت ورودی آن. این بدان معناست که با اعمال پالس ساعت و شروع به کار پروسس، داده ثابت و پایدار است و داده بعدی با شروع به کار مجدد پروسس خارج می‌شود.

d_out به عنوان خروجی پروسس A به ورودی پروسس B متصل است. تنها چیزی که به نظر مهم می‌آید اطلاعات روی پایه d_out است که باید قبل از اعمال کلاک و شروع به کار پروسس‌ها پایدار باشد. کمترین دوره تناوب پالس ساعت با بیشترین زمان مورد نیاز پروسس B برای پایدار کردن مقدار d_out پس از اعمال کلاک تعیین می‌شود. در شبیه‌سازی VHDL این زمان ۱۰ نانوثانیه در نظر گرفته شده است (برای این مثال).

کد VHDL مربوط به شکل ۶-۴:

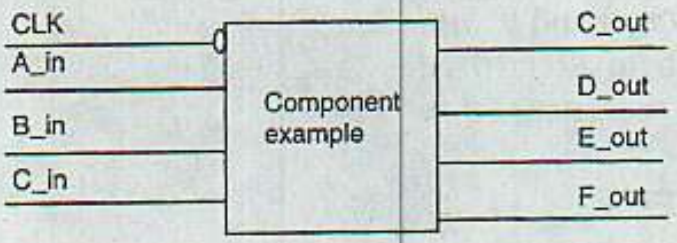
```
A_process:process
begin
  wait until clk= '0';
  c_out <= not (a_in and b_in);
  d_out <= not b_in after 10 ns;
end process A_process;

B_process:process
begin
  wait until clk= '0';
  e_out <= not (d_out and c_in);
  f_out <= not c_in;
end process B_process;
```

1- Synchronous



شکل ۶-۴ پروسس‌های پالسی با یک کلاک راه‌انداز

این دو پروسس می‌توانند یک **جزء ترکیبی**[1] ایجاد کنند (شکل ۷-۴).



شکل ۷-۴ پروسس‌های A و B تلفیق شده به فرم یک جزء ترکیبی

```
Entity comp_ex is
  port (clk, a_in, b_in, c_in: in    std_logic;
        d_out :                out   std_logic;
        c_out, e_out:          out   std_logic);
end;

Architecture rtl of comp_ex is
--* internal signal declaration --*
begin
```

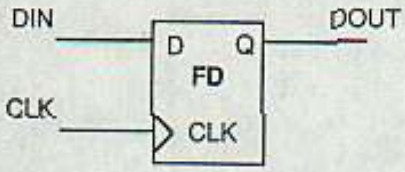
1- Component

```
    A_process:process
    begin
    . . .
    end process A_process;

    B_process:process
    begin
    . . .
    end process B_process;
end;
```

مقداردهی‌های انجام شده در داخل پروسس‌های پالسی منجر به ایجاد فلیپ فلاپ می‌شوند.

مثالی از یک پروسس پالسی :

```
example:process
begin
    wait until clk= '1';
    dout <= din;
end process example;
```



شکل ۸-۴ نتیجه سنتز

مثال قبل چگونگی ترجمه شدن یک پروسس پالسی به فلیپ فلاپ را نشان می‌دهد. همان طور که در شکل ۸-۴ مشخص است، din فوراً به dout منتقل می‌شود. در هنگام سنتز برنامه، این انتقال بیان‌کننده مشخصات یک فلیپ فلاپ خواهد بود. در یک مدل دقیق‌تر انتقال dout ⇐ din بعد از یک نانوثانیه انجام می‌گیرد.

متغیرها نیز در پروسس‌ها موجب ایجاد فلیپ فلاپ می‌شوند. چنان‌چه یک متغیر را قبل از آنکه به آن مقداری نسبت داده شود، 'بخوانیم' یک فلیپ فلاپ برای آن منظور خواهد شد.

مثال :

```
process
variable count: std_logic_vector (1 downto 0);
begin
```

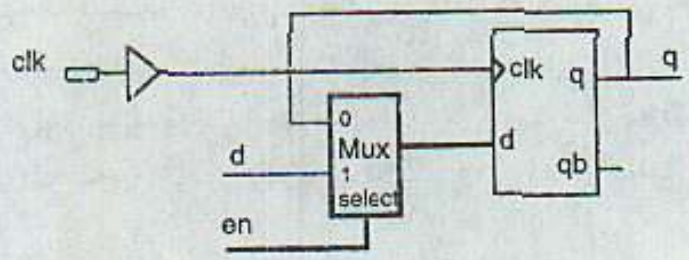
```
        wait until clk='1';
        count:=count + 1;
        if count= "11" then
                q<= '1';
        else
                q<= '0';
        end if;
end process;
```

در سنتز مثال بالا با متغیر count سه فلیپ فلاپ تشکیل خواهد شد. یکی برای سیگنال q و دو تا برای متغیر count، زیرا قبل از آنکه مقداری برای آن تعیین کنیم آن را 'خواندیم' (count := count + 1;).

چنان‌چه در یک پروسس پالسی مقداردهی به سیگنال انجام نشود، سیگنال مقدار قبلی خود را حفظ می‌کند. نتیجه سنتز این توصیف فیدبکی از سیگنال خروجی فلیپ فلاپ و انتقال به ورودی آن از طریق یک مالتی پلکسر خواهد بود (شکل ۹-۴). این روش باعث می‌شود که طرح همچنان هم‌زمان بماند و قابلیت تست کردن آن از بین نمی‌رود (به فصل ۱۲ رجوع کنید، «آشنایی با روشهای آزمایش»). به عنوان انتخابی دیگر می‌توان پالس ساعت و سیگنال 'en' را از طریق یک گیت به پایه کلاک فلیپ فلاپ متصل کرد (شکل ۱۰-۴). این انتخاب از نقطه نظر روشهای تست و طراحی بسیار نامناسب‌تر از مورد اول است.

مثال (مناسب) :

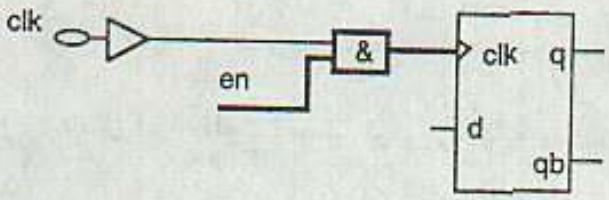
```
process
begin
        wait until clk='1';
        if en='1' then
                q<=d;
        end if;
end process;
```



شکل ۹-۴ طرح هم‌زمان

مثال (نامناسب) :

```
clk2<= clk and en;
process
begin
      wait until clk2= '1';
      q<=d;
end process;
```



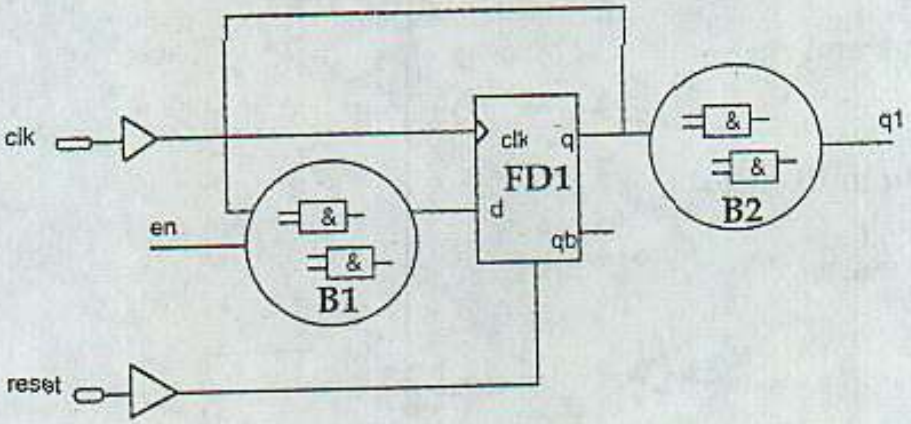
شکل ۱۰-۴ ترکیب گیتی پالس ساعت فلیپ فلاپ

پروسس پالسی می‌تواند به منطق ترکیبی نیز منتج شود.

مثال :

```
process (clk,resetn)
begin
      if reset = '1' then
            q<=(others=> '0');
      elsif clk'event and clk= '1' then
            if en= '1' then
                  q<=a + b;
            end if;
       end if;
end process;
```

منطقی که بر اثر مقداردهی به سیگنال‌ها در یک پروسس پالسی به‌وجود می‌آید در سمت 'چپ' فلیپ فلاپ، قبل از ورودی آن، قرار می‌گیرد. بنابراین در مثال بالا یک جمعگر و یک مالتی پلکسر در ورودی فلیپ فلاپ منظور می‌شود. ساختار منطقی در شکل ۱۱-۴ نشان داده شده است. به طور مثال بلوک منطقی B1 در پروسس پالسی که توصیف‌کننده فلیپ فلاپ FD1 است قرار دارد. از طرف دیگر باید کد VHDL توصیف‌کننده B2 در یک پروسس ترکیبی تعریف شود. البته امکان اضافه کردن بلوک منطقی B2 در پروسس پالسی وجود دارد که این کار توصیه نمی‌شود.



شکل ۱۱-۴ بلوک منطقی ترکیبی ـ پالسی

کد VHDL شکل ۱۱-۴ را می‌توان به صورت زیر نوشت :

```
Architecture rtl of ex is
signal q: std_logic;
begin
 FD1_B1:process (clk,reset)
 begin
      if reset='1' then
        q<= '0';
      elsif clk'event and clk= '1' then
        if en= '1' then
          q<=. . .                    -- some boolean expression (B1)
        end if;
      end if;
 end process;
 q1<= q and . . .                     -- som boolean expression (B2)
end;
```

بلوک منطقی B2 می‌تواند در یک پروسس ترکیبی نیز توصیف شود.

٤-٤ عبارت If

یک دستور If شامل یک یا چند عبارت داخلی است که بسته به برقراری شرط یا شرایط خاص یکی از عبارتها انتخاب می‌شوند.

عبارت If همارز عبارت when else در بخش موازی VHDL است.

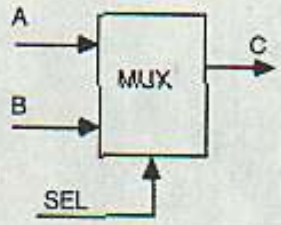
Syntax:
if <condition> **then** <sequence_of_statements>
[**elsif** <condition> **then** <sequence_of_statements>] . . .
[**else** <sequence_of_statements>]
end if;

<condition>=
<expression> [<relational_operator> <expression>] . . .
<relational_operator>=
= /= < <= > >=

مثال :

```
if sel = '1' then
  c <= b;
else
  c <= a;
end if;
```

نتیجه سنتز مثال بالا در شکل ۱۲-۴ نشان داده شده است.



شکل ۱۲-۴ نتیجه سنتز

مثال :

```
if sel = '1' then
  c <= '1'; . . .
elsif John = "1001" then
  c<= '0'; . . .
elsif David > John then
  c<= '1'; . . .
else
  c<= 'X'; . . .
end if;
```

در VHDL مجاز به استفاده از چندین else-if هستیم اما فقط یک else باید در عبارت If به کار رود. به ELSIF و END IF توجه شود.

مثال :

```
Entity ex_if is
 port (ssel:     in  std_logic;
       a,b,syn:  in  std_logic;
       sout:     out std_logic);
end;

 Architecture rtl of ex_if is
 begin
 process (ssel,syn, a,b)
 begin
  if ssel= '0' and syn= '1' then
     sout<=a;
  else
     if ssel= '1' and syn= '0' then
       sout<=b;
     else
       sout<= '0';
     end if;
   end if;
 end process;
 end;
```

مثال :

```
Entity ex_if2 is
 port (a,b:  in  integer;
       c:    out boolean);
end;

 Architecture rtl of ex_if2 is
 begin
 p1: process (a,b)
 begin
  if a>b then
     c<=true;
  else
```

```
      c<=false;
   end if;
 end process;
end;
```

۵-٤ عبارت Case

این دستور همارز دستور with select در بخش موازی VHDL است. استفاده از عبارت case در اغلب موارد به فهم بهتر کد VHDL کمک می‌کند.

Syntax:

```
case <expression> is
      when <choice> => <sequence_of_statements>;
      when <choice> => <sequence_of_statements>;
      . . .
      [when others => [<sequence_of_statements>] ;]
end case;
<choice> = <choice> [| <choice> [. . .]]
| = or
```

مثال :

```
Entity case_mux is
port (a,b,sel:  in   bit;
      c:        out bit);
end;

Architecture rtl of case_mux is
begin
      p1: process (sel,a,b)
      begin
       case sel is
         when '0' => c<=a;
         when '1' => c<=b;
       end case;
      end process;
end;
```

کلیه انتخابها و حالتهای مختلف باید در عبارات case معین گردند. در ضمن نمی‌توان یک انتخاب را مشترکاً در چندین عبارت when برشمرد. اگر متغیر case دارای مقادیر و حالتهای بسیاری باشد، معین کردن هرحالت کار مشکلی‌خواهد بود. در این مورد می‌توان از کلمه "others" استفاده کرد.

مثال :

```
Entity case_ex2 is
 port ( a:   in    integer range 0 to 30;
        c:   out   integer range 0 to 6);
end;

Architecture rtl of case_ex2 is
begin
      p1: process (a)
      begin
       case a is
         when 0 =>          b<=3;
         when 1 | 2=>       b<=2;
         when others  =>    b<=0;
       end case;
      end process;
end;
```

معمولاً از داده نوع std_logic در طراحی‌ها استفاده می‌شود. از آنجایی که std_logic دارای ۹ ارزش متفاوت است بنابراین کلیه مقادیر باید توسط عبارت case تحت پوشش قرار گیرند. در این مورد نیز می‌توان از کلمه "others" استفاده کرد.

مثال :

```
Entity case_ex3 is
 port ( a,b,sel:  in   std_logic;
        c:        out  std_logic);
end;

Architecture rtl of case_ex3 is
begin
      p1: process (sel,a,b)
      begin
       case sel is
         when '0' =>          c<=a;
         when others  =>      c<=b;
       end case;
      end process;
end;
```

چنان‌چه سیگنال ورودی sel در مثال بالا ارزشی غیر از '0' به خود بگیرد، به سیگنال c مقدار سیگنال b نسبت داده می‌شود. در شبیه‌سازی VHDL این مقدار می‌تواند یکی از ۹ مقداری باشد که داده از نوع std_logic می‌تواند به خود بگیرد، اما در سخت‌افزار آنچه غیر از '0' است '1' تلقی خواهد شد. ابزارهای سنتز تشخیص می‌دهند که سخت‌افزار فقط یکی از دو مقدار '0' و '1' را به خود می‌گیرد که این امر منجر به سنتز صحیح کد VHDL می‌شود.

همچنین معین کردن گستره لیست انتخاب با به کار بردن 'to' و یا 'downto' امکان‌پذیر است.

مثال :

```
Entity case_ex4 is
 port ( a:   in   integer range 0 to 30;
        q:   out  integer range 0 to 6);
end;

Architecture rtl of case_ex4 is
begin
      p1: process (a)
      begin
       case a is
         when 0 =>                  q<=3;
         when 1 to  17=>            q<=2;
         when 23 downto  18 =>      q<=6;
         when others  =>            q<=0;
       end case;
      end process;
end;
```

از آنجایی که برای بردار نمی‌توان گستره تعریف کرد بنابراین امکان معین کردن گستره انتخاب به صورت بردار وجود ندارد.

مثال (نامناسب) :

```
Entity case_ex5 is
 port ( a:   in   std_logic_vector(4 downto 0);
        q:   out  std_logic_vector(2 downto 0));
end;

Architecture rtl of case_ex5 is
begin
      p1: process (a)
```

```
    begin
     case a is
       when "00000" =>              q<= "011";
       when "00001" to "11110"=> q<= "010";    --Error
       when others  =>              q<= "000";
     end case;
    end process;
end;
```

چنان‌چه تعریف گستره مورد نظر باشد ابتدا باید std_logic_vector به یک داده از نوع integer تبدیل شود. این کار با تعریف یک متغیر از نوع integer در پروسس و سپس استفاده از تابع تبدیل‌کننده conv_integer انجام می‌گیرد. چنان‌چه بخواهیم از یک integer به جای یک بردار در لیست انتخابها استفاده کنیم باید روندی مشابه کار بالا را انجام دهیم.

مثال (مناسب) :

```
Entity case_ex6 is
 port ( a:   in   std_logic_vector (4 downto 0);
        q:   out  std_logic_vector (2 downto 0));
end;

Architecture rtl of case_ex6 is
begin
     p1: process (a)
     variable int: integer range 0 to 31;
     begin
      int:=conv_integer(a);
     case int is
       when 0 =>                  q<= "011";
       when 1 to 30 =>            q<= "010";
       when others  =>            q<= "000";
     end case;
    end process;
end;
```

نتیجه سنتز به علت استفاده از تابع تبدیل‌کننده تغییر نخواهد کرد.

اگر لازم باشد چندین بردار در لیست انتخاب یک عبارت case دخالت داشته باشند، از آنجایی که متغیر case باید ساکن و پایدار باشد، نمی‌توان آنها را به عنوان یک انتخاب با هم ترکیب کرد.

مثال (نامناسب) :

```
Entity case_ex7 is
 port ( a,b:  in    std_logic_vector (2 downto 0);
        q:    out   std_logic_vector (2 downto 0));
end;

Architecture rtl of case_ex7 is
begin
      p1: process (a,b)
      begin
      case a & b is                          -- Error
        when "000000"  => q<= "011";
        when "001110"  => q<= "010";
        when others    => q<= "000";
      end case;
      end process;
end;
```

به عنوان یک راه حل می‌توان یک متغیر داخل پروسس تعریف کرد، مقدار a & b را به آن نسبت داد و از آن به عنوان متغیر case استفاده کرد.

مثال (مناسب) :

```
Architecture rtl of case_ex8 is
 begin
      p1: process (a,b)
      variable int: std_logic_vector(5 downto 0);
      begin
       int: =a  & b;
       case int  is
         when "000000"  => q<= "011";
         when "001110"  => q<= "010";
         when others    => q<= "000";
       end case;
       end process;
end;
```

راه حل دیگر تعریف کردن یک subtype و استفاده از آن در عبارت case است.

مثال (مناسب) :

```
Architecture rtl of case_ex9 is
begin
      p1: process (a,b)
      subtype mytype is std_logic_vector (5 downto 0);
      begin
      case mytype '(a  &  b) is
        when "000000"  => q<= "011";
        when "001110"  => q<= "010";
        when others      => q<= "000";
      end case;
      end process;
end;
```

۶-٤ مقداردهی‌های چندگانه

در بخش موازی VHDL و در مقداردهی به سیگنال‌ها نیاز به وجود یک درایور یا محرک است که به طور معمول مطلوب نخواهد بود. در بخش متوالی امکان مقداردهی چند باره یک سیگنال بدون نیاز به چند محرک در یک پروسس وجود دارد. این روش مقداردهی به یک سیگنال می‌تواند به عنوان نسبت دادن مقادیر اولیه به سیگنال‌ها به کار رود. مثالی که در زیر آورده شده است از هر دو جهت شبیه‌سازی و سنتز نتایج یکسانی ایجاد می‌کند :

مثال ۱ :

```
Architecture rtl of ex1 is
begin
 p1:process(a)
 begin
    case a is
      when "00" =>    q1<= '1';
                      q2<= '0';
                      q3<= '0';
      when "10" =>    q1<= '0';
                      q2<= '1';
                      q3<= '1';
      when others =>  q1<= '0';
                      q2<= '0';
                      q3<= '1';
```

```
    end case;
 end process;
end;
```

مثال ۲ :

```
Architecture rtl of ex2 is
begin
 p1:process(a)
 begin
      q1<= '0';
      q2<= '0';
      q3<= '0';
      case a is
           when "00" =>    q1<= '1';
           when "10" =>    q2<= '1';
                           q3<= '1';
           when others =>  q3<= '1';
      end case;
 end process;
end;
```

با مقایسه دو برنامه فوق در می‌یابیم که تعداد مقداردهی‌های به کار رفته در مثال ۲ کمتر از مثال ۱ است. اگر از if-then-else استفاده شود اصل مشابهی را می‌توان اعمال کرد. به عنوان توضیح بیشتر می‌توان گفت که هیچ زمانی در اجرای دستورات داخل پروسس صرف نمی‌شود و فقط در مقداردهی‌ها مقادیر جدید جایگزین مقادیر قبلی می‌شوند. از سویی دیگر چنان‌چه به یک سیگنال در پروسس‌های مختلف کد VHDL مقادیر مختلفی نسبت داده شود نیاز به چند محرک خواهد بود.

۷-٤ عبارت Null

در VHDL عبارتی به مفهوم «هیچ کاری انجام نشود» وجود دارد. این دستور می‌تواند پس از انجام مقداردهی‌های اولیه در پروسس و در حالتهایی از عبارت case که‌نیازی به‌تغییر مقادیر سیگنال‌ها نیست به کار برده شود.

مثال :

```
Architecture rtl of ex2 is
begin
 p1:process(a)
 begin
```

```
    q1<= '0';
    q2<= '0';
    q3<= '0';
    case a is
        when "00" =>    q1<= '1';
        when "01" =>    q2<= '1';
                        q3<= '1';
        when others =>  null;
    end case;
 end process;
end;
```

در مثال بالا می‌توان از دستور null استفاده نکرد اما اگر این دستور به کار برده شود خوانایی برنامه بیشتر خواهد بود.

۸-٤ عبارت Wait

چهار روش گوناگون توصیف یک عبارت wait در پروسس‌ها وجود دارد :

```
1. process (a,b)
2. wait until a=1;
3. wait on a,b;
4. wait for 10 ns;
```

چنان‌چه عبارت wait on a,b در انتهای پروسس آورده شود مورد اول و سوم یکسان خواهند بود. بنابراین دو پروسس زیر نتیجه مشابهی دارند :

Example 1:

```
Process(a,b)
begin
    if a>b then
      q<= '1';
    else
      q<= '0':
    end if;
end process;
```

Example 2:

```
Process
begin
    if a>b then
      q<= '1';
    else
      q<= '0':
    end if;
    wait on a,b;
end process;
```

پروسس مثال ۱ با هر بار تغییر مقدار سیگنال‌های a یا b شروع به کار خواهد کرد (a 'event یا b 'event). برای آنکه نتیجه مثال ۲ کاملاً مشابه مثال ۱ باشد باید wait on a, b در انتهای پروسس قرار گیرد؛ زیرا کلیه پروسس‌ها از ابتدا شروع به اجرا می‌کنند تا به اولین عبارت wait برسند. با توجه به استاندارد VHDL استفاده از لیست حساسیت کاملاً مشابه به کارگیری wait on در انتهای پروسس است. چنان‌چه wait on هر جای دیگری جز انتهای پروسس قرار گیرد، مقادیر سیگنال‌های خروجی متفاوت خواهد بود (به مثال wait زیر توجه کنید).

در صورتی که از لیست حساسیت استفاده شود مجاز به به‌کارگیری دستور wait در پروسس نخواهیم بود. لازم به یادآوری است که امکان استفاده از چندین دستور wait در یک پروسس وجود دارد.

wait until a = '1';	به معنای این است که برای برآورده شدن wait و ادامه اجرای کد نوشته شده ضروری است که سیگنال a تغییر وضعیت دهد و مقدار جدیدش '1' باشد، به طور مثال دارای لبه بالارونده باشد.
wait on;	به مفهوم تغییر یکی از سیگنال‌های a یا b به منظور برآورده شدن شرط wait است.
wait for 10ns;	به مفهوم توقف اجرای پروسس به اندازه 10ns و سپس ادامه اجرای آن است. می‌توان دستور wait for را به صورت زیر نیز به کار برد.

```
constant period: time := 10  ns;
wait for  2  * period;
```

حالتهای ۲، ۳ و ۴ می‌توانند به فرم زیر با هم ترکیب شوند :

```
wait on a until b = '1' for 10ns;
```

اما حالت ۱ هیچ گاه نمی‌تواند با حالتهای دیگر ترکیب شود. فرض کنید دستور wait به شکل زیر در برنامه به کار برده شده باشد :

wait until *a = '1' for 10ns;*

شرط wait زمانی برآورده می‌شود که یا سیگنال a تغییر کند و مقدارش '1' شود و یا اجرای پروسس به اندازه حداکثر 10ns توقف داشته باشد. شرط wait در اصل نوعی 'یا' است. به طور مثال توقف تا زمان تغییر حالت سیگنال a و '1' شدن آن 'یا' ادامه اجرای پروسس پس از 10ns، برآورده کننده شرط wait هستند.

مثالهای زیر که اختلاف دستورات مختلف wait را مشخص خواهند کرد، نتایج حاصل از شبیه‌سازی آنها نیز در شکل ۱۳-۴ رسم شده است. برای درک و فهم بهتر مثالها اطلاعات مورد نیاز از چگونگی عملکرد شبیه‌ساز VHDL در زیر تکرار شده است :

- در زمان صفر کلیه سیگنال‌ها ***سمت چپ‌ترین ارزش***[1] خود را که در تعریف نوع آنها در نظر گرفته شده است خواهند داشت. بنابراین یک سیگنال تعریف شده از نوع bit ارزش اولیه '0' را به خود می‌گیرد زیرا نوع بیت به فرم زیر تعریف شده است :

type bit is ('0', '1');

همچنین کلیه پروسس‌ها در لحظه صفر شروع به کار می‌کنند. سپس سطر به سطر اجرا می‌شوند تا اینکه به عبارت wait برسند.

- همان طور که در مثال wait زیر دیده می‌شود مثالهای ۱ و ۲ مشابه یکدیگر هستند. در مثال ۳، wait on نباید در انتهای پروسس آورده شود. در مثال ۴ شرط wait فقط زمانی تحقق می‌یابد که سیگنال a دارای لبه بالارونده باشد و به مقدار '1' برسد. وارونه و معکوس '1' مقدار '0' خواهد بود بنابراین c4 در زمانهای ۲۰ns و ۴۷ns ارزش '0' را خواهد داشت. در مثال ۵ که پیچیده‌ترین مورد بین ۵ مثال است، شرط wait زمانی درست خواهد بود که a تغییر مقدار دهد و مقدارش برابر '1' گردد یا توقف به اندازه ۱۰ns ایجاد شود.

مثال (a, c1 ... c5 = بیت)

Example 1
process (a)
begin

1- Type 'left

```
  c1<=not a;
end process;
```

Example 2

```
process
begin
  c2<=not a;
  wait on a;
end process;
```

Example 3

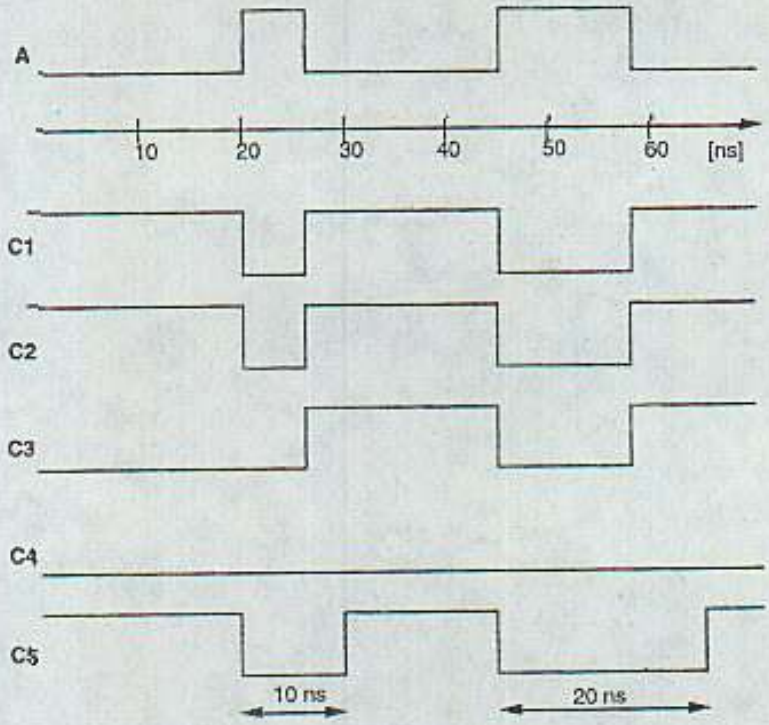
```
process
begin
  wait on a;
  c3<=not a;
end process;
```

Example 4

```
process
begin
  wait until a= '1';
  c4<=not a;
end process;
```

Example 5

```
process
begin
  c5<=not a;
  wait until a= '1' for 10 ns;
end process;
```



شکل ۱۳-۴ نتایج شبیه‌سازی

ابزارهای سنتز wait for 10ns را حمایت نمی‌کنند. توصیف این گونه مدار باعث ایجاد خطا در سنتز می‌شود. به علاوه، اکثر ابزارهای سنتز فقط استفاده از لیست حساسیت (مثال ۱) و نه مواردی مشابه مثال ۲ را برای پروسسهای ترکیبی می‌پذیرند. بعضی سیستمهای پیشرفته مثال ۲ را قبول می‌کند در حالی که مثال ۳ را نمی‌تواند سنتز کنند. مثال ۴ یک پروسس پالسی است و نتیجه آن پس از سنتز یک فلیپ فلاپ از نوع D خواهد بود. بنابراین مثالهای ۳ و ۵ فقط برای شبیه‌سازی و نه طراحی به کار می‌روند.

همان طور که قبلاً اشاره شد می‌توان در یک پروسس از چندین دستور wait استفاده کرد.

مثال :

```
process
begin
  wait until clk='1';
  ...
  wait until clk='1';
  ...
  wait until clk='1';
  ...
end process;
```

این روش توصیفی فقط توسط بعضی ابزارهای سنتز، آن هم به شرط وجود فقط یک پالس ساعت قابل قبول است (wait until clk = '1';). یک لبه پالس ساعت فعال‌کننده کلیه دستورات wait خواهد بود. نتیجه سنتز یادآوری از فضای حالت است، به این ترتیب که مجموعه دستورات بعد از اولین عبارت wait پس از اولین لبه پالس ساعت، مجموعه دستورات بعد از دومین عبارت wait پس از دومین لبه پالس ساعت و ... اجرا می‌شوند. از آنجایی که کل دستورات پروسس به صورت یک حلقه نامحدود اجرا خواهند شد، ***اشاره‌گر برنامه***[1] پس از رسیدن به خط آخر پروسس (end process;) به ابتدای آن پرش خواهد کرد.

دستور wait یک دستور متوالی است و می‌توان از آن در processها و procedureها استفاده کرد اما مجاز به استفاده از آن در functionها نیستیم.

۹-٤ عبارت Loop

دو نوع عبارت حلقه در VHDL وجود دارد :

- For loop
- While loop

۱-۹-۴ حلقه For loop

Syntax:

```
[loop_label:] for <identifier> in <discrete_range> loop
  sequence_of_statement
end loop [loop_label];
```

مثال :

```
Entity ex is
 port (a,b,c:  in   std_logic_vector (4 downto 0);
       q:      out  std_logic_vector (4 downto 0));
end;

Architecture rtl of ex is
begin
  process (a,b,c)
  begin
     for i in 0 to  4 loop     -- Not allowed to declare the loop index 'i'.
```

1- Program Pointer

```
            if a(i) = '1' then
              q(i)<=b(i);
            else
              q(i)<=c(i);
            end if;
        end loop;
    end process;
end;
```

امکان تغییر مقدار متغیر اندیس حلقه در پروسس وجود ندارد. در سیستمهای پیشرفته دستور for loop تنها با شرط ثابت بودن گستره حلقه پذیرفته است.

گستره حلقه for loop را می‌توان حذف کرد. در این صورت یک حلقه با تکرار بی‌نهایت خواهیم داشت. برای توقف تکرار در این گونه حلقه‌ها می‌توان از عبارت exit در داخل حلقه استفاده کرد. در ضمن هر دستور حلقه می‌تواند دارای عنوان باشد.

مثال :

```
Architecture rtl of ex is
begin
  process
  begin
    reset_loop: loop
      q<=(others=> '0');
      wait until clk= '1';
      exit reset_loop when resetn='0';

      main_loop:  loop
        wait until  clk= '1';
        exit reset_loop  when resetn = '0';
        q<=a+b;
        wait until  clk= '1';
        exit reset_loop  when resetn = '0';

        while en= '0'  loop
          exit reset_loop  when resetn = '0';
        end loop;

        wait until  clk= '1';
        exit reset_loop  when resetn = '0';
```

```
      q<=b+c;
    end loop;
   end loop;
  end process;
end;
```

این روش توصیفی بیشتر در سنتز رفتاری کاربرد دارد (به فصل ۱۷ مراجعه کنید، «سنتز رفتاری»).

۲-۹-۴ حلقه While loop

Syntax:

```
[loop_label:] while <condition> loop
 sequence_of_statement
end loop [loop_lable];
```

مثال :

```
process(a,b,resetn)
variable i: integer range 0 to 31;
begin
 if resent = '0' then
   i: =0;
   q<=(others=> '0');
 else
  while q<12 loop
    if i=31 then
     exit;
    else
      i:=i+1;
      q<=a(i)+b(i)+q;
    end if;
  end loop;
 end if;
end process;
```

فقط سیستمهای پیشرفته می‌توانند حلقه while loop را حمایت کنند. مثال فوق فقط می‌تواند در شبیه‌سازی به کار رود و نمی‌تواند سنتز گردد.

۱۰-٤ پروسس به تعویق انداخته شده[1]

پروسس به تعویق انداخته شده یکی از مباحثی است که در استاندارد VHDL-93 مطرح شده است. با استفاده از این پروسس امکان اجرای آن در آخرین سیکل زمانی Δt فراهم می‌شود. مطابق معمول، پروسس به تعویق انداخته شده نیز با تغییر یکی از متغیرهای لیست حساسیت و یا عبارت wait فعال می‌شود، با این تفاوت که پروسس به تعویق انداخته شده در لحظه‌ای که پروسس فعال می‌شود شروع به اجرا شدن نمی‌کند بلکه منتظر می‌شود تا کلیه سیگنال‌ها به حالت پایدار خود برسند. بدین معنی که پروسس به عنوان آخرین رخداد آن لحظه اجرا می‌شود. اگر چندین پروسس به تعویق انداخته شده مورد استفاده قرار گیرند، استاندارد VHDL-93 نمی‌تواند تعیین کند که کدام یک باید اول اجرا شوند. بنابراین استفاده بیش از یک پروسس به تعویق انداخته شده توصیه نمی‌گردد.

سنتز این نوع پروسس‌ها مطلقاً توسط هیچ ابزار سنتزی حمایت نمی‌شود. از آنجایی که در VHDL امکان نوشتن wait for 0ns; وجود ندارد، این طرح اولیه توسط VHDL-93 به عنوان مدلی فقط برای شبیه‌سازی معرفی شده است. با پروسس‌های به تعویق انداخته شده می‌توان مقداردهی سیگنال‌ها را برای طراحی و توصیف مدلهای خاص شبیه‌سازی تضمین کرد.

مثال :

```
Architecture behv of ex is
begin
  process
  begin
    . . .
  end process;

  postponed process(a,b)
  begin
    if a>b then
       q<='1' after 5 ns;
    else
       q<='0' after 5 ns;
    end if;
  end process;
end;
```

1- Postponed Process

کلیه مقداردهی‌ها در پروسس به تعویق انداخته شده باید پس از یک تأخیر انجام گیرند.

۱۱-٤ نشانهای[1] سیگنال

نشان‌های سیگنال در طراحی مدلهای VHDL بسیار مفید هستند. به طور معمول در کدهای VHDL از نشان 'event' برای بررسی لبه یک پالس استفاده می‌شود. بعضی ابزارهای سنتز مثل Mentor Graphics Autologic 1 از نشان 'last_value' به عنوان تعیین‌کننده لبه پالس استفاده می‌کنند.

یک نشان در حقیقت اطلاعاتی است که می‌توان از بلوک‌ها، آرایه‌ها، سیگنال‌ها و یا نوعهای گوناگون داده به دست آورد.

Syntax:
<name> 'attribute_identifier

از نشانهای متداول سیگنال می‌توان به موارد زیر اشاره کرد :

<signal> 'event : ارزش درست یا نادرست خواهد داشت، بسته به اینکه رخدادی در لحظه فعلی در بازه Δt روی دهد یا نه.

<signal> 'active : ارزش درست یا نادرست خواهد داشت، بسته به اینکه فعالیتی در لحظه فعلی در بازه Δt روی دهد یا نه.

<signal> 'stable (t) : ارزش درست یا نادرست خواهد داشت، بسته به این که در (t) واحد زمان رویدادی اتفاق افتاده باشد یا نه.

<signal> 'quiet (t) : ارزش درست یا نادرست خواهد داشت، بسته به این که در (t) واحد زمان فعالیتی انجام شده باشد یا نه.

<signal> 'transaction : چنان‌چه فعالیتی روی سیگنال انجام شود در همان لحظه تغییری را اعلام می‌کند.

<signal> 'delay (t) : سیگنالی با تأخیر (t) واحد زمان ایجاد می‌نماید.

<signal> 'last_event : مقدار زمان را از آخرین رویداد اعلام می‌کند.

<signal> 'last_active : مقدار زمان را از آخرین فعالیت اعلام می‌نماید.

1- Attribute

<signal> 'last_value : ارزشی برابر با ارزش قبلی اعلام می‌کند.

<signal> 'range : گستره سیگنال را اعلام می‌نماید. به طور مثال :

```
signal a:std_logic_vector(3 downto 0);
for a 'range loop    -- 'range = 3 downto 0
  . . .
end loop;
```

مثالی از نشانهای تعریف شده برای نوعهای داده :

<type name> 'left : سمت چپ‌ترین شماره آرایه را اعلام می‌کند.

<type name> 'right : سمت راست‌ترین شماره آرایه را اعلام می‌نماید.

<type name> 'high : بزرگ‌ترین شماره آرایه را اعلام می‌کند.

<type name> 'low : کوچک‌ترین شماره آرایه را اعلام می‌نماید.

<type name> 'length : طول و درازای نوعهای داده را اعلام می‌کند.

مثال :

```
subtype my_type1 is integer range 15 downto 0;
subtype my_type2 is integer range 0 to 15;
```

my_type1 'left = 15	my_type2 'left = 0
my_type1 'right = 0	my_type2 'right = 15
my_type1 'high = 15	my_type2 'high = 15
my_type1 'low = 0	my_type2 'low = 0
my_type1 'length = 16	my_type2 'length = 16

<type name> 'succ (data value) : مقدار بعدی در نوع داده را اعلام می‌کند.

<type name> ' pred (data value) : مقدار قبلی در نوع داده را اعلام می‌نماید.

<type name> 'rightof (data value) : ارزش سمت راست نوع داده را اعلام می‌کند.

<type name> 'leftof (data value) : ارزش سمت چپ نوع داده را اعلام می‌نماید.

مثال :

```
type my_type3 is (John, Tom, Petra);
type my_type4 is my_type3 range Tom downto John;
```

my_type3'succ(John) = Tom	my_type4'succ(John) = Error
my_type3'pred(John) = Error	my_type4'pred(John) = Tom

my_type3'rightof(John) = Tom
my_type3'leftof(John) = Error

my_type4'rightof(John) = Error
my_type4'leftof(John) = Tom

۱۲-٤ توصیفهای گوناگون پالس ساعت در پروسس‌های پالسی

چندین روش توصیف پالس ساعت برای یک فلیپ فلاپ در یک پروسس پالسی وجود دارد. متداول‌ترین آنها را در زیر بررسی می‌کنیم :

```
Alt 1: process(clk)
       begin
        if clk 'event and clk='1' then
            q<=d;
        end if;
       end process;
Alt 2: process (clk)          -- Alt 2 is not valid for asynchronous reset
       begin
        if clk= '1' then
            q<=d;
        end if;
       end process;
Alt 3: process (clk)
       begin
        if clk 'event and clk= '1' and clk 'last_value= '0' then
           q<=d;
        end if;
       end process;
Alt 4: process
       begin
        wait until clk='1';
            q<=d;
      end process;
Alt 5: process
       begin
       wait until prising (clk);          -- "prising" is a function
          q<=d;
       end process;
```

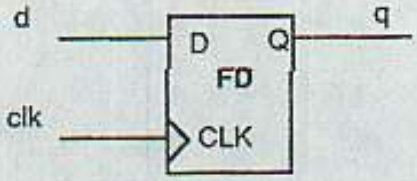
انتخاب روش توصیف به ابزار سنتزی که استفاده می‌شود بستگی دارد. جدول ۴-۴ چهار ابزار سنتز متداول و مواردی که هر یک حمایت می‌کنند را به طور خلاصه بررسی می‌کند.

	Synopsys	Autologic 1	Autologic 2	view logic
روش ۱	بله	خیر	بله	بله
روش ۲	بله	خیر	بله	خیر
روش ۳	خیر	بله	بله	خیر
روش ۴	بله	بله	بله	بله
روش ۵	خیر	بله	بله	بله

جدول ۴-۴ توصیفهای متفاوت پالس ساعت و ابزارهایی که آنها را حمایت می‌کنند

در شبیه‌سازی VHDL تنها روش ۳ به درستی لبه پالس ساعت را هنگامی که از سطح منطقی '0' به '1' می‌رود تضمین می‌کند. دیگر روشها لبه کلاک را از سطح 'X' به '1' قابل قبول می‌دانند. ارزش 'X' در سخت‌افزار وجود ندارد، بنابراین در حقیقت هیچ مشکلی در لبه رونده از 'X' به '1' به عنوان لبه بالارونده پالس ساعت وجود ندارد.

روشهای ۱ و ۲ و ۴ از متداول‌ترین روشهای توصیف پالس ساعت هستند. البته کلیه روشها دارای نتیجه یکسانی خواهند بود (شکل ۱۴-۴).



شکل ۱۴-۴ پروسس پالسی

در اوایل دهه ۱۹۹۰ و به هنگام توصیف پالس ساعت در اغلب ابزارهای سنتز مشکل به وجود می‌آمد. برای این کار لازم بود کلاک به صورت زیر تعریف شود :

```
wait until clk'event and clk = '1';  -- Not good
```

این توصیف با تعریف زیر معادل است :

```
wait until clk = '1';                -- good
```

برای برآورده شدن شرط wait until باید تغییری در سیگنال رخ دهد (clk'event) و پس از آن، مقدار جدید سیگنال برابر '1' گردد. بنابراین استفاده از مدل اول به معنای نوشتن عبارتی غیرضروری است که در دستور wait until نهفته است. متأسفانه این روش نادرست تعریف پالس ساعت در کتابها رواج پیدا کرده است.

۱۳-٤ ریست[1] سنکرون و آسنکرون

برای ریست کردن یک فلیپ فلاپ دو روش متفاوت وجود دارد: ریست سنکرون یا آسنکرون. همان طور که قبلاً اشاره شد، چندین روش توصیف پالس ساعت در پروسس‌های پالسی وجود دارد. توصیف چگونگی لبه کلاک، کنترل‌کننده چگونگی عملکرد reset در یک فلیپ فلاپ خواهد بود. دو روش متفاوت که توسط اغلب ابزارهای سنتز حمایت می‌شوند در زیر آورده شده است.

۱-۱۳-۴ ریست آسنکرون

به محض فعال شدن reset در ریست آسنکرون، فلیپ فلاپ بدون توجه به تغییر کلاک reset می‌شود. این بدان معناست که باید در پروسس پالسی که reset آسنکرون دارد و دارای لیست حساسیت است، در کنار clk در این لیست، سیگنال reset نیز باشد (alt 1). از نقطه نظر نگارشی به سادگی می‌توان به داشتن یا نداشتن reset آسنکرون یک پروسس پالسی پی برد. این کار با چک کردن لیست حساسیت انجام می‌شود. اگر reset در آن بود پروسس پالسی دارای reset آسنکرون است.

روش عمومی توصیف ریست آسنکرون که مورد حمایت اکثریت ابزارهای سنتز مثل Synopsys و Autologic 2 است در زیر آورده شده است :

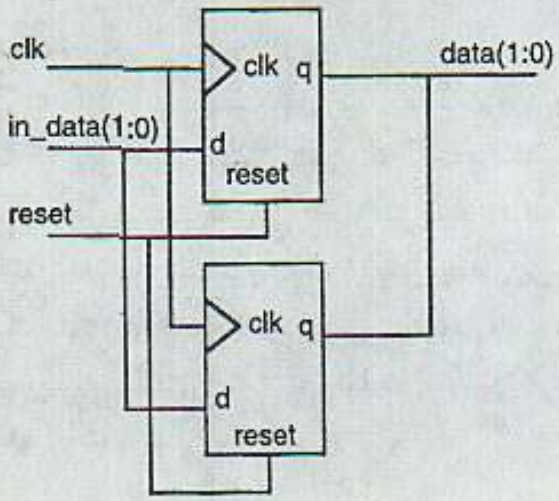
```
Alt 1:process (clk, reset)
      begin
       if reset= '1' then
          data<= "00";
       elsif clk 'event and  clk= '1' then
            data<=in_data;
       end if;
      end process;
```

1- Reset

روش دیگر توصیف ریست آسنکرون که توسط ViewLogic حمایت می‌شود به صورت زیر است:

```
Alt 2:process
      begin
       wait until (prising (clk) or resetn= '0');
       if resetn= '0' then
          data<= "00";
       else
          data<=in_data;
       end if;
      end process;
```

هر دو گونه توصیف این نوع reset دارای نتایج سنتز مشابهی هستند (شکل ۱۵-۴).



شکل ۱۵-۴ ریست آسنکرون

از نقطه نظر روشهای گوناگون طراحی، تنها یک سیگنال می‌تواند به عنوان کلاک یا reset وارد بلوک منطقی شود. ابزارهای سنتز نمی‌توانند چندین پالس ساعت را در یک پروسس حمایت کنند. به علاوه هر گونه ترکیب گیتی با پالس ساعت باید خارج از پروسس انجام شود (به فصل ۱۲ مراجعه کنید، «آشنایی با روشهای آزمایش»). همین موضوع در مورد reset نیز برقرار است. اگر چندین سیگنال برای reset کردن یک فلیپ فلاپ به کار رود برای استفاده از مدل بالا باید ترکیب گیتی آنها خارج از پروسس پالسی انجام شود.

۴-۱۳-۲ ریست سنکرون

در ریست سنکرون، یک فلیپ فلاپ تنها زمانی می‌تواند reset شود که پالس ساعت در لبه فعال باشد. به این معنا که فقط clk در لیست حساسیت پروسس پالسی که ریست سنکرون برای آن در نظر گرفته شده است قرار گیرد (alt 1).

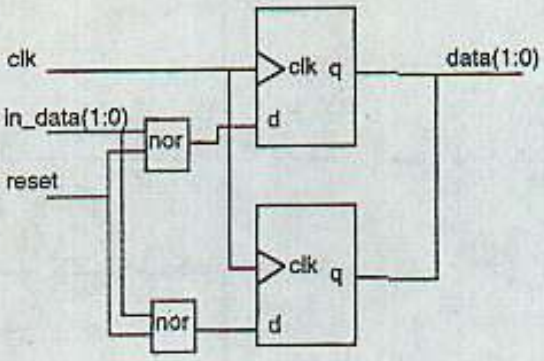
روش عمومی توصیف ریست سنکرون که مورد حمایت اکثر ابزارهای سنتز است (مثل Synopsys و Autologic 2) در زیر آورده شده است.

```
Alt 1:process(clk)
      begin
       if clk 'event and clk= '1' then
         if reset= '1' then
            data<= "00";
         else
            data<=not in_data;
         end if;
       end if;
      end process;
```

روش دیگر توصیف ریست سنکرون که توسط ViewLogic حمایت می‌شود به صورت زیر است.

```
Alt 2:process
      begin
         wait until prising (clk);
         if reset= '1' then
            data<= "00";
         else
            data<=not in_data;
         end if;
      end process;
```

نتیجه سنتز هر دو روش در شکل ۱۶-۴ نشان داده شده است.



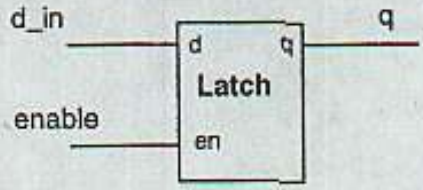
شکل ۱۶-۴ ریست سنکرون

۱۴-٤ لچ[1]ها

معمولاً برای توصیف لچها در VHDL عبارت پروسس به کار می‌رود. در یک لچ نرمال (لچ شفاف[2]) به محض فعال شدن سیگنال enable مقدار d_input بر روی q_output قرار می‌گیرد. با غیرفعال شدن سیگنال enable مقدار q_out حفظ خواهد شد. لچها در VHDL توسط پروسس‌های ترکیبی توصیف می‌شوند. لازم به یادآوری است، سیگنال‌هایی که در پروسس 'خوانده' می‌شوند باید در لیست حساسیت پروسس قرار گیرند. خروجی لچ فقط باید در زمانی که enable فعال است مقداردهی شود.

مثال :

```
process (enable, d_in)
begin
  if enable= '1' then
    q<=d_in;
  end if;
end process;
```



شکل ۱۷-۴ لچ

1- Latch

2- Transparent Latch

۱۵-٤ تمرین

۱- (i) دو نوع اصلی پروسس را توضیح دهید.

(ii) هر کدام بیانگر چه نوع منطقی هستند ؟

۲- (i) مشخصه یک پروسس ترکیبی ناقص چیست ؟

(ii) چگونه می‌توان از پروسس‌های ترکیبی ناقص اجتناب کرد ؟

۳- (i) چرا روشهای گوناگونی برای توصیف لبه پالس ساعت وجود دارد ؟

(ii) حداقل به سه مورد اشاره کنید.

۴- اگر به یک سیگنال خروجی در یک پروسس‌پالسی مقداری داده نشود چه اتفاقی رخ می‌دهد؟

۵- کدهای VHDL زیر به چند فلیپ فلاپ نیاز خواهند داشت ؟

(a)

```
Architecture rtl of ex is
signal a,b: std_logic_vector (3 downto 0);
begin
 process (clk)
 begin
      if clk= '1' and clk'event then
        if q(3)/= '1' then
            q<=a+b;
        end if;
      end if;
 end process;
end;
```

(b)

```
Architecture rtl of ex is
signal a,b: std_logic_vector (3 downto 0);
begin
 process (clk)
 variable int: std_logic_vector (3 downto 0);
 begin
      if clk= '1' and clk'event then
        if int(3)/= '1' then
            int:=a+b;
```

```
            q<=int;
        end if;
      end if;
  end process;
  end;
```

۶- (i) دو روش برای صفر کردن یک فلیپ فلاپ بنویسید.

(ii) اگر از لیست حساسیت برای یک پروسس پالسی استفاده شود، سیگنال reset داخل لیست حساسیت باید از چه نوعی باشد ؟

۷- در کد VHDL زیر چه تعداد فلیپ فلاپ برای reset سنکرون و چه تعداد برای reset آسنکرون اختصاص می‌یابد ؟

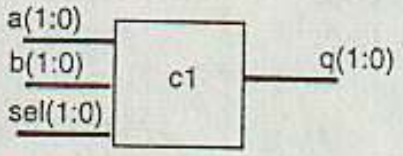
```
process (clk , resetn)
begin
    if resetn= '0' then
      q1<= '0';
      q2<= '0';
    elsif clk'event and clk= '1' then
      q1<= a and b;
      q2<= c;
    end if;
end process;
process (clk)
begin
    if clk= '1' then
      if resetn= '0' then
        q3<= '0';
      else
        q3<= q2;
      end if;
    end if;
end process;
```

۸- کد VHDL زیر چه تعداد لچ تولید می‌کند ؟

```
Architecture rtl of ex is
signal a,b,c,d,e: std_logic_vector (1 downto 0);
begin
  process (clk)
```

```
variable int: std_logic_vector (3 downto 0);
begin
    if en= '1' then
            a<=c;
            b<=d;
    else
            a<=e;
    end if;
end process;
end;
```

۹- جزء ترکیبی با ورودیها و خروجیهای زیر را طراحی کنید.



جزء ترکیبی باید عملکرد زیر را داشته باشد.

sel	q
00	a XOR b
01	a OR b
10	a NOR b
11	a AND b
others	"XX"

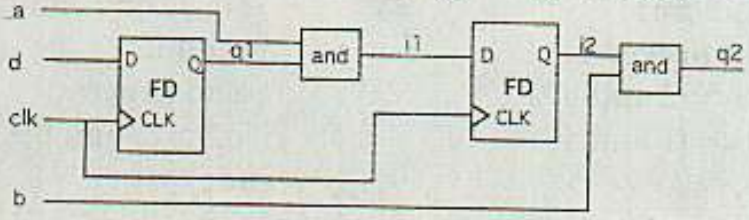
(i) با به کارگیری یک دستور if

(ii) با به کارگیری یک دستور case

(iii) با به کارگیری یک دستور when else

(iv) اگر هر سه مورد فوق سنتز شوند، کدام یک کمترین منطق را تولید خواهد کرد ؟

۱۰- فرض کنید منطق زیر باید پیاده‌سازی شود :



(i) کدام یک از گزینه‌های زیر کد VHDL صحیح قابل سنتز برای شماتیک فوق خواهد بود ؟

(1)
```
Architecture rtl of ex is
signal i1:std_logic;
begin
 process
 begin
   wait until clk= '1';
     i1<=d and a;
     q2<=i1 and b;
 end process;
end;
```

(2)
```
Architecture rtl of ex is
signal i1,i2:std_logic;
begin
 process
 begin
       wait until clk= '1';
       i1<=d and a;
       i2<=i1;
 end process;
 q2<=i2 and b;
end;
```

(3)
```
Architecture rtl of ex is
signal q1,i1,i2:std_logic;
begin
 process
 begin
   wait until clk= '1';
     q1<=d;
     i1<=a and q1;
     i2<=i1;
     q2<=i2 and b;
 end process;
end;
```

(4)
```
Architecture rtl of ex is
signal q1,i2:std_logic;
begin
 process
 begin
       wait until clk= '1';
       q1<=d;
       i2<=q1 and a;
 end process;
 q2<=i2 and b;
end;
```

(5)
```
Architecture rtl of ex is
signal q1,i1,i2:std_logic;
begin
  process
  begin
    wait until clk='1';
    q1<=d;
    i2<=i1;
  end process;
  q2<=i2 and b;
  i1<=q1 and a;
end;
```

(6)
```
Architecture rtl of ex is
signal q1,i2:std_logic;
begin
  process(clk)
  begin
  if clk'event and clk='l' then
     q1<=d;
     i2<=q1 and a;
  end if;
  end process;
  q2<=i2 and b;
end;
```

(ii) نتیجه هر یک از کدهای فوق را بصورت بلوک دیاگرامی بکشید.

فصل پنجم

کتابخانه، بسته و زیربرنامه‌ها

به هنگام طراحی سیستمهای مختلف، امکان استفاده از component ها، function ها و procedure های از قبل تعریف شده و آماده که در کتابخانه ذخیره هستند وجود دارد. طبیعی است که برای طراحی‌های سخت‌افزاری المان‌های تست شده به کار می‌روند (مثل سری 74LSXX). در دنیای طراحی نرم‌افزاری با استفاده از VHDL چندین ساختار به منظور طراحی المان‌های مختلف مورد نیاز فراهم است.

این فصل بیشتر به چگونگی ذخیره function ها، procedure ها و اجزای ترکیبی و چگونگی استفاده دوباره آنها در طراحی‌های جدید می‌پردازد.

۵-۱ کتابخانه[1]ها

وقتی یک المان VHDL کمپایل می‌شود به طور خودکار در ***کتابخانه کار***[2] ذخیره می‌گردد. کتابخانه نام یک ***شاخه***[3] از کامپیوتری که نرم‌افزار در آن قرار دارد نیست، بلکه نام اشاره‌گری است که به هنگام راه‌اندازی نرم‌افزار و شروع به کار آن به طور اتوماتیک تعیین می‌شود. این بدان معناست که

1- Library

2- Work Library

3- Directory

کتابخانه‌های کار بسته به اینکه کمپایلر VHDL در چه محلی شروع به کار کند تشکیل می‌شوند.

کلیه المان‌های کمپایل شده و اغلب بسته‌ها در کتابخانه ذخیره می‌شوند. یک بسته می‌تواند مشتمل بر function ها، procedure ها، مقادیر ثابت و *نوع*[1]های مختلف داده، باشد. به منظور استفاده از بسته‌ها و اجزای ترکیبی، باید کتابخانه مورد نظر را با دستورالعمل زیر در ابتدای کد VHDL معرفی و مشخص کرد.

```
Library <library_name>;
Use <library_name>.<package_name>.ALL;
```

استاندارد VHDL به گونه‌ای تعیین شده است که ***کتابخانه std***[2] و کتابخانه کار همواره در دسترس هستند. بنابراین ضرورتی برای مشخص کردن این دو کتابخانه در کد VHDL وجود ندارد. کلیه توابع و نوعهای داده که به عنوان پیش فرض برای کمپایلر VHDL تعریف شده‌اند در بسته‌ای به نام استاندارد در دسترس هستند. این بسته در کتابخانه std قرار دارد. به طور مثال داده‌هایی از نوع bit، bit_vector، character، time و integer در این بسته تعریف شده‌اند (می‌توانید به «بسته‌های VHDL» در پیوست (ب) مراجعه کنید). با توجه به استاندارد VHDL، بسته استاندارد همواره برای کمپایلر قابل رؤیت است و ضرورتی برای مشخص کردن این بسته نیز در ابتدای کد VHDL وجود ندارد. بنابراین همواره سر خط نامرئی زیر در هر کد VHDL وجود خواهند داشت:

```
Library work;
Library std;
Use std.standard.ALL;
```

چنان‌چه لازم باشد کتابخانه‌ها و بسته‌های دیگری به کار گرفته شوند، باید در بالای کد VHDL قبل از entity معین شوند. داده‌هایی از نوع std_logic، std_logic_vector، std_ulogic و std_ulogic_vector در بسته‌ای به نام std_logic_1164 تعریف شده‌اند. این بسته توسط IEEE تعیین شده است و نام کتابخانه‌ای که بسته std_logic_1164 در آن قرار گرفته نیز ieee می‌باشد. بنابراین اگر داده‌هایی از این نوع در برنامه به کار روند (کد توصیه می‌شود) حتماً باید خطوط زیر را قبل از entity وارد کنیم:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
entity . . .
```

1- Type

2- Standard Library

۵-۲ بسته[1]ها

در مورد طراحی‌های بزرگ امکان استفاده از توابع، مقادیر ثابت، انواع دیتا و غیره در بلوک‌های مختلف وجود دارد. مقدار ثابت می‌تواند به طور مثال اندازه (پهنای) باس سیستم باشد. تابع می‌تواند محاسبه‌کننده مجموع پیغام اطلاعاتی در یک سیستم باشد. اگر مدیر پروژه پهنای باس و یا محاسبات سیستم را تغییر دهد، ترجیح داده می‌شود که این تغییرات فقط در یک محل اعمال شوند. این کار با استفاده از بسته‌ها امکان‌پذیر خواهد شد. اگر طراحان بخشهای مختلف پروژه بخواهند از توابع و یا مقادیر ثابت در پروژه استفاده کنند می‌توانند با کمک اشاره‌گر Use بسته مورد نظر را در اختیار گیرند. با اعمال تغییرات لازم در بسته، کلیه زیربخشهای پروژه که از این بسته استفاده کرده‌اند تغییرات را دریافت خواهند کرد.

مثال :

```
Library project;
Use project.proj _ pack.ALL;
```

کلیه زیربرنامه‌ها، نوعهای داده و ثابتهایی که در یک بسته معین شده‌اند را می‌توان در طراحی‌های بعدی با استفاده از دستور Use در ابتدای کد VHDL به کار برد.

یک بسته می‌تواند شامل function ها، producer ها، اجزای ترکیبی، نوعهای داده، ثابتها و *نشان*[2]ها باشد. یک بسته از دو بخش زیر تشکیل شده است :

- *اعلام*[3] بسته
- *بدنه*[4] بسته

بخش اعلام بسته شامل معرفی fuction ها، procedure ها، ثابتها، نوعها و component ها (اجزای ترکیبی) است که در آن بسته مورد استفاده قرار گرفته‌اند. این بخش از یک بسته می‌تواند با بخش entity یک جزء ترکیبی مقایسه شود. مثلاً آنچه را که باید به عنوان خروجی در نظر گرفته شود تعیین می‌کند، درست همانند کاری که entity انجام می‌دهد. تفاوت در این است که entity مشخص می‌کند کدام سیگنال باید خروجی جزء ترکیبی باشد در حالی که بخش اعلام بسته مشخص‌کننده زیربرنامه‌ها، ثابتها و نوعهای داده‌ایی است که به عنوان خروجی بسته خواهند بود.

1- Package

2- Attribute

3- Declaration

4- Body

بدنه شامل برنامه‌ای است متناسب با function ها و procedure هایی که در بخش اعلام معرفی شده‌اند. اگر برحسب نیاز زیربرنامه‌هایی در داخل بدنه یک بسته به وجود آیند، این برنامه‌ها در فضای خارج از بسته قابل استفاده نیستند. بدنه را می‌توان همتای architecture (معماری) کد VHDL یک جزء ترکیبی دانست. معماری توصیف‌کننده رفتاری یک جزء ترکیبی است اما بدنه بسته بیان‌کننده رفتار زیربرنامه‌هایی است که در بخش اعلام مشخص شده‌اند.

```
Syntax for a package:
package <package_name> is
        [exported_subprogram_declarations]
        [exported_constant_declarations]
        [exported_components]
        [exported_type_declarations]
        [attribut_declarations]
        [attribut_specifications]
end [<package_name>];

package body <package_name> is
        [exported_subprogram_bodies]
        [internal_subprogram_declarations]
        [internal_subprogram_bodies]
        [internal_constant_declarations]
        [internal_type_declarations]
end [<package_name>];
```

مثال :

```
Example:
package mypack is
 function minimum (a,b:in std_logic_vector)
       return std_logic_vector;
-- declarations of exported constant and types:
 constant maxint: integer := 16#FFFF#;
 type arithmetic_mode_type is (signed, unsigned);
end mypack;

package body mypack is
 function minimum (a,b:in std_logic_vector)
       return std_logic_vector is
 begin
```

```
    if a<b then
        return a;
    else
        return b;
    end if;
  end minimum;
end mypack;
```

Syntax for a package (VHDL-93):

```
    package <package_name> is
        . . .
    end [package] [<package_name>];
    package body <package_name> is
        . . .
    end [package body] [<package_name>];
```

کلمه کلیدی Use را می‌توان برای استفاده از بسته‌هایی که کاربر نوشته و یا به صورت آماده وجود دارد، به کار برد. این کلمه را باید قبل از بخش اعلام و بعد از کتابخانه‌ها نوشت :

مثال :

```
Library ieee;
Library mylib;

Use ieee.std_logic_1164.ALL;
Use mylib.mypack.ALL;

Entity <name> is
port (a: in   std_logic;
      q: out std_logic;
      . . .
```

در مثال بالا فرض شده که بسته مورد نظر در کتابخانه mylib کامپایل و ذخیره شده است. سیستم باید یک اشاره‌گر داشته باشد تا بتواند محل قرارگیری کتابخانه mylib را مشخص کند. در مثال فوق همچنین به mypack.ALL اشاره شده است. ALL به مفهوم در دسترس بودن کلیه زیربرنامه‌هایی است که در بسته وجود دارد. چنان‌چه فقط یک یا چند زیربرنامه از بسته، مثلاً فقط تابع minimum مورد نیاز باشد می‌توان به صورت زیر آن را فراخوانی کرد :

```
Use mylib.mypack.minimum;
```

اگر چندین تابع مختلف از بسته‌های متفاوت مورد استفاده باشد باید به صورت زیر عمل نمود :

```
Library mylib, ieee;
Use mylib.mypack.minimum, ieee.std_logic_1164.ALL;
```

و یا :

```
Library mylib;
Library ieee;
Use mylib.mypack.minimum;
Use ieee.std_logic_1164.ALL;
```

۳-۵ زیربرنامه[1]ها

در استاندارد VHDL دو قسم زیربرنامه تعریف شده است :

- Function
- Procedure

function ها تنها یک مقدار تولید می‌کنند نظر به اینکه یک procedure می‌تواند هیچ مقداری تولید نکند و یا چندین مقدار را برگرداند. در VHDL، زیربرنامه‌ها می‌توانند در سه قسمت زیر تعریف شوند :

- بسته
- معماری
- پروسس

از آنجایی که تنها زیربرنامه‌های یک بسته را می‌توان چندین بار فراخوانی و استفاده کرد، بنابراین توصیه می‌شود کلیه زیربرنامه‌ها در بسته نوشته شوند.

کد VHDL داخل هر زیربرنامه متوالی است. بنابراین فقط باید از دستورات متوالی VHDL مانند if-then-else و case در داخل زیربرنامه‌ها استفاده کرد. مجاز به استفاده از دستورات موازی مثل پروسس و when else در زیربرنامه‌ها نیستیم. بخش اعلام بسته بین زیربرنامه و شروع (begin) به صورت یک اعلام متوالی خواهد بود. بدین معنا که فقط متغیرها و نه سیگنال‌ها می‌توانند در داخل

1- Subprogram

زیربرنامه‌ها معرفی و به کار برده شوند. با این وجود لازم به یادآوری است که مجاز به استفاده از دستور متوالی wait در داخل یک function نیستیم.

۵-۳-۱ procedure ها

یک پروسیجر می‌تواند آرگمان خود را تغییر داده و متغیر، سیگنال و یا مقدار ثابت را به عنوان انواع پارامترهای ورودی - خروجی بپذیرد. سه مد گوناگون برای پروسیجرها وجود دارد : in، out و inout. اگر مد مشخص نشده باشد، حالت پیش فرض in در نظر گرفته می‌شود. اگر مد out یا inout تعیین شود، نوع پارامتر ورودی - خروجی باید متغیر یا سیگنال باشد. چنان‌چه نوع پارامتر ورودی - خروجی تعیین نشده باشد با توجه به مد (in، out و inout) متغیر یا مقدار ثابت فرض خواهد شد. به بیان دیگر، پیش فرض نوع پارامتر ورودی و خروجی در مد in مقدار ثابت و در مد out یا inout متغیر است. در فراخوانی موازی پروسیجر، مجاز به به‌کارگیری پارامترهای متغیر نیستیم.

In the declaration part of the package:

procedure <procedure_name>
[([object_class] arg_name {, arg_name}:
[mode] **type** [(index_range [, index_range])];
{[object_class] arg_name {, arg_naje}:
[mode] **type** [(index_range [, index_range])]})];

In the package body:

procedure <procedure_name>
[(object_class] arg_name {, arg_name}:
[mode] type [(index_range [, index_range])];
{[object_class] arg_name {, arg_name}:
[mode] type [(index_range [, index_range])]})] **is**
[constant_declarations]
[variable_declarations]
[type_declarations]
begin
sequence_of_statements
end [<procedure_name>];

Syntax for procedures (VHDL-93):

```
procedure <procedure_name> is
    . . .
end [procedure] [<procedure_name>];
```

مثال از یک پروسیجر :

```
procedure ff (signal clk, data : in  std_logic;
              signal q:          out std_logic) is
begin
  loop
    wait until clk= '1';
    q <= data;
  end loop;
end ff;
```

مثال ۲ (نامناسب) :

```
Architecture rtl of ex2 is
procedure calc (a,b:          in  integer;
                avg, max: out integer) is      -- Default variables
begin
  avg:=(a+b)/2;
  if a>b then
    max:=a;
  else
      max:=b;
  end if
end calc;

begin
  calc(d1, d2 , q1, q2);          --Error, q1,q2 are not variables
                                  --Concurrent procedure call
  process (d3,d4)
  variable a,b: integer;
  begin
    calc(d3, d4 , a, b);          -- Good, a and b are variables
                                  -- Sequential procedure call
    q3<=a;
    q4<=b;
```

```
    . . .
  end process;
  . . .
end;
```

مثال ۳ (مناسب) :

```
Architecture rtl of ex3 is
procedure calc (a,b:              in integer;
                signal avg, max: out integer) is
begin
      avg<=(a+b)/2;
      if a>b then
        max<=a;
      else
        max<=b;
      end if;
end calc;

begin
  calc(d1,d2,q1,q2);              -- Concurrent procedure call
  process (d3,d4)
  begin
    calc(d3,d4,q3,q4);            -- Sequential procedure call
    . . .
  end process;
  . . .
end;
```

۲-۳-۵ function ها

یک تابع نمی‌تواند آرگمان خود را تغییر داده و تنها می‌تواند پارامترهایی از نوع مقدار ثابت یا سیگنال را فقط در مد ورودی (in) بپذیرد. چنان‌چه نوع پارامتر و مد مشخص نشده باشد، فرض بر مد ورودی و نوع مقدار ثابت خواهد بود.

```
In the declaration part of the package:
      function <function_name>
          [([object_class] arg_name {, arg_name}:
          [IN] type [(index_range [, index_rame])]
```

```
        {[object_class] arg_name {, arg_name}:
        [IN] type [(index_range [, index_range])]})]
    return type;
```

In the package body:

```
    function <function_name>
        [([object_class] arg_name {, arg_name}:
        [IN] type [(index_range [, index_rame])]
        {[object_class] arg_name {, arg_name}:
        [IN] type [(index_range [, index_range])]})]
    return type is
        [constant_declarations]
        [varialbe_declarations]
        [type_declarations]
    begin
        sequence_of_statements
        return (expression);
    end [<function_name>];
```

Syntax for functions (VHDL-93):

```
    function <function_name> is
        . . .
    end [function] [<function_name>];
```

همان طور که قبلاً نیز اشاره شده، زیربرنامه‌ها می‌توانند در بسته، معماری و پروسس تعریف شوند. در زیر مثالهایی از چگونگی تعریف و به کارگیری آنها در سه فضای ذکر شده بررسی خواهد شد.

مثال ۱ : بسته

```
package mypack is
  function max (a,b: in std_logic_vector) return
        std_logic_vector;
end;
package body mypack is
  function max (a, b: in std_logic_vector) return
        std_logic_vector is
  begin
        if a>b then
           return a;
```

```
        else
            return b;
        end if;
    end;
end;
```

تابع max در بسته mypack می‌تواند در یک معماری برنامه، به شرط نوشتن خط Use work.mypack.ALL در ابتدای آن به کار گرفته شود.

```
Use work.mypack.ALL;
. . .
Entity ex is
    port(. . .
end;

Architecture rtl of ex is
begin
    . . .
    q<=max (d1,d2);                    -- Concurrent function call
    process (data,g)
    begin
        data_out<=max(data,g);         -- Sequential function call
    end process;
    . . .
end;
```

مثال ۲ : معماری

```
Architecture rtl of ex2 is
    function max (a,b: in std_logic_vector) return
        std_logic_vector is
    begin
        if a>b then
            return a;
        else
            return b;
        end if;
    end max;
    . . .
begin
    q<=max(d1,d2);                     -- Concurrent function call
```

```
  process(data,g)
  begin
      data_out<=max(data,g);          -- Sequential function call
  end process;
  . . .
end;
```

مثال ۳ : پروسس

```
Architecture rtl of ex3 is
begin
  process(data,g)
    function max (a,b: in std_logic_vector) return
       std_logic_vector is
    begin
       if a>b then
            return a;
       else
            return b;
       end if;
    end max;
  begin                          -- Start of process
      data_out<=max(data,g);
  end process;
  . . .
end;
```

معمولاً کلیه function ها و producer ها به گونه‌ای نوشته می‌شوند که در آنها طول بردار ورودی یا خروجی معین نشده باشد. بنابراین می‌توان یک زیربرنامه را با هر برداری با طول دلخواه به کار برد.

مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use work.mypack.ALL;

Entity ex is
Port (a,b: in   std_logic_vector (3 downto  0);
      c,d: in   std_logic_vector (5  downto 0);
      q1:  out std_logic_vector (3  downto 0);
      q2:  out std_logic_vector (5  downto 0));
```

```
end;

Architecture rtl of ex is
begin
        q1<=max(a,b);  -- Vector length = 4 bits
        q2<=max(c,d);  -- Vector length = 6 bits
end;
```

همان طور که مثالهای بالا نشان می‌دهند یک فانکشن می‌تواند چندین عبارت return داشته باشد. تنها نکته‌ای که باید مورد توجه باشد اجرای فقط یکی از این عبارات است. تقریبا اکثر ابزارهای سنتز هر دو نوع زیربرنامه (function و procedure) را حمایت می‌کنند و تنها تعدادی از ابزارهای ساده سنتز محدودیتی در ارتباط با اینکه فقط یک عبارت return باید در انتهای فانکشن باشد، دارند. به منظور رفع این مشکل می‌توان یک متغیر میانی در داخل فانکشن تعریف کرد و خروجی هر حالت را در آن متغیر میانی به طور موقت ذخیره کرده و نهایتاً آن را به عنوان تنها نتیجه تابع، در مقابل عبارت return قرار داد.

مثال :

```
function max (a,b: in integer) return integer is
variable int: integer;
begin
  if a>b then
      int:=a;          -- Store return value
  else
      int:=b;          -- Store return value
  end if;
  return int;          -- Return value
end max;
```

البته اغلب ابزارهای سنتز چندین عبارت return را حمایت می‌کنند.

در فراخوانی یک فانکشن می‌توان سیر کامل دستیابی به آن را در کد VHDL مشخص کرد. برای این کار باید مطابق زیر عمل شود :

<library> . <pakage> . <function>

مثال :

```
q <= work.mypack.max (a, b);
```

چنان‌چه آدرس کامل مشخص شده باشد نیازی به معرفی بسته (Use library.package.ALL) در ابتدای برنامه نیست.

۳-۳-۵ تابعهای تعیین‌کننده[1]

داده‌های از نوع std_logic و std_logic_vector به ***داده‌های از نوع تعیین شده***[2] معروف هستند. به این مفهوم که نوع std_logic به فرم زیر در بسته std_logic_1164 تعریف شده است.

Subtype std_logic **is** resolved std_ulogic;

هنگامی که چندین مقدار به یک سیگنال از نوع std_logic نسبت داده شود از تابع تعیین کننده استفاده می‌شود. این تابع یکی از آن مقادیر را برمی‌گرداند. این تابع در بسته std_logic_1164 به فرم زیر تعیین شده است :

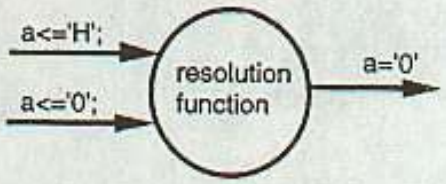
```
function resolved (s : std_ulogic_vector) return std_ulogic is
variable result : std_ulogic := 'Z';          -- weakest state default
begin
        if (length = 1) then
          return s(s'low);
        else
          for i in s'range loop
                result := resolution_table (result, s(i));
          end loop;
        end if;
        return result;
end resolved;
constant resolution_table : stdlogic_table := (

    ---------------------------------------------
    | U   X   0   1   Z   W   L   H   -       |   |
    ---------------------------------------------
    ('U','U','U','U','U','U','U','U','U' ), -- | U |
    ('U','X','X','X','X','X','X','X','X' ), -- | X |
    ('U','X','0','X','0','0','0','0','X' ), -- | 0 |
    ('U','X','X','1','1','1','1','1','X' ), -- | 1 |
    ('U','X','0','1','Z','W','L','H','X' ), -- | Z |
    ('U','X','0','1','W','W','W','W','X' ), -- | W |
    ('U','X','0','1','L','W','L','W','X' ), -- | L |
    ('U','X','0','1','H','W','W','H','X' ), -- | H |
    ('U','X','X','X','X','X','X','X','X' )  -- | - |);
```

1- Resolution Function

2- Resolved Data Types

مثالی از تابع تعیین‌کننده در شکل ۱-۵ آورده شده است :

```
Architecture rtl of ex is
signal a,b: std_logic;
begin
 a<='H';
 a<='0';
end;
```



شکل ۱-۵ استفاده از تابع تعیین‌کننده

٤-۵ افزایش ظرفیت یک تابع

چنان‌چه نوع پارامتر ورودی در یک تابع bit_vector باشد، نمی‌توان آن را در برنامه‌ای با ورودی از نوع std_logic_vector فراخوانی کرد. به منظور حل این مشکل و امکان فراخوانی یک تابع با هر پارامتر ورودی، باید تابع افزایش ظرفیت پیدا کند. افزایش ظرفیت یک تابع به مفهوم تعریف کردن تابع با انواع پارامترهای ورودی (به عنوان آرگمان آن تابع) است. در ضمن تابعها به گونه‌ای نوشته می‌شوند که بتوان آنها را با پارامترهای برداری با طولهای مختلف به کار برد.

مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Package ex_pack is
  function max(a,b: in  std_logic_vector) return
        std_logic_vector;
  function max(a,b: in vlbit_vector) return vlbit_vector;
  function max(a,b: in integer) return integer;
end;

Package body ex_pack is
  function max(a,b: in std_logic_vector) return
        std_logic_vector is
  begin
        if a>b then
              return a;
        else
              return b;
        end if;
```

```
  end;
  function max(a,b: in vlbit_vector) return vlbit_vector is
  begin
        if a>b then
              return a;
        else
              return b;
        end if;
  end;
  function max(a,b: in integer) return integer is
  begin
        if a>b then
              return a;
        else
              return b;
        end if;
  end;
end;
```

یک مثال درست از فراخوانی تابع max (با افزایش ظرفیت) در زیر آورده شده است :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use work.ex_pack.ALL;

Entity ex is
port (a1,b1:      in    std_logic_vector(3 downto 0);
      a2,b2:      in    vlbit_vector(4 downto 0);
      a3,b3:      in    integer;
      c1:   out   std_logic_vector(3 downto 0);
      c2:   out   vlbit_vector(4 downto 0);
      c3:   out   integer);
end;

Architecture ex_beh of ex is
begin
      c1<=max(a1,b1);        -- Function max (a,b:std_logic_vector)
      c2<=max(a2,b2);        -- Function max (a,b:vlbit_vector)
      c3<=max(a3,b3);        -- Function max (a,b:integer)
end;
```

در کتابخانه ieee چندین تابع بسیار مفید که در مورد آنها مکانیسم افزایش ظرفیت اعمال شده است وجود دارد. به طور مثال توابع '+'، '-'، '*'، '='، '<'، '>'، '/='، '<='، '>=' در بسته ieee_std_logic_unsigned برای داده‌های از نوع std_logic_vector یا integer افزایش ظرفیت پیدا کرده و قابل استفاده هستند. بعضی از ابزارهای پیشرفته سنتز این گونه توابع را حمایت می‌کنند. بنابراین کدهای VHDL می‌توانند بسیار خواناتر نوشته شوند. این توابع نه تنها دو ورودی را از نوع std_logic_vector (به عنوان آرگمان) می‌پذیرند بلکه می‌توانند یکی را از نوع std_logic_vector و دیگری را از نوع integer بپذیرند.

مثال :

```
function "=" (L:std_logic_vector; R:integer) return boolean;
function ">" (L:std_logic_vector; R:integer) return boolean;
```

مثال :

```
Architecture rtl of ex is
signal a,b,c: std_logic_vector(15 downto 0);
begin
   process(a,b,c,d1,d2,d3)
   begin
        if a=12 or c=11 then        -- Overloaded function "=", left side
                                    -- std_logic_vector, right side integer
              q<=d1;
        elsif  b>5 then
              q<=d2;
        else
              q<=d3;
        end if;
   end process;
end;
```

کتابخانه ieee حاوی دو بسته std_logic_signed و std_logic_unsigned است که std_logic_vector را به عنوان یک نوع داده می‌پذیرند، به بیان دیگر افزایش ظرفیت پیدا کرده‌اند. با توجه به اینکه کدام یک از این دو بسته در کد VHDL مشخص شده باشد، پارامترهای std_logic_vector از نوع signed یا unsigned در نظر گرفته می‌شوند.

مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
```

```
Use ieee.std_logic_unsigned.ALL;
...
Architecture rtl of ex is
begin
      q<=a+b;    -- unsigned add
end;

Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_signed.ALL;
...
Architecture rtl of ex is
begin
      q<=a+b;    -- signed add
end;
```

اگر در یک کد VHDL هر دو پارامتر signed و unsigned مورد استفاده باشند باید به روش زیر عمل کرد:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_arith.ALL;

Architecture rtl of ex is
begin
      q1<=unsigned(a)  + unsigned(b);      -- unsigned add
      q2<=signed(a)  + signed(b);          -- signed add
end;
```

بیشتر ابزارهای سنتز (بجز Synopsys و Autologic2) بسته‌های std_logic_signed و std_logic_unsigned را حمایت نمی‌کنند. گرچه بسته std_logic_arith برای اغلب آنها پذیرفته شده است.

در بعضی از مثالهای این کتاب از بسته std_logic_unsigned استفاده شده است. چنان‌چه خواننده این کتاب از ابزار سنتزی استفاده می‌کند که این بسته را حمایت نمی‌کند باید خط Use تعریف‌کننده بسته را با آنچه مورد قبول نرم‌افزار است جایگزین نماید.

اگر از بسته std_logic_arith استفاده می‌شود باید طول برداری که توسط تابع برگشت داده می‌شود معین شده باشد.

مثال :

```
signal a, b : std_logic_vector (3 downto 0);
q1 <= unsigned (a) + unsigned (b);
q2 <= unsigned (a) + signed (b);
q3 <= signed (a) + signed (b);
```

جدول زیر مربوط به تابع '+' است :

	unsigned	signed
unsigned	۴ بیت	۵ بیت
signed	۵ بیت	۴ بیت

چنان‌چه برای جمع دو بردار از بسته ieee.std_logic_unsigned استفاده شود و در نتیجه حاصل از این جمع، بیت نقلی مورد توجه باشد، باید یکی از آرگمان‌های تابع '+' یک بیت بیشتر داشته باشد ('0'). طول بردار نتیجه تابع همواره برابر با طویل‌ترین آرگمان تابع است.

مثال :

```
Use ieee.std_logic_unsigned.ALL;
. . .
signal a,b,q2:    std_logic_vector(7 downto 0);
signal q1:        std_logic_vector(8 downto 0);
. . .
        q1<=('0' & a) + b;          -- carry (9 bits)
        q2<=a+b;                    -- no carry bit (8 bits)
```

۵-۵ تبدیل نوع[1]

در زبان VHDL «نوع» بسیار مهم است، به این مفهوم که مطلقاً نمی‌توان به یک سیگنال، مقدار و ارزش سیگنال دیگر، با نوع متفاوت را داد. به کارگیری سیگنال‌هایی از نوع یکسان در برنامه می‌تواند این مشکل را تا حدی برطرف کند. برای تعداد بسیاری از اپراتورهای متداول مکانیسم افزایش ظرفیت اعمال شده است، به طور مثال تابع '=' می‌تواند دو متغیر std_logic_vector و integer را بدون نیاز به تبدیل نوع با هم مقایسه کند. از آنجایی که بعضی از ابزارهای سنتز این روش را حمایت

1- Type Conversion

نمی‌کنند بنابراین نیاز به تبدیل نوع بیشتر احساس می‌شود. به همین منظور تعدادی تابع آماده وجود دارد. تفاوت ابزارهای گوناگون سنتز در نام بسته‌ایی است که توابع تبدیل نوع را در خود دارند. در کتابخانه ieee تعداد زیادی بسته وجود دارد، که در میان آنها بسته std_logic_1164 شامل توابع تبدیل نوع است (می‌توانید به «بسته‌های VHDL» در پیوست (ب) مراجعه کنید).

```
std_logic          <----> bit
std_logic_vector <----> bit_vector
std_ulogic         <----> bit
std_ulogic_vector<----> bit_vector
```

مثال : در بسته ieee.std_logic_1164 :

```
function to_stdlogicvector (s:bit_vector) return
      std_logic_vector;
```

مثال زیر چگونگی استفاده از یک تابع تبدیل نوع را مشخص می‌کند:

```
Library ieee;
Use icee.std_logic_1164. ALL;

Entity ex is
port ( a,b: in     bit_vector (3  downto  0);
       q:   out    std_logic_vector (3 downto 0));
End;

Architecture rtl of ex is
begin
       q<=to_stdlogicvector(a and b);
end;
```

یکی از تبدیلهای متداول، تبدیل از نوع std_logic_vector به نوع integer و تبدیل معکوس آن است. این تابع تبدیل در بسته ieee.std_logic_unsigned قرار دارد. به طور مثال :

```
function conv_integer (arg: std_logic_vector) return integer;
function conv_std_logic_vector (arg: integer, size of integer)
  return std_logic_vector;
```

مثال :

```
Entity ex is
port (a,b,c: in integer range 0 to 15;
      q:     out std_logic_vector(3 downto 0));
end;

Architecture rtl of ex is
begin
  q<=conv_std_logic_vector(a, 4) when conv_integer (c) = 8 else
      conv_std_logic_vector(b, 4);
end;
```

اگر کلیه سیگنال‌ها از نوع std_logic باشند برنامه خواناتر خواهد بود. بنابراین می‌توان کد را به صورت زیر نوشت :

```
Entity ex is
port (a,b,c: in   std_logic_vector(3 downto 0);
      q:     out std_logic_vector(3 downto 0));
end;

Architecture rtl of ex is
begin
  q<=a when conv_integer (c) = 8 else b;
end;
```

چنان‌چه ابزار سنتز تابع افزایش ظرفیت داده شده '=' را برای داده‌هایی از نوع std_logic_vector و integer حمایت کند، می‌توان کد VHDL را خواناتر از قبل نیز نوشت :

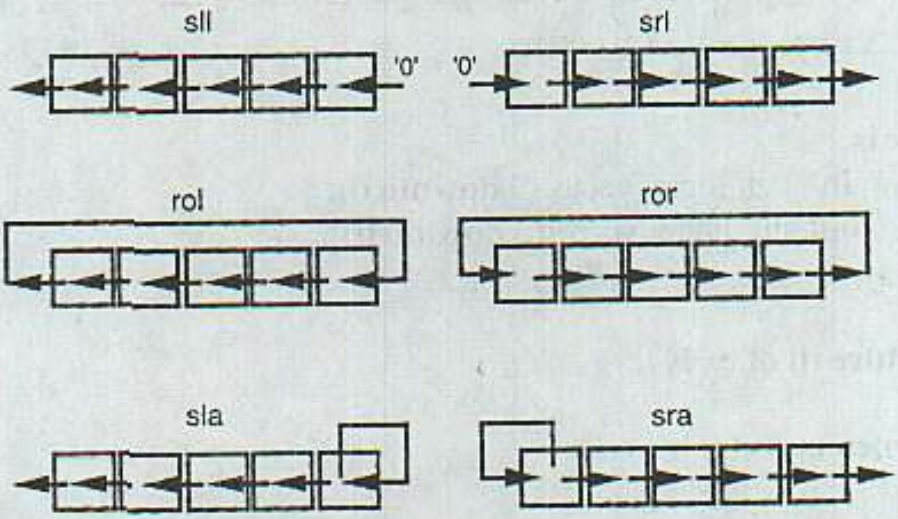
```
Entity ex is
port (a,b,c: in   std_logic_vector(3 downto 0);
      q:     out std_logic_vector(3 downto 0));
end;

Architecture rtl of ex is
begin
  q<=a when c=8 else b;
end;
```

از نقطه نظر سنتز برنامه، هیچ تفاوتی در استفاده کردن یا نکردن از توابع تبدیل نوع وجود ندارد. تابع تبدیل نوع هیچ گیتی استفاده نمی‌کند. یک تابع تبدیل نوع تنها به دو دلیل، یکی برای آسان‌تر شدن نوشتن برنامه و دیگری آسان‌تر شدن فهم متن نوشته شده، به کار می‌رود. یک طراحی خوب باید شامل تعداد کمی تابع تبدیل نوع باشد، به همین سبب توصیه شده است که در صورت امکان، داده‌های یک برنامه همگی از نوع std_logic_vector تعریف شوند. با ابزارهای سنتز مناسب که مقایسه std_logic_vector با integer را حمایت می‌کنند (توابع افزایش ظرفیت داده شده را حمایت می‌کنند) طراحی‌های پیچیده ASIC با VHDL بدون نیاز به استفاده از تابعهای تبدیل نوع انجام‌پذیر می‌شوند.

۵-۶ اپراتورهای شیفت

اپراتورهای شیفت به استاندارد VHDL-93 اضافه شده است. شش اپراتورهای شیفت برای داده از نوع bit_vector تعریف شده است :

* sll : شیفت به چپ
* srl : شیفت به راست
* rol : چرخه به چپ
* ror : چرخه به راست
* sla : شیفت به چپ با حفظ بیت سمت راست
* sra :شیفت به راست با حفظ بیت سمت چپ



شکل ۲-۵ اپراتورهای شیفت

مثال :

```
Architecture behv of ex is
begin
        a<= "01101";
        q1<= a sll 1;          -- q1= "11010"
        q2<= a srl 3;          -- q2= "00001"
        q3<= a rol 2;          -- q3= "10101"
        q4<= a ror 1;          -- q4= "10110"
        q5<= a sla 2;          -- q5= "10111"
        q6<= a sra 1;          -- q6= "00110"
end;
```

هنوز اکثر ابزارهای سنتز اپراتورهای شیفت را حمایت نمی‌کنند (بجز Autologic2)، گرچه بیشتر آنها در نظر دارند به زودی این اپراتورها را بپذیرند. اما تا آن زمان می‌توان توابع شیفت داخل بسته را به کار برد. مثلاً دو تابع شیفت در بسته ieee.std_logic_unsigned معین شده است («بسته‌های VHDL» در پیوست (ب) را ببینید). این توابع به صورت زیر تعریف شده‌اند :

```
function shl (arg: std_logic_vector, count: std_logic_vector)
            return std_logic_vector;
function shr (arg: std_logic_vector, count: std_logic_vector)
            return std_logic_vector;
        -- shl = Shift left
        -- shr = Shift right
```

مثال :

```
        q1<= shl (data, "1");          -- Shift on step left
        q2<= shr (data, "101");        -- Shift five step right
        q3<= shr (data, count);        -- Shift count step right
```

البته می‌توان کد VHDL را به گونه‌ای نوشت که نیازی به استفاده از اپراتورهای شیفت نباشد. به طور مثال :

```
Architecture rtl of ex is
signal data: std_logic_vector(7 downto 0);
begin
  process (clk, resetn)
  begin
        if resetn='0' then
```

```
            q1<=(others=>'1');
            q2<=(others=>'1');
        elsif clk'event and clk='1' then
            -- Shift one step right
             q1(6 downto 0)<=q1(7 downto 1);
             q1(7)<=d_in;
             -- Shift one step left
             q2(7 downto 1)<=q2(6 downto 0);
             q2(0)<=d_in;
        end if;
    end process;
end;
```

۷-۵ تمرین

۱- (i) در چه کتابخانه و بسته‌ای نوع داده bit تعریف شده است ؟

(ii) چه کتابخانه‌ها و بسته‌هایی همواره قابل رؤیت هستند و نیازی به نوشتن آنها نیست ؟

۲- چه المانی‌هایی را می‌توان در یک بسته ذخیره کرد؟ به چهار مورد اشاره کنید.

۳- (i) سه محلی را که یک function می‌تواند در آنجا تعریف شود مشخص کنید.

(ii) کدام یک از سه مورد فوق امکان استفاده دوباره از function را به راحتی فراهم می‌کند ؟

(iii) برای استفاده از بسته‌های دیگران چه کاری باید انجام شود ؟

۴- مفهوم اینکه می‌گویند «نوع داده std_logic یک نوع تعیین شده است» چیست ؟

۵- تفاوت بین یک function و یک procedure چیست ؟

۶- چرا طول بردار در بخش اعلام یک function/procedure تعریف نمی‌شود ؟

۷- افزایش ظرفیت یک تابع به چه معناست ؟

۸- (i) یک بسته را که شامل دو تابع زیر است طراحی کنید : average و sum. تابع average باید میانگین دو عدد (عمل گرد کردن به سمت پایین انجام شود) و تابع sum باید مجموع دو عدد را برگرداند. این دو تابع باید برای نوعهای داده integer و std_logic_vector تعریف شده باشند.

(ii) جزء ترکیبی c1 را با به کارگیری توابع فوق طراحی کنید. این جزء ترکیبی باید دارای چهار ورودی a، b، c و d باشد. ورودی‌های a و b از نوع integer (0 to 127) و سیگنال‌های ورودی c و d از نوع std_logic_vector (7 downto 0) تعریف شوند. جزء ترکیبی باید شامل چهار خروجی (average 1، average 2، sum 1 و sum 2) و هر کدام برای یک تابع باشد. average 1 و sum 1 باید از نوع integer (0 to 127) و average 2 و sum 2 از نوع std_logic_vector (7 downto 0) باشند.

۹- (i) تبدیل نوع چیست ؟

(ii) آیا در استاندارد VHDL توابع آماده‌ای برای تبدیل نوع وجود دارد ؟

۱۰- کدام یک از functionهای زیر صحیح هستند ؟

(1)

```
function f(a,b: in std_logic) return std_logic is
begin
        if a'event then
                return b;
        else
```

```
            return a;
        end if;
end;
```

(2)

```
function f (signal a,b: std_logic) return std_logic is
begin
        if a'event then
                return b;
        else
                return a;
        end if;
end;
```

(3)

```
function f (constant a,b: in std_logic) return std_logic is
begin
        if a'event then
                return b;
        else
                return a;
        end if;
end;
```

(4)

```
function f (a,b: in std_logic) return std_logic is
begin
        wait on a;
        return a;
end;
```

(5)

```
function f (signal a: in std_logic) return std_logic is
begin
      wait on a;
      return a;
end;
```

۱۱- (i)کدام یک از اپراتورهای شیفت در نسخه استاندارد VHDL-93 تعریف شده‌اند ؟

(ii) آیا امکان توصیف شیفت رجیسترها بدون استفاده از اپراتورهای شیفت وجود دارد ؟

فصل ششم

VHDL ساختاری

VHDL ساختاری را می‌توان به شبکه‌ای که اجزای مختلفی را به یکدیگر پیوند می‌دهد تشبیه نمود. یک entity به طور معمول از چندین جزء ترکیبی تشکیل می‌شود و کار VHDL ساختاری در طراحی سلسله مراتبی مشخص کردن رابطه درونی این اجزای است. در این سیستم هر جزء ترکیبی خود یکی از سطوح سلسله مراتبی را تشکیل می‌دهد. به طور مثال اگر یک جزء ترکیبی شامل چهار ورودی و سه خروجی باشد اما ۱۰۰۰ گیت داشته باشد، تنها پوسته این جزء ترکیبی (ورودی‌ها و خروجی‌های این واجد) در سطح بالای طراحی سلسله مراتبی قابل رؤیت خواهد بود و گیت‌ها در رده پایین‌تری قرار خواهند گرفت. در انجام پروژه‌های بزرگ حتماً باید با طراحی سلسله مراتبی کاملاً آشنایی داشت. مشخص کردن مرزهای طراحی سلسله مراتبی به طور دقیق امکان‌پذیر نیست و بستگی به پروژه طراحی دارد. به عنوان یک دستورالعمل کلی هنگامی که طراحی به بیش از ۳۰۰۰-۲۰۰۰ گیت نیاز داشته باشد، استفاده از طراحی سلسله مراتبی توصیه می‌شود. همان طور که در فصل «آشنایی با روشهای طراحی» توصیه شده است هر سطح طراحی سلسله مراتبی باید حدوداً بین ۵۰۰ تا ۶۰۰۰ گیت داشته باشد.

به طور ساده، VHDL ساختاری به نحوه ارتباط درونی و تسلسل داخلی تعدادی از اجزای ترکیبی اطلاق می‌شود. این رابطه درونی و تسلسل می‌تواند به صورت شماتیکی (در مقابل کدنویسی)

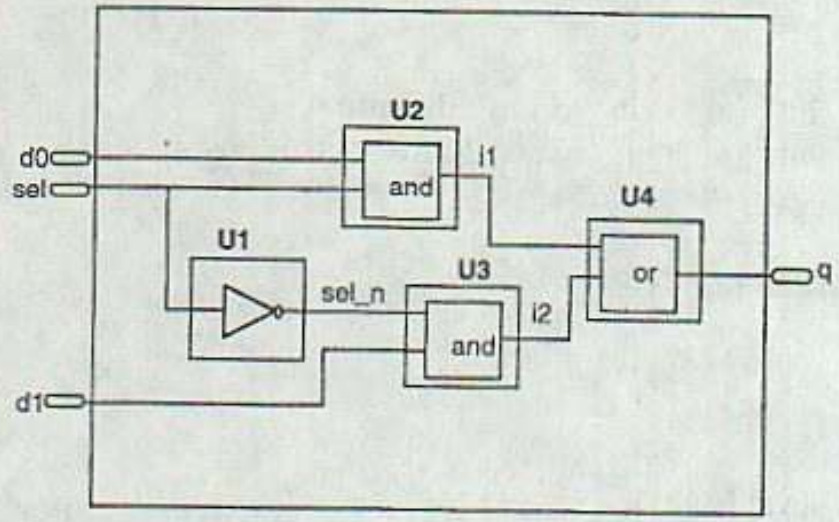
به نمایش درآید و به این ترتیب در صورتی که طرح خیلی بزرگ و پیچیده نباشد می‌توان یک نمای کلی از طراحی را ارائه داد. نمای شماتیکی می‌تواند اطلاعات فراوانی درباره اصول به کار رفته در طراحی را دارا باشد در حالی که کد VHDL می‌تواند حاوی مقادیر زیادی از جزییات باشد. در یک نمای شماتیکی شما به دنبال تمام سیگنال‌ها نیستید بلکه فقط به سیگنال اصلی نظر دارید. به این جهت می‌توان از طریق کشیدن یک بلوک دیاگرام که فقط نشان‌دهنده رابطه درونی اجزا باشد، بی‌آنکه تمام سیگنال‌ها را نمایش دهد، کار را کامل‌تر نمود. بلوک دیاگرام به راحتی با کمک خودکار و کاغذ و یا ابزارهای گرافیکی VHDL قابل ترسیم است. یکی از معایب استفاده از ابزارهای گرافیکی، وابسته کردن طرح به ابزارها و نسخه‌های خاص است، در حالی که کد ساختاری VHDL کاملاً غیروابسته به ابزار، نسخه و «ورژن» خاصی است. این بدان مفهوم است که اگرچه خواندن یک کد ساختاری کمی مشکل می‌شود ولی بسیار مفید است. چنان‌چه در طراحی سلسله مراتبی از VHDL ساختاری استفاده شود کل طرح به مدد VHDL تعریف می‌گردد و به این ترتیب می‌توان آن را از وابستگی به ابزار، نسخه، بستر و «ورژن»های دیگر تکنولوژی رها نمود.

مثال زیر اصول VHDL ساختاری را نمایش می‌دهد. نتیجه سنتز این برنامه در شکل ۱-۶ نشان داده شده است ـ گرچه به طور معمول نباید یک مالتی پلکسر را به این روش توصیف کرد. ادامه این فصل به بررسی اجزای دیگر VHDL ساختاری می‌پردازد.

مثال :

```
Entity mux is
port (d0,d1,sel: in    std_logic;
      q:         out   std_logic);
end;

Architecture struct_mux of mux is
  -- Component declaration.
  component and_copm
      port (a,b: in  std_logic;
            c:   out std_logic);
  end component;
  component or_comp
      port (a,b: in  std_logic;
            c:   out std_logic);
  end component;
  component inv_comp
      port (a: in std_logic;
            b: out std_logic);
```

```
end component;
signal i1,i2,sel_n: std_logic;

-- Component specification.
for U1: inv_comp Use Entity work.inv_comp(rtl);
for U2,U3: and_comp Use Entity work.and_comp(rtl);
for U4: or_comp Use Entity work.or_comp(rtl);

begin
-- Component instantiation.
U1 : inv_comp port map (sel,sel_n);
U2 : and_comp port map (d0,sel,i1);
U3 : and_comp port map (sel_n,d1,i2);
U4 : or_comp port map (i1,i2,q);
end;
```



شکل ۱-۶ نتیجه سنتز

۱-۶ معرفی جزء ترکیبی[1]

برای آنکه جزء ترکیبی بتواند در VHDL ساختاری فراخوانی شود باید ابتدا آن را در بخش اعلام VHDL ساختاری مابین architecture (بدنه و معماری) و begin (شروع) معرفی نمود.

1- Component

Syntax:
component <entity_name>
[**generic** (<generic_association-list>);]
port (<port_association-list>);
end component;
Syntax (VHDL-93):
component <entity_name>
. . .
end component [<entity_name>];

Port-list در بخش اعلام اجزای ترکیبی باید دقیقاً همتای آنچه باشد که در entity جزء ترکیبی آورده شده است. بهترین راه حصول اطمینان بر این نکته کپی کردن entity در بخش اعلام اجزای ترکیبی و یا برعکس است. در بخش اعلام، نیازی به تعریف ***پارامترهای عام***[1] نیست. فرض کنید جزء ترکیبی زیر (ex) باید در VHDL ساختاری فراخوانی شود. در این صورت کد VHDL برای معرفی جزء ترکیبی به فرم زیر خواهد بود :

```
Entity ex is
port ( a,b:  in   std_logic_vector(2 downto 0);
       q:    out  std_logic_vector(2 downto 0));
end;

Architecture rtl of ex is
begin
  . . .
end;

Entity top_level is
Port (. . .
      . . .);
end;

Architecture rtl of top_level is
  component ex                          -- Component declaration
     port (a,b: in  std_logic_vector(2 downto 0);
           q:  out std_logic_vector(2 downto 0));
```

1- Generic Parameters

```
  end component;
  ...
begin
    ...
end;
```

اگر ترتیب سیگنال‌ها در دستور port با آنچه در ابتدای ex (entity ex) آمده مطابق نباشد، به هنگام کامپایل کردن، خطا اعلام خواهد شد. بنابراین معرفی جزء ترکیبی به نحوی که در زیر آورده شده غلط است.

مثال (ناصحیح) :

```
Architecture bad of top_level is
  component ex                          -- Component declaration
      port (b,a:   in  std_logic_vector(2 downto 0); -- Error
            q:     out std_logic_vector(2 downto 0));
end component;
```

از آنجایی که در طراحی سلسله مراتبی به طور معمول چندین جزء ترکیبی فراخوانی می‌شود برای معرفی اجزای ترکیبی چندگانه از معماری top_level استفاده می‌شود.

مثال :

```
Architecture rtl of top_level is
  component c1
      port (a,b:   in  std_logic_vector(2 downto 0);
            q:     out std_logic_vector(2 downto 0));
  end component;

  component c2
      port (a,b:   in  std_logic_vector(4 downto 0);
            q:     out std_logic_vector(1 downto 0));
  end component;
  ...
```

۲-۶ مشخصات یک جزء ترکیبی

اگر شبیه‌ساز VHDL بخواهد یک جزء ترکیبی معرفی شده را شبیه‌سازی کند، باید مشخصه آن ذکر شده باشد. این کار توسط دستور «Use» انجام می‌گیرد.

Syntax:

For <lable>: <component_name> **Use entity**
<library> . <entity_name> (<architecture_name>);

از این مشخصه برای انتخاب کتابخانه‌ای که جزء ترکیبی باید در آنجا کامپایل شود و اینکه کدام معماری باید شبیه‌سازی شود استفاده می‌گردد. یک جزء ترکیبی VHDL این قابلیت را دارد که به چندین معماری مختلف مرتبط گردد. این بدنه‌ها می‌توانند معرف روشهای اجرایی یا سطوح توصیف مختلفی باشند. مشخص ساختن نام بدنه در مشخصه جزء ترکیبی، معماری لازم را برای شبیه‌سازی (یا کامپایل کردن) انتخاب می‌کند. فرض کنید جزء ترکیبی زیر (ex2) دارای دو معماری متفاوت باشد : معماری behv و معماری rtl . اگر بخواهیم بدنه rtl برای شبیه‌سازی انتخاب شود، مشخصه جزء ترکیبی باید به نحو زیر نوشته شود :

```
Entity ex2 is                    -- Component ex2
      Port (. . .
            . . .);
end;

Architecture behv of ex2 is      -- Architecture behv of ex2
begin
  . . .
end;
Architecture rtl of ex2 is       -- Architecture rtl of ex2
begin
  . . .
end;

Entity top_level is
  component ex2
      port (     . . .
                 . . .);
  end component;
  For U1: ex2 Use entity work.ex2(rtl);
  --Use architecture rtl of ex2
  begin
    . . .
  end;
```

مشخصه جزء ترکیبی نمی‌تواند توسط ابزارهای سنتز حمایت شود. استفاده از ابزارهای سنتز زمانی میسر است که یا جزء ترکیبی معرفی شده در حافظه ابزار سنتز وارد شده باشد یا آنکه در یکی از مسیرهای مشخص شده جستجو در ابزار سنتز قرار داشته باشد. ابزار سنتز حضور مشخصات جزء ترکیبی را در کد VHDL قبول می‌کند اما آن را نادیده می‌گیرد.

۳-۶ دستور Port map

آنچه تا به حال انجام دادیم معرفی و اعلام مشخصه جزء ترکیبی بوده است که باید فراخوانی شود. جزء ترکیبی با استفاده از دستور Port map فراخوانی و به سایر اجزای ترکیبی مرتبط می‌گردد.

Syntax:
port map (<port_association_list>);

فرض کنید که جزء ترکیبی زیر (ex3) را می‌خواهیم در کد top_level فراخوانی کنیم.

```
Entity ex3 is
port (a,b:  in  std_logic;
      q:    out std_logic);
end;

Architecture rtl of ex3 is
begin
  . . .
end;
```

بنابراین جزء ترکیبی (ex3) می‌تواند به گونه زیر در جزء ترکیبی top_level فراخوانی شود.

```
Entity top_level is
port (d1,d2 in  std_logic;
      q1:   out std_logic);
end;

Architecture rtl of top_level is
  component ex3
       port (a,b: in  std_logic;
             q:   out std_logic);
```

```
end component;
For U1: ex3 Use entity work.ex3 (rtl);
begin
    U1: ex3 port map (d1, d2, q1);
end;
```

این مثال روشی را که طی آن ارتباط درونی بین اجزای ترکیبی برقرار می‌شود روشن می‌سازد. به‌طور مثال سیگنال ورودی d1 از جزء ترکیبی top_level به سیگنال ورودی a در جزء ترکیبی ex3 متصل می‌گردد. دستور port map را همچنین می‌توان به صورتی نوشت که روشن سازد چه سیگنالی به چه سیگنالی متصل شود (خصوصاً که ترتیب سیگنال‌ها اهمیتی ندارد). بنابراین دستور port map را می‌توان به این صورت ذکر کرد:

```
U1 : ex3 port map (a => d1, b => d2, q => q1);
```

این نحو استفاده از دستور port map هیچ اثری بر روی نتیجه شبیه‌سازی یا سنتز برنامه ندارد. هر برچسبی را می‌توان به جای U1 انتخاب کرد. در سنتز می‌توان هم نام ex3 (به عنوان نام جزء ترکیبی) و هم نام U1 (به عنوان نام بخش فراخوانی شده) را به جزء ترکیبی نسبت داد.

۱-۳-۶ خروجی‌های متصل نشده

بعضی اوقات به هنگام فراخوانی یک جزء ترکیبی یکی از خروجی‌ها باید بدون اتصال باقی بماند. این کار به وسیله کلمه open انجام‌پذیر است. فرض کنید خروجی q2 در فراخوانی جزء ترکیبی ex4 باید بدون اتصال باقی بماند. در این صورت باید کد VHDL را به صورت زیر بنویسیم:

```
Architecture rtl of top_level is
  component ex4
      port (a,b:     in  std_logic;
            q1,q2:  out std_logic);
  end component;
  For U1: ex4 Use entity work.ex4 (rtl);
  begin
      U1: ex4 port map (a=>a, b=>b, q1=>dout, q2=>open);
  end;
```

همچنین مجاز خواهیم بود که خروجی متصل نشده را از دستور port map حذف کنیم. به این

ترتیب کد مربوط را می‌توان به شکل زیر نوشت :

```
U1 : ex4 port map (a => a, b => b, q1 => dont);
```

فقط در شرایطی می‌توان مطابق بالا عمل کرد که سیگنال حذف شده (سیگنال متصل نشده) در لیست ورودی‌ها و خروجی‌های جزء ترکیبی، آخرین سیگنال باشد. به عنوان مثال چنان‌چه سیگنال q1 متصل نشده باشد کد VHDL را نمی‌توان به شکل زیر نوشت :

```
U1 : ex4 port map (a, b, d2);
```

اگر کد VHDL به این صورت نوشته شود کامپایلر VHDL و ابزار سنتز سیگنال q2 (آخرین سیگنال) را متصل نشده تلقی خواهد کرد. بنابراین در این گونه موارد باید از نماد '=>' در دستور port map استفاده کرد.

مثال (صحیح) :

```
U1 : ex4 port map (a => a, b => b, q1 => open, q2 => d2);
```

بعضی از تولیدکنندگان ASIC سیگنال‌های متصل نشده را در یک بلوک طراحی سلسله مراتبی قبول نمی‌کنند. در این مورد باید کد VHDL را دوباره به گونه‌ای نوشت که سیگنال q2 خروجی جزء ترکیبی ex4 نباشد. این مشکل معمولاً زمانی روی می‌دهد که یک جزء ترکیبی در طراحی جدیدی دوباره به کار برده شود.

۲-۳-۶ ورودی‌های متصل نشده

هیچ نرم‌افزاری شناور بودن و متصل نبودن یک سیگنال ورودی را نمی‌پذیرد. چنان‌چه یک ورودی جزء ترکیبی مورد استفاده نباشد باید به VCC یا GND متصل شود. نمی‌توان مستقیماً ورودی را در دستور port map به کار برد؛ باید از یک سیگنال داخلی یا واسط که '1' یا '0' به آن نسبت داده شده است استفاده کرد. سپس آن را در سیگنال ورودی جزء ترکیبی که مورد استفاده نیست نگاشت داد. مثال ۱ چگونگی این کار را نشان می‌دهد. مثالهای ۲ و ۳ به نحوی که در بالا شرح داده شد در استاندارد VHDL-87 معتبر نیستند.

مثال ۱ (صحیح) :

```
Architecture rtl of top_level is
  component ex1
```

```
        port (a,b:      in  std_logic;
              q1,q2:    out std_logic);
  end component;
  For U1: ex1 Use entity work.ex4 (rtl);
  signal gnd: std_logic;
begin
        gnd<='0';
        U1: ex1 port map (a=>gnd, b=>b, q1=>dout, q2=>d2);
end;
```

مثال ۲ (ناصحیح) :

```
Architecture bad of top_level is
  component ex1
        port (a,b:      in  std_logic;
              q1,q2:    out std_logic);
  end component;
  For U1: ex1 Use entity work.ex4 (rtl);
begin
        U1: ex1 port map (a=> open, b=>b, q1=dout, q2=>d2);
                        -- Error
end;
```

مثال ۳ (VHDL-93) :

```
Architecture bad of top_level is
  component ex1
        port (a,b:      in  std_logic;
              q1,q2:    out std_logic);
  end component;
  For U1: ex1 Use entity work.ex4 (rtl);
begin
        U1: ex1 port map (a=>'0', b=>b, q1=>dout, q2=>d2);
                        -- Error (VHDL-87), valid in VHDL-93
end;
```

در استاندارد VHDL-93 امکان اتصال مستقیم به '0' و '1' وجود دارد. بنابراین می‌توان برنامه را همان طور که در مثال ۳ آورده شده است به کار برد.

٤-۶ دستور Generic map

چنان‌چه پارامترهای عمومی (generic) در یک جزء ترکیبی که باید فراخوانی شود مشخص شده باشند، ارزش و مقدار آنها می‌تواند در حین فراخوانی با دستور generic map تغییر کند. این دستور چنین است :

Syntax:
generic map (<generic_values>);

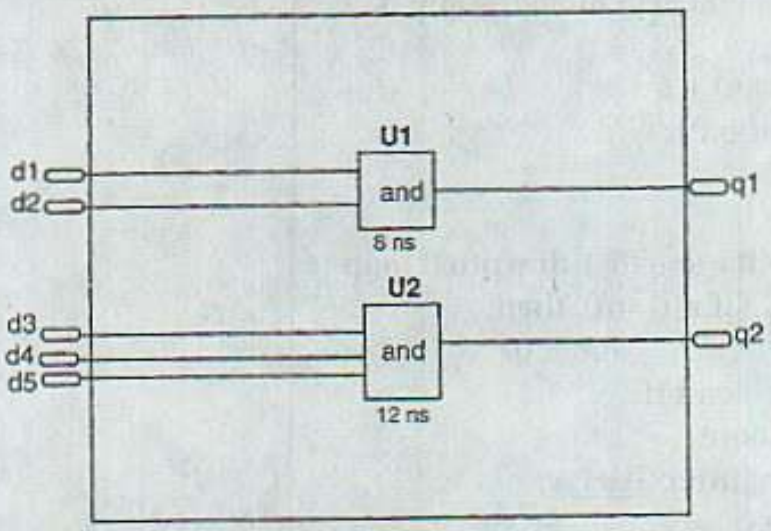
با استفاده از پارامترهای عمومی می‌توان اجزایی را طراحی کرد که قابلیت پارامترگذاری داشته باشند. در زیر مثالی از یک جزء ترکیبی آورده شده است که در آن پارامتر عمومی تأخیر و تعداد ورودی‌ها نامعلوم است. در زمان به کارگیری این جزء ترکیبی باید این دو مقدار را معین نمود.

```
Entity and_comp is
        generic (Tdelay:time:=10 ns;
                 n:positive:=2);
   port (a: in   bit_vector(n-1 downto 0);
         c: out bit);
end;

Architecture and_beh of and_comp is
begin
  p0: process(a)
  variable int:bit;
  begin
        int:= '1';
        for i in a'length-1 downto 0 loop
              if a(i) = '0' then
                    int=: '0';
              end if;
        end loop;
        c<=int after Tdelay;
  end process;
end;
```

چنان‌چه یک جزء ترکیبی and با سه ورودی و تأخیر ۱۲ns و یک جزء ترکیبی دیگر با دو ورودی و ۸ns تأخیر موردنظر باشد، می‌توان به صورت زیر عمل کرد :

```
Entity ex is
port (d1, d2, d3, d4, d5:  in  bit;
      q1,q2:               out bit);
end;

Architecture ex_beh of ex is
  component and_comp
        generic (Tdelay:time;
                 n: positive);
        port (a:   in  bit_vector(n-1  downto  0);
              c:   out bit);
  end component;
  For U1,U2 : and_comp Use entity work.and_comp (and_beh);
begin
  U1: and_comp generic map(n=>2, Tdelay=>8 ns)
      port map (a(0)=> d1, a(1)=> d2, c=> q1);
  U2: and_comp generic map(n=>3, Tdelay=> 12 ns)
      port map (a(0)=d3, a(1)=> d4, a(3)=>d5, c=>q2);
end;
```

نتیجه سنتز در شکل ۲-۶ نشان داده شده است.



شکل ۲-۶ نتیجه سنتز جزءهای ترکیبی and با پارامترهای مختلف

همان طور که قبلاً نیز اشاره شد، تأخیرهای ۱۲ns و ۸ns توسط ابزارهای سنتز نادیده گرفته می‌شوند و تنها در شبیه‌سازی اثرشان معلوم خواهد شد.

۶-۵ دستور Generate

چنان‌چه لازم باشد در یک برنامه از یک جزء ترکیبی چندین بار استفاده شود، بهتر است که دستور port map را در یک حلقه قرار دهیم.

Syntax:

```
[label:] For <loop_index> <iteration scheme> generate
   <lable>: <entity> port map (<port_association_list>);
end generate [lable];
```

فرض کنید جزء ترکیبی C1 را باید پنج بار در جزء ترکیبی top استفاده کنیم. این کار با کمک دستور generate انجام می‌شود.

مثال :

```
Entity top is
port (a,b:  in   std_logic_vector(4 downto 0);
      q:    out  std_logic_vector(4 downto 0));
end;

Architecture rtl  of top  is
  component c1
  port (a,b:  in   std_logic;
        q:    out  std_logic);
  end copmonent;
For U1: c1 Use entity work.c1 (rtl);
begin
  c_gen: For i in 0 to 4 generate
      U: c1 port map (a(i), b(i), q(i));
  end generate c_gen;
end;
```

با به کارگیری دستور generate استفاده چندین باره از یک جزء ترکیبی بسیار انعطاف‌پذیرتر و عمومی‌تر می‌شود. در کلیه ابزارهای سنتز که دستور generate را حمایت می‌کنند باید تعداد تکرار مشخص باشد.

۶-۶ پیکره‌بندی[1]

کار پیکره‌بندی مرتبط ساختن یک معماری یا بدنه با entity است. ساده‌ترین حالت استفاده نکردن از هیچ نوع پیکره‌بندی است. در این صورت آخرین معماری از برنامه‌ای که کمپایل شده است شبیه‌سازی می‌شود. حالت دیگر پیکره‌بندی اولیه است. زمانی از این نوع پیکره‌بندی استفاده می‌شود که در برنامه فقط یک جزء ترکیبی به کار رفته باشد. در زیر مثالی از یک MUX آورده شده که دارای دو معماری behv و rtl است. پیکره‌بندی مشخص می‌کند که کدام یک از این دو باید شبیه‌سازی شوند. نام پیکره‌بندیی که در طی آن معماری rtl شبیه‌سازی می‌شود mux_rtl و در مورد معماری behv نام پیکره‌بندی mux_behv است.

خط A

مثال :

```
Entity mux is
port (a,b,c,d: in  std_logic_vector(3 downto 0);
      sel:     in  std_logic_vector(1 downto 0);
      q:       out std_logic_vector(3 downto 0));
end;

Architecture behv of mux is
begin
  process(a,b,c,d,sel)
  variable int: std_logic_vector(3 downto 0);
  begin
       case sel is
           when "00"  => int<=a;
           when "01"  => int<=b;
           when "10"  => int<=c;
           when "11"  => int<=d;
           when others => int<=(oters=> 'X');
       end case;
       q<=int after 10 ns;
  end process;
end;
```

1- Configuration

```
Architecture rtl of mux is
begin
  process(a,b,c,d,sel)
  begin
        case sel is
            when "00"  => q<=a;
            when "01"  => q<=b;
            when "10"  => q<=c;
            when others => q<=d;
        end case;
  end process;
end;

-- Default configuration for architecture behv
Configuration mux_behv of mux is
   For behv
   end for;
end mux_behv;

-- Default configuration for architecture rtl
Configuration mux_rtl of mux is
   For rtl
   end for;
end mux_rtl;
```

برحسب اینکه کدام یک از پیکره‌بندی‌های فوق باید کامپایل شود معماری behv یا rtl شبیه‌سازی خواهد شد. اگر در داخل جزء ترکیبی اجزای دیگری فراخوانی نشده باشد به طور نرمال پیکره‌بندی در طرح حذف می‌شود. به این معنا که می‌توان معماریی را که باید شبیه‌سازی شود مستقیماً وارد نمود. ابزارهای سنتز معمولاً تمام پیکره‌بندی‌ها را نادیده گرفته و در عوض به سنتز کردن آخرین معماریی که به ابزار وارد شده است می‌پردازد.

اگر جزء ترکیبی که باید شبیه‌سازی شود اجزای دیگری در بر داشته باشد اطلاعات اضافی باید در پیکره‌بندی گنجانده شود. ساده‌ترین حالت در این مورد در اصل همان اعلام مشخصات جزء ترکیبی است که در اوایل این فصل به آن اشاره شد. مشخصه جزء ترکیبی در بخش اعلام معماری برنامه آورده می‌شود.

مثال ۱ :

```
Entity ex is
port (a,b: in   std_logic;
      c:   out  std_logic);
end;

Architecture behv of ex is
begin
  . . .
end;

Entity top_level is
  component c1
       port (a,b: in   std_logic;
             c:   out  std_logic);
  end component;
  For U1: ex Use entity work.ex(rtl);    -- Simple form of
begin                                    -- configuration
   . . .
end;
```

مثال ۲ :

```
Entity c1 is
port (a,b: in   std_logic;
      c:   out  std_logic);
end;

Architecture behv of c1 is
begin
       c<=a nor b after 10 ns;
end;

Configuration c1_conf is
  For behv
  end for;
end;

Entity c2 is
port (a,b: in   std_logic;
```

```
      c:    out  std_logic);
end;

Architecture behv of c2 is
begin
        c<=a xor b after 12 ns;
end;

Configuration c2_conf is
  For behv
  end for;
end;

Entity top_level is
port (d1,d2,d3:  in   std_logic;
      q1, q2:    out  std_logic);
end;

Architecture top_behv of top_level is
  component c1
        port (a, b:  in   std_logic;
                c:    out  std_logic);
  end copmonent;
  component c2
        port (a,b:  in    std_logic;
                c:    out  std_logic);
  end component;
  signal i1: std_logic;
  begin
        U1: c1 port map (d1, d2, i1);
        U2: c2 port map (d3, i1, q2);
        q1<=il;
  end;

Configuration top_conf is
  For top_behv
        For U1: c1 Use configuration work.c1_conf;
        end for;
```

```
        For U2: c2 Use configuration work.c2_conf;
        end for;
    end for;
end top_conf;
```

پیکره‌بندی top_conf مشخص می‌کند که c1_conf و c2_conf به ترتیب برای پیکره‌بندی اجزای ترکیبی c1 و c2 به کار می‌روند. همچنین تعیین می‌کند که معماری top_behv از جزء ترکیبی top_level برای شبیه‌سازی مورد استفاده قرار می‌گیرد.

به طور خلاصه می‌توان گفت که پیکره‌بندی غالباً برای طراحی مدلهای مختلف شبیه‌سازی به کار می‌رود و در طراحی‌های سخت‌افزاری کاربرد اندکی دارد.

۷-۶ فراخوانی مستقیم (VHDL-93)

فراخوانی مستقیم اجزای ترکیبی در استاندارد VHDL-93 امکان‌پذیر است. در این روش نیازی به بخش اعلام جزء ترکیبی و پیکره‌بندی آن نیست. مثالهای زیر چگونگی فراخوانی در استاندارد VHDL-87 و به دنبال آن روش جدید فراخوانی در استاندارد VHDL-93 را نشان می‌دهند. لازم به یادآوری است که امکان فراخوانی به روش VHDL-87 در نسخه جدید استاندارد VHDL نیز وجود دارد.

مثال (VHDL-87) :

```
Architecture rtl of top_level is
  component c1
      port (a, b:  in   std_logic;
            q:    out  std_logic);
  end component;
For U1: c1 Use entity work.c1(rtl);
begin
     U1: c1 port map (a,b,q);
end;
```

مثال (VHDL-93) :

```
Architecture rtl of top_level is
begin
      U1: entity work.c1 (rtl) port map (a,b,q);
end;
```

متأسفانه تعدادی از ابزارهای سنتز تاکنون روش جدید و سریع فراخوانی یک جزء ترکیبی را حمایت نمی‌کنند.

۸-۶ اجزای ترکیبی در بسته‌ها

امکان تعریف کردن جزء ترکیبی در یک بسته وجود دارد. یکی از امتیازات این کار بی‌نیاز شدن از بخش اعلام جزء ترکیبی در معماری برنامه به هنگام فراخوانی آن است.

مثال :

```
package mypack is
  function minimum (a,b:in std_logic_vector)
           return std_logic_vector;
  component mycomp1
       port (clk,resetn,din: in   std_logic;
             q1,q2:          out std_logic);
  end component;
  component mycomp2
       port (a,b : in  std_logic;
             q:    out std_logic);
  end component;
end mypack;

package body mypack is
  function minimum (a,b: in std_logic_vector)
           return std_logic_vector is
  begin
       if a<b then
            return a;
       else
            return b;
       end if;
  end minimum;
end mypack;
```

اجزای ترکیبی mycomp1 و mycomp2 با معین کردن بسته mypack در ابتدای کد VHDL قابل استفاده می‌شوند. از آنجایی که جزء ترکیبی در بسته تعریف شده و بسته نیز در کتابخانه work ذخیره شده است دیگر نیازی به معرفی و بیان مشخصات جزء ترکیبی به هنگام فراخوانی آن نیست.

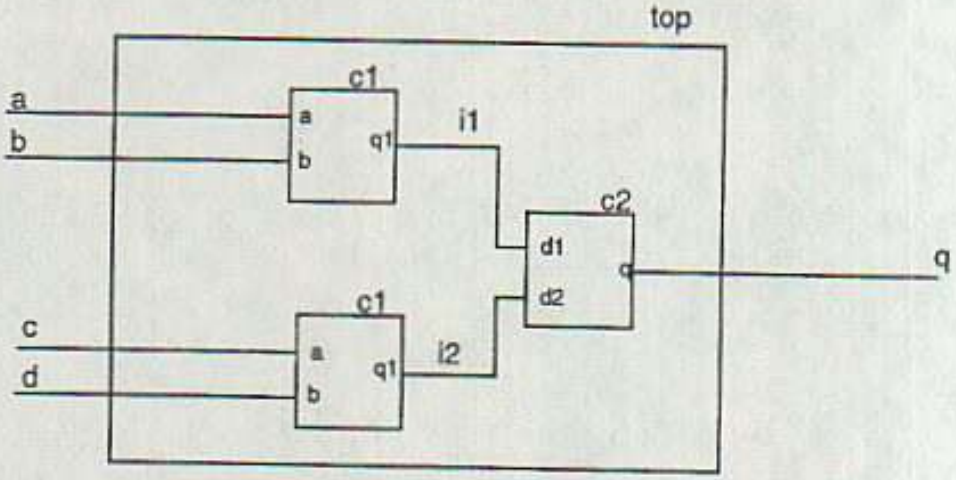
مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use work.mypack.ALL;
Entity ex is
port (clk,resetn:       in  std_logic;
      d1,d2:            out std_logic;
      a,b:              in  std_logic_vector(3 downto 0);
      q1,q2,q3:         out std_logic;
      q4:               out std_logic_vector(3 downto 0));
end;

Architecture rtl of ex is
begin
      U1: mycomp1 port map (clk,resetn,d1,q1,q2);
      U2: mycomp 2 port map (d1,d2,q3);
      q4<=minimum (a,b);
end;
```

۹-۶ تمرین

۱- طراحی سلسله مراتبی چیست ؟

۲- بلوک دیاگرام زیر را با VHDL ساختاری توصیف کنید. با فراخوانی جزء ترکیبی c1 (دو بار) و جزء ترکیبی c2 در جزء ترکیبی top .



۳- (i) آیا امکان مرتبط بودن چندین entity با یک architecture وجود دارد ؟

(ii) آیا امکان مرتبط بودن چندین architecture با یک entity وجود دارد ؟

۴- اگر چندین architecture وجود داشته باشد، چگونه می‌توان مشخص کرد که کدام یک باید شبیه سازی یا سنتز شود ؟

۵- برای تغییر پارامتر عمومی در هنگام فراخوانی از چه دستوری باید استفاده کرد ؟

۶- (i) یک گیت OR عمومی با N ورودی و تأخیر T نانوثانیه طراحی کنید. در حالت اولیه N = ۳ و T = ۳ نانوثانیه باشد.

(ii) جزء ترکیبی عمومی طراحی شده در فوق را در جزء ترکیبی top دو بار فراخوانی کنید. در یکی پارامترهای عمومی N = ۴ و T = ۲ نانوثانیه و در دیگری N = ۵ و T = ۳ نانوثانیه باشند.

۷- جزء ترکیبی c1 را ۱۰ بار در جزء ترکیبی top فراخوانی کنید.

```
Entity c1 is
port (a,b: in   std_logic;
      q:   out std_logic);
end;
```

جزء ترکیبی top باید دارای entity زیر باشد :

```
Entity top is
port (a,b: in   std_logic_vector(9 downto 0);
      q:   out std_logic_vector(9 downto 0));
end;
```

راهنمایی : از دستور generate استفاده کنید.

فصل هفتم

RAM و ROM

هر دو نوع حافظه RAM و ROM می‌توانند در طراحی‌های ASIC و FPGA به کار روند. این فصل به چگونگی نوشتن کد این دو نوع حافظه می‌پردازد.

۷-۱ ROM

دو روش برای تعیین ROM (Read Only Memory) در VHDL وجود دارد :

- به کارگیری آرایه ثابت
- به کارگیری ROM با مشخصات تکنولوژیکی خاص

۷-۱-۱ به‌کارگیری آرایه ثابت

اگر ROM موردنیاز تقریباً کوچک باشد، برای ذخیره کردن و استفاده مؤثر از فضا، می‌توان آن را با یک آرایه توصیف کرد. اگر ROM دارای ظرفیت بالا باشد با این روش حجم زیادی اشغال خواهد شد. پیشنهاد می‌شود ROM تعریف شده با آرایه ثابت را در یک بسته قرار دهید. این کار امکان استفاده

مجـدد ROM را فراهـم می‌کنـد. اندازه ROM بایـد توسط بسـته معین شـود. در زیـر مثالـی از یـک ROM 8 × 4 بیتی (۴ بایتی) آورده شده است :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Package rom is
   constant rom_width:integer:=8;
   constant rom_length:integer:=4;
   subtype rom_word is std_logic_vector
      (rom_width - 1 downto 0);
   type rom_table is array (0 to rom_length - 1) of rom_word;
   constant rom: rom_table:=rom_table'("00101111",
                                       "11010000",
                                       "01101010",
                                       "11101101");

end;
```

بنابراین می‌توان ROM را در یک طراحی انجام شده با VHDL به شکل زیر به کار برد :

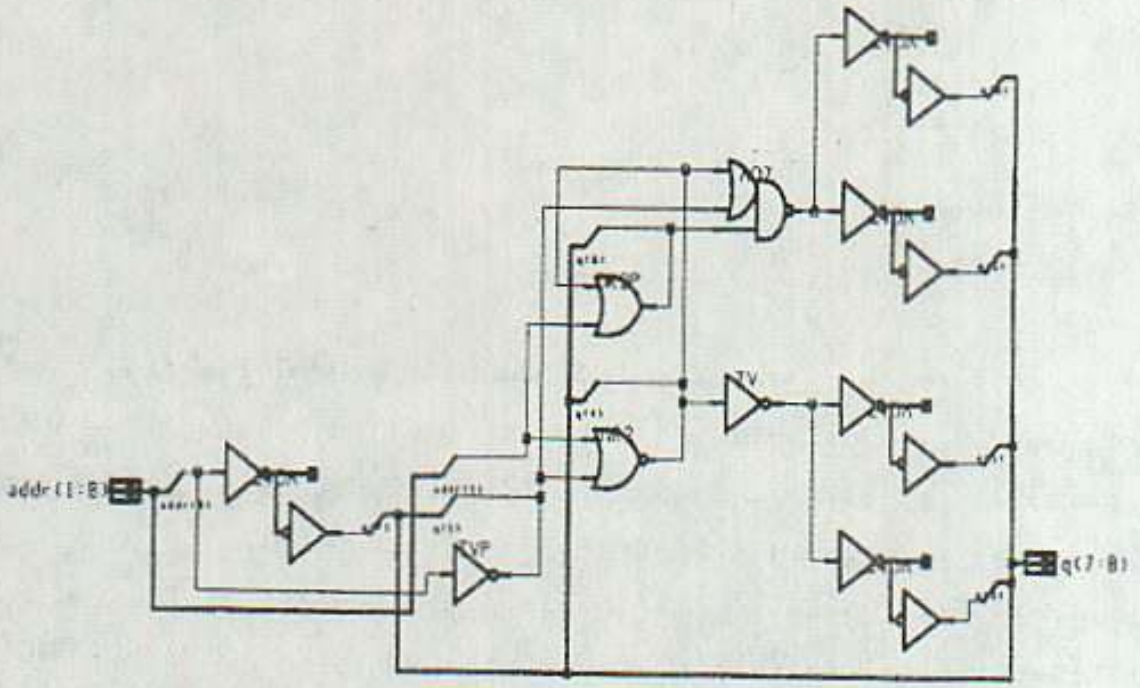
```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.rom.ALL;

Entity waveform is
port (addr: in std_logic_vector(1 downto 0);
      q:    out rom_word);
end;

Architecture testbench of waveform is
begin
   q<=rom(conv_integer(addr));  -- Read data from ROM
end;
```

ابزارهای سنتز ROM را با کمک منطق ترکیبی می‌سازند. شکل ۱-۷ نتیجه سنتز مثال بالا را نشان می‌دهد.

در مثال بالا، آدرس ROM (addr) مشخص نبود. اگر آدرس قابل محاسبه باشد سیگنال خروجی q یک مقدار ثابت خواهد بود، در این صورت ابزار سنتز هیچ منطق ترکیبی را برای چنین طراحی به وجود نخواهد آورد.

مثالی از مشخص بودن عدد آدرس ROM می‌تواند به صورت زیر باشد :

```
Architecture testbench of waveform is
begin
   q<=rom(2);          -- Read data from ROM
end;
```

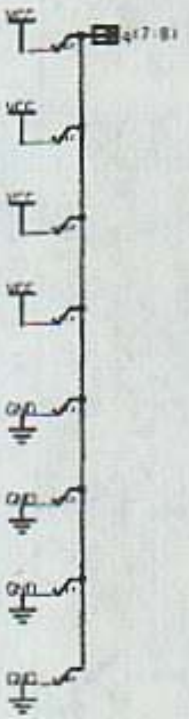


شکل ۱-۷ نتیجه سنتز ROM، مثال اول

از آنجایی که آدرس ROM مشخص بوده، بنابراین هیچ منطقی حاصل نشده است. در این صورت کد VHDL فقط مقدار rom(2) را به سیگنال خروجی q نسبت داده است که به صورت مقدار ثابت "01101010" خواهد بود.

با تغییر دادن مقادیر ثابت در داخل بسته ظرفیتهای مختلفی از ROM به دست خواهد آمد.

چنان‌چه مثال فوق سنتز شود نتیجه آن به صورت شکل ۲-۷ خواهد بود :



شکل ۲-۷ نتیجه سنتز ROM، مثال دوم

۲-۱-۷ به کارگیری ROM با مشخصات تکنولوژیکی خاص

به هنگام استفاده از ROM های بزرگ، از آنجایی که استفاده از آرایه‌های ثابت حجم زیادی اشغال خواهند کرد بنابراین باید از کتابخانه‌های خاص استفاده شود. به کارگیری این ROM ها معایبی نیز دارد. یکی از این معایب آن است که چون از ROM بزرگ در کتابخانه به عنوان مرجع استفاده می‌شود کد VHDL وابسته به تکنولوژی می‌گردد. یکی دیگر از معایب، آن است که اگر هیچ مدل VHDL برای ROM ارائه نشده باشد در هنگام شبیه‌سازی VHDL مشکلاتی بروز خواهد کرد. به عنوان راه‌حل می‌توان از یک شبیه‌ساز آمیخته (برای VHDL و برای سطح گیتی ROM) استفاده کرد. فراخوانی ROM دقیقاً مشابه دیگر اجزای ترکیبی است. (برای اطلاعات بیشتر در مورد فراخوانی به فصل ۶ مراجعه کنید، «VHDL ساختاری»).

۳-۱-۷ خلاصه

دو انتخابی که برای معرفی ROM وجود دارد به طور خلاصه در جدول ۱-۷ با هم مقایسه شده‌اند.

	استفاده بهینه از فضا	شبیه‌سازی آسان	عدم وابستگی به تکنولوژی
انتخاب ۱	خیر	بله	بله
انتخاب ۲	بله	خیر	خیر

جدول ۱-۷ مقایسه

RAM ۷-۲

عیناً مانند ROM دو روش برای معرفی RAM (Randon Access Memory) در کد VHDL وجود دارد :

- استفاده از رجیسترها
- فراخوانی RAM

۱-۲-۷ استفاده از رجیسترها

برای RAM های بسیار کوچک می‌توان از این روش استفاده کرد. اما در مورد RAM های بزرگ‌تر، فراخوانی RAM در مقایسه با به کارگیری رجیسترها به فضای کمتری نیاز خواهد داشت. همان طور که در فصل چهارم توضیح داده شد رجیسترها با استفاده از پروسس پالسی توصیف می‌شوند.

```
Process (clk,resetn)
begin
  if resetn='0' then
       q<=(others=> '0');
  elsif clk'event and clk= '1' then
    if wr= '1' then
            q<=data;
    end if;
  end if;
end process;
```

۲-۲-۷ فراخوانی RAM

هرگاه RAM فراخوانی شود می‌توان از سنتز آن نتیجه مناسبی را به دست آورد. RAM نیز درست مانند ROM معایبی دارد، از جمله آن که کد VHDL وابسته به تکنولوژی می‌شود و در نتیجه

مشکلاتی در کار شبیه‌سازی VHDL بروز خواهد کرد. RAM های موجود بر حسب سلیقه سازنده و نوع تکنولوژی آن با هم تفاوت دارند.

می‌توان با فراخوانی چندین RAM یا با فراخوانی چندباره یک RAM به اندازه و حجم مورد نیاز دست یافت. دستور generate می‌تواند برای این کار مفید باشد. به طور مثال اگر یک RAM 4 × 1 بیتی را چهار بار فراخوانی کنیم، RAM 4 × 4 بیتی به دست خواهد آمد :

```
Architecture rtl of RAM4 is
component RAM4x1
   port (d,a0,a1,we: in   std_logic;
         q:          out std_logic);
end component;
-- for U1: RAM4x1 use Entity work.RAM4x1(rtl);
                -- Only used for simulation

begin
   for i in 0 to 3 generate
        RAM_b: RAM4x1 port map (d_in(i),a0,a1,write,d_out(i));
   end generate;
end;
```

با استفاده از ***ابزارهای سنتز رفتاری***[1] نیز می‌توان به طور خودکار RAM را معرفی کرد. این روش سریع‌ترین، ساده‌ترین و بهترین راه معرفی RAM است، زیرا هم از فضا استفاده بهتری می‌کند، هم شبیه‌سازی را آسان‌تر می‌سازد و هم مستقل از تکنولوژی عمل می‌نماید. کد VHDL از یک آرایه برای خواندن و نوشتن سیگنالها استفاده می‌کند و سپس ابزار سنتز این آرایه را در هر RAM دیگری از میان RAM های مختلف جایگزین خواهد کرد (به فصل ۱۷ مراجعه کنید، «سنتز رفتاری»).

1- Behavioural Synthesis Tools

۳-۷ تمرین

۱- مزایا و معایب دو روش معرفی ROM در کد VHDL چیست ؟

۲- مزایا و معایب سه روش معرفی RAM در کد VHDL چیست ؟

۳- جزء ترکیبی c1 را که در آن یک RAM از نوع 8 × 4 بیت فراخوانی شده است طراحی کنید. کتابخانه فقط شامل RAM از نوع 1 × 4 بیتی است.

```
Entity RAM4x1 is
port (d,a0,a1,we: in  std_logic;
      q:          out std_logic);
end;
```

فصل هشتم

محیط آزمایش

پس از توصیف پروژه توسط طراح، طرح حاصل باید مورد بررسی قرار گیرد تا اطمینان حاصل شود که خصوصیات و ویژگی‌های موردنظر را به طور کامل دارا است یا خیر.

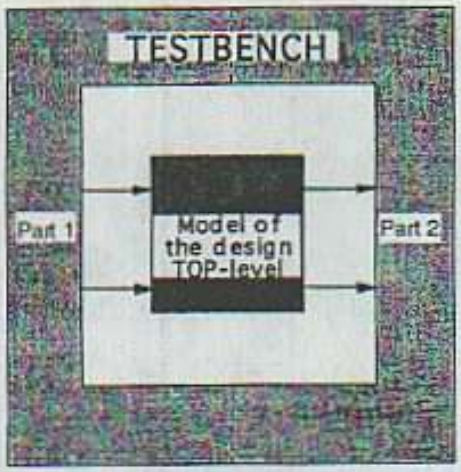
معمول‌ترین روش، به کارگیری سیگنال‌های محرک ورودی در خلال یک پروسه شبیه‌سازی و متعاقباً «خواندن» سیگنال‌های خروجی از طرح است (آنالیز منطقی). یک عیب عمده، متفاوت بودن محرکهای ورودی در شبیه‌سازهای گوناگون است.

یک روش دیگر برای تأیید درستی طرح، نوشتن برنامه یک مدل آزمایشی برای تست و بررسی خروجی‌های ناشی از کد موردنظر است. این کار به معنای طراحی یک محیط آزمایشی است که هم سیگنال‌های ورودی را در اختیار می‌گذارد و هم سیگنال‌های خروجی طرح را مورد آزمایش قرار می‌دهد. از آنجایی که این محیط آزمایش نیز ممکن است خطا ایجاد کند، خود آن نیز باید تست شود. توصیه می‌شود که هم از محیط آزمایش VHDL و هم از تست اجزای ترکیبی با به کارگیری این محیط، که به نام «شبیه‌سازی سیستم» معروفند استفاده شود (مراجعه نمایید به فصل یازدهم، «آشنایی با روشهای طراحی»).

مزیت شبیه‌سازی سیستم آن است که محرکهای مربوط به اجزای ترکیبی توسط مدلهایی که بیانگر سیگنال‌های حقیقی هستند تولید می‌شود. عیب شبیه‌سازی سیستم آن است که وقت زیادی را مصرف می‌کند. از مزایای محیط آزمایش VHDL یکی سرعت آن و دیگری عدم وابستگی به تکنولوژی و بستر اجرایی است، زیرا با کد VHDL نوشته می‌شود. اما چون طراح پروژه، این محیط را طراحی

می‌کند ممکن است خطاهای منطقی مشابه خطاهای داخل پروژه در محیط آزمایش به وجود آیند که رفع نخواهند شد.

در بعضی از موارد بهتر آن است که به ساخت یک محیط آزمایش کامل یا یک محیط نیمه کامل اقدام نمود. شکل ۱-۸ ساختار منطقی یک محیط آزمایش را نشان می‌دهد. مدل، محرکهای ورودی (سیگنال‌های ورودی) را دریافت می‌کند و پاسخ (سیگنال‌های خروجی) را به محیط آزمایش ارسال می‌دارد.



شکل ۱-۸ مدل طرح و محیط آزمایش

محیط آزمایش از دو بخش تشکیل می‌شود. یک بخش سیگنال‌های ورودی را برای مدلی که تحت آزمایش است تولید می‌کند و بخش دیگر به چک کردن سیگنال‌های خارج شده از مدل می‌پردازد.

بخش ۱

مثال زیر چگونگی تولید سیگنال‌های ورودی مدل را بیان می‌کند :

```
in_model<=      '1' after 5 ns,
                '0' after 100 ns,
                '1' after 200 ns,
                '0' after 300 ns + 4 ns;
clk<= not clk after 50 ns;
```

سیگنال in_model در اصل تولیدکننده سیگنالی است که هر ۵، ۱۰۰، ۲۰۰ و ۳۰۴ نانوثانیه عکس می‌شود. سیگنال clk نیز با دوره تناوب ۱۰۰ نانوثانیه نوسان می‌کند.

اگر سیگنال clk از نوع bit تعیین نشود، عبارت clk <= not clk after 50ns ایجاد مشکل خواهد کرد. در ضمن از آنجایی که توصیه شده است در طراحیهای VHDL از داده‌های نوع std_logic استفاده شود، باید سیگنال clk از این نوع باشد تا بتوانیم از توابع تبدیل نوع اجتناب کنیم. مشکلی که در اینجا پیش می‌آید این است که کلیه سیگنال‌ها در VHDL دارای مقدار اولیه‌ای برابر با سمت چپ‌ترین ارزش در تعریف نوع داده خواهند بود. در این مورد یعنی ارزش 'U' (نامعین) برای سیگنال clk. وارونه 'U' همان 'U' خواهد بود. این مشکل به راحتی با مقداردهی اولیه به سیگنال clk حل خواهد شد. به طور مثال در بخش اعلام، برای سیگنال clk خواهیم داشت :

```
signal clk : std_logic := '0';
```

چنان‌چه المان تست‌شونده شامل ***ماشین حالت***[1] باشد، اگر از دستور wait برای نسبت دادن مقادیری به سیگنال‌های ورودی استفاده شود بهتر خواهد بود.

```
Process
begin
   in_signal1<= '1',      -- Input signals value at reset
   in_signal2<= '0';
   wait until resetn= '1';

   wait until clk= '1';
   in_signal1<= '1';
   in_signal2<= '0';

   wait until clk= '1';
   in_signal1<= '1';
   in_signal2<= '0';
end process;
```

یک روش دیگر انتظار برای لبه پالس ساعت، تعریف دوره تناوب این پالس به عنوان یک مقدار ثابت و استفاده از آن در دستور wait است. به طور مثال :

```
constant period : time := 50ns;
wait for period;
```

1- State Machine

مزیت این روش این است که مدل رفتاری نیازی به سیگنال clk نخواهد داشت. در ضمن امکان انتظار به اندازه چندین سیکل پالس ساعت وجود دارد :

```
wait for 2 * period;
```

چنان‌چه لازم باشد از چند برابر دوره تناوب پالس ساعت به طور مکرر استفاده شود بهتر است آن زمان را به عنوان یک ثابت دیگر تعریف کنیم :

```
constant period : time := 50ns;
constant period 2 : time := 2 * period;
...
wait for period 2;
```

از آنجایی که شبیه‌ساز مجبور به محاسبه چندین باره مقدار 2 * period خواهد بود بنابراین توصیه می‌شود از مقدار ثابت استفاده گردد تا زمان کمتری برای شبیه‌سازی صرف شود.

بخش ۲

بخش دوم محیط آزمایش وظیفه بررسی سیگنال‌های خروجی مدل را بر عهده دارد. عبارت assert معمولاً برای این منظور به کار می‌رود. دستور assert زمانی اجرا می‌شود که عبارت تعیین شده نادرست (false) باشد.

یک مثال از دستور assert در زیر آورده شده است :

```
Process(clk)
begin
assert now < 900 ns
   report "stopping simulator (max simulation time 900 ns)"
   severity Failure;
end process;
```

شبیه‌ساز پس از ۹۰۰ns متوقف می‌شود.

```
Process(out_model, in_model)
begin
if out_model=1 and in_model=1 then
        assert false
        report "The signals out and in are '1' at the same time"
        severity Error;
end process;
```

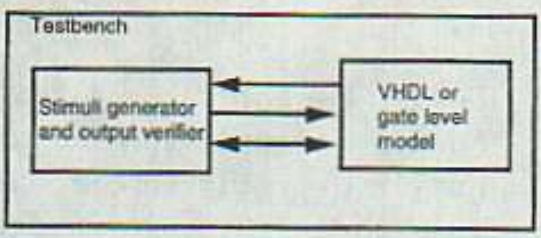
چنان‌چه سیگنال‌های ورودی و خروجی مشابه باشند، شبیه‌ساز کار خود را متوقف می‌کند و پیغام خطا را صادر خواهد کرد. در مثال بالا از دستور if برای تعیین شرط عبارت assert استفاده شده است. در این مثال شرط عبارت assert همواره نادرست خواهد بود ('assert false').

همچنین می‌توان از یک سیگنال خروجی و غیرفعال کردن آن به هنگام عملکرد ناصحیح مدل استفاده کرد.

```
process(out_model, in_model)
begin
   if out_model = 1 and in_model = 1 then
        test_ok<= '0';
   end process;
end process;
```

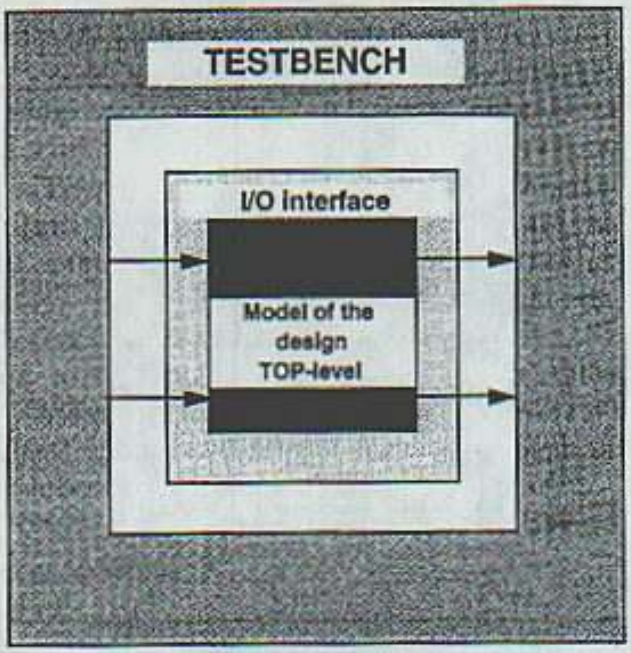
تکنیک استفاده از دستور wait until clk برای بررسی سیگنال‌های خروجی یک «ماشین حالت» می‌تواند بسیار مفید باشد. محیط آزمایش باید به گونه‌ای نوشته شود که بتوان از آن در سطح گیتی نیز استفاده کرد. در سطح گیتی سیگنال‌های خروجی بلافاصله پس از تغییر لبه پالس ساعت تغییر نمی‌کنند و این امر در شبیه‌سازی VHDL. یک رویداد طبیعی است. این موضوع باید در محیط آزمایش پیش‌بینی شود. یک روش ساده در این خصوص به کارگیری دستور ;'0' = wait until clk است. در این حالت سیگنال‌های خروجی باید به حالت پایدار رسیده باشند که معمولاً نصف دوره تناوب پالس ساعت انتظار لازم است. بررسی سیگنال‌های خروجی می‌تواند پس از تغییر لبه پالس ساعت یا هم‌زمان با آن انجام شود.

مدل طرح می‌تواند به صورت یک کد VHDL، یک شبکه و یا با هر فرمت دیگری نوشته شود، به آن معنا که محیط آزمایش یکسانی را می‌توان در حین کار در فازهای گوناگون طراحی به کار گرفت (شکل ۲-۸). مدل طرح توسط محیط آزمایش بررسی می‌شود. اما این کار زمانی انجام می‌شود که یک مدل رفتاری از طرح و علاوه بر آن یک مدل در سطح رجیستری از طرح آماده باشد. پس از آماده شدن مدل، شبکه بررسی می‌شود. در نهایت تحلیل زمانی مدل انجام می‌گیرد.



شکل ۲-۸ پیکره‌بندی‌های گوناگون محیط آزمایش

با تغییر مدل، واسطه‌های بین ورودی ـ خروجی با محیط آزمایش نیز باید تغییر کنند (شکل ۳-۸). کار با انواع داده‌ها در سطح رفتاری آسان‌تر از سطوح پایین‌تر مثل سطح گیتی خواهد بود. این موضوع باعث ایجاد واسطه‌های گوناگون بین محیط آزمایش و مدل طرح خواهد شد. یک مثال، کار با اعداد اعشار در سطح رفتاری در مقایسه با ترجمه عدد اعشار به تعدادی بردار در سطوح پایین‌تر است.



شکل ۳-۸ واسط ورودی ـ خروجی در محیط آزمایش

مسلماً داشتن یک واسط واحد در کلیه مدلها بسیار ساده خواهد بود. این کار با به کارگیری بردارها در entity برنامه در سطح رفتاری و استفاده از انواع بیشتر داده‌ها در معماری برنامه انجام‌پذیر خواهد شد. یک مزیت این روش، امکان جابه‌جایی و پرش بین مدلهای گوناگون است. تنها کاری که باید انجام شود تغییر مشخصات جزء ترکیبی در محیط آزمایش است.

مثال :

```
Behavioural level:   For U1: Use entity work.my_comp(behv);
RTLVHDL model:       For U1: Use entity work.my_comp(rtl);
Gate level:          For U1: Use entity work.my_comp(gate);
```

۱-۸ سطوح گوناگون محیط آزمایش

محیطهای آزمایش را می‌توان به سه «کلاس» متفاوت تقسیم کرد. کلاس اول ساده‌ترین و کلاس سوم مشکل‌ترین آنهاست (جدول ۱-۸ را ببینید). تفاوت آنها در چک کردن زمان و ارزش سیگنال‌های خروجی است. وظیفه کلیه محیطهای آزمایش، تولید سیگنال‌های ورودی برای مدلی است که تحت آزمایش قرار دارد.

	محرک ورودی	بررسی سیگنال خروجی	بررسی زمانی
کلاس ۱	توسط محیط آزمایش	-	-
کلاس ۲	توسط محیط آزمایش	توسط محیط آزمایش	-
کلاس ۳	توسط محیط آزمایش	توسط محیط آزمایش	توسط محیط آزمایش

جدول ۱-۸ انواع گوناگون محیطهای آزمایش

در یک محیط معمولی آزمایش، جزء ترکیبی مورد تست با VHDL ساختاری فراخوانی می‌شود. از آنجایی که محیط آزمایش محرک ورودی را برای جزء ترکیبی تولید می‌کند بنابراین نیازی به تعریف سیگنال ورودی در بخش اعلام محیط آزمایش وجود ندارد. فرض کنید جزء ترکیبی زیر را برای تست انتخاب کرده‌ایم.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity my_comp is
port (clk,resetn, d_in:      in  std_logic;
      a,b:                   in  std_logic_vector(2 downto 0);
      d_out,en,overflow:     out std_logic;
      q:                     out std_logic_vector(2 downto 0));
end;

Architecture rtl of my_comp is
begin
  Controller: Block
  type state_type is (s0,s1,s2);
  signal state:state_type;
```

```
begin
    process(clk,resetn)
    begin
      if resetn= '0' then
          state<= s0;
          d_out<= '0';
          en<= '0';
      elsif clk'event and clk= '1' then
        case state is
          when s0 => if d_in= '1' then
                       state<=s1;
                     end if;
                     d_out<= '1';
                     en<= '0';
          when s1 => if d_in= '0' then
                       state<=s2;
                     end if;
                     d_out<= '1';
                     en<= '1';
          when s2 => state <= s0;
                     d_out<= '0';
                     en<= '0';
          when others => state <= s0;
                         d_out<= '1';
                         en<= '0';
        end case;
      end if;
    end process;
  end block;

  check: Block
  signal q_b: std_logic_vector (2 downto 0);
  begin
      q <= q_b;
      q_b <= a + b;
      overflow <= '1' when q_b > 2 else '0';
  end block;
end;
```

کلاس ۱

محیطهای آزمایش از نوع کلاس۱ تنها سیگنال‌های ورودی مدل را تولید می‌کنند. سیگنال‌های خروجی با کمک دست آزمایش و تأیید می‌شوند.

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity test_my_comp is
port (d_out,en,overflow: out std_logic;
      q:                 out std_logic_vector(2 downto 0));
end;

Architecture testbench of test_my_comp is
  component my_comp
  port ( clk,resetn,d_in: in  std_logic;
         a,b:             in  std_logic_vector(2 downto 0);
         d_out,en:        out std_logic;
         overflow:        out std_logic;
         q:               out std_logic_vector(2 downto 0));
  end component;
  signal clk:          std_logic:= '0';
  signal resetn:       std_logic:= '0';
  signal d_in:         std_logic;
  signal a,b:          std_logic_vector(2 downto 0);
  For U1:my_comp Use Entity work.my_comp(rtl);
begin
      U1:my_comp port map (clk, resetn, d_in,a,b,d_out,en,overflow,q);
      clk <= not clk after 50 ns;
      resetn <= '1' after 125 ns;

      a <=    "000",
              "010" after 125 ns,
              "100" after 175 ns;

      b <=    "000",
              "100" after 125 ns,
              "011" after 175 ns;
```

```
    process
      begin
        d_in <= '0';
        wait until resetn= '1';
          d_in <= '1';
        wait until clk= '1';
          d_in <= '0' after 10 ns;
        wait;                          -- Wait for end of simulation
      end process;
end;
```

محیط آزمایش کلاس ۱ که در بالا گفته شد، سیگنال‌های ورودی را برای جزء ترکیبی آماده می‌کند در حالی که سیگنال‌های خروجی باید به طور دستی بررسی شوند. در این مثال، سیگنال‌های خروجی، در بخش اعلام محیط آزمایش قرار داده شده‌اند، که این روش از نقطه‌نظر نگارش و خواناتر شدن برنامه مزایایی دارد و اگر این قاعده همواره رعایت شود تصمیم بر اینکه کدام یک از سیگنال‌های خروجی باید مورد آزمایش قرار گیرند بسیار آسان‌تر خواهد شد.

کلاس ۲

محیط آزمایش در کلاس ۲ نه تنها سیگنال‌های ورودی برای مدل را تولید می‌کند بلکه درستی ارزش سیگنال‌های خروجی را نیز بررسی می‌نماید. در این حالت نیز تحلیل زمانی باید به طور دستی انجام گیرد.

مثالی از محیط آزمایش کلاس ۲ می‌تواند به صورت زیر باشد :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Entity test2_my_comp is
   port (test_ok: out std_logic:= 'H');
end;

Architecture testbench of test2_my_comp is
   component my_comp
        port (clk,resetn,d_in:      in  std_logic;
              a,b:                  in  std_logic_vector(2 downto 0);
              d_out,en,overflow:    out std_logic;
              q:                    out std_logic_vector(2 downto 0));
   end component;
```

```
signal clk:                 std_logic:= '0';
signal resetn:              std_logic:= '0';
signal d_in:                std_logic;
signal a,b:                 std_logic_vector(2 downto 0);
signal d_out,en,overflow:   std_logic;
signal q:                   std_logic_vector(2 downto 0);

For U1:my_comp Use Entity work.my_comp(rtl);
begin
    U1:my_comp port map (clk, resetn,d_in,a,b,d_out,en,overflow,q);
    clk <= not clk after 50 ns;
    resetn <= '1' after 125 ns;

    a <=    "000",
            "010" after 125 ns,
            "100" after 175 ns;

    b <=    "000",
            "100" after 125 ns,
            "011" after 175 ns;

    p0: process
    begin
      wait for 125 ns;
      if q/= "000" or overflow/= '0' then
        test_ok<='0';
      end if;

      wait for 50 ns;
      if q/= "110" or overflow/='1' then
           test_ok<='0';
      end if;

      wait for 50 ns;
      if q/= "111" or overflow/='1' then
           test_ok<='0';
      end if;
      wait;
    end process;
```

```
process
begin
   d_in<= '0';
   wait until resetn= '1';
   d_in<='1';

   wait until clk='1';
   d_in<= '0' after 10 ns;
   wait;
end process;

p1: process
begin
   wait for 30 ns;
   if en/= '0' or d_out/= '1' then
        test_ok<= '0';
   end if;

   wait until resetn= '1';

   wait until clk= '1';
   if en/= '0' or d_out/= '1' then
        test_ok<= '0';
   end if;

   wait until clk= '1';

   wait until clk= '1';
   if en/= '1' or d_out/= '0' then
        test_ok<= '0';
   end if;

   wait until clk= '1';
   if en/= '0' or d_out/= '0' then
        test_ok<= '0';
   end if;
   wait;
 end process;
end;
```

محیط آزمایش کلاس ۲ مذکور سیگنال‌های خروجی را توسط سیگنال Test_ok بررسی می‌کند. این سیگنال در بخش اعلام به ازرش 'H' (یک ضعیف) مقداردهی شده است.

سیگنال Test_ok با نوع std_logic معرفی شده است. این نوع داده یک نوع ***تعیین شده***[1] است. به این معنا که اگر مقدارهای متعددی به سیگنال نسبت داده شوند، تابع تعیین‌کننده‌ای فراخوانی خواهد شد. این تابع تعیین می‌کند که چه مقداری باید به سیگنال داده شود. محیط آزمایش بالا ارزش 'H' را به سیگنال Test_ok می‌دهد، در ضمن به هنگام ایجاد خطا مقدار '0' به آن نسبت داده می‌شود. از آنجایی که این سیگنال چندین بار به طور همزمان مقداردهی می‌شود تابع تعیین‌کننده به کار می‌آید که در این صورت ارزش '0' به این سیگنال نسبت داده می‌شود. از طرفی دیگر چنان‌چه خطایی در جزء ترکیبی وجود نداشته باشد مقدار این سیگنال کماکان 'H' باقی خواهد ماند که به مفهوم بدون خطا بودن جزء ترکیبی است.

در مثال بالا سیگنال Test_ok به ارزش 'H' مقداردهی شده است نه به ارزش '1'، زیرا این سیگنال از دو محل و به طور همزمان مقداردهی خواهد شد یعنی پروسس‌های P0 و P1 . این بدان معناست که تابع تعیین‌کننده برای تعیین ارزش دقیق سیگنال به کار می‌آید. چنان‌چه یکی از پروسس‌ها خطایی پیدا کند، مقدار '0' به سیگنال Test_ok داده می‌شود، به این معنا که تابع تعیین‌کننده با دو مقدار 'H' و '0' فراخوانی می‌شود که در این حالت مقدار '0' مقدار غالب خواهد بود. اگر سیگنال Test_ok به ارزش '1' به جای 'H' مقداردهی اولیه می‌شد، آنگاه تابع تعیین‌کننده در هنگام رخ دادن خطا در برنامه، ارزش 'X' را به Test_ok می‌داد. اگر محیط آزمایش به گونه‌ای نوشته می‌شد که فقط یک دستور موازی (در این مورد فقط یک پروسس) سیگنال Test_ok را مقداردهی می‌کرد، می‌توانستیم از مقدار اولیه '1' به جای 'H' استفاده کنیم.

بررسی سیگنال‌های خروجی و d_out به طور همزمان با لبه پالس ساعت انجام می‌شود. نه مدلهای VHDL و نه طراحی در سطوح گیتی هیچ کدام قادر به نسبت دادن مقادیر جدید به خروجی‌های خود (***روزآمدسازی خروجی‌ها***[2]) نیستند، به عبارت دیگر سیگنال‌های خروجی کماکان ارزشهای گذشته و قدیمی خود را حفظ می‌کنند. خروجی‌های مدلهای VHDL پس از گذشت زمان یک دلتا و طراحی‌های در سطح گیتی پس از گذشت چندین نانوثانیه (بسته به تکنولوژی انتخاب شده) مقدار جدید را به سیگنال‌های خروجی خود می‌دهند.

به عنوان یک جایگزین به جای سیگنال Test_ok می‌توانیم از دستور assert برای اعلام خطا استفاده کنیم. در دستور assert امکان ایجاد و تنظیم شدت خطا وجود دارد که با کمک آن می‌توان

1- Resolved

2- Update the Outputs

تعیین کرد که شبیه‌ساز به هنگام مواجهه با خطا متوقف شود یا کارش را ادامه دهد. به عنوان حالت اولیه برای اغلب شبیه‌سازهای VHDL، سطح خطا در وضعیت 'توقف' قرار دارد. البته امکان ترکیب سیگنال خروجی Test_ok با دستور assert نیز وجود دارد. امتیاز این کار در این است که روند تست بخشهای مختلف تا انتها ادامه می‌یابد و در نهایت دستور assert بخشهای مختلف را که در آنها خطا رخ داده است اعلام می‌کند.

مثال :

```
if en/= '0' or d_out/= '0' then
        test_ok<= '0';
        assert false
        report "en/= '0' or d_out/= '0' "
        severity warning;
end if;
```

کلاس ۳

محیطهای آزمایش در این کلاس، سیگنال‌های ورودی مورد نیاز مدل را تولید کرده و سیگنال‌های خروجی را از نظر ارزش و زمان مورد بررسی قرار می‌دهند.

مثالی از محیط آزمایش کلاس ۳ می‌تواند به شکل زیر باشد :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity test3_my_comp is
   port (test_ok: out std_logic:= 'H');
end;

Architecture testbench of test3_my_comp is
   component my_comp
        port (clk,resetn,d_in:      in  std_logic;
              a,b:                  in  std_logic_vector(2 downto 0);
              d_out,en,overflow:    out std_logic;
              q:                    out std_logic_vector(2 downto 0));
   end component;
```

```
signal clk:                 std_logic:= '0';
signal resetn:              std_logic:= '0';
signal d_in:                std_logic;
signal a,b:                 std_logic_vector(2 downto 0);
signal d_out,en,overflow:   std_logic;
signal q:                   std_logic_vector(2 downto 0);

For U1:my_comp Use Entity work.my_comp(rtl);
begin
    U1:my_comp port map (clk, resetn,d_in,a,b,d_out,en,overflow,q);
    clk <= not clk after 50 ns;
    resetn <= '1' after 125 ns;

    a <=    "000",
            "010"after 125 ns,
            "100"after 175 ns;

    b <=    "000",
            "100"after 125 ns,
            "011"after 175 ns;

    process
    begin
      wait for 125 ns;
      if q/= "000" or not q'stable(15 ns) then
            test_ok<= '0';
      elsif overflow/= '0' or not overflow'stable(15 ns) then
            test_ok<='0';
      end if;

      wait for 50 ns;
      if q/= "110" or not  q'stable(15 ns) then
          test_ok<='0';
      elsif overflow/= '1' or not overflow'stable(15 ns) then
          test_ok<='0';
      end if;

      wait for 50 ns;
      if q/= "111" or not q'stable(15 ns) then
```

```
        test_ok<='0';
    elsif overflow/= '1' or not overflow'stable(15 ns) then
        test_ok<='0';
    end if;

    wait;
end process;
process
begin
    d_in<= '0';
    wait until resetn= '1';
    d_in<='1';
    wait until clk='1';
    d_in<= '0' after 10 ns;

    wait;
end process;

process
begin
    wait for 30 ns;
    if en/= '0' or d_out/= '1' then
        test_ok<= '0';
    end if;
    wait until resetn= '1';
    wait until clk= '1';
    if en/= '0' or d_out/= '1' then
        test_ok<= '0';
    elsif not en'stable(80 ns) or not d_out'stable(80 ns) then
        test_ok<= '0';
    end if;

    wait until clk= '1';
    wait until clk= '1';
    if en/= '1' or d_out/= '0' then
        test_ok<= '0';
    elsif not en'stable(80 ns) or not d_out'stable(80 ns) then
        test_ok<= '0';
    end if;
```

```
        wait until clk= '1';
        if en/= '0' or d_out/= '0' then
            test_ok<= '0';
        elsif not en'stable(80 ns) or not d_out'stable(80 ns) then
            test_ok<= '0';
        end if;

        wait;
    end process;
end;
```

محیط آزمایش کلاس ۳ فوق پایداری سیگنال‌های خروجی را بـه ترتیب برای مـدت زمـانهـای ۱۵ns و ۸۰ns بررسی می‌کند. جمع q ⇐ a + b توسط دستور q 'stable (15) بعد از ۵۰ns چک می‌شود، بنابراین عمل جمع باید حداکثر در طول مدت ۳۵ = ۱۵ - ۵۰ نانوثانیه انجام شود. در لبه بالارونده پالس ساعت سیگنال‌های خروجی en و d_out چک می‌شوند که آیا برای مدت زمان ۸۰ns پایدار هستند یا خیر. این سیگنال‌ها برای انتقال از فلیپ فلاپ داخلی تا سیگنال خروجی مدار مدت زمانی برابر ۲۰ = ۸۰ - ۱۰۰ نانوثانیه فرصت دارند. این محدودیت زمانی بدون هیچ مشکلی در شبیه‌سازی کد VHDL برقرار خواهد شد. تنها در شبیه‌سازی در سطح گیتی است که ممکن است خطای زمانی رخ دهد.

Pull up/down ۸-۲

چنان‌چه سیگنال ورودی یا خروجی یک جزء ترکیبی نیاز به pull up یا pull down داشته باشد می‌توان این دو مفهوم را به یکی از دو روش زیر توصیف کرد :

- در کد VHDL جزء ترکیبی
- در کد VHDL محیط آزمایش

فرض کنید جزء ترکیبی زیر نیاز به pull up دو جهته در پایه خروجی io داشته باشد.

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity my_comp is
port (a,b: in    std_logic;
      io:  inout std_logic;
      q:   out   std_logic);
end;
```

```
Architecture rtl of my_comp is
begin
        q<=a and io;
        io<= '1' when a= '1' and b= '0' else 'Z';
end;
```

مدل‌سازی این حالت با مقداردهی سیگنال دو جهته io با ارزش 'H' انجام می‌گیرد. سیگنال باید از نوع std_logic باشد. همان طور که قبلاً نیز گفته شد std_logic یک نوع داده «تعیین شده» است. بنابراین اگر این سیگنال به طور پیوسته مقدار 'H' را دارا باشد و بعضی اوقات مقادیر دیگری را مثل '0' یا '1' بگیرد، «تابع تعیین‌کننده» فراخوانی خواهد شد. چنان‌چه به سیگنال دوجهته io توسط محیط آزمایش و یا از داخل مدار، ارزش '0' داده شود، در نهایت تابع تعیین‌کننده مقدار '0' را به این سیگنال خواهد داد، که البته این عمل با واقعیت هماهنگی دارد.

اگر بخواهیم شرح مفهوم pull up را در داخل جزء ترکیبی وارد کنیم، معماری برنامه به شکل زیر درخواهد آمد :

```
Architecture rtl of my_comp is
begin
        io<= 'H';
        q<=a and io;
        io<= '1' when a= '1' and b= '0' else 'Z';
end;
```

و اگر بخواهیم تنها ارزش اولیه‌ای به سیگنال موردنظر بدهیم وضعیت زیر را خواهد داشت :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity my_comp is
port (a,b: in      std_logic;
      io: inout    std_logic:='H';
      q: out       std_logic);
end;
```

دقت شود که به محض نسبت دادن یک مقدار معین به سیگنال (در معماری برنامه)، ارزش اولیه‌ای که به سیگنال داده شده بود از بین خواهد رفت و سیگنال مقدار جدید را به خود می‌گیرد.

ابزارهای سنتز مقدار اولیه 'H' و مفهوم pull up را در نظر نمی‌گیرند. زمانی باید pull up اعمال شود که مشخص شده باشد جزء ترکیبی چه نوع بلوک I/O یی را خواهد داشت. در شبیه‌ساز VHDL چنان‌چه سیگنال دیگری (داخلی یا خارجی) به درگاه io متصل نباشد ارزش صحیح 'H' به آن داده خواهد شد.

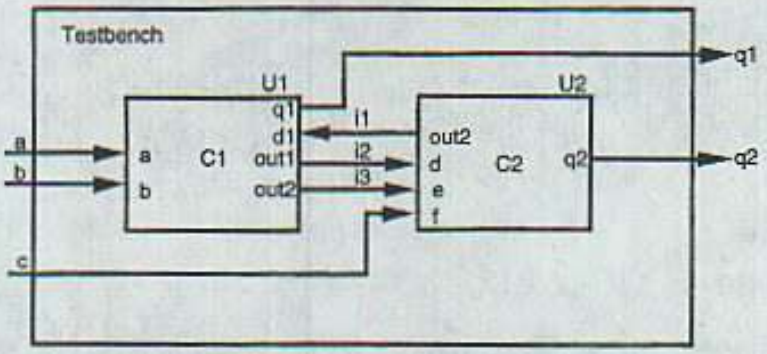
به عنوان یک راه‌حل دیگر، می‌توانیم pull up را در محیط آزمایش اعمال کنیم. در زیر یک مثال ساده از محیط آزمایش کلاس ۱ آورده شده که در آن pull up اعمال شده است :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity tb is
port ( io: inout  std_logic;
       q:  out    std_logic);
end;

Architecture testbench of tb is
begin
   io<= 'H';                    -- Pull up
   a<= '0',
       '1' after 100 ns,
       '0' after 50 ns;

   b<= '0',
       '1' after 150 ns,
       '0' after 50 ns;

   io<= '0',
        'Z' after 100 ns,       -- io internally driven
        '0' after 50 ns;
end;
```

۳-۸ چندین جزء ترکیبی در یک محیط آزمایش

اگر چندین جزء ترکیبی در یک طرح با هم مرتبط باشند، می‌توان همه آنها را در یک محیط آزمایش مورد بررسی قرار داد. در این مورد تعدادی از سیگنال‌های ورودی توسط اجزای ترکیبی و تعدادی دیگر توسط برنامه تست کنترل می‌شوند. در این شکل از آزمایش باید ترکیبی از شبیه‌سازی

سیستم و شبیه‌سازی محیط آزمایش را به کار گرفت. فرض کنید که در طرح، یک ارتباط منطقی بین دو جزء ترکیبی مطابق شکل ۴-۸ داشته باشیم.



شکل ۴-۸ دو جزء ترکیبی در یک محیط آزمایش

بنابراین برنامه آزمایش را می‌توان به شکل زیر نوشت :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity tb2comp is
   port ( q1,q2:  out  std_logic);
end;

Architecture testbench of tb2comp is
   component c1
        port (a,b,d1:                  in   std_logic;
              q1, out1, out2:          out  std_logic);
   end component;

   component c2
        port (d,e,f:                   in    std_logic;
              q2,out2:                 out   std_logic);
   end component;

   For U1:c1 Use entity work.c1(rtl);
   For U1:c2 Use entity work.c2(rtl);

   signal  a,b,c,i1,i2,i3: std_logic;
   begin
```

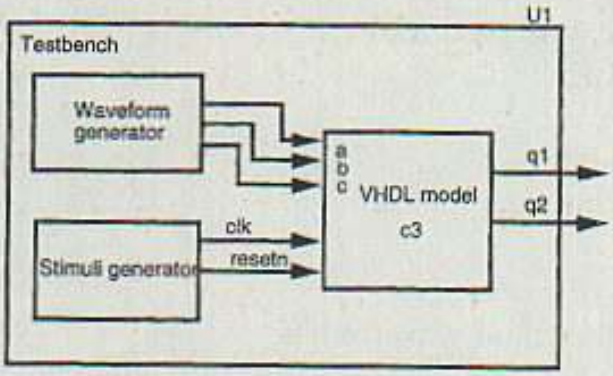
```
U1:c1 port map (a,b,i1,q1,i2,i3);
U2:c2 port map (i2,i3,c,q2,i1);
a<=     '1',
        '0' after 50 ns,
        '1' after 125 ns;
b<=     '0',
        '1' after 70 ns,
        '0' after 125 ns,
        '1' after 225 ns;
c<=     '1',
        '0' after 50 ns,
        '1' after 150 ns,
        '0' after 250 ns;
end;
```

٤-٨ مولدهای شکل موج[1]

روش دیگر در ایجاد محرکهای ورودی محیط آزمایش، تولید شکل موجهای دلخواه است. در این روش از یک ROM که شامل اطلاعاتی درباره ارزش و مقدار محرکها است، استفاده می‌شود.

در فصل گذشته چگونگی توصیف ROM توسط آرایه‌ها توضیح داده شد. توصیه می‌شود که معرفی ROM و اعلام محتویات آن در یک بسته انجام گیرد.

فرض کنید که محیط آزمایش شکل ۵-۸ را در اختیار داشته باشیم :



شکل ۵-۸ محیط آزمایش با به کارگیری مولد شکل موج

1- Waveform Generator

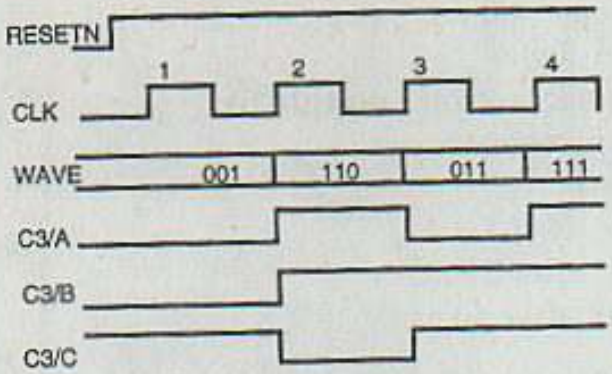
در زیر مثالی از یک محیط آزمایش را آورده‌ایم که در آن از یک مولد شکل موج برای تولید سیگنال‌های محرک جزء ترکیبی c3 استفاده شده است :

```
Library ieee;
Use ieee.std_logic_1146.ALL;

Package rom is
   constant  rom_width:integer:=3;
   constant  rom_length:integer:=10;
   subtype rom_word is std_logic_vector(rom_width-1 downto 0);
   type rom_table is array (0 to rom_length-1) of rom_word;
   constant rom: rom_table:= rom_table'("001",
                                        "110",
                                        "011",
                                        "111",
                                        "101",
                                        "000",
                                        "101",
                                        "001",
                                        "101",
                                        "010");
end;

Library ieee;
Use ieee.std_logic_1146.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.rom.ALL;

Entity waveform is
   port ( q1,q2:   out  std_logic);
end;

Architecture testbench of waveform is
   component c3
        port (clk,resetn,a,b,c:  in   std_logic;
              q1,q2:             out std_logic);
   end component;

   For U1:c3 Use entity work.c3(rtl);
```

```
signal clk: std_logic:= '0';
signal resetn: std_logic;
signal step:std_logic_vector(3 downto 0);
signal waves:rom_word;
begin
      resetn<=  '0',
                '1' after 25 ns;
      clk<=not clk after 50 ns;

      process(clk,resetn)
      begin
        if resetn= '0' then
             step<=(others=> '0');              -- Restart the waveform
        elsif clk'event and clk= '1' then
             if step/=rom_length-1 then
                  step<=step+1;                 -- Increment the waveform
             else
                  step<=(others=> '0');         -- Restart the waveform
             end if;
        end if;
      end process;
      waves<=rom(conv_integer (step));  -- Read waveform value from
                                        -- ROM
U1:c3 port map (clk,resetn,waves (2), waves (1), waves(0), q1,q2);
end;
```

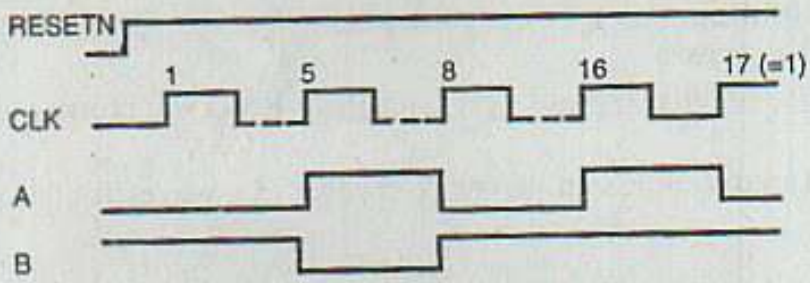
الگوی آزمایش نشان داده شده در شکل ۶-۸ از شبیه‌سازی مولد شکل موج مثال قبل پس از گذشت چهار پالس ساعت به وجود آمده است.

از آنجایی که مولد شکل موج بر پایه سنتز ROM استوار است، می‌توان از این تکنیک در طراحی‌های مختلف نیز استفاده کرد. این روش در مقایسه با روش «ماشین حالت» برای تولید سیگنال‌های خروجی متوالی از لحاظ مصرف حجم و فضا بسیار بهینه است.



شکل ۶-۸ شکل موجها

اگر مقدار سیگنال‌های ورودی به سیستم از مولد شکل موج تغییر نکند، تعدادی از گیت‌ها بلااستفاده می‌مانند. از نظر طراحی و همچنین از نظر محیطهای آزمایش، بهتر است که فقط مقادیر متغیر ROM و همچنین تعداد سیکل‌هایی از پالس ساعت که این مقادیر پایدار می‌مانند را مشخص کنیم. در این حالت نیز از مدل ROM بهره می‌گیریم. فرض کنید شکل موج دو سیگنال خروجی A و B باید به شکل ۷-۸ باشد، که باید هر ۱۶ پالس ساعت تکرار گردد.



شکل ۷-۸ شکل موجها

چنان‌چه از روش قدیمی تولید شکل موج استفاده کنیم، برنامه به صورت زیر خواهد شد :

```
Library ieee;
Use ieee.std_logic_1146.ALL;

Package rom is
   constant rom_width:integer:=2;
   constant rom_length:integer:=16;
   subtype rom_word is std_logic_vector(rom_width-1 downto 0);
   type rom_table is array (0 to rom_length-1) of rom_word;
```

```
  constant rom: rom_table:= rom_table '("01",
                                        "01",
                                        "01",
                                        "01",
                                        "10",
                                        "10",
                                        "10",
                                        "01",
                                        "01",
                                        "01",
                                        "01",
                                        "01",
                                        "01",
                                        "01",
                                        "01",
                                        "01",
                                        "11");
end;

Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.rom.ALL;

Entity waveform is
port (clk,resetn:  in   std_logic;
      waves:       out  rom_word);
end;

Architecture testbench of waveform is
signal start_up:std_logic;
signal step:std_logic_vector(3 downto 0);
begin
  process(clk,resetn)
  begin
       if resetn= '0' then
              step<=(others=> '0');       -- Restart the waveform
              start_up<= '1';
       elsif clk'event and clk= '1' then
              start_up<= '0';
```

```
            if start_up= '0' then          -- Delay one clock cycle after reset
              if step/=rom_length-1 then
                  step<=step+1;        -- Increment the waveform
              else
                  step<=(others=> '0');    -- Restart the waveform
              end if;
            end if;
        end if;
    end process;
    waves<=rom(conv_integer(step));        -- Read waveform from ROM
end;
```

اگر از روش جدید تولید شکل موج استفاده شود، موارد زیر باید در برنامه منظور گردد :

- مقدار سیگنال‌های خروجی
- تعداد سیکلهای پالس ساعت که به ازای آنها سیگنال‌های خروجی پایدار می‌مانند.

مثال :

```
Library ieee;
Use ieee.std_logic_1146.ALL;

Package rom is
   constant rom_width:integer:=2;
   constant rom_length:integer:=4;
   subtype rom_word is std_logic_vector(rom_width-1 downto 0);
   type rom_table is array (0 to rom_length-1) of rom_word;
   constant rom: rom_table:= rom_table ' ("01",
                                          "10",
                                          "01",
                                          "11");
   subtype max_cycle is integer range 0 to 15;
   type cycle_length_table is array (0 to rom_length-1)
        of max_cycle;
   constant cycle_length: cycle_length_table:=cycle_length_table'(4,3,8,1);
end;

Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
```

```
Use work.rom.ALL;

Entity wave_adv is
port (clk,resetn:  in   std_logic;
      waves:       out rom_word);
end;

Architecture testbench of wave_adv is
signal step:std_logic_vector(3 downto 0);
begin
  process(clk,resetn)
  variable count:max_cycle;
  begin
      if resetn= '0' then
            step<=(0=> '1', others=> '0');    -- step=1
            count:=cycle_length(0);
      elsif clk'event and clk= '1' then
            if count=0 then
              count:=cycle_length(conv_integer(step));
              if step/=rom_length-1 then
                  step<=step+1;               -- Increment the waveform
              else
                  step<=(others=> '0');       -- Restart the waveform
              end if;
            end if;
            count:=count-1;
      end if;
  end process;
  waves<=rom(conv_integer(step));               -- Read waveform from ROM
end;
```

در بسته مثال فوق حافظه ROM تنها شامل خانه‌های ۲ × ۴ بیتی به جای ۲ × ۱۶ بیتی (مطابق مثال قبلی) است. تعداد سیکل‌های پالس ساعت که مدت زمان پایدار ماندن سیگنال‌های خروجی مولد را مشخص می‌کند به طور جداگانه در ROM دیگری تعیین شده‌اند.

```
constant rom: rom_table:= rom_table' ("01",
                                      "10",
                                      "01",
                                      "11");
```

cycle_length:cycle_length_table:=cycle_length_table'(4,3,8,1);

مقادیر ثابت فوق می‌توانند به صورت زیر تفسیر شوند :
مقدار "01" برای مدت ۴ پالس ساعت در خروجی قرار می‌گیرد.
مقدار "10" برای مدت ۳ پالس ساعت در خروجی قرار می‌گیرد.
مقدار "01" برای مدت ۸ پالس ساعت در خروجی قرار می‌گیرد.
مقدار "11" برای مدت ۱ پالس ساعت در خروجی قرار می‌گیرد.

اگر پس از سنتز این دو مثال کوچک، حجم و فضای مصرف شده توسط هر کدام را مقایسه کنیم به این نتیجه می‌رسیم که در این مورد مولد شکل موج قدیمی (اولیه) فضای کمتری اشغال می‌کند. اما چنان‌چه تعداد سیگنال‌ها از دو بیشتر و توالی شکل موجها طولانی‌تر شود نتیجه معکوس خواهد بود. انتخاب مولد شکل موج باید بر اساس شکل موجی که شما درصدد تولید آن هستید صورت گیرد.

TextIO ۸-۵

دستورهای TextIO که توسط آنها فایل‌ها می‌توانند خوانده یا نوشته شوند در استاندارد VHDL منظور شده است. این دستورها در زیر ذکر شده‌اند :

read (...)
readline (...)
write (...)
writeline (...)

چنان‌چه بخواهیم از این دستورها استفاده کنیم باید خط زیر را در ابتدای کد VHDL بنویسیم :

Use std.textio.ALL;

یک فضای کاربردی textIO محیطهای آزمایش است. به این ترتیب می‌توان سیگنال‌های ورودی مدل را در یک فایل خارجی قرار داد و همچنین می‌توان سیگنال‌های خروجی موردنظر مدل را نیز در یک فایل دیگر ذخیره نمود.

فرض کنید که مقادیر سیگنال‌های محرک مدل در فایل my_inputs.vec تعیین شده باشد. محیط آزمایش باید بتواند این فایل را در ارتباط با سیگنال‌های ورودی مدل (d1 و d2 و d3) از جزء ترکیبی c4 قرار دهد :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_arith.ALL;
Use std.textio.ALL;

Entity text_io is
  port ( q1,q2:  out  std_logic_vector(2 downto 0));
end;

Architecture testbench of text_io is
  component c4
      port (clk,resetn:  in   std_logic;
          d1,d2,d3: in   std_logic_vector(3 downto 0);
          q1,q2:    out  std_logic_vector(2 downto 0));
  end component;
  signal clk:            std_logic:= '0';
  signal resetn:         std_logic:= '0';
  signal d1,d2,d3:       std_logic_vector(3 downto 0);

  For U1:c4 Use Entity work.c4 (rtl);

  begin
  U1:c4 port map (clk,resetn,d1,d2,d3,q1,q2);

  clk<=not clk after 50 ns;
  resetn<= '1' after 125 ns;

  process
  variable text_line:line;      -- Stores a complete line
  variable i1:integer;          -- Used for instimuli for signal d1
  variable i2:integer;          -- Used for instimuli for signal d2
  variable i3:integer;          -- Used for instimuli for signal d3

  file my_file: text is in "my_inputs.vec";   -- File where the input
  begin                                       -- stimuli are stored

    while not endfile(my_file) loop
      readline(my_file,text_line);            -- Read one line from file
                                              -- my_inputs.vec into text_line
```

```
        read (text_line,i1);                    -- Read the first value into i1
        d1<=conv_std_logic_vector(i1, d1'length);

        read(text_line,i2);                     -- Read the second value into i2
        d2<=conv_std_logic_vector(i2,d2'length);

        read(text_line,i3);
        d3<=conv_std_logic_vector(i3,d3'length);

        wait until clk= '1';
    end loop;
    wait;
  end process;
end;
```

فرض کنید که فایل my_inputs.vec دارای مقادیر زیر باشد :

۸	۶	۲
۲	۷	۵
۹	۱	۱۲

my_input.vec

بنابراین محیط آزمایش مقادیر زیر را به سیگنال‌های d1 و d2 و d3 می‌دهد :

مقدار	سیگنال	زمان
"0011"	d1	0
"0110"	d2	0
"1000"	d3	0
"0101"	d1	50
"0111"	d2	50
"0010"	d3	50
"1100"	d1	150
"0001"	d2	150
"1001"	d3	150

متأسفانه در این مورد، استاندارد VHDL-93 با نسخه قدیمی‌تر خود یعنی VHDL-87 هماهنگ نبوده و در مورد مثال فوق چنان‌چه از VHDL-87 استفاده شود مشکلی وجود نخواهد داشت

ولی اگر از نسخه VHDL-93 استفاده می‌شود باید تغییرات زیر در برنامه اعمال گردد :

```
file my_file : text is in "my_inputs.vec";          -- VHDL-87
file my_file : text open read_mode is "my_inputs.vec "; -- VHDL-93
```

در VHDL-93، عبارت read-mode به عنوان حالت اولیه دستور است و نیاز به آوردن این عبارت در متن برنامه نیست، بنابراین می‌توان به طور خلاصه خط زیر را جایگزین خط قبلی کرد :

```
file my_file : text is "my_inputs.vec";          -- VHDL-93
```

تفاوت دیگر نسخه‌های استاندارد 1987 و 1993 در این است که یک فایل در هر دو استاندارد می‌تواند در یک بار شبیه‌سازی هم خوانده و هم نوشته شود اما در استاندارد VHDL-93 نمی‌توان این کار را به طور همزمان انجام داد.

این روش به کارگیری TextIO در محیطهای آزمایش بسیار انعطاف‌پذیر و تا حدودی آسان است. عیب این روش آن است که فراخوانی TextIO نسبت به کدهای معمولی VHDL کندتر انجام می‌گیرد.

اشاره به این نکته بسیار حائز اهمیت است که فراخوانی TextIO فقط در شبیه‌سازی کاربرد دارد و نمی‌توان در سنتز از آن استفاده کرد.

۶-۸ تمرین

۱- بخشهای مختلف یک محیط آزمایش را بنویسید.

۲- سه کلاس مختلف محیط آزمایش و تفاوت بین هر یک را بنویسید.

۳- (i) مولد شکل موج در VHDL چیست ؟

(ii) تفاوت بین یک مولد شکل موج ساده و نوع پیشرفته آن چیست ؟

۴- (i) TextIO چیست ؟

(ii) چگونه از TextIO و محیطهای آزمایش استفاده می‌شود ؟

۵- یک محیط آزمایش برای تست جزء ترکیبی زیر طراحی کنید.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity my_comp is
port (clk,resetn,a,b:    in  std_logic;
      c,d:               in  std_logic_vector(2 downto 0);
      q1,q2:             out std_logic;
      q3:                out std_logic_vector(5 downto 0));
end;

Architecture rtl of my_comp is
type state_type is (s0,s1,s2);
signal state:state_type;
signal q3_b: std_logic_vector(5 downto 0);
  q3<=q_b;
  q3_b<=c * d;
  process (clk,resetn)
  begin
      if resetn= '0' then
            state<=s0;
            q1<= '0';
            q2<= '1';
      elsif clk'event and clk= '1' then
            case state is
              when s0 => if a= '1' then
```

```
                    state<=s1;
                end if;
                q1<= '1';
                q2<= '0';
            when s1 => if b= '0' then
                    state <=s2;
                end if;
                q1<= '0';
                q2<= '1';
            when s2=> state<=s0;
                q1<= '0';
                q2<= '0';
            when others=> state<=s0;
                    q1<= '1';
                    q2<= '0';
        end case;
    end if;
  end process;
end;
```

(i) محیط آزمایش کلاس ۱ (فقط تولید محرک برای سیگنال‌های ورودی).

(ii) محیط آزمایش کلاس ۲

(iii) محیط آزمایش کلاس ۳ (سیگنال‌های خروجی باید ۳۰ نانوثانیه بعد از هر پالس ساعت پایدار باشند).

(iv) محیط آزمایش با یک مولد شکل موج ساده (فقط تولید محرک برای سیگنال‌های‌ورودی).

(v) محیط آزمایش با یک مولد شکل موج پیشرفته (فقط تولید محرک برای سیگنال‌های ورودی).

(vi) محیط آزمایش با TextIO (فقط تولید محرک برای سیگنال‌های ورودی).

فصل نهم

ماشینهای حالت

از ***ماشینهای حالت***[1] می‌توان به عنوان یکی از ابزارهای مؤثر کنترل استفاده کرد. به طور مثال ساختار ***ون نیومن***[2] (CPU) به عملیاتی مانند فازهای ***واکشی***[3] و اجرا، مسیرهای داده، رجیسترهای ALU و غیره احتیاج دارد. عملکرد این ماشین حالت بسیار عظیم‌تر از عملکرد یک CPU است زیرا کد رفتاری ماشین حالت در شبکه‌ای از گیت‌ها ذخیره می‌شود که می‌تواند با معادلات بولی مقایسه گردد. در ضمن حالتهای مختلف یک ماشین حالت در تعدادی فلیپ فلاپ ذخیره می‌شوند.

کد زیر می‌تواند هم در یک CPU اجرا شود و هم در یک ماشین حالت که در فضای VHDL نوشته شده باشد.

```
if a>37 and c<7 then
        state <= alarm;
        out_a <= '0';
        out_b <= '0';
        out_analog <= a+b;
else
        state <= running;
end if;
```

1- State Machine

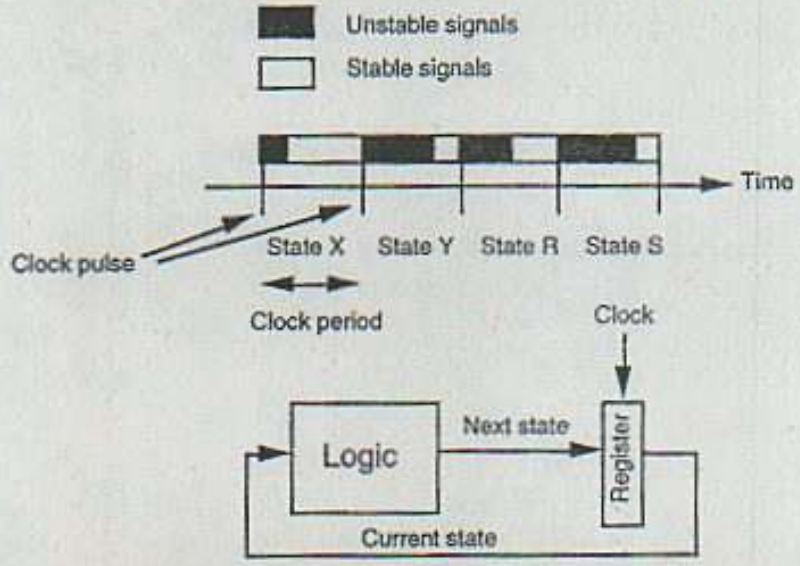
2- Von Neumann

3- Fetch

چنان‌چه این برنامه در یک CPU اجرا شود، به حدود ۱۰ تا ۲۰ دستورالعمل ترجمه می‌شود. اجرای این برنامه بسته به اینکه اسمبلر چه مسیری را در دستور «if» انتخاب کند نیاز به تعداد دستورالعملهای متنوعی دارد. این بدان مفهوم است که نمی‌توان زمان دقیق اجرای این برنامه را تعیین کرد و به جای آن باید یک زمان حداکثر و حداقل را در نظر گرفت. اگر همین برنامه با گیت‌ها و فلیپ فلاپ‌ها پیاده‌سازی شود در فاصله زمانی یک پالس ساعت قابل اجرا می‌گردد. نتیجه آنکه تعیین نحوه اجرا و مدت زمان عمل در ماشینهای حالت که با گیت‌ها و فلیپ فلاپ‌ها به اجرا در می‌آید خیلی بهتر از اجرای آن در CPU است.

ماشین حالت در دو فاز مجزا کار می‌کند. در فاز اول حالت جدید محاسبه می‌شود و در فاز دوم از مقدار حالت جدید نمونه‌برداری شده و در یک رجیستر قرار داده می‌شود. آنچه حد بالای فرکانس پالس ساعت ماشین حالت را تعیین می‌کند در اصل حداکثر زمان لازم برای تعیین حالت بعدی است. در مثال قبل نشان دادیم که شرایط " a > 37 و c < 7" معین می‌سازند که در نوبت بعد باید حالت "alarm" و یا "running" باشد.

این منطق در شکل ۱-۹ به طور گرافیکی نشان داده شده است. در هر خط از مقدار جدید نمونه‌برداری شده و داخل رجیستر قرار می‌گیرد. این بدان معنی است که ممکن است سیگنال‌های داخلی در فواصل بین این نمونه برداری‌ها ناپایدار شوند. علت اینکه این سیگنال‌ها ناپایدار می‌شوند آن است که سیگنال‌های ورودی به بلوک منطقی بعد از هر پالس ساعت تغییر می‌کنند، این در حالی است که مدت زمان خاصی برای set کردن گیت‌ها موردنیاز است. سیگنال‌های ماشین حالت به هنگام رسیدن پالس ساعت بعدی باید کاملا پایدار باشند.



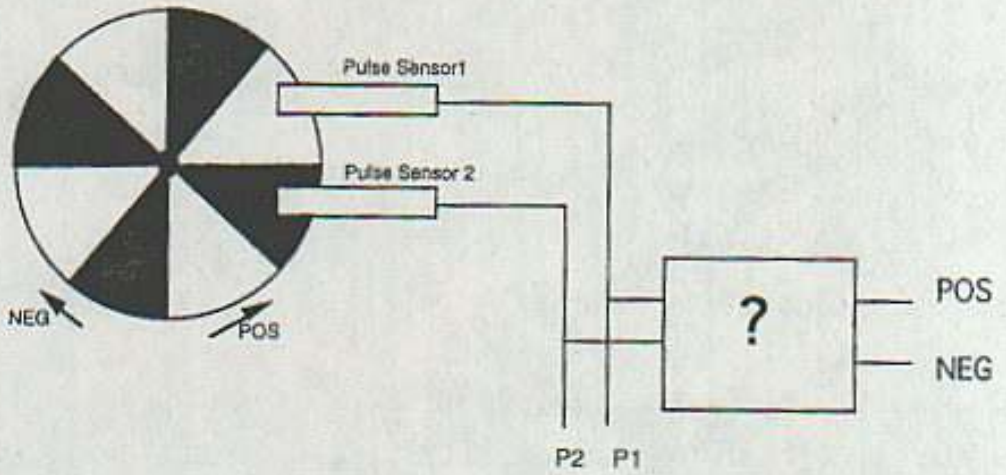
شکل ۱-۹ ماشین حالت

ماشینهای حالت دارای دو نوع اساسی هستند : ***موور***[1] و ***میلی***[2]. تفاوت بین این دو به خروجی‌های آنها مربوط است.

- **ماشین میلی:** خروجی‌های ماشین میلی تابعی از حالت (وضعیت) فعلی ماشین و کلیه ورودیهای آن هستند.
- **ماشین موور:** خروجی‌های ماشین موور تابعی از فقط حالت (وضعیت) فعلی ماشین می‌باشند.

به علاوه، یک ماشین میلی همواره به اندازه یک پالس ساعت جلوتر از یک ماشین موور کار می‌کند، زیرا که خروجی‌های یک ماشین میلی بلافاصله پس از تغییر ورودی‌ها تغییر می‌کنند، در حالیکه در ماشین موور ابتدا باید حالت تغییر کند و پس از یک پالس ساعت خروجی‌ها تغییر می‌کنند.

مثال زیر چگونگی عملکرد یک ماشین حالت را نشان می‌دهد. هدف از این مثال تعیین جهت چرخش (POS و NEG) یک موتور است. تشخیص جهت توسط دو سنسور پالسی انجام می‌شود. هنگامی‌که پس‌زمینه روشن باشد، خروجی سنسور '1' و چنان‌چه تیره باشد، '0' خواهد بود. به منظور ترجمه این مثال به گیت‌ها (از نوع NAND) و همچنین انجام عمل ساده‌سازی و بهینه‌سازی، از ماشین حالت و جدول کارنو استفاده شده است. ابزار سنجش و چگونگی انجام آزمایش در شکل ۲-۹ نشان داده شده است.

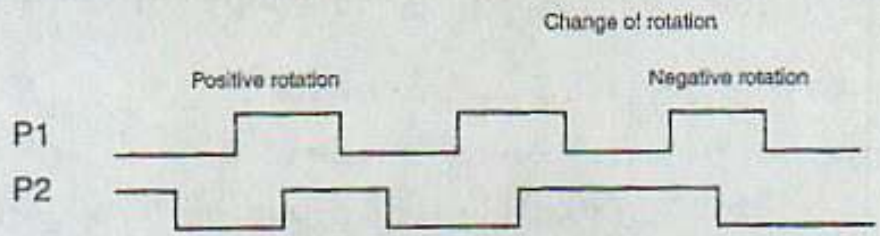


شکل ۲-۹ دیاگرام شماتیکی آزمایش

1- Moore

2- Mealy

سنسور پالسی دو پالس با شیفت فاز ۹۰ درجه تولید می‌کند، سیگنال‌های P1 و P2 جهت چرخش بسته به اینکه کدام پالس زودتر رسیده است مثبت (NEG=0 و POS = 1) یا منفی (NEG = 1 و POS = 0) اعلام می‌شود. تنها هنگامی‌که سیگنال‌های P1 و P2 از حالت "00" به "01" یا "10" تغییر وضعیت می‌دهند جهت چرخش تعیین می‌شود (شکل ۳-۹).



شکل ۳-۹ پالس‌های P1 و P2

توصیف طراحی

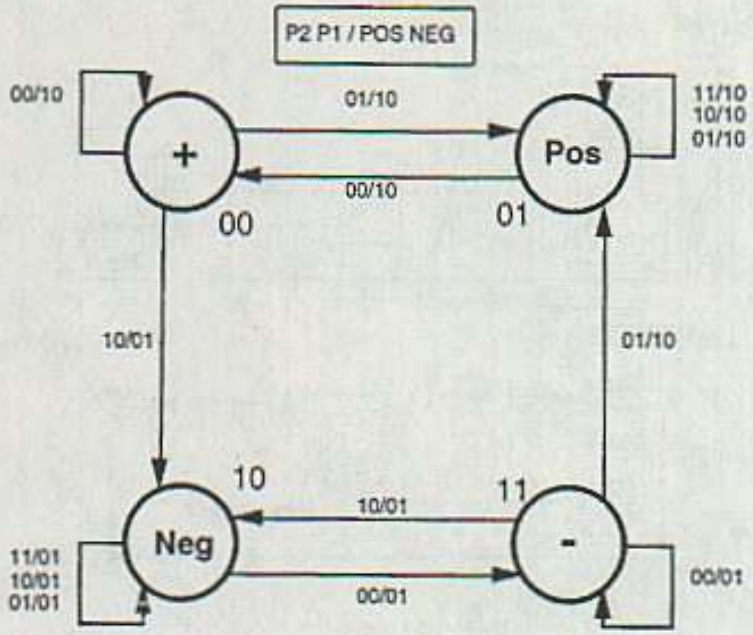
ماشین نوع «میلی» با چهار حالتی که دارد برای توصیف واحد کنترل‌کننده کفایت می‌کند (شکل ۴-۹ را ببینید). هر چهار حالت می‌تواند توسط دو بیت تعریف شوند.

حالت

نحوه رمزگشایی حالتها به صورت زیر است:

نشانه حالت مثبت	+	(00)
نشانه حالت منفی	-	(00)
نشانه حالت مثبت	Pos	(01)
نشانه حالت منفی	Neg	(10)

اگر سیگنال‌های ورودی به صورت "01" (p1, p2) باشند، حالت '+' می‌تواند به حالت 'Pos' تغییر کند و اگر این سیگنال‌ها به صورت "10" باشند به حالت 'Neg' تغییر می‌یابد. در همان لحظه‌ای که حالت تغییر می‌کند، سیگنال خروجی به "0" تغییر خواهد کرد (PosNeg). این از مشخصه‌های ماشین میلی است (شکل ۴-۹ را ببینید).



شکل ۴-۹ ماشین حالت

نقشه «کارنو» برای خلاصه کردن و توصیف منطق برنامه به کار می‌رود. نقشه‌های کارنو در کلیه کتابهای پایه‌ای الکترونیک توضیح داده شده‌اند. معادلات بولی برای این ماشین حالت به شکل زیر است :

$$D2 = q_2 P_1 + q_2 \overline{q_1} + \overline{q_1} P_2 = ((q_2 P_1)' \cdot (q_2 \overline{q_1})' \cdot (\overline{q_1} P_2)')'$$

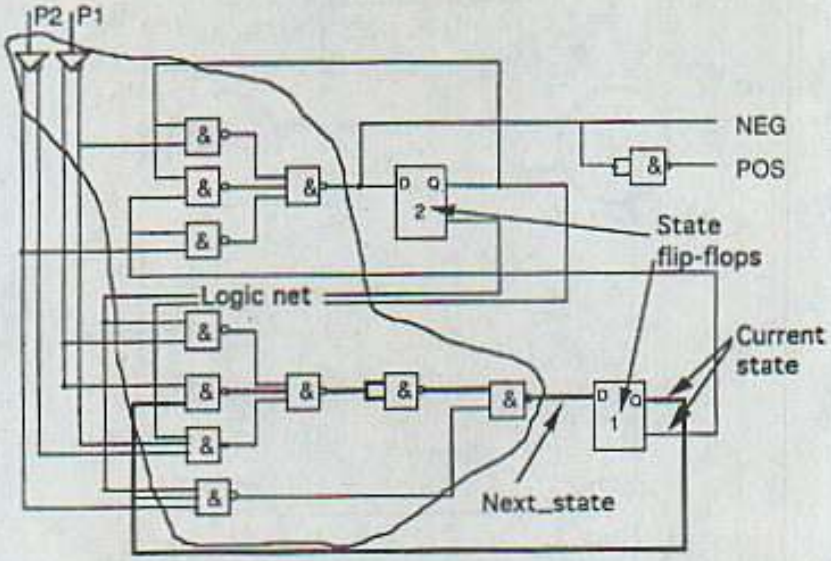
$$D1 = \overline{q_2} P_1 + q_1 P_1 + q_2 P_2 P_1 + \overline{q_2} q_1 P_2 = ((\overline{q_2} P_1)' \cdot (q_1 P_1)' \cdot (q_2 P_2 P_1)')' \cdot \overline{q_2} q_1 P_2 =$$
$$= ((((\overline{q_2} P_1)' \cdot (q_1 P_1)' \cdot (q_2 P_2 P_1)')')' \cdot (\overline{q_2} q_1 P_2)')'$$

$$POS = \overline{q_2} P_2 + q_1 P_1 + \overline{q_2} q_1 = ((\overline{q_2} P_2)' \cdot (q_1 P_1)' \cdot (\overline{q_2} q_1)')'$$

$$NEG = q_2 P_1 + q_2 \overline{q_1} + \overline{q_1} P_2 = ((q_2 P_1)' \cdot (q_2 \overline{q_1})' \cdot (\overline{q_1} P_2)')'$$

D1 و D2 بیانگر حالت بعدی و ورودی‌های دو فلیپ فلاپ هستند. q1 و q2 بیانگر حالت فعلی و خروجی‌های دو فلیپ فلاپ می‌باشند. با توجه به این معادله‌ها، D2 و NEG با هم برابر هستند و بنابراین می‌توان از یک مدار برای هر دو استفاده کرد. POS نیز باید وارونه NEG باشد، بنابراین نیازی نیست که POS مدار فرعی یا «زیرمدار» خود را داشته باشد بلکه به جای آن سیگنال NEG تقسیم و وارونه می‌شود.

نتیجه عبارات «بولی»، در شکل ۵-۹ به صورت شماتیک گیتی نشان داده شده است.



شکل ۵-۹ شماتیک در سطح گیتی

فلیپ فلاپ‌های 1 و 2، فلیپ فلاپ‌های حالت هستند. سیگنال‌های clk و reset برای فلیپ فلاپ‌ها ترسیم نشده‌اند ولی اتصال دارند. شبکه منطقی تعیین‌کننده حالت بعدی در داخل خط منحنی نشان داده شده است. سیگنال‌های خروجی NEG و POS به شبکه منطقی متصل‌اند. این بدان معناست که این سیگنال‌ها ممکن است دربردارنده نویز باشند. طولانی‌ترین شبکه گیتی شامل چهار گیت (که با خط پررنگ به فلیپ فلاپ D1 متصل است) می‌باشد، بنابراین اگر تأخیر مربوط به هر گیت را ۲/۵ns فرض کنیم، سریع‌ترین پالس ساعت ۱۰۰MHZ خواهد بود (۱۰ns).

طراحی مصور شده در شکل ۵-۹ را ممکن است به عنوان تمرینی در ترجمه توصیف در سطح انتقال رجیستری (RLT) به توصیف در سطح گیتی قبول نمود اما نمی‌توان آن را به عنوان یک طراحی در سطح RT پذیرفت. متداول‌ترین نکات این مساله به صورت زیر می‌باشد :

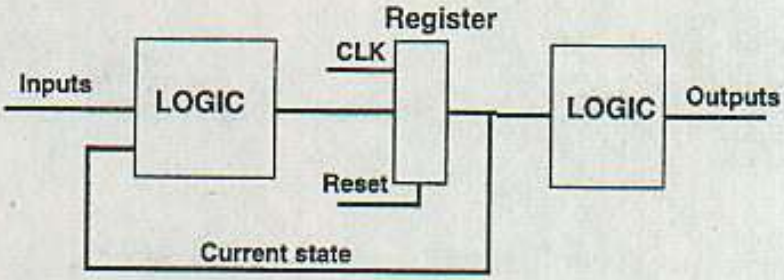
- تغییر مسیر عمل در ماشین حالت فقط زمانی مشخص می‌شود که سیگنال ورودی قبلی '00' باشد.
- خروجی‌ها بدون نویز نیستند.

این نکات به اندازه‌ای اهمیت دارند که باعث طراحی مجدد مدار می‌شوند.

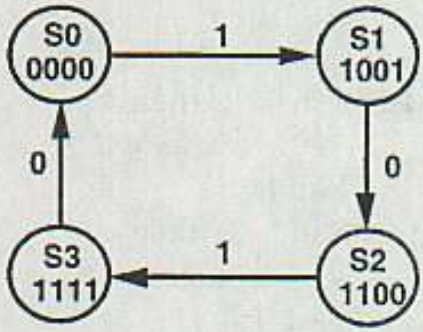
امروزه، دیگر طراحان نیازی به استفاده از جدول کارنو، خلاصه‌سازی، ترجمه به گیت‌های وابسته به تکنولوژی یا کشیدن شماتیک ندارند. کلیه این کارها به طور خودکار توسط ابزارهای سنتز RLT انجام می‌شوند.

۹-۱ ماشین موور

همان طور که در بلوک دیاگرام شکل ۹-۶ مشاهده می‌کنیم، سیگنال‌های خروجی تابعی از بردارهای حالت و به عبارت دیگر یک منطق ترکیبی مابین بردارهای حالت و خروجی‌ها هستند. یک ماشین «موور» را می‌توان با یک دیاگرام حالت نیز معرفی کرد (شکل ۹-۷).



شکل ۹-۶ بلوک دیاگرام یک ماشین «موور»



شکل ۹-۷ دیاگرام حالت یک ماشین «موور»

همان طور که شکل ۹-۶ نشان می‌دهد، سیگنال‌های خروجی تنها به حالتی که ماشین در آن قرار دارد، بستگی دارند. به همین دلیل است که معمولاً سیگنال‌های خروجی را در داخل حباب حالتها می‌نویسند. «صفرها» و «یک‌ها» یی که در شکل ۹-۷ در داخل حبابها قرار دارند نشان‌دهنده ارزشهای سیگنال‌های خروجی می‌باشند. به طور مثال در حالت S0 سیگنال خروجی ارزش "0000" و در حالت S1 ارزش "1001" را دارد. این مطلب نباید با کدگذاری حالتهای S0، S1، S2 و S3 اشتباه شود. شکل ۹-۷ هیچ نکته‌ای را در مورد کدگذاری حالتها بیان نمی‌کند. کدگذاری حالتها هیچ اثری بر روی رفتار حالتها ندارد و از این رو نیازی به توصیف آنها در کد VHDL نیست. اغلب ابزارهای سنتز دربردارنده یک قسمت ویژه برای بهینه‌سازی ماشین حالت نیز هستند.

یک نمونه از ماشین «موور» می‌تواند به صورت زیر باشد :

```
Entity demo is
port (clk,in1,reset:      in  std_logic;
      out1:               out std_logic);
end demo;

Architecture moore of demo is
type state_type is (s0,s1,s2,s3);              -- State declaration
signal state: state_type;

begin
  demo_process: process(clk,reset)             -- Clocked process
  begin
      if reset = '0' then
            state<=s0;                         -- Reset state
      elsif clk'event and clk= '1' then
            case state is
              when s0=> if in1= '1' then
                              state<=s1;
                        end if;
              when s1=> if in1= '0' then
                              state<=s2;
                        end if;
              when s2=> if in1= '1' then
                              state<=s3;
                        end if;
              when s3=> if in1= '0' then
                              state<=s0;
                        end if;
            end case;
      end if;
  end process;

  output_p:process(state)                      -- Combinational process
  begin
      case state is
        when s0=> out1 <= "0000";
```

```
            when s1=> out1 <= "1001";
            when s2=> out1 <= "1100";
            when s3=> out1 <= "1111";
        end case;
    end process;
end moore;
```

این ماشین حالت شامل سه قسمت است : بخش اعلام، بخش پروسس پالسی و بخش پروسس ترکیبی.

بخش اعلام

بخش اعلام بیان‌کننده حالت است. هر اسمی را می‌توان برای حالتها در نظر گرفت، اما توصیه می‌شود که برای توصیف از نامهای توضیحی استفاده شود.

مثال :

```
type state_type is (start_state,run_state,error_state);
signal state: state_type;
```

نام مناسب، شبیه‌سازی را نیز آسان می‌کند. بردار حالت ارزشهایی را اختیار می‌کند که برای نوع داده‌ها تعریف شده است، مثل start، idle، wait، run و error. این ارزشها در طول مدت شبیه‌سازی در پنجره شکل موجها در مقابل سیگنال‌ها نوشته می‌شوند.

پروسس پالسی

پروسس پالسی تعیین می‌کند که چه زمانی ماشین حالت باید تغییر حالت دهد. این پروسس با پالس ساعت ماشین حالت فعال می‌شود. با هر لبه پالس ساعت وضعیت ماشین بسته به حالت کنونی و مقدار سیگنال‌های ورودی تغییر خواهد کرد. پس از اطمینان از فعال شدن پالس ساعت، دستور case برای چک کردن وضعیتی که ماشین در آن قرار دارد به کار گرفته می‌شود. دستور if-then-else برای چک کردن مقدار سیگنال‌های ورودی بسیار مناسب است. به کارگیری دستور case و به دنبال آن دستور if-then-else معمولاً ساختار مناسب و خوانایی برای کد VHDL به وجود می‌آورد. بردار حالت مطابق مثال بالا در سیگنال state در بخش اعلام معماری برنامه ذخیره می‌شود.

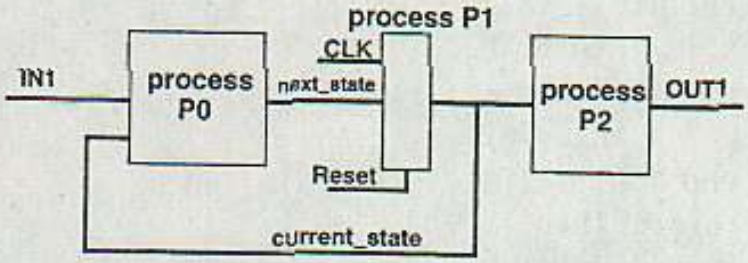
پروسس ترکیبی

پروسس ترکیبی، پروسسی است که طی آن ارزش سیگنال‌های خروجی بر حسب حالت فعلی ماشین به آنها داده می‌شود. در مثال فوق تنها بردارهای حالت در لیست حساسیت پروسس output_p فهرست می‌شوند که با مفهوم ماشین «موور» مطابقت دارد.

مدل عمومی فوق یکی از طرق متعددی است که برای توصیف حالت در VHDL به کار می‌رود. مدلهای دیگر از سه پروسس متفاوت استفاده می‌کنند : یکی برای رمزگشایی حالت بعدی، یکی جهت دادن حالت فعلی به بردار حالت و دیگری برای مشخص کردن سیگنال‌های خروجی. کد VHDL زیر تا حدی شبیه بلوک دیاگرام تعریف شده برای ماشین «موور» است (به شکل ۸-۹ مراجعه شود). عیب این روش آن است که به نوشتن کد VHDL بیشتری نیاز دارد و شبیه‌سازی آن نیز قدری کندتر است. اینکه کدام یک از این روشها مورد استفاده قرار گیرد چندان مهم نیست ولی آنچه توصیه می‌شود آن است که همه طراحان یک شرکت از یک مدل استفاده کنند. از لحاظ نتایج سنتز هیچ تفاوتی بین مدل زیر یا سه پروسس گفته شده در بالا و مدل قبل با دو پروسس مربوط به توصیف ماشین «موور» وجود ندارد.

```
Entity demo is
port (clk,in1,reset:      in  std_logic;
      out1:               out std_logic);
end demo;

Architecture moore of demo is
type state_type is (s0,s1,s2,s3);      -- State declaration
signal current_state, next_state: state_type;
begin
  P0: process(state,in1)
  begin
      case state is                    -- Combinational process
        when s0=> if in1= '1' then
                          next_state<=s1;
                  end if;
        when s1=> if in1= '0' then
                          next_state<=s2;
                  end if;
        when s2=> if in1= '1' then
                          next_state<=s3;
                  end if;
```

```
            when s3=> if in1= '0' then
                                next_state<=s0;
                      end if;
        end case;
    end process;

    P1:process(clk,seset)              -- Clocked process
    begin
        if reset = '1' then
                state<=s0;             -- Reset state
        elsif clk'event and clk= '1' then
                current_state<=next_state;
        end if;
    end process;

    P2: process (current_state)                -- Combinational process
    begin
        case state is
          when s0=> out1 <= "0000";
          when s1=> out1 <= "1001";
          when s2=> out1 <= "1100";
          when s3=> out1 <= "1111";
        end case;
    end process;
end moore;
```



شکل ۸-۹ ماشین «موور» با سه پروسس

بعضی از ابزارهای ساده سنتز قادر به تعریف بردار حالت از طریق تعیین نوع داده‌های لیست شده نیستند. این ابزار مستلزم کدگذاری حالت با کد VHDL می‌باشند. بعضی از این ابزارهای ساده سنتز از لحاظ دستور case نیز محدودیت دارند. ابزارهای سنتز کامپیوتری ViewLogic مثالی از

ابزارهایی است که محدودیتهای فوق در آنها دیده شده است. در این مورد کد VHDL برای ماشین «موور» را باید به صورت زیر نوشت :

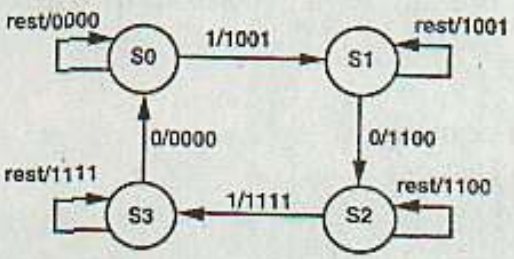
```
Entity demo is
port (clk,in1,reset:      in  vlbit;
      out1:               out vlbit_vector(3 downto 0));
end demo;

Architecture moore of demo is
   type state_type is array (1 downto 0) of vlbit;
   constant s0: state_type := "00"; -- State encoding
   constant s1: state_type := "01";
   constant s2: state_type := "10";
   constant s3: state_type := "11";

   begin
   demo_process: process
   begin
        wait until prising(clk);
        if reset = '1' then
              state<=s0;
        else
          if state = s0 then
              if in1= '1' then
                    state<=s1;
              end if;
          elsif state=s1 then
              if in1= '0' then
                    state<=s2;
              end if;
          elsif state=s2 then
              if in1= '1' then
                    state <=s3;
              end if;
          else
              if in1= '0' then
                    state<=s0;
              end if;
```

```
        end if;
    end if;
end process;

output_p:process(state)
begin
    if state=s0 then
        out1<= "0000";
    elsif state=s1 then
        out1<= "1001";
    elsif state=s2 then
        out1<= "1100";
    else
        out1<= "1111";
    end if;
end process;
end moore;
```

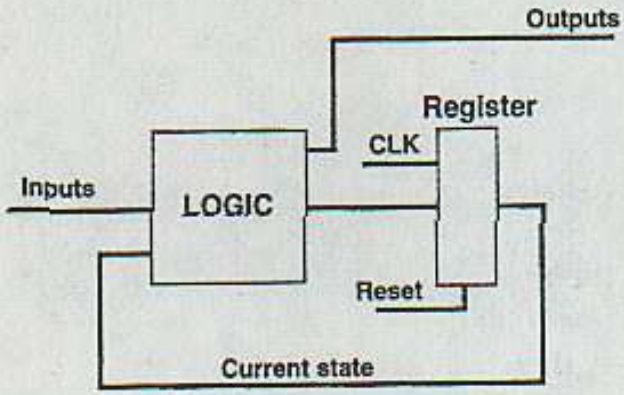
در مثال بالا نوع داده vlbit (ViewLogic bit) مورد استفاده قرار گرفته است. اصول مدل عمومی ماشین حالت با آنچه ViewLogic در این مورد حمایت می‌کند، یکی هستند. هر دو دارای یک پروسس پالسی برای دادن مقدار به بردار حالت و یک پروسس ترکیبی برای دادن مقدار به سیگنال‌های خروجی هستند. اگر ابزار سنتز مورد استفاده، مدل عمومی ماشین را حمایت کند (که اغلب این طور است)، توصیه می‌شود که ماشین حالت بر اساس این مدل طراحی شود. در این صورت طراح نیازی به کدگذاری مراحل و حالتها ندارد. این کار به ابزار سنتز واگذار خواهد شد.

۹-۲ ماشین میلی

ماشین «میلی» همیشه به اندازه یک پالس ساعت از ماشین «موور» معادل آن جلوتر است. شکلهای ۹-۹ و ۹-۱۰ دیاگرام حالت و بلوک دیاگرام یک ماشین «میلی» را نشان می‌دهند.



شکل ۹-۹ دیاگرام حالت یک ماشین «میلی»



شکل ۱۰-۹ بلوک دیاگرام یک ماشین «میلی»

همان‌طور که در شکل ۱۰-۹دیده می‌شود سیگنال‌های‌خروجی، هم وابسته به کلیه سیگنال‌های ورودی و هم وابسته به حالت فعلی هستند. این بدان معناست که هرگاه سیگنال‌های ورودی و یا حالت سیستم تغییر کنند، سیگنال‌های خروجی نیز بلافاصله تغییر می‌کنند. دیاگرام حالت نمی‌تواند ارزش سیگنال‌ها را به طور روشن بر پایه حالت فعلی معین کند، بلکه مقدار سیگنال‌های ورودی نیز باید در محاسبه وارد شوند. این موضوع باعث می‌شود که سیگنال‌های خروجی بر روی فلشهای دیاگرام حالت ترسیم شوند، به طور مثال در حالت S0، اگر سیگنال ورودی برابر '1' باشد، سیگنال‌های خروجی مقدار "1001" را اختیار خواهند کرد و اگر سیگنال ورودی دارای هر مقدار دیگری باشد خروجی "0000" خواهد بود.

ساختار ماشین «میلی» دقیقاً مشابه ساختار ماشین «موور» است. تفاوت آنها در پروسس ترکیبی است که مقدار سیگنال‌های خروجی را تعیین می‌کند، این پروسس باید تابع بردار حالت و تمام سیگنال‌های ورودی باشد. از طرف دیگر، پروسس پالسی صرف‌نظر از اینکه طراحی ماشین بر اساس مدل «میلی» یا «موور» باشد یکی هستند.

مثال ماشین «میلی»:

```
Entity demo is
port (clk,in1,reset:      in  std_logic;
      out1:               out std_logic_vector (3 downto 0));
end demo;

Architecture mealy of demo is
type state_type is (s0,s1,s2,s3);
signal state: state_type;
begin
   demo_process: process(clk,reset)
```

```
begin
      if reset = '1' then
            state<=s0;
      else clk'event and clk= '1' then
        case state is
            when s0=> if in1= '1' then
                        state <= s1;
                      end if;
            when s1=> if in1= '0' then
                        state <= s2;
                      end if;
            when s2=> if in1= '1' then
                        state <= s3;
                      end if;
            when s3=> if in1= '0' then
                        state <= s0;
                      end if;
        end case;
      end if;
end process;

output_p:process(state,in1)
begin
      case state is
        when s0=> if in1= '1' then
                    out1<= "1001";
                  else
                    out1<= "0000";
                  end if;
        when s1=> if in1= '0' then
                    out1<= "1100";
                  else
                    out1<= "1001";
                  end if;
        when s2=> if in1= '1' then
                    out1<= "1111";
                  else
                    out1<= "1001";
                  end if;
```

```
        when s3=> if in1= '0' then
                    out1<= "0000";
                  else
                    out1<= "1111";
                  end if;
    end case;
  end process;
end mealy;
```

همان طور که در مثالهای ماشین «موور» اشاره شد، ابزارهای ساده سنتز وجود دارند که مدل عمومی ماشین «میلی» (مثال بالا) را حمایت نمی‌کنند. اشکال عمده کار این است که کدگذاری حالتها باید در کد VHDL منظور شود. در این حالت کد VHDL را می‌توان به شکل زیر نوشت:

```
Architecture mealy of demo is
  type state_type is array (1 downto 0) of vlbit;
  constant s0: state_type := "00";
  constant s1: state_type := "01";
  constant s2: state_type := "10";
  constant s3: state_type := "11";
  begin
  demo_process: process
      begin
          wait until prising (clk);
          if reset = '1' then
              state<=s0;
          else
            if state=s0 then
              if in1= '1' then
                  state<=s1;
              end if;
            elsif state=s1 then
              if in1= '0' then
                  state<= s2;
              end if;
            elsif state=s2 then
              if in1= '1' then
                  state<= s3;
              end if;
```

```
                else
                    if in1= '0' then
                        state<=s0;
                    end if;
                end if;
            end if;
    end process;

    output_p:process(state, in1)
    begin
        if state=s0 then
            if in1= '1' then
                out1<= "1001";
            else
                out1<= "0000";
            end if;
        elsif state=s1 then
            if in1= '0' then
                out1<= "1100";
            else
                out1<= "1001";
            end if;
        elsif state=s2 then
            if in1= '1' then
                out1<= "1111";
            else
                out1<= "1100";
            end if;
        else
            if in1= '0' then
                out1<= "0000";
            else
                out1<= "1111";
            end if;
        end if;
    end process;
end;
```

۹-۳ تنوع ماشینهای «میلی» و «موور»

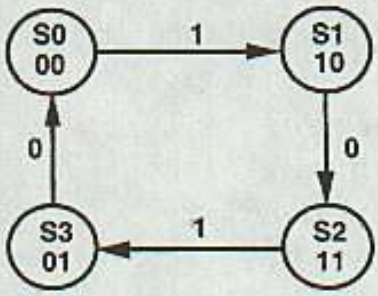
خروجی‌ها در هر دو ماشینهای حالت «موور» و «میلی» ممکن است دارای جهشهای نویزی باشند و دلیل آن این است که سیگنال‌های خروجی محصول منطق ترکیبی هستند. ممکن است ماشینهای حالت در درجه حرارت و ولتاژ خاصی بدون جهشهای نویزی باشند، همچنان که ممکن است در درجه حرارتی دیگر و یا ولتاژی دیگر دارای جهشها نویزی باشند. به طور معمول این جهشها اهمیتی ندارند. در طراحی هم‌زمان، داده باید تنها در لبه فعال کلاک پایدار باشد. سیگنال‌های داده ممکن است کمی بعد از لبه پالس ساعت، جهش نشان دهند که روی کار سیستم اثر چندانی نخواهد داشت. از سویی دیگر، نمی‌توان از سیگنال‌های خروجی ماشینهای حالت «میلی» و «موور» برای سیگنال فعال‌کننده بافر سه حالته و یا پالس ساعت استفاده کرد، زیرا تضمینی برای بدون نویز بودن آنها نیست.

اگر خروجی‌های بدون جهش نویزی موردنیاز باشد می‌توان از سه نوع دیگر این ماشینهای حالت استفاده کرد:

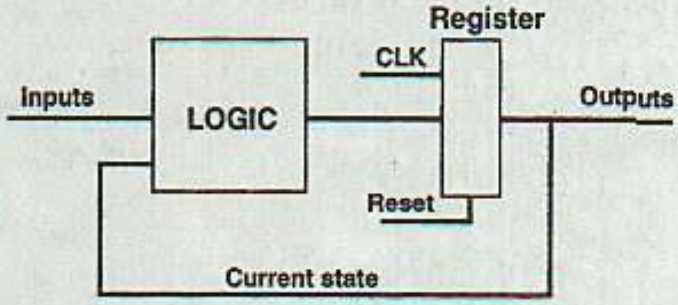
- خروجی = ماشین حالت
- ماشین «موور» با خروجی پالسی
- ماشین «میلی» با خروجی پالسی

۹-٤ خروجی = ماشین حالت

در نوع خروجی = ماشین حالت، مقدار و ارزش بردار حالت مستقیماً به سیگنال‌های خروجی داده می‌شود. بدین معنا که کدگذاری حالت باید در کد VHDL انجام شود. «خروجی = ماشین حالت» یکی از مثالهای خاص ماشین حالت «موور» است که در آن پروسس ترکیبی برای تعیین مقدار خروجی‌ها کاملاً بهینه شده است. اگر بخواهیم یک دیاگرام حالت برای «خروجی = ماشین حالت» ترسیم کنیم، کاملاً مشابه یک ماشین «موور» معمولی خواهد بود (شکل ۱۱-۹)، در حالی که دیاگرام حالت بیان مناسبی برای آن نیست. با توجه به بلوک دیاگرام شکل ۱۲-۹ واضح است که مقادیر سیگنال‌های خروجی، از بردار حالت (مثلاً فلیپ فلاپ‌های حالت) گرفته می‌شوند که در این صورت بدون نویز بودن خروجی حتمی است.



شکل ۱۱-۹ دیاگرام حالت



شکل ۱۲-۹ بلوک دیاگرام

یک مثال از «خروجی = ماشین حالت»، در زیر آورده شده است :

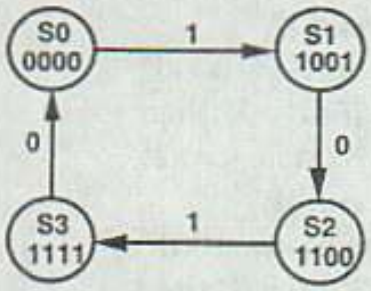
```
Entity demo is
port (clk,in1,reset:    in  std_logic;
      out1:             out std_logic_vector (1 downto 0));
end demo;
Architecture moore of demo is
   type state_type is array (1 downto 0) of std_logic;        -- State encoding
   constant s0: state_type := "00";
   constant s1: state_type := "10";
   constant s2: state_type := "11";
   constant s3: state_type := "01";
begin
   demo_process: process(clk,reset)
   begin
        if reset = '1' then
              state<=s0;
        elsif clk'event and clk= '1' then
          case state is
```

```
            when s0=> if in1= '1' then
                          state<=s1;
                      end if;
            when s1=> if in1= '0' then
                          state<=s2;
                      end if;
            when s2=> if in1= '1' then
                          state<=s3;
                      end if;
            when s3=> if in1= '0' then
                          state<=s0;
                      end if;
        end case;
    end if;
  end process;
  out1<=state;          -- Assign state vector to output
end;
```

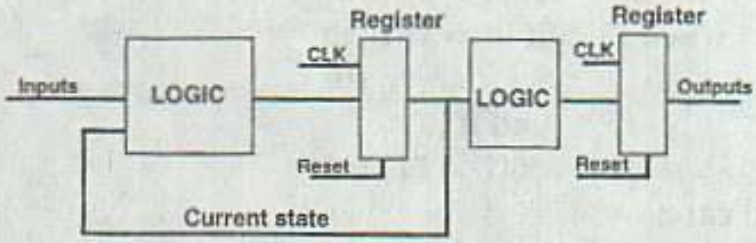
۵-۹ ماشین مورو با خروجی‌های پالسی

تفاوت بین یک «ماشین مورو با خروجی‌های پالسی» و یک ماشین «مورو» معمولی این است که در «ماشین مورو با خروجی‌های پالسی» کلیه خروجی‌ها با یک فلیپ فلاپ اضافی از نوع D، *هم‌زمان شده‌اند*[1] و بدین ترتیب خروجی‌ها بدون جهش نویزی خواهند بود. خروجی‌ها در این حالت به اندازه یک پالس ساعت عقب‌تر از خروجی‌های ماشین «مورو» معمولی هستند. دیاگرام حالت بیان مناسبی برای این ماشین نیست، زیرا این دیاگرام کاملاً مشابه دیاگرام حالت ماشین «مورو» معمولی خواهد بود (شکل ۱۳-۹). در بلوک دیاگرام شکل ۱۴-۹ مشخص شده است که سیگنال‌های خروجی با یک فلیپ فلاپ نوع D سنکرون شده‌اند. بیان این مطلب در کد VHDL به مفهوم به کارگیری پروسس پالسی است. این کار باعث می‌شود که به کلیه سیگنال‌های خروجی یک فلیپ فلاپ از نوع D داده شود.

1- Synchronized



شکل ۱۳-۹ دیاگرام حالت برای «ماشین موور با خروجی‌های پالسی»



شکل ۱۳-۹ بلوک دیاگرام برای «ماشین موور با خروجی‌های پالسی»

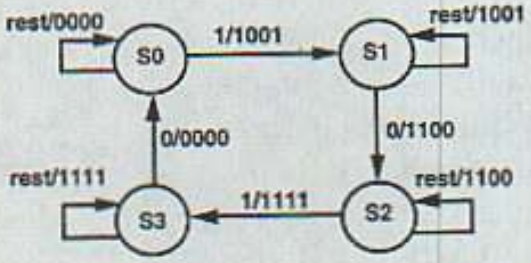
مثالی از یک «ماشین موور با خروجی‌های پالسی» می‌تواند به صورت زیر باشد :

```
Entity demo is
port (clk,in1,reset:      in  std_logic;
      out1:               out std_logic);
end demo;

Architecture moore of demo is
  type state_type is (s0,s1,s2,s3);
  signal state: state_type;

begin
  demo_process: process(clk,reset)
  begin
      if reset = '1' then
            state<=s0;
            out1<=(others=> '0');
      elsif clk'event and clk= '1' then
        case state is
            when s0=> if in1= '1' then
                        state <= s1;
```

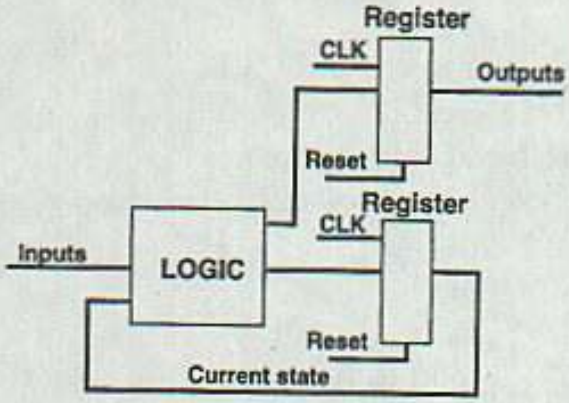
```
                end if;
                out1<= "0000";
        when s1=> if in1= '0' then
                    state <= s2;
                end if;
                out1<= "1001";
        when s2=> if in1= '1' then
                    state <= s3;
                end if;
                out1<= "1100";
        when s3=> if in1= '0' then
                    state <= s0;
                end if;
                out1<= "1111";
    end case;
  end if;
end process;
end;
```

۹-۶ ماشین میلی با خروجی‌های پالسی

تفاوت بین یک «ماشین میلی با خروجی‌های پالسی» و یک ماشین «میلی» معمولی این است که در «ماشین میلی با خروجی‌های پالسی» کلیه خروجی‌ها با یک فلیپ فلاپ اضافی از نوع D، هم‌زمان شده‌اند که این کار منجر به بدون نویز شدن خروجی‌ها می‌شود. خروجی‌ها در این حالت به اندازه یک پالس عقب‌تر از ماشین «میلی» معمولی هستند. دیاگرام حالت بیان مناسبی برای این ماشین نیست، زیرا این دیاگرام کاملاً مشابه دیاگرام حالت ماشین «میلی» خواهد بود (شکل ۱۵-۹). در بلوک دیاگرام شکل ۱۶-۹ مشخص شده است که سیگنال‌های خروجی با یک فلیپ فلاپ از نوع D سنکرون شده‌اند.



شکل ۱۵-۹ دیاگرام حالت برای «ماشین میلی با خروجی‌های پالسی»



شکل ۱۶-۹ بلوک دیاگرام برای «ماشین میلی با خروجی‌های پالسی»

مثالی از یک «ماشین میلی با خروجی‌های پالسی» می‌تواند به صورت زیر باشد :

```
Entity demo is
port (clk,in1,reset:      in  std_logic;
      out1:               out std_logic_vector(3 downto 0));
end demo;
Architecture mealy of demo is
  type state_type is (s0,s1,s2,s3);
  signal state: state_type;
begin
  demo_process: process(clk,reset)
  begin
      if reset = '1' then
            state<=s0;
            out1<=(others=> '0');
      elsif clk'event and clk= '1' then
        case state is
            when s0=> if in1= '1' then
                        state <= s1;
                        out1<= "1001";
                      else
                        out1<= "0000";
                      end if;
            when s1=> if in1= '0' then
                        state <= s2;
                        out1<= "1100";
```

```
                else
                    out1<= "1001";
                end if;
            when s2=> if in1= '1' then
                    state <= s3;
                    out1<= "1111";
                else
                    out1<= "1100";
                end if;
            when s3=> if in1= '0' then
                    state <= s0;
                    out1<= "0000";
                else
                    out1<= "1111";
                end if;
        end case;
    end if;
  end process;
end;
```

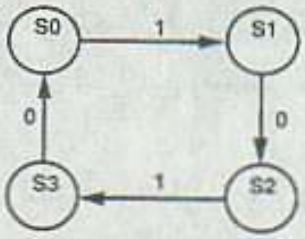
۷-۹ کدگذاری حالت

همان طور که قبلاً اشاره شد، کدگذاری حالت بر روی عملکرد ماشین حالت بی‌اثر خواهد بود و می‌تواند توسط ابزار سنتز انتخاب شود. تنها مورد استثنای از این قاعده «خروجی = ماشین حالت» است. بعضی از ابزارهای سنتز دارای بخش بهینه‌ساز حالت هستند که امکان تعیین کد حالت را به هنگام سنتز کد VHDL دارند. انواع متداول کدهای حالت به شرح زیر است :

- Sequential
- Gray
- One-hot
- Random
- Auto (کوچک کردن فضای موردنیاز)

فرض کنید که حالتهای یک ماشین حالت به صورت زیر تعریف شده باشند (شکل ۱۷-۹).

```
type state_type is (s1, s2, s3);
```



شکل ۱۷- ۹ دیاگرام حالت

در موارد گوناگون ممکن است کدگذاری‌های زیر به دست آید :

Sequential	Gray	One-hot
S0 = "00";	S0 = "00";	S0 = "0001";
S1 = "01";	S1 = "01";	S1 = "0010";
S2 = "10";	S2 = "11";	S2 = "0100";
S3 = "11";	S3 = "10";	S3 = "1000";

اگر بهینه‌سازی ماشین حالت بدون استفاده از بهینه‌سازهای خاص انجام شود، کدگذاری حالت به صورت Sequential خواهد بود. کد Gray به این معناست که وقتی ماشین حالت تغییر حالت می‌دهد فقط یک بیت در بردار حالت مقدار خود را تغییر می‌دهد. کد One-hot به تعداد حالتها فلیپ فلاپ دارد و این بدان معناست که تعداد گیت‌ها افزایش می‌یابد که با توجه به زیاد بودن تعداد فلیپ فلاپ‌ها در ساختار بعضی از FPGA ها (نوع Xlinix) مشکل خاصی ایجاد نخواهد کرد. این بدان معناست که به هنگام سنتز در FPGA می‌توان از کد One-hot استفاده کرد.

۸-۹ حالتهای باقیمانده[1]

اگر از کدگذاری One-hot استفاده نشود، کل تعداد حالت برابر با 2N خواهد بود، که N برابر طول بردار حالت است. در یک ماشین حالت که در آن از کلیه حالتها استفاده نشده است به سه طریق می‌توان عمل نمود :

۱) اگر وارد یک حالت تعریف نشده شدید بگذارید عامل شانس تعیین کند که چه اتفاقی باید رخ دهد.

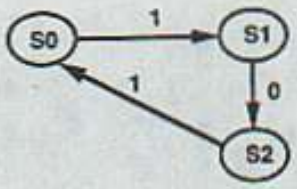
۲) اگر وارد یک حالت تعریف نشده شدید از طریق دستور when others حالت را خاتمه دهید.

۳) تمام حالتهای ممکن را در کد VHDL تعریف کنید.

1- Residual

به طور معمول گزینه اول به دلیل آنکه در آن نیازی به کدگذاری حالتهای تعریف شده وجود ندارد به منطق کمی احتیاج خواهد داشت. لیکن این وضعیت ممکن است ماشین را در مواجهه با یک حالت تعریف نشده دچار اشکال کند. تنها راه بیرون آمدن از این حالت وضعیت فعال کردن سیگنال reset است. گزینه دوم مطمئن‌تر است اما نیاز به منطق‌سازی بیشتری دارد ولی متأسفانه همه ابزارهای سنتز این گزینه را حمایت نمی‌کنند. بنابراین باید از راه حل سوم یعنی تعریف کردن کلیه حالتهای ممکن در کد VHDL استفاده کرد.

در شکل ۱۸-۹ مثالی از یک ماشین حالت ارائه شده است.



شکل ۱۸-۹ مثال ماشین حالت

ماشین حالت در شکل فوق تنها دارای سه حالت است. اگر بهینه‌سازی ماشین با منطق معمولی انجام شود (بدون به کارگیری ابزار بهینه‌ساز) کدگذاری Sequential مورد استفاده قرار می‌گیرد. به طور مثال :

```
S0 = "00";
S1 = "01";
S2 = "10";
```

حالت "11" تعریف نشده است. بهتر است در این مورد نگاهی به هر سه راه حل بیندازیم.

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex is
port (clk,resetn,a:     in  std_logic;
      q:                out std_logic);
end;
```

راه حل اول :

```
Architecture alt1 of ex is
type state_type is (s0,s1,s2);
signal state: state_type;
```

```
begin
   process(clk,resetn)
   begin
        if reset = '0' then
                state<=s0;
                q<= '0';
        elsif clk= '1' and clk'event then
          case state is
                when s0=> state<=s1;
                          q<= '0';
                when s1=> if a= '1' then
                             state<=s2;
                          end if;
                          q<= '1';
                when s2=> state<=s0;
                          q<= '0';
          end case;
        end if;
   end process;
end;
```

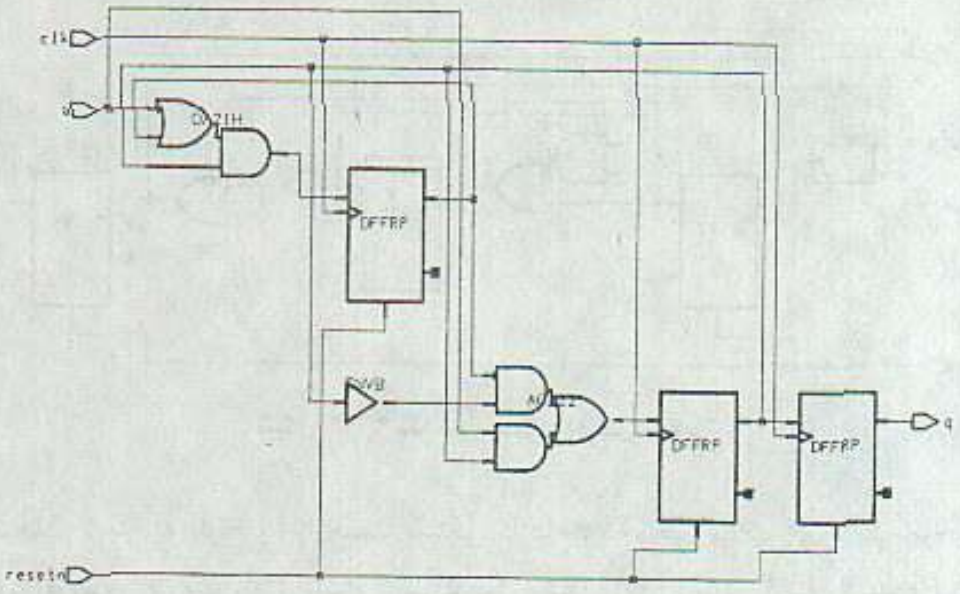
راه حل دوم :

```
Architecture alt2 of ex is
type state_type is (s0,s1,s2);
signal state: state_type;
begin
   proess(clk,resetn)
   begin
        if reset = '0' then
                state<=s0;
                q<= '0';
        elsif clk= '1' and clk'event then
          case state is
                when s0=> state<=s1;
                          q<= '0';
                when s1=> if a= '1' then
                             state<=s2;
                          end if;
```

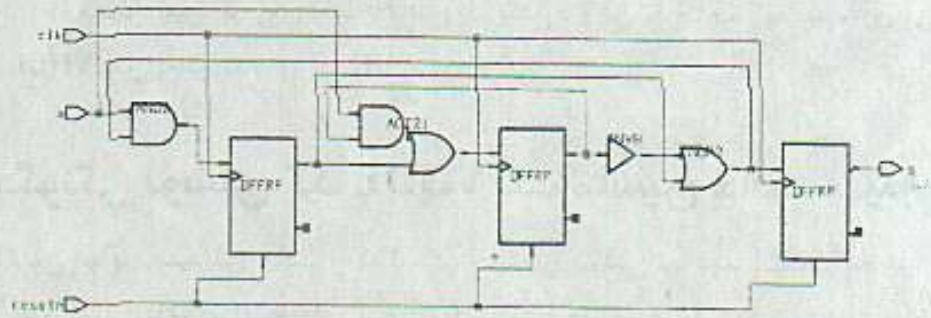
```
                    q<= '1';
            when s2=> state<=s0;
                    q<= '0';
            when others => state<=s0;
                        q<= '0';
        end case;
    end if;
  end process;
end;
```

راه حل سوم :

```
Architecture alt3 of ex is
type state_type is (s0,s1,s2,s3);
signal state: state_type;
begin
  proess(clk,resetn)
  begin
      if reset = '0' then
            state<=s0;
            q<= '0';
      elsif clk= '1' and clk'event then
        case state is
            when s0=> state<=s1;
                    q<= '0';
            when s1=> if a= '1' then
                        state<=s2;
                    end if;
                    q<= '1';
            when s2=> state<=s0;
                    q<= '0';
            when s3 => state<=s0;
                    q<= '0';
        end case;
      end if;
  end process;
end;
```

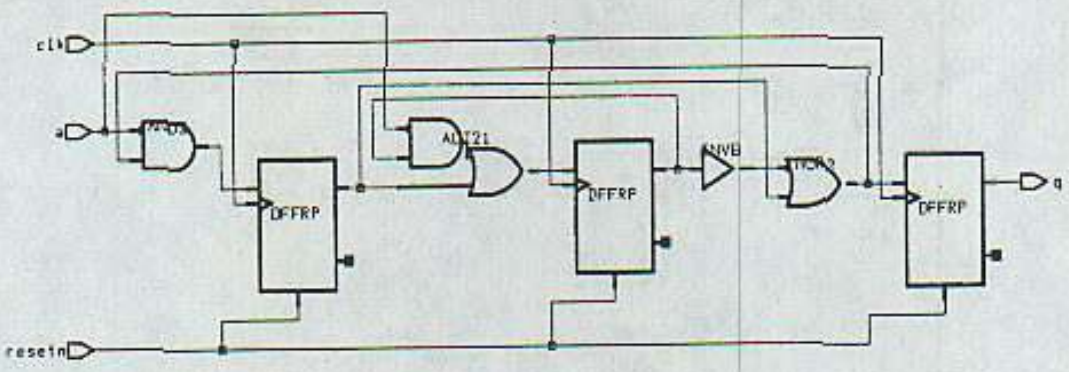


شکل ۱۹-۹ نتیجه سنتز راه حل اول



شکل ۲۰-۹ نتیجه سنتز راه حل دوم

با توجه به نتیجه سنتز در شکلهای ۲۰-۹ و ۲۱-۹، راه‌حلهای ۲ و ۳ یکسان هستند. هر گاه یک عمل غلط، ماشین را به حالت تعریف نشده "11" هدایت کند، با توجه به راه حل اول، تا زمانی که سیگنال ورودی a برابر '1' شود ماشین در این حالت باقی می‌ماند. این رفتار سیستم فقط بعد از سنتز به وجود می‌آید. نتیجه حاصل بر حسب اینکه ابزار سنتز چگونه به بهینه‌سازی منطقی بپردازد تصادفی خواهد بود. بنابراین این راه‌حل برای طراحی مناسب نیست. بدیهی است چنان‌چه نخواهیم ریسک گیر افتادن ماشین بر اثر یک عمل اشتباه را قبول کنیم، گزینه فوق نباید انتخاب شود. بنابراین توصیه می‌شود که همواره کلیه حالتها برای ماشین حالت تعریف شود تا از گرفتار شدن آن جلوگیری گردد. گزینه اول فقط زمانی باید مورد استفاده قرار گیرد که گزینه‌های ۲ و ۳ به حجم بیش از حد نیاز داشته باشند.



شکل ۲۱-۹ نتیجه سنتز راه حل سوم

انتخاب گزینه به طراح و ابزار سنتز بستگی دارد. اگر نخواهیم ماشین وارد یک مرحله تعریف نشده بشود (فرضاً بر اثر یک عمل اشتباه یا بر اثر یک مشکل زمانی) باید یکی از گزینه‌های ۲ و ۳ را بسته به نوع ابزار سنتز انتخاب نماییم. Synopsys یکی از ابزارهایی است که راه حل دوم را حمایت می‌کند. چنان‌چه از Synopsys و راه حل دوم استفاده کنیم باید دقت نماییم که ابزار بهینه‌ساز حالت فعال نشود زیرا ممکن است بر روی خط others اثر بگذارد و آن را حذف کند. به جای آن می‌توان از بهینه‌ساز VHDL مثل دیگر کدهای VHDL استفاده کرد.

۹-۹ چگونگی نوشتن کد VHDL یک ماشین حالت بهینه

تاکنون انواع ماشینهای حالت و کدگذاری حالت را بررسی نمودیم. انتخاب ماشین حالت کاملاً به طرح (بدون نویز بودن، فضای مورد نیاز، شروط زمانی و غیره) بستگی دارد. بعضی اوقات ترکیبی از ماشینهای «میلی»، «موور»، «میلی پالسی» یا «موور پالسی» می‌تواند انتخاب بهینه باشد. به طور مثال اگر می‌خواهید بعضی خروجی‌ها بدون جهش نویزی باشند کافی است آنها را با یک رجیستر سنکرون کنید (میلی/موور با خروجی‌های پالسی) و دیگر خروجی‌ها را به صورت میلی «خالص» یا موور «خالص» استفاده کنید.

مثالی از یک ماشین ترکیبی از میلی، موور، میلی پالسی و موور پالسی می‌تواند به صورت زیر باشد :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity mix is
port (clk,resetn,a,b: in  std_logic;
      q1:             out std_logic;  -- Mealy output
```

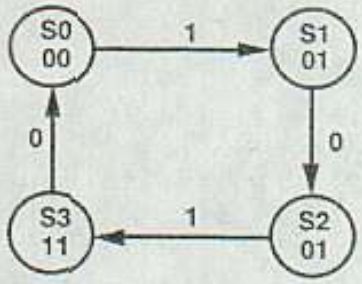
```
        q2:          out std_logic;  -- Moore output
        q3:          out std_logic;  -- S. Mealy output
        q4:          out std_logic); -- S. Moore output
end;

Architecture tr1 of mix is
type state_type is (s0,s1);
signal state: state_type;
begin
   process(clk,resetn)
   begin
        if resetn= '0' then
                state<=s0;
                q3<= '0';
                q4<= '0';
        elsif clk'event and clk= '1' then
           case state is
                when s0=> if a= '1' then
                                state <= s1;
                                q3<= '0';
                          else
                                q3<= '1';
                          end if;
                          q4<= '1';
                when s1=> if b= '1' then
                                state <= s2;
                                q3<= '1';
                          else
                                q3<= '0';
                          end if;
                          q4<= '0';
           end case;
        end if;
   end process;

   process(state,a,b)
   begin
        case state is
           when s0=> if a= '1' then
```

```
                q1<= '0';
              else
                q1<= '1';
              end if;
                q2<= '0';
        when s1=> if b= '1' then
                q1<= '1';
              elsif a= '0' then
                q1<= '0';
              else
                q1<= '1';
              end if;
                q2<= '1';
      end case;
    end process;
end;
```

همچنین می‌توانید گزینه «خروجی = ماشین حالت» را به عنوان یک نوع ماشین حالت انتخاب کنید زیرا این راه حل از لحاظ زمان سریع‌ترین و از لحاظ فضا کمترین حجم را دارد. تنها استثنا در این قاعده ماشین میلی است که می‌تواند تغییرات سیگنال خروجی را سریع‌تر انجام دهد. خروجی‌های ماشین میلی بلافاصله پس از تغییر مقدار سیگنال‌های ورودی آن تغییر خواهند کرد، در حالی که یک ماشین «خروجی = ماشین حالت» به اندازه یک پالس ساعت صبر می‌کند و سپس خروجی‌ها تغییر خواهند کرد.

اغلب در ماشینهای حالت کوچک از مدل «خروجی = ماشین حالت» استفاده می‌شود. در ضمن در «خروجی = ماشین حالت» حتماً باید سیگنال‌های خروجی در دو یا چند حالت با هم برابر باشند. چنان‌چه سیگنال‌های خروجی برابر باشند، کدگذاری حالتها در این موارد مانند هم خواهند بود که البته مجاز به داشتن این حالتها نیستیم.

در مثال زیر ماشین حالتی داریم که در آن سیگنال‌های خروجی در دو حالت یکسان هستند (S1 و S2). بعضی اوقات اضافه کردن یک فلیپ فلاپ اضافی از نوع D برای ایجاد تفاوت بین دو حالت و در نتیجه استفاده از مدل «خروجی = ماشین حالت» مفید خواهد بود.



شکل ۲۲-۹ ماشین حالتی که خروجی آن در دو حالت مثل هم است

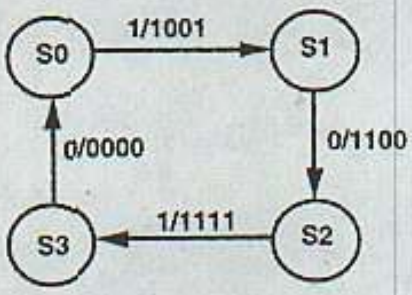
همان طور که در شکل ۲۲-۹ دیده می‌شود، سیگنال‌های خروجی در حالتهای S1 و S2 یکسان می‌باشند.

در مثال زیر با به کارگیری یک فلیپ فلاپ از نوع D برای بردار حالت امکان انتخاب مدل «خروجی = ماشین حالت» فراهم شده است :

```
Architecture rt1 of ex is
constant s0: std_logic_vector(2 downto 0):= "000";
constant s1: std_logic_vector(2 downto 0):= "001";
constant s2: std_logic_vector(2 downto 0):= "101";
constant s3: std_logic_vector(2 downto 0):= "011";
constant state: std_logic_vector(2 downto 0);
begin
  process(clk,resetn)
  begin
      if resetn= '0' then
            state<=s0;
      elsif clk'event and clk= '1' then
        case state is
            when s0=> if a= '1' then
                        state<=s1;
                      end if;
            when s1=> if b= '1' then
                        state<=s2;
                      end if;
            when s2=> if a= '0' then
                        state<=s3;
                      end if;
           when others => state <= s0;
```

```
        end case;
      end if;
  end process;
  q_out <= state(1 downto 0);
end;
```

پس از طراحی یک ماشین حالت باید ارزش و مقدار کلیه سیگنال‌ها در هر پالس ساعت تعیین گردد. مثالی از آن در شکل ۲۳-۹ آورده شده است :



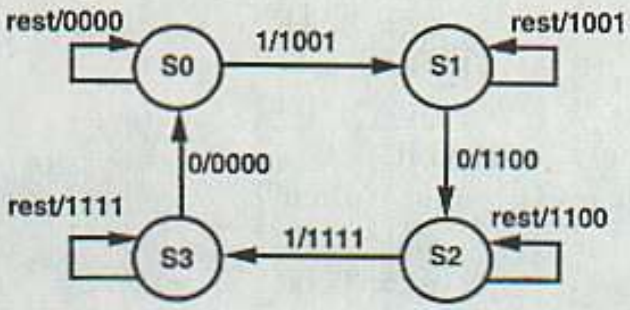
شکل ۲۳-۹ دیاگرام حالت

مدل زیر از نقطه نظر عملکرد صحیح است، گرچه به سیگنال‌های خروجی در هر لبه فعال پالس ساعت مقدار داده نشده است. به طور مثال، چنان‌چه ماشین در حالت S1 و سیگنال ورودی a برابر '1' باشد، به سیگنال خروجی هیچ مقداری داده نمی‌شود. این موضوع توسط ابزار سنتز این طور تفسیر می‌شود که سیگنال‌های خروجی مقدار قبلی خودشان را نگه می‌دارند (که درست است).

```
Architecture bad of ex is
type state_type is (s0,s1,s2,s3);
signal state: state_type;
begin
  process(clk,resetn)
  begin
      if resetn= '0' then
            state<=s0;
            q<=(others=>'0');
      elsif clk'event and clk= '1' then
        case state is
            when s0=> if a= '1' then
                        state<=s1;
```

```
                    q<= "1001";
                end if;
        when s1=> if a= '0' then
                    state<=s2;
                    q<= "1100";
                end if;
        when s2=> if a= '0' then
                    state<=s3;
                    q<= "1111";
                end if;
        when s3 => if a= '0' then
                    state<= s0;
                    q<= "0000";
                end if;
    end case;
  end if;
end process;
end;
```

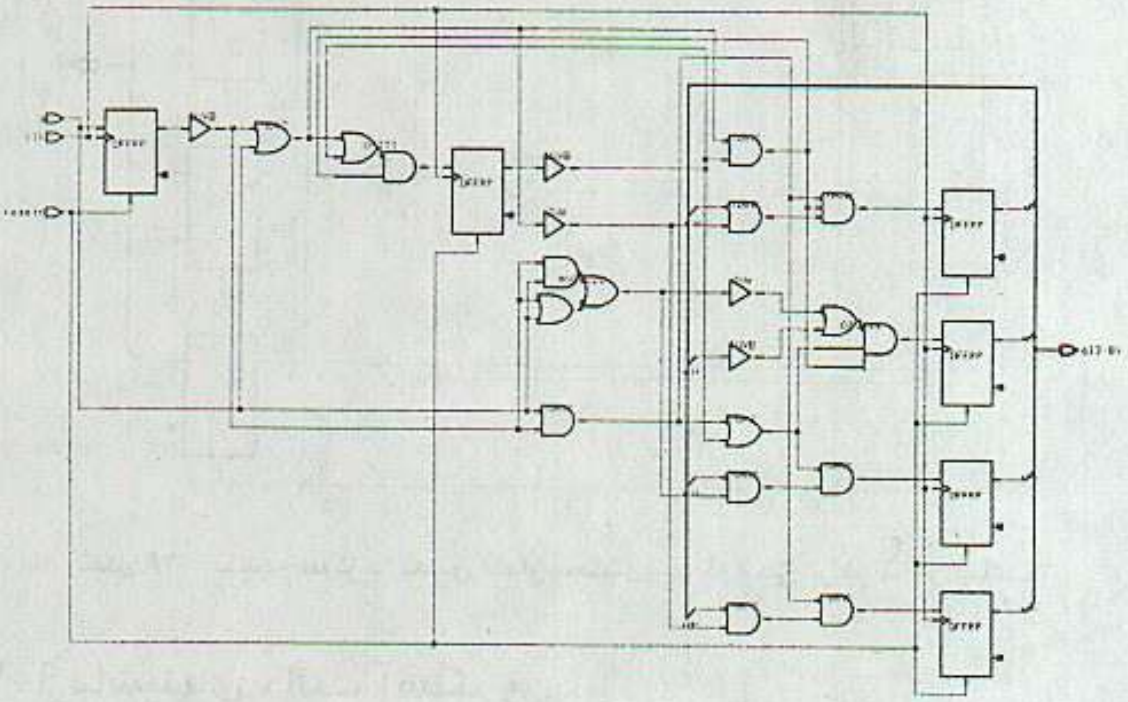
در کد VHDL بالا، چنان‌چه ماشین تغییر حالت ندهد به سیگنال‌های خروجی هیچ مقداری داده نمی‌شود. دیاگرام حالت شکل ۲۳-۹ این رفتار را نمایش می‌دهد. اگر دیاگرام حالت دوباره ترسیم شده و کلیه مقداردهی‌ها به سیگنال‌های‌خروجی در هر حالت بدون تغییر عملکرد ماشین مشخص شود (شکل ۲۴-۹)، برنامه به فرم زیر خواهد بود :



شکل ۲۴-۹ دیاگرام حالت

```
Architecture good of ex is
type state_type is (s0,s1,s2,s3);
signal state: state_type;
begin
   process(clk,resetn)
   begin
        if resetn= '0' then
                state<=s0;
                q<=(others=> '0');
        elsif clk'event and clk= '1' then
          case state is
                when s0=> if a= '1' then
                                state<=s1;
                                q<= "1001";
                          else
                                q<= "0000";
                          end if;
                when s1=> if a= '0' then
                                state<=s2;
                                q<= "1100";
                          else
                                q<= "100";
                          end if;
                when s2=> if a= '0' then
                                state<=s3;
                                q<= "1111";
                          else
                                q<= "1100";
                          end if;
                when s3 => if a= '0' then
                                state<= s0;
                                q<= "0000";
                           else
                                q<= "1111";
                           end if;
          end case;
        end if;
   end process;
end;
```

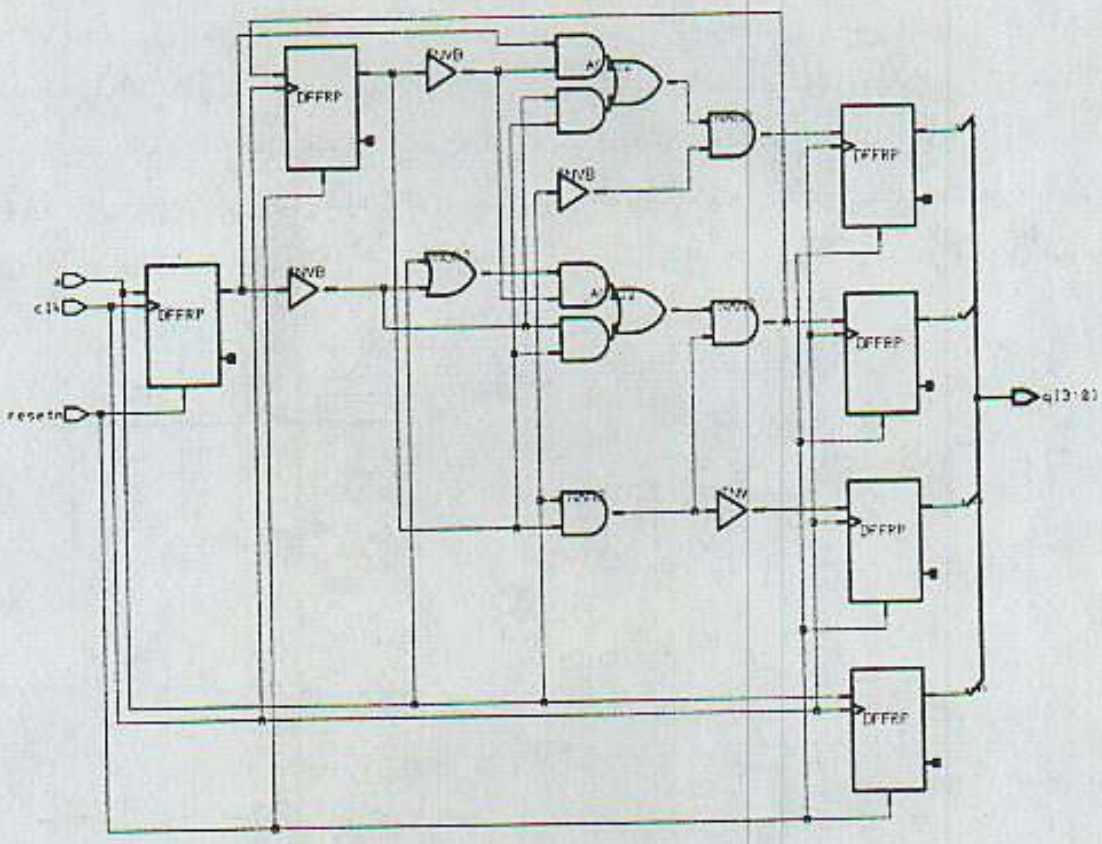
همان طور که قبلاً گفته شد، عملکرد هر دو مثال فوق در شبیه‌سازی VHDL و بعد از سنتز در سطح گیتی کاملاً همانند خواهد بود. عیب مدل اول آن است که به اندازه حدود ۱۲ گیت بزرگ‌تر است زیرا در این مدل به سیگنال‌های خروجی در هر پالس ساعت هیچ مقدار و ارزشی داده نمی‌شود. بنابراین در این مدل برای حفظ مقدار قبلی به سخت‌افزار بیشتری نیاز خواهد بود. ابزارهای سنتز این کار را با ایجاد یک مسیر فیدبک از خروجی و گذراندن آن از یک مالتی پلکسر (شکل ۲۵-۹) انجام می‌دهند. در پالس‌های ساعتی که مقدار جدیدی به سیگنال‌ها داده نمی‌شود، مقدار قبلی توسط مالتی پلکسر انتخاب شده و به آنها داده می‌شود. به این طریق مقدار داده شده به سیگنال‌های خروجی حفظ می‌شود. چنان‌چه در هر پالس ساعت به سیگنال‌های خروجی مقدار داده می‌شد، به مسیر فیدبک و مالتی پلکسر نیازی نداشتیم (شکل ۲۶-۹) و به این ترتیب برای هر خروجی۳ گیت صرفه‌جویی می‌شد.



شکل ۲۵-۹ نتیجه سنتز بدون تعیین مقدار برای سیگنال‌های خروجی در هر پالس ساعت

سه گیت برای هر خروجی ممکن است چندان زیاد به نظر نیاید اما اگر طراحی کوچک باشد و یا از چندین ماشین حالت استفاده شده باشد، این گیت‌های اضافی می‌توانند یک مدار بزرگ‌تری که برای منطق‌سازی موردنیاز خواهد بود ایجاد نمایند. به علاوه باید در نظر داشت که همان چند گیت اضافی می‌تواند در طراحی‌های بزرگ نقش چند گاه اضافی را داشته باشد که کمر شتر را خواهد شکست! علاوه بر این، زمان‌بندی مناسب ماشین حالت نیز بر اثر وجود مالتی پلکسرهای غیرضروری به

هم می‌ریزد. همچنین اگر از فیدبک و مالتی پلکسر استفاده شود، ظرفیت خازنی خروجی افزایش می‌یابد و این امر از نقطه نظر زمان‌بندی به مفهوم کند شدن تغییرات خروجی است که این امر موجب لطمه دیدن برنامه زمانی ماشین می‌گردد.

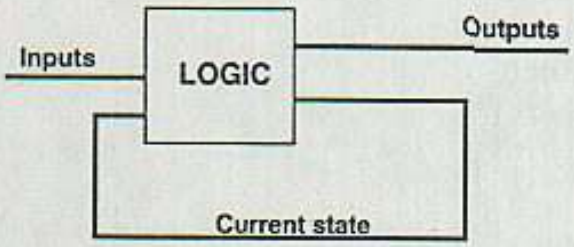


شکل ۲۶-۹ نتیجه سنتز با تعیین مقدار سیگنال‌های خروجی در هر پالس ساعت

۹-۱۰ ماشینهای حالت آسنکرون

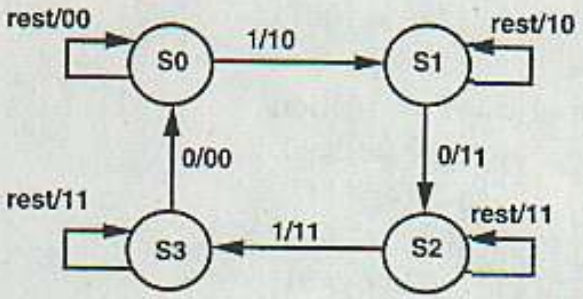
کلیه ماشینهای حالت که تاکنون در مورد آنها بحث کردیم پالسی و سنکرون بودند. اما اگر فیدبک بردار حالت بدون استفاده از هیچ گیتی انجام شود (شکل ۲۷-۹)، ماشین حالت به صورت غیرهم‌زمان (آسنکرون) در خواهد آمد. از مزایای ماشینهای حالت آسنکرون این است که معمولاً این ماشینها در مقایسه با ماشینهای سنکرون و پالسی سریع‌تر عمل می‌کنند. از نقطه نظر روشهای طراحی، ماشینهای آسنکرون گزینه بهینه نیستند (به فصل ۱۱ مراجعه کنید، «آشنایی با روشهای طراحی») و بهتر است تنها تحت شرایط سخت و خاص که تنها این گونه ماشینها راه حل خواهند بود از آنها استفاده کرد.

به هنگام طراحی ماشینهای حالت آسنکرون، طراح باید مطمئن شود که آنچه به نام racing شناخته شده است واقع نخواهد شد. racing به تغییر چندباره حالتهای یک ماشین حالت در آن واحد تحت شرایط غیرقابل کنترل اطلاق می‌شود. کدگذاری حالتها نیز در ماشینهای حالت غیرهم‌زمان بسیار مهم است. بردار حالت می‌تواند در هر نوبت فقط یک بیت را تغییر دهد و این به معنای آن است که کدگذاری حالت باید در برنامه VHDL گنجانده شود. بسیاری از ابزارهای سنتز ماشینهای حالت آسنکرون را حمایت نمی‌کنند و آنهایی که حمایت می‌کنند هیچ گونه توانایی برای بهینه‌سازی زمانی و حجمی ندارند.



شکل ۲۷-۹ ماشین حالت غیرهم‌زمان

فرض کنید ماشین حالت شکل ۲۸-۹ را بخواهیم به طور هم‌زمان طراحی کنیم.



شکل ۲۸-۹ دیاگرام حالت

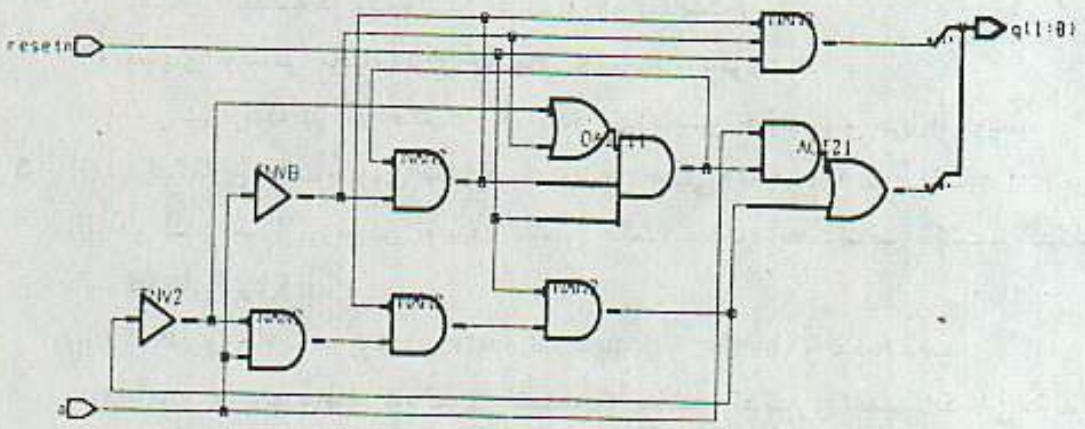
بنابراین کد VHDL آن به صورت زیر خواهد بود :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity mix is
port (a,resetn:    in  std_logic;
      q:           out std_logic_vector (1 downto 0));
end;
```

```
Architecture tr1 of test is
type state_type is array (1 downto 0) of std_logic;
constant s0: state_type:= "00";
constant s1: state_type:= "01";
constant s2: state_type:= "11";
constant s3: state_type:= "10";
signal state: state_type;
begin
   process(resetn,state,a)
   begin
        if resetn= '0' then
                state<= s1;
                q<= "10";
        else
           case state is
                when s0 => if a= '1' then
                                state<=s1;
                                q<= "10";
                           else
                                state<=s0;
                                q<= "00";
                           end if;
                when s1 => if a= '0' then
                                state<=s2;
                                q<= "11";
                           else
                                state<=s1;
                                q<= "10";
                           end if;
                when s2 => if a= '1' then
                                state<=s3;
                                q<= "10";
                           else
                                state<=s2;
                                q<= "11";
                           end if;
                when others => if a= '0' then
                                        state<=s0;
                                        q<= "11";
```

```
                else
                    state<=s3;
                    q<= "11";
                end if;
            end case;
        end if;
    end process;
end;
```

نتیجه سنتز در شکل ۲۹-۹ نشان داده شده است.



شکل ۲۹-۹ نتیجه سنتز

۱۱-۹ تمرین

۱- تفاوت بین یک «ماشین میلی» و یک «ماشین موور» چیست ؟

۲- (i) آیا تضمینی برای بدون نویز بودن خروجی‌های ماشینهای «میلی» و «موور» وجود دارد ؟

(ii) چه ماشینهای حالتی دارای خروجی‌های بدون جهشهای نویزی هستند ؟

(iii) دو موقعیت را مشخص کنید که در آنها بدون نویز بودن خروجی‌های ماشین حالت مورد نیاز باشد.

۳- آیا امکان ترکیب انواع مختلف ماشینهای حالت در یک ماشین وجود دارد ؟

۴- (i) حالتهای باقیمانده در یک ماشین حالت به چه معناست ؟

(ii) روشهای توصیفی مختلف برای حالتهای باقیمانده را ذکر کنید.

(iii) دو روشی را که در آنها حالتهای باقیمانده برای ماشین حالت پیش می‌آید بنویسید.

۵- (i) آیا کدگذاری حالت بر روی عملکرد یک ماشین «میلی» یا «موور» اثر می‌گذارد ؟

(ii) در کدام یک از روشهای توصیف ماشین حالت، عملکرد تحت تأثیر مکانیسم کدگذاری حالت قرار می‌گیرد ؟

(iii) کدگذاری حالات به روشهای Gary، Sequential و One-hot را توضح دهید.

(iv) آیا برای تکنولوژی‌های گوناگون، مکانیسمهای مختلف کدگذاری اهمیت یکسانی دارند؟

(v) اگر بردار حالت به شکل زیر تعیین شده باشد :

```
type state_type is (S0, S1, S2, S3, S4);
signal state : state_type;
```

چند فلیپ فلاپ برای مکانیسم کدگذاری به روشهای Sequential و One-hot موردنیاز خواهد بود؟

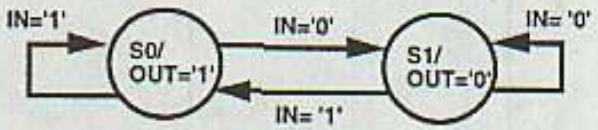
۶- در کدام یک از ماشینهای حالت، تغییرات خروجی بلافاصله بعد از تغییرات ورودی صورت می‌گیرد ؟

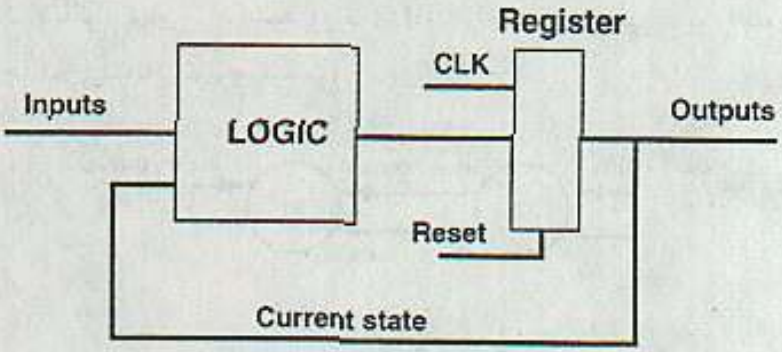
۷- (i) اگر امکان استفاده از «خروجی = ماشین حالت» وجود داشته باشد، سیگنال‌های خروجی چه شرایطی باید داشته باشند ؟

(ii) آیا هیچ راه‌حلی برای عبور از مشکل بخش (i)-۷ وجود دارد ؟

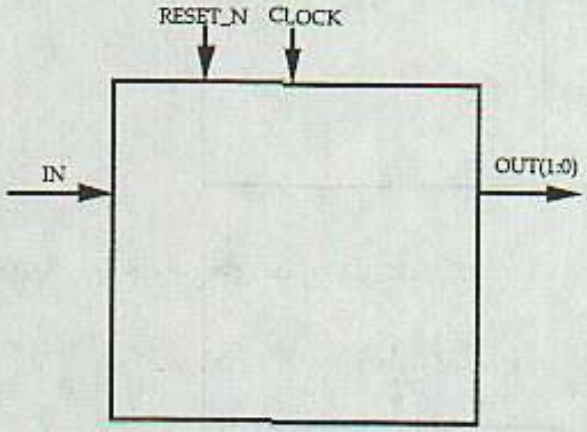
۸- اگر در ماشین «میلی» یا «موور» با خروجی‌های پالسی در هر پالس به هر یک از خروجی‌ها مقداری داده نشود چه اتفاقی می‌افتد ؟

۹- (i) ماشین حالت باید از نوع «خروجی = ماشین حالت» باشد. شکل زیر دیاگرام حالت و بلوک دیاگرام ماشین را نشان می‌دهد. برای entity و architecture آن یک کد VHDL بنویسید. حالت اولیه هنگامی که reset_n = '0' حالت S0 است.

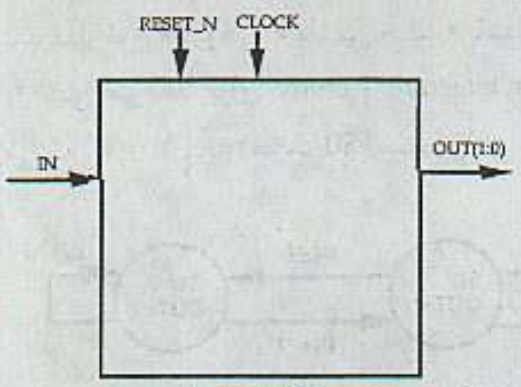




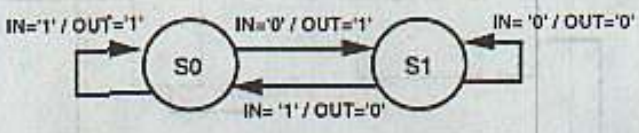
(ii) بلوک دیاگرام نهایی را بکشید. کد VHDL ماشین «موور» زیر را بنویسید (نیازی به حذف جهشهای نویزی نیست).

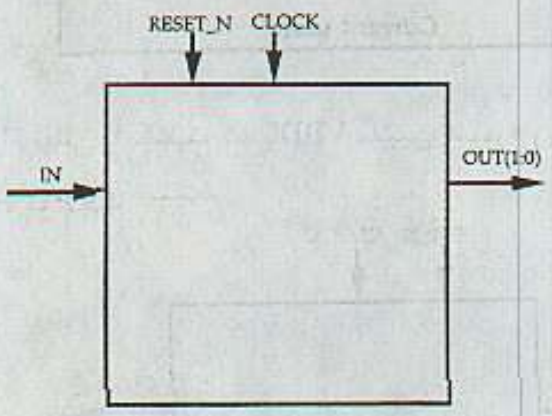


(iii) چنان‌چه از راه حل (ii)-۹ استفاده نشود، به چه طریق می‌توان نویز تمرین (ii)-۹ را حذف نمود.

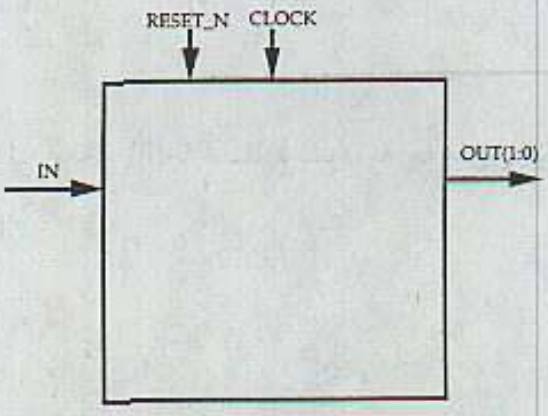


(iv) بلوک دیاگرام نهایی را بکشید و کد VHDL را ماشین «میلی» زیر را بنویسید (نیازی به حذف جهشهای نویزی نیست).

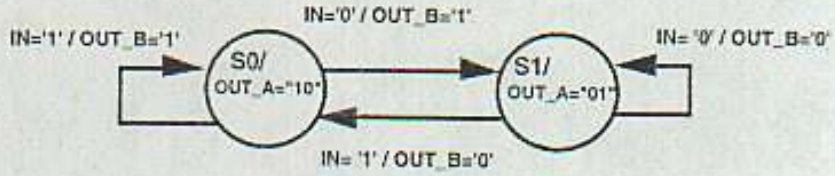


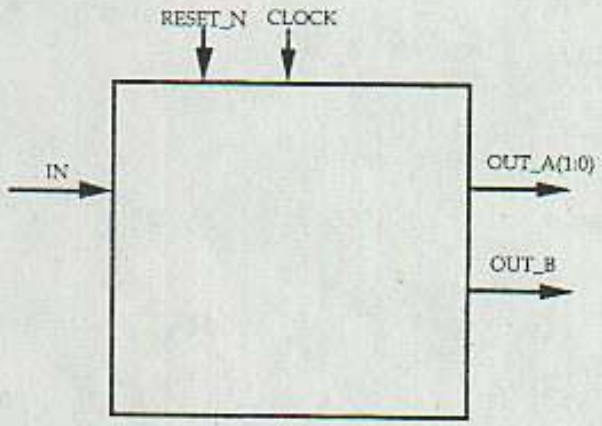


(v) تمرین (iv)-۹ را با خروجی‌های بدون نویز تکرار کنید.



(vi) بلوک دیاگرام نهایی را بکشید و کد VHDL برای ماشین حالت زیر بنویسید (بدون جهشهای نویزی).

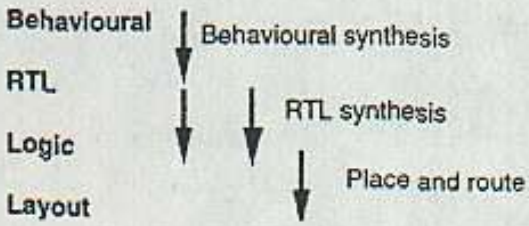




فصل دهم

سنتز در سطح RT

این فصل به سنتز کد VHDL در سطح RT می‌پردازد (شکل ۱-۱۰ را ببینید). سنتز در سطح رفتاری در فصل ۱۷، «سنتز رفتاری» مورد بررسی قرار می‌گیرد.



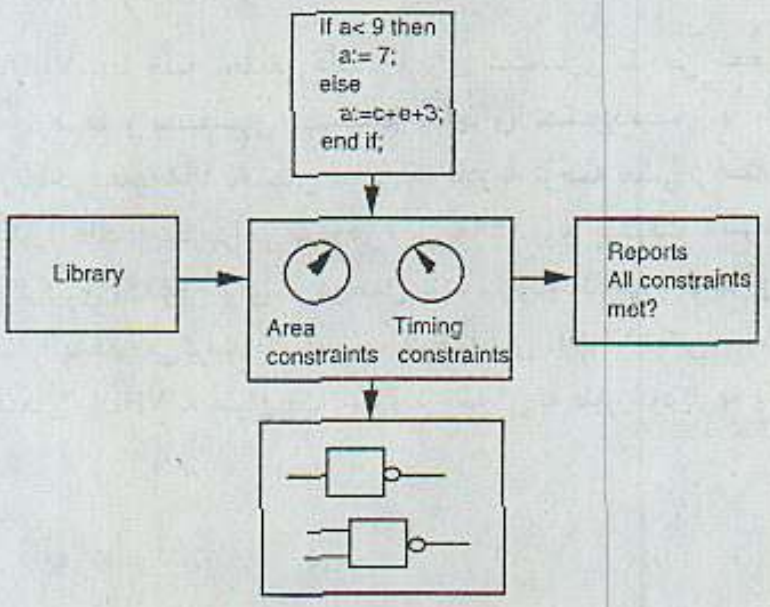
شکل ۱-۱۰ خلاصه سطوح توصیفی

در ابتدا VHDL تنها جنبه تحقیقی داشت و برای شبیه‌سازی طراحی شده بود. فکر و هدف اولیه این بود که توصیف و شبیه‌سازی سیستمها و اجزای تشکیل‌دهنده آنها با زبان سطح بالا امکان‌پذیر شود. در اواخر دهه ۱۹۸۰ نخستین ابزاری که قادر به ترجمه بعضی از استانداردهای VHDL به صورت یک شبکه (netlist) بود طراحی شد. در اوایل ۱۹۹۰ روند پیشرفت با سرعت بیشتری ادامه یافت و توانایی به کارگیری یک زبان برای هر دو عمل شبیه‌سازی و طراحی و فواید آن مورد توجه افراد بیشتری قرار گرفت. آنها فکر می‌کردند اگر توصیف طرح با زبان VHDL امکان‌پذیر گردد، گام بزرگی برای استفاده از کدهای VHDL به عنوان یک مبنا و ترجمه آن به طور خودکار به یک netlist خواهد بود.

امروزه عمل سنتز برای اکثر طرحهای ASIC با استفاده از یک زبان توصیف سخت‌افزاری انجام می‌گیرد. ابزار سنتز قادر است زبانهایی به غیر از VHDL را نیز قبول کند. بعضی از آنها به طور مثال Verilog را نیز می‌پذیرند. در حال حاضر VHDL و Verilog دو تا از زبانهای بزرگ توصیف سخت‌افزاری هستند و تقریباً ۵۰ درصد بازار را به خود اختصاص داده‌اند. لیکن زبان VHDL با سرعتی بیشتر از Verilog توسعه پیدا کرد و به زودی وسیع‌ترین زبان رایج چه در شبیه‌سازی و چه در سنتز خواهد شد.

سنتز کلیه دستورات استاندارد IEEE با زبان VHDL امکان‌پذیر نیست. تنها بخشی از این دستورات توسط ابزارهای سنتز حمایت می‌شوند. همچنین تفاوتهایی بین انواع ابزارهای سنتز و دستوراتی که هر کدام حمایت می‌کنند وجود دارد. در فصول قبل با دستوراتی که توسط بعضی از ابزارها حمایت نمی‌شوند آشنا شدیم. البته تلاش برای استاندارد کردن آن بخشی از VHDL که قابل سنتز شدن است همچنان ادامه دارد. چنین استانداردی یک سطح حداقلی را برای عملیاتی که یک ابزار سنتز باید قادر به انجامش باشد تعیین خواهد کرد. لیکن باید گفت که امروزه امکان جابه‌جایی بین ابزارهای سنتز وجود دارد. اما برای حرکت از یک ابزار پیشرفته و پیچیده به سوی یک ابزار ساده‌تر هنوز کار زیادی باید انجام گیرد.

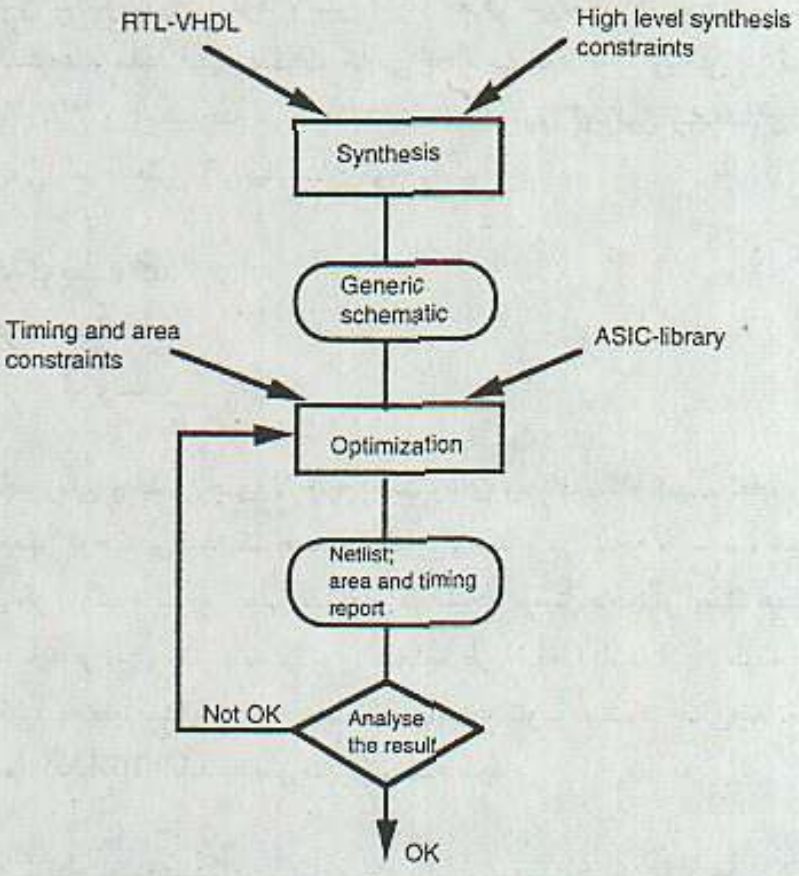
کار سنتز شبیه کاری است که کامپایلر در نرم‌افزار انجام می‌دهد. وظیفه آن ترجمه یک رشته مفاهیم پیچیده از یک توصیف سطح بالا به سطح پایین‌تر است. در این کار می‌توان پارامترهایی مانند سرعت، حجم و شرایط خاص برای هر یک از سیگنال‌ها را تعیین نمود.



شکل ۲-۱۰ ابزار سنتز

داده‌های ورودی به ابزار سنتز عبارتند از کد VHDL، فایل مربوط به تکنولوژی و محدودیتها و تنگناهای عمل (شکل ۲-۱۰). داده‌های خروجی آن نیز عبارتند از یک فایل گزارشی و یک netlist، که netlist را می‌توان یک نمای شماتیکی با گیت‌های مختلف در نظر گرفت. فایل گزارشی نیز حاوی اطلاعاتی است که نحوه عمل سنتز و نتیجه حاصل از آن را نشان می‌دهد.

یک فلوچارت نمونه برای نشان دادن روند سنتز و بهینه‌سازی در شکل ۳-۱۰ آورده شده است.



شکل ۳-۱۰ فلوچارت چگونگی روند سنتز و بهینه‌سازی

اکثر ابزارهای پیشرفته سنتز کار خود را با تولید یک شماتیک عمومی (مستقل از تکنولوژی) بر پایه کد VHDL شروع می‌کنند. ابزار سنتز به هنگام تولید شماتیک، بهینه‌سازی در سطح بالا مانند مشترک‌سازی منابع را امکان‌پذیر می‌سازد.حجم و اندازه شماتیک را نیز در این مرحله می‌توان مشخص نمود. این مرحله به نام سنتز و مرحله بعدی به نام بهینه‌سازی خوانده می‌شوند. در مرحله بهینه‌سازی باید یک کتابخانه حاوی اطلاعات مربوط به تکنولوژی خاص انتخاب شود. از آنجایی که قبل از شروع سنتز باید مدار تحلیل شود بنابراین کد VHDL باید عاری از هر گونه خطای عملکردی باشد. مسائلی

که در این مرحله ممکن است بروز نماید یکی مربوط به زمان و دیگری مربوط به فضا و حجم و سومی مربوط به شرایط خاص طرح از جمله زمانهای صعود و نزول سیگنال‌ها در شبکه‌های داخلی است.

۱-۱۰ بهینه‌سازی و نگاشت

هنگامی که ابزار سنتز کد VHDL را به یک شماتیک عمومی ترجمه می‌کند، جمعگرها، ضرب‌کننده‌ها، مالتی پلکسرها، رجیسترها و غیره را نیز معرفی می‌نماید. در مرحله بهینه‌سازی بر روی جمعگرها، ضرب‌کننده‌ها و مانند اینها عملیات خاصی انجام می‌شود. توابع ریاضی در کد VHDL که با نمادهای '='، '+'، '-'، '*' و '/' شناخته می‌شوند می‌توانند به طرق گوناگون پیاده‌سازی شوند. به عنوان مثال یک جمعگر می‌تواند به طرق زیر پیاده‌سازی و اجرا شود.

- جمعگر کوچک و کند
- جمعگر متوسط
- جمعگر بزرگ و سریع

اگر از لحاظ زمان و مقدار جای لازم به جمعگرهای مختلف نگاه کنیم، تفاوت میان آنها قابل ملاحظه است. جمعگر کوچک و کند یک جمعگر بسیار کوچک و در نتیجه با سرعت خیلی پایین است. از سوی دیگر جمعگر بزرگ و سریع خیلی سریع و در نتیجه نسبتاً حجیم‌تر است. همان طور که قبلاً گفته شد تأکید ما باید بر روی مدار و نه بر روی انتخاب طریقه اجرا باشد. این وظیفه ابزار سنتز است که باید تصمیم بگیرد به طور مثال چه نوع جمعگری باید به کار برده شود. چنان‌چه بخواهیم دو بردار را با هم جمع کنیم، کد VHDL باید به صورت زیر نوشته شود :

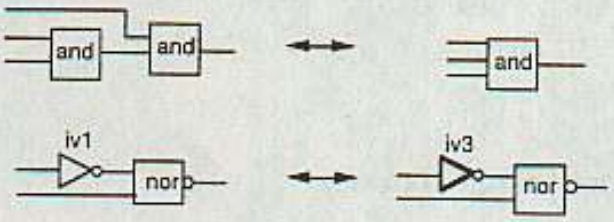
```
q <= a + b;
```

با توجه به تنگناها و محدودیتهای ابزار سنتز باید آن روش اجرایی را انتخاب کرد که با در نظر گرفتن شروط زمانی، کمترین جا را اشغال نماید.

به طور معمول منطق ترکیبی باید از چندین مرحله که به نام ***بهینه‌سازی‌های نگاشت***[1] معروف هستند، عبور نماید. نگاه ابزار سنتز در خلال بهینه‌سازی نگاشت بر روی پنجره کوچک منطق سازی است، به طور مثال پنجره‌ای با پنج ورودی و دو خروجی. اندازه این پنجره در ابزارهای گوناگون متفاوت خواهد بود. سپس ابزار سنتز از طریق این پنجره منطق‌سازی سعی خواهد کرد کار انتخاب گیت را با گزینش گیت‌های دیگری از کتابخانه (مطابق شکل ۴-۱۰) آسان‌تر سازد. ابزار سنتز همچنین

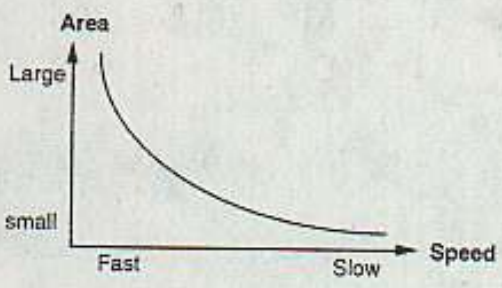
1- Mapping Optimization

قدرت درایورهای گیت‌ها را نیز انتخاب می‌کند. ابزار سنتز تمام این کارها را برای دستیابی به مدت زمان مطلوب و به حداقل رساندن جای مورد نیاز انجام می‌دهد. بنابراین ابزار سنتز به دنبال یافتن سریع‌ترین روش برای ایجاد پنجره منطق‌سازی نیست، بلکه به دنبال چگونگی پیاده‌سازی طرح است که با شروط زمانی طرح دقیقاً سازگاری داشته باشد، آنگاه به کم کردن حجم مصرفی می‌پردازد. بعد از آن، کار بهینه‌سازی نگاشت در این پنجره کوچک منطق‌سازی انجام می‌گیرد. عمل بهینه سازی نگاشت از طریق حرکت دادن این پنجره، برای تمام منطق‌مدار اجرا می‌شود. این کار تا زمانی که شروط زمانی مدار محقق شود ادامه می‌یابد. در بعضی از ابزارها این امکان وجود دارد که سطح تلاشی که ابزار سنتز در حین نگاشت به خود وارد می‌سازد را تعیین نمود. به طور معمول سه سطح تلاش وجود دارد که عبارتند از : پایین، متوسط و بالا که می‌توان یکی را از میان آنها انتخاب نمود.



شکل ۴-۱۰ نگاشت تکنولوژی

در بهینه‌سازی‌های معمولی، ابزار سنتز کار بهینه‌سازی را با رعایت محدودیتها انجام می‌دهد. نمایش این روند بر روی محورهای مکان و زمان به صورت یک شیب ملایم خواهد بود. به این معنا که هر چه شروط زمانی بیشتر باشد طرح بزرگ‌تر خواهد شد و بالعکس (شکل ۵-۱۰).

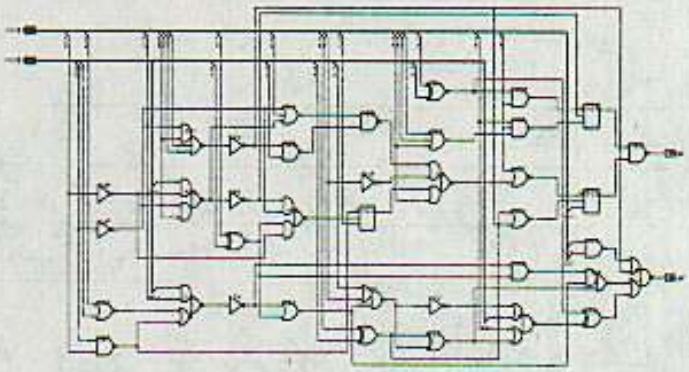


شکل ۵-۱۰ رابطه بین مکان و زمان

چنان‌چه محدودیتهای زمانی به طور نادرست برای ابزار سنتز تعیین شوند، به نقطه نامناسبی در منحنی فوق می‌رسیم. اگر کد VHDL زیر را با اهداف گوناگون بهینه‌سازی سنتز کنیم نتایج حاصل بسیار با هم متفاوت خواهند بود.

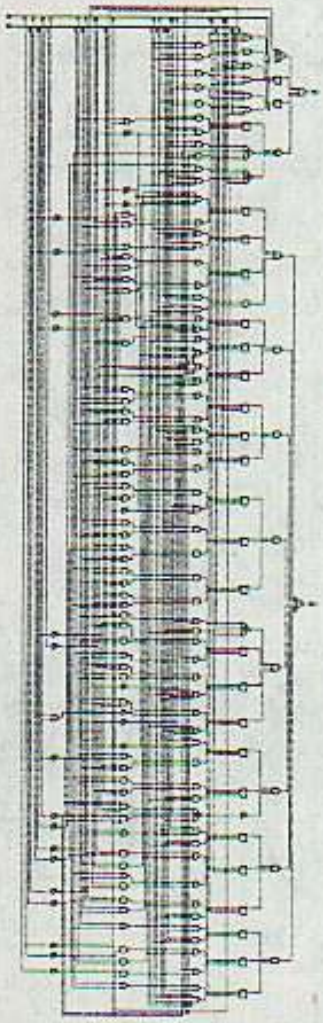
```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity syn is
port (a,b:        in  std_logic_vector (4 downto 0);
      q1,q2:      out std_logic);
end;

Architecture alt1 of syn is
signal i1,i2:std_logic;
begin
   i1<= '1' when a+b<12 else '0';
   i2<= '1' when a=10 else '0';
   q2<=i1 or i2;
   process(a,b)
   variable int:integer range 0 to 63;
   begin
        int:=conv_integer(a);
        case int is
          when 2|5          => q1<=a(0) and b(0);
          when 3|4          => q1<=a(1) and b(1);
          when 1|6          => q1<=a(0) and b(2);
          when 8 to 10      => q1<=a(0) and b(3);
          when 11 to 24     => q1<=a(0) and b(4);
          when others       => q1<=a(0) xor b(0);
        end case;
   end process;
end;
```

اما اگر کوچک‌ترین فضای مصرفی ممکن برای بهینه‌سازی کد فوق منظور گردد، نتیجه سنتز مطابق شکل ۶-۱۰ خواهد شد.



شکل ۶-۱۰ نتیجه سنتز برای پیاده‌سازی در کمترین فضا

اگر کد VHDL فوق را به نحوی بهینه‌سازی کنیم که سریع‌ترین اجرای ممکن را داشته باشد. شماتیک آن مطابق شکل ۷-۱۰ خواهد بود.



شکل ۷-۱۰ نتیجه سنتز برای پیاده‌سازی در کمترین زمان

چنان‌چه نتایج حاصل را با هم مقایسه کنیم جدول زیر به دست می‌آید.

	تعداد گیت‌ها	سرعت (ns)
با در نظر گرفتن محدودیتهای زمانی	487	4.5
با در نظر گرفتن محدودیتهای مکانی	69	10.8

همان طور که به وضوح مشاهده می‌شود تفاوت حاصل از پیاده‌سازی‌های گوناگون بسیار چشمگیر است. این موضوع نشان می‌دهد که تعیین تنگناها و محدودیتهای زمانی برای ابزار سنتز چه اندازه اهمیت دارد.

۲-۱۰ محدودیتها

کلیه ابزارهای پیشرفته سنتز برای بهینه‌سازی بر اساس محدودیتهای مکانی و زمانی ساخته شده‌اند. در زیر مثالی از چگونگی تنظیم محدودیتها در سنتزگر Synopsys آورده شده است. به طور کلی اصول در همه ابزارهای سنتز یکسان است و تفاوت تنها در تنوع دستورالعملهای مربوط به محدودیتها می‌باشد.

می‌توان محدودیتها را به سه گروه اصلی تقسیم کرد :

- قواعد طراحی
- محدودیتهای زمانی
- محدودیتهای مکانی

از هیچ ابزار سنتزی نمی‌توان انتظار معجزه داشت. اگر محدودیتها و تنگناهای تعیین شده نتوانند به طور هم‌زمان برآورده شوند، خود اقدام به اولویت‌بندی مطابق با ترتیب فوق خواهد کرد. ترتیب اولویتها را به سادگی می‌توان تغییر داد. معمولاً محدودیتهای زمانی در اولویت بالاتری نسبت به محدودیتهای مکانی قرار دارند. مثلاً اگر محدودیت ۴۰ مگاهرتز در نظر گرفته شده است، به کارگیری طرحی که تا حد ۳۸ مگاهرتز کار کند بی‌فایده خواهد بود. از سوی دیگر اگر محدودیت مکانی پاسخگو نباشد باید مدار بزرگ‌تری انتخاب شود. البته این امر موجب افزایش قیمت اجزای تشکیل‌دهنده خواهد شد. بنابراین راه حل دیگر می‌تواند تعویض تکنولوژی (معمولاً گران‌تر)، طراحی مجدد طرح و مدار (صرف زمان بیشتر) و یا کاهش انتظارات اجرایی مدار باشد.

چنان‌چه طرح تماماً سنکرون باشد، فقط محدودیتهای زمانی برای پایه‌های I/O در سطح بالا باید مشخص گردد.

موارد زیر همواره باید مشخص شوند :

- پالس‌های ساعت ورودی و دوره تناوب آنها
- تأخیر ورودی در ارتباط با پالس ساعت
- تأخیر خروجی در ارتباط با پالس ساعت
- هر گونه تأخیر پایه ـ به ـ پایه (منطق ترکیبی از ورودی تا خروجی)
- هر مسیر نادرست

چنان‌چه ابزار سنتز باید قادر به محاسبه زمان‌بندی و بهینه‌سازی مناسب باشد لازم است موارد زیر نیز برای آن مشخص گردد :

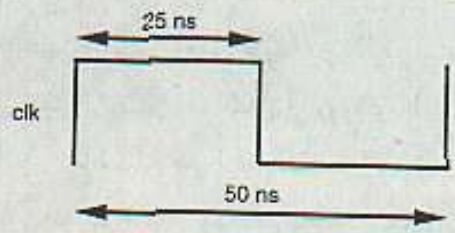
- منبع تغذیه
- گستره دما
- مدل سیمی بار

چنان‌چه مدار بزرگ باشد (بیشتر از ۲۰۰۰۰ گیت)، بلوک‌های به کار رفته در طراحی سلسله مراتبی معمولاً باید در ابتدا به طور جداگانه بهینه‌سازی شوند. در چنین حالتی همان محدودیتهای گفته شده در فوق باید برای تمام ورودی‌ها و خروجی‌های بلوک مشخص شوند.

۱-۲-۱۰ تعیین پالس‌های ساعت ورودی

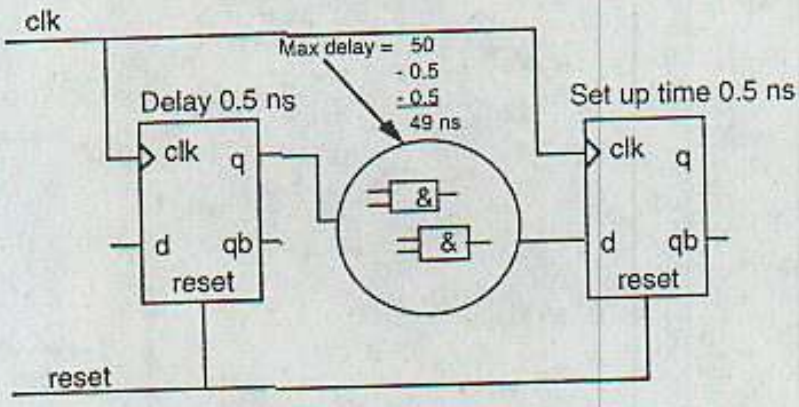
تعیین کلاک بسیار ساده است. به طور مثال در Synopsys دستور زیر (شکل ۸-۱۰) را می‌توان به کار برد.

```
create_clock -period 50 -waveform [0 25] clk
```



شکل ۸-۱۰ تعریف پالس ساعت

همچنین می‌توان با تعیین پایه مربوط به پالس ساعت و انتخاب نحوه کار آن از منوی لیست شده، آن را مشخص و تعریف نمود. سایر ابزارهای پیشرفته سنتز به همین نحو عمل می‌کنند و فقط ممکن است دستورالعملها کمی فرق داشته باشد. چنان‌چه سیستم چندین کلاک داشته باشد همه آنها را می‌توان به روش بالا مشخص و تعریف نمود. سپس ابزار سنتز به طور خودکار به بررسی محدودیتهای زمانی بین کلیه فلیپ فلاپهای داخلی می‌پردازد. اگر دوره تناوب کلاک را ۵۰ نانوثانیه تعیین کنیم، ابزار سنتز منطق ترکیبی بین همه فلیپ فلاپ‌های تحریک شده با این کلاک را بررسی می‌کند که بفهمد آیا با کسر زمان تأخیر هر یک از فلیپ فلاپ‌ها و همچنین زمان راه‌اندازی فلیپ فلاپ‌های بعدی می‌تواند ماکزیمم تأخیر ۵۰ نانوثانیه را ایجاد کند یا خیر (شکل ۹-۱۰).



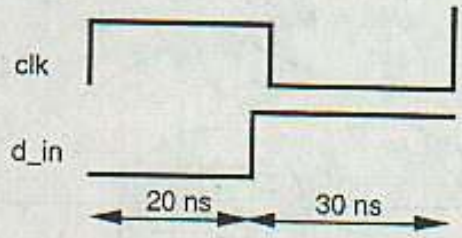
شکل ۹-۱۰ محدودیتهای زمانی

چنان‌چه سیستم دارای چندین کلاک مرتبط با هم باشد (به طور مثال ضریبی از یکدیگر باشند)، ابزار سنتز به طور اتوماتیک به محاسبه رابطه بین آنها و چگونگی ساختن آنها با کمک منطق ترکیبی می‌پردازد. اما اگر رابطه‌ای بین کلاک‌ها وجود نداشته باشد، ابزار سنتز هر یک از آنها را جداگانه می‌سازد.

۲-۲-۱۰ تعیین تأخیر ورودی و خروجی

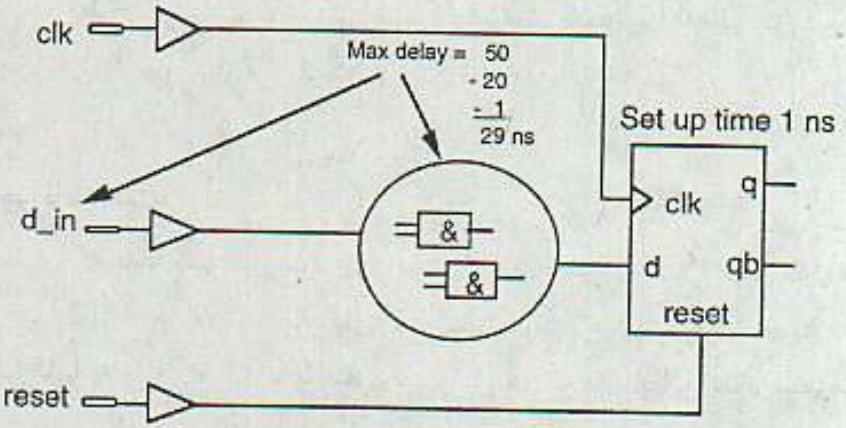
فاصله زمانی که سیگنال‌های ورودی نیاز دارند تا در روی پایه‌های ورودی - خروجی در ارتباط با زمان پالس ساعت سیستم، به حالت پایدار برسند باید برای ابزار سنتز مشخص و تعریف شده باشد. در ضمن ابزار سنتز باید حداکثر زمان رسیدن سیگنال از ورودی تا فلیپ فلاپ‌های منطق ترکیبی را بداند. فرض کنید تأخیر ورودی ۲۰ نانوثانیه در ارتباط با کلاک (با دوره تناوب ۵۰ نانوثانیه) برای سیستم تعریف شده باشد، در این صورت فاصله زمانی رسیدن سیگنال از ورودی تا فلیپ فلاپ ۳۰ = ۲۰ - ۵۰ نانوثانیه خواهد بود (شکل ۱۰-۱۰).

```
set_input_delay 20 -clock clk d_in
```



شکل ۱۰-۱۰ محدودیت تأخیر ورودی

از آنجایی که به طور نرمال شرایط هر ورودی با دیگر ورودی‌ها متفاوت خواهد بود، تأخیر ورودی برای هر مدار باید به طور جداگانه معین شود (شکل ۱۱-۱۰).

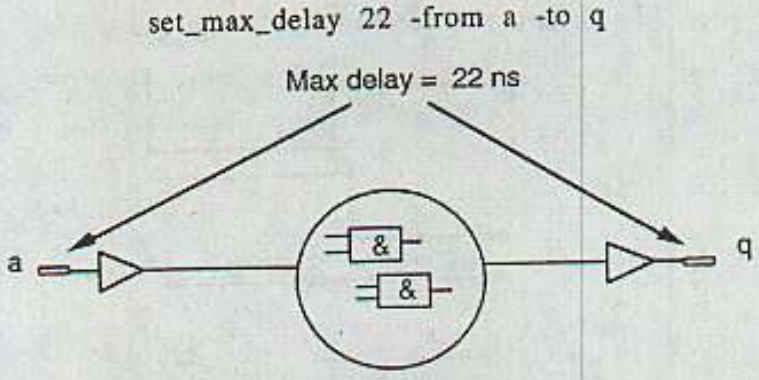


شکل ۱۱-۱۰ محدودیت تأخیر ورودی

دقیقاً مشابه ورودی‌ها، فاصله زمانی در دسترس قرار گرفتن خروجی‌ها در ارتباط با زمان پالس ساعت سیستم (محدودیتهای تأخیر خروجی) نیز باید برای ابزار سنتز مشخص شود. اگر ۲۸ نانوثانیه به عنوان تأخیر خروجی در نظر بگیریم، حداکثر فاصله زمانی بین کلاک فلیپ فلاپ داخلی تا خروجی باید ۲۲ = ۲۸ - ۵۰ نانوثانیه باشد.

چنان‌چه از ورودی تا خروجی فقط مسیرهای ترکیبی خالص (بدون فلیپ فلاپ) وجود داشته باشد، حداکثر تأخیر با آنچه به ***زمان‌بندی نقطه به نقطه***[1] معروف است مشخص می‌شود (شکل ۱۲-۱۰).

1- Point-To-Point Timing



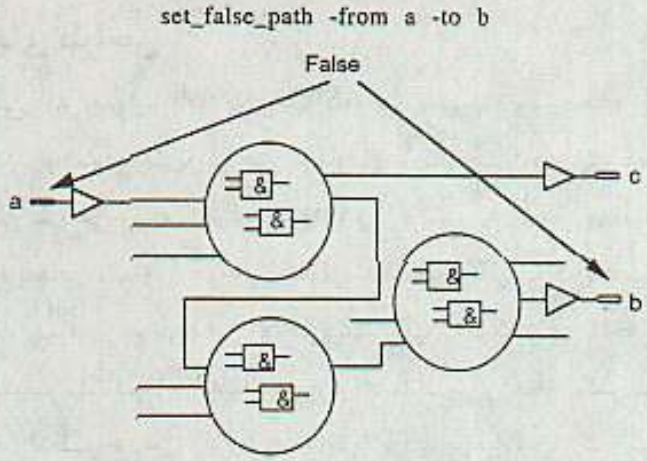
شکل ۱۲-۱۰ زمان‌بندی نقطه به نقطه

البته می‌توان ورودی‌ها و خروجی‌ها را به یک کلاک مرتبط ساخت و سپس شروط زمانی را در ارتباط با آن کلاک مطرح کرد. این روش کار بسیار توصیه می‌شود زیرا که در موارد پیچیده، تصویر واضحی از شروط زمانی را در اختیار قرار می‌دهد.

۳-۲-۱۰ مسیر نادرست

نکته دیگری که باید مشخص شود این است که آیا مسیر نادرستی برای داده‌ها در طراحی وجود دارد یا خیر. ASIC ها معمولاً پین‌های سنجشی دارند که مدار را در جریان ساخت تست می‌کنند. این پین‌ها یا به ولتاژ ۵ ولت و یا صفر ولت وصل می‌شوند. بنابراین تمام مسیرهای زمانی که از پین سنجش شروع می‌شوند باید به عنوان مسیر نادرست داده‌ها تعریف شوند، زیرا در غیر این صورت ابزار سنتز این مسیرها را از نقطه نظر زمانی مورد بهینه‌سازی قرار می‌دهد بی‌آنکه نیازی به این کار باشد. بعضی اوقات، مسیرهایی در مدار هست که طراح می‌داند هیچ گاه در حالت عملکردی به وجود نمی‌آیند. این نوع مسیرهای داده نیز باید مشخص شوند. تمام مسیرهای داده داخلی تا حد امکان توسط ابزار سنتز محاسبه می‌شوند و تنها آن مسیرهایی که طراح می‌داند به خاطر کاربرد یا محیط مدار نادرست هستند باید مشخص شوند و بقیه مسیرها را ابزار سنتز کنترل خواهد کرد.

فرض کنید به دلیل شرایط خارجی مدار می‌دانیم که مسیر داده از ورودی a به خروجی b هیچ‌گاه در حین استفاده عادی مدار تشکیل نخواهد شد (به شکل زیر توجه کنید). از این رو این مسیر داده به عنوان مسیر نادرست مشخص می‌شود تا بی‌جهت از لحاظ زمانی مورد بهینه‌سازی قرار نگیرد.



شکل ۱۳-۱۰ محدودیت مسیر نادرست

این نکته خصوصاً در مورد طراحی با FPGA مهم است که هر مسیر داده نادرست مشخص و ذکر شود زیرا بخش سیم‌بندی برای پایین آوردن سطح اولویت مسیرهای داده به این اطلاعات احتیاج دارد. اگر اولویت بخش سیم‌بندی برای یک مسیر داده نادرست در سطح «سریع» انتخاب شود، این عمل به دلیل محدود بودن منابع سیم‌بندی در مدار FPGA معمولاً به بهای کندتر شدن یک مسیر داده دیگر تمام خواهد شد. همین اصل در مورد ASIC ها نیز مطرح است اما اثرات آن در ASIC معمولاً به آن شدت نخواهد بود، زیرا عمل سیم‌بندی در یک ASIC در مقایسه با بسیاری از FPGA ها از لحاظ درصد کل تأخیر، وقت بسیار کمتری می‌گیرد. بنابراین به دلیل کوچک بودن عرض خطوط (مسیرها) در تکنولوژی ASIC (کمتر از ۰/۶ میکرومتر)، تأخیر سیم‌بندی با همان درجه از شدت که در تأخیرهای گیتی در مدار مشاهده می‌شود آغاز خواهد شد.

۴-۲-۱۰ محدودیتهای مکانی

بعد از آنکه کلیه محدودیتهای زمانی برای مدار معین شد، مرحله بعدی تعیین محدودیتهای مکانی است. این محدودیت معمولاً با یک گیت NAND مشخص می‌شود.

مثال :

```
set_max_area  20000
```

چنان‌چه این محدودیت تعیین نشود، ابزارهای سنتز پس از برآورده کردن شرطهای لازم مدار تلاش برای بهینه‌سازی فضای مدار نخواهند کرد.

۵-۲-۱۰ محدودیتهای طراحی

محدودیت سومی که همواره باید برای ASIC ها تعیین شود، محدویت طراحی است. تولید کنندگان ASIC هر کدام محدودیتهای طراحی متفاوتی دارند. چنان‌چه این محدودیتها رعایت نشوند سازندگان ASIC ، شبکه طراحی شده را برگشت داده و تا وقتی که اشکال برطرف نشده هیچ مداری را نخواهند ساخت. محدودیتهای متداول طراحی که توسط سازندگان ASIC مطرح می‌شود شامل زمان اوج و فرود سیگنال‌ها، میزان جریان‌دهی و ظرفیت کلیه شبکه‌های داخلی است. چنان‌چه این محدودیتها که برای عملیات بهینه‌سازی بسیار ضروری هستند برای ابزار سنتز مشخص نشوند مشکلات بزرگی به وجود می‌آید. تنظیم دستی این محدودیتها برای مدارات بزرگ اگر ناممکن نباشد بسیار مشکل خواهد بود.

مثال زیر را به عنوان نمونه نگاه می‌کنیم :

```
set_max_capacitiance      10
set_max_transition        5
set_max_fanout            4
```

از آنجایی که کلیه زمان‌بندی‌ها به ولتاژ تغذیه، دمای سیستم و فاکتور عملکرد بستگی دارند، بدین جهت باید کلیه این موارد نیز برای سیستم معرفی و تعیین شوند. معمولاً گفته می‌شود مدار تحت ولتاژ تغذیه ۴/۵-۵/۵ ولت کار می‌کند و دمای دستگاه بین ۸۵-۰ درجه سانتی‌گراد بوده و باید فاکتور عملکرد یک معیار مشخص را داشته باشد. معیار فاکتور عمل در زمان‌بندی ماکزیمم بدترین حالت و در زمان‌بندی مینیمم بهترین حالت خوانده می‌شود. آنچه که به عنوان مدل سیمی بار معروف است نیز باید مشخص و تعیین شود. ابزار سنتز به مدل سیمی بار جهت برآورد ظرفیت نیاز دارد. این مدل مشخص می‌کند که شبکه داخلی با توجه به ***جریان‌دهی خروجی***[1]، چه ظرفیتی باید داشته باشد. آنچه که به طور معمول باید انجام شود تخمین اندازه و حجم مدار نهایی و انتخاب مدل بار است.

مثال :

```
Set_operating_conditions WCCOM (worst case commerical 0-70°C,
                                4.75-5.25V)
Set_wire_load 20000             (gates)
```

1- Fan-out

۳-۱۰ بهینه‌سازی بهترین حالت

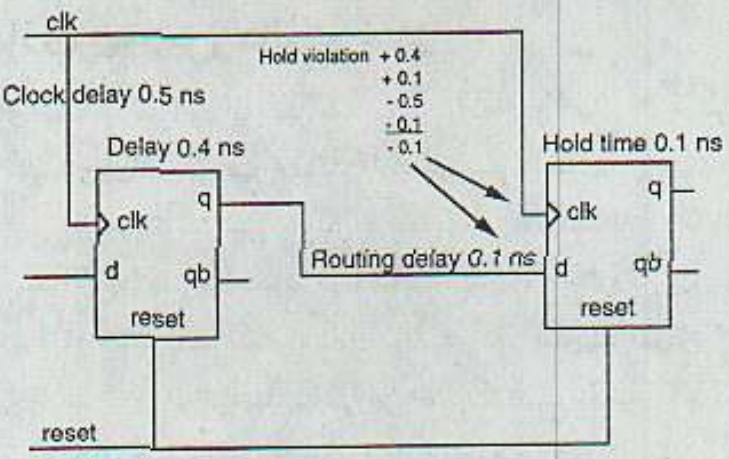
در طراحی‌های ASIC زمان حداقل نیز باید مورد توجه قرار گیرد. کلیه شروط زمانی که قبلاً شرح داده شد می‌توانند به جای محدودیت حداکثر با محدودیت حداقل نیز تنظیم شوند. به طور مثال ممکن است در بخشی از طراحی تأخیر بین ورودی a تا خروجی b بیشتر از ۱۰ نانوثانیه و کمتر از ۳۰ نانوثانیه تعریف شود. بنابراین بزرگ‌ترین مسأله در وضعیت بهترین حالت (کمترین زمان‌بندی) مشکل **مکث**[1] است. فرض کنید که شیب پالس ساعت ۰/۵ نانوثانیه باشد، همچنین فرض کنید که تأخیر گذر سیگنال از یک فلیپ فلاپ ۰/۴ نانوثانیه باشد. در این صورت فلیپ فلاپ، یک مکث به مقدار ۰/۱ نانوثانیه و ارتباط درونی، یک تأخیر سیم‌بندی برابر ۰/۱ نانوثانیه خواهند داشت. این بدان معناست که ورودی D فلیپ فلاپ می‌تواند هم‌زمان با کلاک تغییر کند، اما از آنجا که محدودیت مکث برابر ۰/۱ نانوثانیه است، این کار قابل قبول نخواهد بود (به شکل ۱۴-۱۰ مراجعه کنید).

راه حل این مشکل این است که اجازه دهیم ابزار سنتز نیز کار بهینه‌سازی خود را در ارتباط با وضعیت بهترین زمان‌بندی انجام دهد. در ابتدا ابزار سنتز را باید reset کرد تا بتواند با پروسه بهترین حالت کار کند. محدودیت تعریف شده برای ابزار سنتز می‌تواند شیب کلاک ۰/۶ نانوثانیه بین کلیه فلیپ فلاپ‌ها باشد. در ضمن زمان در دسترس بودن زودترین (اولین) سیگنال ورودی نیز باید مشخص شود، زیرا که مسأله مکث می‌تواند در اولین فلیپ فلاپ ورودی نیز مطرح شود.

مثال :

```
Set_operating_condition BCCOM (best case commerical)
Set_clock_skew   -uncertainty 0.6 clk
Set_fix_hold clk
Set_input_delay 2  -min   -clock clk d_in
```

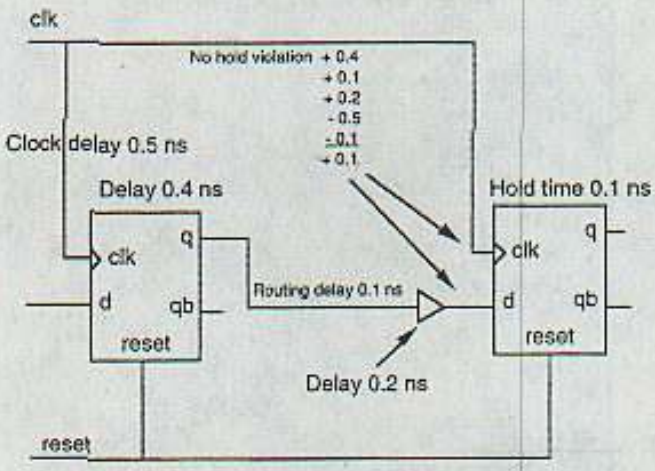
1- Hold



شکل ۱۴-۱۰ تناقض زمان مکث

پروسه بهینه‌سازی را همچنین باید زمانی که تمام محدودیتهای زمانی به بهترین شکل برآورده شده باشند انجام داد. در پروسه بهینه‌سازی تحت بدترین شکل زمان‌بندی، این خطر وجود دارد که عمل بهینه‌سازی باعث اختلال در بدترین حالت زمان‌بندی گردد. اگر ابزار سنتز کار خود را با تغییر گیت‌هایی که به دلیل وجود مسأله مکث نباید تغییر کنند، آغاز کند مشکل به‌وجود می‌آید. روش Synopsys برای روبه‌رو شدن با این مشکل این است که پروسه بهینه‌سازی تحت بهترین شکل را طوری آغاز کند که ابزار سنتز مجاز به تغییر مسیرهای داده‌ای که از محدودیتهای مکث تجاوز نکرده‌اند، نباشد.

ابزار سنتز در این نوع بهینه‌سازی می‌تواند با ایجاد تأخیر در مسیر عبور سیگنال، از طریق قرار دادن یک بافر بین فلیپ فلاپ‌ها، مسأله مکث را حل کند (شکل ۱۵-۱۰).



شکل ۱۵-۱۰ حل شدن مشکل مکث

```
Set_operating_condition BCCOM
Set_max_clock_skew   -uncertainty 0.6
Set_fix_hold clk
compile  -priortize_min_path  -only_design_rule
```

۱۰-٤ اگر ابزار سنتز به اهداف بهینه‌سازی دست نیابد چه باید کرد ؟

چنان‌چه در مورد اهداف تعیین شده بهینه‌سازی مشکلاتی به وجود آید، گزینه‌های زیر برای انتخاب در اختیار طراح خواهد بود :

۱- امتحان کردن یک الگوریتم بهینه‌سازی دیگر در ابزار سنتز
۲- تغییر کدگذاری حالت (تنها در ماشینهای حالت)
۳- بازنویسی کد VHDL
۴- افزایش گستره ولتاژ تغذیه مجاز مدار
۵- کاهش گستره دمایی عملکرد مدار
۶- تغییر تکنولوژی
۷- عدم پوشش کلیه خطاها (مخصوصاً در مشکلات مربوط به فضا)
۸- به کارگیری ابزار سنتز بهتر
۹- بهینه‌سازی دستی منطق برنامه
۱۰- تغییر اهداف بهینه‌سازی

توصیه می‌شود آزمایش راه‌حلهای فوق به همان ترتیبی که در بالا گفته شده انجام شود.

راه حل اول

ابزارهای سنتز پیشرفته الگوریتم‌های گوناگونی برای بهینه‌سازی دارند. در ابتدای فصل تفاوتهای آنها نشان داده و گفته شد که با تعیین محدودیتهای مختلف زمان و فضا و استفاده از الگوریتم‌های مختلف بهینه‌سازی می‌توان به این تفاوتها دست یافت. محدودیتهای زمان و مکان برای دو الگوریتم مختلف بهینه‌سازی بسیار متفاوت خواهد بود. البته کلیه ابزارهای سنتز از لحاظ اینکه در چهارچوب یک تکنولوژی خاص و با در نظر گرفتن عاملهای زمان و مکان چقدر قادر به بهینه‌سازی خواهند بود محدودیتهایی دارند، بنابراین راه حل اول نمی‌تواند همیشه مفید واقع شود.

راه حل دوم

چنان‌چه مشکل زمان یا مکان در یک ماشین حالت باشد می‌توان کدگذاری حالت را با به کارگیری بهینه‌ساز ماشین حالت که در ابزارهای سنتز پیشرفته منظور شده است تغییر داد. این کار می‌تواند تأثیر قابل ملاحظه‌ای بر زمان و فضا بگذارد (به فصل ۹ مراجعه کنید، «ماشینهای حالت»).

راه حل سوم

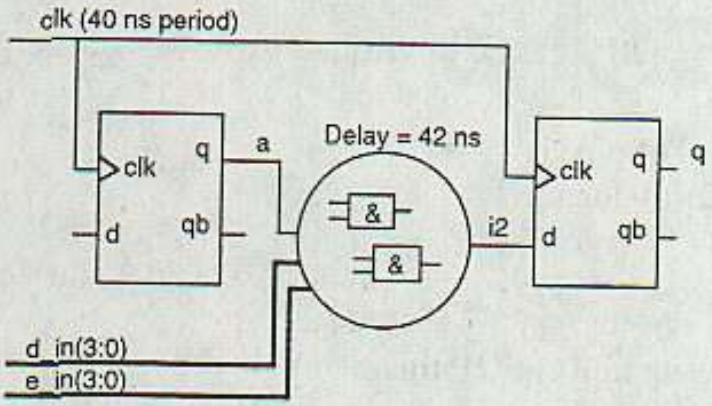
چنان‌چه لازم باشد کد VHDL بازنویسی شود می‌توان یکی از گزینه‌های زیر را انتخاب کرد :

- تغییر کد VHDL بدون تغییر عملکرد آن
- تغییر عملکرد کد VHDL

غالباً این امکان وجود دارد که کد VHDL را بدون ایجاد تغییر در عملکرد آن تغییر داد و آن را بازنویسی کرد. یک ابزار پیشرفته سنتز به طور هم‌زمان می‌تواند به همان نتایج عملکردی قبلی بدون توجه به روش توصیف کد VHDL، برای بلوک‌های ترکیبی کوچک برسد. چنان‌چه بلوک ترکیبی پیچیده باشد، نتیجه سنتز با توجه به روشی که کد VHDL آن را توصیف می‌کند متفاوت خواهد بود. به طور مثال، نتیجه سنتز در صورت به کارگیری دستور if-then-else با نتیجه حاصل از به کارگیری دستور case متفاوت خواهد بود. به طور کلی می‌توان گفت که در طراحی‌های پیچیده نتیجه دستور case سریع‌تر اما حجیم‌تر از نتیجه دستور if-then-else است.

از آنجایی که کد VHDL باید قبل از مرحله سنتز مورد بررسی قرار گیرد، چنان‌چه تغییری در کد VHDL داده شود باید دوباره آن را بررسی کرد. شبیه‌سازی مجدد طرح معمولاً بسیار وقت‌گیر است، بنابراین توصیه می‌شود برای چک کردن و اثبات صحت عمل از ابزار بررسی و تست استفاده شود. هنگام استفاده از ابزار تست صحت عمل (به فصل ۱۱، «آشنایی با روشهای طراحی» مراجعه شود)، می‌توان کد قدیمی و سنتز شده VHDL را با کد جدید مورد مقایسه قرار داد. اگر نتیجه از لحاظ وظایف و عملکرد حاکی از یکسان بودن آنها باشد دیگر نیازی به شبیه‌سازی مجدد طرح نخواهد بود. این مقایسه فقط در مورد یک بلوک از طراحی سلسله مراتبی که در آن کد VHDL تغییر یافته است باید انجام گیرد و به این ترتیب این روش اجازه می‌دهد روشهای توصیفی مختلف در کد VHDL به سرعت مورد ارزیابی قرار گیرند. در استدلال فوق فرض بر این است که کد VHDL نیازی ندارد که از لحاظ عملکرد تغییر یابد. لازم به ذکر است که اینک ابزارهایی در حال عرضه به بازار هستند که می‌توانند به طراح برای نوشتن کد VHDL مطابق با هر هدف طراحی خاص کمک مؤثری کنند.

چنان‌چه با این کار باز هم به اهداف بهینه‌سازی نرسیم، می‌توانیم کد VHDL را از لحاظ عملکرد اصلاح کنیم. مثلاً اگر مشکل محدودیتهای زمانی بر سر راه باشد، طراح باید در گام اول کار خود را با تحلیل دقیق مشکل زمانی برای شناخت بهتر آن آغاز کند. این عمل به راحتی از طریق دریافت گزارش زمانی از ابزار سنتز انجام خواهد شد. یکی از طرق حل این گونه مسائل ***خط لوله‌ای***[1] کردن مسیر عبور داده‌ها است.



شکل ۱۶-۱۰ طرح بدون اعمال مکانیسم خط لوله

فرض کنید که شکل ۱۶-۱۰ بیانگر نتیجه سنتز کد VHDL زیر باشد :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity No_pipe is
port (clk,d:      in  std_logic;
      q:          out std_logic);
end;

Architecture rt1 of No_pipe is
signal a,i1,i2:std_logic;
begin
  process(clk)
  begin
      if clk'event and clk= '1' then
```

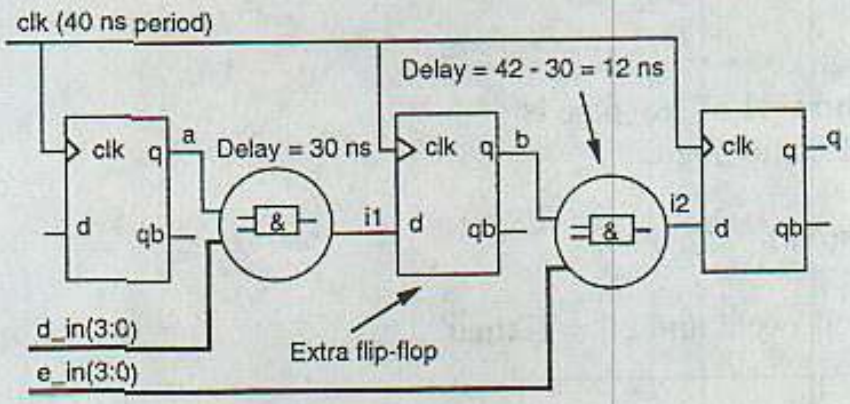
1- Pipeline

```
            a<=d;
            q<=i2;
        end if;
    end process;
    i1<= '1' when d_in= "1100" else '0';
    i2<= '1' when i1= '1' and e_in= "0101" else '0';
end;
```

اگر از مکانیسم خط لوله استفاده شود، کد VHDL زیر به دست می‌آید :

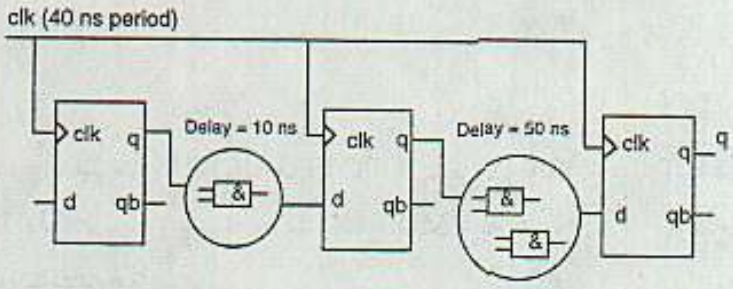
```
Architecture rtl of pipe is
signal a,b,i1,i2:std_logic;
begin
    process(clk)
    begin
        if clk'event and clk= '1' then
            a<=d;
            b<=i1;
            q<=i2;
        end if;
    end process;
    i1<= '1' when d_in= "1100" else '0';
    i2<= '1' when b= '1' and e_in= "0101" else '0';
end;
```

عیب این روش ایجاد تأخیر به اندازه یک پالس ساعت در مسیر داده است. نتیجه سنتز در شکل ۱۷-۱۰ نشان داده شده است.



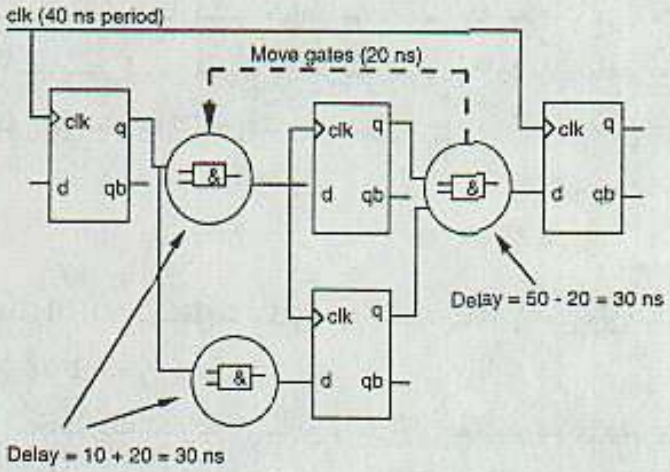
شکل ۱۷-۱۰ طرح با اعمال مکانیسم خط لوله

راه‌حل دیگر که ***موازنه رجیستری***[1] خوانده می‌شود می‌تواند مفید باشد. این کار در سطح ***RT***[2] با بعضی از ابزارهای پیشرفته سنتز قابل اجرا خواهد بود. برای نمونه طرحی با موازنه رجیستری و بدون آن در شکلهای ۱۸-۱۰ و ۱۹-۱۰ نشان داده شده است.



شکل ۱۸-۱۰ طرح بدون موازنه رجیستری

تعداد فلیپ فلاپ‌ها با به کارگیری موازنه رجیستری تغییر خواهد کرد.



شکل ۱۹-۱۰ طرح با موازنه رجیستری

چنان‌چه عمل موازنه رجیستری توسط ابزار سنتز انجام شود دیگر نیازی به شبیه‌سازی مجدد مدار نخواهد بود. از سوی دیگر می‌توانیم با استفاده از شیوه‌های دستی آزمایش تأیید کنیم که مدار از لحاظ عملکرد در قبل و بعد از موازنه رجیستری یکسان باقی مانده است. این شیوه‌های آزمایش و تأیید در ساخت بعضی از ابزارهای پیشرفته سنتز منظور شده و از این رو استفاده از آنها خیلی سریع و آسان

1- Register Balance

2- Register Transfer

است. وقتی به عمل موازنه رجیستری اقدام می‌کنیم باید آن را به دقت ثبت کنیم زیرا همان طور که قبلاً گفتیم تعداد فلیپ فلاپ‌ها قابل تغییر هستند. در مورد کد VHDL در سطح RT تعداد فلیپ فلاپ‌ها معمولاً به وسیله کد VHDL معین می‌شود. این قاعده آن بخش از طرح را که روی آن موازنه رجیستری انجام شده، شامل نمی‌شود.

راه حل چهارم

نحوه زمان‌بندی مدارات ASIC/FPGA CMOS تابعی از ولتاژ تغذیه آنها است. مثلاً اگر ولتاژ تغذیه بین ۵/۵-۴/۵ ولت باشد، زمان‌بندی مدارات CMOS به میزان ۳۰-۱۰ درصد نسبت به ولتاژ تغذیه ۵/۲۵-۴/۷۵ ولت بهتر خواهد بود.

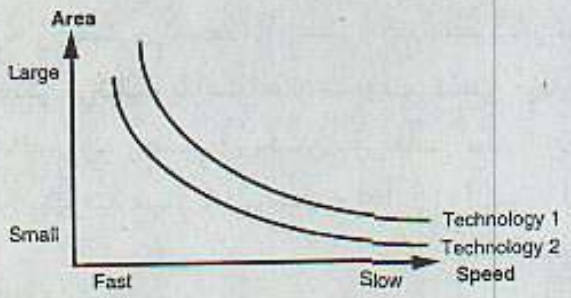
راه حل پنجم

نحوه زمان‌بندی مدارات ASIC/FPGA CMOS تابعی از دمای سیلیکون است. هر چه دمای سیلیکون بیشتر باشد، سرعت گیت‌ها کمتر خواهد بود. چنان‌چه سرعت در دمای اتاق (۲۵°C) را با سرعت در دمای ۷۰°C مقایسه کنیم، تفاوت با ضریب تقریبی ۲-۱/۵ خواهد بود. با کاهش ماکزیمم دمای مجاز برای سیلیکون می‌توان به نتایج بهتری رسید.

راه حل ششم

از آنجایی که کد VHDL مستقل از تکنولوژی است، بنابراین می‌توان تکنولوژی‌های گوناگونی را برای رسیدن به نتایج بهتر آزمایش کرد.

به طور مثال با انتخاب یک تکنولوژی سریع‌تر می‌توان به زمان‌بندی بهتری دست یافت، این در حالی است که به علت بزرگ‌تر شدن حجم مدار مجبور به انتخاب المان‌های گران‌قیمت‌تر خواهیم بود.



شکل ۲۰-۱۰ رابطه بین زمان و فضا

راه حل هفتم

چنان‌چه مشکل فضا مطرح باشد باید شیوه ***اسکن کامل***[1] را انتخاب نمود. اسکن کامل به معنای این است که کلیه المان‌های حافظه در طرح موردنظر، با معادل اسکن آنها تعویض و وارد چرخه اسکن شوند. این نوع اسکن به فلیپ فلاپ‌های بیشتری نیاز دارد. تعداد گیت‌ها را می‌توان با خارج کردن فلیپ فلاپ‌ها از چرخه‌های اسکن کاهش داد اما این کار به قیمت افزایش سطح خطا همراه خواهد بود. برای جزئیات بیشتر به فصل ۱۲، «آشنایی با روشهای آزمایش» مراجعه کنید.

راه حل هشتم

اگر از بهترین ابزار سنتز استفاده نمی‌شود، تغییر آن به نوع پیشرفته‌تر می‌تواند راه حل مناسبی باشد، منتهی این ریسک وجود دارد که مجبور شویم کد VHDL را مختصراً باز نویسی کنیم زیرا اصول و قواعد در یک ابزار با ابزار دیگر تفاوت دارد. چون برای این کار نیاز به خرید یک ابزار سنتز جدید خواهیم داشت این گزینه راه‌حل گران قیمتی خواهد بود. اگر طراحی‌های ASIC/FPGA به طور منظم انجام گیرد، به دلیل آنکه منجر به اشغال فضای کمتر و اجرای بهتر خواهند شد، هزینه ابزار غالباً جبران می‌شود. ابزارهای گوناگون سنتز می‌توانند تا ۴۰-۳۰ درصد برای زمان و فضا مؤثر باشند.

راه حل نهم

چنان‌چه از یک ابزار سنتز ضعیف استفاده می‌شود، گاهی اوقات بهتر است که بهینه‌سازی نتیجه سنتز به طور دستی انجام شود که بی‌شک بسیار وقت‌گیر خواهد بود. بنابراین توصیه می‌شود که در صورت امکان از یک ابزار پیشرفته‌تر استفاده شود. اما اگر از یک ابزار مناسب استفاده می‌شود دیگر نیازی به بهینه‌سازی دستی نیست مگر آنکه اطلاعات کافی در مورد چگونگی استفاده مؤثر از ابزار وجود نداشته باشد. این امکان نیز وجود دارد که تمام netlist هایی را که یک ابزار سنتز تولید می‌کند به طور دستی به وجود آورد، اما اشکال اصلی این است که این روش بسیار وقت‌گیر است و ضمناً همواره نمی‌توان همان نتایجی را به دست آورد که از طریق ابزار سنتز ممکن است به دست آید.

در مواقعی که طراحی یک ASIC با بهترین کیفیت مدنظر باشد شرایط تا حدودی فرق خواهد کرد. طراح در این مورد می‌تواند به منظور بهبود زمان‌بندی و فضای مصرفی شبکه، در مسیرهای عبور دیتا تعدادی ***ماکرو***[2]های مخصوص تعریف کند. ماکروها باید در کتابخانه ذخیره شوند تا ابزار سنتز

1- Full Scan

2- Macro

بتواند از آنها استفاده کند. این روش نیز تا حدودی وقت‌گیر است اما بعضی اوقات می‌تواند مشکل را حل کند.

راه حل دهم

به عنوان آخرین راه حل می‌توان اقدام به کاهش کیفیت اجرا یا محدودیتهای مکانی مدار نمود. در بعضی پروژه‌ها کاسته شدن از کیفیت اجرا در یک بخش را می‌توان با افزایش کیفیت اجرا در بخشهای دیگر سیستم جبران نمود.

۵-۱۰ خلاصه

اطلاع از نحوه نوشتن یک کد VHDL به منظور به دست آوردن نتیجه سنتز بهتر بسیار مهم است. تسلط بر نحوه عمل ابزارهای سنتز نیز به همان اندازه اهمیت دارد.

تبحر و تسلط در نوشتن کد VHDL و استفاده از ابزارهای سنتز و روش طراحی به طریق ***بالا به پایین***[1] در طراحی سلسله مراتبی در مقایسه با طراحی‌های معمولی و قدیمی‌تر از امتیازات بیشتری برخوردار است. یک ابزار پیشرفته سنتز می‌تواند روند طراحی را بسیار آسان کند و سرعت آن را بالا ببرد. اگر از یک برنامه سنتز و بهینه‌سازی پیشرفته استفاده کنیم می‌توانیم تمام محدودیتهای زمانی، مکانی و طراحی را به نحو موردنیاز تنظیم کنیم تا به بهترین مدار موردنظر برسیم. در این سیستم غالباً یک سنتزگر زمانی که امکان تحلیل زمانی مدار را فراهم می‌کند نیز وجود دارد. فایل‌های گزارشی نیز درباره محدودیتهای مکانی و احتمالاً محدودیتهای طراحی که برآورده نشده‌اند قابل تحصیل خواهد بود. اگر گزارشهای ابزار سنتز بیانگر برآورده شدن کلیه محدودیتهای مدار باشند، می‌توان با اطمینان زیادی گفت که مدار به خوبی عمل خواهد کرد، زیرا این مدار باید در خلال شبیه‌سازی VHDL توانایی عملیاتی خود را به اثبات رسانده باشد.

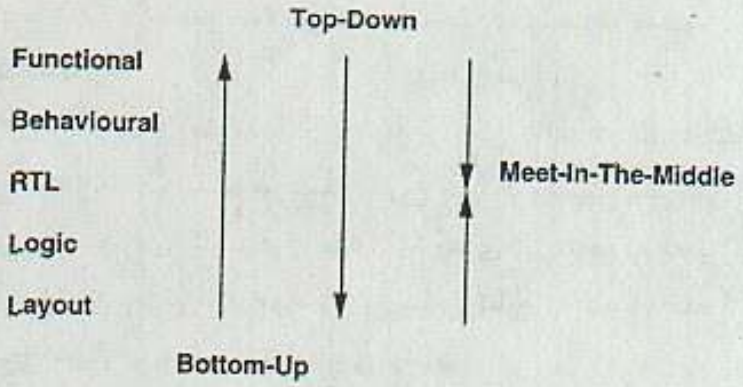
به طور خلاصه می‌توان گفت که این سنتز است که امکان طراحی «بالا به پایین» با کمک VHDL را فراهم می‌کند.

1- Top-Down Design

فصل یازدهم

آشنایی با روشهای طراحی

در طراحی‌های ASIC/FPGA روشهای گوناگونی وجود دارد (شکل ۱-۱۱). یکی از روشهایی که بیشتر مورد توجه واقع شده طراحی «بالا به پایین» است.



شکل ۱-۱۱ روشهای طراحی

روشی که قبل از به وجود آمدن VHDL زیاد از آن استفاده می‌شد، روش meet-in-the-middle بود. بیشتر پروژه‌ها با نوشتن مشخصات مورد نیاز شروع می‌شدند. سپس طراحی‌ها مستقیماً در سطح گیتی

آغاز می‌شد. گیت‌ها به طور دستی بر روی شماتیک مدار قرار می‌گرفتند و به هم متصل می‌شدند. در این روش طراحان در همان زمانی که سعی می‌کردند خصوصیات موردنیاز پروژه را پیاده‌سازی کنند مجبور بودند روی موضوعاتی فکر کنند که هیچ ربطی به وظایف و عملکرد مدار نداشت. در طراحی‌های ASIC طراح باید برای کلیه گیت‌ها میزان قدرت محرک را انتخاب کند. او همچنین مجبور بود معماری طرح را با توجه به محدودیتهای مکانی که روش اجرایی طرح را کنترل می‌کرد انتخاب نماید. زمان‌بندی نیز باید در موقعی که گیت‌ها روی شماتیک قرار می‌گرفتند به طور دستی محاسبه می‌شد. در جریان این محاسبات طراح باید دقت می‌کرد که تعداد گیت‌ها از تعدادی که باید در مدار موردنظر جاسازی می‌شد تجاوز نکند. از این رو طراح باید در حین انجام محاسبات زمان‌بندی، بر روی میزان نیروی محرکه و ظرفیت گیت‌ها و شبکه‌ها کنترل دستی اعمال می‌کرد. بنابراین اگر معلوم می‌شد که در شبیه‌سازی زمانی مدار اشکالاتی رخ داده و یا تعداد گیت‌های مورد استفاده بیش از حد زیاد بوده، طراح باید به طور دستی این مشکلات را در شماتیک اصلاح می‌کرد. گاهی اوقات طراح مجبور به تغییر کل بلوک‌های بخشهای مختلف طراحی به منظور دستیابی به اهداف پروژه می‌شد. اگر مدیر پروژه بعضی از فاکتورهای پروژه را مانند محدودیتهای زمانی و یا اندازه پالس داخلی را در اثنای پروژه تغییر می‌داد، طراح مجبور بود کل طرح را از نو طراحی کند. این عمل درست مانند آن بود که بخواهند تکنولوژی انتخاب شده را به کلی تغییر دهند. به این ترتیب بود که طراح باید به همان اندازه که وقت خود را صرف توصیف عملکرد صحیح مدار می‌کرد وقت زیادی را صرف موضوعی کند که تأثیری بر روی عملکرد مدار نداشت.

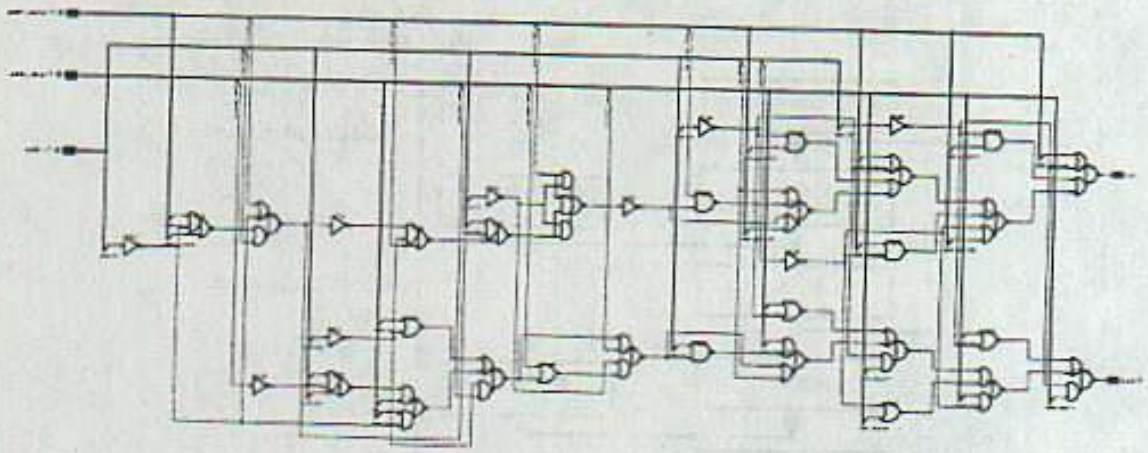
تکنولوژی «پایین به بالا» روش دیگری بود که از روش meet-in-the-middle نیز بدتر بود. در این روش طراح کار خود را با تهیه نقشه بورد مدار شروع می‌کرد و سپس سعی می‌کرد بررسی کند که چه مدارهایی را می‌توانست روی کارت قرار دهد. گام بعدی این بود که به بررسی عملکرد کارت بپردازد. کار پروژه با نوشتن ویژگی‌های موردنیاز بر اساس نتایج به دست آمده از شبیه‌سازی پایان می‌یافت. در واقع هیچ کس از این روش استفاده نمی‌کرد.

به موازات پیچیده‌تر شدن اکثر طرحها، سازندگان درصدد یافتن راههای جدیدی برای طراحی مدارهای خود برآمدند. حتی در مورد طرحهایی که از لحاظ جا نیازی به افزایش نداشتند فاز طراحی پروژه باید از لحاظ زمان‌بندی کاهش زیادی می‌یافت تا بتواند شرایط زمانی موردنظر بازار و صرفه‌جویی مالی را برآورده سازد. اگر طراح می‌توانست به جای تمرکز بر روی انتخاب گیت‌ها، ظرفیت، نیروی محرکه و از این قبیل به وظایف عملکردی مدار بپردازد، ممکن بود که هر دو اشکال فوق برطرف شود. برای اکثر سازندگان ایجاد یک زبان توصیف سخت‌افزاری، یعنی VHDL، پاسخ تمام این مشکلات بود.

طراحی با VHDL نیازی به هیچ یک از پیش فرضهایی که در شکل ۱-۱۱ ذکر شد ندارد. با VHDL می‌توان بهترین و مناسب‌ترین روش طراحی یک پروژه را انتخاب کرد و آن طراحی به روش «بالا به پایین» است که دارای امتیازات زیر است :

- طرحهای پیچیده می‌توانند به سادگی کنترل شوند.
- زمان طراحی کاهش می‌یابد.
- کیفیت افزایش می‌یابد.
- نمونه‌سازی سریع با FPGA امکان‌پذیر می‌شود.
- امکان تغییر چند باره طرح افزایش می‌یابد.

در زیر ابتدا توصیف متداول یک طرح در محیط شماتیکی شکل ۲-۱۱ و سپس توصیف همان طرح در سطح RT با کمک زبان VHDL (شکل ۳-۱۱) آورده شده است.



شکل ۲-۱۱ طرح در سطح گیتی

```
Architecture rtl of ex is
begin
    cs<='1' when addr > addr_data else '0';
    overf<='1'  when addr_dma > addr else '0';
end;
```

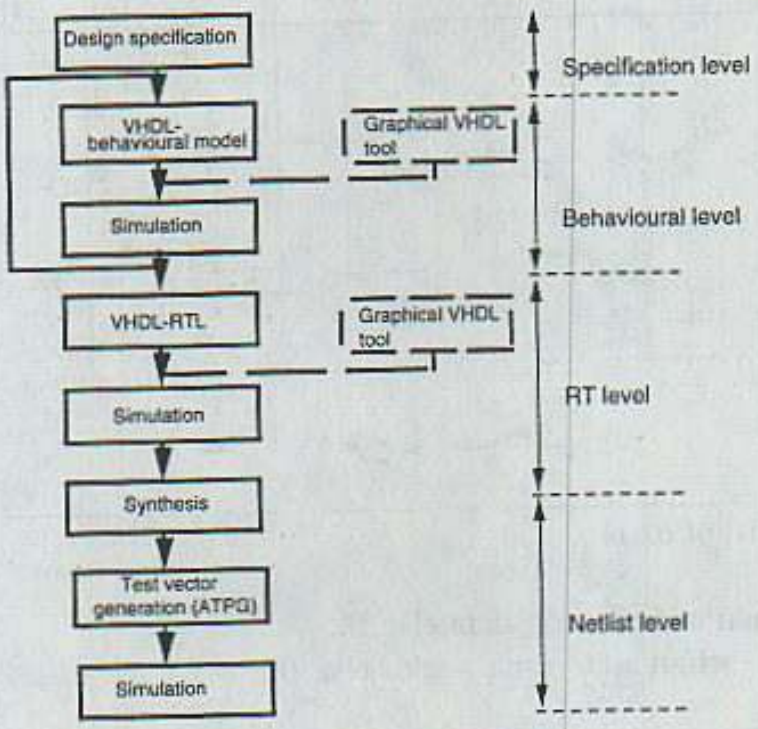
شکل ۳-۱۱ طرح با VHDL

چنان‌چه توصیف قدیمی و متداول شماتیکی را با طراحی VHDL مقایسه کنیم فوراً به سهولت

طراحی با VHDL پی می‌بریم. تقریباً هیچ کسی نمی‌تواند با نگاه کردن به شکل ۲-۱۱ عملکرد تعدادی گیت را در پنج یا شش ثانیه بیان کند. از سوی دیگر اکثر افراد می‌توانند عملکرد کد VHDL شکل ۳-۱۱ را به راحتی متوجه شوند. البته تفاوت همیشه به این وضوح نخواهد بود (که اغلب این طور است) اما این مثال به خوبی تفاوت را نشان می‌دهد. در شکل بالا، به منظور مقایسه بهتر عملکرد مدارها، بخش entity برنامه نوشته نشده است. کلیه بردارها در بالا دارای طول ۸ بیت هستند. چنان‌چه بخواهیم طول آنها را تغییر دهیم، کافی است در بخش اعلام برنامه این تغییر را اعمال کنیم، در حالی که برای تغییر پهنای باس تغییرات دستی بسیار زیادی باید در طرح شماتیکی مدار اعمال شود.

۱-۱۱ جریان بالا به پایین[1]

در طراحی بالا به پایین، در ابتدا باید روی مشخصات و سپس طراحی عملکرد مدار تمرکز کرد (شکل ۴-۱۱). هر چه پروژه به مراحل پایین‌تر حرکت کند، میزان جزئیات بیشتر خواهد شد. تا قبل از مرحله سنتز نیازی به تعیین تکنولوژی نیست، گرچه مدار باید قبل از سنتز از نظر عملکردی کاملاً بررسی و درستی عمل آن اثبات شده باشد.



شکل ۴-۱۱ جریان بالا به پایین

1- Top-Down Flow

هر آنچه تاکنون در این کتاب به زبان VHDL توصیف شده در سطح انتقال رجیستری (سطح RT) بوده است. RT سطحی است که ابزارهای سنتز آن را قبول می‌کنند و مدل سخت‌افزاری آن را می‌سازند. VHDL رفتاری از همان دستورات VHDL استفاده می‌کند بدون آنکه به سخت‌افزار بیندیشد. هدف از مدل رفتاری VHDL صرفاً بررسی صحت عملکرد مدل است. در مدلهای رفتاری معمولاً از توعهای مختلف داده استفاده می‌شود. در ضمن در اغلب موارد، مدلها کم و بیش آسنکرون هستند و بنابراین در بسیاری از آنها نیازی به کلاک وجود ندارد. به علاوه تأخیرهای زمانی در کد VHDL می‌تواند با دستور after به جای استفاده از شمارنده در سطح RT توصیف شوند.

یک مثال از کد رفتاری یک watchdog timer می‌تواند به فرم زیر باشد :

```
Architecture behv of watchdog is
begin
        time_out<= '0' when (trig= '1' or resetn= '0') else
                   '1' after 10000 ns;
end;
```

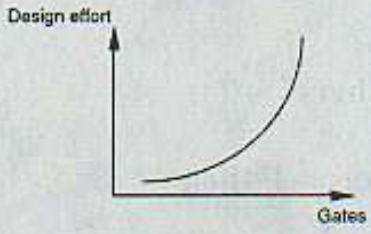
همین مثال در سطح RT به صورت زیر است :

```
Architecture rtl of watchdog is
signal count:std_logic_vector(6 downto 0);
begin
  process
  begin
       if resetn= '0' then
              count<= (others=> '0');
              time_out<= '0';
       elsif clk'event and clk= '1' then
         if trigg= '1' then
              count<= (others=> '0');
              time_out<= '0';
         elsif count/=100 then
              count<=count+1;
              time_out<='0';
         else
              time_out<='1';
         end if;
       end if;
  end process;
end;
```

مدل رفتاری را نباید با کد رفتاری که برای سنتز رفتاری به کار می‌رود اشتباه کرد. چنان‌چه هدف سنتز تحلیل مدل رفتاری باشد، سخت‌افزار مورد استفاده در کد VHDL باید مورد توجه قرار گیرد، به این معنا که پالس ساعت و رجیستر باید در مدل پیش‌بینی شده باشد. مطالب بیشتر در این خصوص در فصل ۱۷، «سنتز رفتاری» آورده شده است.

در طراحی بالا به پایین، مدار باید در ابتدا توسط مدل رفتاری طرح‌ریزی گردد. از امتیازات مدل رفتاری (سنتزناپذیر) این است که طراح می‌تواند در هر مرحله‌ای از طراحی آن را شبیه‌سازی کرده و خطاهای موجود در سیستم را کشف کند. لیکن در خیلی از موارد این مرحله را می‌توان در جریان بالا به پایین حذف نمود. تنها در مدارهای با پیچیدگی بالا (سیستمهای پیچیده) و یا مدارهایی که باید به طور هم‌زمان طراحی شوند، این سطح توصیفی می‌تواند مورد قبول باشد. اگر پروژه در سطحی باشد که بتوان آن را سنتز رفتاری کرد، توصیه می‌شود که مدل رفتاری مورد استفاده قرار گیرد. بنابراین مدل باید به گونه‌ای نوشته شود که بتوان آن را با سنتز رفتاری سنتز کرد. از آنجایی که در این موارد سنتز در سطح رفتاری انجام خواهد شد، دیگر نیازی به بازنویسی کد VHDL در سطح RT نخواهد بود.

یکی از دلایل مهمی که تسلط در طراحی به شیوه بالا به پایین را پراهمیت می‌سازد آن است که کار طراحی همراه با ازدیاد تعداد گیت‌ها چندین برابر افزایش پیدا می‌کند (شکل ۵-۱۱). این بدان معناست که فرضاً وقتی اندازه طرح دو برابر شود، مساعی و تلاشهای طراحی مدار بیش از دو برابر افزایش می‌یابد.



شکل ۵-۱۱ مشکل شدن طراحی به عنوان تابعی از تعداد گیت‌ها

۲-۱۱ آزمایش و اثبات درستی عمل

در جریان بالا به پایین سه مرحله امتحان درستی عمل وجود دارد :

- مرحله مدل رفتاری
- مرحله مدل VHDL در سطح RT
- مرحله سطح گیتی قبل و بعد از طرح‌ریزی

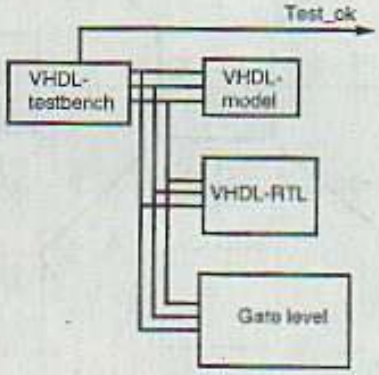
یک المان و یا جزء ترکیبی را می‌توان به گونه زیر در هر سه مرحله فوق مورد آزمایش قرار داد:

- ایجاد بردارهای آزمایش در مرحله شبیه‌سازی
- استفاده از محیط آزمایش VHDL
- انجام شبیه‌سازی سیستم

قبل از ایجاد VHDL روش قدیمی آزمایش و تأیید یک المان طراحی شده از طریق ایجاد محرکهای ورودی برای هر گونه شبیه‌سازی که مورد استفاده بوده انجام می‌گرفت. عیب این روش آن است که عمل شبیه‌سازی وابسته به محیط آزمایش مورد استفاده می‌شد. همچنین اگر همان شخصی که طراحی را انجام داده به تهیه بردارهای آزمایش بپردازد این امکان وجود دارد که چنان‌چه یک خطای منطقی در جزء طراحی شده وجود داشته باشد همان خطا در بردارهای آزمایش بروز نماید و در نتیجه خطای موجود در مدار پیدا نشود. بنابراین آنچه این روش مورد آزمایش قرار می‌دهد در اصل چیزی جز بررسی طرح بر اساس آنچه در ذهن طراح بوده است نبوده و نمی‌تواند درستی عمل طرح را وقتی بر روی بورد سوار می‌شود تأیید نماید.

روش دیگر ایجاد محیط آزمایش است. محیط آزمایش در حقیقت یک برنامه VHDL است که کار آن تولید محرکهایی برای آزمایش مدل طراحی شده است و می‌تواند نشان دهد که آیا برنامه به درستی عمل می‌کند یا خیر. میزان جزئیات محیط آزمایش از یک مورد به مورد دیگر فرق می‌کند و از تست کلیه توابع مدار که توصیف‌کننده مشخصات آن هستند مثل زمان‌بندی سیگنال‌های مدار تا آزمایش فقط چند تابع خاص را در بر می‌گیرد.

محیط آزمایش همچنین می‌تواند برای تست کردن کد VHDL در سطح RT و شبیه‌سازی نهایی در سطح گیتی به کار رود (شکل ۶-۱۱). همچنین می‌توانید به فصل ۸، «محیط آزمایش» مراجعه کنید.

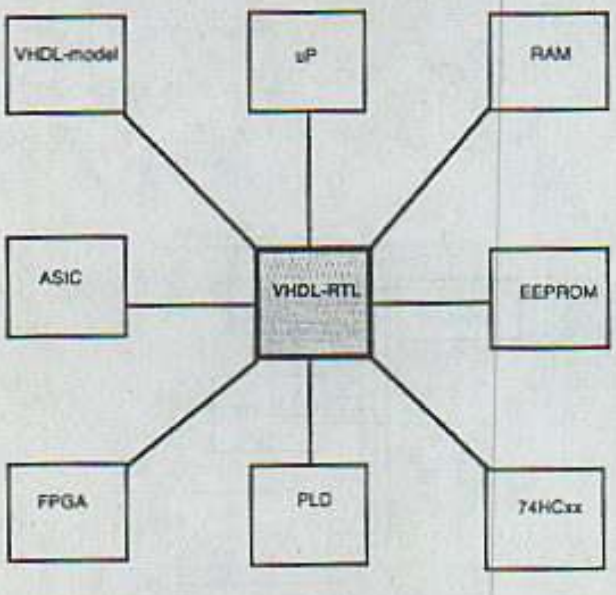


شکل ۶-۱۱ پیکره‌بندی‌های گوناگون محیط آزمایش

یکی از امتیازات محیطهای آزمایش در VHDL وابسته نبودن آنها به ابزار سنتز و محیط طراحی است. در اینجا نیز احتمال به وجود آمدن همان خطاهای منطقی برنامه اصلی در برنامه آزمایش و تست وجود دارد. یک روش برای کاهش چنین ریسکی آن است که شخص دیگری غیر از طراح پروژه به نوشتن برنامه تست بپردازد، این شخص می‌تواند فرضاً کسی باشد که مشخصه‌های موردنیاز را نوشته است.

روش سوم اتصال مدلهای تشکیل‌دهنده طرح و اجرای آن چیزی است که شبیه‌سازی سیستم می‌نامند. چنان‌چه مدلهای VHDLبرای بررسی همه المان‌های سیستم وجود نداشته باشد طراحی باید به شبیه‌سازهایی که بتوانند از عهده دو سطح در همه آمیخته (مثل VHDL در ترکیب با شماتیک و سطح گیتی) برآیند دسترسی داشته باشد. اما اگر برای تمام المان‌های مدار، مدلهای VHDL وجود داشته باشد می‌توان از شبیه‌ساز متداول VHDL برای شبیه‌سازی سیستم استفاده کرد.

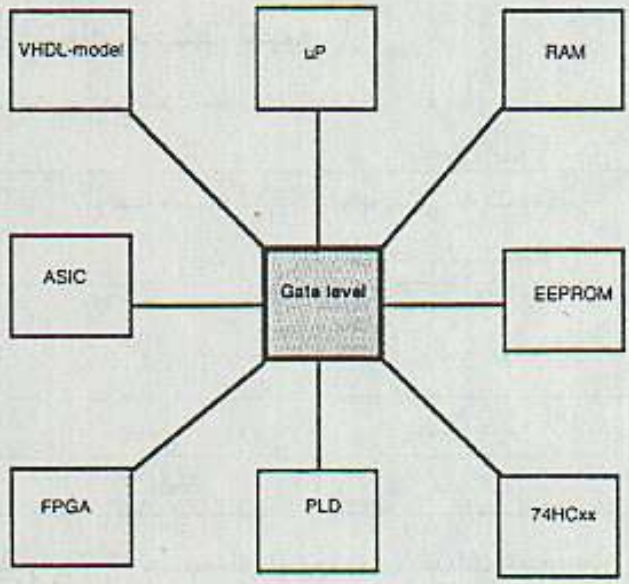
این روش اثبات صحت عمل (شبیه‌سازی سیستم) هنگامی که برای کشف خطاهای احتمالی طراحی در مواجهه با ابزارها و المان‌های جانبی مورد استفاده قرار گیرد می‌تواند یکی از بهترین روشها باشد. مدار طراحی شده غالباً در مواجهه با قطعاتی مثل میکروپروسسور، EEPROM، RAM، منطقهای کنترلی مثلاً از نوع 74HCXX، مدلهای‌دیگر VHDL مثل PLD ها، FPGA ها و ASIC ها کار می‌کند. در حال حاضر امکان خرید انواع مدلهای میکروپروسسورها وجود دارد. مدارهای FPGA، PLD و ASIC می‌توانند یا از مدلهای VHDL یا از شماتیکهای قابل شبیه‌سازی باشند. این مدارها می‌توانند در یک شبیه‌ساز سخت‌افزاری برای چک کردن و شبیه‌سازی مدار قرار گیرند.



شکل ۷-۱۱ شبیه‌سازی سیستم در سطح RT

این روش شبیه‌سازی همواره باید بر روی یک مدل VHDL توصیف شده در سطح RT انجام شود (شکل ۷-۱۱).

همچنین می‌توان شبیه‌سازی سیستم را در سطح رفتاری و سطح گیتی انجام داد. به طور نرمال مدار در سطح RT تنها از لحاظ عملکردی بازبینی می‌شود. آزمایش زمان‌بندی نیز هم در ابزارهای سنتز و هم در شبیه‌سازی در سطح گیتی انجام می‌گیرد. محدودیتهای زمانی در این روش در ابزار سنتز تعیین می‌شوند. پس از بهینه‌سازی، گزارشهایی در مورد برآورده شدن یا نشدن این محدودیتها آماده خواهد شد. برای اطمینان از اینکه در موقع تعیین محدودیتهای زمانی هیچ خطایی صورت نپذیرفته است باید در عین حال، مدل سطح گیتی را نیز در یک شبیه‌سازی سیستمی مورد آزمایش قرار داد (شکل ۸-۱۱).



شکل ۸-۱۱ شبیه‌سازی سیستم در سطح گیتی

از معایب شبیه‌سازی سیستم در مقایسه با دیگر روشهای شبیه‌سازی، طولانی‌تر بودن زمان عملکرد آن است.

پس از آماده شدن نقشه طرح، مجدداً باید مدار را تحت بررسی قرار داده و اطمینان یافت که محدودیتهای زمانی مراعات شده است. زمان‌بندی بعد از آماده شدن طرح همیشه با زمان‌بندی قبل از آماده شدن طرح به خاطر اینکه به طور تخمینی انجام شده بود تفاوت دارد.

با به کارگیری بردارهای آزمایش (به فصل ۱۲ مراجعه کنید، «آشنایی با روشهای آزمایش») می‌توان به راحتی پی برد که آیا در طراحی، خطاهای عملکردی رخ داده است یا خیر. زمان‌بندی را می‌توان به دو طریق زیر مورد بررسی قرار داد :

- انجام شبیه‌سازی
- استفاده از تحلیل‌گر زمانی

معمولاً آزمایش زمانی به وسیله شبیه‌سازی به وقت زیادی نیاز دارد زیرا چک کردن اینکه تمام مسیرهای داده در مدار محدودیتهای زمانی را رعایت کرده‌اند یا خیر کار بسیار دشواری است. اما یک روش بسیار سریع‌تر و کارآمدتر استفاده از یک تحلیل‌گر زمانی است. با به کارگیری این نوع تحلیل‌گر کار آزمایش با درجه اطمینان بالا ظرف چند دقیقه انجام‌پذیر می‌گردد. معمولاً در ابزارهای پیشرفته سنتز یک تحلیل‌گر زمانی برای همین منظور قرار داده شده است.

۱-۲-۱۱ جمع‌بندی روشهای مختلف شبیه‌سازی

مزایا و معایب انواع گوناگون شبیه‌سازی در جدول ۱-۱۱ آورده شده است.

	سرعت شبیه‌سازی	وابستگی به شبیه‌ساز و محیط شبیه‌سازی	نزدیکی به حقیقت
بردارهای آزمون با زبان شبیه‌ساز	بسیار خوب	بد	ضعیف
محیط آزمایش	بسیار خوب	بسیار خوب	خوب
شبیه‌سازی سیستم	ضعیف	ضعیف	بسیار خوب

جدول ۱-۱۱ خلاصه‌ای از روشهای مختلف شبیه‌سازی

توصیه ما این است که برنامه آزمایش VHDL با شبیه‌سازی سیستم توأم شود. در بلوک‌های VHDL (در طراحی سلسله مراتبی) می‌توان از یک برنامه آزمایش ساده برای تشخیص و رفع خطاهای طراحی استفاده کرد. برای بالا بردن سرعت شبیه‌سازی می‌توان شبیه‌سازی سیستم را در سطح بلوک‌های VHDL به کار برد. به طور مثال فرض کنید که یک باس داخلی و یا رابط CPU طراحی کرده‌ایم. می‌توانیم بلوک‌های مختلف را در ارتباط با اجزای جانبی با مدارهای واسط بین باس‌ها و یا یک مدل CPU در این مثال شبیه‌سازی کنیم.

در آینده طراح می‌تواند شبیه‌سازی سیستم را با یک شبیه‌ساز معمولی VHDL انجام دهد. در حال حاضر بعضی از شبیه‌سازهای VHDL می‌توانند به شبیه‌سازهای سخت‌افزاری متصل شوند.

فعالیتهایی نیز برای ساختن یک سیستم استاندارد به نام *VITAL*[1] به منظور آنکه بتوان در شبیه‌سازی، VHDL در سطح گیتی را نیز شبیه‌سازی نمود، در حال انجام است. هم اکنون تعدای از شبیه‌سازهای VHDL و تولیدکنندگان ASIC هستند که از این استاندارد حمایت می‌کنند.

بیشترین زمان و تلاش باید برای شبیه‌سازی VHDL در سطح RT صرف شود. زمانی که کد VHDL شبیه‌سازی شود، دیگر هیچ گونه خطای عملکردی نباید در مدار مشاهده گردد. این موضوع اهمیت زیادی دارد، زیرا روش طراحی بالا به پایین بر اساس تشخیص و رفع سریع خطاهای عملکردی در مراحل اولیه طراحی استوار است. تنها خطاهایی که ممکن است در این مقطع از طراحی به وجود آیند مشکلات زمانی و مکانی خواهد بود. این مشکلات ذاتاً به نوع تکنولوژی مورد استفاده بستگی دارد. اما این روش طراحی (بالا به پایین) تاکنون کاملاً غیروابسته به تکنولوژی بوده و از این جهت جایی برای نگرانی در خصوص مشکلاتی از این نوع وجود ندارد. در ضمن تکنولوژی کار باید در ابزار سنتز انتخاب شود.

۲-۲-۱۱ سرعت شبیه‌سازی

در طول دهه ۱۹۹۰ قدرت عمل شبیه‌سازهای VHDL رشد قابل توجهی داشتند. اندازه مدارها و سیستمهایی که توسط آنها شبیه‌سازی می‌شوند افزایش یافت. نکته مهم این بود که کد VHDL چگونه باید نوشته شود تا سرعت شبیه‌سازی افزایش یابد. اما متأسفانه نوشتن کد VHDL به نحوی که خواندن آن آسان‌تر و سرعت شبیه‌سازی آن بیشتر شود بعضی اوقات با یکدیگر تضاد پیدا می‌کند. در این موارد طراح باید تصمیم بگیرد که کدام یک مهم‌تر است. در زیر به نکاتی که طراح باید برای کاهش زمان شبیه‌سازی به آنها توجه کند اشاره شده است :

۱- کم کردن تعداد پروسس‌ها
۲- اجتناب از تبدیل چندباره مقادیر
۳- اضافه نکردن سیگنال‌های غیرضروری به لیست حساسیت پروسس‌ها
۴- چک کردن اینکه شبیه‌ساز انتخاب شده چه نوع داده‌ای را ترجیح می‌دهد
۵- زیرک کردن کدها
۶- کاهش تعداد مراحل در طراحی سلسله مراتبی
۷- اجتناب از خواندن و نوشتن VHDL
۸- به کارگیری متغیر به جای سیگنال

1- VHDL Initiate To ASIC Library

۱- کم کردن تعداد پروسس‌ها

در VHDL دو نوع پروسس وجود دارد : ترکیبی و پالسی. پروسس‌های پالسی به طور نرمال با کلاک سیستم و سیگنال reset فعال می‌شوند. معمول این است که از چندین پروسس پالسی با لیستهای حساسیت یکسان در بخشهای مختلف معماری برنامه استفاده شود. این عمل خوانایی و اصلاح برنامه را آسان‌تر می‌کند. اما از سوی دیگر از نقطه نظر افزایش سرعت شبیه‌سازی کار مؤثری انجام نمی‌دهد. چنان‌چه کلیه پروسس‌های پالسی را بتوان در قالب یک پروسس واحد خلاصه‌کرد، طول مدت شبیه‌سازی بسیار کاهش خواهد یافت. همین قانون در مورد پروسس‌های ترکیبی نیز صادق است.

مثالی از یک کد VHDL با خوانایی سهل و آسان که از لحاظ عملیات شبیه‌سازی خیلی کامل نیست می‌تواند چنین باشد :

```
Architecture rtl of ex is
begin
   p0:process(clk,resetn)
   begin
        . . .
        ql<=. . .
   end process;
   . . .
   p4:process(clk,resetn)
   begin
        . . .
        q4<=. . .
   end process;
end;
```

استفاده از همان مثال قبلی ولی با به کارگیری فقط یک پروسس باعث می‌شود که کد VHDL بسیار سریع‌تر عمل کند، لیکن روش توصیف آن از لحاظ خوانایی برنامه چیز مناسب و خوبی نیست.

```
Architecture rtl of ex is
begin
   p0:process(clk,resetn)
   begin
        . . .
        q1<=. . .
        . . .
        q4<=. . .
   end process;
end;
```

۲- اجتناب از تبدیل چندباره مقادیر

ممکن است تبدیل نکردن چندباره یک مقدار در کد VHDL به نظر امری بدیهی بیاید، اما این اشتباهی است که به سادگی رخ می‌دهد. اگر به طور مثال دستور زیر در یک محیط آزمایش VHDL به کار گرفته شود، هر مرتبه که این خط اجرا شود عمل تبدیل نیز اجرا خواهد شد :

```
wait for 2*period;
```

چنان‌چه از یک کمپایلر قوی استفاده شود، می‌توان از اشتباه تبدیل به دفعات، در این مثال جلوگیری کرد. مطمئن‌ترین راه حل این مشکل معرفی یک مقدار ثابت به صورت زیر است :

```
constant period2 : time := 2*period;
```

بنابراین کد VHDL می‌تواند این گونه نوشته شود :

```
wait for period2;
```

با این روش مقدار period2 در طی شبیه‌سازی فقط یک بار محاسبه می‌شود.

۳- اضافه نکردن سیگنال‌های غیرضروری در لیست حساسیت پروسس‌ها

لیست حساسیت است که تصمیم می‌گیرد یک پروسس چه موقعی باید شروع به کار کند. چنان‌چه سیگنال‌های غیرضروری در لیست قرار گیرند، شبیه‌سازی کندتر خواهد شد. یکی از اشتباهات متداول گنجاندن سیگنال‌هایی بجز کلاک و reset در پروسس پالسی است.

مثال :

```
process(clk,resetn,d)
begin
  if resetn= '0' then
      q<= '0';
  elsif clk'event and clk= '1' then
      q<=d;
  end if;
end process;
```

در کد VHDL بالا، سیگنال d در داخل لیست حساسیت قرار دارد. از آنجایی که پروسس پالسی است، ارزش سیگنال خروجی q تنها می‌تواند در لبه پالس ساعت و یا در صورت فعال شدن

سیگنال reset تغییر کند. این طرح از لحاظ شبیه‌سازی و سنتز به خوبی عمل می‌کند، لیکن عمل شبیه‌سازی آن بیشتر از حد لازم زمان می‌گیرد و علت آن این است که این پروسس علاوه بر سیگنال کلاک و سیگنال reset با هر رویدادی بر سیگنال d نیز فعال می‌شود. پروسس فوق باید به شکل زیر نوشته شود :

```
process(clk,resetn)
begin
   if resetn= '0' then
        q<= '0';
   elsif clk'event and clk= '1' then
        q<=d;
   end if;
end process;
```

۴- چک کردن اینکه شبیه‌ساز انتخاب شده چه نوع داده‌ای را ترجیح می‌دهد

بعضی از شبیه‌سازهای VHDL به گونه‌ای طراحی شده‌اند که به طور مثال داده‌هایی از نوع std_logic و std_logic_vector را نسبت به std_ulogic و std_ulogic_vector سریع‌تر شبیه‌سازی می‌کنند. بنابراین طراح باید در مورد وجود این گونه ترجیحات پیش‌بینی شده در سیستم از سازنده شبیه‌ساز VHDL سؤال کند.

۵- زیرک کردن کدها

طراح می‌داند که به طور مثال کدام یک از شرطهای داخل دستور if-then-else اغلب اوقات رخ خواهد داد و از آنجایی که معمولاً تنها یکی از شرطهای داخل دستور if-then-else اجرا خواهد شد، بنابراین با قرار دادن این شرط در ابتدای دستورات سرعت شبیه‌سازی بیشتر خواهد شد.

مثال :

```
process(a,b)
begin
   if a>12 and b<11 then      -- The most common condition
        . . .
   elsif a=13 and b=2 then    -- The second commonest condition
        . . .
        . . .
   end if;
end process;
```

۶- کاهش تعداد مراحل در طراحی سلسله مراتبی

هر چه تعداد مراحل بیشتر باشد، زمان شبیه‌سازی طولانی خواهد شد. یکی از دلایل این موضوع افزایش تعداد کدهای VHDL است.

۷- اجتناب از خواندن و نوشتن TextIO

به کارگیری TextIO در کد VHDL به طور نرمال باعث افزایش زمان شبیه‌سازی می‌شود. بنابراین چنان‌چه زمان شبیه‌سازی مهم باشد باید از نوشتن در یک فایل خارجی یا خواندن از یک فایل خارجی اجتناب شود.

۸- به کارگیری متغیر به جای سیگنال

متغیرها در شبیه‌سازی بسیار سریع‌تر از سیگنال‌ها هستند. اگر لازم است زمان شبیه‌سازی کاهش یابد تا آنجایی که ممکن است باید از متغیر استفاده شود. متغیرها سریع‌تر هستند زیرا متغیرها مثل سیگنال‌ها به هنگام تغییر مقدار با یک سری عملیات مواجه نیستند. یک متغیر فقط می‌تواند یک مقدار در لحظه کنونی داشته باشد، در حالی که یک سیگنال می‌تواند علاوه بر مقدار کنونی، چندین مقدر در ردیفی از وقایع به خود بگیرد (به فصل ۴ رجوع کنید، «VHDL متوالی»).

۳-۲-۱۱ تست و آزمایش

آزمایش و اثبات یک حوزه جدیدی است که انتظار می‌رود در آینده برای تأیید صحت عمل نقش مهمی داشته باشد. در حال حاضر دو اصل متفاوت برای آزمایش و تأیید وجود دارد :

- استفاده از «مدل طلایی»
- نوشتن مشخصات موردنیاز در یک محیط نوشتاری

این روش به این منظور به کار می‌رود که معلوم کند دو طرح از لحاظ عملکرد عین هم هستند. این یک متد کارآمد و سریع برای آزمایش طرحی است که در آن باید کد VHDL فرضاً به دلیل مسائل زمانی و مکانی دوباره نوشته شود. از طریق آزمایش، طرح صحیح و بی‌عیب را می‌توان به عنوان «مدل طلایی» برای مقایسه با هر طرح جدیدی مورد استفاده قرار داد. این بدان معناست که چنان‌چه کد VHDL تغییر کند نیازی به شبیه‌سازی دوباره طرح نیست و به این ترتیب زمان زیادی صرفه‌جویی خواهد شد (به فصل ۱۰ مراجعه کنید، «سنتز در سطح RT»).

روش دیگر به‌کارگیری آزمایش، توصیف مشخصات یک جزء ترکیبی و یا المان موردنظر با زبانی است که برای آزمایش کننده شناخته شده و قابل فهم باشد. بر پایه این مشخصات و نیازمندی‌های مدار است که ابزار تحلیل‌گر می‌تواند به آزمایش طرح پرداخته و معلوم کند که آیا طرح تهیه شده تمام نیازمندی‌های مدار را تحت پوشش قرار داده است یا خیر. این نوع آزمایش تا حدودی می‌تواند به جای شبیه‌سازی و یا مکمل آن عمل کند.

۴-۲-۱۱ توصیه‌هایی در مورد آزمایش و بررسی صحت عمل

سطح رفتاری

توصیه می‌شود که همواره برای شبیه‌سازی در سطح رفتاری و در سطح RT از شبیه‌سازی سیستم استفاده کنیم. قبل از آنکه شبیه‌سازی سیستم شروع شود، طرح باید به وسیله محیط آزمایش VHDL مورد بررسی و تأیید قرار گیرد. همان طور که قبلاً گفته شد، علت این امر سریع‌تر بودن محیط آزمایش VHDL نسبت به شبیه‌سازی سیستم است.

سطح گیتی

عملاً نباید یک طرح در سطح گیتی شبیه‌سازی شود. زمان‌بندی باید با تحلیل‌گر زمانی داخل ابزار سنتز بررسی شود، لیکن توصیه می‌گردد که هم محیط آزمایش VHDL و هم شبیه‌ساز سیستم که در سطح گیتی اجرا می‌شوند مورد استفاده قرار گیرند، زیرا ممکن است ابزار سنتز دارای اشکال بوده و منجر به ارائه یک netlist غلط گردد. همچنین توصیه می‌شود که با انجام دادن چند مورد شبیه‌سازی در سطح گیتی اطمینان حاصل نمود که محدودیتهای زمانی تعیین شده صحیح هستند.

پس از طرح‌ریزی

در طراحی‌های ASIC بسیار توصیه می‌شود پس از طرح‌ریزی، با استفاده از بردارهای آزمایش اطمینان حاصل کرد که طرح پس از طرح‌ریزی کماکان صحیح و بی‌عیب است. زمان‌بندی نیز باید با کمک یک تحلیل‌گر زمانی بررسی و تأیید گردد.

۳-۱۱ چگونگی نوشتن کد VHDL در سطح RT برای سنتز

توصیف کد VHDL به طرز صحیح برای دستیابی به یک نتیجه سنتز مطلوب بسیار مهم است. یکی از متداول‌ترین اشتباهات نوآموزان آن است که طرحهای خود را به‌صورت تعداد زیادی entity های

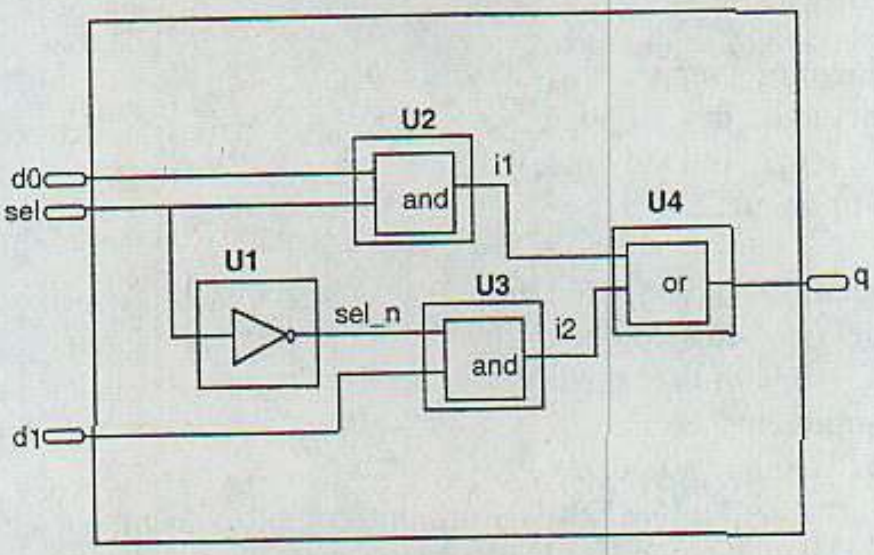
بسیار کوچک (کمتر از ۵۰۰ گیت) توصیف می‌کنند. استفاده از این بلوک‌های سلسله مراتبی کوچک دو عیب عمده دارد. اول آنکه به تعداد بالایی از کدهای ساختاری VHDL برای متصل کردن و ربط دادن کلیه اجزای ترکیبی به هم موردنیاز خواهد بود و باید در نظر داشت که کدهای ساختاری VHDL خوانا نیستند. در ضمن بیشتر شدن تعداد کدهای VHDL احتمال خطا را نیز افزایش می‌دهد. دوم آنکه نتیجه سنتز در بلوک‌های کوچک و متعدد در سطوح مختلف طراحی سلسله مراتبی نامطلوب‌تر خواهد بود زیرا ابزار سنتز نمی‌تواند منطق را بین سطوح مختلف تقسیم کند و این امر معمولاً منجر به تعداد بیشتری گیت و موجب زمان‌بندی ضعیف‌تر خواهد شد. به منظور توضیح این اصل در زیر مثال کوچکی آورده شده است. در ابتدا کد VHDL با سطوح سلسله مراتبی کوچک :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity mux is
port (d0,d1,sel:  in  std_logic;
      q:          out std_logic);
end;

Architecture struct_mux of mux is
  component and_comp
       port (a,b: in  std_logic;
             c:   out std_logic);
  end component;

  component or_comp
       port (a,b: in  std_logic;
             c:   out std_logic);
  end component;

  component inv_comp
       port (a:   in  std_logic;
             b:   out std_logic);
  end component;
  signal i1,i2,sel_n:std_logic;
  For U1: inv_comp Use Entity work.inv_comp(inv_beh);
  For U2,U3: and_comp Use Entity work.and_comp(and_beh);
  For U4: or_comp Use Entity work.or_comp(or_beh);
```

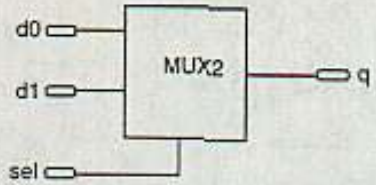
```
begin
   U1: inv_comp port map (sel,sel_n);
   U2: and_comp port map (d0,sel,il);
   U3: and_comp port map (sel_n,dl,i2);
   U4: or_comp port map (i1,i2,q);
end;
```

نتیجه سنتز در شکل ۹-۱۱ نشان داده شده است. اگر کل طرح را در قالب یک جزء ترکیبی توصیف کنیم، کد VHDL زیر حاصل خواهد شد. نتیجه سنتز این حالت در شکل ۱۰-۱۱ آورده شده است :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity good is
port(d0,d1,sel: in  std_logic;
     q:         out std_logic);
end;

Architecture rtl of good is
begin
     q<=(d0 and sel) or (not sel and d1);
end;
```



شکل ۹-۱۱ نتیجه سنتز، چندین سطح در طراحی سلسله مراتبی



شکل ۱۰-۱۱ نتیجه سنتز، یک سطح در طراحی سلسله مراتبی

اگر نتیجه دو سنتز را از لحاظ فضایی که اشغال می‌کند مورد مقایسه قرار دهیم، به نتایج زیر می‌رسیم :

۷ گیت : تعداد زیاد سطوح در طراحی سلسله مراتبی

۳ گیت : یک سطح در طراحی سلسله مراتبی

بنابراین در این مثال نتیجه سنتز برای تعداد زیادی سطوح کوچک طراحی بیش از دو برابر حجیم‌تر است. البته این یک مثال افراطی و مبالغه‌آمیز است. اما اگر تعداد واقعی‌تری اجزای ترکیبی یعنی ۱۵۰-۱۰۰ گیتی و اجزای ۶۰۰۰-۵۰۰ گیتی را با هم مقایسه کنیم، تفاوت آنها با هم چه از لحاظ زمانی و چه از لحاظ مکانی ۵ تا ۲۰ درصد خواهد بود. این تفاوت به این دلیل است که یک سطح سلسله مراتبی منطق برنامه را قفل می‌کند و کار ابزار سنتز را برای بهینه‌سازی منطق بسیار مشکل می‌سازد. لیکن استدلال فوق ایجاب می‌کند که ۶۰۰۰-۵۰۰ گیت از لحاظ عملکرد با هم پیوند داشته باشند. چنان‌چه جزء ترکیبی دارای بلوک‌های ۱۵۰ گیتی باشد و این گیت‌ها از لحاظ عملکرد هیچ رابطه‌ای با هم نداشته باشند، در این صورت تفاوتی بین تعداد زیاد سطوح سلسله مراتبی کوچک و یک سطح سلسله مراتبی بزرگ مشاهده نخواهد شد. سطوح سلسله مراتبی با بیش از ۱۰۰۰۰ گیت نیز توصیه نمی‌شوند زیرا سنتز آنها به زمانی بسیار طولانی نیاز خواهد داشت و این امر از لحاظ زمان‌بندی بلوک‌های بزرگ منطقی به منظور جداسازی گیت‌ها عمل مطلوبی نخواهد بود زیرا باعث می‌شود که گیت‌ها با فاصله بسیار دوری از یکدیگر بر روی سیلیکون قرار گیرند.

پیشنهادات

۱- گروه‌بندی بلوک‌های منطقی با یکدیگر در معماری

۲- جداسازی بلوک‌های منطقی که باید با استراتژی‌های گوناگون مورد بهینه‌سازی قرار می‌گیرند (به فصل ۱۰ رجوع کنید، «سنتز در سطح RT»).

۳- جداسازی بلوک‌های منطقی که اهداف بهینه‌سازی گوناگون دارند (زمان/مکان)

۴- پایین‌ترین سطح طراحی سلسله مراتبی باید ۵۰۰ تا ۶۰۰۰ گیت داشته باشد.

از آنجایی که ۵۰۰ تا ۶۰۰۰ گیت مستلزم کدهای متعدد و زیادی است، ممکن است فهم

عملکرد هر بلوک خیلی سخت باشد. یک راه حل مفید می‌تواند به کارگیری دستور block باشد که یک دستور موازی VHDL بوده و می‌تواند در کد VHDL در کنار سایر دستورات موازی قرار گیرد.

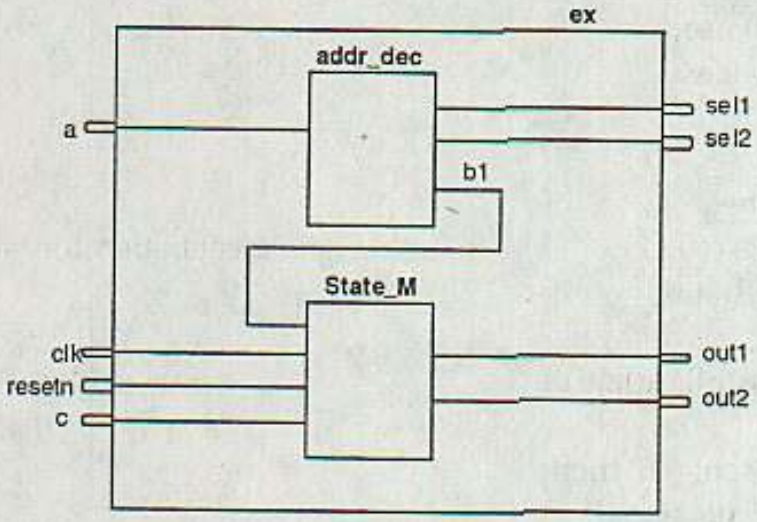
چنان‌چه دستور block به منظور تشخیص بخشهای مختلف یک برنامه به‌کار برده شود، خوانایی برنامه به طور عمومی بالا می‌رود. به علاوه، سیگنال‌های داخلی که تنها در داخل block مورد نیاز هستند را می‌توان در بخش اعلام دستور block معرفی کرد (شکل ۱۱-۱۱). این بدان معناست که سیگنال‌ها می‌توانند در بخشهایی از کد VHDL با جزئیات بیشتری معرفی شوند. چنان‌چه نامگذاری "بلوک"ها به گونه‌ای توضیحی انجام گیرد می‌توان به همان درجه از خوانایی که بلوک‌های کوچک در سطوح سلسله مراتبی از آن برخوردارند، رسید. همچنین دیگر نیازی به کدهای ناخوانای ساختاری VHDL به منظور ترکیب و ربط دادن بلوک‌های کوچک در سطوح سلسله مراتبی نخواهد بود. در ضمن نتیجه سنتز مناسب‌تری نیز به دست می‌آید.

در حین سنتز، بسته به آنکه از چه ابزار سنتزی استفاده می‌شود، می‌توان یک سطح سلسله مراتبی برای هر دستور block در کد VHDL به دست آورد. اما چون این کار منجر به ایجاد سطوح کوچک سلسله مراتبی خواهد شد، توصیه نمی‌شود.

از امتیازات دیگر دستور block این است که در حین شبیه‌سازی، هر دستور block می‌تواند به عنوان یک سطح سلسله مراتبی دیده شود (شکل ۱۲-۱۱). این بدان معناست که چنان‌چه بخواهیم کلیه سیگنال‌های مرتبط با یک بلوک منطقی مثل بلوک دیکدر آدرس را بررسی کنیم، کافی است نام بلوک را در فهرست سطوح سلسله مراتبی کلیک کرده و بعد تمام سیگنال‌های آن را انتخاب کنیم. گرچه تمام شبیه‌سازها این امکان را ندارند اما بعضی از آنها مثل Mentor Graphic Quick VHDL و V-system از این قابلیت برخوردارند.

```
Architecture rtl of ex is
<Declarative part>  -- Visible in whole architecture
begin
...
<Block_name>:Block
<Declarative part>  -- Visible only within the block
begin
 Parallel VHDL commands
end block <Block_name>;

<Block_name>:Block
<Declarative part>  -- Visible only within the block
begin
 Parallel VHDL commands
end block <Block_name>;
...
end;
```

شکل ۱۱-۱۱ دستور block



شکل ۱۲-۱۱ دیدگاه سلسله مراتبی VHDL

مثالی از چگونگی به‌کارگیری دستور block در زیر آورده شده است :

```
Entity ex is
port (a:                    in    integer;
      clk,resetn,c:         in    std_logic;
      out1,out2,sel1,sel2:  out   std_logic);
end;

Architecture small_ex of ex is
signal b1:std_logic;
begin
  addr_dec: Block
  begin
      process(a)
        case a is
            when 123=>      sel1<= '1';
                            sel2<= '0';
                            b1<= '0';
            when 130=>      sel1<= '0';
                            sel2<= '1';
                            b1<= '0';
            when others=>   sel1<= '0';
                            sel2<= '0';
                            b1<= '1';
```

```
        end case;
    end process;
end Block;

State_M:Block
type states is (s0,s1);        -- Internal signal declaration for block State_M
signal state:states;
begin
    process(clk,resetn)
    begin
      if resetn= '0' then
          state<=s0;
      elsif clk'event and clk= '1' then
          case state is
            when s0 => if  b1= '1' then
                          state<=s1;
                       end if;
            when s1 => if  c= '1' then
                          state<=s0;
                       end if;
          end case;
      end if;
    end process;
    process(state)
    begin
          case state is
            when s0 =>      out1<= '0';
                            out2<= '0';
            when s1 =>      out1<= '0';
                            out2<= '1';
          end case;
    end process;
 end block;
end;
```

پیشنهادات

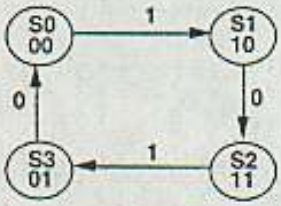
۱- همواره از دستور block در برنامه‌های بزرگ استفاده کنید زیرا خوانایی و رفع اشکال در طی شبیه‌سازی آسان‌تر می‌شود.

۲- از VHDL ساختاری به منظور ایجاد طراحی سلسله مراتبی در یک کد VHDL به جای دستور block استفاده نمایید.

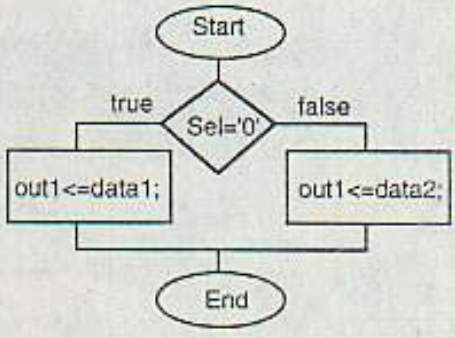
۳- سطوح در طراحی سلسله مراتبی نباید بیشتر از چهار سطح باشند.

تعداد سطوح سلسله مراتبی باید در حد کمی نگه داشته شود، زیرا اگر از چهار سطح بیشتر باشند علاوه بر دلایل فوق، اسامی سیگنال‌های داخلی بسیار طولانی می‌شوند. بسیاری از سازندگان ASIC محدودیتهایی در خصوص طول اسامی سیگنال‌ها اعمال می‌کنند. اگر طرح دارای سطحهای متعددی باشد کار آزمایش و تأیید به دلیل آنکه باید سطوح بیشتر مورد بررسی قرار گیرد مشکل‌تر خواهد شد.

بعضی اوقات خصوصاً به هنگام طراحی یک مدل رفتاری و کد VHDL در سطح RT، استفاده از ابزارهای گرافیکی VHDL مفید خواهد بود. با کمک این ابزارها امکان توصیف بعضی توابع و عملکردها به شرح زیر به عنوان مکملی بر کد VHDL فراهم می‌شود. دیاگرام حالت، فلوچارت و جدول درستی به ترتیب در شکلهای ۱۳-۱۱ و ۱۴-۱۱ و جدول ۲-۱۱ نشان داده شده‌اند.



شکل ۱۳-۱۱ دیاگرام حالت



شکل ۱۴-۱۱ فلوچارت

a	b	c
0		0
	0	0
1	1	1

جدول ۲-۱۱ جدول درستی

کد C، کد VHDL برای شبیه‌سازی و یا کد VHDL برای سنتز (RTL) می‌توانند از این توصیفها ساخته شوند. چنان‌چه کد VHDL برای سنتز به‌عنوان خروجی انتخاب شود (در طراحی سخت‌افزاری)، به طور نرمال امکان انتخاب ابزار سنتزی که بهینه‌سازی در ارتباط با آن انجام شود وجود دارد.

برای کسانی که عادت به طراحی با VHDL ندارند این روش توصیفی می‌تواند سریع‌تر از نوشتن مستقیم کد VHDL باشد. لازم به یادآوری است که کلیه ابزارهای گرافیکی VHDL تا حدودی روش توصیف را محدود می‌کنند، به طور مثال در مورد ماشینهای حالت، پیاده‌سازی طرح قدری محدود می‌شود (به فصل ۹ مراجعه کنید، «ماشینهای حالت»). یکی از بزرگ‌ترین مزایای این ابزار این است که تبدیل به کد VHDL به صورت خوانا و مرتب انجام می‌گیرد و فهم و اصلاح آن را آسان‌تر می‌سازد. اما عیب بزرگ آن این است که طرح را مجدداً وابسته به ابزار و محیط طراحی می‌نماید. از آنجا که این ابزار بر اساس استاندارد IEEE نیست اگر فرضاً لازم شود که طراحی پس از مدتی مثلاً چهار سال اصلاح شود، این ریسک وجود دارد که ابزار، فرمت داده‌ها را عوض کرده و یا خود ابزار کلاً تغییر کرده باشد.

چنان‌چه کد VHDL به‌طور ناصحیح نوشته شده‌باشد، کار سنتز به‌طور قابل ملاحظه‌ای سخت‌تر می‌شود. کلیه محدودیتهای زمانی باید در ابزار سنتز تعیین و تنظیم شوند. در مورد طراحی‌های عظیم امکان سنتز یک باره کل طرح وجود ندارد. به جای این کار باید هر زیر بلوک را به طور جداگانه سنتز کرد. این بدان معناست که باید برای هر بلوک محدودیتهای زمانی کلیه ورودی‌ها و خروجی‌ها تنظیم شود. چنان‌چه به توصیه‌های ذکر شده توجه شود سنتز مدار آسان‌تر خواهد شد.

پیشنهادات

۱- طراحی ۱۰۰ درصد سنکرون باشد.

۲- تنها از یک پالس ساعت در هر بلوک استفاده شود.

۳- خروجی هر بلوک باید مستقیماً از طریق یک فلیپ فلاپ در دسترس باشد.

۴- هر بلوک باید حداقل مشتمل بر ۵۰۰ گیت باشد.

۵- از طراحی‌های **چند سیکلی**[1] اجتناب شود.

1- Multicycle

چنان‌چه مورد سوم موردنظر باشد، نباید سطوح سلسله مراتبی به کار رفته خیلی کوچک باشد زیرا که منجر به خط لوله‌ای شدن سیگنال‌ها و هدایت آنها به سوی تأخیر زمانی غیرضروری می‌شود. این امر باعث گنجاندن مورد چهارم شده است. پیشنهاد چهارم نیز با دیگر پیشنهادات هماهنگی دارد. باید سعی کرد که هر سیگنال خروجی مستقیماً به دست آید زیرا این عمل در مورد طراحی‌های کلاک‌دار باعث می‌گردد محدودیتهای زمانی به آسانی انجام‌پذیر گردند. به طور مثال چنان‌چه دوره تناوب پالس ساعت ۲۵ نانوثانیه و تأخیر هر فلیپ فلاپ ۱ نانوثانیه باشد، کلیه خروجی‌ها باید به مدت ۲۴ نانوثانیه تا رسیدن کلاک بعدی بر روی لبه خروجی باقی بمانند. برای کلیه ورودی‌هایی که از دیگر بلوک‌ها با کلاک یکسان می‌آیند، محدودیت زمانی می‌تواند به صورت خلاصه این طور بیان شود : ارزش منطقی باید حداکثر به مدت ۲۴ نانوثانیه در داخل هر بلوک باقی بماند. در عوض چنان‌چه خروجی‌ها از بلوک‌های ترکیبی بزرگ بیایند تشخیص محدودیتهای زمانی قبل از سنتز بسیار مشکل خواهد بود. همچنین استفاده از چند پالس ساعت مجزا در سطح سلسله مراتبی یکسان توصیه نمی‌شود. آنچه که به نام طراحی چند سیکلی معروف است منجر به ایجاد مشکلاتی در سنتز خواهد شد. طراحی چند سیکلی به این معناست که یک سیگنال نباید بعد از یک پالس ساعت بلکه پس از چند پالس ساعت پایدار شود. تعداد پالس‌های ساعت از قبل تعیین می‌شود.

مثالی از طراحی چند سیکلی به صورت زیر است :

```
q ⇐ (a * b) * c;
```

فرض کنید که سیگنال‌های a ، b و c مستقیماً از یک رجیستر بیایند و سیگنال q به یک رجیستر دیگر منتقل شود. چنان‌چه دوره تناوب پالس ساعت ۲۵ نانوثانیه باشد و عمل ضرب حداکثر ۴۰ نانوثانیه زمان لازم داشته باشد، یک طرح چند سیکلی با دو سیکل پالس ساعت انجام گرفته است. از ساده‌ترین روشهای عبور از این مشکل ذخیره نتیجه a * b در یک رجیستر و سپس انجام عمل ضرب با c است. البته این کار منجر به تأخیر به اندازه یک پالس ساعت خواهد شد. به کارگیری ابزار سنتز رفتاری غالباً راه حل مناسبی برای این‌گونه مسائل خواهد بود. ابزارهای رفتاری با تعیین شرایط طراحی چند سیکلی و تحصیل نتیجه صحیح از سنتز از عهده وظیفه‌ای که بعضاً سخت می‌نماید به خوبی برخواهند آمد (به فصل ۱۷ مراجعه کنید، «سنتز رفتاری»).

نکات فوق ممکن است بیش از حد لازم سخت و اکید به نظر بیایند، تا آنجا که تعداد زیادی از طراحان حاضر نمی‌شوند زحمت آنها را در طراحی‌های اولیه خویش به خود بدهند. اگر مدار کوچک باشد، اشکال خاصی از این موارد به وجود نمی‌آید ولی چنان‌چه طرح عظیم و بزرگ باشد (بیشتر از ۱۵۰۰۰ گیت) اثر و نتیجه آنها جدی‌تر خواهد بود. مثالهای فراوانی از طراحی‌های گوناگون وجود دارد که به علت بی‌توجهی به پیشنهادات فوق به انجام نرسیده‌اند.

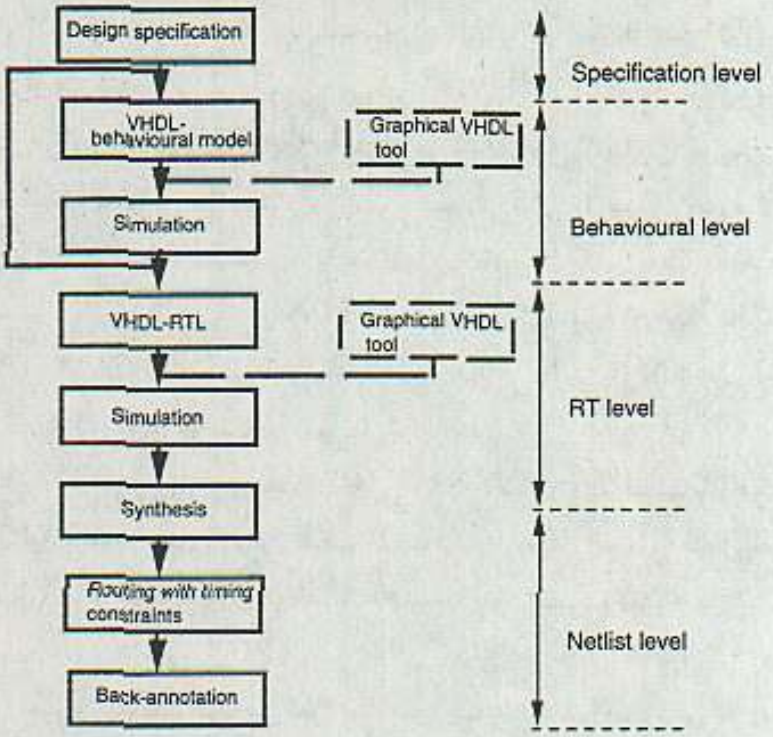
طراحی آسنکرون معایب بسیاری دارد که به ده مورد از مهم‌ترین آنها اشاره می‌کنیم :

۱- طرح وابسته به تکنولوژی و چگونگی طرح‌ریزی مدار می‌شود.
۲- قوانین ATPG رعایت نشده و در نتیجه جلوگیری از خطا به‌طور قابل ملاحظه‌ای کاهش می‌یابد (به فصل ۱۲ مراجعه کنید، «آشنایی با روشهای آزمایش»).
۳- اغلب ابزارهای آزمایش و تست ASIC، طرحهای آسنکرون را حمایت نمی‌کنند.
۴- پشتیبانی از سوی ابزارها بسیار کم است، به عبارت دیگر ابزارهای سنتز رفتاری آنها را حمایت نمی‌کنند.
۵- شبیه‌سازی و پیدا کردن نقایص و مشکلات مدار مشکل‌تر است.
۶- اصلاح و تغییر طرح بسیار مشکل می‌شود.
۷- ابزارهای سنتز نمی‌توانند طرح را بهینه کرده و نتیجه را تحلیل زمانی کنند.
۸- عملکرد طرحهای آسنکرون اغلب به دما و ولتاژ تغذیه وابسته خواهد بود.
۹- بررسی مدل نمونه پیاده‌سازی شده در FPGA بسیار مشکل خواهد بود.
۱۰- فاصله زمانی تعیین مشخصات پروژه تا اتمام طراحی مدار با ضریب بین ۱/۵ تا ۱۰ برابر افزایش می‌یابد.

از جانب دیگر، مزیت طرحهای آسنکرون سریع‌تر بودن و کم‌مصرف‌تر بودن آنها در مقایسه با طرحهای سنکرون است. معایب این گونه مدارات بسیار قابل توجه‌تر از مزیتهای آنهاست و از این رو توصیه می‌شود که تا حد امکان از طراحی‌های آسنکرون خودداری شود. از آنجا که یک طرح آسنکرون باعث بروز اشکالات زیادی می‌شود و حمایت کافی از سوی ابزارهای قوی سنتز نیز برای این گونه طرحها وجود ندارد، زمان طراحی یک پروژه ASIC که به روش آسنکرون طراحی می‌شود بین ۱/۵ تا ۱۰ مرتبه بیشتر از طراحی مدارات سنکرون خواهد بود. البته اگر ابزارهایی در آینده تولید شوند که طراحی آسنکرون را حمایت کنند تصویر قضیه متفاوت خواهد بود. در دهه ۱۹۹۰ روند کار بیشتر به سوی طراحی‌های سنکرون بوده است.

FPGA ۱۱-٤

طراحی بالا به پایین که در مورد طرحهای ASIC گفته شد، برای استفاده در طراحی‌های FPGA تفاوت زیادی ندارد (شکل ۱۵-۱۱). تا قبل از سنتز تفاوتی بین ASIC و FPGA وجود ندارد. یک تفاوت عمده در این است که لازم نیست بردارهای آزمایش را برای یک FPGA تولید نمود، زیرا FPGA ها معمولاً توسط خود سازنده چک و آزمایش می‌شوند.



شکل ۱۵-۱۱ طراحی بالا به پایین برای FPGA ها

این امکان وجود دارد که از FPGA ها به عنوان ماتریس‌های گیتی منتهی به فرم کمی ساده‌تر شده استفاده کرد. FPGA ها در بسته‌های کوچک‌تر ارزان‌ترند در حالی که ماتریس‌های گیتی هر چه بزرگ‌تر باشند ارزان‌تر می‌شوند. FPGA ها از نوع Xilinx شامل تعداد زیادی فلیپ فلاپ هستند در حالی که انواع دیگر FPGA ها منطقی از نوع Actel دارند.

از آنجا که روند طراحی FPGA مشابه ASIC است، بنابراین امکان ***نگاشت***[1] و سنتز طرح VHDL ابتدا در یکFPGA و سپس استفاده از آن به‌عنوان نمونه اولیه وجود دارد. پس از بررسی نمونه اولیه پیاده‌سازی شده‌در FPGA، می‌توان طرح را با کمک ابزارهای سنتز در یک ASIC پیاده‌سازی کرد. یک ASIC معمولاً هزینه تولید بسیار بالایی دارد. اگر بتوان ASIC را در ابتدا با یک یا چند FPGA تحت آزمایش و تست قرار داد، ریسک طراحی مجدد و در نتیجه هزینه‌های بالایی که به دنبال آن می‌آید تا اندازه زیادی کاهش می‌یابد. از سوی دیگر FPGA ها و نسبت به ASIC ها سرعت بسیار کمتری دارند. این بدان معناست که اغلب اوقات نمی‌توان سیستم را در سرعت ماکزیمم خود آزمایش کرد. البته این موضوع مشکل مهمی نیست زیرا می‌توان عملکرد سیستم را با پالس‌های ساعت با

1- Map

فرکانس کمتر مورد آزمایش قرار داد. معمولاً سیستم تحت شرایطی مانند دمای اتاق و ولتاژ تغذیه ۵ ولت مورد بررسی قرار می‌گیرد. این امر موجب می‌شود FPGA با سرعتی به مراتب بیشتر از آنچه در گزارش زمانی حاصل از سیم‌بندی داخل FPGA اعلام می‌شود، عمل کند. این گزارش زمانی معمولاً بر پایه بدترین شرایط است. بنابراین در بعضی موارد سیستم می‌تواند با حداکثر سرعت خود با FPGA مورد بررسی واقع شود. چنان‌چه سیستم تنها در دمای اتاق کار کند و در دماهای بالاتر فعالیت صحیحی نداشته باشد، نمی‌توان آن را در FPGA پیاده‌سازی کرد و به خریدار تحویل داد. هدف از ایجاد نمونه اولیه با کمک FPGA بررسی کارکرد سیستم قبل از تولید ASIC است. مسلماً ASIC باید به گونه‌ای طراحی گردد که بتواند در بدترین شرایط عمل کند.

ارتباط بین ابزار سنتز و ابزار سیم‌بندی داخلی FPGA بسیار مهم است. با وجود یک ابزار سنتز و سیم‌بندی خوب کلیه محدودیتهای زمانی تنظیم شده در ابزار سنتز به طور کامل به ابزار سیم‌بندی منتقل می‌شوند. سپس ابزار سیم‌بندی با توجه به محدودیتهای زمانی واقعی عمل سیم‌بندی را انجام می‌دهد. پس از سیم‌کشی داخلی FPGA، زمان‌بندی مدار در بخش تحلیل‌گر زمانی ابزار سنتز یادداشت و مورد بررسی و آزمایش قرار می‌گیرد تا معلوم نماید آیا شرایط زمانی به خوبی برقرار شده است یا خیر. این روش سیم‌کشی داخلی FPGA ها به‌نام طرح‌ریزی FPGA بر اساس زمان‌بندی خوانده می‌شود و چنان‌چه FPGA ها در طراحی صحیح روش بالا به پایین به کار برده شوند باید سیم‌بندی داخلی آنها به این روش انجام گیرد.

فصل دوازدهم

آشنایی با روشهای آزمایش

در هنگام طراحی و ساخت ASIC برای اطمینان از صحت عملکرد آن لازم است آزمایشهایی انجام گیرد. این آزمایش معمولاً توسط سازنده بلافاصله پس از تولید مدار انجام می‌شود و نباید آن را با شبیه‌سازی که طراح قبل از ساخت بر روی طرح انجام می‌دهد اشتباه گرفت. در اینجا طراح سعی می‌کند معلوم نماید که آیا مدار مطابق مشخصات موردنیاز طراحی شده است یا خیر. اما از سوی دیگر سازنده ASIC تنها به این موضوع علاقه‌مند است که آیا مدار ساخته شده با netlist هایی که طراح به عنوان مبنای ساخت تنظیم کرده منطبق است یا خیر. این کار توسط بردارهای آزمایش که طراح به هنگام طراحی مدار تهیه می‌کند انجام می‌گیرد. پاسخ مدار ساخته شده به بردارهای آزمایش معلوم می‌کند که آیا ASIC را می‌توان به خریدار تحویل داد یا آنکه باید آن را به دور انداخت. بنابراین مؤثر بودن بردارهای آزمایش بسیار مهم است. این فصل به بررسی چگونگی تولید خودکار بردارهای مؤثر و کارآمد آزمایش و قواعدی که باید در کد VHDL به منظور تولید آنها رعایت شود می‌پردازد. در این فصل همچنین به روشهای گوناگون آزمایش، نحوه ***اسکن‌های مرزی***[1] و بردارهای تکمیلی آزمایش آشنا می‌شویم.

1- Boundary Scans

اگر به دهه ۱۹۸۰ بازگردیم می‌بینیم که در آن زمان معمولاً بردارهای آزمایش به طور دستی ساخته می‌شدند. بخشی از بردارهایی که برای تأیید درستی عمل مدار (شبیه‌سازی) به کار گرفته می‌شد را دوباره برای آزمایش مدار استفاده می‌کردند. همچنین برای کنترل هر چه بیشتر خطا بردارهای آزمایش جداگانه‌ای نوشته می‌شد. معمولاً بین ۷۰ تا ۷۵ درصد از خطاها نسبتاً آسان تحت کنترل در می‌آمد. اگر طرح موردنظر از لحاظ اندازه ۱۵۰۰۰ گیت داشت برای نوشتن بردارهایی که تمام نیازمندی‌های مدار را بتواند برآورده نماید به چهار یا پنج هفته نیاز بود. به ندرت کنترل خطا به ۸۵ درصد می‌رسید. اگر سازنده ASIC تغییر می‌کرد، فرمت بردارهای آزمایش مورد استفاده توسط سازنده قبلی معمولاً قابل قبول نبود.

در دهه ۱۹۹۰ این روند رفته رفته به سمت خودکار شدن حرکت کرده و تولید ماشین بردارهای آزمایش با استفاده از ***ATPG***[1] به طور روزافزونی رواج پیدا کرد. با استفاده از ATPG کنترل خطاهای سیستم تا بیش از ۹۹ درصد در ظرف یک یا دو روز امکان‌پذیر می‌گردد. زمانی که به مقایسه مقادیر کنترل خطا می‌پردازیم باید دقت نماییم که از همان تعریف‌هایی که برای نقایص در مدار داشته‌ایم استفاده نماییم. نقایصی که ابزارهای ATPG کشف می‌کنند اصطلاحاً به ***گیر کردن در صفر***[2] و ***گیر کردن در یک***[3] معروفند. «گیر کردن در یک» به این معناست که یک گره داخلی به ولتاژ ۵ ولت اتصال کوتاه شده باشد. این نوع نقصها معمولاً سهم بزرگی از نقایص محصولات ASIC را به خود اختصاص می‌دهند. به طور کلی تست محصول باید با بردارهای آزمایش مکمل بردارهای ATPG همراه باشد. در ادامه این فصل به توضیح نکات بیشتری در این باره می‌پردازیم.

شکل ۱-۱۲ روند آزمایش با ATPG را نشان می‌دهد.

۱-۱۲ روشهای اسکن

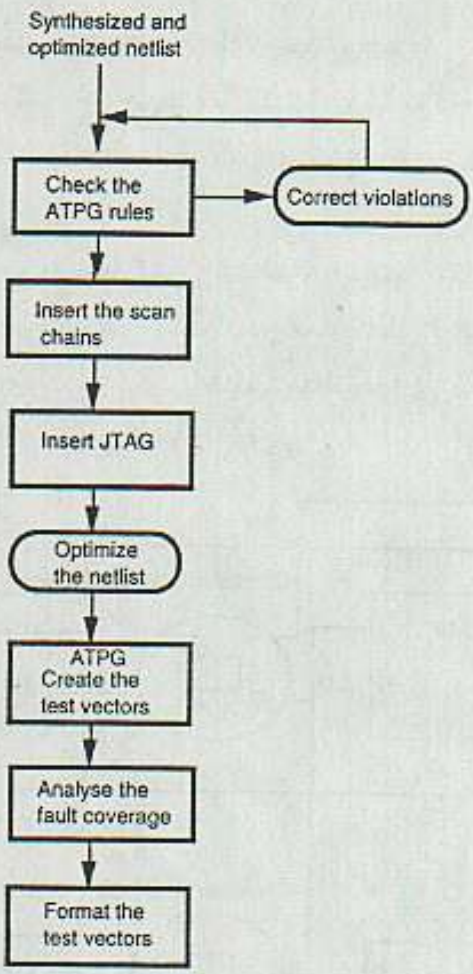
اگر بخواهیم از ATPG استفاده نماییم باید از آنچه به نام ***زنجیره‌های اسکن***[4] خوانده می‌شود، استفاده کنیم. از زنجیره‌های اسکن می‌توان برای کشف نقایصی مثل «گیر کردن در» که قبلاً به آنها اشاره شده استفاده کرد. فرض کنید در طرح ساده‌ای که در شکل ۲-۱۲ نمایش داده شده ورودی داخلی گیت NAND به ولتاژ ۵ ولت متصل شده باشد.

1- Automatic Test Pattern Generator

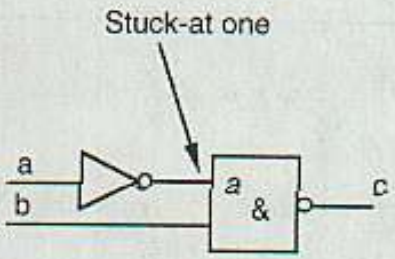
2- Stuck-at Zero

3- Stuck-at One

4- Scan Chain



شکل ۱-۱۲ روند تولید بردار آزمایش ATPG

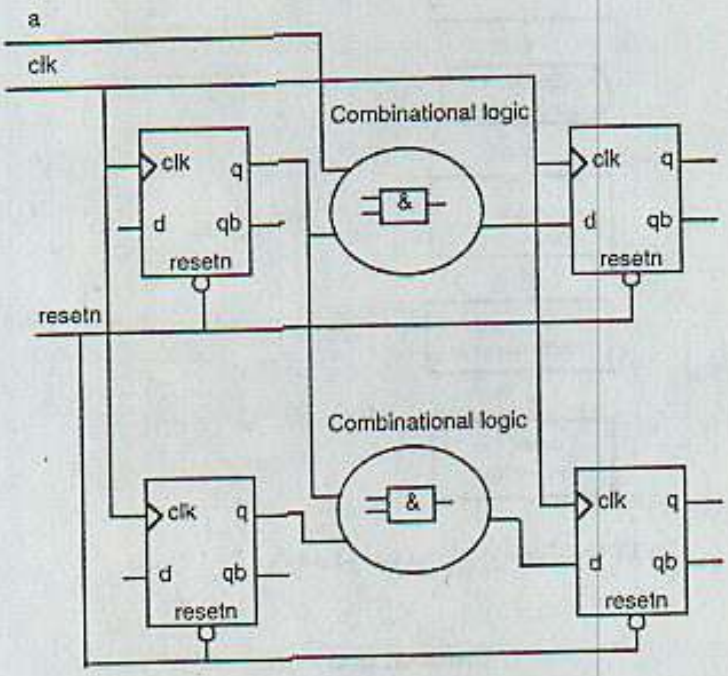


شکل ۲-۱۲ گیر کردن در یک

این نقص با اعمال مقادیر a = 1 و b = 1 به سیگنال‌های ورودی مشخص خواهد شد. انتظار می‌رود که سیگنال خروجی برابر 1 گردد. چنان‌چه مدار فوق در مسیر تولید نقصی پیدا کرده باشد، سیگنال خروجی مقدار '0' را نشان خواهد داد. این بدان معناست که مدار معیوب است.

چنان‌چه قرار باشد عیوب و نقایص مدارهای بسیار پیچیده‌تر از مثال ساده فوق را با همین تکنولوژی تشخیص دهیم باید کلیه خروجی‌ها قابل کنترل و از بلوک‌های ترکیبی داخلی مدار قابل مشاهده باشند. یک طرح عادی معمولاً از تعدادی فلیپ فلاپ و منطق ترکیبی بین آنها تشکیل می‌شود (به شکل ۳-۱۲ مراجعه کنید).

فرض کنید که کلیه فلیپ فلاپ‌ها کنترل شده‌اند و همچنین مقدار هر یک از فلیپ فلاپ‌ها از خارج قابل دیدن باشد. در این صورت امکان تست کردن تمام منطق ترکیبی داخل مدار با قاعده گفته شده در مثال قبل وجود دارد. برای اجرای تست باید فلیپ فلاپ‌ها قابل کنترل و قابل رؤیت باشند :



شکل ۳-۱۲ یک طرح نوعی

- **قابل کنترل بودن** : یک گره داخلی مدار وقتی قابل کنترل است که بتوان آن را به هر ارزشی مقداردهی کرد.
- **قابل رؤیت بودن** : یک گره داخلی مدار وقتی قابل رؤیت است که بتوان ارزش آن را پیش‌بینی کرده و سپس برای چک کردن مقدار آن را به خارج از مدار منتقل نمود.

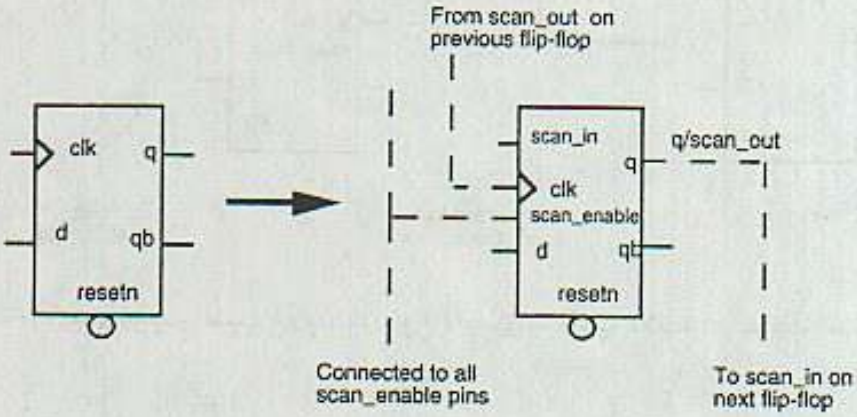
موارد فوق نکاتی هستند که با گنجاندن زنجیره‌های اسکن امکان‌پذیر می‌شوند.

اسکن کردن با ATPG روشهای مختلفی دارد که متداول‌ترین آنها عبارتند از :

- *فلیپ فلاپ مرکب*[1]
- اسکن پالسی
- LSSD

اسکن با فلیپ فلاپ مرکب

ساده‌ترین و متداول‌ترین روش، عوض کردن فلیپ فلاپ‌های مدار با فلیپ فلاپ‌هایی است که اصطلاحاً مرکب نامیده می‌شوند.



شکل ۴-۱۲ جای‌گزینی فلیپ فلاپ‌های مدار با فلیپ فلاپ‌های اسکن مرکب

در اسکن مرکب می‌توان با کمک ورودی scan_enable تعیین کرد که کدام یک از ورودی‌ها (d یا scan_in) به داخل فلیپ فلاپ راه یابد. در مد عملکردی مدار scan_enable به ولتاژ 0 ولت متصل می‌شود. در تست با ATPG، مد آزمایش نیز به کار خواهد آمد.

اسکن با فلیپ فلاپ مرکب برای اتصال به پایه scan_enable نیاز به یک مدار جانبی دارد. در ضمن فلیپ فلاپ‌های مرکب به اندازه سه گیت از فلیپ فلاپ‌های معمولی بزرگ‌تر هستند.

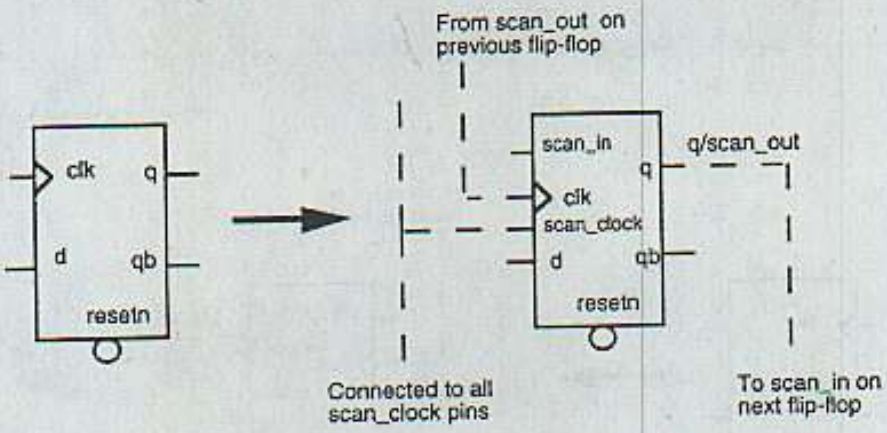
اسکن پالسی

فلیپ فلاپ‌های اسکن پالسی دارای پالس‌های ساعت جداگانه برای ورودی scan_in می‌باشند. بسته به اینکه کدام پالس ساعت فعال شود، داده از طریق ورودی d و یا scan_in وارد فلیپ فلاپ

1- Multiplexed Flip-Flop

خواهد شد. بنابراین فلیپ فلاپ‌های اسکن پالسی نیازی به ورودی scan_enable ندارند. مثالی از اسکن پالسی در شکل ۵-۱۲ نشان داده شده است.

اسکن پالسی نیاز به یک مدار جانبی برای اتصال به پایه scan_clock دارد. در ضمن فلیپ فلاپ‌های مرکب اسکن پالسی معمولاً به اندازه سه گیت بزرگ‌تر از فلیپ فلاپ‌های معمولی می‌باشند.



شکل ۵-۱۲ جای‌گزینی فلیپ فلاپ‌های نوع D مدار با فلیپ فلاپ‌های مرکب اسکن پالسی

LSSD

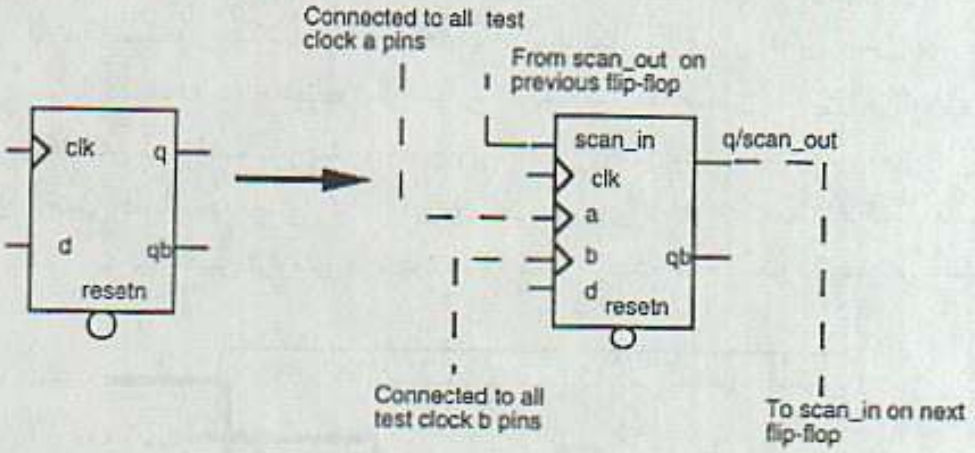
فرمهای گوناگونی از ***LSSD***[1] وجود دارد. LSSD برای مداراتی که حاوی فلیپ فلاپ و یا لچ باشند به کار می‌رود. تفاوت فلیپ فلاپ و لچ در مد عملکردی آنهاست، در مد آزمایش فلیپ فلاپ اسکن و لچ رفتار مشابهی خواهند داشت. یک فلیپ فلاپ اسکن LSSD دارای دو ورودی پالس ساعت مجزا (a و b) برای کلاک زدن سیگنال ورودی scan_in به داخل فلیپ فلاپ است. در داخل، فلیپ فلاپ شامل یک ***لچ غالب***[2] و یک ***لچ مغلوب***[3] است. کلاک‌های a و b که به پالس‌های ساعت دو فاز معروفند، نباید هیچ گونه روی هم افتادگی یا سطح مشترک داشته باشند. به عبارت دیگر نباید به طور هم‌زمان فعال شوند. فعال شدن کلاک b منجر به انتقال مقدار از scan_in به لچ غالب می‌شود. اگر پس از آن کلاک a فعال شود، این مقدار از لچ غالب به لچ مغلوب و همچنین به خروجی q/scan_out انتقال می‌یابد. کلاک‌های a و b در مد عملکردی فعال نخواهند بود. در مد عملکردی، فلیپ فلاپ‌ها به مانند یک فلیپ فلاپ نرمال با یک کلاک معمولی و خروجی d عمل می‌کنند. یکی از مزایای این روش اسکن

1- Level Sensitive Scan Design

2- Master Latch

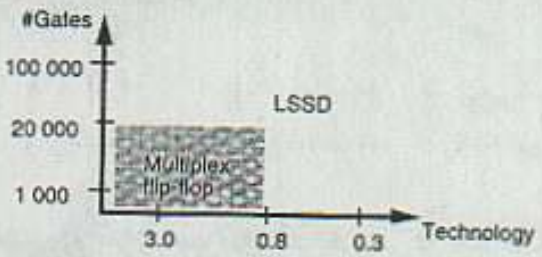
3- Slave Latch

که در آن از یک کلاک دو فاز استفاده می‌شود، حساس نبودن طرح به شیب کلاک در مد تست است. از معایب این نوع اسکن بزرگ‌تر بودن فلیپ فلاپ‌ها به اندازه یک تا سه گیت و معمولاً کمی کندتر بودن آنها نسبت به فلیپ فلاپ‌های مرکب است. نقص دیگر، نیاز مدار به دو پالس ساعت اضافی (a و b) است. مثالی از این نوع اسکن در شکل ۶-۱۲ نشان داده شده است.



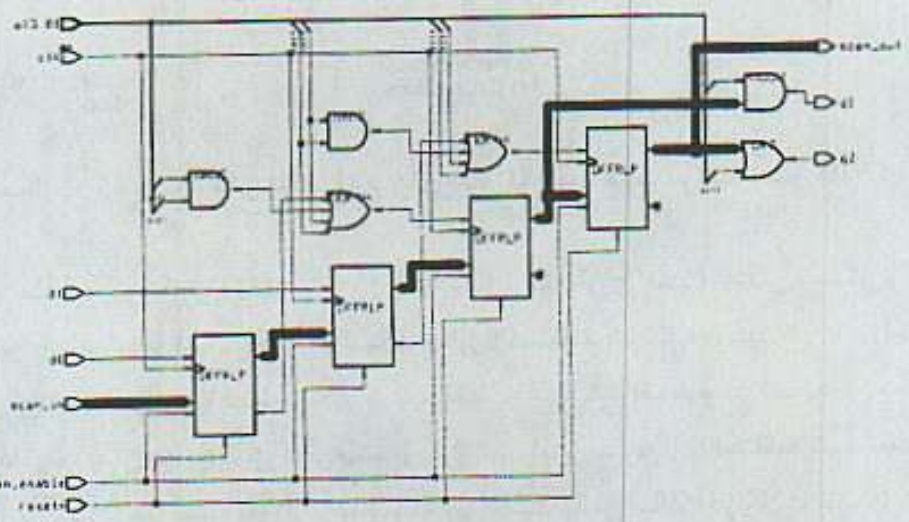
شکل ۶-۱۲ جای‌گزینی فلیپ فلاپ‌های مدار با فلیپ فلاپ مرکب اسکن LSSD

برای مدارات کوچک‌تر (کمتر از ۲۰۰۰۰ گیت) و تکنولوژی نسبتاً کندتر (بیشتر از ۰/۸ µm) توصیه می‌شود از اسکن با فلیپ فلاپ مرکب (نوع اول) استفاده شود (به شکل ۷-۱۲ مراجعه کنید). این به دلیل آن است که فلیپ فلاپ اسکن مرکب تنها به یک پایه اضافه و در نتیجه به حجم کمتری نیاز خواهد داشت. چنان‌چه مدار بزرگ‌تر باشد و یا از تکنولوژی سریع‌تری استفاده شود بعضی از انواع کلاک دو فاز باید برای مد تست به کار برده شوند، زیرا شیب پالس ساعت در کلاک تک فاز غالباً باعث ایجاد مکث فلیپ فلاپ‌ها و بی‌نظمی در مد تست می‌شود. علت این امر اتصال یافتن فلیپ فلاپ‌ها به یک شیفت رجیستر طولانی است و همین نکته فلیپ فلاپ‌ها را به شیب پالس ساعت بسیار حساس می‌سازد. به فصل ۱۱ مراجعه کنید، «آشنایی با روشهای طراحی».



شکل ۷-۱۲ روش‌های اسکن

چنان‌چه فلیپ فلاپ‌های مدار با فلیپ فلاپ‌های اسکن (مرکب) تعویض شوند، فلیپ فلاپ‌ها قابل کنترل و قابل رؤیت می‌شوند. کلیه فلیپ فلاپ‌های اسکن در مد عملکردی دقیقاً مشابه فلیپ فلاپ‌های معمولی عمل می‌کنند. اما در مد، تست عملکرد فلیپ فلاپ‌ها تغییر می‌کند. چنان‌چه فلیپ فلاپ‌ها در یک زنجیره طویل اسکن به هم متصل شوند، امکان انتقال مقادیر مشخص به فلیپ فلاپ‌ها با فعال کردن scan_enable (در اسکن مرکب) وجود دارد. این مقادیر را می‌توان با تغییر به مد عملکردی از تمام بلوک‌های ترکیبی به داخل فلیپ فلاپ‌ها نمونه‌سازی کرد (قابل کنترل بودن). اگر در این زمان سیگنال scan_enable دوباره فعال شود محتویات فلیپ فلاپ‌های اسکن را می‌توان برای چک کردن مقدارهایشان از شیفت ـ رجیستر خارج کرد (قابل رؤیت بودن). این روش اجازه می‌دهد که بلوک‌های ترکیبی داخل مدار حتی اگر ورودی‌ها و خروجی‌های داخلی در منطق ترکیبی مستقیماً در دسترس نباشد، مورد آزمایش قرار گیرند. شکل ۸-۱۲ مثالی از یک طرح اسکن را نشان می‌دهد.



شکل ۸-۱۲ زنجیره اسکن

کار ابزار ATPG ساختن بردارهای آزمایش برای کشف نقایص مثل «گیر کردن در» است. یک بردار آزمایش از بخشهای زیر تشکیل می‌شود :

- مجموعه‌ای از '1' ها و '0' ها که به ورودی‌های مدار اعمال می‌شوند.
- بخشی که مقادیر و ارزشهای مورد انتظار در خروجی‌ها را محاسبه می‌کند.

این ابزار ATPG است که مقادیر مورد انتظار خروجی‌ها را با توجه به مقادیر اعمال شده به ورودی‌ها محاسبه می‌کند. سپس این مقدارها با مقادیر تولید شده توسط مدار مقایسه می‌شوند.

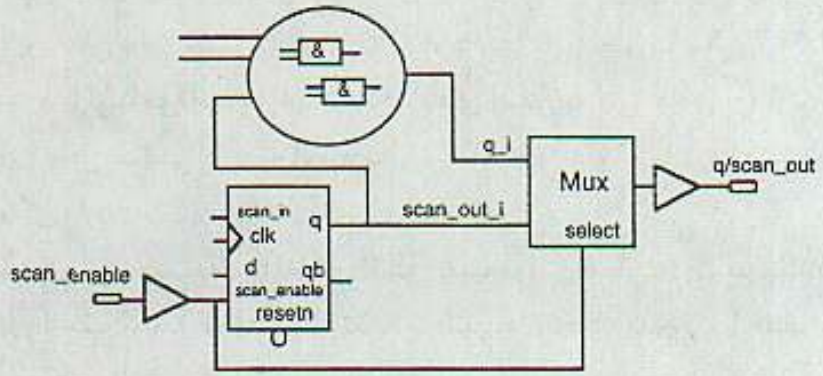
چنان‌چه این مقادیر با هم تطبیق داشته باشند، مدار مرحله تست و آزمایش را با موفقیت پشت سر گذاشته است.

اصول و نحوه عمل ATPG را می‌توان در پنج مرحله زیر خلاصه کرد :

۱- مدار در مد اسکن (فعال شدن scan_enable) قرا می‌گیرد و مقادیر جدید به داخل زنجیره اسکن وارد می‌گردد.
۲- مدار در مد عملکردی قرار می‌گیرد و مقادیر مشخص به ورودی‌ها اعمال می‌گردد.
۳- کلاک سیستم یک نوبت فعال می‌شود تا از کلیه بلوک‌های منطقی نمونه‌برداری کند.
۴- مقادیر سیگنال‌های خروجی مدار چک می‌شوندکه آیا دارای مقادیر مورد انتظار هستند یا خیر.
۵- scan_enable مجدداً فعال می‌شود، مقادیر فلیپ فلاپ‌ها به خروجی آنها (scan_out) انتقال می‌یابد. در آنجا این مقادیر چک می‌شوند تا معلوم شود که آیا مقادیر مورد نظر را دارند یا خیر.

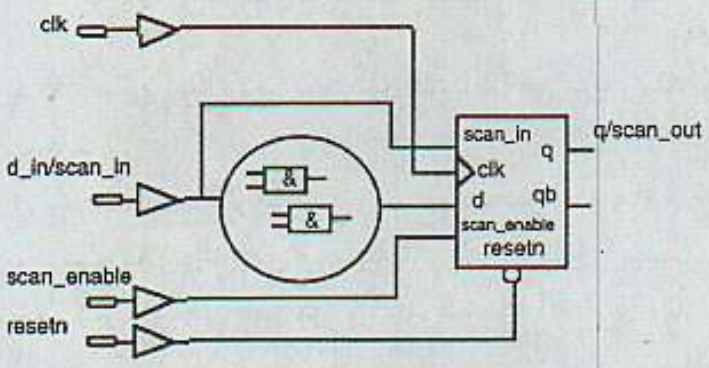
معمولاً این پنج فاز روی هم می‌افتند، یعنی مقادیر جدید در همان لحظه‌ای وارد زنجیره اسکن می‌شوند که مقادیر قدیمی در حال خروج از آن هستند.

مقدار مربوط به پایه scan_in اولین فلیپ فلاپ معمولاً از یک ورودی آزمایشی جداگانه وارد مدار می‌شود (scan_in). سیگنال خروجی آخرین فلیپ فلاپ نیز مستقیماً از مدار خارج شده و به یک پین آزمایش می‌رود (scan_out). در مداری که کمبود پایه وجود دارد، خروجی scan_out می‌تواند با خروجی معمولی مدار از طریق یک مالتی‌پلکسر یکی شود. در این حالت مالتی‌پلکسر باید توسط سیگنال scan_enable کنترل گردد (شکل ۹-۱۲).



شکل ۹-۱۲ scan_out مالتی پلکس شده

ورودی scan_in نیز می‌تواند با ورودی معمولی مدار به منظور صرفه‌جویی در پایه مشترک شوند (شکل ۱۰-۱۲).



شکل ۱۰-۱۲ scan_in در پایه ورودی معمولی

تعویض و اتصال تمام فلیپ فلاپ‌های اسکن در یک طرح کار آسانی نیست. بنابراین می‌توان با خرید ابزارهای جدید و ضمیمه کردن آنها به ابزارهای فعلی سنتز این کار را آسانتر نمود. این نرم‌افزارها به طور خودکار ضمیمه زنجیره‌های اسکن شده و آنها را به خوبی به هم مرتبط می‌سازند. از آنجا که تعداد بردارهای آزمایش متناسب با طول طویل‌ترین زنجیره اسکن تعیین می‌شود، بنابراین در طرحهای بزرگ همیشه چند زنجیره موازی مورد استفاده قرار می‌گیرد. این کار بسیار واجب است زیرا سازندگان ASIC به طور عادی فقط تعداد محدودی بردار تست و آزمایش را در سیستم خود منظور می‌کنند. در ابزارهای ضمیمه پیشرفته‌تر امکان تعریف تعداد زنجیره‌های اسکن و یا تعیین حداکثر طول آنها به هنگام ضمیمه شدن به زنجیره‌های اسکن وجود دارد. فرض کنید که یک سیگنال ATPG، N عدد بردار آزمایش را برای طرح ارائه شده در شکل ۳-۱۲ تولید کند. اگر ابزار ضمیمه حاوی این دستور باشد که دو زنجیره موازی را به جای یک زنجیره وارد عمل نماید تعداد بردارهای آزمایش نصف شده و به N/2 خواهد رسید. این بدان علت است که ابزار ATPG داده‌ها را به صورت موازی به داخل زنجیره‌های موازی می‌نویسد و یا از داخل آنها می‌خواند. زنجیره‌های اسکن به طور عادی ۱۰۰ تا ۵۰۰ فلیپ فلاپ طول دارند و این طول بستگی دارد به اندازه مدار و تعداد بردارهای آزمایش که سازنده ASIC منظور کرده است، بستگی دارد.

همچنین در ابزار ضمیمه‌ای آزمایش امکان تعیین روش اسکن با کمک یک دستور ساده و یا انتخاب از داخل لیست برنامه وجود دارد. تعویض فلیپ فلاپ‌های معمولی با فلیپ فلاپ‌های اسکن (مرکب) برای ابزارهایی از این نوع، بسیار ساده و سریع است. به طور مثال این روند برای مداری با ۱۰۰۰۰ گیت تنها در چند دقیقه انجام می‌شود، در حالی که تولید بردارهای آزمایش توسط ATPG

برای همین مدار تقریباً ۱۰ دقیقه زمان لازم خواهد داشت. از سوی دیگر در مورد مدارات بزرگ‌تر (بیشتر از ۱۰۰۰۰۰ گیت) زمان مورد نیاز برای تولید بردارهای آزمایش با کمک ATPG خیلی طولانی‌تر (بیشتر از ۱۰ ساعت) خواهد بود.

۱۲-۲ اسکن کامل[1] و اسکن جزئی[2]

دو نوع الگوریتم ATPG وجود دارد. یکی بر این اساس است که کلیه فلیپ فلاپ‌های مدار در زنجیره اسکن به یکدیگر متصل شوند، که به نام اسکن کامل معروف است. در دیگری که اسکن جزئی نام دارد تنها زیرمجموعه‌ای از فلیپ فلاپ‌ها در زنجیره اسکن وارد می‌شوند. درصد گیت‌هایی که برای هر کدام از الگوریتم‌ها موردنیاز است از طرح به طرح فرق می‌کند ولی با یک تخمین کلی می‌توانیم بگوییم:

- اسکن کامل نیاز به ۲۰-۱۰ درصد گیت‌های اضافی دارد.
- اسکن جزئی نیاز به ۱۵-۵ درصد گیت‌های اضافی دارد.

با ابزار ATPG و این تعداد گیت‌های اضافی سطح کنترل خطاهای سیستم برای اسکن کامل به حدود ۹۹ درصد و برای اسکن جزئی به درصدی نزدیک به آن می‌رسد و علت این تفاوت آن است که در اسکن جزئی کلیه فلیپ فلاپ‌ها در زنجیره اسکن وارد نشده و بنابراین نمی‌توان آنها را چک کرد. البته می‌توان الگوریتم اسکن جزئی را برای رفع این اشکال مقداری اصلاح کرد اما با این حال نمی‌توان آن را به سطح اسکن کامل رسانید. اسکن جزئی را زمانی می‌توان به جای اسکن کامل به‌کار برد که محدودیتهای مکانی و در بعضی موارد محدودیتهای زمانی طرح شدیداً انعطاف‌ناپذیر باشند. همان طور که قبلاً نیز گفته شد فلیپ فلاپ‌های اسکن (مرکب) معمولاً کمی کندتر از فلیپ فلاپ‌های معمولی هستند. بنابراین انتخاب هر یک از الگوریتم‌ها بسته به درجه اولویتی است که برای رعایت محدودیت مکانی و زمانی و یا برای کنترل بیشتر خطا قائل هستیم.

۱۲-۳ قواعد طراحی با ATPG

اگر بخواهیم بیشترین مقدار خطا را تحت کنترل درآوریم باید قواعد طراحی خاصی را علاوه بر استفاده از زنجیره اسکن مورد عمل قرار دهیم. اگر این قواعد نادیده گرفته شوند سطح پوشش خطا به‌طور قابل ملاحظه‌ای افت می‌کند. این قواعد ممکن است بسته به نوع ابزار ATPG و یا زنجیره اسکن

1- Full Scan

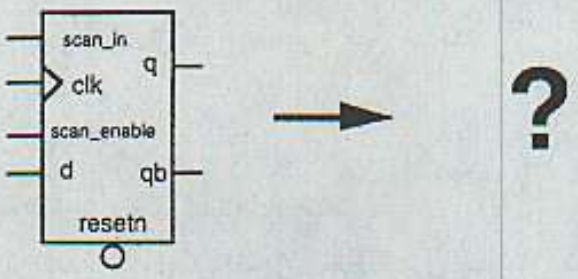
2- Partial Scan

اندکی تغییر یابد اما اصول پایه‌ای در تمام موارد یکی است. در زیر به بررسی مهم‌ترین این قواعد می‌پردازیم.

قاعده ۱ : فلیپ فلاپ‌ها معادل فلیپ فلاپ مرکب داشته باشند.

برای آنکه امکان اتصال به زنجیره اسکن وجود داشته باشد، فلیپ فلاپ‌ها باید معادل فلیپ فلاپ‌های اسکن (مرکب) داشته باشند.

یک مثال از معادل نداشتن فلیپ فلاپ‌ها در شکل ۱۱-۱۲ نمایش داده شده است.

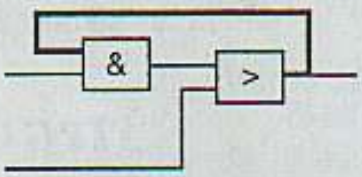


شکل ۱۱-۱۲ نقص نبود معادل

قاعده ۲ : وجود نداشتن حلقه فیدبک ترکیبی در مدار

منطق ترکیبی نباید حلقه فیدبک داشته باشد مگر آنکه فیدبک از مسیر یک فلیپ فلاپ عبور کند. این فیدبک به معنی آسنکرون بودن مدار است.

مثالی از حلقه فیدبک ترکیبی در شکل ۱۲-۱۲ نشان داده شده است.

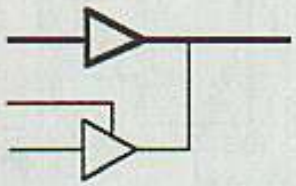


شکل ۱۲-۱۲ حلقه فیدبک ترکیبی

قاعده ۳ : وجود درایورهای سه حالته

اگر یک سیگنال توسط بیش از یک درایور مقدار می‌گیرد، کلیه درایورها باید از نوع سه حالته باشند.

مثالی از اعمال درایور نادرست در شکل ۱۳-۱۲ نشان داده شده است.

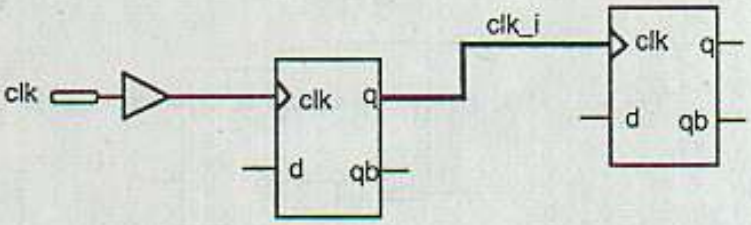


شکل ۱۳-۱۲ درایور غیر سه حالته

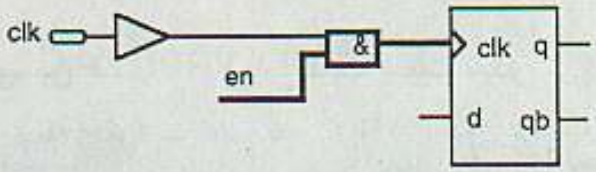
قاعده ۴ : قابل کنترل بودن پالس‌های ساعت

باید امکان کنترل کلاک برای کلیه فلیپ فلاپ‌های داخل مدار از طریق یک ورودی در سیستم وجود داشته باشد.

در مورد غیرقابل کنترل بودن کلاک برای یک فلیپ فلاپ داخلی دو مثال در شکلهای ۱۴-۱۲ و ۱۵-۱۲ آورده شده است.



شکل ۱۴-۱۲ کلاک غیرقابل کنترل ـ مورد ۱

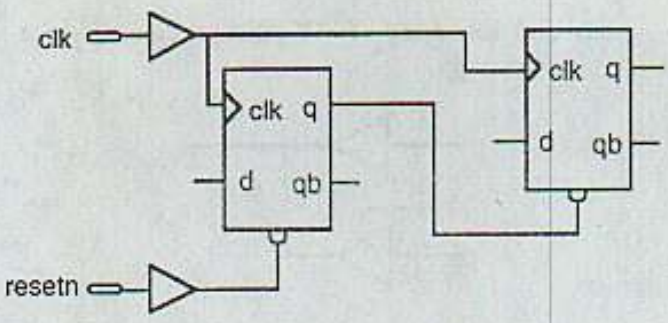


شکل ۱۵-۱۲ کلاک غیرقابل کنترل ـ مورد ۲

قاعده ۵ : قابل کنترل بودن reset آسنکرون

باید امکان کنترل reset آسنکرون برای کلیه فلیپ فلاپ‌های داخل مدار از طریق یک ورودی در سیستم وجود داشته باشد.

مثالی از غیرقابل کنترل بودن reset آسنکرون در شکل ۱۶-۱۲ نشان داده شده است.

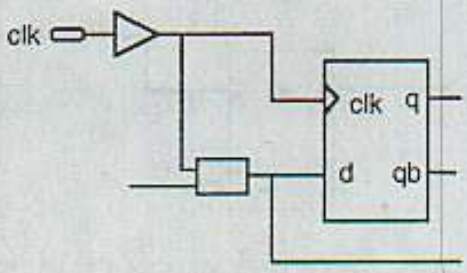


شکل ۱۶-۱۲ غیرقابل کنترل بودن reset آسنکرون

قاعده ۶: قرار نگرفتن پالس ساعت در ورودی داده

کلاک نباید چه مستقیماً و چه از طریق یک منطق ترکیبی به ورودی dی فلیپ فلاپ‌های داخلی وصل شود.

مثالی از اتصال کلاک به ورودی داده در شکل ۱۷-۱۲ نشان داده شده است.



شکل ۱۷-۱۲ پالس ساعت در ورودی داده

۱-۳-۱۲ چگونگی نوشتن کد VHDL با قابلیت تست شدن

در زیر خواهیم دید که چگونه می‌توانیم کد VHDL را بدون شکستن قواعد ATPG مذکور در فوق بنویسیم. همچنین نکاتی در مورد اینکه کد VHDL را چگونه نباید بنویسیم آورده شده است.

قاعده ۱: فلیپ فلاپ‌ها معادل فلیپ فلاپ مرکب داشته باشند.

داشتن یا نداشتن معادل برای فلیپ فلاپ‌ها ارتباطی با چگونگی نوشتن کد VHDL ندارد. عامل تعیین‌کننده همان فلیپ فلاپی است که در حین بهینه‌سازی و قبل از تولید ATPG توسط ابزار سنتز انتخاب شده است. اگر ابزار سنتز فلیپ فلاپ‌های اسکن را در حین بهینه‌سازی انتخاب کند، ابزار ضمیمه‌ای تست قادر نخواهد بود آنها را با معادل دیگری جایگزین نماید. همچنین وقتی ابزار سنتز

فلیپ فلاپ‌هایی را انتخاب کند که معادل اسکن نداشته باشند نیز همین مسأله بروز می‌کند. راه حل این مسأله این است که به ابزار سنتز اجازه انتخاب فقط فلیپ فلاپ‌هایی را که معادل اسکن دارند داده شود. اگر بعضی فلیپ فلاپ‌های معمولی در روش خاصی از اسکن دارای معادل بوده و در روشهای دیگر فاقد معادل باشند، لازم است روش اسکنی را که ابزار سنتز باید اختیار کند در همان ابتدای اولین مرحله بهینه‌سازی معلوم کنیم. این کار در synopsys با دستور زیر انجام می‌شود :

```
set_scan_style LSSD
```

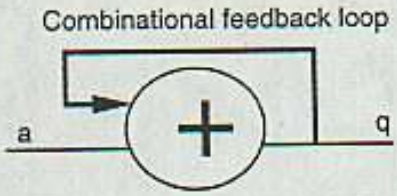
دستور فوق باید قبل از اولین مرحله بهینه سازی اجرا شود.

قاعده ۲ : وجود نداشتن حلقه فیدبک ترکیبی در مدار

معمولاً ATPG با طرحهای آسنکرون سازگاری ندارد. این بدان معناست که حلقه‌های فیدبک ترکیبی نباید در طرح وجود داشته باشند. فرض کنید بخواهیم خط زیر را سنتز کنیم :

```
q <= q + a;
```

از آنجایی که سیگنال q در هر دو طرف علامت مقداردهی (<=) وجود دارد، بنابراین یک حلقه فیدبک تشکیل خواهد شد. نتیجه سنتز این خط در شکل ۱۸-۱۲ نشان داده شده است.

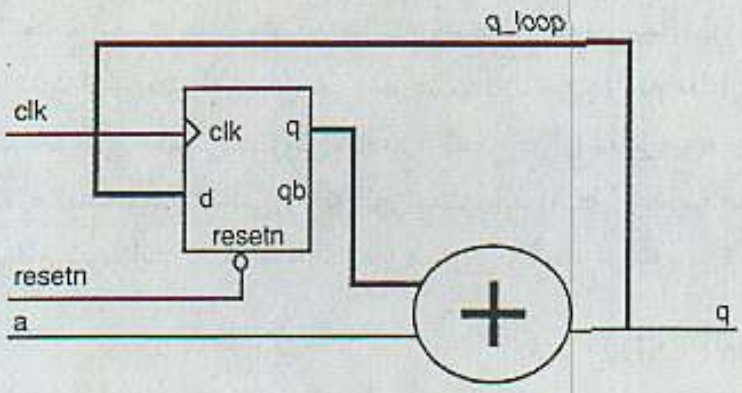


شکل ۱۸-۱۲ حلقه فیدبک ترکیبی

راه حل، شکستن حلقه فیدبک ترکیبی با کمک یک فلیپ فلاپ است. بنابراین کد VHDL را می‌توان به صورت زیر نوشت :

```
process(clk,resetn)
begin
   if resetn= '0' then
        q_loop<=(others=> '0');
   elsif clk'event and clk= '1' then
        q_loop<=q;
   end if;
end process;
q<=q_loop + a;
```

نتیجه سنتز در شکل ۱۹-۱۲ نشان داده شده است.



شکل ۱۹-۱۲ نقص ATPG ندارد.

قاعده ۳ : وجود درایورهای سه حالته

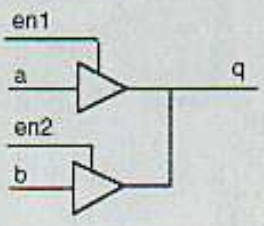
خطا زمانی رخ می‌دهد که کد VHDL به صورت زیر نوشته شود :

```
q <= a;
q <= b when en2 = '1' else 'Z';
```

سیگنال q توسط دو دستور VHDL درایور می‌شود که تنها آخرین مورد به صورت سه حالته عمل می‌نماید. این عمل قواعد ATPG را نقض می‌کند. بنابراین کد VHDL را باید به فرم زیر بنویسیم:

```
q <= a when en1 = '1' else 'Z';
q <= b when en2 = '1' else 'Z';
```

نتیجه این روش توصیف در شکل ۲۰-۱۲ نشان داده شده است.



شکل ۲۰-۱۲ نقض قواعد ATPG مشاهده نمی‌شود.

قاعده ۴ : قابل کنترل بودن پالس‌های ساعت

معمولاً پالس‌های ساعت در ترکیب منطقی با سیگنال‌های دیگر به صورت شکل ۱۵-۱۲ به‌کار

برده می‌شوند، زیرا لازم می‌شود که فقط وقتی en = '1' است فلیپ فلاپ مقدار جدید بگیرد. کد VHDL برای این‌گونه مدارات به شکل زیر است :

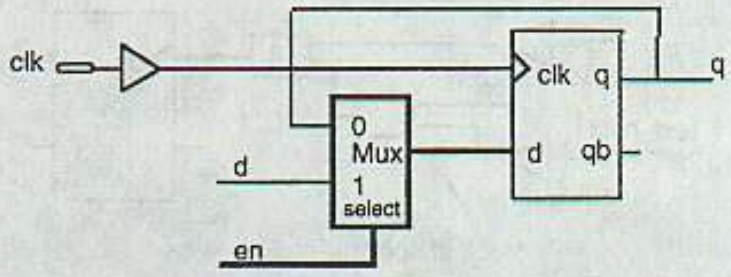
```
clk_g<=en and clk;
process(clk_g)
begin
   if clk_g'event and clk_g= '1' then
        q<=a;
   end if;
end process;
```

ترکیب منطقی کلاک به این نحو مغایر قواعد ATPG است. بنابراین یکی از دو راه حل زیر را می‌توان انتخاب کرد :

راه حل اول : کد VHDL می‌تواند به شکل زیر نوشته شود :

```
process(clk)
begin
   if clk'event and clk = '1' then
        if en= '1' then
                q<=a;
        end if;
   end if;
end process;
```

نتیجه سنتز این کد VHDL در شکل ۲۱-۱۲ نشان داده شده است :



شکل ۲۱-۱۲ نقض قواعد ATPG وجود ندارد.

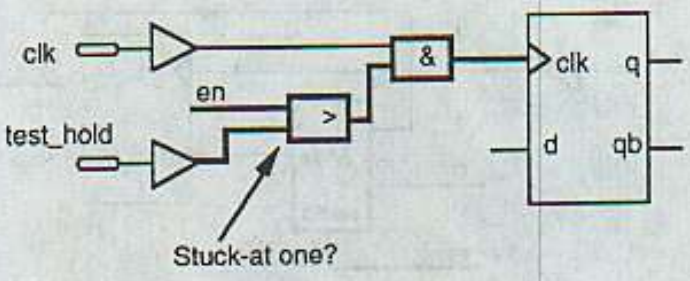
شکل ۲۱-۱۲ نشان می‌دهد که سیگنال خروجی فلیپ فلاپ از طریق یک مالتی پلکسر حلقه فیدبک خود را درست کرده است و چون مالتی پلکسر با سیگنال en کنترل می‌شود، اگر en = '1' باشد، فلیپ فلاپ با لبه کلاک مقدار سیگنال a را به خود می‌گیرد. از نظر عملکردی این مدار مانند

مثال قبل عمل می‌کند، با این تفاوت که روش توصیف آن موجب نقض قواعد ATPG نمی‌شود. این روش را باید ترجیح داد زیرا که می‌توان آن را در سراسر طرح به جای دخالت مستقیم کلاک در منطق ترکیبی به‌کار برد.

راه حل دوم : اگر لازم باشد پالس ساعت از یک بلوک منطقی عبور کند، باید امکان کنترل آن از طریق کلاک ورودی مدار وجود داشته باشد. اگر سیگنال en به عنوان یک ورودی مدار باشد مشکلی ایجاد نمی‌شود زیرا در این مورد سیگنال کلاک قابل کنترل خواهد بود. اگر این سیگنال یک سیگنال داخلی باشد، باید سیگنالی ویژه تست، مثل test_hold معرفی شود. این سیگنال تست باید اطمینان دهد که پالس ساعت عبوری از یک بلوک منطقی، سیگنال کلاک اصلی را بدون توجه به مقدار سیگنال en در تست ATPG دنبال می‌کند. برای این منظور کد VHDL باید به فرم زیر نوشته شود :

```
clk_g<=(en or test_hold) and clk;
process(clk_g)
begin
   if clk_g'event and clk_g= '1' then
        q<=a;
   end if;
end process;
```

سیگنال test_hold باید یک ورودی مدار باشد و در خلال تست ATPG باید در سطح منطقی '1' و در حالت عادی در سطح منطقی '0' قرار داده شود. نتیجه سنتز کد VHDL فوق در شکل ۲۲-۱۲ نشان داده شده است.



شکل ۲۲-۱۲ سیگنال test_hold

یکی از معایب روش فوق این است که مدار نیاز به یک ورودی اضافی پیدا می‌کند. نقص دیگر کشف‌نشدن احتمال یک خطای «گیر کردن در یک» در یکی از پایه‌های گیت OR (ورودی test_hold) توسط بردارهای تست ATPG است. علت آن نگه داشتن سیگنال test_hold در سطح '1' در سرتاسر

تست ATPG است که مانع از تست کردن این ورودی می‌شود. بعضی ابزارهای ATPG استفاده از سیگنال scan_enable را به جای test_hold در مثال بالا قبول می‌کنند. این بدان معناست که مدار نیاز به ورودی اضافی ندارد.

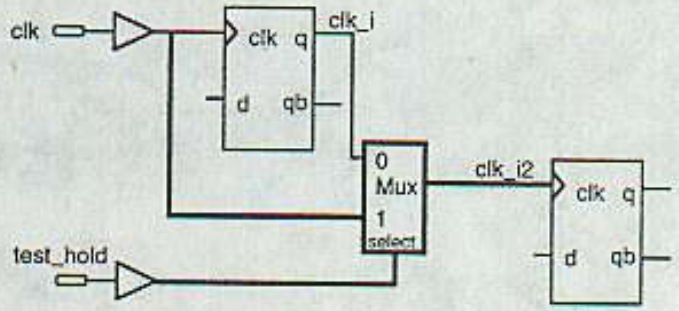
یکی از دلایل متداول برای استفاده از کلاک گیتی (قرار دادن بلوک منطقی در مسیر پالس ساعت) صرفه‌جویی در مصرف انرژی است. در این حالت می‌توان کلاک را از طریق غیرفعال ساختن سیگنال en برای مدت زمان موردنظر خاموش نگه داشت.

راه حل فوق کاملاً خالی از عیب و نقص نیست. از آنجایی که پالس ساعت از یک بلوک منطقی عبور می‌کند مسائل ناشی از شیب پالس ساعت می‌تواند هم در وضعیت عملکردی و هم در مد تست ATPG بروز نماید. بنابراین مسائل ناشی از شیب کلاک باید بعد از طرح‌ریزی مورد توجه ویژه قرار گیرد.

تولید پالس ساعت برای یک فلیپ فلاپ داخلی در داخل مدار نیز با قواعد ATPG مغایرت دارد (به شکل ۱۴-۱۲ رجوع کنید). حل این مشکل یا با نوشتن کد VHDL و به‌کارگیری کلاک خارجی و یا یک مالتی پلکسر امکان‌پذیر است. مالتی پلکسر باید به گونه‌ای کنترل شود که کلاک داخلی در مد عملکردی و کلاک خارجی در مد ATPG فعال گردند. کد VHDL را برای تولید مالتی پلکسر می‌توان به صورت زیر نوشت :

```
Clk_i2<=clk_i when test_hold= '0' else clk;
process(clk_i2)
begin
  if clk_i2'event and clk_i2= '1' then
      q<=d;
  end if;
end process;
```

نتیجه سنتز در شکل ۲۳-۱۲ نشان داده شده است :



شکل ۲۳-۱۲ سیگنال test_hold

قاعده ۵ : قابل کنترل بودن reset آسنکرون

ساده‌ترین راه حل این مشکل استفاده از یک reset آسنکرون واحد برای کلیه فلیپ فلاپ‌ها است. این کار از طریق اضافه کردن همان سیگنال مشترک موجود در لیست حساسیت پروسس‌های پالسی مدار انجام‌پذیر است، به طور مثال :

```
process (clk, resetn)
```

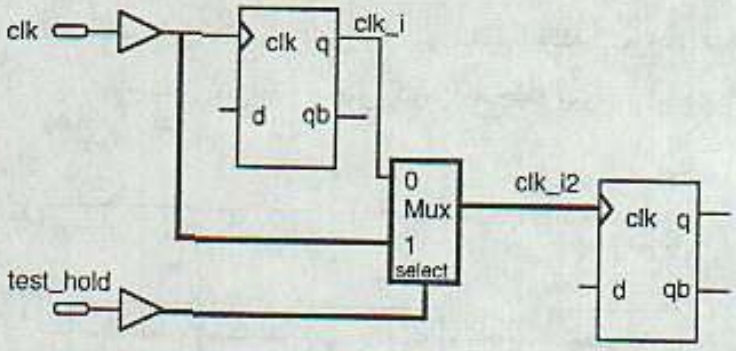
سیگنال resetn باید یک ورودی مدار باشد. چنان‌چه بخواهیم reset آسنکرون توسط یک سیگنال داخلی کنترل شود، این سیگنال باید در مد ATPG توسط یک سیگنال ورودی به مدار قابل کنترل باشد. این کار مستلزم معرفی سیگنال تست یعنی سیگنال test_hold است. اگر سیگنال test_hold به خاطر اینکه کلاک گیتی شده است معرفی گردیده (به توضیح بخش قبل درباره «کلاک‌های قابل کنترل» مراجعه کنید)، همان را می‌توان مورد استفاده قرار داد. در زیر ابتدا کد VHDL برای مواقعی که قواعد ATPG نقض شده نشان داده شده است.

```
process(clk,resetn_i)          -- resetn_i is an internal signal
begin
  if resetn_i = '1' then
     . . .
end process;
```

به منظور اجتناب از شکستن قواعد ATPG، کد VHDL زیر توصیه می‌شود :

```
-- resetn external input
resetn_i2<=resetn_i when test_hold= '0' else resetn;
process(clk,resetn_i2)
begin
  if resetn_i2= '0' then
     . . .
end process;
```

نتیجه سنتز در شکل ۲۴-۱۲ نشان داده شده است.



شکل ۱۲-۲۴ سیگنال test_hold

هم اکنون سیگنال داخلی reset آسنکرون توسط test_hold کنترل می‌گردد. در حین تست ATPG سیگنال test_hold در سطح منطقی '1' نگه داشته می‌شود، این بدان معناست که reset داخلی می‌تواند مستقیماً توسط سیگنال ورودی resetn کنترل شود. در مد عملکردی (test_hold = '0')، سیگنال reset داخلی عیناً مانند قبل برابر سیگنال (resetn_i) خواهد بود :

سیگنال reset	test_hold
resetn-i	'0'
resetn	'1'

قاعده ۶: قرار نگرفتن پالس ساعت در ورودی داده

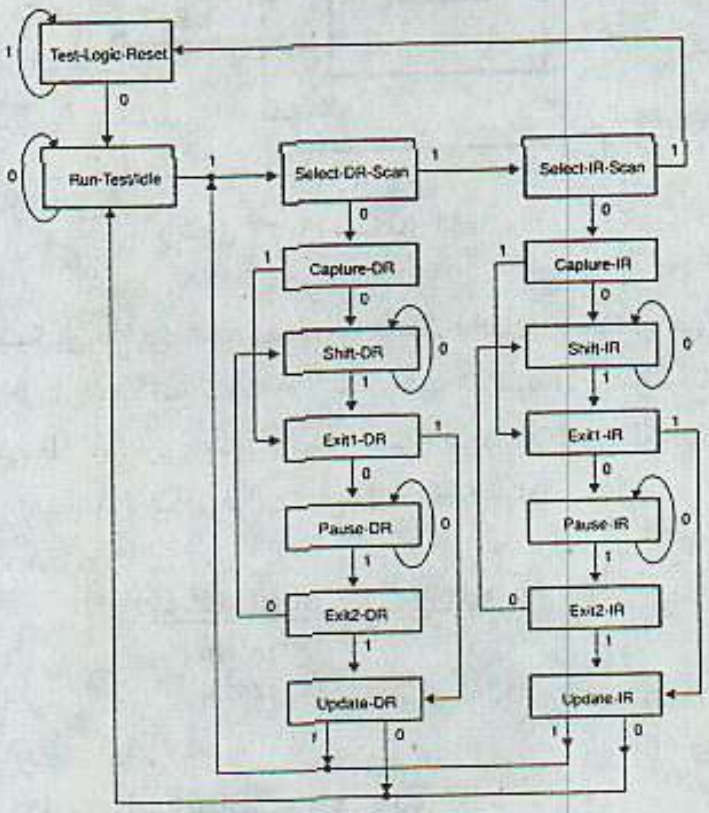
کد VHDL باید به گونه‌ای نوشته شود که پالس ساعت به عنوان ورودی داده به‌کار برده نشود و این امر کاملاً بستگی به آن دارد که چه نوع عملکردی مورد درخواست باشد.

به‌طور خلاصه می‌توان گفت که تولید بردارهای تست ATPG حتی با رعایت نکردن بعضی از قوانین آن نیز امکان‌پذیر است، اما در مواردی که قواعد ATPG رعایت نشده باشد سطح پوشش خطا به شدت کاهش می‌یابد و بعضاً به صفر می‌رسد.

۱۲-٤ اسکن مرزی

اسکن مرزی (زیرمجموعه‌ای از استاندارد JTAG IEEE-1149) معمولاً آزمایشی است برای اطمینان از اینکه مدار به طرز صحیح بر روی بورد قرار گرفته باشد. این روش مبتنی بر اتصال کلیه سلولهای I/O (داخل مدار) به یک شیفت رجیستر طویل است (به شکل ۱۲-۲۵ مراجعه کنید). در این تست خروجی‌ها می‌توانند از طریق انتقال یک الگوی شناخته شده به داخل شیفت رجیستر، یکی از

مقادیر '1' یا '0' را اختیار کنند. این مقادیر را می‌توان در همان لحظه از ورودی‌ها نمونه‌برداری کرد. با خارج کردن نتیجه از شیفت رجیستر می‌توان حاصل کار را ملاحظه نمود.



شکل ۲۵-۱۲ اسکن مرزی

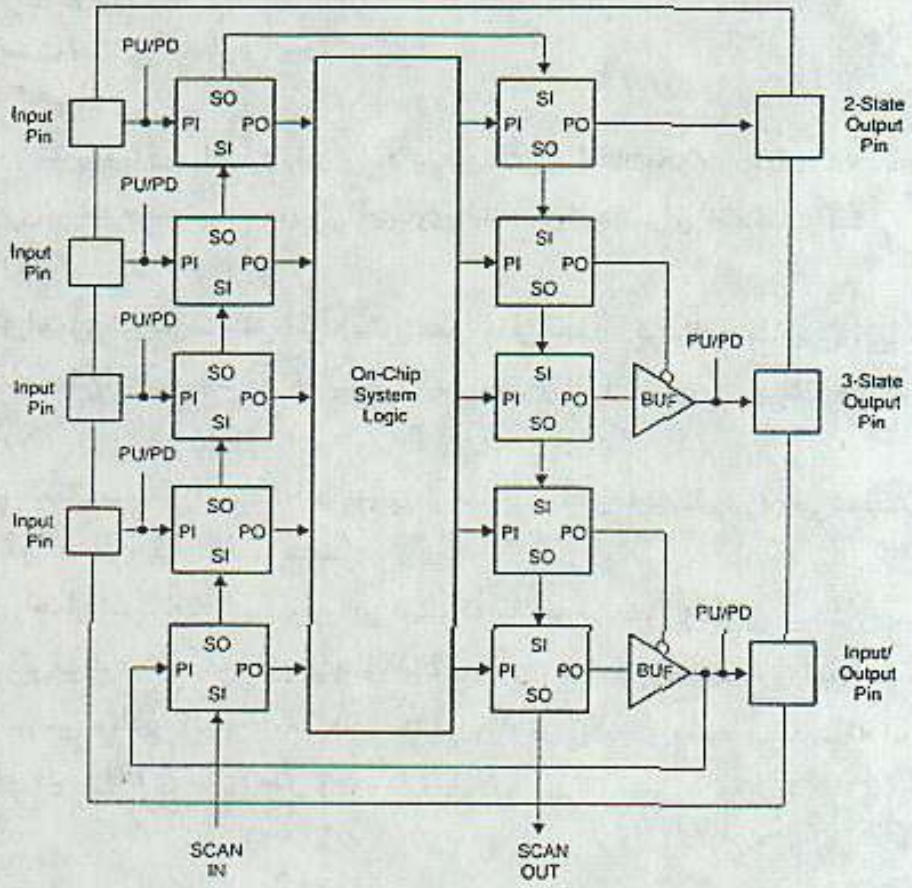
به منظور کنترل شیفت رجیستر در مدار، باید یک ماشین حالت و یک **رجیستر *دستورالعمل***[1] در کنار شیفت رجیستر به مدار اضافه شوند. ماشین حالت به نام کنترلر TAP معروف است (شکل ۲۶-۱۲).

در اواخر دهه ۱۹۸۰ و اوایل دهه ۱۹۹۰، طراحان مجبور بودند که برای طراحی اسکن مرزی به‌طور دستی زمانی بین یک تا دو هفته وقت صرف کنند. اما در حال حاضر ابزارهایی وجود دارد که این روند را به طور اتوماتیک انجام می‌دهند. برای این منظور دو روش عمل وجود دارد:

- ابزار سنتز بتواند به طور خودکار اسکن مرزی را انجام دهد.
- ابزار تست بتواند به طور خودکار کد VHDL قابل سنتز برای اسکن مرزی را تولید کند.

1- Instruction Register

در بعضی مدارات، خصوصاً در FPGA ها و PLD ها، اسکن مرزی بر روی سیلیکن اضافه می‌گردد.



شکل ۲۶-۱۲ کنترلر TAP

با توجه به استاندارد IEEE چهار پایه I/O اجباری و یک پایه اختیاری به مدار اضافه می‌شود.

- Test Data In (TDI)
- Test Data Out (TDO)
- Test Mode Select (TMS)
- Test Clock (TCK)
- Test Reset (TRST) (اختیاری)

مدار علاوه بر شیفت رجیسترها و کنترلر TAP باید حاوی یک رجیستر دستورالعمل برای انجام دادن اسکن مرزی باشد. محتویات این رجیستر تعیین می‌کند که چه نوع تستی باید برای اسکن مرزی به اجرا درآید. طبق استاندارد IEEE سه نوع دستورالعملی که همیشه باید به کار گرفته شود عبارتند از:

- ***بای‌پس*** [1]
- ***نمونه‌برداری*** [2]
- ***تست اضافی*** [3]

چند دستورالعمل غیراجباری نیز وجود دارد : Intest، Runbist، IDcode و Usercode. در زیر شرح مختصری درباره دستور العملهای اجباری ارائه می‌شود. برای اطلاعات بیشتر، استاندارد IEEE_1149.1 را ببینید.

دستورالعمل بای‌پس ـ طبق این دستورالعمل مدار فقط به فرستادن مقادیری می‌پردازد که از طریق خروجی اسکن مرزی به داخل هدایت شده‌اند، یعنی از TDI با عبور از رجیستر بای‌پس به TDO منتقل می‌شوند.

دستورالعمل نمونه‌برداری ـ در این روش از مقادیر ورودی‌ها نمونه‌برداری می‌شود. این نمونه‌ها برای چک شدن به خارج فرستاده می‌شوند.

دستورالعمل تست اضافی ـ منظور این است که مقادیر مشخص از طریق شیفت رجیستر اسکن مرزی وارد مدار می‌گردند. سپس خروجی‌های I/O مطابق با مقادیر داده شده ***روزآمد*** [4] می‌شوند. این روش امکان بررسی مقادیر کلیه خروجی‌های مدار را فراهم می‌کند و معلوم می‌دارد که آیا بین بورد مدار و پایه‌های فلزی I/O ی مدار اتصال وجود دارد یا خیر.

خلاصه

اگر بخواهیم از اسکن مرزی استفاده کنیم مشکل خاصی در گنجاندن این منطق به داخل مدار وجود ندارد، زیرا ابزارهایی هستند که می‌توانند تمام پروسه را به طور خودکار به جریان بیندازند. روش کار برای وارد کردن اسکن مرزی در مدار بستگی به آن دارد که از چه ابزاری استفاده می‌کنیم و چه نوع مداری را می‌خواهیم طراحی کنیم.

1- Bypass

2- Sampling

3- Extest

4- Update

۱۲-۵ بردارهای تست تکمیلی[1]

قبل از اتمام کار ASIC، باید از بردارهای تست تکمیلی استفاده شود. طراح باید به عنوان آخرین مرحله بردارهای تست زیر را به مدار اعمال کند :

- بردارهای تست ATPG
- بردارهای تست پارامتری
- بردارهای تست IDDQ
- بردارهای تست اسکن مرزی
- بردارهای تست تکمیلی دستی
- بردارهای تست RAM

در زیر شرح مختصری درباره این بردارهای تست ارائه می‌شود. در طراحی ASIC توصیه می‌گردد که طراح در مورد جزئیات بردارهای تست با سازندگان ASIC گفتگو کند.

بردارهای تست ATPG

ابزار ATPG این بردارها را به طور خودکار تولید می‌کند.

بردارهای تست پارامتری

در این تست سلولهای I/Oی مدار چک می‌شوند. بردارهای تست پارامتری سطحهای مختلف تغییر حالت در تمام ورودی‌ها و خروجی‌ها را تنظیم می‌کنند. خروجی‌ها توسط بردار تست به سطوح '0'، '1' و یا به صورت سه حالته (در صورت امکان) تنظیم می‌شوند. ورودی‌ها نیز به منظور تأیید عمل سلولهای I/O به سطوح '1' و '0' درایو می‌شوند. این آزمایش می‌تواند تست DC و AC مدار را نیز دربرگیرد.

بردارهای تست IDDQ

تست IDDQ برای سنجش میزان انرژی مصرفی مدار در حالت غیر فعال (استراحت) است. اگر توان مصرفی در مدار غیرفعال مشاهده شود بدان معناست که یک اتصال کوتاه در مدار وجود دارد. اغلب سازندگان ASIC تنها یک بردار برای این نوع تست پیش‌بینی کرده‌اند.

1- Supplementary Test Vectors

بردارهای تست اسکن مرزی

اکثر ابزارهای ATPG نمی‌توانند بردارهای تست اسکن مرزی را به نحوی که قادر به پوشش کافی خطا در منطق اسکن مرزی باشند، ایجاد کنند. بنابراین تعدادی بردار تست باید به طور دستی برای این آزمایش ساخته شود.

بردارهای تست تکمیلی دستی

معمولاً نیازی به این گونه بردارهای تست پیش نمی‌آید، به ویژه آنکه بردارهای تست ATPG خود پوشش خطای بسیار بالایی دارند. با این حال ممکن است در بعضی موارد استفاده از بردارهای تست دستی به عنوان تکمیل‌کننده تستهای ATPG کار مناسبی باشد. این بردارهای دستی معمولاً از طریق بعضی شبیه‌سازی‌های اجرا شده بر روی مدار ساخته می‌شوند.

بردارهای تست RAM

اگر مدار دارای RAM باشد به بردارهای تست ویژه‌ای نیاز خواهد بود زیرا اکثر ابزارهای ATPG بردارهای تست ویژه RAM را تولید نمی‌کنند.

ساختن تمام این بردارهای تست ممکن است به نظر کار بسیار مشکلی بیاید. اما با توجه به اینکه ابزار ATPG بیش از ۹۰ درصد بردارهای تست را تولید می‌کند، وقت مورد نیاز برای ساخت بقیه بردارهای تست معمولاً یکی دو روز بیشتر نخواهد بود. در عین حال باید در نظر داشت که تعداد سایر بردارهای تست در مقایسه با تعداد بردارهای ATPG بسیار اندک است.

باید به این نکته نیز دقت داشت که تعداد و کیفیت بردارهای تست به علاقه و تمایل طراح و سفارش‌دهنده بستگی دارد. اگر مدار مراحل تست را بدون آنکه خطایی مشاهده شود بگذراند می‌توان آن را به سفارش‌دهنده تحویل داد. به علاوه، سازندگان ASIC تعداد نامحدود بردارهای تست را به دلیل وقت زیادی که می‌برد و در نتیجه بر هزینه تولید می‌افزاید، نمی‌پذیرند.

فصل سیزدهم

نمونه‌سازی اولیه با سرعت بالا

۱-۱۳ مقدمه

خیلی از شرکتها برای آنکه از رقابت عقب نمانند سعی می‌کنند مدت زمان تولید محصولات جدیدشان را کوتاه‌تر نمایند. به این ترتیب مدت زمان لازم برای نمونه‌سازی‌های اولیه باید کاهش یابد.

یکی از روشهای کوتاه کردن مدت زمان تولید، توصیف سیستم به وسیله یک زبان توصیف سخت‌افزاری مانند VHDL و سنتز کردن توصیف به طور اتوماتیک و پیاده‌سازی نتیجه در یک FPGA است. اگر محصول به دست آمده با تیراژ بالا به فروش برسد، می‌توان کد VHDL را به صورت یک *طراحی سلولی*[1] که یک راه حل اقتصادی مناسب است عرضه کرد.

در این فصل نشان می‌دهیم که چگونه یک هسته کوچک *پردازشگر آنی*[2] و یک CPU ساده می‌توانند در چند تراشه FPGA پیاده‌سازی شوند. در طول پروسه تولید محصول، از VHDL و عمل سنتز به منظور جایگزین کردن توصیف در سطح RT به جای توصیف در سطح گیتی برای تکنولوژی انتخاب شده استفاده نمودیم. برای آزمایش و صحت عمل نیز یک برنامه تست را در حافظه اصلی قرار

1- Cell Design

2- Real-Time Kernel

دادیم و سپس CPU آن برنامه را در خلال شبیه‌سازی به اجرا درآورد (طراحی مشترک HW/SW). مدارهای پایانی در همان نوبت اول به درستی عمل نمودند.

۱-۱-۱۳ نمونه‌سازی سریع

کاهش مدت زمان تولید در فاز تولید یکی از مهم‌ترین خواسته‌های طراحان و مدیران پروژه‌هاست. یکی از طرق رسیدن به این هدف ***نمونه‌سازی سریع***[1] نام دارد. امروزه، نمونه‌سازی سریع به معنای استفاده از VHDL به عنوان یک زبان توصیف‌کننده رفتار و دربردارنده برنامه‌های آزمایش و تأیید درستی عمل است. در تکنولوژیهای کاربردی، از تراشه‌های FPGA برای اثبات اعتبار مفاهیم یا تهیه بهتر نمونه‌های اولیه فیزیکی استفاده می‌شود. استفاده از تراشه‌های FPGA به جای ASIC موجب کاهش مدت زمان تولید از چند هفته به چند ساعت می‌گردد. اعمال تغییرات در طراحی نیز ظرف چند ساعت یا چند روز امکان‌پذیر خواهد شد.

۲-۱۳ هسته پردازشگر آنی ـ یک توصیف کوتاه

در بیشتر سیستمهای کنترلی یک هسته پردازشگر آنی وجود دارد که فعالیتهای CPU را سامان می‌دهد. این بدان معناست که هسته پردازشگر آنی رأساً تصمیم می‌گیرد که کدام برنامه باید اجرا شود و به برنامه‌ها اطلاع می‌دهد که آیا فرضاً باید منتظر یک سیگنال خارجی و یا مقدار مشخصی از زمان باشند یا خیر.

مثال زیر حاوی طراحیی است که از یک هسته پردازشگر آنی، تعدادی فایل‌هالی رجیستر و یک CPU تشکیل شده است. این سیستم به نام ***FASTCHART***[2] معروف است. این اولین نمونه‌ای است که به وسیله آن می‌توان درستی عمل این ایده را امتحان کرد. شکل ۱-۱۳ بخشهای مختلف آن را نشان می‌دهد.

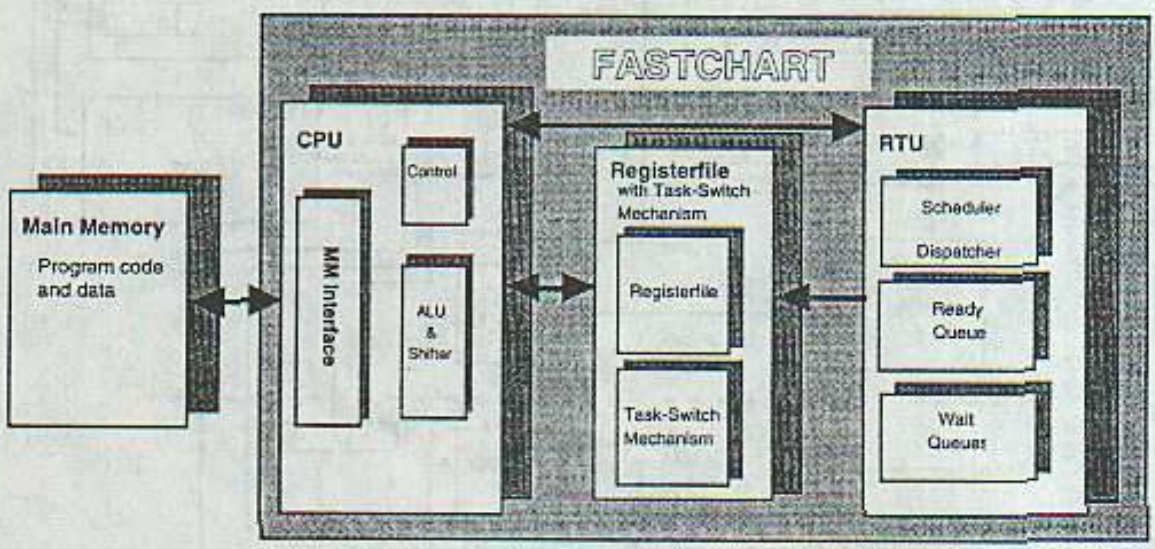
در این نمونه، CPU با هشت بیت کار می‌کند و شامل تعدادی دستورالعمل CPU و فعال‌سازی پردازشگر آنی داده‌هاست. فایل رجیستر نیز حاوی رجیستر CPU، رجیستر آدرس دستورالعملها و غیره برای برنامه‌های موردنظر است.

RTU در بردارنده معمولی‌ترین توابع در یک هسته پردازشگر آنی مثل scheduler و غیره می‌باشد. RTU می‌تواند هشت برنامه با دو ردیف اولویت‌بندی را اداره و کنترل کند. واحد حافظه اصلی

1- Rapid Prototyping

2- a FAST and time deterministic CPU and HArdwar-based Real-Time kernel

متشکل از مدارهای استاندارد EPROM بوده و فقط در شبیه‌ساز (بدون قابلیت سنتز) نمونه‌سازی می‌شود.اگر CPU با هسته پردازشگر آنی ادغام شود، کیفیت عملکرد توابع آنی بسیار بالاتر از زمانی است که هسته پردازشگر در یک کد ماشینی پیاده‌سازی شود. به این ترتیب تغییر برنامه بلافاصله و در مدت زمان صفر انجام‌پذیر می‌شود. این بدان معناست که می‌توانیم برنامه را بین هر دستورالعمل بدون آنکه از کیفیت عملکرد کاسته شود تغییر دهیم. در نتیجه با فعال‌سازی هسته پردازشگر آنی که می‌تواند در یک سیکل CPU انجام شود، عملکرد سیستم بالاتر می‌رود.



شکل ۱-۱۳ بلوک دیاگرام FASTCHART

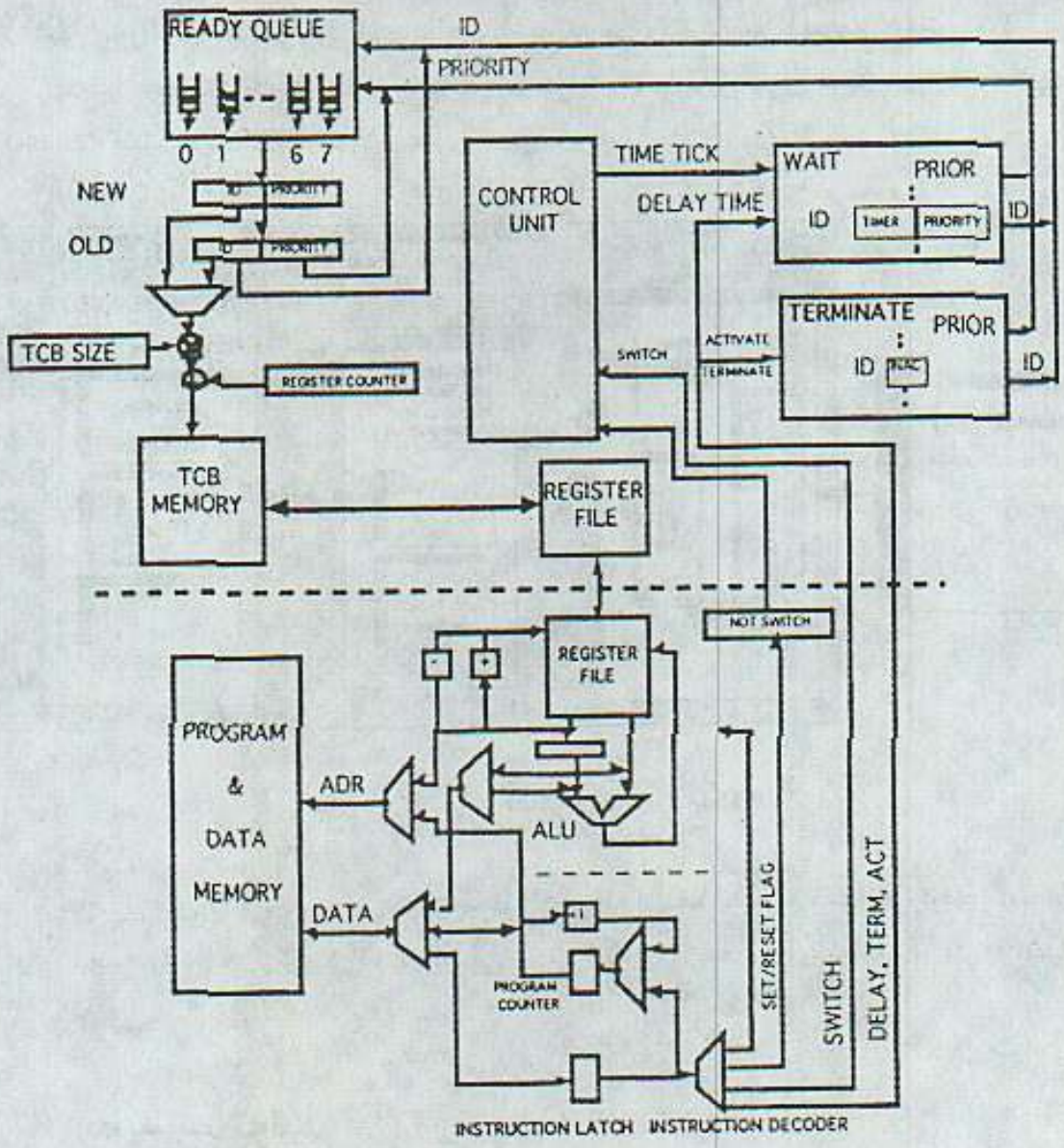
در مثال فوق تمام بخشها، به غیر از حافظه اصلی که از یک کتابخانه شبیه‌ساز گرفته شد، با کد VHDL توصیف شده‌اند. شکل ۲-۱۳ پیچیدگی طرح را نشان می‌دهد. برای مطالعه بیشتر به منابعی که در پایان این فصل ذکر شده مراجعه نمایید.

۳-۱۳ سیستم تولید

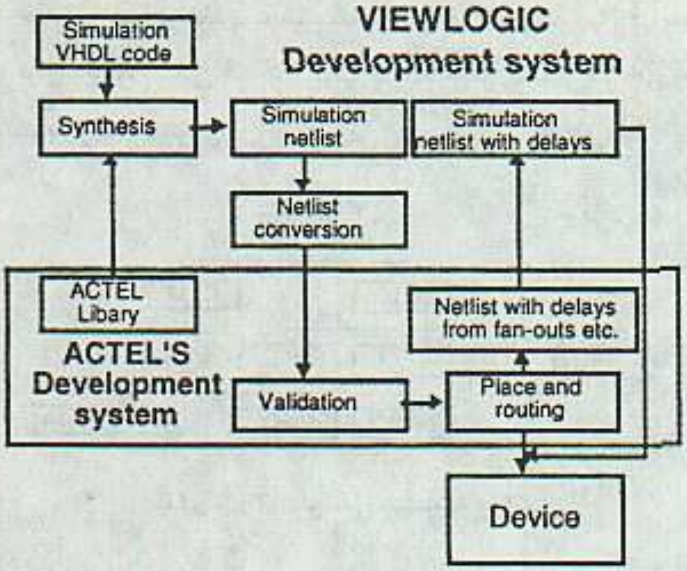
در اوایل دهه ۱۹۹۰ تعداد ابزارهایی که می‌توانستند طرحی را در داخل FPGA ها پیاده‌سازی کنند کم بود. برای انجام طراحی، سیستم مبنی بر PC را ViewLogic و تکنولوژی مورد استفاده را ACTEL برگزیدیم.

شکل ۳-۱۳ چگونگی ارتباط و اتصال دو سیستم مختلف تولید یعنی ViewLogic و ACTEL را نشان می‌دهد. شبیه‌سازی VHDL و سنتز توسط سیستم ViewLogic و سیم‌بندی (و تأخیر ناشی از

آن)، تعیین شدت جریان‌دهی خروجی و برنامه‌نویسی مدارهای ACTEL را در نرم‌افزار ACTEL انجام دادیم. برای ایجاد ارتباط بین بخشهای نرم‌افزاری از برنامه‌های تبدیل‌کننده استفاده گردید.



شکل ۲-۱۳ پیچیدگی FASTCHART



شکل ۳-۱۳ مکانیسم عمل سیستمهای تولید

۱۳-٤ مراحل یا فازهای تولید

هنگامی که بر روی یک اندیشه آزمایش نشده کار می‌کنیم در لابه‌لای مراحل پروسه تولید بین گامهای ۱ و ۵ به جلو و عقب می‌رویم. بنابراین در اصل باید از روش طراحی بالا به پایین استفاده گردد.

شش گام یا مرحله در پروسه تولید عبارتند از :

گام ۱ـ مشخصات و طراحی سیستم

گام ۲ـ توصیف وظایف سیستمهای فرعی

گام ۳ـ ادغام و شبیه‌سازی اجزای تشکیل‌دهنده

گام ۴ـ نگاشت تکنولوژی

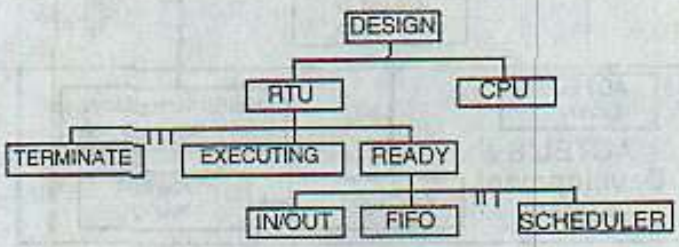
گام ۵ـ بهینه‌سازی، سیم‌بندی و آنالیز دقیق زمانی

گام ۶ـ طراحی و تأثیر اعتبار نمونه اولیه

در گام ـ ۱ ویژگی‌های عملکرد نمونه اولیه مشخص گردید. چند برنامه تست نیز برای تعیین آنچه که نمونه اولیه باید بعد از خاتمه کار انجام دهد نوشته شد.

بعد از تعیین مشخصات، سیستم را به سیستمهای فرعی (اجزای ترکیبی) تقسیم کردیم. این مرحله طراحی معماری نامیده می‌شود. این تقسیم‌بندی به راحتی انجام شد زیرا ما کل سیستم را با زبان C شبیه‌سازی کرده بودیم و تقسیم‌بندی درست به صورت یک «سیستم نرم‌افزاری» درآمد. به هنگام تقسیم‌بندی، ارتباط بین سیستمهای فرعی را نیز تعریف کردیم. وقتی سیستم به چند زیربخش

تقسیم می‌شود آنچه حاصل خواهد شد یک سیستم با طراحی سلسله مراتبی است (شکل ۴-۱۳). این سلسله‌بندی را فقط برای چیره شدن بر پیچیدگی‌ها انجام می‌دهیم. در این طراحی، چهار سطح با حدود سی جزء ترکیبی به وجود آمد.



شکل ۴-۱۳ طراحی سلسله مراتبی

در گام ـ ۲ کد VHDL را برای ***اجزای برگی***[1] در سطح RT نوشتیم. اجزای برگی آنهایی هستند که از اجزای دیگری تشکیل نشده‌اند. برای مثال، FIFO در شکل ۴-۱۳ یک جزء برگی است. در حالی که RTU جزئی است که از چندین جزء دیگر تشکیل شده است. تا آنجایی که امکان دارد باید ساده‌ترین کد را نوشت و برای سنجش سنتزپذیری آن را تست کرد. در چندین مورد برنامه سنتز نتوانست از عهده تبدیل کد برآید و از این رو مجبور شدیم کد را برای انطباق با نرم‌افزار سنتز بازنویسی کنیم. بعد از چند روز اطلاعات کافی در مورد آنچه که قابل سنتز شدن بود به دست آوریم.

مثال زیر مثال کاملی نیست و بر اساس قابلیتهای ابزار سنتز در سال ۱۹۹۱ (ViewLogic) دستکاری‌هایی در آن انجام دادیم:

```
process
begin
   wait until clk = '1';
   if push_task_id = '1' then
        temp := addum(last_p,one);
        last_p(2 downto 0) <= temp(2 downto 0);
   elsif pop_task_id = '1' then
        temp := addum(next_p,one);
   end if;
next_p(2 downto 0) <= temp(2 downto 0);
end process;
```

1- Leaf Component

```
process(last_pointer, next_pointer)
begin
   if last_pointer = next_poiner then
        empty <= '1';
   else
        empty <= '0';
   end if;
end process;
```

در گام ـ ۳ ادغام و شبیه‌سازی اجزا و تست، بررسی و تأیید صحت عمل هر جزء انجام می‌شود. مرحله تست و تأیید را با اعمال محرکهای ورودی و چک کردن سیگنال‌های خروجی هر جزء انجام دادیم. زمانی که تمام اجزا به طور جداگانه مورد آزمایش و بررسی قرار گرفتند، آنها را به هم متصل و اجزای جدید را به وجود آوردیم (عمل ادغام) و سپس این اجزای جدید را نیز تست کردیم. این پروسه را همچنان ادامه دادیم تا به سطح اصلی رسیدیم. به این ترتیب مشاهده می‌شود که عملیات تست و تأیید اساساً «از پایین به بالا» و طراحی «از بالا به پایین» انجام می‌شود. تست بالاترین سطح عملیات را با استفاده از برنامه‌های آزمایش مأخوذ از فاز اول (مشخصات سیستم) انجام دادیم. برنامه آزمایش و تست را در داخل حافظه اصلی (RAM/ROM) قرار دادیم و بعد از آن کل سیستم را با این برنامه‌ها به عنوان محرکهای ورودی، شبیه‌سازی کردیم. برنامه تست توسط CPU به ترتیب زیر اجرا گردید :

```
-- Description of the test program (assembler) --
-------------------------------------
begin INIT_TASK
   ACTIVATE(TASK_1,PR_0);
   STARTADDRESS(20H);
   ACTIVATE(TASK_2,PR_1);
   STARTADDRESS(40H);
   ACTIVATE(TASK_3, PR_0);
   STARTADDRESS(60H);
   TERMINATE;
end INIT_TASK;
-------------------------
begin TASK_1
start: DELAY (16 cpu_clocks);
   JMP start;
end TASK_1;
-------------------------
```

```
begin TASK_2
start: DELAY (16 cpu_clocks);
   JMP start;
end TASK_2;
-----------------------------
begin TASK_3
start: DELAY (16 cpu_clocks);
   JMP start;
end TASK_3;
-----------------------------
```

برنامه تست فوق از چهار برنامه جداگانه تشکیل شده است. برنامه INIT_TASK بعد از reset شدن CPU آغاز به کار می‌کند و سپس برنامه‌های TASK_1 تا TASK_3 به وسیله INIT_TASK فعال می‌شوند. آنگاه کار INIT_TASK خاتمه می‌یابد. TASK_1 ، TASK_2 و TASK_3 هسته پردازشگر آنی یا RTU را فعال ساخته و درخواست یک تأخیر ۱۶ سیکل ساعتی را می‌کنند.

با اجرای برنامه تست فوق ثابت شد که عملیات و وظایف تعیین شده در RTU و CPU به خوبی کار می‌کنند.

سه گام نخست غیروابسته به نوع تکنولوژی هستند. تست و تأیید تمام مراحل با کد VHDL انجام گرفت. اگر بخواهیم از لحاظ مکان (حجم) بهترین نتیجه به دست آید توصیه می‌کنیم که تکنولوژیهای مختلف FPGA را در خلال سنتز، مورد آزمایش قرار دهیم. این بدان علت است که به‌طور مثال در تکنولوژی‌شرکت Xilinx فلیپ‌فلاپ‌های فراوانی به‌کار گرفته‌شده در حالی‌که در مدار ACTEL از تعداد بسیار کمی فلیپ فلاپ استفاده شده است.

در گام ـ ۴ تکنولوژی مورد استفاده باید انتخاب شود. برنامه سنتز برای آنکه بتواند مطابق با تکنولوژی انتخاب شده (ACTEL 1A020) کار ترجمه و بهینه‌سازی را انجام دهد نیاز به یک فایل تکنولوژی دارد. در حال حاضر مدل جدیدی مرکب از چندین گیت تولید شده است.

هر گیت یک تأخیر ۱۰ نانوثانیه‌ای دارد. در اینجا باید یک شبیه‌سازی جدید و اولین نوبت آنالیز زمانی صورت پذیرد.

در گام ـ ۵ پکیج و اندازه FPGA ها باید انتخاب شود. ما ACTEL 1020 و ACTEL 1010 را انتخاب کردیم. netlist به دست آمده از سنتز را در پکیج انتخابی پیاده‌سازی نموده و سپس فایل حاوی تمام تأخیرهای حاصل از گیت‌ها و اتصالات داخلی را از ابزار دریافت کرده و بعد از آن آنالیز دقیق زمانی را اجرا کردیم. می‌توانستیم به جای آن در طراحی FASTCHART از تست و تأیید به وسیله

شبیه‌ساز استفاده کنیم. یک راهکار بهتر دیگر آن است که تمام شروط زمانی را برای ابزار سنتز مشخص کنیم تا به این ترتیب این ابزار بتواند مطابق با آنها بهینه‌سازی را انجام دهد. همچنین این امکان وجود دارد که با کمک تحلیل‌گر زمانی داخلی ابزار سنتز آنالیز زمانی قبل و بعد از سیم‌بندی را انجام دهیم. برای این کار ابزار سنتز هم باید یک تحلیل‌گر زمانی پیشرفته داشته باشد و هم قادر باشد از لحاظ زمانی مطابق با تکنولوژی انتخاب شده بهینه‌سازی را انجام دهد.

در گام ـ ۶ در پروسه تولید کمترین زمان را مصرف کرد. شش تراشه FPGA را برنامه‌ریزی کرده و یک بورد مدار را سیم‌بندی نمودیم. آنگاه برنامه‌های تست را در EPROM قرار دادیم و بالاخره برنامه را به اجرا درآوردیم. نمونه اولیه به دست آمده در همان آزمایش اول به نتیجه رسید.

در زیر جدول زمانی تقریبی برای فعالیتهای فوق ارائه شده است :

گامهای ۱ تا ۳	۵ هفته
گامهای ۴ و ۵	۲ هفته
گام ۶	۱ هفته
مجموع	۸ هفته

بدیهی است که مراحل اولیه پروسه تولید زمان خاص خود را می‌طلبند اما مراحل نهایی سریع‌تر انجام می‌گیرند، زیرا عملیات در این مراحل عمدتاً به طور اتوماتیک صورت می‌پذیرند.

البته نمی‌توان انتظار داشت که نمونه‌سازی سریع، بهترین نمونه را تولید کند، اما هدف از آزمایش ما این بود که بتوانیم یک سیستم را با سرعت بیشتری تست و با کیفیت بهتری طراحی کنیم. وقتی نمونه اولیه ساخته شد برنامه‌های جدیدی در آن پیاده گردید و با سرعتی به مراتب بیشتر از بلوک‌های کامپیوتری مدل‌کننده همین رفتار به اجرا درآمد (حداقل هزار بار سریع‌تر).

۵-۱۳ مراجع مطالعاتی بیشتر

The following references describe FASTCHART:

Lindh, L. and F. Stanischewski (1991) 'FASTCHART - a fast time deterministic CPU and hardware based real-time-kernel', *IEEE Real-Time Workshop, Paris,* 12-14 June.

Lindh, L. and F. Stanischewski (1991) 'A design of a real-time unit in hardware', *Svenska Nationella Arbetsgruppen i Realtid - SNART, Uppsala*, 19-20 August.

Lindh, L. and F. Stanischewski (1991) 'FASTCHART - idea and implementation', *IEEE International Conference on Computer Design (ICCD), Cambridge, Mass.*, 14-16 October.

More information on rapid prototyping can be found in the following article:

Lindh, L., K.D. Müller-Glaser and H. Rauch (1993) 'Rapid prototyping with VHDL and FPGAs', *Lecture Notes in Computer Science* no. 705, Springer-Verlag.

فصل چهاردهم

خطاهای متداول در VHDL و چگونگی اجتناب از آنها

شاید متداول‌ترین خطاهایی که توسط تازه‌کاران و مبتدیان در نوشتن کد VHDL به وجود می‌آید استفاده از دستورات متوالی مثل if و case در بخش موازی VHDL باشد. در فصول مربوط به VHDL متوالی و VHDL موازی بخشهای مختلف یک کد VHDL و دستورات مجاز در هر بخش توضیح داده شده است کلیه دستورات VHDL در پیوست (الف) آورده شده و همچنین مشخص شده است که از چه دستوراتی را می‌توان در بخشهای متوالی یا موازی VHDL استفاده کرد.

۱-۱۴ سیگنال‌ها و متغیرها

یک خطای متداول در هنگام شروع طراحی با VHDL عدم تشخیص تفاوت بین سیگنال‌ها و متغیرها است. همان طور که در جدولهای ۱-۴ و ۲-۴ نشان داده شده است، رفتار سیگنال‌ها و متغیرها به طور کامل از هم متفاوت است. به علاوه، یک سیگنال هیچ وقت نمی‌تواند منطبق با یک خط سیگنال سخت‌افزاری باشد. متغیرها تنها به منظور ذخیره موقت داده‌ها در یک پروسس یا زیر ـ برنامه به‌کار می‌روند و از این رو مقداردهی یک متغیر باید با یک دستور متوالی انجام گیرد. بنابراین یک متغیر

فقط می‌تواند در بخش متوالی VHDL معرفی شود، این در حالی است که مقداردهی به یک سیگنال می‌تواند هم متوالی و هم موازی باشد. لیکن سیگنال‌ها تنها می‌توانند در بخش موازی VHDL معرفی و اعلام شوند. همان طور که در مثال زیر نشان داده شده است، متغیرها بسیار موردنیاز هستند زیرا مقدارشان را بدون هیچ گونه تأخیری دریافت می‌کنند، در حالی که سیگنال مقدار خود را پس از تأخیر یک دلتا اختیار می‌کند. این بدان معناست که کد VHDL بسته به اینکه سیگنال‌ها و یا متغیرها به‌کار رفته باشند باید به گونه‌های مختلفی نوشته شوند. راه حل توصیه شده برای استفاده از سیگنال یا متغیر به طور کامل وابسته به طرح است. مثال زیر این تشخیص را روشن می‌سازد.

فرض کنید بخواهیم تابع زیر را در یک پروسس با کد VHDL توصیف کنیم :

```
int <= a and b and c;
q <= int or d;
```

این مقداردهی‌های موازی و هم‌زمان سیگنال‌ها را می‌توان مستقیماً در معماری برنامه نوشت، ولی اصولی در این کار وجود دارد که مثال زیر برای توضیح آن برگزیده شده است. برای روشن شدن مطلب مثال کوچکی را انتخاب کرده‌ایم.

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex1 is
port (a,b,c,d:    in    std_logic;
      q:          out   std_logic);
end;
```

معماری برنامه با برگزیدن پارامتر داخلی int به عنوان یک سیگنال :

```
Architecture sig of ex1 is
signal int:std_logic;
begin
   process(a,b,c,d,int)
   begin
        int<= a and b and c;
        q<= int and d;
   end process;
end;
```

معماری برنامه با برگزیدن پارامتر داخلی int به عنوان یک متغیر :

```
Architecture var of ex1 is
begin
   process(a,b,c,d)
   variable int:std_logic;
   begin
        int:= a and b and c;
        q<= int and d;
   end process;
end;
```

تفاوت بین این دو مثال را می‌توان در لیست حساسیت پروسس‌ها در کنار دیگر تفاوتها پیدا کرد. در مثال اول که int به عنوان یک سیگنال تعریف شده، این سیگنال در لیست حساسیت پروسس وارد شده است. در حالی که در مثال دوم که int به عنوان یک متغیر به‌کار گرفته شده است فقط سیگنال‌های حقیقی ورودی (d و c و b و a) در لیست حساسیت قرار دارند. سیگنال int باید در مثال اول گنجانده می‌شد زیرا فقط بعد از تأخیر یک دلتا مقدار جدیدش را می‌گیرد. زمانی که خط q ⇐ int and d; به اجرا درآمد، سیگنال int هنوز مقدار قبلی (ناصحیح) خود را دارد. برای گذر از این مشکل، int باید در لیست حساسیت پروسس قرار گیرد، زیرا که پروسس پس از تغییر int (یک دلتا پس از مقداردهی سیگنال int به مقدار جدید) دوباره فعال می‌شود. عیب برگزیدن int به عنوان یک سیگنال در این مثال به خوبی آشکار است. نه تنها به خاطر سپردن مقداردهی به تمام سیگنال‌هایی که نقش واسط را در لیست حساسیت دارند کار مشکلی است، بلکه مثال سیگنال به دلیل آنکه پروسس باید دوبار اجرا شود (در مقایسه با فقط یک بار اجرا شدن پروسس در مثال متغیر) به وقت طولانی‌تری برای شبیه‌سازی نیاز دارد. برتری مثال اول این است که می‌توان از سیگنال int در شبیه‌سازی و برای ایجاد شکل موج استفاده کرد. اما اگر int را به عنوان یک متغیر اعلام کنیم استفاده فوق غیرممکن می‌شود زیرا هیچ زمانی به آن تعلق نمی‌گیرد. این موضوع ***عیب‌زدایی***[1] از مثال متغیر را از مثال سیگنال سخت‌تر می‌کند.

نکته : توصیه می‌شود تنها زمانی از متغیر استفاده گردد که بخواهیم مقداری را به طور موقت ذخیره کنیم.

---

1. Debug

۲-۱۴ سنتز منطقی و لیستهای حساسیت پروسس‌ها

بعضی از ابزارهای سنتز هیچ گونه توجهی به لیست حساسیت نمی‌کنند. این امر منجر به یک عدم هماهنگی بین شبیه‌سازی در سطح RT و سطح گیتی می‌شود. به طور مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex2 is
port (clk1, clk2, resetn, din,en: in  std_logic;
      q:                          out std_logic);
end;

Architecture bad of ex2 is
begin
  process(clk1, resetn)
  begin
       if resetn= '0' then
         q<= '0';
       elsif ____________ and clk2= '1' then
         if en= '1' then
              q<=din;
         end if;
       end if;
  end process;
end;
```

نتیجه سنتز مثال فوق یک فلیپ فلاپ از نوع D با یک clk enable خواهد بود که به کلاک آن توسط clk2 پالس داده می‌شود و این به معنای آن است که برنامه سنتز، لیست حساسیت و clk1 را نادیده گرفته است. این یک روش توصیفی نادرست است. معمولاً برنامه سنتز یک اخطار نیز در مورد اینکه clk2 در لیست حساسیت قرار نگرفته صادر می‌کند. اگر این مثال بخشی از یک جزء ترکیبی بزرگ‌تر را تشکیل دهد این اخطار به راحتی گم خواهد شد. خطا باید قبل از سنتز یعنی هنگام شبیه‌سازی در سطح RT کشف شود. بیش از این نمی‌توانیم تأکید کنیم که وقتی می‌خواهیم خطاهای عملکردی طرح را پیدا کنیم عمل شبیه‌سازی در سطح RT چقدر اهمیت پیدا می‌کند (به فصل ۱۱ مراجعه کنید، «آشنایی با روشهای طراحی»).

۳-۱۴ بافرها و سیگنال‌های واسط داخلی

چنان‌چه بخواهیم مقدار یک سیگنال خروجی را بازخوانی کنیم، سه راه کار وجود دارد :

- تعریف سیگنال در مد بافر در بخش اعلام برنامه
- به‌کارگیری یک سیگنال واسط داخلی در معماری برنامه
- استفاده از نشان سیگنال 'driving_value' (در استاندارد VHDL-93)

مثالهایی از سه مورد بالا در زیر آورده شده است :

مثال (بافر) :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex3 is
port (clk1, resetn, din1: in      std_logic;
      q1:                 buffer std_logic;
      q2:                 out    std_logic);
end;

Architecture buf of ex3a is
begin
  process(clk, resetn)
  begin
       if resetn= '0' then
              q1<= '0';
              q2<= '0';
       elsif clk'event and clk= '1' then
              q1<= din1;
              q2<= q1;       -- Reading signal q1
       end if;
end process;
end;
```

مثال (سیگنال واسط داخلی) :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex3b is
port (clk, resetn, din1:  in    std_logic;
```

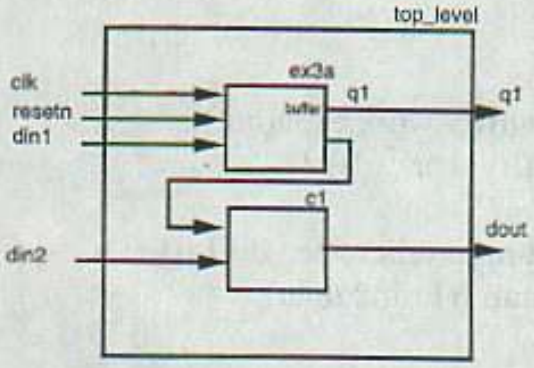
```
    q1:              out  std_logic;
    q2:              out  std_logic);
end;

Architecture buf of ex3b is
signal q1_b: std_logic;        -- The "dummy signal"
begin
q1<=q1_b;
  process (clk, resetn)
  begin
      if resetn= '0' then
            q1_b<= '0';
            q2<= '0';
      elsif clk'event and clk= '1' then
            q1_b<= din1;
            q2<= q1_b;
      end if;
  end process;
end;
```

مثال (VHDL-93 و 'driving_value') :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity ex3c is
port (clk, resetn, din1:  in   std_logic;
      q1:                 out  std_logic;
      q2:                 out  std_logic);
end;

Architecture buf of ex3c is
begin
  process (clk, resetn)
  begin
      if resetn= '0' then
            q1<= '0';
            q2<= '0';
      elsif clk'event and clk= '1' then
            q1<= din1;
```

```
            q2<= q1'driving_value;
        end if;
    end process;
end;
```

راه کار سوم تنها در استاندارد VHDL-93 معتبر است. چون ابزارهای سنتز هنوز این نشان را حمایت نمی‌کنند، بنابراین می‌توان از آن برای مدلهای شبیه‌سازی استفاده کرد. از نقطه نظر سنتز منطقی، نتیجه سنتز بدون توجه به راه حل اول یا دوم یکسان خواهد بود. لیکن اگر جزء ترکیبی در VHDL ساختاری، در راه کار بافر، فراخوانی شود، در حین شبیه‌سازی مشکلاتی به وجود خواهد آمد زیر طراح غالباً خود را در موقعیتی می‌یابد که باید سعی کند دستور portmap را بین یک سیگنال در مد out و یک سیگنال دیگر در مد buffer اجرا کند و این امر خلاف نسخه استاندارد VHDL است.



شکل ۱-۱۴ مثال بافر

فرض کنید که المان توصیف شده در ex3a که دارای سیگنال خروجی q1 (تعریف شده در مدل بافر) است توسط المان دیگری در سطح بالاتر فراخوانی شود. همچنین فرض کنید سیگنال q1 مستقیماً به خروجی المان سطح بالاتر نگاشت داده شود. اگر سیگنال q1 در المان سطح بالاتر در مد خروجی تعریف گردد، به هنگام کمپایل کردن برنامه خطا ایجاد می‌شود زیرا شما می‌خواهید یک سیگنال از مد بافر را به یک سیگنال از مد خروجی نگاشت دهید و این کار مجاز نیست. برای این کار باید سیگنال خروجی q1 در المان سطح بالاتر را نیز از نوع بافر تعریف کنید. این کار عملی نیست زیرا سیگنال خروجی q1 از منظر top_level مطلقاً به عنوان یک سیگنال خروجی و نه چیز دیگر قلمداد می‌شود.

مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
```

```
Entity top_level is
port (clk,resetn,din1, din2:   in  std_logic;
      q1, dout:                out std_logic);
end;

Architecture bad of top_level is
  component ex3a
  port (   clk,resetn,din1:   in       std_logic;
           q1:                buffer   std_logic;
           q2:                out      std_logic);
  end component;
  component c1
  port(...
      ...
  end component;
  signal i1: std_logic;
  For U1: ex3a Use entity work.ex3a(buf);
  For U2: c1 Use entity work.c1(rtl);
  begin
       U1: ex3a port map (clk,resten,din1,q1,i1);   -- Error
       U2: c1 port map (i1,din2,dout);
  end;
```

اما مشکل فوق را می‌توان با کمک سیگنال واسط داخلی بر طبق راه کار دوم حل کرد. به‌کار بردن مد ***ورودی ـ خروجی***[1] درست نیست. این مد را تنها باید برای سیگنال‌های ***دوجهته***[2] و یا ***منطق سیمی***[3] (مثل wired-OR) به‌کار گرفت. شماتیک سیگنال معرفی شده در مد ورودی ـ خروجی قطعاً صحیح خواهد بود اما به هنگام استفاده از دستور port map خطا ایجاد خواهد شد. اصولاً استفاده از مد ورودی ـ خروجی هنگامی که سیگنال دوطرفه نباشد شیوه مناسبی برای طراحی نیست.

اگر از VHDL ساختاری استفاده می‌گردد به‌کارگیری راه کار سیگنال واسط داخلی بسیار توصیه می‌شود. از سوی دیگر چنانچه از VHDL ساختاری استفاده نشود و مراحل طراحی سلسله مراتبی در محیط شماتیک رسم گردد (به فصل ۱۱ مراجعه کنید. «آشنایی با روشهای طراحی»)

---

1. Inout
2. Bidirectional
3. Wired Logic

انتخاب راه حل، فاقد اهمیت است، مگر آنکه بخواهیم در آینده جزء ترکیبی را توسط VHDL ساختاری فراخوانی نماییم.

۱۴-۴ معرفی بردارها با downto یا to

در VHDL دو راه برای معرفی بردارها وجود دارد :

```
signal a : std_logic_vector (0 to 3);
signal a : std_logic_vector (3 downto 0);          --Recommended
```

توصیه می‌شود که بردارها همواره با downto معرفی شوند. این بدان علت است که اگر بردارها با downto معرفی شوند، بیت با ارزش (MSB) همیشه بیتی است که بزرگ‌ترین شماره اندیس را داشته و محل آن در دورترین قسمت در سمت چپ قرار می‌گیرد. اما اگر از to استفاده شود، بیت با شماره اندیس صفر بر بیت با ارزش (MSB) انطباق پیدا می‌کند.

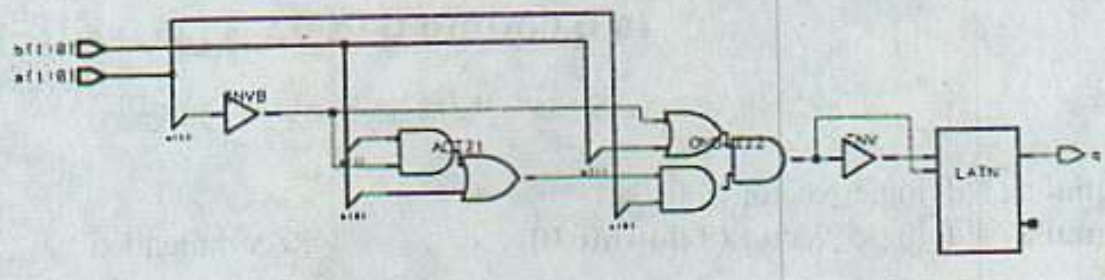
۱۴-۵ پروسس‌های ترکیبی ناقص

هرگاه یک پروسس ترکیبی (و نه یک المان حافظه) موضوع طراحی باشد سیگنال‌های خروجی باید مقداردهی شوند، به این معنا که امکان گذر از یک پروسس بدون حداقل یک بار مقدار دادن به سیگنال‌های خروجی وجود ندارد و حداقل یک بار باید به کلیه خروجی‌ها پروسس ترکیبی ارزشی را نسبت داد.

مثال یک پروسس ناقص می‌تواند چنین باشد :

```
Architecture bad is
begin
  process(a,b)
  begin
      if a>b then
          q<= '0';
      elsif a<b then
          q<= '1';
      end if;
  end process;
end;
```

همان طور که شکل ۲-۱۴ نشان می‌دهد، برای سیگنال‌های خروجی یک لچ به دست آمده که نشانه وجود یک پروسس ترکیبی ناقص است.

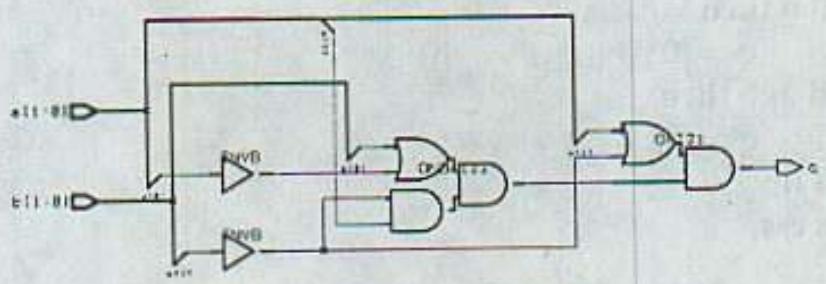


شکل ۲-۱۴ نتیجه سنتز یک پروسس ناقص

روش صحیح توصیف پروسس می‌تواند به فرم زیر باشد :

```
Architecture good is
begin
   process(a,b)
   begin
        if a>b then
                q<= '0';
        else
                q<= '1';
        end if;
   end process;
end;
```

پروسس فوق در هر بار فعال شدن مقداری را به سیگنال خروجی q می‌دهد، بدین معنی که ابزار سنتز نیازی به در نظر گرفتن یک لچ برای خروجی ندارد. نتیجه سنتز در شکل ۳-۱۴ نشان داده شده است.



شکل ۳-۱۴ نتیجه سنتز یک پروسس کامل

فصل پانزدهم

مثالهایی از طراحی و نکات راهنما

مثالهایی که در این فصل آورده شده است تماماً مثالهای کوچکی هستند. یک طرح بزرگ دارای صدها خط از دستورات VHDL در معماری برنامه خواهد بود. البته طراحی‌های بزرگ معمولاً شامل ترکیبی از این مثالها هستند.

راهنمایی‌های مربوط به اینکه جزء ترکیبی چقدر بزرگ باشد بستگی به نوع ابزار سنتز مورد استفاده و مقدار حافظه موجود دارد. به طور نرمال اندازه اجزای طرح باید بین ۵۰۰ تا ۶۰۰۰ گیت (معادل NAND) باشد. اگر کوچک‌تر از ۵۰۰ گیت باشد توصیف اتصالات بین آنها به وقت بسیار طولانی در ارتباط با اندازه اجزای نیاز خواهد داشت و این زمان طولانی ربطی به اینکه VHDL ساختاری مورد استفاده بوده یا ابزارهای گرافیکی نخواهد داشت. چنان‌چه اجزای توصیفی بزرگ‌تر از ۶۰۰۰ گیت باشند، اداره کردن آنها مشکل خواهد بود و سنتز کردن آنها نیز به زمانی طولانی نیاز خواهد داشت.

مثالهای زیر توسط ابزار سنتز Synopsys سنتز شده‌اند. هیچ قاعده خاصی برای Synopsys به‌کار نرفته و همه مثالها استاندارد IEEE را به طور کامل برآورده ساخته‌اند و این بدان معناست که مثالهای طراحی ذکر شده را می‌توان با اکثر ابزارهای سنتز بدون آنکه نیازی به تغییر در آنها باشد تحت سنتز قرار داد. تنها تغییری که ممکن است مجبور به اعمال آن باشیم در مورد مثالهایی است که از بسته ieee.std_logic_unsigned استفاده کرده‌اند. چنان‌چه ابزار سنتز مورد استفاده، این بسته را حمایت نکند می‌توان آن را با بسته‌هایی که حمایت می‌شوند تعویض کرد (به فصل ۵ مراجعه کنید،

«کتابخانه، بسته و زیربرنامه‌ها».

مثالهای ۱-۱۵ تا ۱۱-۱۵ با استفاده از کتابخانه Motorola's HDC سنتز شده‌اند. Motorola یکی از سازندگان پیشرو ASIC و دارنده بهترین بسته‌های طراحی ASIC در بازار است.

۱-۱۵ جمعگرها

۱-۱-۱۵ جمعگر تک بیتی با بیت نقلی[1]

همان طور که در فصل ۶، «VHDL ساختاری» توضیح داده شد، امکان توصیف یک جمعگر تک بیتی با دو گیت AND، دو گیت XOR و دو گیت OR وجود دارد. این روش توصیفی تنها به عنوان یک مثال برای نشان دادن اصول VHDL ساختاری بود و به منظور طراحی یک جمعگر تک بیتی مناسب نیست، زیرا استفاده از VHDL ساختاری هم دست و پاگیر و هم مستلزم صرف وقت زیاد است.

به منظور جمع کردن داده‌هایی از نوع std_logic_vector می‌توان از تابع '+' با آرگمان‌های ورودی و خروجی از نوع std_logic و std_logic_vector استفاده کرد. این تابع در بسته‌های ieee.std_logic_unsigned و ieee.std_logic_signed وجود دارد.

به طور خلاصه تفاوت بسته‌های گفته شده در بالا در جمع با علامت برای بسته signed و جمع بدون علامت برای بسته unsigned است («بسته‌های IEEE» را در پیوست (ب) ببینید). علت استفاده از '0'& در بعضی از مثالهای زیر این است که تابع '+' باید برداری را که طول قابل قبول داشته باشد برگشت دهد. تابع '+' یک بردار به اندازه طویل‌ترین بردار آرگمان ورودی را برمی‌گرداند. برای مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity add1 is
port (a,b,cin:     in  std_logic;
      q1, dout:    out std_logic);
end;

Architecture rtl of add1 is
signal s: std_logic_vector (1 downto 0);
```

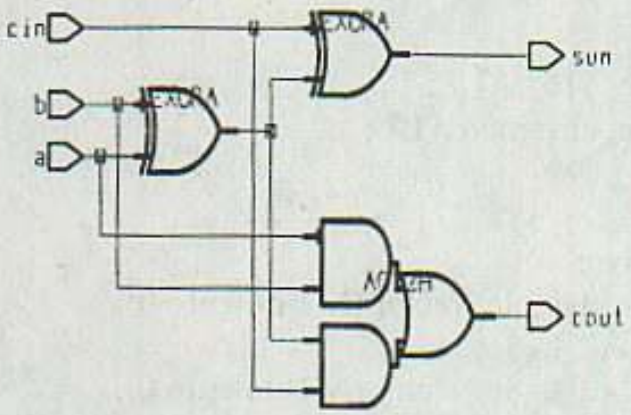
1- Carry

```
begin
        s<=('0' & a)+b+cin;
        sum<=s(0);
        cout<=s(1);
end;
```

نتیجه سنتز در شکل ۱-۱۵ نشان داده شده است.

می‌توان از بسته ieee.std_logic_arith نیز استفاده کرد. در این صورت عمل جمع به فرم زیر خواهد بود :

```
s <= unsigned ('0' & a) + unsigned (b) + cin;
```



شکل ۱-۱۵ نتیجه سنتز، جمعگر تک بیتی

۲-۱-۱۵ جمعگر هشت بیتی با بیت نقلی

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity add8 is
port (a,b:  in   std_logic_vector(7 downto 0);
      cin:  in   std_logic;
      sum:  out  std_logic_vector(7 downto 0);
      cout: out  std_logic);
end;
```

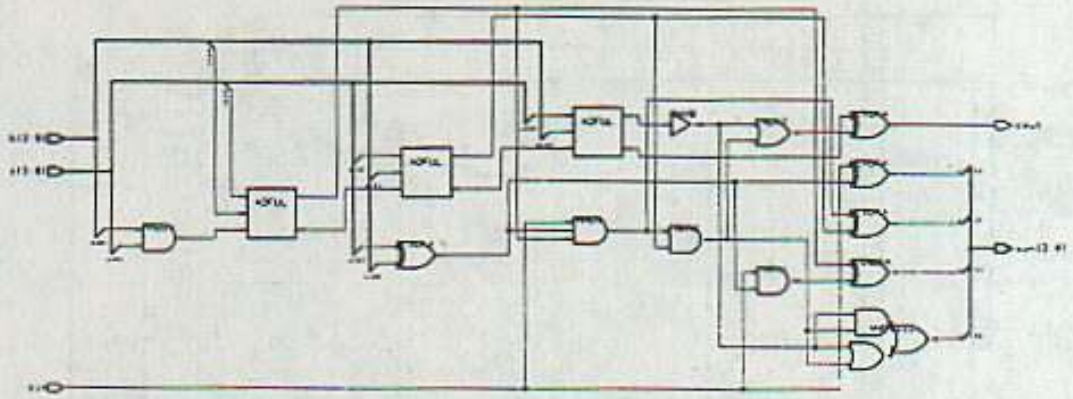
```
Architecture rtl of add8 is
signal s: std_logic_vector(8 downto 0);
begin
        s<=('0' & a)+b+cin;
        sum<=s(7 downto 0);
        cout<=s(8);
end;
```

۳-۱-۱۵ جمعگر عمومی با بیت نقلی

جزء ترکیبی زیر دو بردار با طول N را با هم جمع می‌کند. مقدار N به هنگام فراخوانی این جزء ترکیبی مشخص می‌شود.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity g_dd is
generic (N: positive:=4);
port (a,b:  in   std_logic_vector (N-1 downto 0);
      cin:  in   std_logic;
      sum:  out  std_logic_vector (N-1 downto 0);
      cout: out  std_logic);
end;

Architecture rtl of g_dd is
signal s:std_logic_vector (N-1 downto 0);
begin
        s<=('0' & a)+b+cin;
        sum<=s(N-1 downto 0);
        cout<=s(N);
end;
```

نتیجه سنتز در شکل ۲-۱۵ نشان داده شده است.

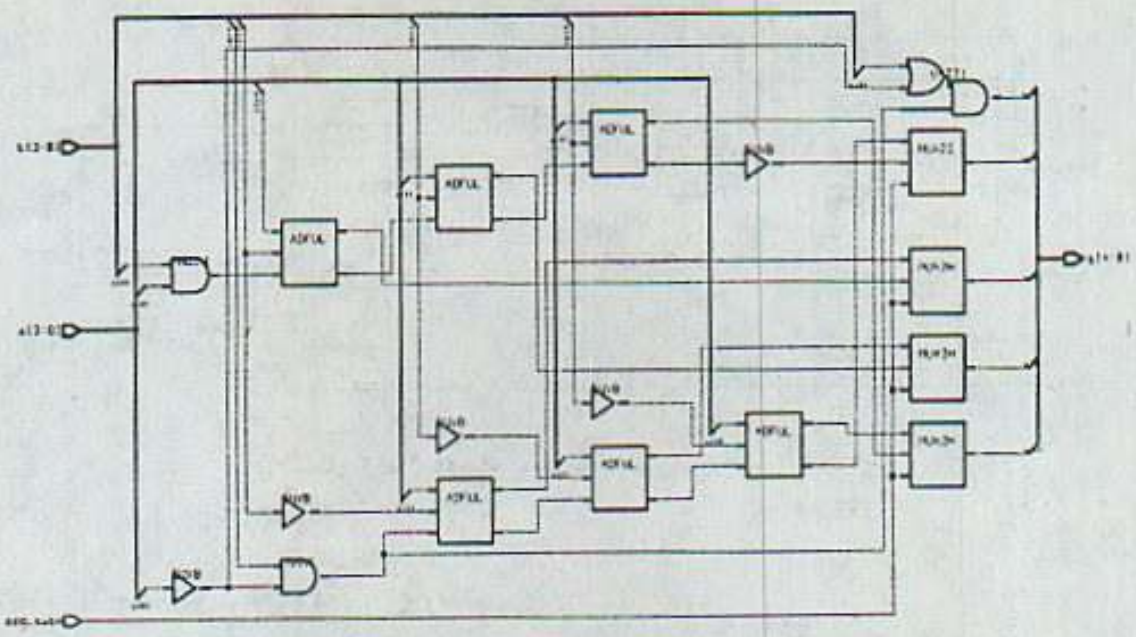


شکل ۲-۱۵ جمعگر عمومی سنتز شده با N = ۴

۴-۱-۱۵ جمعگر ـ تفریقگر چهاربیتی

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity add_sub is
port (add_subn:  in    std_logic;
      a,b:       in    std_logic_vector(3 downto 0);
      q:         out   std_logic_vector(4 downto 0));
end;

Architecture rtl of add_sub is
begin
  process(a,b,add_subn)
  begin
       if add_subn= '1' then
              q<=('0' & a) + b;
       else
              q<=('0' & a)  - b;
       end if;
  end process;
end;
```

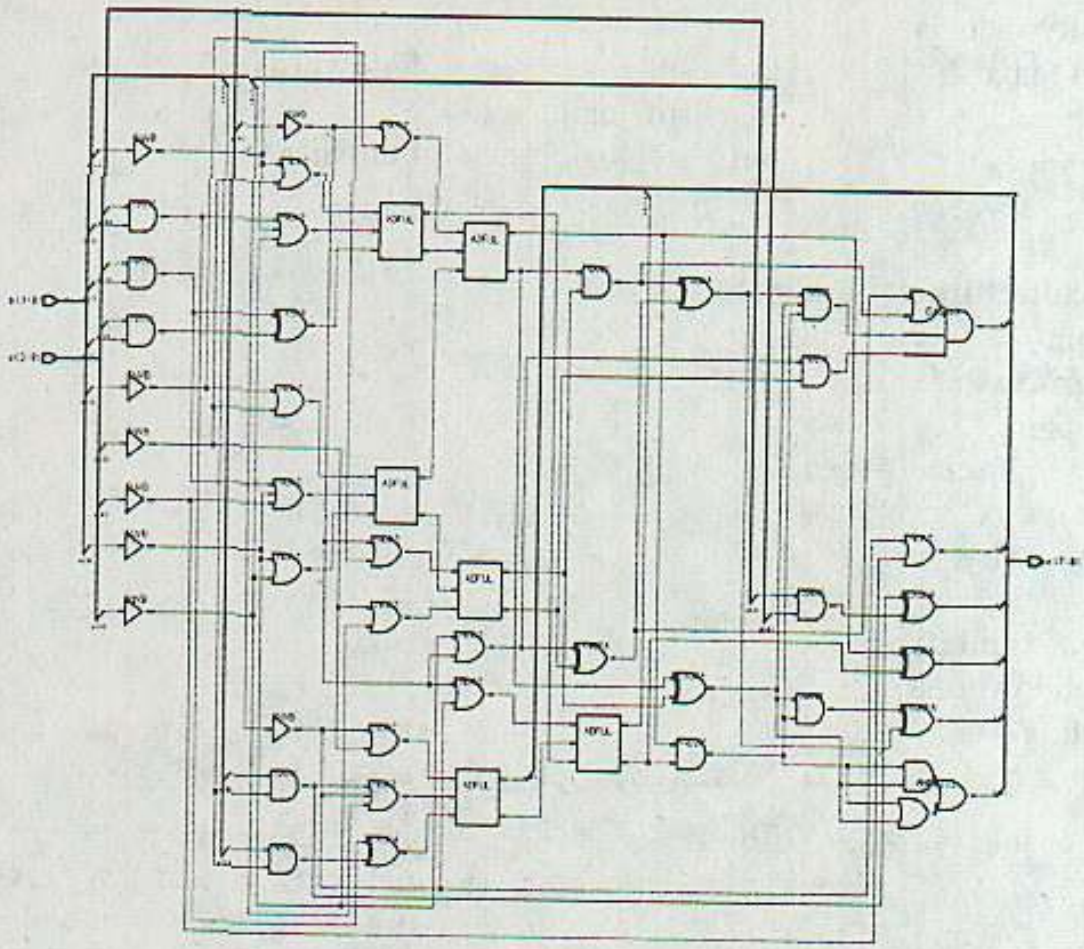
نتیجه سنتز در شکل ۳-۱۵ نشان داده شده است.



شکل ۳-۱۵ جمعگر ـ تفریقگر چهاربیتی

۲-۱۵ ضرب کننده برداری

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity v_mult is
port (a,b:        in    std_logic_vector(3 downto 0);
      c:          out   std_logic_vector(7 downto 0));
end;

Architecture rtl of v_mult is
begin
        c<=a * b;
end;
```



شکل ۴-۱۵ ضرب‌کننده چهاربیتی

۳-۱۵ اشتراک منابع

۱-۳-۱۵ مثالی از حالتی که در آن اشتراک منابع یک جمعگر امکان‌پذیر نمی‌شود

ابزارهای سنتز پیشرفته شامل بخشی به نام بهینه‌سازی سطح بالا هستند. این بدان معناست که ابزار سنتز می‌تواند منابع را مشترک کند. اگر بخواهیم از این پالایش استفاده کنیم باید کد VHDL را به طور صحیح بنویسیم. از شرایط اشتراک منابع این است که بردارها نباید در یک لحظه از جمعگرها استفاده کنند. به طور مثال :

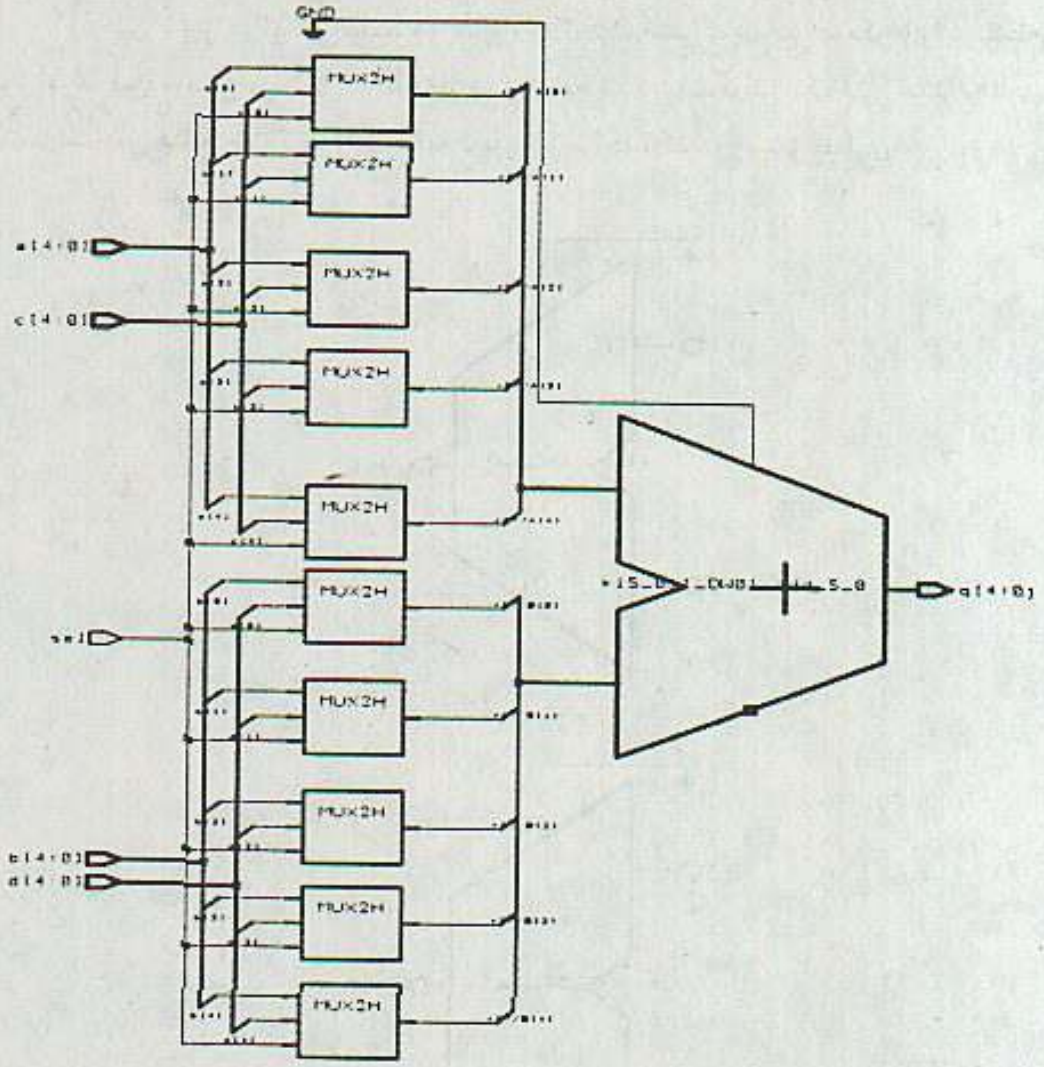
```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
```

```
Entity share is
port (a,b,c,d:          in    std_logic_vector(4 downto 0);
      sel:              in    std_logic;
      q:                out   std_logic_vector(4 downto 0));
end;

Architecture rtl of share is
begin
  process(a,b,c,d,sel)
  begin
      if sel= '0' then
            q<=a+b;
      else
            q<=c+d;
      end if;
  end process;
end;
```

از آنجایی که دو عمل جمع در یک پروسس تعریف شده‌اند و هیچ گاه هم‌زمان با هم اجرا نمی‌شوند (به علت عبارت if-then-else)، ابزار سنتز می‌تواند کد VHDL در مثال بالا را به طریقی بهینه‌سازی کند که هر دو عمل جمع مشترکاً از یک جمعگر استفاده کنند.

۲-۳-۱۵ مثال از حالتی که در آن اشتراک منابع یک جمعگر امکان‌پذیر نمی‌شود

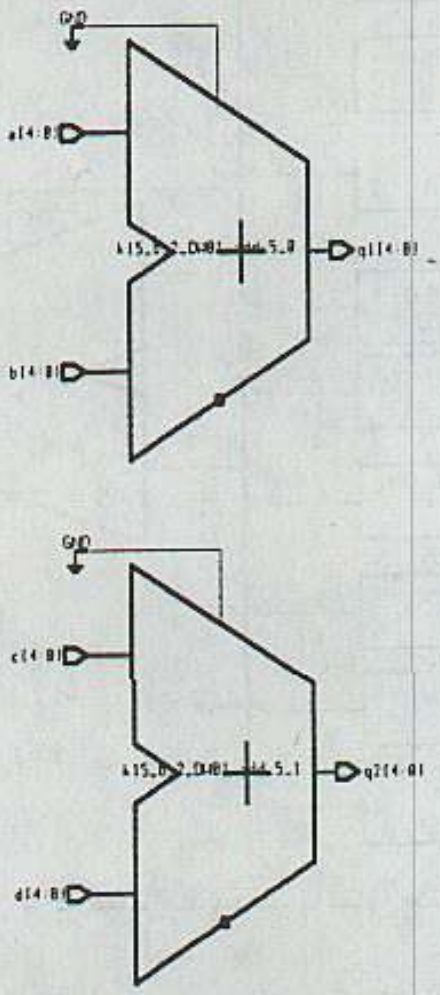
```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity share is
port (a,b,c,d:          in    std_logic_vector(4 downto 0);
      q1,q2:            out   std_logic_vector(4 downto 0));
end;

Architecture rtl of share is
begin
      q1<=a+b;
      q2<=c+d;
end;
```



شکل ۵-۱۵ سنتز یک جمعگر با منابع مشترک

همان طور که از شکل ۵-۱۵ برداشت می‌شود، تنها یک جمعگر مورد استفاده قرار گرفته است. ابزار سنتز سرتاسر کد VHDL را بررسی کرده و به این نتیجه می‌رسد که دو عمل جمع هیچ‌گاه به‌طور هم‌زمان انجام نمی‌شوند. ابزار سنتز مناسب‌ترین گزینه‌ها را بر اساس محدودیتهای زمانی و مکانی که طراح مشخص کرده است انتخاب می‌کند. در بعضی موارد، سنتز ممکن است منتج به دو جمعگر همراه با یک مالتی پلکسر در سمت راست آنها گردد. همان طور که در فصل ۱۰، «سنتز در سطح RT» توضیح داده شد، نتیجه سنتز در ابزارهای سنتز پیشرفته با محدودیتهای زمانی و مکانی که طراح تنظیم کرده است کنترل می‌شود.

ابزارهای سنتز هرگز نمی‌توانند در مثال (۲-۳-۱۵) اشتراک منابع جمعگرها را در سطح RT به وجود آورند. این موضوع در شکل ۶-۱۵ نیز نشان داده شده است. لیکن یک ابزار سنتز رفتاری توانایی مشترک ساختن منابع جمعگرها را دارد (به فصل ۱۷ مراجعه کنید، «سنتز رفتاری»).



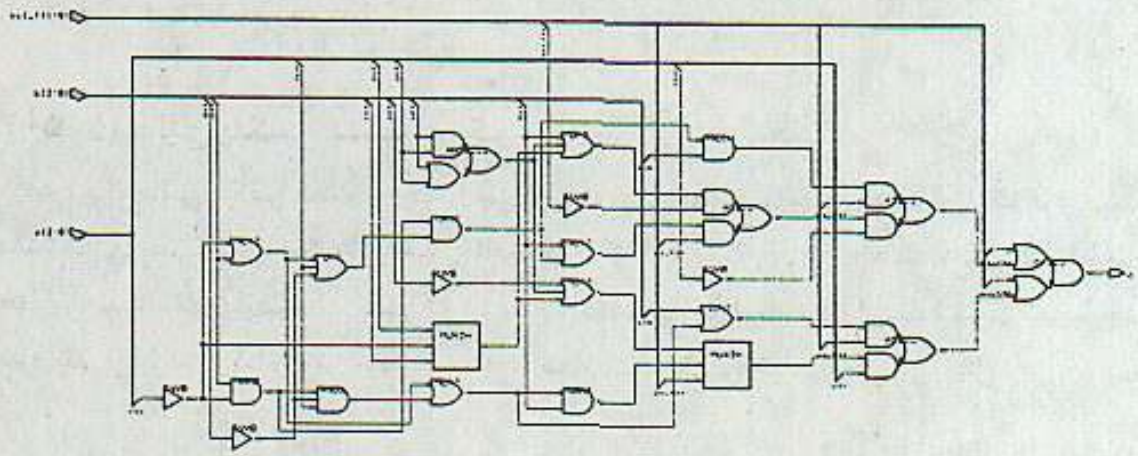
شکل ۶-۱۵ امکان مشترک‌سازی منابع وجود ندارد.

۱۵-٤ مقایسه‌کننده‌ها

از آنجایی که اغلب اوقات مقایسه مقدار بردارها ضروری است، مقایسه‌کننده‌ها به عنوان اجزای ترکیبی مفید مورد نیاز خواهند بود. جزء ترکیبی زیر دو بردار سه بیتی را با هم مقایسه می‌کند و سیگنال خروجی comp را تولید می‌نماید. عملکرد سیگنال خروجی comp توسط سیگنال ورودی sel_f تعیین می‌شود.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity comp is
port (a,b:      in    std_logic_vector(2 downto 0);
      sel_f:    in    std_logic_vector(1 downto 0);
      q:        out   boolean);
end;

Architecture rtl of comp is
begin
  process(sel_f,a,b)
  begin
    case sel_f is
      when "00"      => q <= a=b;
      when "01"      => q <= a<b;
      when "10"      => q <= a>b;
      when others    => q <= false;
    end case;
  end process;
end;
```

نتیجه سنتز در شکل ۷-۱۵ نشان داده شده است.



شکل ۷-۱۵ مقایسه‌کننده

۱۵-۵ مالتی‌پلکسرها و دیکدرها

۱۵-۵-۱ مالتی پلکسر دو به یک

روشهای گوناگونی برای توصیف یک مالتی پلکسر دو به یک در پروسس‌های مختلف وجود دارد، از جمله با استفاده از دستور if-then-else و دستور case و یا با ساختار with-select و یا با کمک VHDL ساختاری. روشی که ما پیشنهاد می‌کنیم استفاده از ساختار when-else است، به دلیل آنکه مالتی پلکسر دو به یک، یک المان بسیار کوچک بوده و به طور نرمال تنها یک خط از چندین خط طرح VHDL را شامل می‌شود. این مالتی پلکسر می‌تواند در یک خط با ساختار when-else توصیف گردیده و فهمیدن و خواندن آن را آسان سازد. برای مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity mux2 is
port (a,b,sel:    in    std_logic;
      q:          out   std_logic);
end;

Architecture rtl of mux2 is
begin
        c<=a  when sel= '0' else b;
end;
```

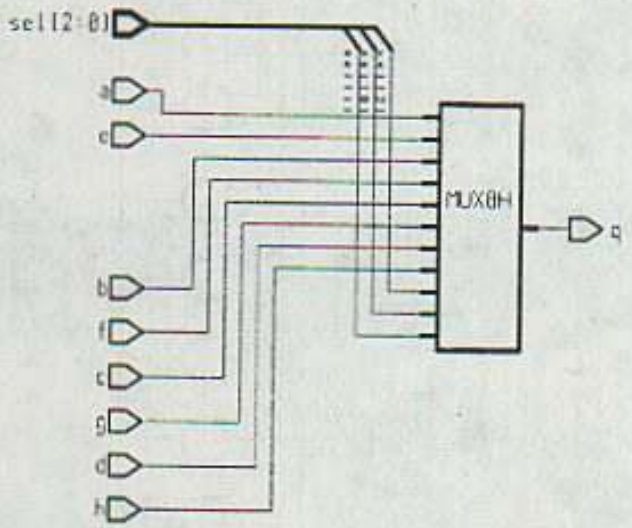
۱۵-۵-۲ مالتی پلکسر هشت به یک

یک مالتی پلکسر هشت به یک را نمی‌توان در یک خط در معماری برنامه توصیف کرد. درست مانند مالتی پلکسر دو به یک، این امکان وجود دارد که از ساختار when-else استفاده کنیم، اما برای خوانایی بهتر برنامه توصیه می‌کنیم که از عبارت case نیز در پروسس‌ها استفاده شود. نتیجه سنتز صرف‌نظر از اینکه از کدام روش توصیف استفاده شده باشد، یکسان خواهد بود. برای مثال :

```
Library ieee;
Use ieee.std_logic_1164.ALL;
```

```
Entity mux8 is
port (a,b,c,d,e,f,g,h:    in    std_logic;
      sel:                in    std_logic_vector(2 downto 0);
      q:                  out   std_logic);
end;

Architecture rtl of mux8 is
begin
  process(a,b,c,d,e,f,g,h,sel)
  begin
      case sel is
        when "000"        => q<=a;
        when "001"        => q<=b;
        when "010"        => q<=c;
        when "011"        => q<=d;
        when "100"        => q<=e;
        when "101"        => q<=f;
        when "110"        => q<=g;
        when others       => q<=h;
      end case;
  end process;
end;
```

نتیجه سنتز در شکل ۸-۱۵ نشان داده شده است.

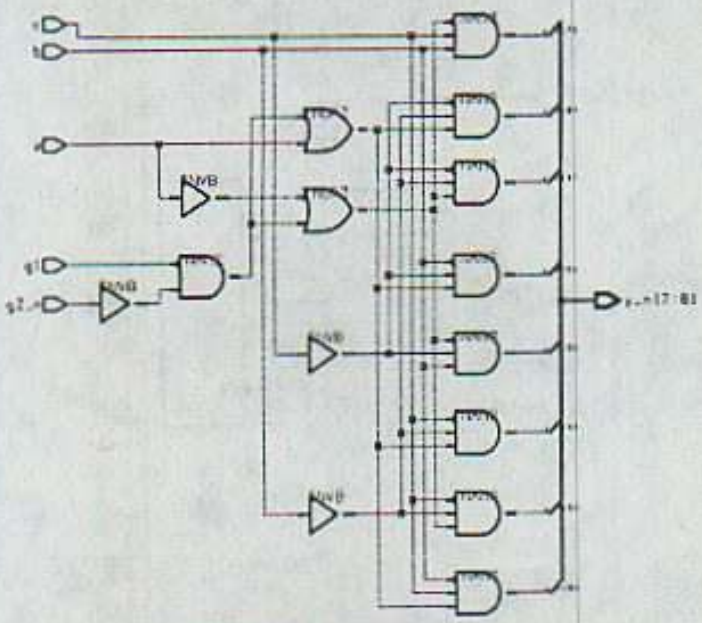


شکل ۸-۱۵ مالتی پلکسر هشت به یک

۳-۵-۱۵ دیکدر سه به هشت

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity decoder is
port (a,b,c,g1,g2_n:    in  std_logic;
      t_n:              out std_logic_vector (7 downto 0));
end;

Architecture rtl of decoder is
begin
  process (a,b,c,g1,g2_n)
  begin
       y_n<=(others=> '1');
       if g1= '1' and g2_n= '0'  then
            y_n(conv_integer (c & b & a))<= '0';
       end if;
  end process;
end;
```

نتیجه سنتز در شکل ۹-۱۵ نشان داده شده است.



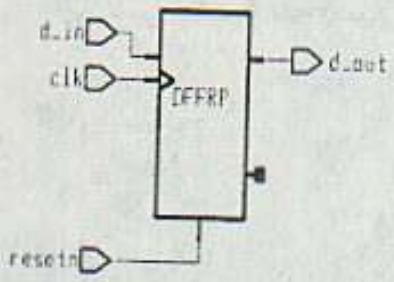
شکل ۹-۱۵ دیکدر سه به هشت

۱۵-۶ رجیستر

۱۵-۶-۱ فلیپ فلاپ با reset آسنکرون

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity d_mem is
port (clk,resetin,d_in:   in  std_logic;
      d_out:              out std_logic);
end;

Architecture rtl of d_mem is
begin
  process (clk,resetn)
  begin
       if resetn= '0' then
              d_out<='0';
       elsif clk'event and clk= '1' then
              d_out<=d_in;
       end if;
  end process;
end;
```

نتیجه سنتز در شکل ۱۵-۱۰ نشان داده شده است.



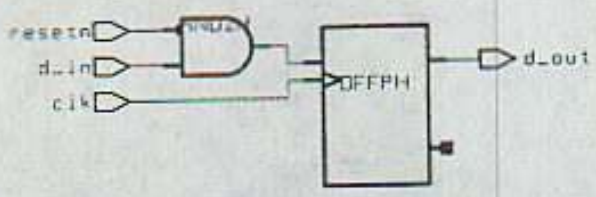
شکل ۱۵-۱۰ فلیپ فلاپ با reset آسنکرون

۱۵-۶-۲ فلیپ فلاپ با reset سنکرون

```
Library ieee;
Use ieee.std_logic_1164.ALL;
```

```
Entity d_mem is
port (clk,resetin,d_in:   in  std_logic;
      d_out:              out std_logic);
end;

Architecture rtl of d_mem is
begin
  process (clk)
  begin
       if clk'event and clk= '1' then
          if resetn= '0' then
               d_out<= '0';
          else
               d_out<=d_in;
          end if;
       end if;
  end process;
end;
```

نتیجه سنتز در شکل ۱۱-۱۵ آورده شده است.



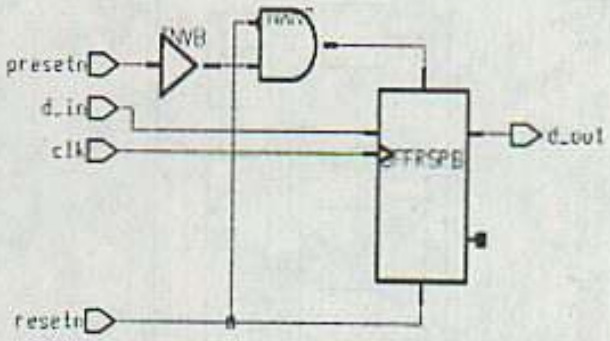
شکل ۱۱-۱۵ فلیپ فلاپ با reset سنکرون

۳-۶-۱۵ فلیپ فلاپ با set و reset آسنکرون

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity d_mem is
port (clk,resetin,presetn,d_in:  in  std_logic;
      d_out:                     out std_logic);
end;

Architecture rtl of d_mem is
begin
```

```
  process (clk,resetn,presetn)
  begin
       if  resen='0' then
              d_out<='0';
       elsif clk'event and clk='1' then
              d_out<=d_in;
       end if;
  end process;
end;
```

نتیجه سنتز در شکل ۱۲-۱۵ نشان داده شده است.



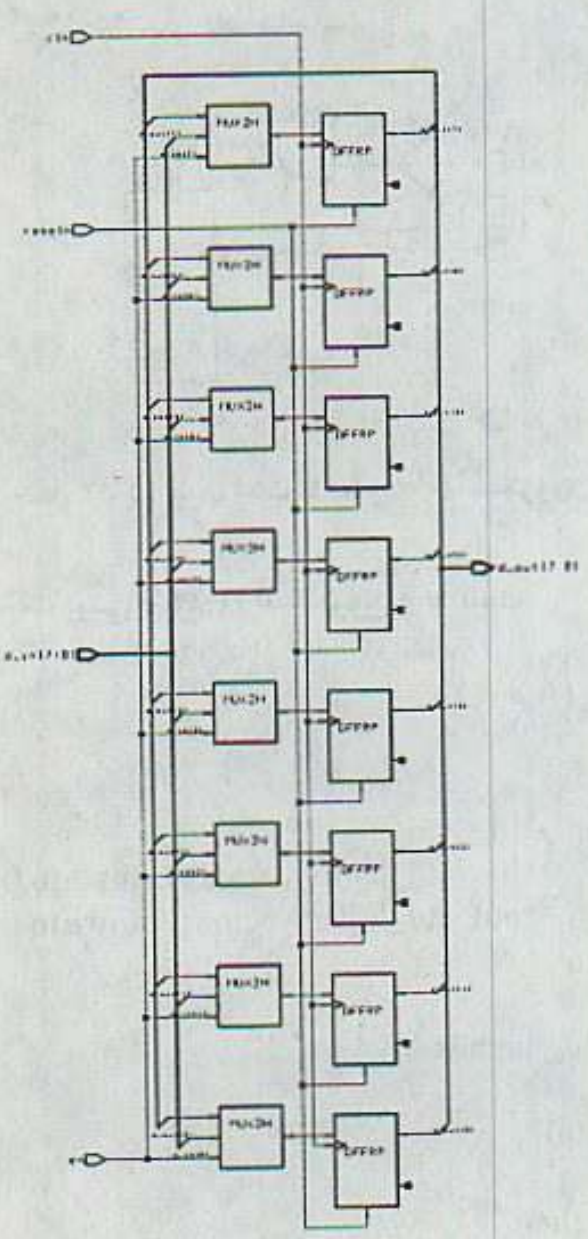
شکل ۱۲-۱۵ فلیپ فلاپ با set و reset آسنکرون

۱۵-۶-۴ رجیستر هشت بیتی با reset آسنکرون و enable

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Entity d_mem  is
port (clk,resetin,en:    in   std_logic;
      d_in:              in   std_logic_vector (7 downto 0);
      d_out:             out  std_logic_vector (7 downto 0));
end;

Architecture rtl of d_mem is
begin
  process (clk,resetn)
  begin
       if  resen='0' then
              d_out<=(others=>'0');
```

```
    elsif clk'event and clk='1' then
      if en= '1' then
          d_out<=d_in;
      end if;
    end if;
  end process;
end;
```

نتیجه سنتز در شکل ۱۳-۱۵ نشان داده شده است.



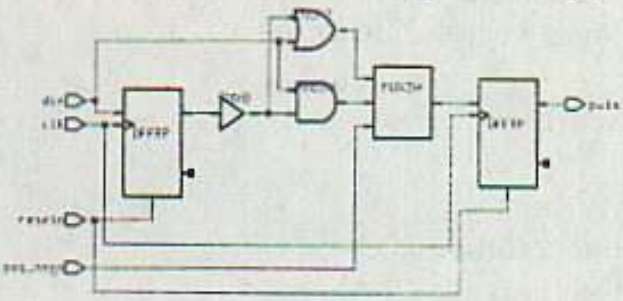
شکل ۱۳-۱۵ رجیستر هشت بیتی با reset آسنکرون و enable

۱۵-۷ مولد پالس کنترل شده با لبه

وظیفه جزء ترکیبی زیر تولید پالس برای یک کلاک در زمانی است‌که ورودی din دارای لبه بالا رونده یا پایین‌رونده باشد. انتخاب لبه بالارونده یا پایین‌رونده توسط سیگنال pos_negn کنترل می‌شود.

```
Entity puls_g is
port (din,pos_negn,clk,resetin:  in   std_logic;
      puls:                      out std_logic);
end;

Architecture rtl of puls_g is
signal din2:std_logic;
begin
   process (clk,resetn)
   begin
        if  resen='0' then
                din2<='0';
                puls<='0';
        elsif clk'event and clk='1' then
                din2<=din;
                if din=pos_negn and din2=not pos_negn then
                        puls<='1';
                else
                        puls<='0';
                end if;
        end if;
   end process;
end;
```

نتیجه سنتز در شکل ۱۴-۱۵ نشان داده شده است.



شکل ۱۴-۱۵ مولد پالس کنترل شده با لبه

۱۵-۸ شمارنده‌ها

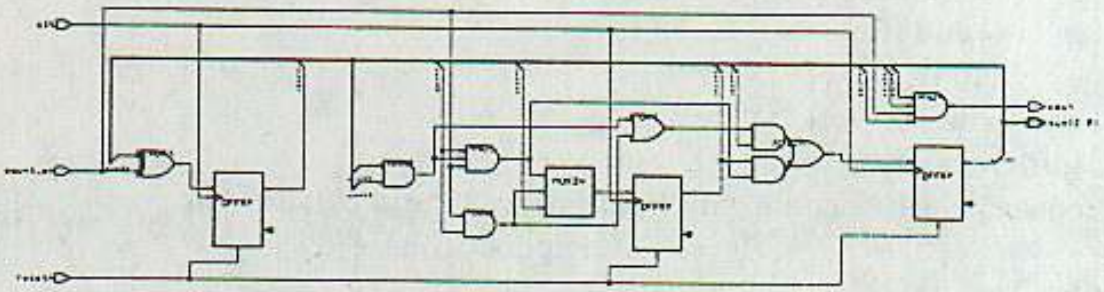
۱۵-۸-۱ شمارنده سه بیتی با بیت نقلی و enable

وقتی از بسته ieee.std_logic_unsigned استفاده می‌کنیم، کافی است برای افزایش یا کاهش مقدار شمارنده توابع '+' یا '-' را به‌کار بریم. اگر شمارنده با نوع بردار معرفی شده باشد، هرگاه تمام بیت‌ها مقدار '1' را اختیار کنند (در مواقع افزایش یا '+') شمارنده به طور خودکار تغییر می‌یابد. چنان‌چه لازم باشد شمارنده در حالت مشخصی فرضاً "111" بایستد، مقدار آن باید قبل از جمع شدن با '1' مورد آزمایش قرار گیرد. در زیر دو برنامه مختلف برای یک شمارنده ارائه می‌شود : یکی بدون چک کردن مقدار شمارنده و دیگری با کنترل مقدار شمارنده. در هر دو مورد نتیجه سنتز یکسان است زیرا عبارت if که حدود شمارنده را چک می‌کند هیچ عملکرد خاصی را به مدار اضافه نمی‌کند. از نقطه نظر نگارشی، مثال با کنترل حدود شمارنده ساده‌تر فهمیده می‌شود اما به نوشتن تعداد خطوط بیشتری نیاز دارد و در نتیجه احتمال خطا بالاتر می‌رود. اگر شمارنده با نوع integer معرفی شده باشد حتماً باید با کنترل مورد باشد زیرا در غیر این صورت در شبیه‌سازی هنگامی که '1' = count و 1+ به اجرا درآید خطا اعلام خواهد شد.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity count3  is
port (clk,resetin,count_en:   in  std_logic;
      sum:                    out std_logic_vector(2 downto 0);
      cout:                   out std_logic);
end;

Architecture rtl of count3 is
signal count:std_logic_vector(2 downto 0);
begin
   process (clk,resetn)
   begin
        if resetn= '0' then
                count<=(others=>'0');
        elsif clk'event and clk= '1' then
           if count_en='1' then
                count<=count+1;
```

```
            end if;
        end if;
    end process;
    sum<=count;
    cout<='1' when count=7 and count_en='1' else '0';
end;

Architecture rtl of count3 is
signal count:std_logic_vector(2 downto 0);
begin
    process (clk,resetn)
    begin
        if resetn= '0' then
                count<=(others=>'0');
        elsif clk'event and clk= '1' then
            if count_en='1' then
             if count/=7 then
                    count<=count+1;
             else
                    count<=(others=>'0');
             end if;
            end if;
        end if;
    end process;
    sum<=count;
    cout<='1' when count=7 and count_en='1' else '0';
end;
```

نتیجه سنتز در شکل ۱۵-۱۵ نشان داده شده است.



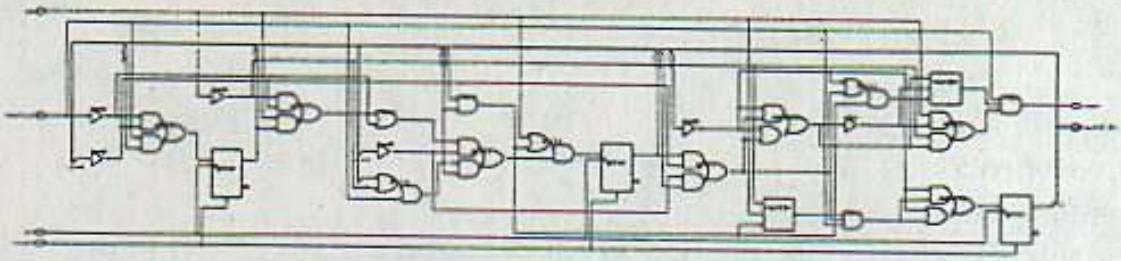
شکل ۱۵-۱۵ شمارنده سه بیتی

۲-۸-۱۵ شمارنده سه بیتی افزایشی ـ کاهشی

اگر مقدار سیگنال ورودی up برابر '1' باشد، شمارنده زیر افزایشی و در غیر این صورت کاهشی خواهد بود. به طور نرمال استفاده از عبارت if یا case هیچ اثری بر روی نتیجه سنتز ندارد و بنابراین انتخاب آنها تنها بر پایه میزان خواناتر بودن برنامه خواهد بود.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity up_down is
port (clk,resetn:      in  std_logic;
      count_en,up:     in  std_logic;
      sum:             out std_logic_vector(2 downto 0);
      cout:            out std_logic);
end;

Architecture rtl of up_down is
signal count:std_logic_vector(2 downto 0);
begin
  process (clk,resetn)
  begin
      if resetn= '0' then
            count<=(others=>'0');
      elsif clk'event and clk= '1' then
        if count_en='1' then
            case up  is
              when '1'          => count<=count+1;
              when others       => count<=count-1;
            end case;
        end if;
      end if;
  end process;
  sum<=count;
  cout<='1' when count_en='1' and ((up='1'  and count=7) or
                    (up='0'  and count=0)) else '0';
end;
```

نتیجه سنتز در شکل ۱۶-۱۵ نشان داده شده است.



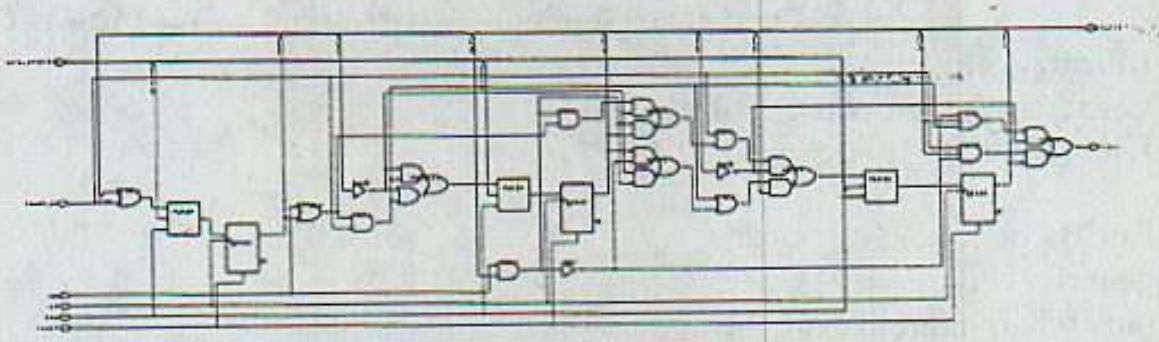
شکل ۱۶-۱۵ شمارنده سه بیتی افزایش ـ کاهشی

۳-۸-۱۵ شمارنده عمومی افزایش ـ کاهشی با ورودی موازی

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity count_par is
generic (N:positive:=3);
port (clk,resetin: in  std_logic;
      count_en:    in  std_logic;
      load,up:     in  std_logic;
      data_in:     in  std_logic_vector(N-1 downto 0);
      sum:         out std_logic_vector(N-1 downto 0);
      cout:        out std_logic);
end;

Architecture rtl of count_par is
signal count:std_logic_vector(N-1 downto 0);
begin
  process(clk,resetn)
  begin
     if reset_n='0' then
            count<=(others<='0');
      elsif clk'event and clk='1' then
        if load='1' then
            count<=data_in;
        elsif count_en='1' then
            case up is
              when '1'          => count<=count+1;
              when others       => count<=count-1;
```

```
            end case;
        end if;
    end if;
end process;
sum<=count;
cout<='1' when count_en='1' and ((up='1'  and count=(2**N)-1) or
                    (up='0'  and count=0)) else '0';
end;
```

نتیجه سنتز در شکل ۱۷-۱۵ آورده شده است.



شکل ۱۷-۱۵ نتیجه سنتز شمارنده عمومی افزایش- کاهش با ورودی موازی (N - ۳)

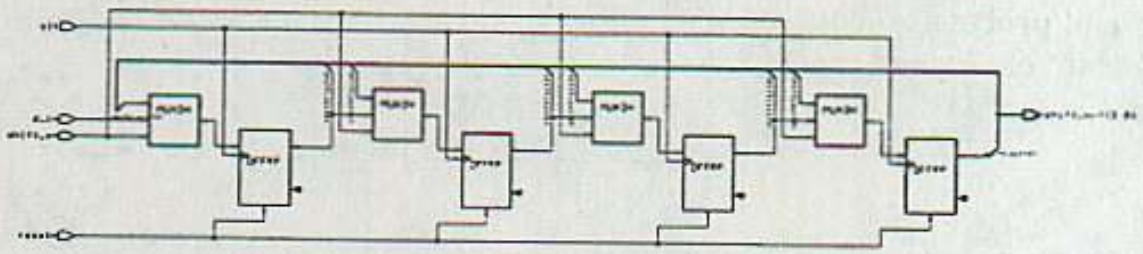
۱۵-۹ شیفت رجیستر

۱۵-۹-۱ شیفت رجیستر چهاربیتی با ورودی سریال وخروجی موازی

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity shift_r is
port (clk,resetn:  in  std_logic;
      d_in:        in  std_logic;
      shift_en:    in  std_logic;
      shift_out:   out std_logic_vector(3 downto 0);
end;

Architecture rtl of shift_r is
signal shift_reg:std_logic_vector(3 downto 0);
begin
```

```
process (clk,resetn)
begin
      if resetn= '0' then
            shift_reg<=(others=>'0');
      elsif clk'event and clk='1' then
        if shift_en='1' then
            shift_reg<=shl(shift_reg, '1');
            shift_reg(0)<=d_in;
        end if;
      end if;
end process;
shift_out<=shift_reg;
end;
```

نتیجه سنتز در شکل ۱۸-۱۵ نشان داده شده است.



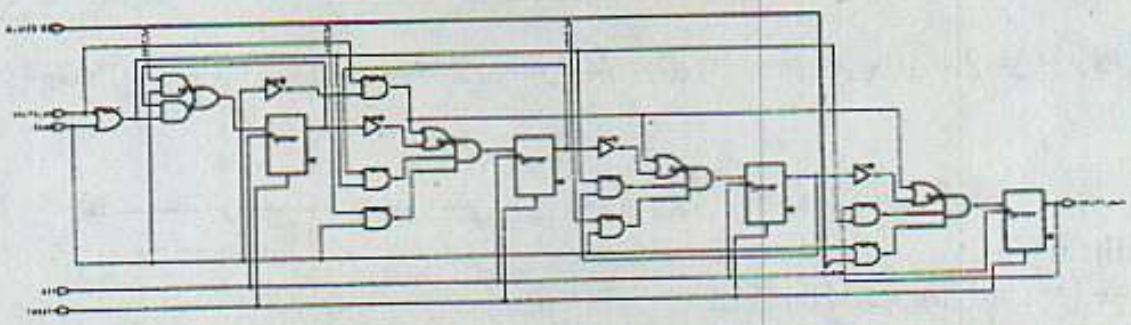
شکل ۱۸-۱۵ نتیجه سنتز یک شیفت رجیستر چهار بیتی با ورودی سریال و خروجی موازی

۲-۹-۱۵ شیفت رجیستر چهار بیتی با ورودی موازی و خروجی سریال

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity shift_p is
port (clk,resetin:      in  std_logic;
      shift_en,load:    in  std_logic;
      d_in:             in  std_logic_vector (3 downto 0);
      shift_out:        out std_logic);
end;
```

```
Architecture rtl of shift_p is
signal shift_reg:std_logic_vector(3 downto 0);
begin
   process (clk,resetn)
   begin
        if resetn= '0' then
              shift_reg<=(others=>'0');
        elsif clk'event and clk='1' then
          if load='1' then
              shift_reg<=d_in;
          elsif shift_en='1' then
              shift_reg<=shl(shift_reg, '1');
              shift_reg(0)<='0';
          end if;
        end if;
   end process;
   shift_out<=shift_reg(3);
end;
```

نتیجه سنتز در شکل ۱۹-۱۵ نشان داده شده است.



شکل ۱۹-۱۵ نتیجه سنتز یک شیفت رجیستر چهار بیتی با ورودی موازی و خروجی سریال

۱۰-۱۵ فیلترها

در زیر مثالی از فیلترهای اولیه آورده شده است.

۱-۱۰-۱۵ فیلتر دیجیتال مقایسه‌کننده چهار ورودی

خروجی این فیلتر با توجه به مقدار سه نمونه‌برداری آخر از ورودی‌ها تعیین می‌شود. به این

ترتیب که :

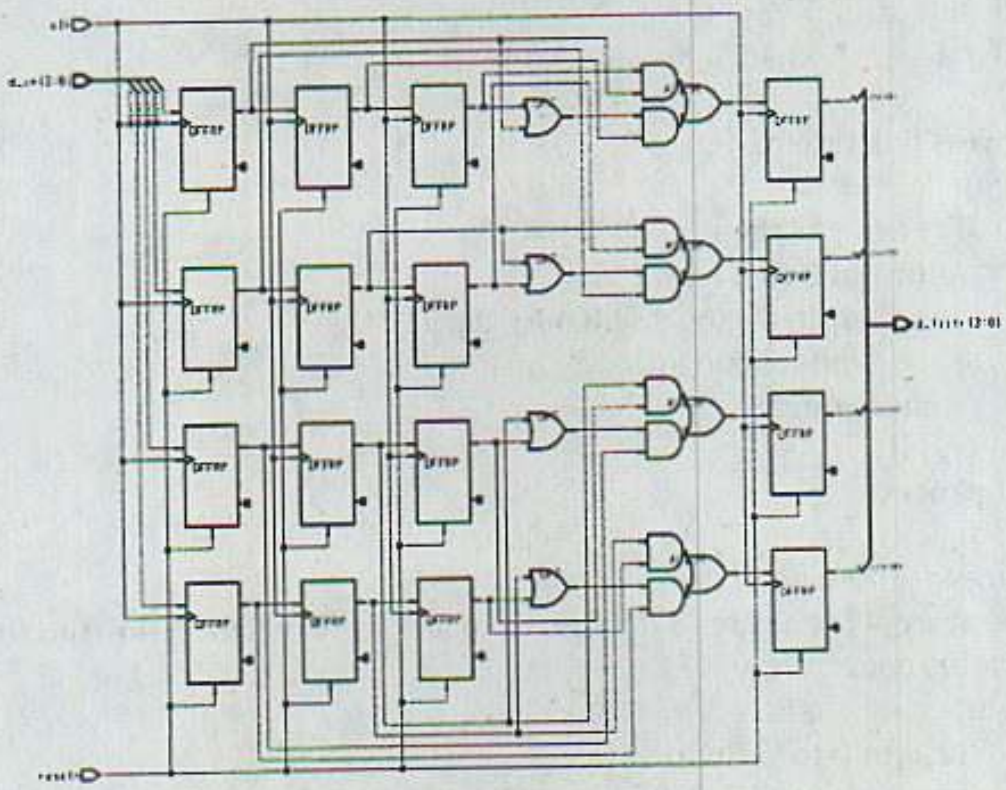
```
if (d_in(N-2) + d_in(N-1) + d_in(N)) > = 2 then
d_filter = '1';
```

در غیر این صورت مقدار فیلتر برابر '0' خواهد بود.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity majority is
port (clk,resetin: in  std_logic;
      d_in:        in  std_logic_vector (3 downto 0);
      d_filter:    out std_logic_vector (3 downto 0));
end;

Architecture rtl of majority is
type array_4 is array (3 downto 0) of std_logic_vector(2 downto 0);
signal shift_data: array_4;
signal d_filter_i:std_logic_vector(3 downto 0);
begin
  process (clk,resetn)
  begin
       if resetn= '0' then
         for i in 0 to 3 loop
              shift_data(i)<=shl(shift_data(i),'1');
              shift_dara(i)(0)<=d_in(i);
         end loop;
       end if;
  end process;

  process(shift_data)
  type array_42 is array (3 downto 0) of std_logic_vector (1downto 0);
  variable count: array_42;
  begin
       for i  in 0 to  3  loop
            count(i):=shift_data(i)(2) + ('0' & shift_data(i)(1))+
                 shift_data(i)(0);
         if cont(i)>=2  then
              d_filter_i(i)<= '1';
```

```
        else
              d_filter_i(i)<= '0';
          end if;
        end loop;
    end process;

    process(clk,resetn)
    begin
        if resetn='0' then
              d_filter<=(others=>'0');
        elsif clk'event and clk='1' then
              d_filtr<=d_filter_i;
        end if;
    end process;
end;
```

نتیجه سنتز در شکل ۲۰-۱۵ نشان داده شده است.



شکل ۲۰-۱۵ نتیجه سنتز یک فیلتر دیجیتال مقایسه‌کننده چهار ورودی

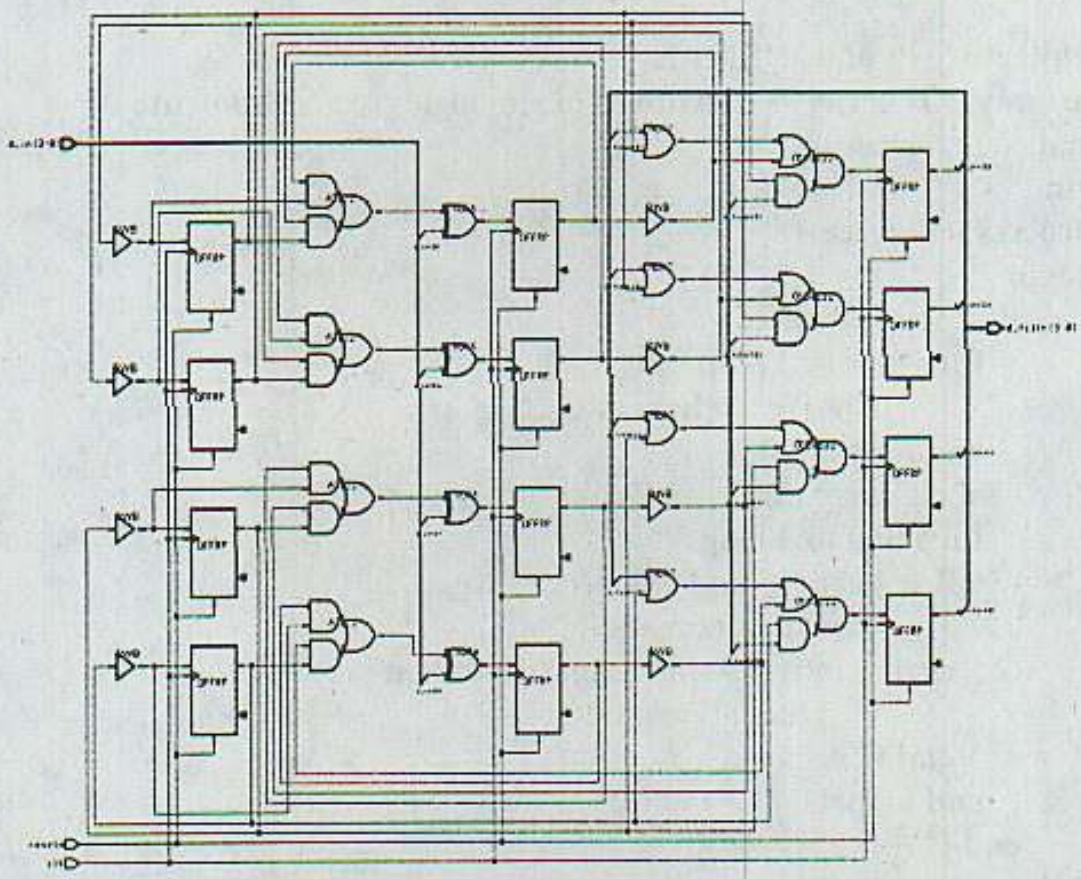
۲-۱۰-۱۵ فیلتر دیجیتال جمعگر چهار ورودی

ورودی‌ها در فیلتر زیر تا زمانی که شمارنده داخلی تا عدد ۳ به طرف بالا یا تا عدد صفر به طرف پایین شمارش نکرده باشد تغییر نمی‌کنند و زمانی که d_in = '1' باشد شمارنده عمل جمع و زمانی که d_in = '0' باشد عمل تفریق را انجام می‌دهد.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity add_filter is
port (clk,resetin: in   std_logic;
      d_in:        in   std_logic_vector (3 downto 0);
      d_filtr:     out std_logic_vector (3 downto 0));
end;

Architecture rtl of add_filter is
type array_4 is array (3 downto 0) of std_logic_vector(1 downto 0);
signal count: array_4;
begin
  process (clk,resetn)
  begin
       if  resetn= '0' then
          for i in 0 to 3 loop
               count(i)<=(others=>'0');
          end loop;
       elsif clk'event and clk='1' then
          for i in 0 to 3 loop
            if d_in(i)= '1' and count(i)/=3 then
                 count<=count+1;
            elsif d_in(i) ='0' and count(i)/=0 then
                      count<=count-1;
            end if;
          end loop;
       end if;
  end process;

  process(clk,resetn)
  begin
```

```
        if resetn='0' then
                d_filtr<=(others=>'0');
        elsif clk'event and clk='1' then
          for i  in 0 to  3  loop
                if cont(i)=3  then
                        d_filtr(i)<= '1';
                elsif count(i)=0 then
                        d_filtr(i)<= '0';
                end if;
          end loop;
        end if;
    end process;
end if;
```

نتیجه سنتز در شکل ۲۱-۱۵ نشان داده شده است.



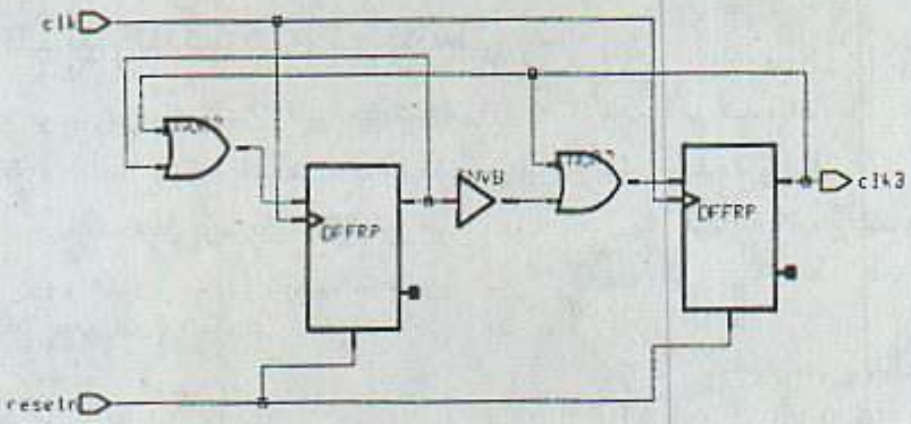
شکل ۲۱-۱۵ نتیجه سنتز یک فیلتر دیجیتال جمعگر چهار ورودی

۱۱-۱۵ تقسیم‌کننده‌های فرکانسی

جزء ترکیبی زیر فرکانس پالس ساعت را به عدد ۳ تقسیم می‌کند. از آنجایی که سیگنال خروجی clk3 مستقیماً از یک رجیستر گرفته می‌شود (بنابراین بدون جهش نویزی است) می‌توان آن را به عنوان پالس ساعت برای بخشهای دیگر طرح مورد استفاده قرار داد. هرگاه فرکانس دیگری مورد نیاز باشد می‌توان به راحتی در مثال زیر تغییرات لازم را اعمال کرد.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity div3 is
port (clk,resetin: in   std_logic;
      clk3:        out std_logic);
end;

Architecture rtl of dav3 is
signal count:std_logic_vector(1 downto 0);
begin
  process (clk,resetn)
  begin
       if resetn= '0' then
              count<=(others=>'0');
       elsif clk'event and clk='1' then
         if count<2 then
              count<=count+1;
         else
              count<=(others=>'0');
         end if;
       end if;
  end process;
  clk3<=count(1)
end;
```

نتیجه سنتز در شکل ۲۲-۱۵ آورده شده است.



شکل ۲۲-۱۵ نتیجه سنتز یک مقسم فرکانسی

فصل شانزدهم

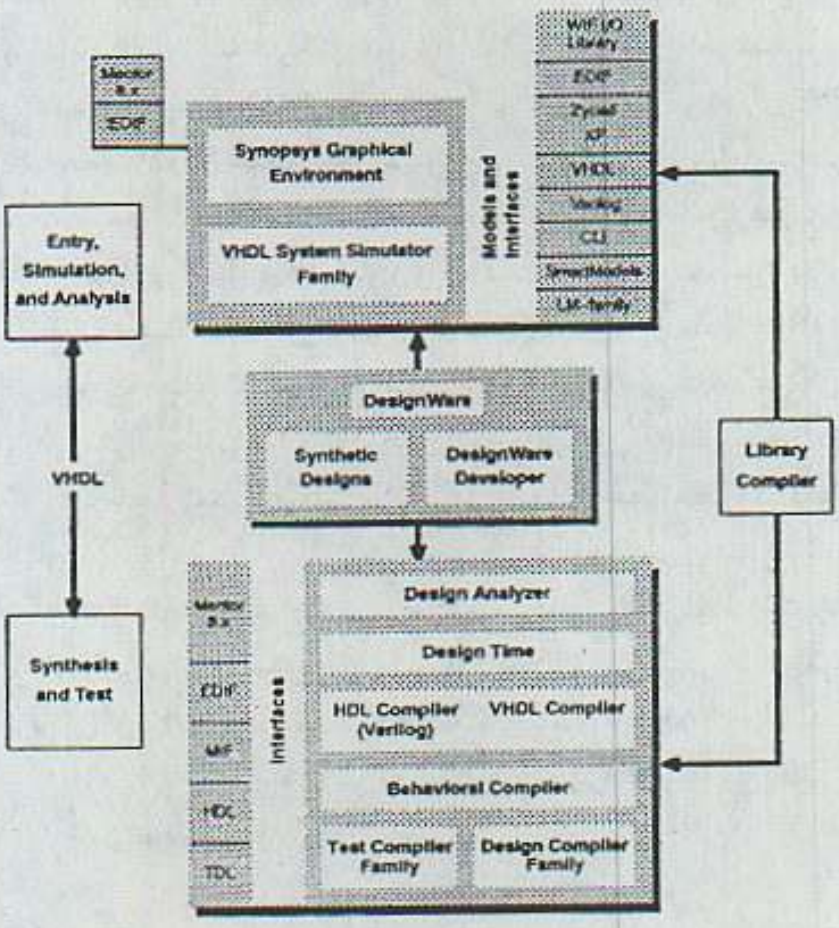
ابزارهای تولید

امروزه تعداد سازندگان ابزارهای VHDL خصوصاً تولیدکنندگان مدل‌های شبیه‌سازی VHDL بسیار زیاد شده است. از آنجایی که VHDL دارای استاندارد IEEE است، بنابراین کلیه شبیه‌سازی‌های VHDL باید قادر باشند مطابق با استانداردهای فوق عمل نمایند. آنچه مدل‌های شبیه‌سازی VHDL را از هم متمایز می‌کند یکی نحوه کامپایل کردن و سرعت عمل شبیه‌سازی و دیگری قابلیت انطباق و ادغام آنها با سایر ابزارهاست. اما در خصوص عمل سنتز، هیچ استانداردی که بتواند مشخص نماید کدام زیر مجموعه از نسخه استاندارد VHDL قابل سنتز شدن است وجود ندارد. ساختارهای زبان که با عمل سنتز سازگار باشند از یک ابزار تا ابزار دیگر بسیار فرق می‌کنند. در عمل سنتز، نکات دیگری هم وجود دارد که اهمیتشان خیلی بیشتر از مشخص کردن آن است که کدام زیر مجموعه VHDL قابلیت سنتز شدن را دارد (رجوع شود به فصل ۱۰، «سنتز در سطح RT»). از جمله این نکات می‌توان به‌طور مثال به نتیجه بهینه‌سازی زمانی و مکانی اشاره کرد، در این مورد ابزارهای مختلفی وجود دارند. در حال حاضر، بازار این تولیدات به‌طور نسبی وضعیت معقولی دارد، به این معنا که هر قدر قیمت ابزار سنتز بیشتر باشد به همان نسبت کیفیت آن نیز بهتر است. در مورد ابزارهای سنتز مخصوص مدارهای ASIC، محصولات Synopsys در دهه ۱۹۹۰ سرآمد سایر محصولات بوده است. اما موسسه Mentor Graphics با تولید ابزارهای سنتزی که مانند Synopsys توانایی سنتز زیر مجموعه وسیعی از VHDL را در حد بالایی دارد (تا ۹۹ درصد) در حال شکستن برتری Synopsys می‌باشد. این تولیدات به‌خاطر آنکه کد VHDL را قادر به انتقال بین این ابزارهای پیشرفته سنتز می‌سازند برای مصرف کنندگان امتیاز بسیار خوبی محسوب می‌شوند.

برای طراحی سریع و کارآمد مدارهای ASIC/FPGA به چندین نوع ابزار پیچیده مانند ابزارهای شبیه‌سازی VHDL، ابزار سنتز، آنالیزکننده زمانی و ابزار ATPG نیاز داریم. گذشته از Synopsys و Mentor Graphics تعداد دیگری از تولیدکنندگان هستند که انواع این محصولات را عرضه می‌کنند. خیلی از این ابزارها می‌توانند با قطعاتی از ابزارهای سایر سازندگان کار کنند. برای آشنایی با ابزارهای تولید امروزی، در زیر به‌طور خلاصه به شرح بخشی از تولیدات Synopsys می‌پردازیم.

Synopsys ۱۶-۱

ابزارهای سنتز Synopsys سرآمد بازار بوده و هنوز هم هستند. Synopsys به غیر از ابزارهای سنتز، تولیدات دیگری نیز دارد و تمام ابزارهای مورد نیاز برای طراحی کامل تراشه‌های ASIC را تولید می‌کند. شکل ۱-۱۶ انواع این محصولات و رابطه بین آنها را نشان می‌دهد.



شکل ۱-۱۶ انواع محصولات مربوط به Synopsys

۱-۱-۱۶ کامپایلر VHDL و آنالیز کننده طرح

کامپایلر VHDL محصول Synopsys، کد VHDL را به یک شماتیک ترجمه می‌کند. این کامپایلر بخش عمده استاندارد VHDL (IEEE-1076) را برای سنتز پوشش می‌دهد. کامپایلر VHDL در هنگام ترجمه کد VHDL به شماتیک، بخشهای مختلف یک طرح مانند فلیپ‌فلاپ‌ها، لچ‌ها، مالتی‌پلکسرها، جمعگرها و غیره را تشخیص می‌دهد. آنگاه کامپیوتر، شماتیک را مطابق با تکنولوژی انتخاب شده بهینه‌سازی می‌کند. برای حصول نتیجه مطلوب از عمل بهینه‌سازی، لازم است پاره‌ای شروط محدودکننده (مانند شروط زمانی) و سایر نیازهای طرح مشخص و معلوم شوند. محدودیتهایی که در محصول Synopsys (V3.3a) به‌کار می‌روند معمولاً شامل موارد زیر هستند (V3.3a):

- Clock frequency
- Multi clock cycles
- Clock skew and clock uncertainty
- Maximum input delay
- Maximum output delay
- Maximum delay
- Minimum delay
- Point to point timing
- Maximum power (ECL designs)
- Maximum transition time
- Maximum fan-out
- Maximum area
- Driving strength
- Load
- Wire load
- Operating conditions
- False paths

شروط محدودکننده فوق را می‌توان با نوشتن در متن، یا مستقیماً در ***ستون دستورات***[1] و یا به صورت گرافیکی به آنالیزکننده طرح وارد کرد.

تدوین نیازهای طرح ممکن است کار سختی به نظر آید، لیکن همیشه این‌طور نیست. اگر فرضاً شما یک طرح پالسی (ساعتی) در دست داشته باشید کافی است با دادن دستور create_clock-period

1- Command Line

clk 25 مشخص نمائید که کلاک تمام فلیپ فلاپ‌های طرح ۴۰ مگاهرتز است.

آنالیزکننده طرح، یک ابزار گرافیکی Synopsys است. در محیط گرافیکی این امکان وجود دارد که از ابزارهایی از قبیل آنچه در زیر نام برده شده، استفاده نمود:

- VHDL Compiler
- Design Compiler
- FPGA Compiler
- Test Compiler
- Test Compiler plus
- Design Ware
- Design Time

تدوین تصویر کاملی از شروط محدودکننده طرح، این امکان را به ابزار Synopsys می‌دهد که کار خود را به نحو احسن انجام دهد. آنالیز طرح بعد از بهینه‌سازی نیز انجام خواهد شد. Synopsys یک تحلیل‌گر زمانی دارد که Design Time نامیده می‌شود. با کمک آن می‌توان به تولید فایل‌های گزارشی پرداخت. این فایل مشخص می‌کند که آیا به‌طور مثال شروط محدود کننده زمانی تأمین گردیده‌اند یا خیر. اگر یک مسیر دیتای غیر عادی وارد طرح شده باشد که قادر به تامین شروط محدوده کننده زمانی نباشد، می‌تواند از تحلیل‌گر زمانی Synopsys خواست که آن مسیر داده را بر روی شماتیک علامت‌گذاری نماید. این عمل باعث می‌شود بتوانیم مشکل را به راحتی تشخیص دهیم. به این ترتیب همیشه می‌توانیم با کلیک بر روی فایل‌های گزارشی تولید شده توسط ابزار Synopsys علل پیدایش مشکلات را به‌طور گرافیکی مشاهده نمائیم.

همچنین می‌توانیم ارتباط بین کد VHDL و شماتیک را با علامت‌گذاری بر روی یکی از خطهای کد و یا یکی از گیت‌ها مشاهده کنیم. به عنوان مثال Synopsys می‌تواند با علامت گذاری روی خط خاصی از کد VHDL، گیتی را که آن خط به‌وجود آورده در شماتیک مشخص کند و یا بالعکس.

۲-۱-۱۶ کتابخانه‌ها و مخازن طرح[1]

Design Ware نام یکی از ابزارهای Synopsys است که به زبان ساده در برگیرنده مجموعه‌ای از کتابخانه‌هایی است که طرق مختلف اجرای اجزای ترکیبی را در خود نگاه می‌دارد.

در حال حاضر نسخه‌ی Synopsys V3.1a حاوی پنج کتابخانه به شرح زیر است:

1- Design Ware

- ALU Family
- Advanced math family
- Basic sequential family
- Fault tolerant family
- Control logic family

بعضی از اجزایی که در این کتابخانه‌ها قرار داده شده‌اند عبارتند از جمعگرها، ضرب کننده‌ها و مالتی‌پلکسرها. ضمناً این امکان نیز وجود دارد که علاوه بر کتابخانه‌ها و مخازن فوق یک Design Ware دیگری حاوی اجزای مورد نظر خود به‌وجود آوریم. امکان تولید و توسعه این مخازن توسط طرح وجود دارد. بسیاری از شرکتهای طراحی نیز Design Ware ویژه خود را دارند که در پکیج‌های طراحی آنان گنجانده شده است.

کامپایلر VHDL اجزای ترکیبی موجود در مخازن طرح، مثلاً یک جمعگر را در حین سنتز شناسایی می‌کند. این جمعگر می‌تواند به روشهای مختلفی پیاده‌سازی شود. مخازن طرح، کلیه جمعگرها را در یکی از کتابخانه‌های خود تست می‌کند و بر حسب اینکه چه شروط و خصوصیاتی برای طرح تعیین شده باشد، آن جمعگری را انتخاب می‌کند که کمترین جا را اشغال نموده و در عین حال شروط زمانی تعیین شده را نیز برآورده سازد.

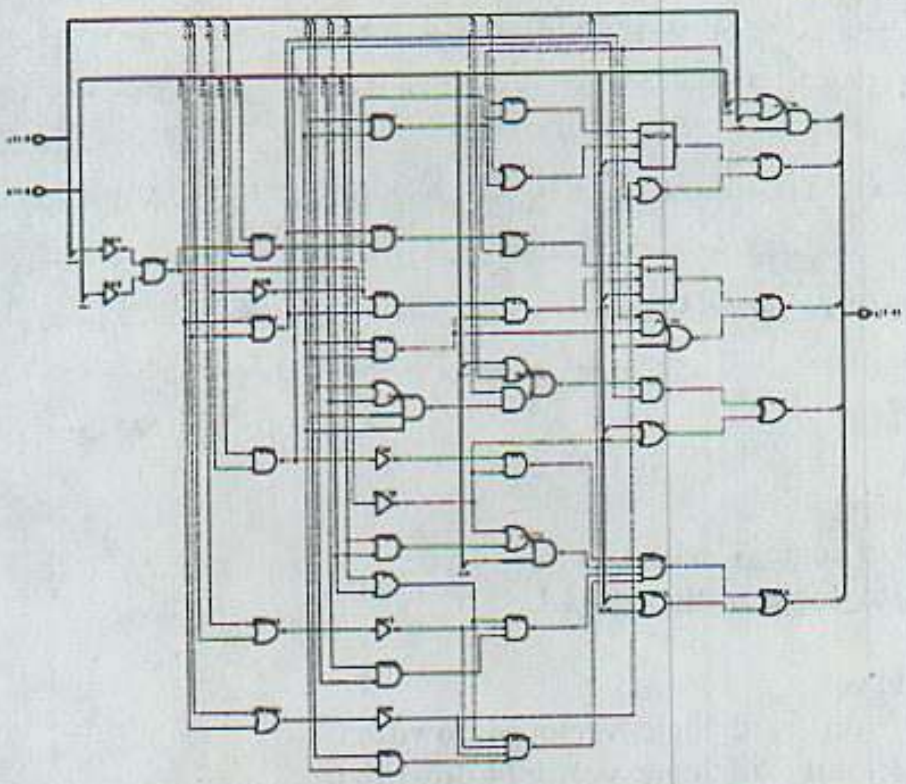
```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;

Entity add is
port (a,b:   in    std_logic_vector (4 downto 0);
      clk3:  out   std_logic_vector (4 downto 0));
end;

Architecture rtl of add is
begin
        q<=a+ b;
end;
```

نخست فرض کنید طرحی داریم که شروط و خصوصیات زیر در ابزار Synopsys برای آن طرح تعیین شده باشد:

```
Set_max_delay 5  - from a  - to q
Set_max_delay 5  - from b  - to q
```

```
Set_max_area 200
Set_operating_conditions "WCCOM"    - library HDCM
Set_wired_load "1k"  - library HDCM    - mode segmented
```

از آنجا که برآوردن شروط محدود کننده زمانی به مدت ۵ نانوثانیه بسیار سخت‌تر از تأمین شروط مکانی (حجمی) به اندازه ۲۰۰ گیت است، بنابراین ابزار Synopsys خود به‌خود آن نوع جمعگر را که از سایر جمعگرها سریع‌تر است انتخاب می‌کند (مانند جمعگر fast carry look-ahead adder). شکل ۲-۱۶ را ببینید.

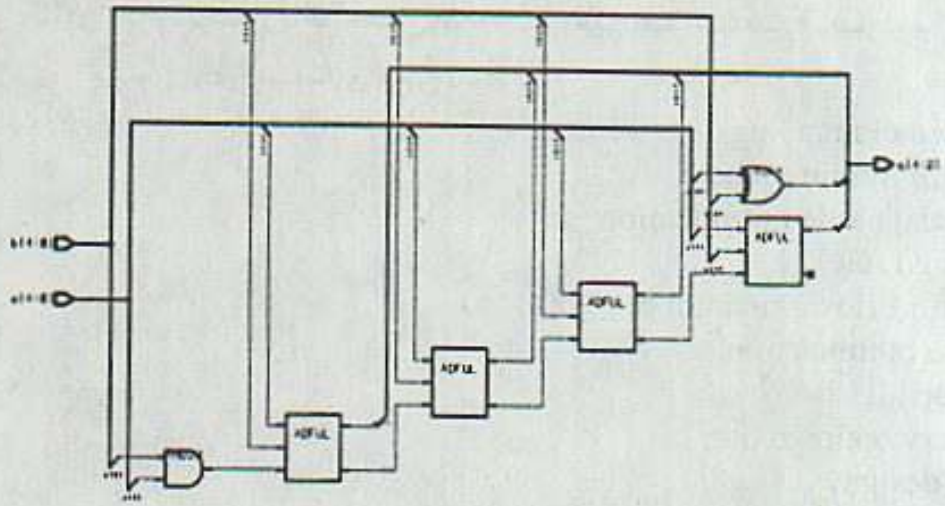


شکل ۲-۱۶ جمعگر سریع (fast carry look-ahead adder)

اما اگر شروط طرح را طوری تعیین کنید که ابزار Synopsys اولویت را به رعایت شروط امکانی (نه شروط زمانی) بدهد، در این صورت ابزار Synopsys جمعگری را انتخاب خواهد کرد که کندتر است ولی به مکان بسیار کمتری نیاز دارد.

آنچه باید برای طراحی این نوع جمعگر در Synopsys تنظیم شود به قرار زیر است:

```
Set_max_delay 15  - from {a,b}  - to q
Set_max_area 50
Set_operating_conditions "WCCOM"    - library HDCM
Set_wired_load "1k"  - library HDCM    - mode segmented
```



شکل ۳-۱۶ جمعگر کند (ripple carry adder)

یک جمعگر سریع در مقایسه با یک جمعگر کند از لحاظ اندازه مکانی (حجم) دو برابر بزرگ‌تر و از لحاظ زمانی دو برابر سریع‌تر است. این مثال نشان می‌دهد که بر حسب آنکه چه شروطی برای ابزار سنتز تعیین شده باشد نتیجه‌های مختلفی از سنتز به دست خواهد آمد. باید دانست که هر دو جمعگر از لحاظ عملکرد کاملاً با ابزار سنتز سازگار هستند. این نکته در هر دو مثال فوق صادق است زیرا در هر دو از کد VHDL واحدی به عنوان نقطه شروع استفاده شده است.

۳-۱-۱۶ کامپایلر طرح

کامپایلر طرح همان ابزار بهینه‌سازی طرح است. معمولاً بهینه‌سازی یک طرح بر اساس شروط و نیازهای تعیین شده کنترل می‌شود. بر پایه این شروط، کامپایلر طرح سعی می‌کند تمام محدودیتهای زمانی را در حداقل مقدار جا برآورده سازد. شروط و نیازهای طرح از لحاظ درجه اولویت با هم فرق دارند. قواعد و قوانین طراحی بالاترین اولویت را دارند. به عنوان مثال بیشترین نقل و انتقال داده‌ها و بیشترین جریان‌دهی خروجی در مقام اول، مهلتهای زمانی در درجه دوم و محدودسازی‌های مکانی (حجمی یا اشغال جا) در درجه آخر قرار دارند.

ابزار Synopsys در طی مرحله بهینه‌سازی می‌تواند به عنوان مثال منابع تغذیه جمعگرها را بین آنها مشترک‌سازی نماید (رجوع شود به فصل ۱۵، «مثالهایی از طراحی و نکات راهنما»).

ابزار Synopsys بر اساس نیازمندی‌ها و شروطی که برای طرح معین شده راساً تصمیم می‌گیرد که آیا منابع را مشترک‌سازی نماید یا خیر و در این کار آن ساختاری را که بیش از همه متناسب با نیازمندی‌های طرح است انتخاب خواهد کرد.

پیاده‌سازی اجزای ترکیبی داخل مخازن طرح نیز در حین عمل بهینه‌سازی صورت می‌گیرد. پارامترهای زیر را می‌توان برای بهینه‌سازی معین نمود:

Incrementel mapping
Prioritize minimum path
Only design rule optimization
No design rule
Map effort (low, medium or high)
In-place optimization
Ungroup all
Baundary optimization
Verify design

Mapping به این معنا است که ابزار بهینه‌سازی سعی می‌کند اجزا را از مخازن کتابخانه‌ای که جای کمتری اشغال می‌کنند یا سریع‌تر قابل دسترسی هستند پیدا نماید. این ابزار، اجزای مورد جستجو را یک به یک امتحان نمی‌کند بلکه معمولاً در هر نوبت چندین جزء نزدیک به هم را یکجا مورد بررسی و امتحان قرار می‌دهد.

In-place optimization زمانی مورد استفاده قرار می‌گیرد که سیم‌بندی مدار انجام شده و به دنبال آن مسائلی از قبیل درایورها بروز می‌نماید. ابزار Synopsys با عمل «بهینه‌سازی در جا» می‌تواند اجزایی را که علامت و مشخصه یکسانی دارند برای رفع مشکل جابه‌جا نماید. جابه‌جایی اجزای مشابه به این معناست که مدار ASIC دیگر نیازی به سیم‌بندی مجدد نخواهد داشت و اجزا فقط در ابزار نقشه‌کش (layout tool) تغییر مکان خواهند داشت.

جدای از پارامترهای بهینه‌سازی فوق می‌توانیم برای بهینه‌سازی، الگوریتم‌های مختلفی را بر حسب اینکه تأکید طرح روی مقدار جای اشغالی یا روی مقدار زمان مصرفی باشد، انتخاب نمائیم. معمولاً شروط و نیازهای تعیین شده اولیه بر روی ابزار نتیجه مطلوب‌تری را به دست می‌دهند.

کامپایلر طرح، یک بهینه‌ساز «ماشین حالت» را نیز در بردارد که می‌تواند به کد نویسی حالت، حداقل‌رسانی حالت و بهینه‌سازی فضای مورد اشغال ماشینهای حالت بپردازد (رجوع شود به فصل ۹، «ماشینهای حالت»).

از آنجا که ابزار Synopsys در این زمینه نسبت به سایر ابزارها سرآمد است در واقع تولیدات تمام سازندگان مدارهای ASIC از کوچک و بزرگ با کامپایلر طراحی Synopsys برای سنتز سازگاری کامل دارند.

همچنین ابزار Synopsys می‌تواند در یک شماتیک معمولی برای بهینه‌سازی خوانده شده و یا آنکه شماتیک را به تکنولوژی دیگری تبدیل نماید. این بدان معناست که اگر شما به توصیف

مشخصه‌های VHDL دسترسی نداشته باشید می‌توانید به راحتی و آسانی تکنولوژی مورد استفاده را تغییر دهید.

۴-۱-۱۶ ابزارهای ATPG

Synopsys دارای دو ابزار ATPG است: یکی ***کامپایلر تست***[1] و دیگری ***کامپایلر تست پیشرفته***[2]. کامپایلر تست یک ابزار ATPG برای اسکن کامل است. در حالی که کامپایلرهای تست پیشرفته یک ابزار ATPG برای اسکن‌های جزئی هستند (به فصل ۱۲ رجوع کنید، «آشنایی با روشهای آزمایش»).

از آنجا که تمام سازندگان مدارهای ASIC بردارهای تست را با هر نوع فورمتی که باشند حمایت می‌کنند، ابزار Synopsys می‌تواند بردارهای تست را به فورمتهای مختلفی که خود بدان مایل است تبدیل نماید.

می‌توانیم علاوه بر تولید بردارهای تست، اسکن مرزی را نیز اجرا نمائیم. اگر پایه‌های ورودی - خروجی طرح را علامت گذاری نمائیم، کامپایلر تست می‌تواند سلولهای اسکن مرزی را که با استاندارد IEEE-1179.1 مطابقت دارند وارد سنتز نماید.

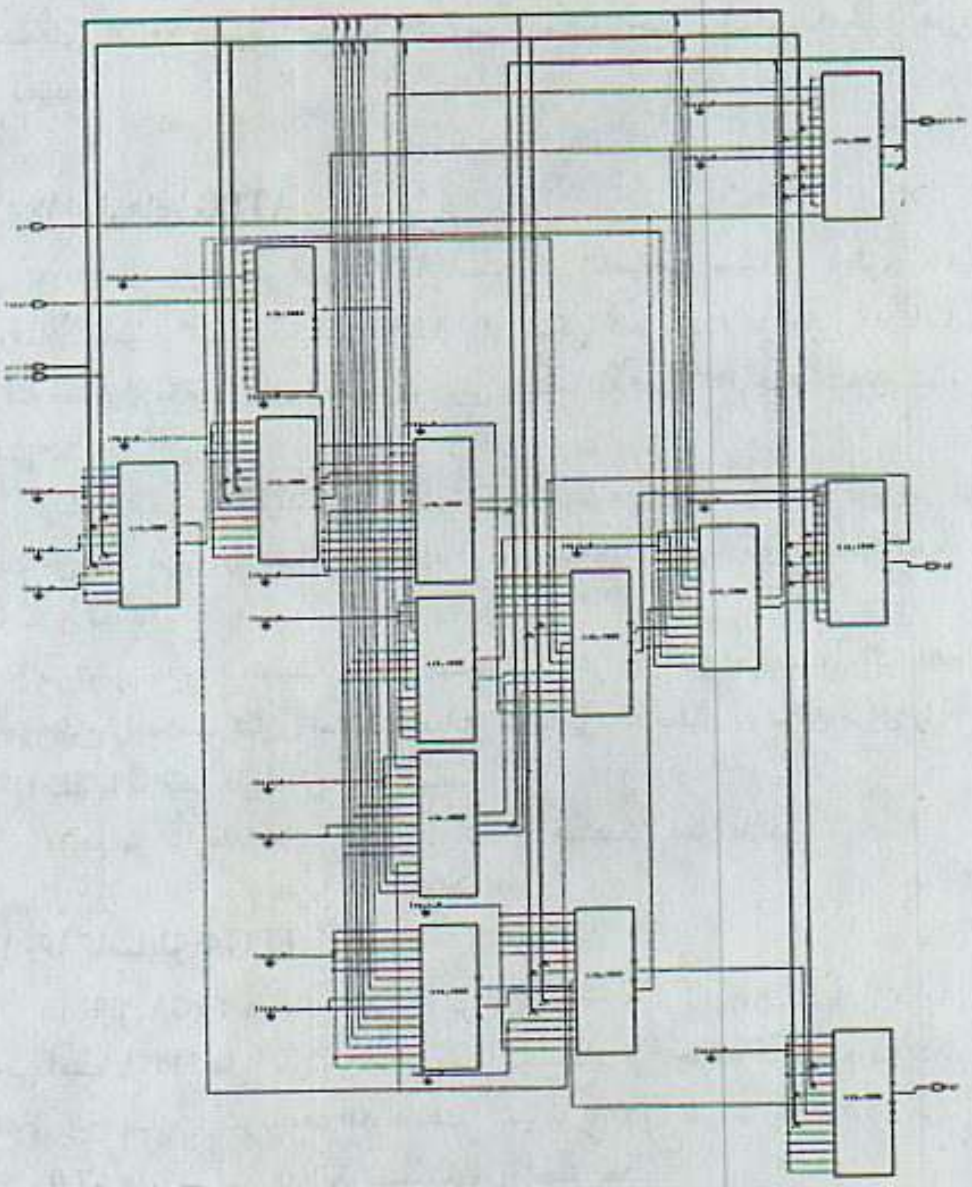
هر دو این ابزارها با سایر ابزارهای Synopsys هماهنگی کامل دارند.

۵-۱-۱۶ کامپایلر FPGA

کامپایلر FPGA در واقع یکی از ابزارهای Synopsys برای بهینه‌سازی در FPGA است و طراحی اغلب FPGA ها را می‌توان با این ابزار انجام داد. از آنجا که شرکت Synopsys یک قرار داد همکاری پنج ساله با شرکتXilinx داشت، کامپایلر FPGA شرکت Synopsys می‌تواند مطابق با ساختار ***CLB***[3]های شرکت Xilinx، بهینه‌سازی را انجام دهد.

کامپایلر FPGA کتابخانه Xilinx X-Blox را نیز حمایت می‌کند. کامپایلر FPGA این کتابخانه را به عنوان یک کتابخانه اضافی در مخازن طرح تلقی می‌نماید. کامپایلر FPGA در خلال سنتز با Xilinx 4000 ، درباره اینکه آیا لازم است یک جزء ترکیبی را از کتابخانه X-Blox انتخاب و آن را برای کمک به زمانبندی‌ها مورد استفاده قرار دهد یا خیر به تحقیق می‌پردازد.

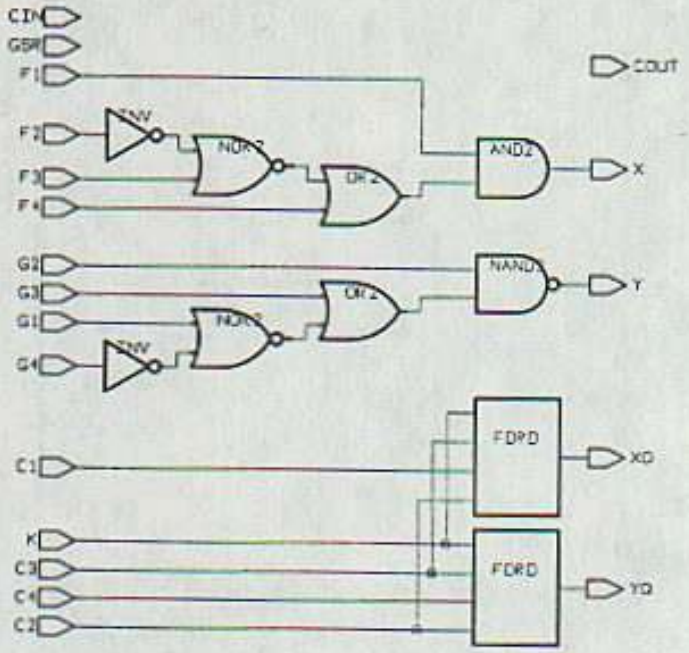
1- Test Compiler
2- Test Compiler Plus
3- Configurable Logic Block



شکل ۴-۱۶ CLB ها

شکل ۴-۱۶ نشان می‌دهد که چگونه یک کامپایلر FPGA می‌تواند طرح را به بلوک‌های CLB نگاشت دهد. در این مرحله آنالیز زمان‌بندی طرح امکان‌پذیر می‌گردد. سپس ابزار Synopsys مقدار تأخیری را که بعد از سیم‌بندی به‌وجود می‌آید برآورد خواهد کرد و این برآورد معمولاً تا حدود ۱۰ درصد با تاخیر واقعی اختلاف دارد.

کامپایلر FPGA همچنین به ما امکان می‌دهد که محتویات بلوک‌های CLB را از لحاظ توانایی‌های منطقی آن مورد بررسی قرار دهیم. در این بلوک‌ها هم معادلات منطقی و هم شکلهای گرافیکی قابل مشاهده خواهند بود (شکل ۵-۱۶).



شکل ۵-۱۶ محتویات یک بلوک CLB

کامپایلر FPGA همچنین قادر است XNF netlist را که حاوی اطلاعات مربوط به محدودسازی زمانی است به نمایش بگذارد. آنگاه ابزار Xilinx می‌تواند این اطلاعات را در سیم‌بندی یا به عبارت بهتر در سیم‌بندی مبتنی بر فاکتورهای زمانی مورد استفاده قرار دهد. این کامپایلر همچنین قادر است XNF netlist را برای تغییر علامت‌گذاری، بهنیه‌سازی یا تبدیل تکنولوژی دریافت کند.

۶-۱-۱۶ شبیه‌ساز VHDL

شبیه‌ساز VSS در ابزار Synopsys یک شبیه‌ساز VHDL است که تماماً با استاندارد VHDL -87 انطباق دارد. VSS یکی از سریع‌ترین شبیه‌سازهای VHDL در بازار است. در حال حاضر امکان شبیه‌سازی کد VHDL و همچنین انجام تست زمانی مدار در سطح گیتی در یک شبیه‌ساز واحد وجود دارد.

برای آنکه شبیه‌ساز VSS بتواند از عهده همه وظایف شبیه‌سازی برآید سه «موتور» مختلف در آن کار گذاشته شده است: یکی برای ترجمه، یکی برای کامپایل کردن و سومی برای شبیه‌سازی در سطح گیتی. این موتورهای مختلف توسط کاربر قابل استفاده نیستند ولی آنچه مهم است آن است که VSS مناسب‌ترین موتورها را برای هر یک از موارد شبیه‌سازی انتخاب می‌کند و به این گونه اجرای کامل شبیه‌سازی از سطح عملکردی و رفتاری گرفته تا سطح گیتی با استفاده از یک شبیه‌ساز واحد امکان‌پذیر می‌گردد.

فصل هفدهم

سنتز رفتاری

۱-۱۷ مقدمه

اغلب موارد پیدا کردن یک طرح خوب و مناسب مستلزم انجام یک رشته عملیات تکراری و پشت سرهم است. اگر این کار را در سطوح بالاتر توصیفی انجام دهیم روند تکرار ساده‌تر و تحلیل مطمئن‌تر خواهد شد. با کمک سنتز رفتاری می‌توانیم حجم، سرعت و هزینه یک ASIC را برآورد کنیم.

یک جنبه مهم دیگر سنتز رفتاری رقابت کردن آن با کد ماشینی CUP ها در آینده است. سنتز رفتاری در بسیاری از موارد در مقایسه با عملکرد CUP ، می‌تواند برای سیستمهای مجتمع راه‌حلهایی کوتاه‌تر، ارزان‌تر و مطمئن‌تر ارائه دهد.

سنتز رفتاری همان چیزی است که بسیاری از طراحان به دنبال آنند. این بدان معناست که کارهایی مثل طراحی ماشینهای کنترلی برای اشتراک توابع، مدارات واسط RAM و مانند آن از طریق سنتز رفتاری به‌طور اتوماتیک انجام‌پذیر می‌گردد.

تعریف

سنتز رفتاری (سنتز الگوریتمی سطح بالا) در اصل به معنای عبور از یک مشخصه الگوریتمی و رسیدن به سطح RT است که آن رفتار را پیاده‌سازی کرده و به اجرا در می‌آورد.

۱-۱-۱۷ اصطلاحات

مسیر داده[1]

مسیر داده شامل تعدادی از اجزای ترکیبی متصل به هم در سطح RT است. از این اجزای ترکیبی می‌توان به عنوان مثال به ضرب‌کننده‌ها، جمعگرها، رجیسترها و تفریقگرها اشاره کرد.

واحد کنترل[2]

واحد کنترل، یک ماشین حالت است که سیگنال‌های کنترلی برای مسیر داده را تولید می‌کند.

زمان‌بندی[3]

زمان‌بندی به معنای تعیین پالس ساعتی است که در آن عملیات خواندن و نوشتن I/O و دسترسی به حافظه باید به اجرا درآیند. یک جدول زمانی برای بهینه‌سازی سرعت استفاده از منابع با توجه به شروط زمانی تعیین شده برای توابع، تنظیم می‌گردد. با سنتز رفتاری می‌توان توابع مختلف را با استفاده از منابع یکسان و مشترک پیاده‌سازی کرد و به این ترتیب تعداد منابع موجود در تراشه و به عبارت دیگر فضای تراشه و در نتیجه بهای ASIC را کاهش داد. واحد برنامه‌ریز زمانی در داخل ماشین حالت (واحد کنترل) عمل می‌کند.

تخصیص منابع[4]

تخصیص منابع به معنای انتخاب اپراتورهایی است که زمان‌بندی آنها در سیکل‌های مختلف پالس ساعت توسط برنامه‌ریز زمانی تعیین شده است.

1- Data Path

2- Control Unit

3- Scheduling

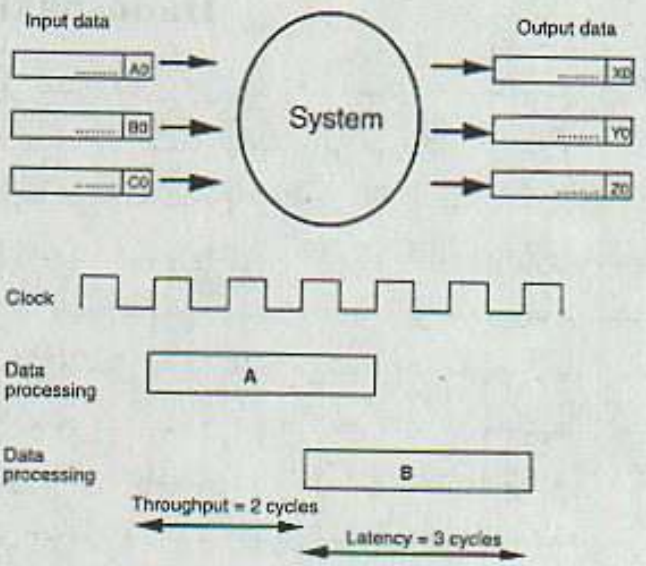
4- Resource Allocation

سیکل‌های خفته[1]

سیکل‌های خفته به آن تعداد از پالس‌هایی گفته می‌شود که برای اجرای هر تابع مورد نیاز است (شکل ۱-۱۷).

تسلسل زمانی داده‌ها[2]

تسلسل زمانی داده‌ها تعیین می‌کند که هر چند وقت یک بار می‌توان داده‌های جدید را با رعایت سیکل زمانی به داخل سیستم فرستاد (شکل ۱-۱۷).



شکل ۱-۱۷ سیکل‌های خفته و تسلسل زمانی داده‌ها

مثال زیر از دو عمل ضرب و سه عمل جمع در سطح RT استفاده کرده است :

$$a \Leftarrow (b * c) + (e * 34) + 56$$

سنتز رفتاری می‌تواند برای به اشتراک گذاشتن منابع مورد استفاده قرار گیرد. این بدان معناست که از طریق سنتز رفتاری می‌توان ظرفیت سخت‌افزاری سیستم را تا حد یک ضرب‌کننده و یک جمعگر کاهش داد. مزیت آن صرفه‌جویی در فضا و عیب آن کندتر شدن محاسبات یا به عبارت دیگر طولانی‌تر شدن سیکل‌های خفته و تسلسل زمانی داده‌هاست.

1- Latency

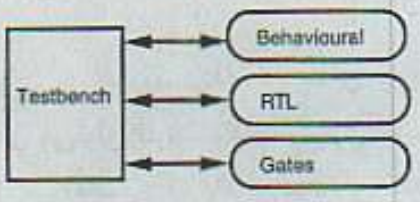
2- Throughput

محدودیتهای زمانی در سطح RT برای هر یک از توابع به‌طور موردی و ***محلی***[1] نوشته می‌شوند در حالی که در سطح رفتاری این محدودیتها برای تمام پروسس‌ها (به‌طور سراسری و ***همگانی***[2]) نوشته می‌شوند. در سطح رفتاری بهینه‌سازی در سرتاسر مراحل کلاک و بر روی اجزای ترکیبی صورت می‌گیرد، در حالی که در سطح RT بهینه‌سازی در هر سیکل پالس ساعت و بر روی توابع بولی (گیت‌ها) انجام می‌گیرد. در سطح RT طراح باید بهره‌وری از منابع را شخصاً مشخص کند، این عمل در سطح رفتاری به‌طور خودکار و اتوماتیک انجام می‌شود.

۲-۱۷ عمل Handshaking

از آنجایی که ابزار سنتز رفتاری می‌تواند رفتار سیکل به سیکل سیگنال‌های I/O را تغییر دهد، در عین حال هم می‌تواند باعث بروز مسائل و مشکلاتی در خلال طراحی و تست شود. اگر قرار باشد از یک محیط تست برای آزمایش کلیه سطوح (رفتاری، RT و گیتی، شکل ۲-۱۷) استفاده کنیم، باید مکانیسم Handshaking را به‌کار گیریم. وقتی که یک کد VHDL رفتاری شبیه‌سازی می‌شود، داده‌ها دقیقاً همان طور که در کد تعیین شده‌اند نوشته یا خوانده می‌شوند. بعد از سنتز رفتاری، رفتار سیکل به سیکل سیگنال‌ها به‌طور معمول دچار تغییر می‌شود. اگر محیط آزمایش انتظار داشته باشد که خواندن و نوشتن سیگنال‌ها در مدل سطح RT یا گیتی دقیقاً در سیکل‌های تعیین شده پالس ساعت انجام شود، جواب آزمایش منفی خواهد بود. راه حل این مشکل استفاده از سیگنال‌های handshake است. این سیگنال‌ها زمان خواندن و نوشتن را برای سیگنال‌های I/O مشخص می‌کنند. بر حسب اینکه آیا بلوکی که باید داده‌ها را بفرستد زمان پاسخ ثابتی دارد یا خیر باید از handshake یک طرفه یا دو طرفه استفاده نمود.

مکانیسم handshake استفاده از محیط آزمایش مشترک را برای کلیه سطوح امکان‌پذیر می‌سازد (شکل ۲-۱۷).



شکل ۲-۱۷ پیکره‌بندی‌های گوناگون محیط آزمایش

1- Locally

2- Globally

دقیقاً همین مشکل در زمانی که داده‌ها باید بین بلوک‌های سلسله مراتبی نقل و انتقال یابند پیش می‌آید، مخصوصاً هنگامی‌که هر یک از بلوک‌ها با استفاده از یک ابزار سنتز رفتاری سنتز شده باشند. از این رو باید از عمل handshaking برای انتقال داده‌های بین بلوک‌ها استفاده کرد.

۱-۲-۱۷ پروتکل handshake یک طرفه[1]

از handshake یک طرفه هنگامی استفاده می‌شود که زمان پاسخ ثابت باشد. برای پیاده‌سازی این مکانیسم فقط به یک سیگنال نیاز داریم : سیگنال ready_for_data (شکل ۳-۱۷).

شکل ۳-۱۷ handshake یک طرفه

شرط ضروری برای بلوک ۱ در شکل ۳-۱۷ آن است که باید پس از فعال شدن سیگنال ready_for_data یک زمان پاسخ ثابت داشته باشد. این روند به‌طور نرمال در مورد محیطهای آزمایش و نه برای بلوک‌های سلسله مراتبی به‌کار گرفته می‌شود. برای بلوک‌هایی که زمان پاسخ ثابت ندارند باید از handshake دو طرفه استفاده کرد.

مثال زیر بیانگر بلوک ۱ و از نوع handshake یک طرفه است. در این برنامه، پروسس در انتظار فعال شدن سیگنال ready_for_data می‌ماند و پس از فعال شدن آن به نوشتن داده بر روی I/O می‌پردازد.

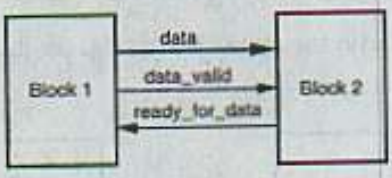
```
process
begin
  for i in 0 to 7 loop
      wait until clk='1';
      while ready_for_data='0' loop;
            wait until clk='1';
      end loop;
      wait until clk='1';
      data<=data_out(i);
```

1- One-Way Handshake Protocol

```
        wait until clk='1';
    end loop;
end process;
```

۲-۲-۱۷ پروتکل handshake دوطرفه[1]

از handshake دوطرفه هنگامی استفاده می‌شود که زمان پاسخ ثابت نباشد. برای پیاده‌سازی و اجرای آن به دو سیگنال نیاز داریم : ready_for_data و data_valid (شکل ۴-۱۷).



شکل ۴-۱۷ handshake دو طرفه

مثال زیر بیانگر بلوک ۲ و از نوع handshake دوطرفه است. در این برنامه، هنگامی که پروسس آماده پردازش داده باشد سیگنال ready_for_data را فعال می‌کند. سپس قبل از خواندن داده از I/O در انتظار فعال شده (رسیدن) سیگنال data_valid می‌ماند.

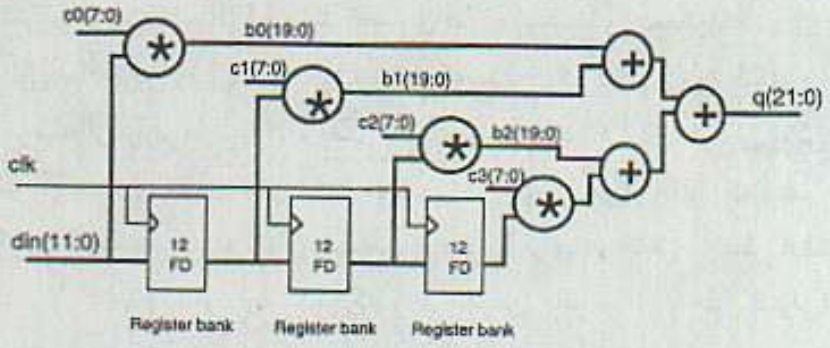
مثال :

```
process
begin
    main: loop
        wait until clk='1';
        ready_for_data<='1';
        while data_valid='0' loop;
                wait until clk='1';
        end loop;
        wait until clk='1';
        ready_for_data<='0';
        data_in<=data;
        wait until clk='1';
        q:=dada_in * b;
        ...
end process;
```

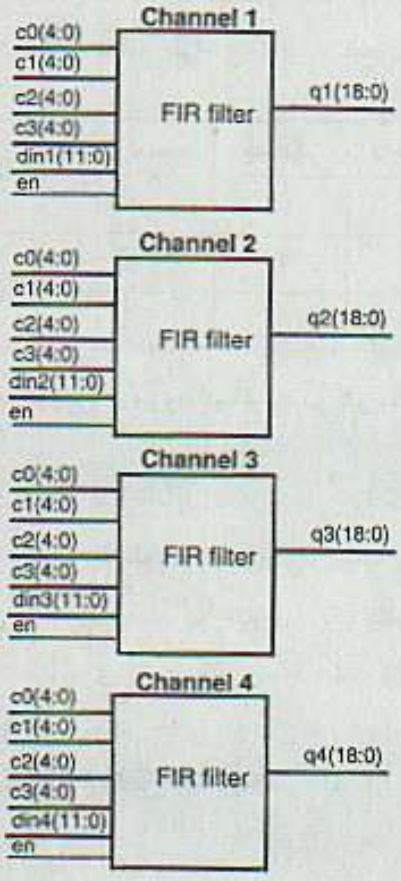
1- Two-Way Handshake Protocol

۳-۱۷ مثالی از سنتز رفتاری/RLT ـ فیلتر FIR

در این بخش چگونگی استفاده از سنتز رفتاری برای طراحی یک فیلتر FIR توضیح داده می‌شود (شکل ۵-۱۷).



شکل ۵-۱۷ فیلتر FIR



شکل ۶-۱۷ فیلتر FIR چهار کاناله

چنان‌چه بخواهیم فیلتر FIR چهار کاناله در شکل ۶-۱۷ را در سطح RT طراحی کنیم، می‌توانیم از هر یک از راه‌حلهای زیر استفاده نماییم :

۱- فراخوانی چهار فیلتر FIR

۲- مشترک‌سازی یک فیلتر FIR برای هر چهار کانال

۳- مشترک‌سازی فیلتر FIR و مشترک‌سازی ضرب‌کننده‌ها و جمعگرها

راه حل اول ساده‌ترین روش است. کافی است که طراح یک فیلتر از نوع شکل ۵-۱۷ را طراحی و سپس چهار مرتبه آن را فراخوانی کند، یعنی یک نوبت برای هر کانال. در این روش طراحی تعداد ۱۶ = ۴ × ۴ ضرب کننده و ۱۲ جمعگر تولید می‌شود. در صورتی می‌توان از یک فیلتر برای هر چهار کانال استفاده کرد (راه حل دوم) که بتوان زمانهای خاصی را برای هر کانال در نظر گرفت. این راه حل چهار ضرب‌کننده و سه جمعگر تولید می‌کند. در عوض باید یک واحد کنترلی برای کنترل مشترک‌سازی منابع میان کانالها طراحی گردد. راه حل سوم پیچیده‌ترین راه حل است. طراح در این روش علاوه بر مشترک‌سازی یک فیلتر FIR برای هر چهار کانال باید ضرب‌کننده‌ها و جمعگرها را نیز مشترک نماید. این شیوه طراحی فقط یک ضرب‌کننده و یک جمعگر تولید خواهد کرد. یک واحد کنترلی نیز باید برای کنترل مشترک‌سازی منابع طراحی گردد. نتایج در جدول ۱-۱۷ خلاصه شده‌اند.

	ضرب‌کننده	جمعگر	واحد کنترل
راه حل اول	۱۶	۱۲	۰
راه حل دوم	۴	۳	۱
راه حل سوم	۱	۱	۱

جدول ۱-۱۷ تعداد اجزای مورد استفاده در هر یک از راه‌حلها

به هنگام طراحی در سطح RT ، طراح باید مدت زمانی را که هر مسیر داده مصرف می‌کند به‌طور دقیق بداند. بسته به آنکه مسیر داده طولانی‌تر از عرض یک پالس ساعت است یا خیر، طراح می‌تواند از مسیر داده ***خط لوله‌ای***[1] استفاده کند، به عبارت دیگر نتایج را به‌طور موقت در یک بانک رجیستری ذخیره کند. در یک نوع تکنولوژی این امکان وجود دارد که ضرب‌کننده و جمعگر هر دو را در یک دوره تناوب پالس ساعت جا دهیم، در حالی که در نوع دیگر تکنولوژی ممکن است چندین پالس ساعت موردنیاز باشد. کد VHDL در سطح RT آنچه را که باید در هر دوره تناوب پالس ساعت

1- Pipeline

انجام گیرد و همچنین تعداد بانکهای رجیستری موردنیاز را تعریف و تعیین نماید این بدان معناست که طرح در سطح RT تا حدی وابسته به تکنولوژی خواهد بود.

فرض کنید که دو نوع تکنولوژی A و B را در اختیار داشته باشیم. در تکنولوژی A عمل ضرب و دو عمل جمع در ۳۴ نانوثانیه انجام می‌شوند در حالی که در تکنولوژی B عمل ضرب و دو عمل جمع در ۱۷ نانوثانیه انجام می‌گیرند. همچنین فرض کنید که دوره تناوب پالس ساعت ۱۸ نانوثانیه باشد. در این مورد تکنولوژی A باید نتایج را در یک بانک رجیستری به‌طور موقت ذخیره کند، در حالی که تکنولوژی B می‌تواند تمام عملیات را در ظرف مدت یک پالس ساعت انجام دهد. به علاوه، چنان‌چه قرار باشد مشترک‌سازی منابع نیز انجام شود، طراح باید به ردیابی دستی پرداخته و معلوم نماید که چه وقت منابع در سطح RT در دسترس خواهند بود. در موارد مشترک‌سازی منابع اگر بخواهیم به یک معماری در سطح RT مطلوب برسیم باید اطلاعات مربوط به سیگنال‌های خفته و تسلسل زمانی داده‌ها را که از محدودیتهای مهم طراحی محسوب می‌شوند به دست آوریم. منظور از سیگنال‌های خفته آن است که برای محاسبه مقادیر داده چند سیکل ساعت موردنیاز می‌باشد، به عبارت دیگر تعیین تعداد سیکل‌های ساعتی است که برای ورود دیتا به فیلتر و خروج آنها از فیلتر موردنیاز خواهد بود. منظور از تسلسل زمانی داده‌ها این است که بدانیم هر چند وقت یک بار مقادیر جدید توسط فیلتر تولید می‌شوند. در مواردی که منابع مشترک‌سازی می‌شوند فیلتر نمی‌تواند با هر سیکل ساعت یک مقدار جدیدی را تولید کند. این دو محدودیت بر روی معماری برنامه در سطح RT اثر می‌گذارند. در حالی که ابزار سنتز رفتاری قادر است همان کد VHDL را پابرجا نگه دارد. ابزار سنتز رفتاری به‌طور اتوماتیک معماری‌های گوناگونی را برای رعایت محدودیتهای سیکل نهفته و تسلسل زمانی داده به وجود می‌آورد.

اگر طراح به ابزار سنتز رفتاری دسترسی داشته باشد، کد VHDL را می‌تواند به صورت زیر بنویسد :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Package types is
Constant L_in:  positive:=12;                -- Number of bits in input data.
Constant L_mult: positive:=8;                -- Number of bit in mult.
Constant L_ch:  positive:=14;                -- Number of channels.
Constant L_reg:  positive:=L_IN+L_mult;      -- Number of bits in register
                                             -- after mult.
Constant L_data:  positive:=L_reg+2;         -- Number of bits in output.

subtype data is std_logic_vector(L_data-1 downto 0);      -- data out
type data_array is array (integer range 0 to L_ch-1) of data;
```

```
subtype reg is std_logic_vector(L_reg-1 downto 0);        -- reg after mult
type data_reg is array (integer range 0 to L_ch-1) of reg;
subtype reg_in is std_logic_vector(L_in-1 downto 0);      -- input reg
type data_reg_in is array (integer range 0 to L_ch-1) of reg_in;
end;

Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.types.ALL;

Entity fir is
port (d_in:            in    reg_in;
      c0,c1,c2,c3:     in    std_logic_vector(L_mult-1 downto 0);
      clk,resetn,en:   in    std_logic;
      done:            out   std_logic;
      d_out:           out   data_array);
end;

Architecture behv of fir is
begin
  p1: process
  variable d_in_i:          reg_in;
  variable d1,d2,d3:        data_reg_in;
  variable b0,b1,b2,b3:     reg;
  variable i1,i2:           std_logic_vector(L-reg downto 0);
  variable d_out_i:         data;
  variable ch:              std_logic_vector(1 downto 0);
  variable ch_nr:           integer range 0 to L_ch;
  begin
      res_loop:loop
        done<='0';
        d_out_i:=(others=>'0');
        ch:=(others=>'0');
        d_out<=(others=> (others=>'0'));
            main:loop
              wait until clk='1';
              exit res_loop when resetn='0';
              ch_nr:=conv_integer(ch);
```

```
                done<='0';
                wait until clk='1';
                exit res_loop when resetn='0';

                while en='0' loop
                    wait until clk='1';
                    exit res_loop when resetn='0';
                end loop;

                d_in_i:=d_in;
                b3:= d3(ch_nr) * c3;
                b2:= d2(ch_nr) * c2;
                b1:= d1(ch_nr) * c1;
                b0:= d_in_i * c0;

                i1:=('0' & b0) + b1;
                i2:=('0' & b2) + b3;

                wait until clk='1';
                exit res_loop when resetn='0';

                d_out_i:=( '0' & i1) + i2;
                done<='1';
                b3(ch_nr):=d2(ch_nr);
                b2(ch_nr):=d1(ch_nr);
                b1(ch_nr):=d_in_i;

                wait until clk='1';
                exit res_loop when resetn='0';

                d_out(ch_nr)<=d_out_i;
                ch:=ch+1;

                wait until clk='1';
                exit res_loop when resetn='0';
            end loop;
        end loop;
    end process;
end;
```

کد VHDL فوق تعیین نمی‌کند که چه منابعی باید به‌طور مشترک مورد استفاده قرار گیرند. این تصمیم بر عهده ابزار سنتز رفتاری است که به طراح اجازه می‌دهد الگوریتم‌ها ـ یعنی رفتار ـ را به درستی تنظیم نماید. منابعی که باید مشترک‌سازی شوند، سیگنال‌های خفته و تسلسل زمانی داده‌ها برای ابزار سنتز رفتاری به عنوان عوامل محدودکننده تعریف می‌شوند. اگر بخواهیم همه این عوامل را در سطح RT توصیف کنیم بسیار وقت‌گیر خواهد بود زیرا مشترک‌سازی منابع باید در کد VHDL توصیف شود. طراح همچنین باید سیگنال‌های خفته و تسلسل زمانی داده‌ها را در کد VHDL توصیف نماید و برای این کار باید آن نوع معماری را انتخاب کند که فکر می‌کند قادر است از عهده این عوامل محدودکننده برآید. اگر طراح معماری غلطی را انتخاب کند در این صورت کد VHDL باید دوباره نوشته شود.

برحسب اینکه چه نوع از عوامل محدودکننده سیگنال‌های خفته و تسلسل زمانی داده‌ها در ابزار سنتز گمارده شده باشد، معماری‌های مختلف و نتایج مختلف به دست خواهد آمد.

کد VHDL فوق می‌تواند با یک ابزار سنتز معمولی در سطح RT سنتز گردد. در این حالت نتیجه دقیقاً مطابق رفتار سیگنال به سیگنال تعریف شده در کد VHDL است. به عبارت دیگر هیچ نوع عوامل محدودکننده سیگنال‌های خفته یا تسلسل زمانی داده‌ها را نمی‌توان در ابزار سنتز مشخص و ذکر کرد.

چنان‌چه طراح بخواهد کار خود را در سطح RT انجام دهد، کد VHDL می‌تواند به نحوی که در شکل زیر نشان داده شده است نوشته شود. به منظور ایجاد هماهنگی بین فرکانس کلاک ۴۰ نانوثانیه با یک حاشیه اطمینان مناسب در تکنولوژی انتخاب شده (Texas Instruments, TGC 2000, 0.65 µm) طرح باید به صورت خط لوله‌ای یعنی ذخیره‌سازی سیگنال‌های b0-b3 در بانکهای رجیستری اجرا شود. به همین دلیل بین خطوط آدرس نیز مکانیسم خط لوله‌ای اعمال شده است.

مثال (سطح RT) :

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Package types is
   Constant L_in:  positive:=12;                -- Number of bits in input data.
   Constant L_mult: positive:=8;                -- Number of bit in mult.
   Constant L_ch:  positive:=14;                -- Number of channels.
   Constant L_reg:  positive:=L_IN+L_mult;      -- Number of bits in reg.
                                                -- after mult.
   Constant L_reg:  positive:=L_IN+L_mult;      -- Number of bits in output.
```

```
   subtype data is std_logic_vector(L_data-1 downto 0);     -- data out
   subtype reg is std_logic_vector(L_reg-1 downto 0);       -- reg after mult
   subtype reg_in is std_logic_vector(L_in-1 downto 0);     -- input reg
end;

Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.types.ALL;

Entity fir_rtl is
port (d_in:             in reg_in;
      c0,c1,c2,c3:      in std_logic_vector(L_mult-1  downto 0);
      clk,resent,en:    in std_logic;
      q:                out data);
end;

Architecture rtl of fir_rtl is
signal d1,d2,d3:  reg_in;
signal b0,b1,b2,b3: reg;
signal d_out_i: data;
signal i1,i2: std_logic_vector(L_reg downto 0);
begin
  p1: process(clk,resetn)
  begin
      if resetn='0' then
            d1<=(others=>'0');
            d2<=(others=>'0');
            d3<=(others=>'0');
            b0<=(others=>'0');
            b1<=(others=>'0');
            b2<=(others=>'0');
            b3<=(others=>'0');
            q<=(others=>'0');
      elsif clk'event and clk='1' then
        if en='1' then
            b0<=d_in * c0;      -- Register bank
            b1<=d1 * c1;        -- Register bank
            b2<=d2 * c2;        -- Register bank
```

```
            b3<=d3 * c3;          -- Register bank
            d3<=d2;               -- Shift data
            d2<=d1;
            d1<=d_in;
        end if;
        i1<=('0' & b0) + b1;      -- Register bank
        i2<=('0' & b2) + b3;      -- Register bank
        q<=('0' & i1) + i2;
      end if;
  end process;
end;
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.types.ALL;

Entity fir_4_rtl is
port (d_in1,d_in2:    in    reg_in;
      d_in3,d_in4:    in    reg_in;
      c0,c1,c2,c3:    in    std_logic_vector(L_mult-1 downto 0);
      cl,resetn,en:   in    std_logic;
      q1,q2,q3,q4:    out   data);
end;

Architecture rtl of fir_4_rtl is
  component fir_rtl
  port( d_in:           in    reg_in;
      c0,c1,c2,c3:      in    std_logic_vector(L_mult-1 downto 0);
      ckl,resetn,en:    in    std_logic;
      q:                out   data);
  end component;
  For ALL:fir_rtl Use entity work.fir_rtl(rtl);
begin
      CH1:fir_rtl port map (d_in1,c0, c1, c2, c3,clk,resetn,en,q1);
      CH2:fir_rtl port map (d_in2,c0, c1, c2, c3,clk,resetn,en,q2);
      CH3:fir_rtl port map (d_in3,c0, c1, c2, c3,clk,resetn,en,q3);
      CH4:fir_rtl port map (d_in4,c0, c1, c2, c3,clk,resetn,en,q4);
end;
```

چنان‌چه مثالهای بالا را برای فیلتر FIR سنتز کنیم، نتایج زیر به دست می‌آیند. یک نتیجه سنتز برای سطح RT و دو نتیجه برای سطح رفتاری نشان داده شده است.

RTL : در این سنتز، کد VHDL بدون اشتراک منابع برای چهار فیلتر FIR فراخوانی شده است.

Behavioural 1 : در این سنتز، کد VHDL رفتاری به‌کار رفته است. عوامل محدودکننده طوری تنظیم شده‌اند که تعداد منابع را به حداقل برسانند.

Behavioural 2 : در این سنتز، کد VHDL رفتاری به‌کار رفته است. عوامل محدودکننده طوری تنظیم شده‌اند که سیکل‌های خفته را در طرح کاهش دهند.

ابزار سنتز	تسلسل زمانی داده‌ها (تعداد سیکل‌های ساعت)	سیکل‌های خفته (تعداد سیکل‌های ساعت)	فضا (تعداد گیت‌ها)	منابع (تعداد جمعگر/تعداد ضرب‌کننده)
RLT	۱	۳	۲۲۰۰۰	۱۶/۱۲
Behavioural 1	۸	۴ × ۲ = ۸	۶۰۰۰	۲/۲
Behavioural 2	۴۰	۴ × ۱۰ = ۴۰	۵۰۰۰	۱/۲

جدول ۲-۱۷ تعداد منابع

در زیر، بخشهایی از گزارش اولین سنتز رفتاری (کنترل فضا) آورده شده است. سه بخش گوناگون از گزارش برای پروسس P1 به نمایش درآمده که عبارتند از : چکیده، زمان‌بندی عملیات و میزان مصرف رجیسترها.

```
**************************************
*  Summary report for process p1:    *
**************************************
-------------------------------------------------------------------
    Timing Summary
-------------------------------------------------------------------
Clock period 40.00
 Loop timing information:
  p1                          10 cycles (cycles 0 - 10)
    res_loop                  10 cycles (cycles 0 - 10)
      main                    10 cycles (cycles 0 - 10)
        _L0                   1 cycle  (cycles 2 - 3)
          (exit) EXIT_L64     (cycle 3)
-------------------------------------------------------------------
    Area Summary
-------------------------------------------------------------------
 Estimated combinational area        2251
 Estimated sequential area           3440
TOTAL                                5691

 11 control states
 12 basic transitions
 2 control inputs
 15 control outputs
```

```
--------------------------------------------------------------------
Resource types
--------------------------------------------------------------------
  Register Types
  ==============
    1-bit  register                1
    2-bit  register                1
    8-bit  register                2
    12-bit  register               2
    20-bit  register               1
    21-bit  register               2
    48-bit  register               3

  Operator Types
  ==============
    (2_2->2)-bit DW01_add          1    (Channel counter)
    (12_8->20)-bit DW02_mult       1
    (21_21->21)-bit DW01_add       1
    (22_22->22)-bit DW01_addsub    1

  I/O Ports
  =========
    1-bit  input port              1
    1-bit  registered  output port 1
    8-bit  input port              4
    12-bit  input port             1
    88-bit  output port            1

--------------------------------------------------------------------
```

همان طور که گزارش چکیده فوق نشان می‌دهد و با توجه به زمان‌بندی، حلقه اصلی در برنامه ۱۰ سیکل ساعت را مصرف می‌کند. از آنجا که طرح از چهار کانال تشکیل یافته است باید حلقه اصلی برای پردازش داده‌ها در تمام کانالها چهار بار اجرا شود. بدین ترتیب سیکل‌های خفته ۱۰ × ۴ یعنی ۴۰ سیکل خواهد بود. در نتیجه، میزان خفتگی سیکل‌ها نسبتاً بالا خواهد بود زیرا عامل محدودکننده در ابزار سنتز رفتاری فقط برای به حداقل رساندن فضا (حجم) و بدون توجه به هزینه‌ها، طرح‌ریزی شده است.

ابزار سنتز رفتاری بعد از تمام شدن زمان‌بندی طرح، حجم آن را تخمین می‌زند. این تخمین فضا (۵۶۹۱ گیت) قبل از بهینه‌سازی در سطح RT توسط ابزار سنتز RTL انجام می‌شود. چنان‌چه ابزار سنتز RTL بهینه‌سازی طرح را با ایجاد بلوک‌های سلسله مراتبی انجام دهد، تخمین حجم معمولاً بسیار دقیق خواهد بود. ابزار سنتز رفتاری (Synopsys) باعث ایجاد بلوک‌های سلسله مراتبی برای ماشین کنترل، اپراتورها و بانکهای رجیستری (غالباً برای ایجاد شماتیک خواناتر) می‌گردد. بعد از بهینه‌سازی، حجم طرح به ۵۰۰۰ گیت می‌رسد (به جدول ۲-۱۷ مراجعه کنید). این مقدار بیش از چهار برابر کمتر از نتیجه سنتز برای کد VHDL در سطح RT است. البته از سوی دیگر سیکل‌های خفته و تسلسل زمانی داده‌ها در این سنتز افزایش قابل توجهی می‌یابند. این افزایش با عوامل محدودکننده انتخابی برای سنتز رفتاری که فقط شامل محدودیتهای مکانی است کاملاً مطابقت دارد. وقتی در محدودیتهای سنتز رفتاری تعادل بیشتری برقرار کردیم تسلسل زمانی داده‌ها و سیکل‌های

خفته بسیار کمتر شد (Behavioural 1). از لحاظ حجم نیز افزایش زیادی به وجود نیامد (فقط در حدود ۱۰۰۰ گیت) در حالی که سیکل‌های خفته و تسلسل زمانی داده‌ها از ۴۰ به ۸ سیکل کاهش‌یافت.

گزارش بالا همچنین نشان می‌دهد که ماشین کنترل که مشترک‌سازی منابع را کنترل می‌کند از ۱۱ حالت تشکیل یافته است. همچنین، امکان خواندن تعداد اپراتورها از گزارش فوق وجود دارد. همان طور که مشاهده می‌شود در این طرح بر طبق انتظار، یک ضرب‌کننده وجود دارد، اما در کمال تعجب دیده می‌شود که دو جمعگر در این طرح حضور پیدا کرده‌اند. علت آن این است که ابزار سنتز محاسبه کرده است که مشترک‌سازی این دو جمعگر از نقطه نظر حجم ارزش این کار را ندارد. مشترک‌سازی منابع نیاز به یک ماشین کنترل بزرگ‌تر و چندین مالتی‌پلکسر (۲۱ عدد) در مسیر داده‌ها نیاز دارد تا بتواند مقادیر صحیح داده‌های ورودی به جمعگر را تعیین کند. هر مالتیپلکسر به سه گیت نیاز دارد. به علاوه، یک بانک رجیستری اضافی نیز برای ذخیره مقادیر بعدی که باید محاسبه شوند موردنیاز است. چنان‌چه فرض کنیم یک رجیستر با reset آسنکرون به هشت گیت نیاز داشته باشد، این امر به بهای مکانی ۳ × ۲۱ + ۸ × ۲۱ به اضافه یک ماشین کنترل بزرگ‌تر و در کل حداقل ۲۳۱ گیت تمام خواهد شد. این بدان معناست که در این مورد دو جمعگر حاضر مشترک سازی نشده‌اند. از آنجا که اگر هیچ گونه مشترک‌سازی انجام نشود به ۱۲ جمعگر نیاز خواهیم داشت، دو جمعگر حاصل شده موجب کاهش فضای مصرفی به ۵۰۰۰ گیت خواهند شد.

```
****************************************
*  Operation schedule of process p1:  *
****************************************

    Resource  types
=============================
c0          8-bit  input  port
c1          8-bit  input  port
c2          8-bit  input  port
c3          8-bit  input  port
d_in        12-bit  input  port
done        1-bit  registered  output  port
en          1-bit  input  port
loop        loop  boundaries
p0          88-bit  output  port  d_out
r46         (12_8->20)-bit  DW02_mult
r52         (22_22->22)-bit  DW01_addsub
r57         (21_21->21)-bit  DW01_add
r411        (2_2->2)-bit  DW01_add
```

cycle	loop	d_in	c1	c2	c3	c0	p0	en	r46	r52	r57	r411	done
0	L6						W1						W0
	L3												
	L0												
1													W4
2	L9							R6					
3	L12	R0	R1	R2	R3				o5				
	L11												
	L10												
4						R4			o7				
5									o3				
6									o4		o9		W3
7											o8		
8										o6			
9							W2			o6			
							R5						
10	L8											o10	
	L7												
	L5												
	L4												
	L2												
	L1												

Operation name abbreviations			
L0	loop	boundaries	p1_design_loop_begin
L1	loop	boundaries	p1_design_loop_end
L2	loop	boundaries	p1_design_loop_cont
L3	loop	boundaries	res_loop/res_loop_design_loop_begin
L4	loop	boundaries	res_loop/res_loop_design_loop_end
L5	loop	boundaries	res_loop/res_loop_design_loop_cont
L6	loop	boundaries	res_loop/main/main_design_loop_begin
L7	loop	boundaries	res_loop/main/main_design_loop_end
L8	loop	boundaries	res_loop/main/main_design_loop_cont
L9	loop	boundaries	res_loop/main/_L0/_L0_design_loop_begin
L10	loop	boundaries	res_loop/main/_L0/_L0_design_loop_end
L11	loop	boundaries	res_loop/main/_L0/_L0_design_loop_cont
L12	loop	boundaries	res_loop/main/_L0/EXIT_L64

گزارش فوق چگونگی عملکرد اپراتورها در زمان‌بندی تعیین شده (۱۰ سیکل پالس ساعت) را نشان می‌دهد. به علاوه، از این گزارش می‌توانیم از زمان خوانده شدن یا نوشته شدن درگاههای I/O آگاه شویم. به‌طور مثال : سیگنال خروجی d_out در سیکل ۹ پالس ساعت نوشته می‌شود (P0, W2). همچنین d_out در سیکل 0 پالس ساعت نیز نوشته می‌شود، که به معنای reset شدن خروجی است.

همچنین می‌توان از گزارش فوق تعداد سیکل‌های ساعتی را که اپراتورها مصرف می‌کنند برداشت کرد. به‌طور مثال ضرب‌کننده r64 در سیکل‌های ۳، ۴، ۵ و ۶ مورد استفاده قرار گرفته است. این بدان معنی است که نرخ استفاده از ضرب‌کننده ۴/۱۰ یا ۴۰ درصد بوده است. به عنوان مثال دیگر، از جمعگر ۲۲ بیتی (r52) توسط اپراتور o6 در سیکل‌های ۸ و ۹ استفاده شده است. این بدان معناست که جمعگرها در طول دو سیکل پالس ساعت کار خود را انجام می‌دهند و همچنین به آن معناست که

در این حالت ابزار سنتز رفتاری آنچه که به نام ***محدودیتهای چندسیکلی***[1] خوانده می‌شود را برای این مسیر داده به‌طور خودکار منظور می‌کند (به فصل ۱۰ مراجعه کنید، «سنتز در سطح RT»). برقراری این محدودیتها به‌طور دستی در سطح RT بسیار وقت‌گیر خواهد بود.

بخش اختصاری و مخفف اسامی اپراتورها در گزارش، نام منابع را به‌طور غیرمستقیم توضیح می‌دهد. این مخففها از قسمتهای گوناگون تشکیل شده‌اند. به‌طور مثال 1_bit write res_loop/main/done_86 به خط مشخصی در کد VHDL اصلی مربوط است.

```
**************************************
*  Register usage of process p1:  *
**************************************

   r21        48-bit  register
   r136       1-bit   register
   r145       2-bit   register
   r292       21-bit  register
   r319       12-bit  register
   r331       12-bit  register
   r429       8-bit   register
   r430       21-bit  register
   r434       8-bit   register
   r444       48-bit  register
   r445       20-bit  register
   r446       48-bit  register

cycle| r444 | r21  | r446 | r292 | r430 | r445 | r331 | r319 | r429 | r434 | r145 |r136
-------------------------------------------------------------------------------
 (48)| (48) | (48) | (21) | (21) | (20) | (12) | (12) | (8)  | (8)  | (2)  | (1)
===============================================================================
  0  |. v0..|..v1..|..v2..|......|......|......|......|.v12..|......|......|.v17..|...
  1  |..v0..|..v1..|..v2..|......|......|......|......|.v12..|......|......|.v17..|...
  2  |..v0..|..v1..|..v2..|.... .|......|......|.v14..|.v12..|......|......|.v17..|v19
  3  |..v3..|..v1..|..v2..|..v9..|......|......|.v11..|.v12..|.v15..|.v16..|.v17..|...
  4  |..v3..|..v1..|..v2..|..v9..|......|.v10..|.v11..|.v12..|.v15..|.v16..|.v17..|...
  5  |..v3..|..v1..|..v2..|..v9..|..v7..|.v10..|.v11..|.v13..|.v15..|......|.v17..|...
  6  |..v3..|..v1..|..v2..|..v8..|..v6..|.v10..|.v11..|......|......|......|.v17..|...
  7  |..v3..|..v4..|..v2..|..v5..|..v6..|......|......|......|......|......|.v17..|...
  8  |..v3..|..v4..|..v2..|..v5..|..v6..|......|......|......|......|......|.v17..|...
  9  |..v3..|..v4..|..v2..|......|......|......|......|......|......|......|.v17..|...
 10  |......|......|......|......|......|......|......|......|......|......|......|...
```

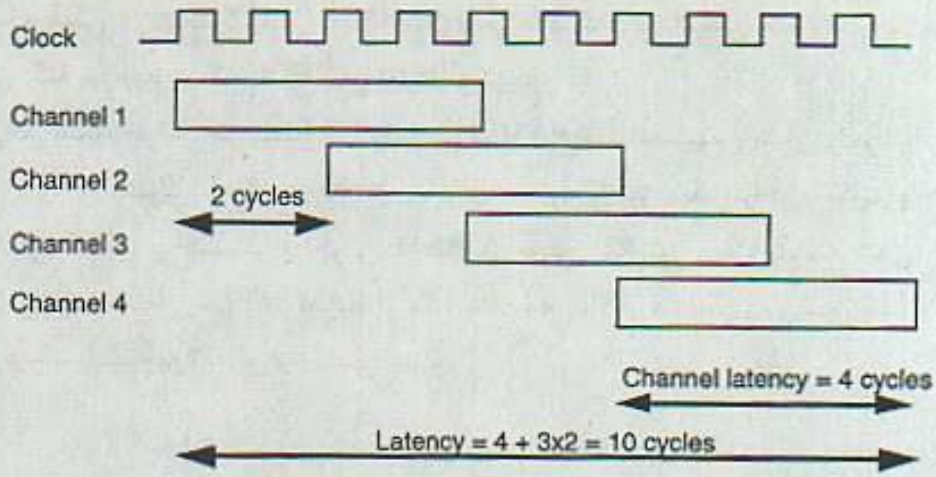
1- Multicycle Constraint

Data	value	name	abbreviations
v0	48-bit	data	value res_loop/main/d2
v1	48-bit	data	value res_loop/main/d1
v2	48-bit	data	value res_loop/main/U1/net
v3	48-bit	data	value res_loop/main/U2/net
v4	48-bit	data	value res_loop/main/U3/net
v5	21-bit	data	value res_loop/main/add_78/plus/plus/Z
v6	21-bit	data	value res_loop/main/add_79/plus/plus/Z
v7	20-bit	data	value res_loop/main/mul_73/mult/mult/Z
v8	20-bit	data	value res_loop/main/mul_75/mult/mult/Z
v9	20-bit	data	value res_loop/main/mul_74/mult/mult/Z
v10	20-bit	data	value res_loop/main/mul_76/mult/mult/Z
v11	12-bit	data	value res_loop/main/d_in_72/net
v12	12-bit	data	value res_loop/main/mul_73/mult/mult/A
v13	12-bit	data	value res_loop/main/mul_75/mult/mult/A
v14	12-bit	data	value res_loop/main/mul_74/mult/mult/A
v15	8-bit	data	value res_loop/main/c1_75/net
v16	8-bit	data	value res_loop/main/c3_73/net
v17	2-bit	data	value res_loop/main/ch
v19	1-bit	data	value res_loop/main/_L0/U2/Z

گزارش فوق میزان مصرف رجیسترها را بعد از سنتز رفتاری پروسس P1 نشان می‌دهد. همان طور که از گزارش برداشت می‌شود داده‌های گوناگون در یک منبع رجیستری واحد ذخیره شده‌اند. این بدان معناست که ابزار سنتز رفتاری هر وقت امکانش باشد و از لحاظ مکانی (حجمی) ارزش آن را داشته باشد، بانکهای رجیستری را به‌طور مشترک مورد استفاده قرار می‌دهد. همچنین می‌توان از گزارش فهمید که رجیسترهای خط لوله‌ای بعد از ضرب‌کننده‌ها (۲۰ بیتی) و جمعگرها (۲۱ بیتی) به‌طور مشترک مورد استفاده قرار گرفته‌اند. اگر اشتراک منابع انجام نمی‌شد باید چهار رجیستر ۲۱ بیتی و دو رجیستر ۲۰ بیتی برای هر کانال به منظور ذخیره داده‌ها مورد استفاده قرار می‌گرفت. گزارش نشان می‌دهد که فقط از دو رجیستر ۲۱ بیتی و یک رجیستر ۲۰ بیتی استفاده شده است و معنای آن این است که منابع مشترک‌سازی شده‌اند.

در سنتز رفتاری Behavioural 1 ، سیکل‌های خفته به اندازه دو سیکل ساعت برای هر کانال و در کل هشت سیکل ساعت به‌دست آمد. اگر مثال بیان شده بزرگ‌تر بود، سیکل‌های خفته می‌توانستند توسط آنچه که ایجاد ***خط لوله حلقوی***[1] خوانده می‌شود، کاهش یابند. منظور از ایجاد خط لوله حلقوی شروع تکرار بعدی قبل از کامل شدن تکرار قبلی است (شکل ۷-۱۷).

1- Loop Pipelining



شکل ۷-۱۷ خط لوله حلقوی

فرض کنید که در مثال فوق حداقل ممکن سیکل‌های خفته برای یک تکرار حلقوی چهار سیکل پالس ساعت باشد. با ایجاد خط لوله حلقوی، تکرار بعدی حلقه (برای کانال بعدی) می‌تواند بعد از دو سیکل ساعت شروع شود. میزان سیکل‌های خفته بدون ایجاد خط لوله حلقوی بالغ بر ۱۶ = ۴ × ۴ سیکل پالس ساعت می‌شود، ولی با ایجاد خط لوله حلقوی سیکل‌های خفته ۱۰ = ۲ × ۴ + ۴ سیکل پالس ساعت می‌شود (شکل ۷-۱۷). در طراحی‌های بزرگ‌تر، ایجاد خط لوله حلقوی می‌تواند میزان سیکل‌های خفته در طرح را کاهش دهد. البته این امر به قیمت مصرف فضای بیشتر تحقق می‌یابد. یک ابزار سنتز رفتاری می‌تواند تولید خط لوله حلقوی را به‌طور اتوماتیک انجام دهد.

در نتیجه سنتز فوق از فلیپ فلاپ‌های معمولی به منظور پیاده‌سازی بانکهای رجیستری استفاده شده است. اگر بخواهیم می‌توانیم از طریق ابزار سنتز یک بلوک RAM را به جای فلیپ فلاپ‌ها به‌طور اتوماتیک وارد کنیم. در این صورت ابزار سنتز رفتاری یک واحد کنترلی به منظور تولید سیگنال‌های read ، write و enable strobe برای RAM به وجود می‌آورد. برای آنکه ابزار سنتز رفتاری (Synopsys Behavioural Compiler) بتواند RAM را به جای فلیپ فلاپ‌ها وارد کند یک ویژگی خاصی باید در کد VHDL گنجانده شود. اگر فرضاً بنا باشد متغیرهای d1 ، d2 و d3 به جای بانکهای رجیستری در RAM قرار داده شوند نشان زیر باید در کد VHDL منظور شود :

```
constant my_RAM : resource := 0;
attribute variable of my_RAM : constant is "d1 d2 d3";
attribute map to module of my_RAM : cosntant is <name_of_RAM>;
```

نسخه فعلی Synopsys یعنی v3.3b ، لازم می‌دارد که رنج اندیس متغیرها روی هم نیفتد، بنابراین این نکته نیز باید در کد VHDL در نظر قرار گیرد.

اگر در هنگام طراحی در سطح RT بخواهیم از RAM استفاده کنیم باید آن را فراخوانی کرده و علاوه بر آن، واحد کنترلی که تولید strobe را برعهده دارد را نیز باید طراحی نماییم و همه اینها کارهای بسیار وقت‌گیری هستند. فراخوانی RAM در سطح RT نیز باعث وابسته شدن RAM به تکنولوژی می‌شود، مانند زمانی که RAM از یک کتابخانه ASIC فراخوانی می‌شود. کد VHDL در سطح رفتاری غیروابسته به تکنولوژی خاصی است.

Appendix A
VHDL syntax

A.1. Library units

Architecture body

Library unit

Several architectures can be linked to an entity.

```
Architecture architecture_name of entity_name
architecture_declaration
begin
  architecture_statement
end architecture_name;
```

```
Architecture rtl of adder is
 constant 5v_const: std_logic:='1';
 type state_type is (s0,s1,s2,s3);
 signal state:state_type;
 signal a_int: std_logic_vector(4 downto 0);

 function add_f (a,b:in std_logic_vector) return std_logic_vector is
 begin
   return (('0' & a) + b);
 end;
begin
   a_int<=not a;
   q<=add_f(a_int,b);
end;
```

Configuration declaration

Library unit

```
Configuration configuration_name of entity_name is
end configuration_name;
```

```
Configuration config of ent is
```

```
for rtl
      for all:comp
        use entity work.dff(rtl);
      end for;
end for;
end config;
```

Entity declaration *Library unit*

```
Entity Entity_name is
entity_declaration
begin
 entity_statement
end Entity_name;
```

Only passive processes are permitted in the entity declaration.

```
Entity ent is
 port( clk,resetn,d_in    in  std_logic;
       d_out:             out    std_logic);
end;

Entity ent2 is
    generic(N:positive:=4);
    port(  a,b:   in       std_logic;
           q:     out std_logic);
begin
    assert not (a='1' and b='0')
    report (ERROR a=1 and b=0 at the same time !)
    severity ERROR;
end;
```

Package body *Library unit*

```
Package body my_pack is
  function add_f (a,b: std_logic_vector) return std_logic_vector is
  begin
    return (('0' & a) + b);
  end;
end;
```

Package declaration *Library unit*

```
Package package_name is
end package_name;

Package my_pack is
  function add_f (a,b: std_logic_vector) return std_logic_vector;
  type my_array is array 4 downto 0 of std_logic_vector(3 downto 0);
```

```
  constant my_5v:std_logic_vector(7 downto 0):=(others=>'1');
end;
```

A.2. Declarations

Alias declaration *Declaration*

```
signal v:std_logic_vector(7 downto 0);
alias vect1 : std_logic_vector(8 downto 1) is v;
```

Attribute declaration *Declaration*

Attribute *attribute_name: type;*

Attribute version: version_number;

Component declaration *Declaration*

```
component component_name
  generic_clause
  port_clause
end component;

component my_comp
  generic (N:positive);
  port(a,b:in std_logic;
       q:out std_logic);
end component;
```

Constant declaration *Declaration*

```
constant my_5v:std_logic_vector(7 downto 0):=(others=>'1');
constant s0:std_logic_vector(1 downto 0):="01";
constant max_delay:time:=12 ns;
```

File declaration *Declaration*

file *file_name* : *file_type* is *mode file_logiskt_name;*

file prom: prom_file_type is in "prom_file.txt";

Signal declaration *Declaration*

It is not allowed to declare signals inside a process, subprogram or in the package body.

signal *signal_name*:type;

```
signal a,b: std_logic;
signal a_bus,b_bus: std_logic_vector(7 downto 0);
signal ab_bus: bit_vector(7 downto 0);
signal viewL_bus: vlbit_vector(7 downto 0);
```

Subprogram body *Declaration*

```
function add_f (a,b: std_logic_vector) return std_logic_vector is
begin
 return (('0' & a) + b;
end;

function reg (signal clk,resetn,d_in:in std_logic) return std_logic is
begin
  if clk'event and clk='1' then
    if resetn='0' then
      return '0';
    else
      return d_in;
    end if;
  end if;
end;

procedure add_p(a,b:in std_logic_vector; sum:out std_logic_vector) is
begin
  sum<=('0' & a) + b;
end;
```

Subprogram declaration *Declaration*

```
function add_f (a,b:in std_logic_vector) return std_logic_vector;
function reg (signal clk,resetn,d_in: std_logic) return std_logic;

procedure add_p(a,b:in std_logic_vector;
                sum:out std_logic_vector);
```

Subtype declaration *Declaration*

```
subtype my_int is integer range 0 to 63;
subtype byte is std_logic_vector(7 downto 0);
```

Type declaration *Declaration*

```
type colour is (red,green,blue);
type state_type is (s0,s1,s2,s3);
```

```
type array_4 is array (3 downto 0) of std_logic_vector(4 downto 0);
type bit is ('0', '1');
type my_bit is ('X', 'Z', '0', '1');
```

Variable declaration *Declaration*

```
variable a:std_logic;
variable a_bus:std_logic_vector(7 downto 0):=(others=>'0');
variable my_int is integer range 0 to 255;
```

A.3. Sequential statements

The sequential VHDL statements can be used in the sequential part of VHDL, i.e. processes, functions and procedures. Note that some statements can be used in both the sequential and the concurrent part of VHDL. Such statements are included in both this section and the section for concurrent statements.

Assert statement *Sequential*

```
Assert condition
 report expression
 severity severity_level

Assert a='1'
 report "a/='1'"
 severity note;

Assert data/=0
 report "data=0"
 severity warning;

Assert boolean_sign
  report "boolean=false"
  severity error;

Assert state/=s4
 report "illegal state (s4)"
 severity failure;
```

Case statement *Sequential*

```
case expression is
  case_alternative
end case;

process(a,b)
variable a_int:integer range 0 to 12;
begin
```

```
 a_int<=a+b;
 case a_int is
    when 0          => en<="001";
    when 2 to 5  => en<="010";
    when 7 | 9     => en<="100";
    when others => en<="000";
  end case;
end process;
```

Exit statement *Sequential*

exit *loop_label* when *condition;*

```
loop
 exit when b=3;
 a<=b-c;
end loop;

l1:for i in 0 to 3 loop
 l2:for j in 0 to 5 loop
     a:=b+1;
     exit l2 when a>12;
 end loop l2;
end loop l1;

loop
 if a>=c-d then
    exit;
 else
    c:=c+1;
end loop;
```

If statement *Sequential*

```
process(clk,resetn)
begin
 if resetn='0' then
    data<=(others=>'0');
 elsif clk'event and clk='1' then
    data<=data_in;
 end if;
end process;

function my_f (a,b:in std_logic_vector) return std_logic_vector is
begin
  if a-b>12 then
     return "00";
  elsif a-b=12 then
     return "01";
  elsif a=12 then
```

```
    return "11";
  else
    if a<2 then
      return "00";
    else
      return "10";
    end if;
  end if;
end;
```

Loop statement *Sequential*

```
loop
 exit when b=3;
 a<=b-c;
end loop;

for i in 3 downto 0 loop
  q(i)<=a(i)-b;
end loop;

while a<12 loop
 a:=a+1;
 data(a)<=c(a+1);
end loop;
```

Next statement *Sequential*

The next statement interrupts the current loop and jumps straight back to the next loop iteration.

next *loop_label* when *condition*;

```
while a<12 loop
  next when a=4
  sum:=a-b;
  a:=a+cin(a);
end loop;
```

Null statement *Sequential*

```
process(a)
begin
  data<=(others=>'1');
  case a is
    when 2 | 4   => data<=d1;
    when 5 to 7  => data<=d2;
    when others => null;
  end case;
end process;
```

Return statement *Sequential*

```
function my_f (a,b:in std_logic_vector(2 downto 0) return std_logic_vector is
begin
  if a-b>12 then
    return "00";
  elsif a-b=12 then
    return "01";
  elsif a=12 then
    return "11";
  else
    if a<2 then
      return "00";
    else
      return "10";
    end if;
  end if;
end;
```

Signal assignment *Sequential*

```
process(a,b,c)
begin
  q1<=a and b after 10 ns;
  q2<=a & b(3 downto 0);
  q3<=c after Tdelay, b after Tdelay + 4 ns;
  q4<=my_func(a,b);
end process;
```

Variable assignment *Sequential*

```
process(a,b,c)
begin
  q1:=a and b after 10 ns;
  q2:=a & b(3 downto 0);
  q3:=c after Tdelay, b after Tdelay + 4 ns;
  q4:=my_func(a,b);
end process;
```

Wait statement *Sequential*

```
wait on a;

wait until clk='1';

Wait for 10 ns;

wait on a until b='1' for 10 ns;
```

Generate statement *Concurrent*

```
l1: for i in 0 to 3 generate
    U1: comp1 port map (clk,resetn,din(i),q(i+1))
end generate;
```

Process statement *Concurrent*

```
process_label : process(sensitivity_list)
process_declaration
begin

end process process_label;

Process(a,b,c)
variable int:std_logic;
begin
  int:=a and b;
   q<=int nor c;
end process;

process(clk,resetn)
begin
  if resetn='0' then
    d_out<='0';
  elsif clk'event and clk='1' then
    d_out<=d_in;'
  end if;
end process;

p1:process
begin
  wait until clk='1';
   a<=not b;
end process;
```

Signal assignment *Concurrent*

```
q1<=a and b after 10 ns;
q2<=a & b(3 downto 0);
q3<=c after Tdelay, b after Tdelay + 4 ns;
q4<=my_func(a,b);
q5<=5 after 2 ns, 3 after 4 ns, 9 after 10 ns;
q6<=guarded d_out after 12 ns;
```

When else statement *Concurrent*

```
q<=a when "00" else
    b when "01" else
    c when "10" else
    d;
```

With select statement *Concurrent*

```
with sel select
  q<=a when "00",
      b when "01",
      c when "10",
      d when others;
```

Appendix B

VHDL-package

B.1. Std-package

```
-- This is Package STANDARD as defined in the VHDL 1987 Language
-- Reference Manual.

package standard is

-- predefined enumeration types:
type boolean is (FALSE, TRUE);
type bit is ('0', '1');

type character is (
      NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
      BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
      DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
      CAN,  EM,   SUB,  ESC,  FSP,  GSP,  RSP,  USP,
       ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
       '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
       '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
       '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
       '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
       'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
       'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
       'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
       '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
       'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
       'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
       'x',  'y',  'z',  '{',  '|',  '}',  '~',  DEL);
type severity_level is (NOTE, WARNING, ERROR, FAILURE);

-- predefined numeric types:
type integer is range implementation_defined
type real is range implementation_defined

--predefined type time
```

```
type time is range implementation_defined
        units
                fs;                             -- femtosecond
                ps      =       1000 fs;        -- picosecond
                ns      =       1000 ps         -- nanosecond
                us      =       1000 ns         -- microsecond
                ms      =       1000 us         -- millisecond
                sec     =       1000 ms         -- second
                min     =       60 sec          -- minute
                hr      =       60 min          -- hour
        end units;

-- function that returns the current simulation time:
function now return time;

-- predefined numeric subtypes:
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;

-- predefined array types:
type string is array (positive range <>) of character;
type bit_vector is array (natural range <>) of bit;

end standard;
```

B.2. IEEE-package

B.2.1. Std_logic_1164

```
-- ------------------------------------------------------------------
-- Title        : std_logic_1164 multi-value logic system
-- Library      : This package shall be compiled into a library
--              : symbolically named IEEE.
-- Developers:  IEEE model standards group (par 1164)
-- Purpose      : This packages defines a standard for designers
--              : to use in describing the interconnection data types
--              : used in vhdl modeling.
--
-- Limitation   : The logic system defined in this package may
--              : be insufficient for modeling switched transistors,
--              : since such a requirement is out of the scope of this
--              : effort. Furthermore, mathematics, primitives,
--              : timing standards, etc. are considered orthogonal
--              : issues as it relates to this package and are therefore
--              : beyond the scope of this effort.
--
-- Note         : No declarations or definitions shall be included in,
--              : or excluded from this package. The "package declaration"
--              : defines the types, subtypes and declarations of
--              : std_logic_1164. The std_logic_1164 package body shall be
--              : considered the formal definition of the semantics of
--              : this package. Tool developers may choose to implement
--              : the package body in the most efficient manner available
```

```
--              : to them.
--
-- ------------------------------------------------------------------

PACKAGE std_logic_1164 IS
  ---------------------------------------------------------------------
  -- logic state system  (unresolved)
  ---------------------------------------------------------------------
  TYPE std_ulogic IS (    'U',    -- Uninitialized
                          'X',    -- Forcing  Unknown
                          '0',    -- Forcing 0
                          '1',    -- Forcing 1
                          'Z',    -- High Impedance
                          'W',    -- Weak Unknown
                          'L',    -- Weak 0
                          'H',    -- Weak 1
                          '-'     -- Don't care
                      );
  ---------------------------------------------------------------------
  -- unconstrained array of std_ulogic for use with the resolution function
  ---------------------------------------------------------------------
  TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF
  std_ulogic;

  ---------------------------------------------------------------------
  -- resolution function
  ---------------------------------------------------------------------
  FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;
  ---------------------------------------------------------------------
  -- *** industry standard logic type ***
  ---------------------------------------------------------------------
  SUBTYPE std_logic IS resolved std_ulogic;
  ---------------------------------------------------------------------
  -- unconstrained array of std_logic for use in declaring signal arrays
  ---------------------------------------------------------------------
  TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
  ---------------------------------------------------------------------
  -- common subtypes
  ---------------------------------------------------------------------
  SUBTYPE X01     IS resolved std_ulogic RANGE 'X' TO '1';   -- ('X','0','1')
  SUBTYPE X01Z    IS resolved std_ulogic RANGE 'X' TO 'Z';
    -- ('X','0','1','Z')
  SUBTYPE UX01    IS resolved std_ulogic RANGE 'U' TO '1';
    -- ('U','X','0','1')
  SUBTYPE UX01Z   IS resolved std_ulogic RANGE 'U' TO 'Z';
      --   ('U','X','0','1','Z')
  ---------------------------------------------------------------------
  -- overloaded logical operators
  ---------------------------------------------------------------------
  FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
```

```
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
--  function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    function xnor   ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

    -------------------------------------------------------------
    -- vectorized overloaded logical operators
    -------------------------------------------------------------
    FUNCTION "and"  ( l, r : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "nand" ( l, r : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "or"   ( l, r : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "nor"  ( l, r : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

    FUNCTION "xor"  ( l, r : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;

--  -------------------------------------------------------------
--  Note : The declaration and implementation of the "xnor" function is
--  specifically commented until at which time the VHDL language has been
--  officially adopted as containing such a function. At such a point,
--  the following comments may be removed along with this notice without
--  further "official" balloting of this std_logic_1164 package. It is
--  the intent of this effort to provide such a function once it becomes
--  available in the VHDL standard.
--  -------------------------------------------------------------
--  function "xnor" ( l, r : std_logic_vector ) return std_logic_vector;
--  function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;
    function xnor   ( l, r : std_logic_vector ) return std_logic_vector;
    function xnor   ( l, r : std_ulogic_vector ) return std_ulogic_vector;

    FUNCTION "not"  ( l : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;
    -------------------------------------------------------------
    -- conversion functions
    -------------------------------------------------------------
    FUNCTION To_bit      ( s : std_ulogic
                         ; xmap : BIT := '0'
                         ) RETURN BIT;

    FUNCTION To_bitvector ( s : std_logic_vector
                         ; xmap : BIT := '0'
                         ) RETURN BIT_VECTOR;

    FUNCTION To_bitvector ( s : std_ulogic_vector
                         ; xmap : BIT := '0'
                         ) RETURN BIT_VECTOR;

    FUNCTION To_StdULogic     ( b : BIT           ) RETURN std_ulogic;
```

```
    FUNCTION To_StdLogicVector ( b : BIT_VECTOR       ) RETURN
        std_logic_vector;
    FUNCTION To_StdLogicVector ( s : std_ulogic_vector ) RETURN std_logic_vector;
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR      ) RETURN
        std_ulogic_vector;
    FUNCTION To_StdULogicVector ( s : std_logic_vector ) RETURN
        std_ulogic_vector;
    ----------------------------------------------------------------------
    -- strength strippers and type convertors
    ----------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION To_X01  ( s : std_ulogic       ) RETURN  X01;
    FUNCTION To_X01  ( b : BIT_VECTOR       ) RETURN  std_logic_vector;
    FUNCTION To_X01  ( b : BIT_VECTOR       ) RETURN std_ulogic_vector;
    FUNCTION To_X01  ( b : BIT              ) RETURN  X01;
    FUNCTION To_X01Z ( s : std_logic_vector ) RETURN  std_logic_vector;
    FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION To_X01Z ( s : std_ulogic       ) RETURN  X01Z;
    FUNCTION To_X01Z ( b : BIT_VECTOR       ) RETURN std_logic_vector;
    FUNCTION To_X01Z ( b : BIT_VECTOR       ) RETURN std_ulogic_vector;
    FUNCTION To_X01Z ( b : BIT              ) RETURN  X01Z;
    FUNCTION To_UX01 ( s : std_logic_vector ) RETURN std_logic_vector;
    FUNCTION To_UX01 ( s : std_ulogic_vector ) RETURN std_ulogic_vector;
    FUNCTION To_UX01 ( s : std_ulogic       ) RETURN  UX01;
    FUNCTION To_UX01 ( b : BIT_VECTOR       ) RETURN std_logic_vector;
    FUNCTION To_UX01 ( b : BIT_VECTOR       ) RETURN std_ulogic_vector;
    FUNCTION To_UX01 ( b : BIT              ) RETURN  UX01;
    ----------------------------------------------------------------------
    -- edge detection
    ----------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    ----------------------------------------------------------------------
    -- object contains an unknown
    ----------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_logic_vector ) RETURN  BOOLEAN;
    FUNCTION Is_X ( s : std_ulogic       ) RETURN  BOOLEAN;
END std_logic_1164;
```

B.2.2. Std_logic_unsigned

```
--------------------------------------------------------------------------------
--
-- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.
-- All rights reserved.
-- This source file may be used and distributed without restriction
-- provided that this copyright statement is not removed from the file
-- and that any derivative work contains this copyright notice.
--
--      Package name: STD_LOGIC_UNSIGNED
--
--      Date:         09/11/92      KN
--                    10/08/92      AMT
--
--      Purpose:      A set of unsigned arithemtic, conversion,
--                    and comparision functions for STD_LOGIC_VECTOR.
--
--      Note:         comparision of same length discrete arrays is defined
--                    by the LRM. This package will "overload" those
--                    definitions
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package STD_LOGIC_UNSIGNED is
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
        STD_LOGIC_VECTOR;
```

```
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT:
        STD_LOGIC_VECTOR)     return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT:
        STD_LOGIC_VECTOR)     return STD_LOGIC_VECTOR;
    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return
        INTEGER;
-- remove this since it is already in std_logic_arith
--    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE:    INTEGER)
- return STD_LOGIC_VECTOR;
end STD_LOGIC_UNSIGNED;
```

B.2.3. Std_logic_signed

```
-------------------------------------------------------------------------------
-- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.
--                         All rights reserved.
-- This source file may be used and distributed without restriction
-- provided that this copyright statement is not removed from the file
-- and that any derivative work contains this copyright notice.
--                                                                           --
--      Package name: STD_LOGIC_SIGNED
--      Date:         09/11/91 KN
--                    10/08/92 AMT change std_ulogic to signed std_logic
--                    10/28/92 AMT added signed functions, -, ABS
--
--      Purpose:      A set of signed arithemtic, conversion,
--                    and comparision functions for STD_LOGIC_VECTOR.
--                                                                           --
--      Note:         Comparision of same length std_logic_vector is defined --
--                    in the LRM. The interpretation is for unsigned vectors
--                    This package will "overload" that definition.
-------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package STD_LOGIC_SIGNED is
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return
        STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return
        STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
```

```
    function "-"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "ABS"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        STD_LOGIC_VECTOR;
    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
        BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
        return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHL(ARG:STD_LOGIC_VECTOR;COUNT:
        STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function SHR(ARG:STD_LOGIC_VECTOR;COUNT:
        STD_LOGIC_VECTOR)      return STD_LOGIC_VECTOR;
    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;
-- remove this since it is already in std_logic_arith
--    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
--       return STD_LOGIC_VECTOR;
end STD_LOGIC_SIGNED;
```

Appendix C

Keywords in VHDL-87

abs
access
after
alias
all
and
architecture
array
assert
attribute
begin
block
body
buffer
bus
case
component
configuration
constant
disconnect
downto
else
elsif
end
entity
exit
file
for
function
generate
generic
guarded
if
in
inout
is
label
library
linkage
loop
map
mod
nand
new
next
nor
not
null
of
on
open
or
others
out
package
port
procedure
process
range
record
register
rem
report
return
select
severity
signal
subtype
then
to
transport
type
units
until
use
variable
wait
when
while
with
xor

Appendix D

Additional keywords in VHDL-93

group
impure
inertial
litteral
postponed
pure
reject
rol
ror
shared
sla
sll
sra
srl
unaffected
xnor

فهرست انتشارات مؤسسه فرهنگی هنری دیباگران تهران

کتابهای کامپیوتر

	کـامـپیوتـر	مؤلف/مترجم
۱	مبانی کامپیوتر	مهندس سعید سعادت
۲	آموزش سریع اپراتوری کامپیوتر	مهندس سعید سعادت
۳	اصول ومفاهیم در فناوری اطلاعات	خسرو مهدی پور
۴	فرهنگ تشریحی لغات و اصطلاحات کامپیوتری مایکروسافت	سعید ظریفی
۵	گواهینامه بین المللی کاربری کامپیوتر(ICDL)(سطح ۱و۲)	مهندس مرتضی متواضع
۶	تمرین و آزمون گواهینامه بین المللی کاربری کامپیوتر(ICDL)(سطح ۱و۲)	مهندس علی اکبر متواضع
۷	آموزش ICDL به زبان ساده (در هفت مهارت)	مهندس علی اکبر متواضع
۸	اپراتوری مقدماتی کامپیوتر	فاطمه کبیری فر- جمشیدصفایی فر
۹	خودآموز سیستم عامل MS-DOS (مقدماتی و پیشرفته)	مهندس سعید سعادت
۱۰	مبانی کامپیوتر (کاردانش)	سعید سعادت و افروز کاشف‌الحق
۱۱	NC و ویروسهای کامپیوتری (کاردانش)	واحد تحقیقات و انتشارات
۱۲	الگوریتم و فلوچارت (کاردانش)	واحد تحقیقات و انتشارات
۱۳	Excel 97 (کاردانش)	مهندس علی اکبر متواضع
۱۴	سیستم عامل DOS (کاردانش)	مهندس سعید سعادت
۱۵	NU (کاردانش)	افروز کاشف الحق و سعید سعادت
۱۶	AutoCAD (کاردانش)	مهندس علیرضا پارسای
۱۷	PowerPoint 97 (کاردانش)	جمیله جامی
۱۸	Q Basic (کاردانش)	واحد تحقیقات و انتشارات
۱۹	ویندوز ۹۸ (۱) (کاردانش)	سعید سعادت و افروز کاشف الحق
۲۰	ویندوز ۹۸ (۲) (کاردانش)	سعید سعادت وافروز کاشف الحق
۲۱	زبان تخصصی رایانه ۱ (کاردانش)	خسرو مهدی پور عطایی
۲۲	زبان تخصصی رایانه ۲ (کاردانش)	خسرو مهدی پور عطایی
۲۳	Word 97 (کاردانش)	مهندس مرتضی متواضع
۲۴	نشر رومیزی MS-Word (مقدماتی و پیشرفته)	مهندس فاطمه کبیری فر
۲۵	برنامه نویسی پاسکال مقدماتی (کمک آموزشی)	مهندس مجتبی الله وردی
۲۶	برنامه نویسی پاسکال پیشرفته (کمک آموزشی)	مهندس مجتبی الله وردی
۲۷	برنامه نویسی Q Basic (کمک آموزشی)	مهندس مجتبی الله وردی
۲۸	Word 97 (کمک آموزشی)	فهیمه وفقی
۲۹	Excel 97 (کمک آموزشی)	فهیمه وفقی
۳۰	راهنمای AutoCAD14 (کمک آموزشی)	نلی آقایی
۳۱	ویندوز ۹۸(۱)(کمک آموزشی)	مهندس مجتبی الله وردی
۳۲	ویندوز ۹۸(۲)(کمک آموزشی)	مهندس مجتبی الله وردی
۳۳	NU (کمک آموزشی)	مهندس مجتبی الله وردی
۳۴	PowerPoint 97(کمک آموزشی)	جمیله جامی
۳۵	راهنمای جامع برنامه نویسی پاسکال	مهرداد اسماعیلی
۳۶	همگام با دلفی ۶ (جلد اول و دوم)	مهرداد اسماعیلی
۳۷	گزارش سازی در دلفی ۶	مهرداد اسماعیلی
۳۸	آموزش Installshield برای دلفی ۶	مهرداد اسماعیلی

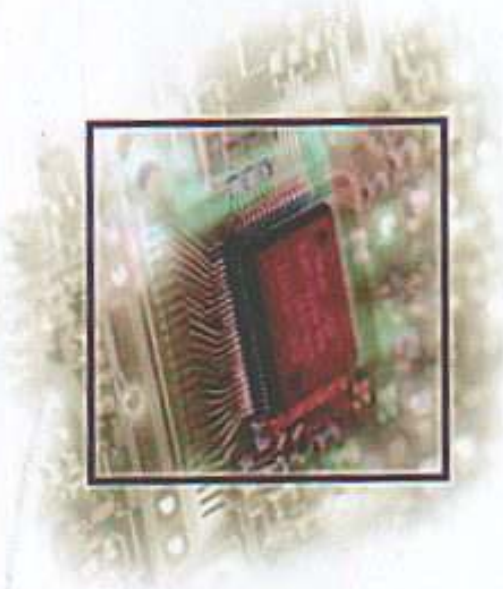
۳۹	پرسشهای چهار گزینه‌ای دلفی ۶	مهرداد اسماعیلی
۴۰	تکنیکهای صوتی و تصویری در دلفی ۶	مهندس محمد عادلی نیا - زینب ستوده‌فر
۴۱	برنامه نویسی دلفی ۷(مقدماتی و پیشرفته)	مهرداد اسماعیلی
۴۲	آموزش سریع Windows 98	مهندس سعید سعادت
۴۳	آموزش گام به گام Windows 98	مهندس افروز کاشف الحق
۴۴	مرجع کامل Windows 98	مهندس علی اکبر متواضع
۴۵	خودآموز تصویری Windows 98 (جلد اول و دوم)	مهندس سیما مجاهد - افشین پزشکی
۴۶	آموزش گام به گام Windows 2000	مهندس سعید سعادت
۴۷	خودآموز سریع Windows 2000 حرفه‌ای	مهندس سهیلا کاویان - خسرو مهدی پور
۴۸	مرجع الفبایی Windows 2000	مسعود پاک نظر
۴۹	ایستگاه کاری Windows NT	علی رحیمی فر - مسعود عسگری
۵۰	خودآموز Windows XP	مهندس سعید سعادت - خسرو مهدی پور
۵۱	خودآموز گام به گام windows XP	اردوان نابینی علی اکبری
۵۲	خودآموز آسان Windows XP	فاطمه کبیری فر - جمشید صفایی فر
۵۳	شبکه سازی خانگی با windows XP	مهندس علی اکبر متواضع
۵۴	عیب یابی در windows XP	مهندس علی اکبر متواضع
۵۵	مرجع الفبایی Windows XP	مهندس سیما مجاهد - افشین پزشکی
۵۶	صوت و تصویر برتر در windows XP	پیمان سمرقندی
۵۷	خودآموز گام به گام Office 2002 XP	مهندس علی اکبر متواضع
۵۸	آموزش گام به گام word 2002 XP	مهندس مرتضی متواضع
۵۹	آموزش گام به گام Excel 2002 XP	مهندس علی اکبر متواضع
۶۰	آموزش گام به گام FrontPage2002 XP	مهندسین امیرحسین رضوی - ملیحه دهقان
۶۱	آموزش گام به گام Access 2002 XP	پدرام پاک گوهر
۶۲	آموزش گام به گام Outlook 2002 XP	خسرو مهدی پور عطایی
۶۳	خودآموز سریع PowerPoint 2002 XP	علی رحیمی فر
۶۴	آموزش گام به گام Office 2000	مهندس مرتضی متواضع
۶۵	آموزش گام به گام word 2000	مهندس مرتضی متواضع
۶۶	آموزش گام به گام Excel 2000	مهندس علیرضا پارسای
۶۷	آموزش گام به گام Access 2000	مهندس علی ناصح
۶۸	آموزش گام به گام Outlook 2000	خسرو مهدی پور عطایی
۶۹	آموزش گام به گام FrontPage 2000	پیمان سمرقندی
۷۰	آموزش گام به گام PowerPoint 2000	بیتا خاقانی
۷۱	خودآموز گام به گام Microsoft Project 2000	مهندس مرتضی متواضع
۷۲	پرسش و پاسخ Office 2000	مهندس مرتضی متواضع
۷۳	ناگفته‌های Excel 2000	مهندس علی‌اکبر متواضع
۷۴	پرسش و پاسخ Frontpage 2000	پیمان سمرقندی
۷۵	مدیریت پایگاه داده‌ها در Access 2000	رضا حاتمیان
۷۶	خودآموز آسان Excel 2000	مهندس علی اکبر متواضع
۷۷	خودآموز آسان Word 2000	مهندس مرتضی متواضع
۷۸	خودآموز Photoshop 7 در ۲۴ ساعت	مهندسین امیرحسین رضوی - ملیحه دهقان

نشانی : سعادت آباد ـ میدان کاج ـ سرو شرقی ـ روبه‌روی خ علامه ـ شماره ۹۷

ISO 9001 Certificated
In Publication & Cover Design

VHDL for Designers

Translated by: Farzan Gity



مؤسسه فرهنگی هنری دیباگران تهران
ساختمان مرکزی: سعادت‌آباد- میدان‌کاج- سرو شرقی
روبه‌روی خیابان علامه- پلاک ۹۷
تلفن:۷-۲۰۹۸۴۴۶ دورنگار:۲۰۹۸۴۴۸
E-mail : publishing@mftmail.com
URL : www.mftsite.com

ISBN 964-354-449-4
9789643544492