

# به نام خدا

درس مدارهای منطقی برنامه پذیر  
( سیستم‌های FPGA )

استاد درس: دکتر هادی گرایلو (کتاب B)

دانشگاه صنعتی شاھروند  
دانشکده برق و رباتیک

## فصل دوم

### ساختار کد

#### ۱-۲ واحدهای اساسی در VHDL

همان طور که در شکل ۱-۲-الف نمایش داده شده است، هر کد پایه‌ای در VHDL از سه بخش تشکیل شده است:

- اعلان کتابخانه/بسته<sup>۱</sup>: شامل لیستی از کتابخانه‌ها و بسته‌های مربوطه است که در طراحی ما مورد نیاز می‌باشند. مهمترین کتابخانه‌های مورد نیاز عبارتند از std، ieee و work (دوتای آخر طبق پیش‌فرض در ویرایشگرها معمولاً به طور خودکار ایجاد می‌شوند).
- ENTITY: این بخش در حالت کلی تعیین کننده‌ی پورت‌های ورودی-خروجی مدار و نیز (به طور اختیاری) ثابت‌های عام<sup>۲</sup> می‌باشد.
- ARCHITECTURE: این بخش شامل کد VHDL بوده و توصیف‌کننده‌ی نحوه عملکرد مدار است.

بخش LIBRARY شامل مجموعه‌ای از تکه کدهای مورد استفاده است. قرار دادن این تکه کدها درون یک کتابخانه موجب می‌شود اولاً<sup>۳</sup> این تکه کدها (در طراحی فعلی) مجدداً قابل استفاده باشند، ثانیاً توسط دیگر طراحی‌ها قابل استفاده و اشتراک‌گذاری باشد. ساختار کلی یک کتابخانه در شکل ۱-۲-ب نشان داده شده است.

<sup>1</sup> Library/Package Declaration

<sup>2</sup> Generic Constants

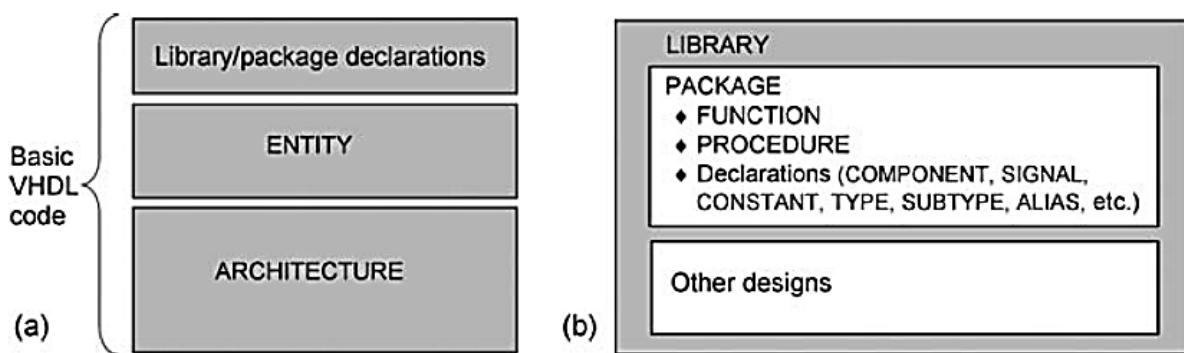


Figure 2.1

(a) Fundamental sections of a VHDL code; (b) Fundamental parts of a library.

هر مداری که قبلاً طراحی شده باشد را می‌توان در یک کتابخانه قرار داد. حسن این کار است که پس از آن در تمامی طراحی‌ها، به کمک کلمه‌ی کلیدی COMPONENT، قابل استفاده (و البته به عبارت بهتر، قابل نمونه‌سازی) خواهد بود. یک راه متداول دیگر این است که تکه کدهای نوشته شده را ابتدا داخل یک FUNCTION (تابع) و یا PROCEDURE (روال) قرار دهیم. به این دو، «زیربرنامه»<sup>۱</sup> نیز گفته می‌شود. سپس این توابع یا رووال‌ها را داخل یک PACKAGE (بسته) قرار دهیم. این بسته به نوبه‌ی خود داخل یک کتابخانه قرار می‌گیرد. در کتابخانه‌ها، معمولاً اعلان‌های عمومی انواع داده نیز انجام می‌شود.

واحدهای اساسی VHDL (شکل ۲-۱-الف) در بخش اول کتاب (اصطلاحاً «VHDL سطح مداری» شامل فصلهای ۱ تا ۷) و واحدهای مرتبط با کتابخانه (یعنی ENTITY، PACKAGE، PROCEDURE و FUNCTION، COMPONENT) در بخش دوم کتاب (اصطلاحاً «VHDL سطح سیستمی» شامل فصلهای ۸ تا ۱۰) بررسی خواهند شد.

## ۲-۲ کتابخانه‌ها و بسته‌های VHDL

کتابخانه‌های استاندارد در VHDL شامل std و ieee بوده و مهمترین بسته‌های موجود در این کتابخانه‌ها عبارتند از:

- **کتابخانه std:** این کتابخانه شامل بسته‌های زیر است:
  - بسته‌ی standard (پیوست H): این بسته در استاندارد IEEE 1076 تعریف شده و از همان نسخه اولیه‌ی VHDL (سال ۱۹۸۷) جزو این زبان بوده است. این بسته شامل تعاریف مربوط به انواع مختلف داده (BIT، BOOLEAN، INTEGER، CHARACTER و غیره) و عملگرهای منطقی، حسابی، شیفت، و تجمعی می‌باشد. این بسته در نسخه ۲۰۰۸ VHDL توسعه داده شد.

<sup>1</sup> Subprogram

- بسته‌ی textio (پیوست M): منبعی برای متن و فایل است. این بسته نیز در IEEE 1076 تعریف و در 2008 VHDL توسعه داده شده است.

#### ◦ کتابخانه‌ی ieee :

- بسته‌ی std\_logic\_1164 (پیوست I): دو نوع داده‌ی 9-مقداره شامل STD\_LOGIC و STD\_ULOGIC را تعریف می‌کند. مهمترین ویژگی این دو نوع داده در مقایسه با نوع اولیه‌ی BIT، معرفی و استفاده از حالت‌های سنتزپذیر جدید «بی‌همیت»<sup>۱</sup> (با نماد -) و امپدانس بالا<sup>۲</sup> (با نماد Z) می‌باشد. (نوع داده‌ی BIT فقط اجازه‌ی گرفتن مقادیر 0 و 1 را می‌دهد). این بسته در استاندارد IEEE 1164 تعریف شده است.
- بسته‌ی numeric\_std (پیوست J): دو نوع داده‌ی SIGNED و UNSIGNED را به همراه عملگرهای مربوطه که نوع پایه‌ای آنها STD\_LOGIC باشد، تعریف می‌کند.
- بسته‌ی numeric\_bit: مانند بسته قبلی است با این تفاوت که نوع پایه‌ای در این بسته، BIT می‌باشد.
- بسته‌ی numeric\_std\_unsigned (پیوست N): این بسته در 2008 VHDL معرفی شده و هدف از آن استفاده به جای بسته غیراستاندارد std\_logic\_unsigned می‌باشد.
- بسته‌ی numeric\_bit\_unsigned: این بسته نیز در 2008 VHDL معرفی شده و مشابه با بسته قبلی است اما با این تفاوت که به جای کار با BIT\_VECTOR با STD\_LOGIC\_VECTOR کار می‌کند.
- بسته‌ی env: این بسته در 2008 VHDL معرفی شده و شامل روالهای شروع و خاتمه جهت ارتباط با محیط شبیه‌ساز می‌باشد.
- بسته‌ی fixed\_pkg (به همراه بسته‌های مربوط به خود): این بسته توسط شرکت Kodak توسعه داده شده و در 2008 VHDL معرفی شده است. این بسته دو نوع داده ممیز ثابت علامتدار و بدون علامت UFIXED و SFIXED را به همراه عملگرهای مربوطه تعریف می‌کند.
- بسته‌ی float\_pkg (به همراه بسته‌های مربوطه): این بسته نیز توسط شرکت Kodak توسعه داده شده و در 2008 VHDL معرفی شده است. این بسته نوع داده ممیز شناور FLOAT را به همراه عملگرهای مربوطه تعریف می‌کند.

#### ◦ بسته‌های غیراستاندارد:

<sup>1</sup> Don't Care

<sup>2</sup> High Impedance

- بسته‌ی std\_logic\_arith (پیوست K): دو نوع داده‌ی UNSIGNED و SIGNED را به همراه عملگرهای مربوطه تعریف می‌کند. این بسته «تاخدی» مشابه با بسته‌ی numeric\_std است.
- بسته‌ی std\_logic\_unsigned: این بسته توابعی را جهت انجام عملیات حسابی، مقایسه، و شیفت روی سیگنال‌هایی از نوع STD\_LOGIC\_VECTOR (به عنوان اعداد بدون علامت) معرفی می‌کند.
- بسته‌ی std\_logic\_signed (پیوست L): این بسته مشابه با بسته‌ی قبلی است تمت به جای کار با اعداد بدون علامت، با اعداد علامتدار کار می‌کند.
- بسته‌هایی که در بالا به آنها اشاره شد، در فصل‌های ۳ و ۴ به طور مفصل‌تر بررسی و معرفی خواهند شد.

### ۲-۳ اعلان کتابخانه/بسته

برای اعلان یک بسته در یک طراحی، نیاز به دو اعلان می‌باشد: یکی اعلان کتابخانه‌ای که بسته داخل آن قرار گرفته است؛ دیگری عبارت USE مربوط به بسته‌ی خاصی که از آن کتابخانه استفاده شده است:

```
LIBRARY library_name;
USE library_name.package_name.all;
```

متدائل‌ترین بسته‌هایی که استفاده می‌شوند عبارتند از:

- بسته‌ی standard از کتابخانه‌ی std (در حالت پیش‌فرض قابل رویت است)،
- کتابخانه‌ی work (پوشه‌ای است که فایل‌های پروژه‌ی شما داخل آن ذخیره می‌شود؛ این کتابخانه نیز به طور پیش‌فرض قابل رویت است)،
- بسته‌ی ieee از کتابخانه‌ی std\_logic\_1164 (این بسته در صورت نیاز باید صراحتاً اعلان شود).

اعلان‌های مربوط به بسته‌ها و کتابخانه‌های فوق به صورت زیر است:

```

1 -----
2 LIBRARY std;           --optional declaration
3 USE std.standard.all; --optional declaration
4 LIBRARY work;          --optional declaration
5 USE work.all;          --optional declaration
6 LIBRARY ieee;
7 USE ieee.std_logic_1164.all;
8 USE work.my_package.all;
9 -----

```

بسته‌ی standard (خطوط شماره ۲-۳ در بالا) به طور پیش‌فرض قابل رویت است لذا نیازی به اعلان صریح آن نمی‌باشد. همین مطلب در مورد کتابخانه‌ی work (خطوط ۴-۵) نیز صادق است. در طرف دیگر، بسته‌ی std logic 1164 (خطوط ۶-۷) در صورت استفاده از انواع داده‌ی STD(U)LOGIC در پروژه، باید اعلان شود. اگر نیاز به اعلان یک بسته‌ی تعریف شده توسط کاربر باشد، باید مطابق با خط شماره ۸ تعریف شود (برای اعلان خط ۸ نیازی به خط شماره ۴ نمی‌باشد زیرا خط ۴ در لیست پیش‌فرض قرار دارد).

در خطوط فوق نماد نقطه ویرگول (:) نشانگر پایان اعلان یا دستور و نماد خط تیره‌ی مضاعف (--) معرف خط توضیح است. زبان VHDL حساس به بزرگی/کوچکی حروف نمی‌باشد.

## ۴-۲ موجودیت (ENTITY)

مهمترین قسمت یک موجودیت (ENTITY)، PORT بوده و شامل لیستی از تمام پورت‌های (یا پین‌های) ورودی و خروجی مدار می‌باشد. گرامر ساده‌ای موجودیت در زیر و گرامر کامل و پیچیده‌ی آن به زودی معرفی می‌شود:

```

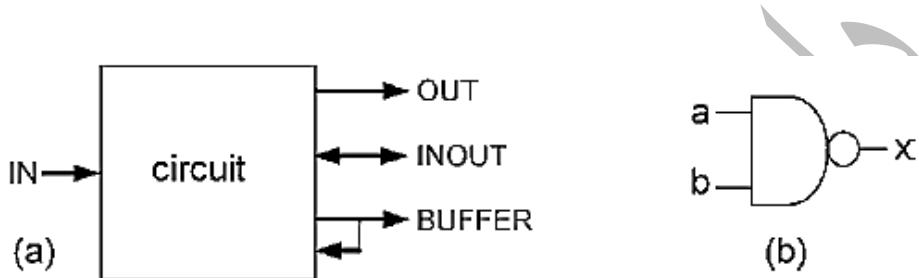
ENTITY entity_name IS
  PORT (
    port_name: port_mode signal_type;
    port_name: port_mode signal_type;
    ...);
END [ENTITY] [entity_name];

```

موجودیت می‌تواند هر نامی به جز کلمه‌ی VHDL و تعداد بسیار کمی از اسمی رزرو شده (پیوست G) باشد. پورت‌های مدار نیز از این قاعده تبعیت می‌کنند.

تمام اعضاء متعلق به لیست PORT از نوع سیگنال، SIGNAL (در مقابل نوع متغیر، VARIABLE) بوده و به عنوان سیمهایی که از/به مدار خارج/داخل می‌شوند تعبیر می‌شوند. هر

کدام از این اعضاء (یا همان پورت‌ها) می‌توانند دارای یک «حالت»<sup>۱</sup> باشند؛ مقادیر ممکن حالت‌ها عبارت است از IN، OUT، inout، BUFFER. همان طور که در شکل ۲-۲-الف نیز نمایش داده شده است، دو نوع IN و OUT واقعاً سیم‌هایی تک‌جهته هستند اما نوع inout نیز زمانی استفاده می‌شود که سیگنالی علاوه بر این که باید به خارج ارسال شود، در داخل نیز قرار است مجدداً استفاده (و خوانده) شود. علاوه بر حالت، هر پورت دارای «نوع»<sup>۲</sup> نیز می‌باشد. انواع ممکن برای هر پورت از قبیل bit، integer، std\_logic و مانند آن بوده و بعدها در فصل ۳ به طور کامل معرفی و بررسی خواهند شد.



**Figure 2.2**  
(a) VHDL port modes; (b) NAND gate.

برای اجتناب از استفاده از نوع BUFFER می‌توان از یک سیگنال کمکی بهره گرفت. به علاوه در VHDL 2008 از نوع OUT برای خواندن داخلی نیز می‌توان استفاده کرد. نوع inout به طور خاص در پیاده‌سازی حافظه‌ها استفاده می‌شود. در حافظه‌ها اغلب از یک گذرگاه داده برای حمل داده‌های مربوط به خواندن و نوشتن استفاده می‌شود (فصل ۱۳ به معرفی حافظه‌ها خواهد پرداخت).

مثال: برای گیت NAND نشان داده شده در شکل ۲-۲-ب می‌توان از موجودیت زیر، به نام nand\_gate استفاده کرد. این کد نشان می‌دهد مدار دارای سه پورت I/O است که دو تای آنها ورودی (به نام‌های a و b و دارای حالت IN) و سومی یک خروجی (به نام x و دارای حالت OUT) می‌باشند. نوع تمام این سه سیگنال، bit می‌باشد.

```
ENTITY nand_gate IS
    PORT (a, b: IN BIT;
          x: OUT BIT);
END ENTITY;
```

<sup>1</sup> Mode

<sup>2</sup> Type

در شکل کلی گرامر یک موجودیت، علاوه بر میدان PORT سه میدان دیگر نیز می‌توانند حضور داشته باشند: بخش اعلان‌های GENERIC (قبل از PORT)، یک بخش اعلان عمومی (بعد از PORT)، و بخشی شامل فراخوانی‌ها<sup>۱</sup> و یا فرآیندهای<sup>۲</sup> پسیو (بعد از PORT):

```

ENTITY entity_name IS
  [GENERIC (
    const_name: const_type const_value;
    ...);
  [PORT (
    signal_name: mode signal_type;
    ...);
  [entity_declarative_part]
  [BEGIN
    entity_statement_part]
END [ENTITY] [entity_name];

```

میدان PORT تنها بخش یک موجودیت است که ضروری (جهت شبیه‌سازی و سنتز) می‌باشد. بخش اختیاری GENERIC برای اعلان ثابت‌هایی که در سرتاسر طراحی قابل رویت می‌باشند، استفاده می‌شود (در بخش ۶-۲ بیشتر توصیف خواهد شد). بخش اعلان اختیاری عمومی، declarative، گرچه به ندرت استفاده می‌شود، می‌تواند شامل موارد زیر باشد: اعلان زیربرنامه، بدنه‌ی زیربرنامه، اعلان نوع، اعلان زیرنوع، اعلان ثابت، اعلان سیگنال، اعلان متغیرهای به اشتراک گذاشته شده، اعلان فایل، اعلان همپوشانی<sup>۳</sup>، ویژگی‌های یک خصوصیت<sup>۴</sup>، ویژگی‌های یک انفال<sup>۵</sup>، عبارت use، اعلان کلیشه‌ی گروه<sup>۶</sup>، و اعلان گروه.

و بالاخره بخش اختیاری دستورات، statements، گرچه به ندرت استفاده می‌شود اما می‌تواند شامل فراخوانی‌های پسیو و یا فرآیندهای پسیو باشد (منتظر از پسیو، آنهایی هستند که هیچ تخصیص سیگنالی ندارند؛ اینها برای مقاصدی مانند تست مقادیر PORT استفاده می‌شوند). بخش GENERIC تنها بخش اختیاری است که متدائل بوده و استفاده می‌شود.

<sup>1</sup> Passive Calls

<sup>2</sup> Passive Processes

<sup>3</sup> Alias Declaration

<sup>4</sup> Attribute Specification

<sup>5</sup> Disconnection Specification

<sup>6</sup> Group Template Declaration

مثال: موجودیت زیر شامل اولین سه بخش از چهار بخش اشاره شده در گرامر اخیر می باشد:

```

ENTITY controller IS
    GENERIC (N: INTEGER := 8);
    PORT (a, b: IN INTEGER RANGE 0 TO 2**N-1;
          x: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
    TYPE byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
    CONSTANT mask: byte "00001111";
END ENTITY;

```

## ۵-۲ معماری (ARCHITECTURE)

بخش معماری وظیفه‌ی توصیف نحوه‌ی عملکرد مدار را بر عهده دارد. از روی این توصیف یک مدار عملی قابل استنتاج است. گرامر ساده‌ی بخش معماری به صورت زیر است:

```

ARCHITECTURE architecture_name OF entity_name IS
    [architecture_declarative_part]
    BEGIN
        architecture_statements_part
    END [ARCHITECTURE] [architecture_name];

```

همان طور که ملاحظه می‌شود یک بخش معماری دارای دو بخش است: یک بخش اختیاری اعلان‌ها، declarative، و بخش شامل دستورات، statements (از کلمه‌ی BEGIN به بعد). بخش اعلان‌ها شامل همان موارد موجود در بخش اعلان موجودیت به علاوه‌ی اعلان‌های اجزاء و مشخصات پیکربندی می‌باشد. بخش دوم، جایی است که دستورات VHDL در آن نوشته می‌شوند. مشابه با نام موجودیت، نام معماری نیز می‌تواند شامل هر کلمه‌ای حتی نام موجودیت باشد.

مثال: یک نمونه معماری برای گیت NAND مربوط به شکل ۲-۲-ب به نام arch در زیر نشان داده شده است. معنای این معماری این است: مدار باید عمل NAND را بین a و b انجام داده، حاصل را به x تخصیص<sup>۱</sup> دهد. در این مثال، هیچ اعلانی در بخش اعلان‌ها استفاده نشده و بخش دستورات تنها شامل یک دستور منطقی است.

<sup>۱</sup> Assignment

```

ARCHITECTURE arch OF nand_gate IS
BEGIN
    x <= a NAND b;
END ARCHITECTURE;

```

## ۶-۲ بخش عام (GENERIC)

بخش GENERIC شامل اعلان مشخصات پارامترهای عام (یعنی ثابت‌های عام که به راحتی می‌توان آنها را در کاربردهای مختلف تغییر داده یا اصلاح نمود) می‌باشد. هدف اصلی این پارامترها، پارامتری کردن<sup>۱</sup> طراحی و درنتیجه افزایش قابلیت انعطاف و استفاده‌پذیری مجدد است. بخش GENERIC تنها بخش مجازی است که قبل از PORT آورده می‌شود. علت این امر، امکان سرتاسری کردن پارامترهای اعلان شده حتی در بخش PORT می‌باشد. گرامر ساده‌ی این بخش به صورت زیر است:

```

GENERIC (constant_name: constant_type := constant_value;
          constant_name: constant_type := constant_value;
          ... );

```

مثال: در بخش GENERIC نوشته شده برای موجودیت زیر، دو پارامتر m و n اعلان شده‌اند. اولی از نوع INTEGER بوده و دارای مقدار ۸ شده است. دومی از نوع BIT\_VECTOR بوده و دارای مقدار "0101" شده است. بنابراین هر جا از کد VHDL که به این دو پارامتر برخورد شود (حتی در خود ENTITY) به طور خودکار از مقادیر گفته شده به جای این پارامترها استفاده می‌شود.

```

ENTITY my_entity IS
    GENERIC (m: INTEGER := 8;
              n: BIT_VECTOR(3 DOWNTO 0) := "0101");
    PORT (...);
END my_entity;

```

**اعلان GENERIC MAP:** اگر بسته‌ای شامل اعلان پارامترها (یا همان ثابت‌ها)ی عام در بخش GENERIC خود باشد (مانند نمونه کد نشان داده شده در مثال اخیر) و این بسته در یک طراحی

<sup>۱</sup> Parameterize

دیگر نمونه‌سازی شود، آن طراحی می‌تواند مقادیری که برای پارامترهای عام آن بسته تعریف شده را در حین نمونه‌سازی (برای نمونه‌ی جدید از آن بسته) تغییر دهد. این کار به کمک اعلان GENERIC MAP انجام می‌شود.

## ۷-۲ مثال‌های مقدماتی از VHDL

هدف از این بخش ارائه‌ی یک دید کلی از ساختار کد در زبان VHDL است.

### مثال ۱-۲ مدار مقایسه-جمع

در سمت چپ از شکل ۳-۲ یک مدار مشکل از دو بلوك نمایش داده شده است. ورودی‌های این مدار دو مقدار بدون علامت ۳ بیتی (a و b) دارای مقادیری بین ۰ تا ۷) و خروجی‌های آن نیز شامل comp (تک بیتی) و sum (که برای جلوگیری از وقوع سرریز چهار بیتی درنظر گرفته شده و لذا مقادیر ممکن آن بین ۰ تا ۱۵ است) می‌باشند. بخش بالایی مدار باید a را با b مقایسه نموده و در صورت بزرگتر بودن a از b (یعنی  $a > b$ ) خروجی ۱ و در غیر این صورت ۰ تولید نماید. بخش پایینی مدار نیز باید حاصل جمع a و b (sum) را تولید نماید.

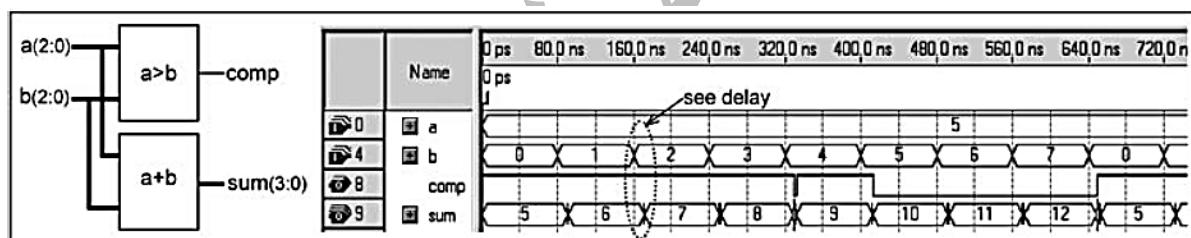


Figure 2.3  
Circuit of example 2.1 and respective simulation results.

یک نمونه کد VHDL برای این مدار در زیر نشان داده شده است. توجه کنید که خطوط شماره ۱، ۴، ۱۰، و ۱۶ جهت خواناتر کردن و متمایزساختن بخش‌های مهم کد استفاده شده‌اند. این خطوط، کد را به سه قسمت مهم و اساسی که قبلًا معرفی شدند، تقسیم کرده‌اند: در خطوط ۳-۲ «اعلان کتابخانه» انجام گرفته است. «موجودیت» دارای نام comp\_add بوده و در خطوط ۹-۵ قرار گرفته است. «معماری» نیز دارای نام circuit بوده و در خطوط ۱۱-۱۵ آورده شده است.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY comp_add IS
6     PORT (a, b: IN INTEGER RANGE 0 TO 7;
7             comp: OUT STD_LOGIC;
8             sum: OUT INTEGER RANGE 0 TO 15);
9 END ENTITY;
10 -----
11 ARCHITECTURE circuit OF comp_add IS
12 BEGIN
13     comp <= '1' WHEN a>b ELSE '0';
14     sum <= a + b;
15 END ARCHITECTURE;
16 -----

```

توجه کنید که موجودیت شامل تمام پورت‌های I/O است. ورودی‌ها a و b (حالت IN، خط ۶) هر دو از نوع INTEGER بوده و بین ۰ تا ۷ (مقادیر بدون علامت ۳ بیتی) تغییر می‌کنند. خروجی‌ها شامل comp (خط ۷) و sum (خط ۸) بوده، اولی از نوع STD\_LOGIC (تک بیتی) و دیگری از نوع INTEGER و مقداری بدون علامت ۴ بیتی (بین ۰ تا ۱۵) می‌باشند. معماری تنها شامل دو دستور است: اولی (خط ۱۳) مقایسه را (به کمک دستور WHEN) انجام داده و دومی (خط ۱۴) عمل جمع را (به کمک عملگر +) انجام می‌دهد. در این مثال، بخش اعلان معماری شامل هیچ اعلانی نمی‌باشد.

نتایج شبیه‌سازی در شکل ۲-۳ نشان داده شده است. در این شکل، قبل از هر سیگنال ورودی یک فلش که داخل آن حرف I نوشته شده، نشان داده شده است. همین کار برای سیگنال‌های خروجی اما با حرف O انجام شده است. به a مقدار ثابتی تخصیص داده شده اما b تمام مقادیر ممکن ۳-۷ بیتی (بین ۰ تا ۷) را گرفته است. نتیجه به این صورت شده است که موقوعی که  $a > b$  باشد، "comp = 1" شده است. خروجی sum نیز برابر با  $a+b$  (بدون در نظر گرفتن سریز) می‌باشد. توجه کنید که شبیه‌سازی انجام شده در اینجا از نوع شبیه‌سازی زمان‌بندی<sup>۱</sup> است زیرا تاخیرهای ناشی از انتشار در نظر گرفته شده‌اند.

به وقوع اختلال<sup>۲</sup> در خروجی comp توجه کنید. این اختلال زمانی که مقدار b از ۳ به ۴ تغییر می‌کند رخ داده و علت آن تغییر تمام بیت‌های b در حین این تغییر است ("0111" -> "1000"). از آنجایی که تمام بیت‌ها همزمان در یک لحظه تغییر نمی‌کنند و نیز با توجه به این که هر تغییر به

<sup>1</sup> Timing<sup>2</sup> Glitch

صورت ناگهانی نیست بلکه مانند یکتابع شیب به صورت پیوسته انجام می‌شود، لذا در زمانهای گذر و تغییر، ممکن است برای لحظاتی  $a \leq b$  شده و لذا نتیجه مقایسه  $a$  و  $b$  مقادیر ناگهانی و دور از انتظار به خود بگیرد. این نوع از اختلال کاملاً طبیعی است.

## مثال ۲-۲ فلیپ فلاپ نوع D

شکل ۲-۴ یک DFF را نشان می‌دهد که یکی از مدارات ذخیره‌سازی پایه محسوب می‌شود. معمولاً هزاران عدد از این مدارات در افزارهای FPGA وجود دارند. ورودی‌های آن شامل  $d$  (داده)،  $clk$  (کلک)، و  $rst$  (ریست) بوده و تنها خروجی آن نیز  $q$  (داده ذخیره شده) می‌باشد. در این مثال، DFF در لبه بالاروندهی کلک تریگر می‌شود. در این لحظات، مدار مقدار ورودی را در خروجی کپی می‌کند ( $q=d$ ). در بقیه لحظات، مقدار خروجی بدون تغییر باقی می‌ماند. ورودی ریست در این مدار از نوع آسنکرون بوده (یعنی به کلک وابسته نیست) و در صورت وقوع  $rst='1'$  خروجی بلاfaciale صفر می‌شود.

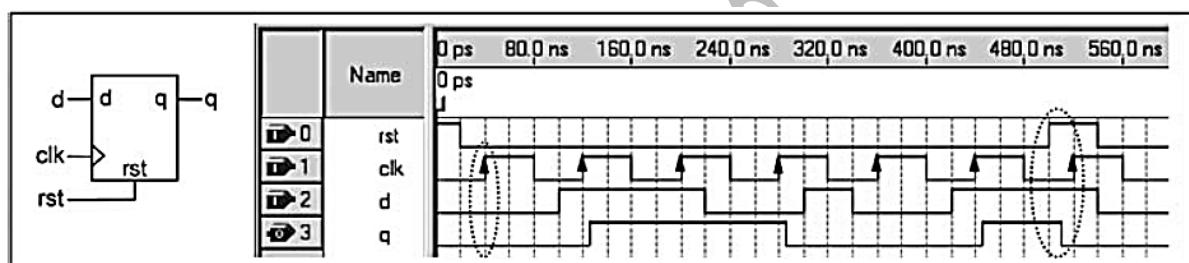


Figure 2.4  
Circuit of example 2.2 and respective simulation results.

راههای زیادی جهت پیاده‌سازی یک DFF وجود دارد؛ یکی از آنها در زیر نشان داده شده است. یک ویژگی منحصر‌بفرد برای زبان‌هایی مانند VHDL این است که آنها برخلاف زبان‌های متداول برنامه‌نویسی کامپیوتری که از نوع ترتیبی<sup>۱</sup> هستند، از نوع همزمان<sup>۲</sup> می‌باشند. این ویژگی سبب می‌شود که برای پیاده‌سازی مدارات کلک‌دار (مانند فلیپ فلاپ‌ها)، VHDL را «اجبار» به انجام ترتیبی کارها کنیم. برای انجام این کار از PROCESS استفاده می‌کنیم.

<sup>1</sup> Sequential

<sup>2</sup> Concurrent

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY flip_flop IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7             q: OUT STD_LOGIC);
8 END ENTITY;
9 -----
10 ARCHITECTURE flip_flop OF flip_flop IS
11 BEGIN
12     PROCESS (clk, rst)
13     BEGIN
14         IF (rst='1') THEN
15             q <= '0';
16         ELSIF (clk'EVENT AND clk='1') THEN
17             q <= d;
18         END IF;
19     END PROCESS;
20 END ARCHITECTURE;
21 -----

```

نکات مرتبط با کد فوق عبارتند از:

خطوط ۲-۳: اولین بخش از کد (یعنی اعلان‌های کتابخانه‌ای) هستند. این بخش شامل نام کتابخانه به همراه عبارت use می‌باشد. از آنجاکه در طراحی فعلی از نوع STD\_LOGIC استفاده شده است، باید از بسته‌ی std\_logic\_1164 استفاده شود. دو کتابخانه‌ی ضروری دیگر (یعنی std و work) به طور پیش‌فرض قابل رویت توسط طراحی هستند (لذا صریحاً اعلان نشده‌اند).

خطوط ۵-۸: دومین بخش اساسی از کد، یعنی موجودیت است که در این مثال دارای نام flip-flop است.

خطوط ۱۰-۱۱: بخش سوم از کد یعنی معماری بوده و دارای همان نام موجودیت است.

خط ۶: پورت‌های ورودی همگی از نوع STD\_LOGIC تعریف شده‌اند.

خط ۷: پورت خروجی نیز از نوع STD\_LOGIC تعریف شده است.

خطوط ۱۹-۲۰: بخش کد معماری که بعد از کلمه‌ی BEGIN شروع می‌شود. در اینجا این کد تنها شامل یک فرآیند (مشخص شده با کلمه‌ی PROCESS) می‌باشد زیرا ما در این مثال به یک مدار ترتیبی (کلاک‌دار) نیازمندیم. کد داخل فرآیند به صورت ترتیبی اجرا می‌شود.

خط ۱۲: در اینجا دو سیگنال `clk` و `rst` در لیست حساسیت<sup>۱</sup> فرآیند قرار گرفته‌اند. هر زمان که یکی از این سیگنال‌ها تغییر کند، کد داخل فرآیند انجام می‌شود.

خطوط ۱۴-۱۵: اگر '`rst = 1`' شود، فلیپ فلاب، مستقل از وضعیت کلاک، ریست می‌شود (ورودی آسنکرون).

خطوط ۱۷-۱۶: اگر ریست (`rst`) فعال نباشد و `clk` تغییر کند (این تغییر به معنای وقوع یک «واقعه» یا EVENT روی کلاک می‌باشد) و تغییر در جهتی باشد که مقدار کلاک پس از تغییر برابر '۱' شود (یعنی لبهی بالاروندهی کلاک رخ داده باشد) آن‌گاه سیگنال ورودی (`d`) در فلیپ فلاب ذخیره می‌شود ( $q \leftarrow d$ ).

خطوط ۱۵ و ۱۷: از عملگر `=` برای تخصیص یک مقدار به یک SIGNAL استفاده شده است (تمام پورتها به طور پیش‌فرض از نوع SIGNAL هستند). به طور معادل، از عملگر `=` برای تخصیص مقدار برای متغیرها (نوع VARIABLE) استفاده می‌شود.

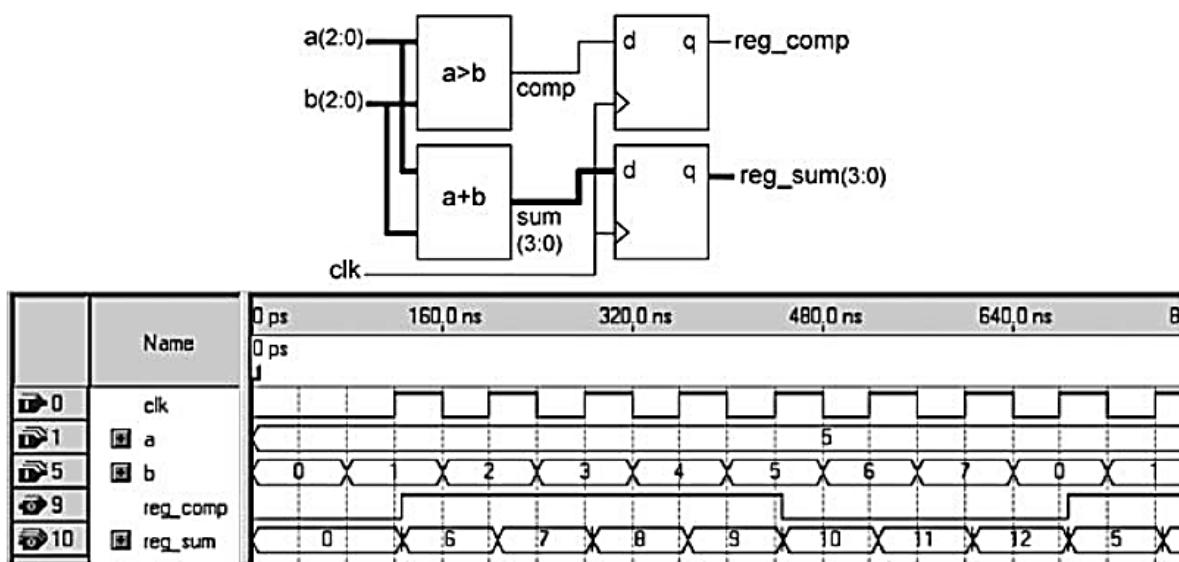
خطوط ۱، ۴، ۹ و ۲۱: برای سازمان‌دهی بهتر کد استفاده شده‌اند.

نتایج شبیه‌سازی این کد در شکل ۴-۲ نشان داده شده است. این شبیه‌سازی از نوع شبیه‌سازی زمان‌بندی است. توصیه می‌شود که حالات مختلف را برای بررسی صحت عملکرد فلیپ فلاب چک کنید. برای مشخص کردن زمان‌هایی که مدار شفاف شده است (یعنی مقدار ورودی را به خروجی منتقل کرده است) از پیکانهایی روی شکل موج کلاک استفاده شده است.

### مثال ۲-۳: مدار مقایسه-جمع رجیستر شده

شکل ۲-۵ مداری را نشان می‌دهد که موارد بررسی شده در دو مثال گذشته را با هم تجمع کرده است. یعنی، از فلیپ فلاب در خروجی مدار مقایسه-جمع استفاده شده است تا بدین ترتیب خروجی‌های `comp` و `sum` را رجیستر (یعنی ذخیره) کند. بنابراین در اینجا این خروجی‌ها `reg_sum` و `reg_comp` نامیده شده‌اند.

<sup>۱</sup> Sensivity List



**Figure 2.5**  
Circuit of example 2.3 and respective simulation results.

یک کد VHDL مربوط به این مدار در زیر نشان داده شده است. سیگنال‌های اولیه (sum و comp) در بخش اولیهی معماری محاسبه می‌شوند (خطوط ۱۶-۱۷). پس از آن از یک فرآیند استفاده شده است (خطوط ۱۸-۲۴) که برای اجرا نیاز به فلیپ فلاپ دارد (مدار ترتیبی). توجه کنید که در اینجا در کل نیاز به پنج فلیپ فلاپ داریم. همچنین توجه کنید که در اینجا sum و comp دیگر سیگنال‌های داخلی هستند لذا در بخش اعلان معماری (خطوط ۱۳-۱۴) آورده شده‌اند (و نه در لیست پورت‌های موجودیت). نتایج شبیه‌سازی در شکل ۲-۵ نمایش داده شده است. در اینجا ملاحظه می‌کنید که برخلاف مثال ۱-۲، خروجی‌ها زمانی به روز رسانی می‌شوند که لبه‌ی بالاروندهی کلاک رخ داده باشد (بنابراین، در اینجا دیگر اختلال در خروجی مشاهده نمی‌شود).

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY registered_comp_add IS
6     PORT (clk: IN STD_LOGIC;
7             a, b: IN INTEGER RANGE 0 TO 7;
8             reg_comp: OUT STD_LOGIC;
9             reg_sum: OUT INTEGER RANGE 0 TO 15);
10 END ENTITY;
11 -----
12 ARCHITECTURE circuit OF registered_comp_add IS
13     SIGNAL comp: STD_LOGIC;
14     SIGNAL sum: INTEGER RANGE 0 TO 15;
15 BEGIN
16     comp <= '1' WHEN a>b ELSE '0';
17     sum <= a + b;
18     PROCESS (clk)
19     BEGIN
20         IF (clk'EVENT AND clk='1') THEN
21             reg_comp <= comp;
22             reg_sum <= sum;
23         END IF;
24     END PROCESS;
25 END ARCHITECTURE;
26 -----

```

#### مثال ۴-۲: رمزگشایی عام آدرس<sup>۱</sup>

دیاگرام کلی (سطح بالا) از یک رمزگشایی عام آدرس N بیتی در شکل ۴-۶ نمایش داده شده است. این مدار دارای دو ورودی به نام‌های آدرس (به نام address، N بیتی) و فعال‌ساز (به نام ena، تک بیتی) و یک خروجی (به نام word\_line 2<sup>N</sup> بیتی) می‌باشد. در شکل مذکور، جدول صحبت این مدار به ازاء  $N=2$  نیز نشان داده شده است. مطابق با این جدول ملاحظه می‌شود که در خروجی، در حالت معمولی و فعال بودن مدار، همواره تمام بیت‌ها به جز یک بیت (که محل این تک بیت توسط ورودی تعیین می‌شود) مانند هم (دارای مقدار '1') هستند. در حالتی که مدار فعال نباشد (یعنی '0' ena='0' باشد)، تمام بیت‌ها دارای مقدار '1' می‌باشند.

<sup>1</sup> Generic Address Decoder

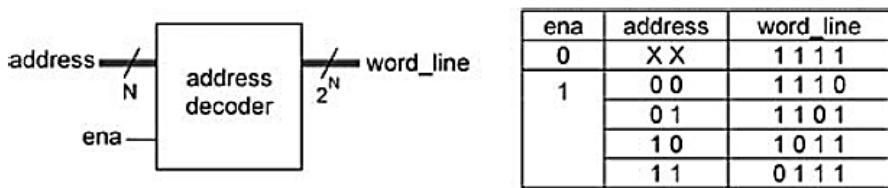


Figure 2.6  
Address decoder of example 2.4.

در زیر یک نمونه کد VHDL برای این مدار نشان داده شده است. در این جا نیازی به اعلان کتابخانه نمی باشد زیرا تنها از انواع دادهای موجود در بسته standard (که همیشه به طور پیشفرض قابل رویت توسط طراحی است) استفاده شده است. موجودیت در خطوط ۷-۲ تعریف شده و شامل اعلان‌های PORT و GENERIC می باشد. در خط ۳، پارامتر N به عنوان یک پارامتر عام (generic) تعریف و استفاده شده است لذا می‌توان از این مدار برای هر رمزگشای آدرس با ابعاد دلخواهی استفاده نمود. سیگنال‌های ورودی و خروجی (اعلان‌های PORT واقع در خطوط ۴-۶) با توجه به شکل ۲-۶ انتخاب و استفاده شده‌اند.

بخش معماری در خطوط ۹-۱۶ نوشته شده است. این بخش کاملاً عام است زیرا با تغییر مقدار N نیاز به هیچ تغییری در کد بخش معماری نیست (تنها تغییر لازم در خط شماره ۳ باید داده شود). از دستور GENERATE (که در فصل پنجم معرفی و توضیح داده خواهد شد) برای ایجاد یک حلقه استفاده شده است. این حلقه موجب می‌شود در حالتی که 'ena='0' است، تمام بیت‌های خروجی مقدار '1' بگیرند و در حالتی که 'ena='1' باشد، تنها یک بیت مقدار '0' و بقیه مقدار '1' بگیرند.

```

1 -----
2 ENTITY address_decoder IS
3     GENERIC (N: NATURAL := 3);
4     PORT (address: IN NATURAL RANGE 0 TO 2**N-1;
5             ena: BIT;
6             word_line: OUT BIT_VECTOR(2**N-1 DOWNTO 0));
7 END address_decoder;
8 -----
9 ARCHITECTURE address_decoder OF address_decoder IS
10 BEGIN
11     gen: FOR i IN address'RANGE GENERATE
12         word_line(i) <= '1' WHEN ena='0' ELSE
13             '0' WHEN i=address ELSE
14             '1';
15     END GENERATE;
16 END address_decoder;
17 -----

```

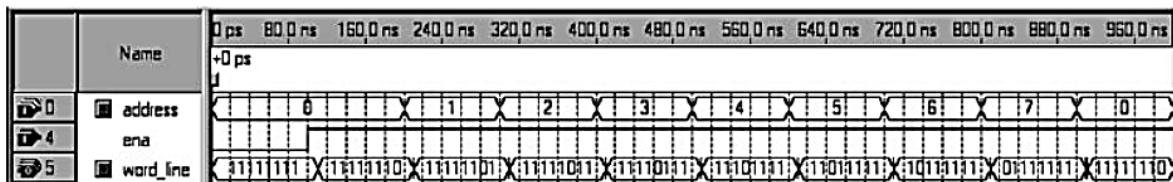


Figure 2.7  
Simulation results from the address decoder of example 2.4.

شکل ۲-۷ نتایج شبیه‌سازی در حالت  $N=3$  نشان می‌دهد. ملاحظه می‌کنید که در حالت '0'  $\text{ena} = '0'$  تمام بیت‌های خروجی برابر '1' (سطح بالا) هستند.

## ۸-۲ رهنمونهای جهت کدنویسی

در این کتاب به منظور صرفه‌جویی در صفحات، از اسمی مختصر برای نام‌گذاری سیگنال‌ها استفاده شده و همچنین در یک خط ممکن است چندین اعلان یا تعریف گنجانده شود. در پروژه‌های بزرگ یا در پروژه‌هایی که چندین تیم مختلف همکاری می‌کنند بهتر است یک قاعده و استاندارد ثابت و مشخصی تعریف و رعایت شود.

برای مثال از اسمی معنادار (بامسماً) برای نام‌گذاری سیگنال‌ها استفاده شود. اگر مدار شکل ۲-۸ را در نظر بگیریم، یک نوع کدنویسی که جنبه‌ی صرفه‌جویی در جا و مکان را در نظر گرفته باشد می‌تواند به صورت زیر نوشته شود (شکل ۲-۸-الف):

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY multiplexer IS
6     PORT (x0, x1, x2, x3: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7             sel: IN NATURAL RANGE 0 TO 3;
8             y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9 END ENTITY;
10 -----
11 ARCHITECTURE multiplexer OF multiplexer IS
12 BEGIN
13     y <= x0 WHEN sel=0 ELSE
14         x1 WHEN sel=1 ELSE
15         x2 WHEN sel=2 ELSE
16         x3;
17 END ARCHITECTURE;
18 -----

```

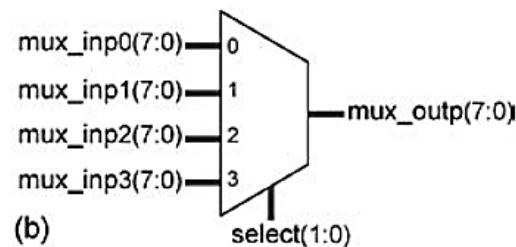
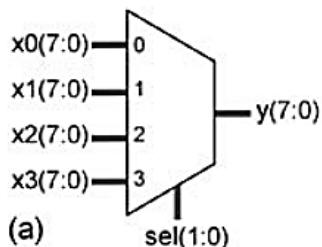


Figure 2.8

Multiplexer represented with (a) short and (b) more meaningful signal names.

از طرف دیگر، یک نمونه کدنویسی جاگیرتر می‌تواند به صورت زیر باشد (شکل ۲-۸-ب):

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY multiplexer_4x8 IS
6   GENERIC (
7     N: NATURAL := 8;      --bits in in/out signals
8     M: NATURAL := 2);    --bits in select
9   PORT (
10     mux_inp0:  IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
11     mux_inp1:  IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
12     mux_inp2:  IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
13     mux_inp3:  IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
14     select:    IN STD_LOGIC_VECTOR(M-1 DOWNTO 0);
15     mux_outp:  OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
16 END ENTITY;
17 -----
19 ARCHITECTURE multiplexer_4x8 OF multiplexer_4x8 IS
20 BEGIN
21   mux_outp <= mux_inp0 WHEN select="00" ELSE
22             mux_inp1 WHEN select="01" ELSE
23             mux_inp2 WHEN select="10" ELSE
24             mux_inp3;
25 END ARCHITECTURE;
26 -----

```

در کد فوق به نکات زیر توجه کنید:

- ۱) از سیگنال‌های با معناتر استفاده شده است (خطوط ۵، ۱۰-۱۵) البته این کار معمولاً منجر به طولانی‌تر شدن نام‌ها می‌شود.
- ۲) در موجودیت، هر سیگنال در یک خط جداگانه نوشته شده است (خطوط ۶-۱۵).
- ۳) برای تمامی پورتها از نوع `STD_LOGIC_VECTOR` استفاده شده است (خطوط ۱۰-۱۵). این نوع برای تمام پورتهای ورودی-خروجی متداول است.
- ۴) برای مشخص کردن بازه‌ی داده‌ها تنها از ترتیب نزولی استفاده شده و همواره مقدار نهایی برابر صفر است. بنابراین، MSB همیشه در سمت چپ قرار داشته و بزرگترین اندیس را (برابر  $'1 - \text{number of bits}$ ) به خود اختصاص می‌دهد.
- ۵) کلمات رزرو شده با حروف بزرگ و مابقی با حروف کوچک نوشته می‌شوند (توجه کنید که VHDL حساس به بزرگی/کوچکی حروف نیست).

۶) استفاده از خطوط جداساز (خطوط ۱، ۴، ۱۷ و ۲۶) موجب سازماندهی بهتر کد و جداسازی اجزاء اساسی آن می‌شود. البته استفاده از خطوط جداساز بیشتر (البته با طول کوتاهتر از خطوط نامبرده اخیر) نیز می‌تواند مفید واقع شود. استفاده از برچسب‌های اختیاری (مثلاً برای PEOCESS) مفید است.

دیگر نکات مفید برای کدنویسی عبارتند از:

- ۱) استفاده از یک نام واحد برای پروژه، فایل اصلی، و موجودیت اصلی،
- ۲) خودداری از استفاده از نوع BUFFER، در صورت نیاز از سیگنالهای کمکی که در بخش اعلان معماری آنها را معرفی کرده باشید، استفاده نمایید،
- ۳) خودداری از استفاده از بیش از یک جفت موجودیت-معماری در یک کد مفروض.

# فصل سوم

## انواع داده

انواع داده به دو دسته تقسیم می‌شوند: تعریف شده توسط کاربر و از پیش تعریف شده. داده‌های از پیش تعریف شده استاندارد بوده و همراه کامپایلر VHDL عرضه می‌شوند اما داده‌هایی که توسط کاربر تعریف می‌شوند استاندارد نبوده و به فراخور شرایط و نیازها متفاوت هستند. این انواع داده از محلی به محلی و از کاربردی به کاربردی و از کاربری به کاربری دیگر تغییر می‌کنند. از جهت دیگر نیز می‌توان داده‌ها را به دو نوع سنتزپذیر<sup>۱</sup> و سنتزناپذیر تقسیم نمود. نوع سنتزناپذیر عمدتاً در شبیه‌سازی استفاده می‌شوند اما نوع سنتزپذیر، علاوه بر شبیه‌سازی در پیاده‌سازی سخت‌افزاری روی افزارهای PLD نیز قابل استفاده هستند. در این فصل، تاکید عمدۀ روی انواع سنتزپذیر بوده و انواع سنتزناپذیر در فصل ۱۰ بحث و بررسی شده‌اند.

### اشیاء VHDL

یک شیء یک قلم دارای نام و نوع مشخص است که می‌تواند مقداری به خود بگیرد. به عبارت دیگر اشیاء تعیین کننده و متشكل از راه و روشی برای تخصیص مقدار هستند. برخی اشیاء VHDL عبارتند از CONSTANT، SIGNAL، VARIABLE و FILE. حتی اقلام PORT و GENERIC نیز جزو اشیاء VHDL هستند زیرا اولی از نوع SIGNAL و دومی از نوع VARIABLE است. دو نوع SIGNAL و VARIABLE پیچیده‌تر از نوع CONSTANT هستند لذا در این کتاب توجه بیشتری به آنها شده و در چند محل مورد بررسی قرار گرفته‌اند:

- بخش ۲-۳ (حال حاضر): به مفهوم، گرامر و مثالهای مرتبط با اعلان اشیاء پرداخته است.
- بخش ۱-۶: به جمع‌بندی و بیان خلاصه‌ای از ویژگی‌های اصلی این انواع که در کدهای ترتیبی مورد نیاز می‌باشند، پرداخته است.
- فصل ۷: به مقایسه بین SIGNAL و VARIABLE پرداخته و مطالبی راجع به ویژگی‌ها و نحوه‌ی استفاده از آنها بیان می‌کند.

### CONSTANT

<sup>۱</sup> Synthesizable

به نوعی (یا شیءی) گفته می‌شود که مقدار آن قابل تغییر نباشد. گرامر ساده‌ی آن به صورت زیر است:

```
CONSTANT constant_name: constant_type := constant_value;
```

منظور از name هر نامی به جز اسمی رزرو شده است. منظور از type هر نوعی شامل از پیش تعريف شده و تعريف شده توسط کاربر است. منظور از value یک مقدار مشخص (ثابت) یا عبارتی بر حسب مقادیر مشخص می‌باشد.

مثالها:

به نمونه‌های زیر توجه کنید. همیشه باید مقدار یک بیت تنها را داخل یک جفت علامت تک نقل قول<sup>۱</sup> قرار داد. مقداری که شامل چند بیت باشد را باید داخل علامت نقل قول مضاعف<sup>۲</sup> قرار داد.

```
CONSTANT bits: INTEGER := 16;
CONSTANT words: INTEGER := 2**bits;
CONSTANT flag: BIT := '1';
CONSTANT mask: BIT_VECTOR(1 TO 8) := "00001111";
```

در بخش اعلان بخش‌های زیر می‌توان ثابت‌ها را اعلان نمود:

- ENTITY
- ARCHITECTURE
- PACKAGE
- PACKAGE BODY
- BLOCK
- GENERATE
- PROCESS
- FUNCTION
- PROCEDURE

به دو تای آخر، زیر برنامه<sup>۳</sup> نیز گفته می‌شود. بسته به محلی که یک ثابت را اعلان و تعريف می‌کنیم، حوزه‌ی دید آن ثابت، متغیر خواهد بود. اگر ثابتی را داخل یک بسته تعريف کنیم، به معنای واقعی سرتاسری خواهد بود زیرا توسط هر طراحی و پروژه‌ای که از آن بسته استفاده نماید قابل دید و استفاده خواهد بود. اگر داخل یک موجودیت و بعد از PORT تعريف شود تنها توسط

<sup>1</sup> Single Quote

<sup>2</sup> Double Quote

<sup>3</sup> Subprogram

معماری‌هایی که بعد از آن موجودیت آمده‌اند قابل رویت و استفاده خواهد بود. اگر داخل یک معماری (بخش اعلان) تعریف شود تنها در همان معماری قابل دید و استفاده خواهد بود.

### ثابت به تعویق افتاده

اگر ثابتی تعریف شود اما مقدار آن تعیین نشود آن را ثابت به تعویق افتاده<sup>۱</sup> می‌گویند. انجام چنین کاری تنها داخل یک بسته (PACKAGE) مجاز می‌باشد. البته مقدار آن ثابت بعداً در بخش بدنی بسته (PACKAGE BODY) باید حتماً مشخص شود.

### کلمه‌ی کلیدی OTHERS

این کلمه‌ی کلیدی مفید در هنگام تخصیص به کار گرفته شده و بیانگر باقی اندیشهایی است که مشخص نشده‌اند.

### مثالها:

در مثال زیر ثابتی که تعریف می‌شود مقداری برابر "000000" خواهد گرفت:

```
CONSTANT a: BIT_VECTOR(5 DOWNTO 0) := (OTHERS=>'0');
```

در مثال زیر، "01111111" = b خواهد شد:

```
CONSTANT b: BIT_VECTOR(7 DOWNTO 0) := (7=>'0', OTHERS=>'1');
```

در مثال زیر، "01100000" = c خواهد بود:

```
SIGNAL c: STD_LOGIC_VECTOR(1 TO 8) := (2|3=>'1', OTHERS=>'0');
```

در مثال زیر، "1111111100000000" = d خواهد بود:

```
VARIABLE d: BIT_VECTOR(1 TO 16) := (1 TO 8=>'1', OTHERS=>'0');
```

### نوع SIGNAL

نوع داده‌ی SIGNAL جهت حمل و نقل مقادیر به داخل و خارج یک مدار و نیز بین واحدهای داخلی آن مدار استفاده می‌شود. به بیان دیگر، نوع SIGNAL میان اتصالات داخلی یک مدار (یا همان سیم‌ها) می‌باشد. تمام پورت‌های یک موجودیت طبق پیش‌فرض از نوع SIGNAL هستند. در بخش اعلان از محل‌های زیر می‌توان سیگنال‌ها را اعلان نمود:

- ENTITY
- ARCHITECTURE
- BLOCK
- GENERATE

<sup>1</sup> Deferred Constant

در کدهای ترتیبی (زیربرنامه‌ها و PROCESS) نمی‌توان سیگنالی را اعلان نمود اما می‌توان از سیگنال‌ها استفاده نمود.  
گرامر ساده اعلان سیگنال به صورت زیر است:

```
SIGNAL signal_name: signal_type [range] [:= default_value];
```

### مثالها:

در نمونه مثالهای زیر، نام اولین سیگنال enable، نوع آن BIT، و مقدار پیش فرض یا اولیه‌ی آن '0'، می‌باشد. دومین سیگنال temp، نوع آن BIT\_VECTOR، جمعاً دارای ۴ بیت و اندیس بیت‌های آن به ترتیب نزولی مرتب شده‌اند. سیگنال سوم نیز byte بوده، نوع آن STD\_LOGIC\_VECTOR، جمعاً دارای ۸ بیت و اندیس بیت‌های آن نیز به ترتیب نزولی مرتب شده‌اند. بالاخره، چهارمین سیگنال count، نوع آن NATURAL، و بازه‌ی تغییرات مقادیر آن ۰ الی 255 می‌باشد. برای سه سیگنال تعریف شده‌ی آخر، مقداری تعیین نشده است.

```
-----
SIGNAL enable: BIT := '0';
SIGNAL temp: BIT_VECTOR(3 DOWNTO 0);
SIGNAL byte: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL count: NATURAL RANGE 0 TO 255;
-----
```

یک نکته‌ی بسیار مهم در ارتباط با نوع SIGNAL این است که اگر از این نوع در بخشی از یک ترتیبی استفاده شود (یعنی زیربرنامه یا PROCESS)، به روزرسانی آن فوری نمی‌باشد. مقدار جدید تنها در پایان اجرای دور فعلی از زیربرنامه یا PROCESS مهیا و قابل استفاده خواهد بود. یک نکته مهم دیگر تخصیص همزمان چند مقدار به یک سیگنال واحد است. اگر داخل کدهای همزمان<sup>۱</sup> این تخصیصهای همزمان انجام شده باشد، کامپایلر پیغام خطأ خواهد داد. اگر این کار داخل یک کد ترتیب انجام شده باشد، آخرین تخصیص ملاک عمل خواهد بود. البته در حالت کلی باید از انجام چنین کاری اجتناب نمود.

برای تخصیص مقدار به یک سیگنال از عملگر => و برای تخصیص مقدار به CONSTANT و یا VARIABLE از عملگر := استفاده می‌شود. همان طور که قبلاً دیدیم، در انجام تخصیص، کلمه‌ی کلیدی OTHRES بسیار مفیدی خواهد بود.

<sup>۱</sup> Concurrent

**VARIABLE** نوع

در مقابل دو نوع سیگنال (SIGNAL) و ثابت (CONSTANT)، نوع متغیر (VARIABLE) تنها قادر به نمایش اطلاعات محلی<sup>۱</sup> است زیرا این نوع را تنها میتوان در داخل همان واحد ترتیبی (یعنی زیربرنامه‌ها و یا PROCESS) که متغیر را داخل آن ایجاد کرده‌ایم، استفاده کرده و یا تغییر داد. البته، به روزرسانی آن فوری بوده و از مقدار جدید در خط کد بعدی می‌توان استفاده نمود. همچنین به دلیل به روزرسانی فوری، تخصیص‌های چندگانه روی متغیرها مجاز است. گرامر ساده‌ی اعلان متغیر به صورت زیر می‌باشد:

```
VARIABLE variable_name: variable_type [range] [:= default_value];
```

مثال:

```
VARIABLE flip: STD_LOGIC := '1';
VARIABLE address: STD_LOGIC_VECTOR(0 TO 15);
VARIABLE counter: INTEGER RANGE 0 TO 127;
```

برای تخصیص مقدار به یک متغیر از عملگر = استفاده می‌شود. در این کار میتوان از عملگر مفید OTHERS استفاده نمود.

**VARIABLE** و **SIGNAL** مقایسه

این مقایسه در فصل ۷ با جزئیات خود بحث و بررسی شده است.

**۳-۳ کتابخانه‌ها و بسته‌های مرتبط با انواع داده**

در VHDL انواع مختلف داده در بسته‌های مختلفی تعریف شده‌اند لذا دانستن و شناختن این بسته‌ها جهت استفاده از انواع مناسب داده‌ها ضروری است. لیست بسته‌های اساسی مربوط به داده‌های از نوع باینری منطقی و صحیح به شرح زیر است:

<sup>۱</sup> Local

- Package *standard* (expanded in VHDL 2008)
- Package *std\_logic\_1164* (expanded in VHDL 2008)
- Package *numeric\_bit* (expanded in VHDL 2008)
- Package *numeric\_std* (expanded in VHDL 2008)
- Package *std\_logic\_arith* (shareware, nonstandard)
- Package *std\_logic\_unsigned* (shareware, nonstandard)
- Package *std\_logic\_signed* (shareware, nonstandard)
- Package *textio* (expanded in VHDL 2008)
- Package *numeric\_bit\_unsigned* (introduced in VHDL 2008)
- Package *numeric\_std\_unsigned* (introduced in VHDL 2008)

در نسخه VHDL 2008 بسته‌های جدیدی جهت کار با اعداد ممیز ثابت و ممیز شناور ایجاد و معرفی شده است. مهمترین این بسته‌ها عبارتند از:

- Package *fixed\_pkg*
- Package *fixed\_generic\_pkg*
- Package *float\_pkg*
- Package *float\_generic\_pkg*
- Package *fixed\_float\_types*

توصیف مختصری از بسته‌های اشاره شده در بالا در ادامه ارائه خواهد شد:

#### بسته *standard* (پیوست H)

این بسته انواع داده زیر را تعریف می‌کند:

- Bit-related (synthesizable): BIT, BIT\_VECTOR, BOOLEAN
- Integer-related (synthesizable): INTEGER, NATURAL, POSITIVE
- Character-related (synthesizable): CHARACTER, STRING
- Floating-point (limited synthesis support): REAL
- Time-related (not for synthesis): TIME, DELAY\_LENGTH
- File-related (not for synthesis): FILE\_OPEN\_KIND, FILE\_OPEN\_STATUS
- Communication with the compiler: SEVERITY\_LEVEL

علاوه بر این، بسته فوق شامل تعاریف عملگرهای منطقی، حسابی، شیفت، مقایسه و تجمعی روی عملوندهایی از انواع مشخص شده در بالا می‌باشد.

در نسخه VHDL 2008 موارد زیر نیز به بسته‌ی فوق افزوده شده است:

- ۱) انواع جدید داده شامل `INTEGER_VECTOR`, `BOOLEAN_VECTOR`, `TIME_VECTOR` و `REAL_VECTOR`
- ۲) عملگرهای ریاضی شامل `=?`, `?<=`, `?<`, `?/=?`, `?>` و `?>=`
- ۳) دیگر توابع نظیر `??`, `RISING_EDGE`, `MAXIMUM`, `MINIMUM`, `TO_HSTRING`, `TO_OSTRING`, `TO_STRING`, `FALLING_EDGE`

#### بسته‌ی `std_logic_1164` (پیوست I)

مهمترین انواع داده تعریف شده در این بسته عبارتند از:

- `STD_ULOGIC`, `STD_ULOGIC_VECTOR`
- `STD_LOGIC`, `STD_LOGIC_VECTOR` (industry standard)

این بسته همچنین شامل تعریف عملگرهای لازم جهت کار روی عملوندھایی از انواع فوق می‌باشد. به علاوه، برخی توابع تبدیل نوع<sup>۱</sup> مانند `TO_BIT_VECTOR`, `TO_STDLOGIC_VECTOR` در این بسته تعریف شده‌اند. در نسخه‌ی VHDL 2008 تغییرات زیر اعمال یا جدیداً افزوده شده‌اند:

- نوع `STD_ULOGIC_VECTOR` هم‌اکنون زیرنوعی از `STD_LOGIC_VECTOR` شده است. بنابراین، عملگرهایی که برای نوع دوم تعریف شده‌اند به طور خودکار روی نوع اول نیز سربارگذاری (یا تعمیم داده) شده‌اند.
- گزینه‌های بیشتری جهت عملگرهای منطقی مهیا و شامل شده است.
- برخی عملگرهای شیفت اضافه شده‌اند.
- اضافه شدن برخی عملگرهای مقایسه و بررسی انتباط شامل `?=`, `?<=`, `?<`, `?/=?`, `?>` و `?>=`
- افزوده شدن شرط `(??)`, تبدیل به رشتہ `TO_OSTRING`, `TO_STRING`, `READ/WRITE` (TO\_HSTRING), و عملگرهای `.TO_STDLOGIC_VECTOR` و `TO_SLV`
- افزوده شدن چندین میانبر نظیر `TO_HSTRING`, `TO_OSTRING`, `TO_STRING`, `READ/WRITE` و `TO_SLV`

#### بسته‌ی `numeric_bit`

این بسته بخشی از کتابخانه‌ی ieee بوده و مشابه با بسته‌ی بعدی است که توضیح داده می‌شود با این تفاوت که به جای `STD_LOGIC` از نوع داده `BIT` استفاده کرده است.

#### بسته‌ی `numeric_std` (پیوست J)

در اولین نسخه‌ی VHDL بوده اما در نسخه‌ی VHDL 2008 اضافاتی به آن افزوده شد. دو نوع داده‌ی زیر در این بسته تعریف شده است:

<sup>۱</sup> Type Conversion

- نوع UNSIGNED (برپایه‌ی STD\_LOGIC)

- نوع SIGNED (این هم برپایه‌ی STD\_LOGIC)

این بسته همچنین شامل عملگرهای منطقی، حسابی، شیفت، و مقایسه که برای کار با انواع فوق الذکر لازم هستند، می‌باشد. برخی توابع تبدیل نوع مانند TO\_INTEGER و TO\_UNSIGNED نیز در این بسته گنجانده شده‌اند.

تغییرات زیر در نسخه‌ی 2008 VHDL به این بسته افزوده شده است:

- تعریف UNSIGNED و SIGNED اندکی تغییر یافته است،
- گزینه‌های بیشتری برای عملگر حسابی (ریاضی) درنظر گرفته شده است،
- گزینه‌های بیشتری برای عملگر منطقی درنظر گرفته شده است،
- گزینه‌های بیشتری برای عملگر شیفت درنظر گرفته شده است،
- اضافه شدن برخی عملگرهای مقایسه و بررسی انتباطی شامل =، /=، <، >، ?= و ?<، ?>، ?<=، ?>= و ?>=،
- اضافه شدن توابع TO\_OSTRING، TO\_STRING، MAXIMUM، MINIMUM، WRITE و READ، TO\_HSTRING

#### بسته‌ی std\_logic\_arith

شامل تعریف انواع زیر به همراه عملگرهای حسابی، شیفت و مقایسه (منطقی شامل نیست) برای کار با این انواع است:

- نوع UNSIGNED (برپایه‌ی STD\_LOGIC)

- نوع SIGNED (برپایه‌ی همان نوع STD\_LOGIC)

برخی توابع تبدیل نوع نظیر CONV\_STD\_LOGIC\_VECTOR و CONV\_INTEGER نیز در این بسته تعریف شده‌اند. این بسته تنها تا حدی مشابه با بسته‌ی numeric\_std می‌باشد.

#### بسته‌ی std\_logic\_unsigned

این بسته کاملاً مشابه با بسته بعدی است تنها با این تفاوت که به جای نوع SIGNED از عملگرهای مبتنی بر نوع UNSIGNED استفاده شده است.

#### بسته‌ی std\_logic\_signed

این بسته شامل تعریف برخی عملگرهای حسابی، مقایسه و شیفت جهت کار با سیگنال‌هایی از نوع STD\_LOGIC\_VECTOR که در نقش اعدادی از نوع SIGNED کار می‌کنند، می‌باشد. این بسته همچنین شامل تابع تبدیل نوع CONV\_INTEGER می‌باشد.

#### بسته‌ی numeric\_bit\_unsigned

این بسته در نسخه VHDL 2008 معرفی شد. اساساً مشابه با بستهٔ بعدی است که در ادامه تعریف می‌شود با این تفاوت که نوع پایه‌ای در آن BIT و BIT\_VECTOR است (بستهٔ بعدی مبتنی بر نوع پایه‌ای STD\_LOGIC\_VECTOR و STD\_LOGIC است).

### بستهٔ numeric\_std\_unsigned

این بسته نیز در VHDL 2008 معرفی شده است. انتظار می‌رود که در آینده این بسته جایگزین بستهٔ غیراستاندارد std\_logic\_unsigned شود. این بسته شامل مجموعه‌ی وسیعی از عملگرهای حسابی، مقایسه، و شیفت بوده و علاوه بر این شامل توابعی جهت تبدیل نوع برای انواع STD\_LOGIC\_VECTOR و STD\_LOGIC می‌باشد (پیوست N ملاحظه شود).

## ۵-۳ نوع‌های استاندارد داده‌ای

در این بخش انواع داده‌ای قابل سنتز متعلق به بستهٔ standard (پیوست H) بررسی خواهند شد:

- BIT
- BIT\_VECTOR
- BOOLEAN
- BOOLEAN\_VECTOR (2008)
- INTEGER
- NATURAL
- POSITIVE
- INTEGER\_VECTOR (2008)
- CHARACTER
- STRING

همراه این انواع، عملگرهای مربوطه و توابع تبدیل نوع مربوطه نیز تعریف شده‌اند که در فصلهای بعدی بررسی خواهند شد. به طور خلاصه لیستی از عملگرها به صورت زیر ارائه شده‌اند:

Logical operators: NOT, AND, NAND, OR, NOR, XOR, XNOR (section 4.2.2)

Arithmetic operators: +, -, \*, /, \*\*, ABS, REM, MOD (section 4.2.3)

Comparison (relational) operators: =, /=, >, >=, <, <= (section 4.2.4)

Shift operators: SLL, SRL, SLA, SRA, ROR, ROL (section 4.2.5)

Concatenation operator: & (section 4.2.6)

Matching comparison operators: ?=, ?/=, ?>, ?>=, ?<, ?<= (section 4.2.7, from VHDL 2008).

**نوع BIT**

یک نوع شمارش‌پذیر دومقداره<sup>۱</sup> است. این نوع از عملیات‌های منطقی و مقایسه پشتیبانی می‌کند (پیوست H). این نوع، از نظر تعداد بیت‌ها، یک نوع اسکالار محسوب می‌شود:

```
TYPE BIT IS ('0', '1');
```

مثال: در تکه کد زیر، اشیاء a، x و y به صورت سیگنال‌هایی از نوع BIT تعریف شده‌اند. سپس مقادیری به هر کدام تخصیص داده شده است.

```
-----  
SIGNAL a, x, y: BIT;  
x <= '1';  
y <= NOT a;  
-----
```

در VHDL 2008 عملگرهای زیر برای کار با این نوع افزوده شده‌اند:  
MINIMUM, MAXIMUM, RISING\_EDGE, FALLING\_EDGE, TO\_STRING.  
?=, ?/=, ?<, ?<=, ?>, ?>=, ??,

**نوع BIT\_VECTOR**

این نوع حالت برداری نوع BIT است؛ به عبارت دیگری آرایه‌ای یک بعدی از عناصری از نوع BIT می‌باشد. در این نوع از عملیات منطقی، مقایسه، شیفت و تجمعیع پشتیبانی می‌شود.

```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT;
```

مشخص‌گر <> NATURAL RANGE (به نماد <> جعبه<sup>۲</sup> گفته می‌شود) در عبارت فوق به معنای نامحدود بودن بازه است. تنها محدودیت این است که این بازه در محدوده NATURAL باشد (بازه‌ی پیش‌فرض عبارت است از [0, 2<sup>31</sup>-1]). در عملیات منطقی و شیفت، لازم است دو بردار طول یکسانی داشته باشند اما در عملیات مقایسه چنین شرطی الزامی نیست.

مثال: در تکه کد زیر، منظور از عملگر SLL<sup>۳</sup> شیفت منطقی به چپ است. توجه کنید که بیت شماره‌ی 7 از بردار x برابر '1' و بیت شماره صفر برابر '0' قرار داده می‌شود.

<sup>1</sup> Two-Value Enumerated

<sup>2</sup> Box

<sup>3</sup> Shift Left Logical

```

-----  

SIGNAL a, b: BIT_VECTOR(7 DOWNTO 0); --8 bits  

SIGNAL x, y: BIT_VECTOR(7 DOWNTO 0); --8 bits  

SIGNAL v: BIT_VECTOR(1 TO 8); --8 bits  

SIGNAL w: BIT; --1 bit  

x <= "11110000";  

y <= a XOR b;  

v <= a SLL 2;  

w <= '1' WHEN a>b ELSE '0';
-----
```

در نسخه VHDL 2008 عملگرهای زیر افزوده شده‌اند:  
 $\neq$ ,  $\neq$ , MINIMUM, MAXIMUM, TO\_STRING, TO\_OSTRING, TO\_HSTRING.

### نوع BOOLEAN

تعریف این نوع به صورت زیر است. این نوع هم یک نوع شمارش شده دو مقداره است. این نوع از عملیات منطقی و مقایسه (پیوست H) پشتیبانی می‌کند. از نظر تعداد بیت‌ها، این نوع یک نوع اسکالر محسوب می‌شود.

```
TYPE BOOLEAN IS (FALSE, TRUE);
```

مثال: در مثال زیر بر حسب این که مقدار ready برابر TRUE باشد یا خیر، x مقدار '111' یا '000' خواهد گرفت:

```
x<="111" WHEN ready ELSE "000";
```

در نسخه VHDL 2008 توابع زیر افزوده شده‌اند:  
MINIMUM, MAXIMUM, RISING\_EDGE, FALLING\_EDGE, TO\_STRING.

### نوع BOOLEAN\_VECTOR

این نوع در نسخه 2008 معرفی و افزوده شده است. این نوع، حالت برداری نوع BOOLEAN است. تعریف آن به صورت زیر است. این نوع باید از عملیات منطقی، مقایسه، شیفت و تجمعی به همراه عملگرهای  $\neq$ ,  $\neq$ , MINIMUM, and MAXIMUM پشتیبانی کند (پیوست H).

```
TYPE BOOLEAN_VECTOR IS ARRAY (NATURAL RANGE <>) OF BOOLEAN;
```

### نوع INTEGER

تعریف این نوع به صورت زیر است. در این نوع از عملیات حسابی و مقایسه پشتیبانی می‌شود.

```
TYPE INTEGER IS RANGE implementation_defined;
TYPE INTEGER IS RANGE -2147483647 TO 2147483647;
```

آخرین خط تعاریف فوق، محدوده‌ی پیشفرض نوع INTEGER را نشان می‌دهد. این نمایش مشخص کننده‌ی یک نمایش ۳۲ بیتی است یعنی از  $(-1)^{2^{31}}\text{--}2^{31}$ . برای دسترسی به محدوده‌ی واقعی می‌توان از مشخص‌گر INTEGER'LOW (سمت چپ) و INTEGER'HIGH (سمت چپ) و (سمت راست) استفاده کرد.

در کدهای VHDL که سنترپذیر باشند، لازم است در نوع INTEGER (یا زیرنوع‌های آن) همواره بازه‌ی اشیاء مشخص شود زیرا در غیر اینصورت، کامپایلر از بازه‌ی ۳۲ بیتی که اشاره شد، استفاده خواهد کرد.

مثال: در تکه کد زیر، ابتدا چهار سیگنال از نوع INTEGER تعریف شده و سپس دو تابع آنها با هم جمع و سپس مقایسه می‌شوند. در هر کدام از این دو عملیات تعداد بیت‌های دو سیگنال لازم نیست با هم برابر باشد.

```
-----
SIGNAL a: INTEGER RANGE 0 TO 15;      --4 bits
SIGNAL b: INTEGER RANGE -15 TO 15;    --5 bits
SIGNAL x: INTEGER RANGE -31 TO 31;    --6 bits
SIGNAL y: BIT;
x <= a + b;
y <= '1' WHEN a>=b ELSE '0';
-----
```

در VHDL 2008 توابع زیر افزوده شده‌اند:

MINIMUM, MAXIMUM, TO\_STRING.

### نوع NATURAL

این نوع شامل اعداد صحیح نامنفی است. همان طور که در تعریف زیر نیز مشخص شده است، این نوع یک زیرنوع نسبت به نوع INTEGER می‌باشد. بنابراین دارای ابعادی برابر با همان ابعاد نوع INTEGER بوده و از همان عملگرها نیز استفاده می‌کند.

```
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH;
```

### نوع POSITIVE

این نوع شامل اعداد صحیح مثبت می‌شود. همان طور که در تعریف زیر نیز مشخص شده است، این نوع یک زیرنوع نسبت به نوع INTEGER می‌باشد. بنابراین دارای ابعادی برابر با همان ابعاد نوع INTEGER بوده و از همان عملگرها نیز استفاده می‌کند.

**SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;**

### نوع INTEGER\_VECTOR

این نوع در VHDL 2008 معرفی شده و حالت برداری نوع INTEGER محسوب می‌شود. در این نوع باید از عملیات مقایسه و تجمعیع به همراه توابع MAXIMUM و MINIMUM و پشتیبانی شود (پیوست H). تعریف این نوع به صورت زیر است:

**TYPE INTEGER\_VECTOR IS ARRAY (NATURAL RANGE <>) OF INTEGER;**

### نوع CHARACTER

این نوع یک نوع شمارش شده دارای ۲۵۶ نماد<sup>۱</sup> یا نشانه است. تعریف این نوع به صورت زیر است. در این نوع فقط عمل مقایسه پشتیبانی شده است (پیوست H).

**TYPE CHARACTER IS (NUL, SOH, ..., '0', '1', '2', ..., 'ÿ');**

مثال:

```
SIGNAL char1, char2: CHARACTER;
SIGNAL outp1, outp2: BIT;
outp1 <= '1' WHEN char1='a' OR char1='A' ELSE '0';
outp2 <= '1' WHEN char1<char2 ELSE '0';
```

در VHDL 2008 توابع MAXIMUM، MINIMUM و TO\_STRING افزوده شده‌اند.

### نوع STRING

این نوع حالت برداری نوع CHARACTER است. تعریف این نوع در زیر نشان داده شده است. این نوع از اعمال مقایسه و تجمعیع پشتیبانی می‌کند (پیوست H).

**TYPE STRING IS ARRAY (POSITIVE RANGE <>) OF CHARACTER;**

<sup>۱</sup> Symbol

مثال: در تکه کد زیر str یک رشته شامل چهار کاراکتر تعریف شده است؛ بنابراین، ابعاد آن ۴ در ۸ بیت است.

---

```
SIGNAL str: STRING(1 TO 4);
SIGNAL output: BIT;
output <= '1' WHEN str="VHDL" ELSE '0';
```

---

در VHDL 2008 توابع MAXIMUM و MINIMUM افزوده شده‌اند.

توجه: دو نوع بعدی که معرفی می‌شوند قابلیت سنتز ندارند اما به جهت اهمیت به طور مختصر معرفی شده‌اند. نوع REAL در سنتز شدن محدودیت دارد. نوع TIME نیز تنها برای شبیه‌سازی قابل استفاده است (در فصل ۱۰ استفاده خواهد شد).

### نوع REAL

شامل اعداد ممیز شناور می‌باشد. در این نوع از عملیات حسابی و مقایسه پشتیبانی شده است. تعریف آن به صورت زیر است:

```
TYPE real IS RANGE implementation_defined;
```

در VHDL 2008 توابع MAXIMUM، MINIMUM و TO\_STRING افزوده شده‌اند.

### نوع REAL\_VECTOR

در نسخه VHDL 2008 معرفی شده و حالت برداری نوع REAL است (پیوست H). تعریف آن به صورت زیر است:

```
TYPE REAL_VECTOR IS ARRAY (NATURAL RANGE <>) OF REAL;
```

### نوع TIME

این نوع با اعداد صحیحی در همان بازه‌ی نوع INTEGER نمایش داده می‌شود. در این نوع از عملیات حسابی و مقایسه پشتیبانی شده است. تعریف آن به صورت زیر بوده و در شبیه‌سازی استفاده می‌شود:

```
TYPE time IS RANGE implementation_defined;
```

در VHDL 2008 توابع MAXIMUM، MINIMUM و TO\_STRING افزوده شده‌اند.

### نوع TIME\_VECTOR

در نسخه 2008 VHDL معرفی شده و حالت برداری نوع TIME است (پیوست H). تعریف آن به صورت زیر است:

```
TYPE TIME_VECTOR IS ARRAY (NATURAL RANGE <>) OF TIME;
```

### ۶-۳ نوعهای استاندارد منطقی

در این بخش دو نوع STD\_LOGIC\_VECTOR و STD\_LOGIC را که دو نوع استاندارد صنعتی هستند، تعریف می‌کنیم. این انواع در استاندارد اولیه 93 VHDL در بسته‌ی std\_logic\_1164 تعریف شده بودند. در 2008 VHDL ویژگی‌های جدیدی به آن افزوده شد. در پیوست I هر دو نسخه 1993 و 2008 از این بسته نمایش داده شده است.

دو نوع STD\_LOGIC و STD\_ULOGIC به ترتیب به «تعیین تکلیف نشده» (Unresolved) و «تعیین تکلیف شده» (Resolved) اشاره دارند. تعریف این دو نوع به صورت زیر است. توجه کنید که این انواع نیز نظیر انواع BIT، BOOLEAN و CHARACTER از نوع شمارش‌پذیر هستند.

```
TYPE STD_ULOGIC IS ('U','X','0','1','Z','W','L','H','-');
TYPE STD_LOGIC IS resolved STD_ULOGIC;
```

معنا و استفاده‌ی هر یک از ۹ مقدار ذکر شده در تعریف بالا به صورت زیر است:

- 'U' Uninitialized
- 'X' Forcing unknown
- '0' Forcing low
- '1' Forcing high
- 'Z' High impedance
- 'W' Weak unknown
- 'L' Weak low
- 'H' Weak high
- '-' Don't care

تعریف شکل برداری دو نوع فوق به صورت زیر است:

```
TYPE STD_ULOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_ULOGIC;
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

مهمترین ویژگی و تفاوت نوع STD\_LOGIC در مقایسه با نوع BIT شامل شدن دو مقدار امپدانس بالا ('Z') و بدون اهمیت ('-') می‌باشد. در نظر گرفتن این مقادیر موجب می‌شود بتوانیم با فر سه حالت ساخته (اولین مقدار) و جدول‌های درستی را به صورت بهتری از نظر سخت‌افزاری بهینه‌سازی کنیم (دومین مقدار).

بسته‌ی 1164 std\_logic تنها شامل تعریف عملگرهای منطقی روی این انواع می‌باشد اما اگر بسته‌های std\_logic\_(un)signed را نیز در کد خود اعلان کنیم، می‌توانیم از برخی عملیات حسابی، مقایسه و شیفت نیز روی این انواع استفاده کنیم.

به این دلیل نوع STD\_LOGIC را «تعیین تکلیف شده» می‌نامیم که اگر چند منبع، یک گره را درایو (مقداردهی) کنند، مقدار نهایی یا «تکلیف» آن گره توسط یکتابع از پیش تعریف شده مشخص خواهد شد. این تابع روشن‌کننده‌ی تکلیف برای نوع STD\_LOGIC به صورت زیر است. این خطوط از بدنه‌ی بسته‌ی std\_logic\_1164 (PACKAGE BODY) عیناً اینجا نوشته شده است.

```

1 -----
2 CONSTANT resolution_table : stdlogic_table := (
3   -----
4   -- U   X   0   1   Z   W   L   H   -   |   |
5   -----
6   ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', -- | U |
7   ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', -- | X |
8   ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X', -- | 0 |
9   ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X', -- | 1 |
10  ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X', -- | Z |
11  ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X', -- | W |
12  ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X', -- | L |
13  ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), -- | H |
14  ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X') -- | - | );
15 -----
16 FUNCTION resolved (s: STD_ULOGIC_VECTOR) RETURN STD_ULOGIC IS
17   VARIABLE result: STD_ULOGIC_ := 'Z';  --weakest state default
18   ATTRIBUTE synthesis_return OF result: VARIABLE IS "WIRED_THREE_STATE";
19 BEGIN
20   IF (s'LENGTH=1) THEN RETURN s(s'LOW);
21   ELSE
22     FOR i IN s'RANGE LOOP
23       result := resolution_table(result, s(i));
24     END LOOP;
25   END IF;
26   RETURN result;
27 END resolved;
28 -----

```

از نظر سنتز نکته مفیدی که لازم به توجه است این است که هر یک از ۹ مقدار اشاره شده در هنگام سنتز به مقداری که به صورت زیر بیان شده، سنتز می‌شوند:

- '0' and 'L' are both synthesized as '0' (for inputs and outputs);
- '1' and 'H' are both synthesized as '1' (for inputs and outputs);
- 'Z' is synthesized as 'Z' (for outputs);
- The others as synthesized as '-' (*don't care*—for outputs).

یک نکته‌ی دیگر این است که در 2008 VHDL نوع STD\_LOGIC\_VECTOR زیرنوعی از نوع STD\_ULOGIC\_VECTOR تعریف شده است. یعنی:

```
TYPE STD_ULOGIC_VECTOR IS array (NATURAL RANGE <>) of STD_ULOGIC;
SUBTYPE STD_LOGIC_VECTOR IS (resolved) STD_ULOGIC_VECTOR;
```

لذا عملگرهایی که برای نوع دوم تعریف شده‌اند، به طور خودکار برای نوع اول سربارگذاری<sup>۱</sup> می‌شوند.

تغییر دیگری که در 2008 VHDL اضافه شده است، بسته‌ی numeric\_std\_unsigned است که برخی عملگرهای بدون علامت را برای انواع STD\_LOGIC و STD\_LOGIC\_VECTOR تعریف کرده است. (به طور مشابه این عملگرها در بسته‌ی numeric\_bit\_unsigned برای دو نوع BIT\_VECTOR و BIT تعریف شده‌اند).

فواید استفاده از نوع STD\_LOGIC در مثالهای بعدی نشان داده شده است.

### مثال ۳-۱: بافر سه حالت

همان طور که اشاره شد، یک مقدار بسیار مهم که سنتزپذیر نیز می‌باشد، مقدار امپدانس بالا ('Z') است که در ساخت بافرهای سه حالت کاربرد دارد. یک بافر سه حالتی به همراه جدول صحت آن در شکل زیر نشان داده شده است. کد VHDL برای توصیف این مدار را بنویسید.



Figure 3.3  
Tri-state buffer of example 3.1.

پاسخ: در کد زیر، توجه کنید که استفاده از بسته‌ی std\_logic\_1164 ضروری است زیرا مقدار 'Z' را تعریف کرده است.

<sup>1</sup> Overloading

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY tri_state IS
6 PORT (input, ena: IN STD_LOGIC;
7         output: OUT STD_LOGIC);
8 END ENTITY;
9 -----
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12     output <= input WHEN ena='1' ELSE 'Z';
13 END ARCHITECTURE;
14 -----

```

مثال ۲-۳: مدار با خروجی‌های بدون اهمیت در شکل ۳-۴-الف مداری نشان داده شده است که ورودی (x) و خروجی (y) آن سیگنالهایی دو بیتی هستند. در شکل‌های ۳-۴-ب و ۳-۴-ج دو دسته مشخصات (جدول صحت) نشان داده شده است. در اولین جدول صحت از مقادیر '0' و '1' برای نمایش مقادیر خروجی استفاده شده است اما در دومین جدول صحت از مقدار بدون اهمیت ("--") نیز استفاده شده است. این مدار را دستی طراحی کرده و سپس برای هر حالت مدار (جدول صحت) یک کد VHDL بنویسید تا اثر استفاده از مقدار منطقی '-' را در کد خود ببینید.

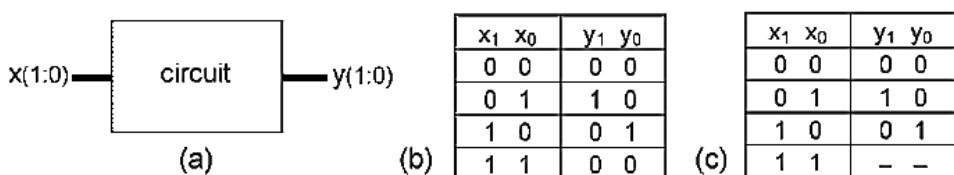


Figure 3.4  
Circuit with “don’t care” outputs of example 3.2.

پاسخ:

معادلات بهینه خروجی y برای مدار شکل (الف) با استفاده از جدول ساده‌سازی کارنو چنین خواهد بود:

$$y_1 = x'_1 \cdot x_0, \quad y_0 = x_1 \cdot x'_0$$

برای مدار شکل (ب) نیز چنین خواهیم داشت:

$$y_1 = x_0, \quad y_0 = x_1$$

با مقایسه معادلات مدارهای (الف) و (ب) ملاحظه می‌شود که در مدار دوم، سخت‌افزار ساده‌تری استفاده شده است (بنابراین هزینه کمتر، توان مصرفی کمتر و سرعت مدار بالاتر خواهد بود).

کد VHDL متناظر با مدار (الف) در زیر نشان داده شده است. برای رسیدن به کد مدار (ب) تنها تغییری که باید داده شود مربوط به خط شماره ۱۵ است.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY circuit IS
6     PORT (x: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
7             y: OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
8 END ENTITY;
9 -----
10 ARCHITECTURE circuit OF circuit IS
11 BEGIN
12     y <= "00" WHEN x="00" ELSE
13         "01" WHEN x="10" ELSE
14         "10" WHEN x="01" ELSE
15         "--";
16 END ARCHITECTURE;
17 -----

```

### ۷-۳ نوعهای داده‌ای علامتدار و بدون علامت

همان طور که قبلاً هم اشاره شد، نوعهای داده‌ای SIGNED و UNSIGNED در دو بسته‌ی جداگانه (و رقیب) numeric\_std (متعلق به کتابخانه‌ی ieee، پیوست J) و std\_logic\_arith (محصول شرکت‌های دیگر، پیوست K) تعریف و معروفی شده‌اند. تعاریف این دو به صورت زیر است:

```
TYPE UNSIGNED IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
TYPE SIGNED IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

بنابراین برای استفاده از دو نوع SIGNED و UNSIGNED، باید یکی از دو بسته‌ی اشاره شده را در کد خود اعلام کنیم. این نکته را هم به یاد داشته باشید که این دو بسته تاحدی (ونه کاملاً) شبیه به هم هستند. بسته‌ی numeric\_std شامل تعریف عملگرهای منطقی، حسابی، مقایسه و شیفت بوده حال آن که بسته‌ی std\_logic\_arith شامل تعریف عملگرهای منطقی نمی‌باشد. همچنین بسته‌ی دوم شامل دسته‌ی وسیع‌تری از توابع تبدیل نوع می‌باشد.

واضح است که در یک کد VHDL نمی‌توان همزمان هر دو بسته‌ی نام بردۀ شده را اعلام و استفاده نمود. همچنین با توجه به این که بسته‌ی numeric\_std یک بسته‌ی استاندارد (متعلق به سازمان IEEE) است، در اولویت بالاتری جهت استفاده کردن قرار دارد و بهتر است از آن استفاده شود.

یک مقدار بدون علامت در بازه‌ی  $0 \text{ تا } 2^N - 1$  (محدود به INTEGER'HIGH) قرار دارد که در آن  $N$  تعداد بیت‌ها است. بنابراین برای مثال، مقدار "0101" معادل با ۵ دهدهی و "1101" معادل با ۱۳ دهدهی است. در مقابل، نوع علامتدار در بازه‌ی  $2^{N-1} - 1 \text{ تا } 2^N - 1$  (محدود بین "0101" و INTEGER'LOW) قرار دارد. بنابراین در این حالت، مقدار "0101" همچنان معادل با ۵ دهدهی است اما مقدار "1101" معادل با ۳ دهدهی است.

متاسفانه نامی که برای دو بسته‌ی std\_logic\_signed و std\_logic\_unsigned (پیوست L) در نظر گرفته شده است، ممکن است موجب گمراحتی و سردرگمی شود. این دو بسته، تنها «عملگرهای» لازم برای کار با عملوندهای از نوع STD\_LOGIC\_VECTOR را تعریف کرده‌اند نه دو «نوع» UNSIGNED و SIGNED را.

مثال: در کد زیر، سیگنال‌های a، b، x و y همگی از نوع SIGNED و دارای عرض ۸ بیت تعریف شده‌اند. اندیس این سیگنال‌ها به ترتیب نزولی تعریف شده است. اما سیگنال‌های v و w با عرض ۱ بیت و از نوع به ترتیب STD\_LOGIC و BIT تعریف شده‌اند. سه عملیاتی که در ادامه آمده است معتبر و مجاز است اما عملیات چهارم غیرمجاز است. عملیات  $x = a + b$  مجاز است زیرا عملگر حسابی + روی نوع INTEGER تعریف شده است. از طرفی عبارتی که برای محاسبه‌ی y استفاده شده به شرطی مجاز و معتبر است که از بسته‌ی numeric\_std استفاده شده باشد زیرا بسته‌ی std\_logic\_arith شامل تعریف عملگر منطقی نمی‌باشد. عملیات مربوط به محاسبه‌ی v نیز معتبر است زیرا یک اسکالر (تک بیتی) از نوع STD\_LOGIC با اسکالری از نوع SIGNED (که دارای همان نوع پایه‌ای STD\_LOGIC است) سازگار است. در نهایت این که عبارت مربوط به محاسبه‌ی w نامعتبر است زیرا نوع پایه‌ای SIGNED با نوع BIT یکسان (سازگار) نیست.

```

-----  

SIGNAL a, b: SIGNED(7 DOWNTO 0);  

SIGNAL x, y: SIGNED(7 DOWNTO 0);  

SIGNAL v: STD_LOGIC;  

SIGNAL w: BIT;  

x <= a + b;           --legal  

y <= a AND b;         --legal only if numeric_std is used  

v <= a(7) XOR b(0);  --legal because of same base type  

w <= a(0);            --illegal because of type mismatch  

-----
```

مثال ۳-۳: ضرب‌کننده‌ی علامتدار/بدون علامت شماره ۱  
یک کد VHDL برای توصیف مداری که قادر به انجام ضرب  $y = a * b$  باشد، بنویسید. به یاد داشته باشید که در صورت استفاده از دو نوع UN(SIGNED) برای عملوندهای ورودی، تعداد بیت‌های

- خروجی حاصلضرب باید برابر با مجموع تعداد بیت‌های عملوندهای ورودی باشد (در بخش ۵ ۷ مجددًا اشاره خواهد شد).

(الف) تمام سیگنال‌ها را به صورت UNSIGNED اعلان کنید که در آن برای a و b تعداد ۴ بیت و برای y تعداد ۸ بیت در نظر گرفته باشید. کد خود را کامپایل کرده و آن را برای حالت "1101" a= و ابتدا "0010" b= و سپس "1110" b= شبیه‌سازی کنید. نتایج را مشاهده و تفسیر کنید.

(ب) طراحی را مجددًا برای استفاده از سیگنال‌هایی از نوع SIGNED تکرار کنید.

### پاسخ:

(الف) کد مربوطه در شکل زیر نشان داده شده است. از بسته‌ی numeric\_std (خطوط ۲ و ۳) استفاده شده است. تمامی سیگنال‌ها از نوع UNSIGNED تعریف شده‌اند (خطوط ۶ و ۷). در این حالت مقادیر دسیمال (دهدهی) به صورت a=13 و b=2 و پس از آن b=14 می‌باشند. در اثر این مقادیر مقدار خورجی به صورت y="10110110" و سپس y="182" و سپس y="00011010" خواهد شد.

(ب) تنها تغییراتی که در این حالت باید نسبت به حالت قبل ایجاد کنیم مربوط به خطوط ۶ و ۷ می‌شود (باید به جای نوع UNSIGNED از نوع SIGNED استفاده کنیم). در این حالت مقادیر دهدهی ورودی به صورت a=-3 و b=-2 و سپس a="1101" و b="0010" شده که موجب تولید خروجی به صورت y=-6 و سپس y=-11111010 و سپس y="00000110" می‌شود. نتایج شبیه‌سازی برای هر دو حالت در شکل ۳-۵ نمایش داده شده است. ملاحظه می‌کنید که نتایج شبیه‌سازی با نتایج عددی که بدست آوردیم، تطابق دارند.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.numeric_std.all;
4 -----
5 ENTITY multiplier IS
6     PORT (a, b: IN UNSIGNED(3 DOWNTO 0);
7             y: OUT UNSIGNED(7 DOWNTO 0));
8 END ENTITY;
9 -----
10 ARCHITECTURE multiplier OF multiplier IS
11 BEGIN
12     y <= a * b;
13 END ARCHITECTURE;
14 -----

```

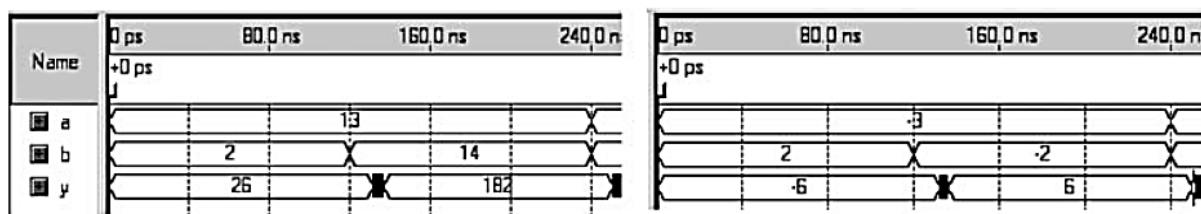


Figure 3.5 Unsigned (left) and signed (right) simulation results from the code of example 3.3.

### مثال ۴-۳: ضرب کننده‌ی علامتدار/بدون علامت شماره ۲

همان مثال قبل را دوباره تکرار کنید. این بار تمام پورت‌ها را از نوع STD\_LOGIC\_VECTOR تعریف کنید. در بخش (الف) از بسته‌ی std\_logic\_unsigned و در قسمت (ب) از بسته‌ی std\_logic\_signed استفاده کنید.

پاسخ:

(الف) کد VHDL در زیر نوشته شده است. در اینجا نیاز به بسته‌ی std\_logic\_1164 (خط ۳) است زیرا این بسته نوع STD\_LOGIC\_VECTOR را تعریف کرده است. به بسته‌ی دیگر (بسته‌ی std\_logic\_unsigned، خط ۴) نیاز نیاز داریم زیرا توابع ریاضی لازم برای کار با این نوع داده را تعریف کرده است. این توابع ریاضی در بسته‌ی std\_logic\_1164 تعریف نشده‌اند.

(ب) تنها تفاوتی که در اینجا نسبت به کد قسمت (الف) باید داده شود جایگذاری کلمه‌ی signed به جای کلمه‌ی signed است. نتایج شبیه‌سازی عین همان نتایجی است که در شکل ۳-۵ نمایش داده شده بود.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_unsigned.all;
5 -----
6 ENTITY multiplier IS
7     PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8             y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9 END ENTITY;
10 -----
11 ARCHITECTURE multiplier OF multiplier IS
12 BEGIN
13     y <= a * b;
14 END ARCHITECTURE;
15 -----
```

توجه: در کد مربوط به مثال ۴-۳ تمام پورت‌ها از نوع STD\_LOGIC\_VECTOR تعریف شده‌اند که این نوع، یک نوع استاندارد صنعتی است. اگرچه این کد و روش استفاده شده در آن صحیح است اما ممکن است منجر به ابهام و سردرگمی شود زیرا طبیعت و ذات علامت‌دار/بدون

علامت بودن کار صراحتا در کد مشخص نمی باشد. در واقع این که عملیات انجام گرفته از نوع علامت دار یا بدون علامت باشد بسته به نوع بسته‌ای است که در خط ۴ اعلان کرده‌ایم. در اینجا توصیه بر این است که پورت‌ها را همچنان از نوع STD\_LOGIC\_VECTOR (که یک استاندارد صنعتی است) تعریف کنیم اما در داخل کد ورودی‌ها را از این نوع به نوع UN(SIGNED) تبدیل کرده و پس از انجام عملیات مورد نظرمان، حاصل را به STD\_LOGIC\_VECTOR برسی و استفاده کرده و در بخش ۷-۵ بیشتر بحث خواهیم کرد.

### ۱۰-۳ نوع‌های اسکالر تعریف شده توسط کاربر

در VHDL علاوه بر انواع داده از پیش تعریف شده امکان تعریف انواع مشخصی از داده‌ها توسط کاربر نیز وجود دارد. این انواع یکی از دو نوع اسکالر (یعنی تک‌مقداری) یا چندمقداری هستند. در حالت چندمقداری از کلمات کلیدی RECORD و یا ARRAY و در حالت تک‌مقداری از کلمه‌ی کلیدی TYPE استفاده می‌شود. متداول‌ترین محل برای استفاده از TYPE داخل کد اصلی (یعنی بخش اعلان معماری) و یا داخل یک بسته می‌باشد.

#### نوع‌های صحیح

نوع INTEGER بدون هیچ محدودیتی قابل سنتز است. به تمام انواعی که مشتق از این نوع باشند نوع صحیح گفته می‌شود. گرامر تعریف یک نوع مبتنی بر INTEGER به صورت زیر است. محدوده یا بازه‌ای که در این گرامر باید مشخص شود باید زیربازه‌ای از INTEGER'LOW تا INTEGER'HIGH (یعنی بین  $-2^{31}$  تا  $2^{31}-1$ ) باشد.

```
TYPE type_name IS RANGE range_specifications;
```

چند مثال:

```
-----  
TYPE negative IS RANGE INTEGER'LOW TO -1;  
TYPE temperature IS RANGE 0 TO 273;  
TYPE my_integer IS RANGE -32 TO 32;  
-----
```

نوع‌های شمارش شده

در این حالت، مقادیر توسط نمادها مشخص شده و باید صراحتاً لیست شوند (به همین دلیل نام «شمارش شده» برای آنها انتخاب شده است). قبل‌اً هم برخی انواع از پیش تعریف شده را در این قالب معرفی و تعریف کرده بودیم. برای مثال:

```
-----  
TYPE BIT IS ('0', '1');  
TYPE BOOLEAN IS (FALSE, TRUE);  
TYPE STD_ULOGIC IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X');  
-----
```

از جمله کاربردهای انواع شمارش شده یکی طراحی ماشین‌های حالت محدود (FSM) و دیگری ایجاد دیگر سیستمهای منطقی است. گرامر ساده‌ی تعریف یک نوع شمارش شده به صورت زیر است:

```
TYPE type_name IS (type_values_list);
```

چند مثال:

```
-----  
TYPE logic_01Z IS ('0', '1', 'Z');  
TYPE state IS (A, B, C, D, E);  
TYPE machine_state IS (idle, transmitting, receiving);  
-----
```

**۱۱-۳ انواع آرایه‌ای تعریف شده توسط کاربر**  
یک ARRAY آرایه‌ای از عناصر همنوع می‌باشد. برای ایجاد یک نوع آرایه‌ای از گرامر ساده‌ی زیر استفاده می‌کنیم:

```
TYPE type_name IS ARRAY (range_specs) OF element_type;
```

برخی انواع از پیش تعریف شده که قبل‌اً معرفی شدند مبتنی بر آرایه بودند. برای مثال:  
Standard types (section 3.5):

```
-----  
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT;  
TYPE BOOLEAN_VECTOR IS ARRAY (NATURAL RANGE <>) OF BOOLEAN;  
TYPE INTEGER_VECTOR IS ARRAY (NATURAL RANGE <>) OF INTEGER;  
TYPE STRING IS ARRAY (POSITIVE RANGE <>) OF CHARACTER;  
-----
```

Standard-logic types (section 3.6):

```
-----  
TYPE STD_ULOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_ULOGIC;  
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

یا برای مثال در 2008 VHDL :

```
TYPE STD_ULONGIC_VECTOR IS array (NATURAL RANGE <>) of STD_ULONGIC;
SUBTYPE STD_LOGIC_VECTOR IS (resolved) STD_ULONGIC_VECTOR;
```

Unsigned/Signed types (section 3.7):

```
TYPE UNSIGNED IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
TYPE SIGNED IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

منظور از علامت <> این است که محدوده در حین کدنویسی مشخص خواهد شد.

انواع آرایه‌ای صحیح تعریف شده توسط کاربر

دانشگاه صنعتی سامانه‌های  
تولید

## فصل چهارم

### عملگرها و مختصه‌ها

مطالعه عملگرها شامل دو دسته عملگر زیر می‌شود:

۱) شش عملگر از پیش تعریف شده شامل حسابی، منطقی، شیفت، مقایسه، تجمعی، و انطباق،

۲) عملگرهای سربارگذاری شده یا تعریف شده توسط کاربر.

مطالعه مختصه‌ها نیز شامل دو دسته‌ی زیر می‌شود:

۱) چهار مختصه از پیش تعریف شده شامل مختصه‌های مربوط به انواع اسکالر، مربوط به انواع آرایه‌ای، مربوط به سیگنال‌ها، و مربوط به موجودیت‌ها نامدار،

۲) مختصه‌های تعریف شده توسط کاربر و مختصه‌های سنتز.

#### ۴-۲ عملگرهای از پیش تعریف شده

این عملگرها متنوع بوده و برخی از آنها عبارتند از:

- عملگرهای تخصیص
- عملگرهای حسابی
- عملگرهای منطقی
- عملگرهای شیفت
- عملگرهای مقایسه‌ای (یا رابطه‌ای)
- عملگر تجمعی
- عملگرهای مقایسه انطباق
- دیگر عملگرها

#### عملگرهای تخصیص

برای تخصیص مقدار به اشیاء VHDL استفاده می‌شوند.

این عملگرها عبارتند از:

- عملگر «=>»: برای تخصیص مقدار به SIGNAL استفاده می‌شود.

- عملگر «:=» : برای تخصیص مقدار به یک CONSTANT یا VARIABLE استفاده می‌شود. همچنین برای تخصیص مقدار اولیه (پیش‌فرض) به سیگنال‌ها و متغیرها نیز استفاده می‌شود. این عملگر در GENERIC نیز که در واقع مربوط به تعریف ثابت است، استفاده می‌شود.
- عملگر «=>» : برای تخصیص مقدار به عناصر یک آرایه یا به صورت تک تک و یا به صورت گروهی (به کمک OTHERS) استفاده می‌شود.

مثال:

```
CONSTANT x: STD_LOGIC_VECTOR(7 DOWNTO 0) := "00010001";
SIGNAL y: STD_LOGIC_VECTOR(1 TO 4);
VARIABLE z: BIT_VECTOR(3 DOWNTO 0);
y(4) <= '1'; --'1' assigned to a signal using "<="
y <= "0000"; --"0000" assigned to a signal with "<="
y <= (OTHERS=>'0') --'0' assigned to all elements of y
y <= x(3 DOWNTO 0); --part of x assigned to y
z := "1000"; --"1000" assigned to a variable with ":="
z := (0=>'1', OTHERS=>'0'); --z="0001"
```

## عملگرهای منطقی

برای انجام عملیات منطقی استفاده شده و عبارتند از:

- NOT •
- AND •
- OR •
- NAND •
- NOR •
- XOR •
- XNOR •

انواعی که با این نوع عملگرها کار کرده و سنتز پذیر هستند عبارتند از:

BIT, BIT\_VECTOR, BOOLEAN, STD\_(U)LOGIC, STD\_(U)LOGIC\_VECTOR  
نوع (UN)SIGNED numeric\_std هم در کنار انواع فوق به شرطی قرار می‌گیرد که از بسته استفاده کرده باشیم.

در VHDL 2008 انواع زیر نیز برای کار با عملگرهای منطقی تعریف شده و قابل استفاده‌اند:  
BOOLEAN\_VECTOR, UFIXED, SFIXED, FLOAT.

مثال:

```

x <= NOT a AND b; --x=a'.b
y <= NOT (a AND b); --x=(a.b)'
z <= a NAND b; --x=(a.b)'

```

**عملگرهای حسابی**

این عملگرها عبارتند از:

- ❖ جمع (+)
- ❖ تفریق (-)
- ❖ ضرب (\*)
- ❖ تقسیم (/)
- ❖ توان (\*\*)
- ❖ مقدار مطلق (ABS)
- ❖ باقیمانده (REM)
- ❖ باقیمانده (MOD)

انواعی که قابلیت کار با این عملگرها را داشته و سنتزپذیر نیز خواهند بود عبارتند از:

INTEGER, NATURAL, POSITIVE, (U)SIGNED, STD\_LOGIC\_VECTOR

انواع (U)SIGNED به شرطی قابل استفاده هستند که از یکی از بسته‌های numeric\_std و یا std\_logic\_arith استفاده کرده باشیم. نوع STD\_LOGIC\_VECTOR نیز در صورتی قابل استفاده است که از یکی از بسته‌های std\_logic\_signed، std\_logic\_unsigned و یا numeric\_std\_unsigned استفاده کرده باشیم.

در VHDL 2008 انواع زیر برای کار با عملگرهای حسابی تعریف شده و قابل استفاده‌اند: UFIXED, SFIXED, FLOAT.

عملگرهای جمع، تفریق، ضرب و تقسیم در مورد کار با اعداد صحیح هیچ محدودیت سنتز شدن ندارند. در مورد عملگر توان یا باید توان عددی ایستا<sup>1</sup> (یعنی ثابت و مشخص) باشد یا اگر توان عددی غیرایستا است، پایه باید ایستا و حتی توان صحیحی از ۲ باشد. سه عملگر بعدی (ABS، REM و MOD) نیز هیچ محدودیت سنتز در حین کار با اعداد صحیح ندارند. در زیر برخی عملگرها توضیح داده شده‌اند:

<sup>1</sup>Static

- $x/y$ : Returns 0 when  $|x| < |y|$ ,  $\pm 1$  when  $|y| \leq |x| < 2|y|$ ,  $\pm 2$  when  $2|y| \leq |x| < 3|y|$ , etc., with the sign obviously negative when the signs of  $x$  and  $y$  are different.

**Examples**  $3/5 = 0$ ,  $-3/5 = 0$ ,  $9/5 = 1$ ,  $-9/5 = -1$ ,  $10/5 = 2$ ,  $-10/5 = -2$ ,  $14/5 = 2$ ,  $-14/5 = -2$ .

- ABS  $x$ : Returns the absolute value of  $x$ .

**Examples**  $\text{ABS } 5 = 5$ ,  $\text{ABS } -3 = 3$ .

- $x \text{ REM } y$ : Returns the remainder of  $x/y$ , with the sign of  $x$ . Its equation is  $x \text{ REM } y = x - (x/y)^*y$ , where both operands are integers.

**Examples**  $6 \text{ REM } 3 = 0$ ,  $7 \text{ REM } 3 = 1$ ,  $7 \text{ REM } -3 = 1$ ,  $-7 \text{ REM } 3 = -1$ ,  $-7 \text{ REM } -3 = -1$ .

- $x \text{ MOD } y$ : Returns the remainder of  $x/y$ , with the sign of  $y$ . Its equation is  $x \text{ MOD } y = x \text{ REM } y + a^*y$ , where  $a = 1$  when the signs of  $x$  and  $y$  are different or  $a = 0$  otherwise. Both operands are integers.

**Examples**  $7 \text{ MOD } 3 = 1$ ,  $7 \text{ MOD } -3 = -2$ ,  $-7 \text{ MOD } 3 = 2$ ,  $-7 \text{ MOD } -3 = -1$ .

### عملگرهای مقایسه

به این عملگرهای رابطه‌ای نیز اطلاق شده و عبارتند از:

- Equal to ( $=$ )
- Not equal to ( $\neq$ )
- Less than ( $<$ )
- Greater than ( $>$ )
- Less than or equal to ( $\leq$ )
- Greater than or equal to ( $\geq$ )

انواعی که قابلیت کار با این عملگرهای نیز خواهند بود عبارتند از:

As defined in the original packages (see chapter 3 or appendices), the synthesizable predefined data types that support comparison operators are BIT, BIT\_VECTOR, BOOLEAN, INTEGER, NATURAL, POSITIVE, CHARACTER, and STRING. If one of the packages for (un)signed types (*numeric\_std* or *std\_logic\_arith*) is declared in the code, then (UN)SIGNED can also be used. If the package *std\_logic\_unsigned*, *std\_logic\_signed*, or *numeric\_std\_unsigned* is also declared, then STD\_LOGIC\_VECTOR can be employed as well.

In VHDL 2008, the following new types with support for comparison operations were included: BOOLEAN\_VECTOR, INTEGER\_VECTOR, UFIXED, SFIXED, and FLOAT.

## عملگرهای شیفت

این عملگرها به منظور شیفت بردارهای داده‌ای استفاده شده و در همان استاندارد اولیه VHDL 1993 معرفی شده‌اند. این عملگرها عبارتند از:

- Shift left logic (SLL): Positions on the right are filled with '0's.
- Shift right logic (SRL): Positions on the left are filled with '0's.
- Shift left arithmetic (SLA): Rightmost bit is replicated on the right.
- Shift right arithmetic (SRA): Leftmost bit is replicated on the left.
- Rotate left (ROL): Circular shift to the left.
- Rotate right (ROR): Circular shift to the right.

در اینجا نیز انواع قابل سنتزی که قابلیت کار با این نوع عملگرها را دارند عبارتند از:

As defined in the original packages (see chapter 3 or appendices), the only synthesizable data type that supports shift operators is BIT\_VECTOR. If one of the packages for (un)signed types (*numeric\_std* or *std\_logic\_arith*) is declared in the code, then (UN)SIGNED can also be used (though the latter package contains very few of such operators—see appendix K). If the package *std\_logic\_unsigned*, *std\_logic\_signed*, or *numeric\_std\_unsigned* is also declared, then STD\_LOGIC\_VECTOR can be employed as well (also a very reduced set).

In VHDL 2008, the following new types with support for shift operations were included: BOOLEAN\_VECTOR, UFIXED, and SFIXED.

گرامر استفاده از عملگرهای شیفت به صورت زیر است:

`<left_operand> <shift_operation> <right_operand>`

عملوند سمت چپ باید از یکی از انواع ذکر شده در پاراگراف قبلی باشد. عملوند سمت راست باید همیشه یک عدد صحیح (INTEGER) باشد که البته می‌تواند شامل علامت + و - نیز باشد. غیر از عملگرهای شیفت، از عملگر تجمعی نیز می‌توان برای تحقق شیفت استفاده کرد. این مطلب در مثال زیر نشان داده شده است.

**Examples** Say that *x* is a BIT\_VECTOR signal with value *x* = "01001". Then the values produced by the assignments below are those indicated in the comments (equivalent expressions, using the *concatenation* operator, are shown between parentheses).

---

```

y <= x SLL 2;    --y<="00100" (y <= x(2 DOWNTO 0) & "00");
y <= x SLA 2;    --y<="00111" (y <= x(2 DOWNTO 0) & x(0) & x(0));
y <= x SRL 3;    --y<="00001" (y <= "000" & x(4 DOWNTO 3));
y <= x SRA 3;    --y<="00001" (y <= x(4) & x(4) & x(4) & x(4 DOWNTO 3));
y <= x ROL 2;    --y<="00101" (y <= x(2 DOWNTO 0) & x(4 DOWNTO 3));
y <= x SRL -2;   --same as "x SLL 2"

```

---

## عملگر تجمعی

به منظور در کنار هم قرار دادن اشیاء و مقادیر استفاده می‌شود (البته همان طور که در مثال قبلی نیز ملاحظه شد به منظور انجام شیفت نیز می‌توان از این عملگر بهره جست). نماد این عملگر به صورت & می‌باشد.

انواع قابل سنتزی که قابلیت کار با این عملگر را دارند عبارتند از:

The synthesizable predefined data types for which the concatenation operator is intended are BIT\_VECTOR, BOOLEAN\_VECTOR (VHDL 2008), INTEGER\_VECTOR (VHDL 2008), STD\_(U)LOGIC\_VECTOR, (UN)SIGNED, and STRING. Recall that the keyword OTHERS (seen in section 3.2) can also be helpful to make array assignments.

(استفاده از کلمه OTHERS مفید می‌تواند واقع شود)

مثال زیر نحوه استفاده از این عملگر را نشان می‌دهد:

**Example** Four VHDL objects ( $v, x, y, z$ ) are declared below, then several assignments are made utilizing the concatenation operator (&). The use of parentheses is optional.

```
-----
CONSTANT v: BIT := '1';
CONSTANT x: STD_LOGIC := 'Z';
SIGNAL y: BIT_VECTOR(1 TO 4);
SIGNAL z: STD_LOGIC_VECTOR(7 DOWNTO 0);
y <= (v & "000");           --result: "1000"
y <= v & "000";             --same as above (parentheses are optional)
z <= (x & x & "11111" & x); --result: "ZZ11111Z"
z <= ('0' & "011111" & x); --result: "0011111Z"
-----
```

مثال زیر نیز استفاده از کلمه OTHERS و ویرگول را نشان می‌دهد.

**Example** Consider the same constants and signals above. Below is a series of individual-bit assignments using the keyword OTHERS and comma instead of the regular concatenation operator. Observe the nominal and positional mapping options. Here, parentheses are required.

```
-----
y <= (OTHERS=>'0');           --result: "0000"
y <= (4=>'1', OTHERS=>'0');   --result: "0001" (nominal mapping)
y <= ('1', OTHERS=>'0');       --result: "1000" (positional mapping)
y <= (4=>'1', 2=>v, OTHERS=>'0'); --result: "0101" (nominal mapping)
z <= (OTHERS=>'Z');            --result: "ZZZZZZZZ"
z <= (4=>'1', OTHERS=>'0');   --result: "00010000" (nominal mapping)
z <= (4=>x, OTHERS=>'0');     --result: "000Z0000" (nominal mapping)
z <= ('1', OTHERS=>'0');       --result: "10000000" (posit. mapping)
-----
```

## عملگرهای مقایسه‌ی انطباق

این عملگرها عبارتند از:

- Matching equality operator ( $?=$ )
- Matching inequality operator ( $?/=?$ )
- Matching less than operator ( $?<$ )
- Matching greater than operator ( $?>$ )
- Matching less than or equal to operator ( $?<=$ )
- Matching greater than or equal to operator ( $?>=$ )

عملگرهای فوق در VHDL 2008 معرفی شده‌اند.

أنواع قابل ستزى که با اين عملگرها مى‌توانند کار کنند عبارتند از:

These operators were introduced in VHDL 2008. They include the types BIT (whole set), BIT\_VECTOR (only equality and inequality), STD\_(U)LOGIC (whole set—see part II of the package *std\_logic\_1164* in appendix I), STD\_(U)LOGIC\_VECTOR (whole set—see the new package *numeric\_std\_unsigned* in appendix N), and (UN)SIGNED (whole set—see part II of the package *numeric\_std* in appendix J). They also include the new types UFIXED and SFIXED.

هدف از عملگر  $=?$  در داده‌های مبتنی بر نوع STD\_ULOGIC امکان مقایسه‌ی مقادیر منطقی با یکدیگر است. برای مثال شرط "IF 'H' = '1'..." مقدار FALSE را برمی‌گرداند زیرا دو نماد مختلف بوده و عین هم نمی‌باشند. اما در مقابل، شرط "IF 'H'?= '1'..." مقدار TRUE را برمی‌گرداند زیرا هر دو نماد 'H' و '1' به مقدار منطقی '1' تفسیر می‌شوند. استدلال مشابهی را می‌توان برای 'L' و '0' به کار برد. هرگاه از مقادیر 'X', 'Z' و یا 'W' در عمل مقایسه استفاده شود، این عملگر مقدار 'X' را برخواهد گرداند. در حالت استفاده از نوع BIT، این عملگر به جای FALSE و TRUE مقادیر '0' و '1' را برخواهد گرداند.

## ساير عملگرها

ساير عملگرها که در VHDL 2008 معرفی شده‌اند عبارتند از:

- MINIMUM and MAXIMUM operators: Return the smallest or largest value in the given set. For example, "MAXIMUM(0, 55, 23)" returns 55. These operators were defined for all VHDL types.
- Condition operator ("??"): Converts a BIT or STD\_(U)LOGIC value into a BOOLEAN value. For example, "??" *a* AND *b*" returns TRUE when *a* AND *b* = '1' or FALSE otherwise.
- TO\_STRING: Converts a value of type BIT, BIT\_VECTOR, STD\_LOGIC\_VECTOR, and so on into STRING. For the types BIT\_VECTOR and STD\_LOGIC\_VECTOR, there are also the options TO\_OSTRING and TO\_HSTRING, which produce an octal or hexadecimal string, respectively. This operator is useful, for example, when reporting synthesizer or simulator information, because the ASSERT statement can only report data of type STRING (this will be studied in section 9.2).

مثال:

### Examples

`TO_STRING(58) = "58"`

`TO_STRING(B"1110000) = "111100000"`

`TO_HSTRING(B"1110000) = "F0"`

### خلاصه عملگرهای

Operator type	Predefined operators	Supported synthesizable predefined data types (*)
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, BOOLEAN, BOOLEAN_VECTOR <sup>(1)</sup> , STD_(U)LOGIC, STD_LOGIC_(U)VECTOR, (UN)SIGNED <sup>(2)</sup> , UFIXED <sup>(1)</sup> , SFIXED <sup>(1)</sup> , FLOAT <sup>(1)</sup>
Arithmetic	+, -, *, /, **, ABS, REM, MOD	INTEGER, NATURAL, POSITIVE, STD_(U)LOGIC_VECTOR <sup>(3)</sup> , (UN)SIGNED <sup>(4)</sup> , UFIXED <sup>(1)</sup> , SFIXED <sup>(1)</sup> , FLOAT <sup>(1)</sup>
Comparison	=, /=, >, <, >=, <=	BIT, BIT_VECTOR, BOOLEAN, BOOLEAN_VECTOR <sup>(1)</sup> , INTEGER, NATURAL, POSITIVE, INTEGER_VECTOR <sup>(1)</sup> , CHARACTER, STRING, STD_(U)LOGIC_VECTOR <sup>(3)</sup> , (UN)SIGNED <sup>(4)</sup> , UFIXED <sup>(1)</sup> , SFIXED <sup>(1)</sup> , FLOAT <sup>(1)</sup>
Shift	SLL, SRL, SLA, SRA, ROL, ROR	BIT_VECTOR, BOOLEAN_VECTOR <sup>(1)</sup> , STD_LOGIC_(U)VECTOR <sup>(3)</sup> , (UN)SIGNED <sup>(4)</sup> , UFIXED <sup>(1)</sup> , SFIXED <sup>(1)</sup>
Concatenation	& (" " and OTHERS too)	BIT_VECTOR, BOOLEAN_VECTOR <sup>(1)</sup> , INTEGER_VECTOR <sup>(1)</sup> , STRING, STD_(U)LOGIC_VECTOR, (UN)SIGNED <sup>(4)</sup>
Matching comparison <sup>(1)</sup>	?=, ?/=, ?>, ?<, ?>=, ?<=	BIT, BIT_VECTOR <sup>(1)</sup> , BOOLEAN_VECTOR <sup>(1)</sup> , STD_(U)LOGIC, STD_(U)LOGIC_VECTOR, (UN)SIGNED <sup>(2)</sup> , UFIXED <sup>(1)</sup> , SFIXED <sup>(1)</sup> , FLOAT <sup>(1)</sup>
Condition <sup>(1)</sup>	??	BIT, STD_(U)LOGIC
Min/Max and String conversion <sup>(1)</sup>	MINIMUM, MAXIMUM, TO_STRING, etc.	Nearly all VHDL types in standard packages (see appendices)

(\*) Note: Some types support only a partial set of operators

(3) Requires package `std_logic_(un)signed` or `numeric_std_unsigned`

(1) Introduced or proposed in VHDL 2008

(4) Requires package `numeric_std` or `std_logic_arith`

(2) With package `numeric_std`

#### ۴-۴ مختصه (صفت)‌های از پیش تعریف شده

مختصه‌ها اطلاعاتی را راجع به اقلام مشخص شده در اختیار ما قرار می‌دهند.

چهار دسته مختصه (یا صفت) از پیش تعریف شده داریم:

- مختصه‌های مربوط به انواع اسکالر
- مختصه‌های مربوط به انواع آرایه‌ای
- مختصه‌های مربوط به سیگنال‌ها
- مختصه‌های مربوط به اقلام نام دار

به دو نکته توجه کنید: (۱) از میان مختصه‌های موجود، مختصه‌ی EVENT جزو متداول‌ترین مختصه‌ها است که تاکنون به آن برخورده‌اید. (۲) ابزار سنتز موجود ممکن است از برخی از این مختصه‌ها پشتیبانی نکند.

#### مختصه‌های مربوط به انواع اسکالر

منظور از انواع اسکالر انواعی مانند عددی<sup>۱</sup> و شمارش شده<sup>۲</sup> هستند. در شکل ۲-۴ اسکالر با علامت T مشخص شده است.

مثال:

دو نوع اسکالر زیر را در نظر بگیرید:

```
-----  
TYPE my_integer IS RANGE 0 TO 255;  
TYPE state IS (a, b, c);  
-----
```

حال مقادیر چند مختصه مربوط به نوع صحیح my\_integer در زیر نشان داده شده است. به توضیحات داده شده نیز توجه کرده و شکل ۲-۴ را نیز ملاحظه کنید.

```
-----  
x1 <= my_integer'LEFT;      --result=0 (type of x1 must be my_integer)  
x2 <= my_integer'RIGHT;     --result=255 (type of x2 must be my_integer)  
x3 <= my_integer'LOW;       --result=0 (type of x3 must be my_integer)  
x4 <= my_integer'HIGH;      --result=255 (type of x4 must be my_integer)  
y <= my_integer'ASCENDING;   --result=TRUE (type of y must be BOOLEAN)  
-----
```

حال مقادیر چند مختصه مربوط به نوع شمارش شده‌ی state در زیر نشان داده شده است. به توضیحات داده شده نیز توجه کرده و شکل ۲-۴ را نیز ملاحظه کنید. در اینجا فرض شده که روش ترتیبی (sequential) برای کدگذاری اقلام شمارش شده انتخاب شده است. اگر برای مثال روش one-hot انتخاب شده باشد، خواهیم داشت:

$$a = "001", b = "010", c = "100"$$

<sup>1</sup>Numeric

<sup>2</sup>Enumerated

```

x1 <= state'LEFT;    --result=a ("00") (type of x1 must be state)
x2 <= state'RIGHT;   --result=c ("10") (type of x2 must be state)
x3 <= state'LOW;     --result=a ("00") (type of x3 must be state)
x4 <= state'HIGH;    --result=c ("10") (type of x4 must be state)
y <= state'POS(b);   --result=1 ("01") (type of y is INTEGER)
z <= state'VAL(1);   --result=b ("01") (type of z must be state)

```

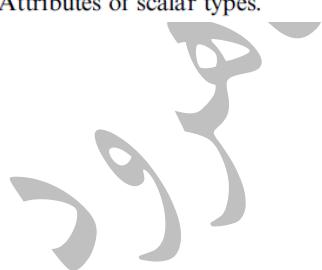
Name	Result TYPE	Result
T'LEFT	Same as T	Leftmost value of T
T'RIGHT	Same as T	Rightmost value of T
T'LOW	Same as T	Lower bound of T
T'HIGH	Same as T	Upper bound of T
T'ASCENDING	BOOLEAN	TRUE if range of T is ascending, FALSE otherwise
T'IMAGE(X)	STRING	String representing the value X in T
T'VALUE(X)	Base type of T	Value of T whose string representation is X
T'POS(X)	INTEGER	Position number of the value X in T
T'VAL(X)	Base type of T	Value whose position number in T is X
T'SUCC(X)	Base type of T	Value whose position number in T is X + 1
T'PRED(X)	Base type of T	Value whose position number in T is X - 1
T'LETOF(X)	Base type of T	Value on the left of the position number X in T
T'RIGHTOF(X)	Base type of T	Value on the right of the position number X in T
T'BASE	Any (sub)type	Base type of T
O'SUBTYPE	Any subtype	Constrained subtype of O with constraint information

Ascending range: T'LEFT = T'LOW, T'RIGHT = T'HIGH

Descending range: T'LEFT = T'HIGH, T'RIGHT = T'LOW

Figure 4.2  
Attributes of scalar types.

مثال:



### Example 4.1: Using Predefined Scalar Attributes

The code below contains an enumerated type called *color*. Several assignments made in lines 12–16. Determine the values of *x* that produce *y* = '1'. (The solution included as comments.)

```

1 -----
2 ENTITY example IS
3     PORT (x: IN INTEGER RANGE 0 TO 3;
4             y1, y2, y3, y4, y5: OUT BIT);
5 END example;
6 -----
7 ARCHITECTURE example OF example IS
8     TYPE color IS (red, green, blue); --assume seq. encoding
9     SIGNAL z: color;
10 BEGIN
11     z <= red WHEN x=0 ELSE green WHEN x=1 ELSE blue;
12     y1<='1' WHEN color'VAL(x)=blue ELSE '0';      --y1='1' for x=2
13     y2<='1' WHEN color'POS(blue)=x ELSE '0';      --y2='1' for x=2
14     y3<='1' WHEN color'RIGHTOF(z)=blue ELSE '0'; --y3='1' for x=1
15     y4<='1' WHEN color'PRED(z)=green ELSE '0';   --y4='1' for x=2,3
16     y5<='1' WHEN color'PRED(green)=z ELSE '0';   --y5='1' for x=0
17 END example;
18 -----

```



مختصه‌های تعریف شده برای انواع آرایه‌ای  
در شکل ۴-۳ نماد A معرف یک نوع آرایه‌ای است.

Name	Result TYPE	Result
A'LEFT [(N)]	Type of the Nth index range of A	Left bound of the Nth index range of A
A'RIGHT [(N)]	Same as above	Right bound of the Nth index range of A
A'LOW [(N)]	Same as above	Lower bound of the Nth index range of A
A'HIGH [(N)]	Same as above	Upper bound of the Nth index range of A
A'RANGE [(N)]	Same as above	Range of the Nth index range of A
A'REVERSE_RANGE [(N)]	Same as above	Reverse range of the Nth index range of A
A'LENGTH [(N)]	INTEGER	Number of values in the Nth index range
A'ASCENDING [(N)]	BOOLEAN	TRUE if Nth index range of A is ascending, FALSE otherwise
A'ELEMENT	Element subtype of A	Element subtype of A with constraint information

Figure 4.3  
Attributes of array types.

مثال:

**Example** Consider the array type *matrix* below (note that the keyword ARRAY is employed in its definition) and the signal *test* declared subsequently.

```
-----  
TYPE matrix IS ARRAY (1 TO 4, 7 DOWNTO 0) OF BIT;  
SIGNAL test: matrix;  
-----
```

The values returned for several of the predefined attributes are shown below (the same information would be returned for *test*).

```
-----  
x1 <= matrix'LEFT(1);           --result=1 (type of x1 must be INTEGER or eq.)  
x2 <= matrix'LEFT(2);           --result=7 (type of x2 must be INTEGER or eq.)  
x3 <= matrix'RIGHT(1);          --result=4 (type of x3 must be INTEGER or eq.)  
x4 <= matrix'RIGHT(2);          --result=0 (type of x4 must be INTEGER or eq.)  
x5 <= matrix'LENGTH(2);         --result=8 (type of x5 must be INTEGER or eq.)  
x6 <= matrix'ASCENDING(1);      --result=TRUE (type of x6 must be BOOLEAN)  
matrix'RANGE(1) -> returns 1 TO 4  
matrix'REVERSE_RANGE(2) -> returns 0 TO 7  
matrix'ELEMENT -> returns BIT  
-----
```

:مثال

**Example** Consider the signal *x* declared below. All five LOOP statements that follow are synthesizable and equivalent.

```
-----  
SIGNAL x: STD_LOGIC_VECTOR(7 DOWNTO 0);  
FOR i IN 7 DOWNTO 0 LOOP ...  
FOR i IN x'RANGE LOOP ...  
FOR i IN x'HIGH DOWNTO x'LOW LOOP ...  
FOR i IN x'LEFT DOWNTO x'RIGHT LOOP ...  
FOR i IN x'LENGTH-1 DOWNTO 0 LOOP ...  
-----
```

مختصه های از پیش تعریف شده سیگنال ها در شکل ۴-۴ S معرف یک سیگنال است.

Name	Result TYPE	Result
S'DELAYED [(t)]	Base type of S	Signal equivalent to signal S delayed t units of time
S'STABLE [(t)]	BOOLEAN	TRUE when no event has occurred on signal S for t units of time, FALSE otherwise
S'QUIET [(t)]	BOOLEAN	TRUE when no transaction has been scheduled for signal S for t units of time, FALSE otherwise
S'TRANSACTION	BIT	A bit that toggles in each simulation cycle in which S becomes active (transaction scheduled)
S'EVENT	BOOLEAN	TRUE if an event has just occurred on S, FALSE otherwise
S'ACTIVE	BOOLEAN	TRUE if a transaction has just been scheduled for S, FALSE otherwise
S'LAST_EVENT	TIME	Amount of time since last event occurred on signal S
S'LAST_ACTIVE	TIME	Amount of time since last time signal S was active (transaction scheduled)
S'LAST_VALUE	Base type of S	Value of signal S previous to the last event (if no event occurred, returns current value of S)
S'DRIVING	BOOLEAN	TRUE if process is driving S, FALSE if S is disconnected from the driver.
S'DRIVING_VALUE	Base type of S	Value of the driver for S in the current process

Figure 4.4  
Attributes of signals.

تفاوت بین یک رخداد (event) و یک «تغییر وضعیت» (transaction) این است که رخداد به وقوع یک لبه روی یک سیگنال گفته می‌شود (یعنی یک گذر<sup>۱</sup> روبرو بالا یا رو به پایین); این تغییر معمولاً مقدمه و شروع‌کنندهٔ وقوع یک تغییر روی یک سیگنال دیگر است. اگر واقعاً چنین باشد آن‌گاه گفته می‌شود که روی سیگنال دوم یک «تغییر وضعیت» برنامه‌ریزی شده است؛ این تغییر وضعیت در زمانی بعد از وقوع رخداد انجام خواهد شد (یعنی در پایان سیکل فعلی فرآیند). همان طور که در شکل ۴-۴ نشان داده شده است، بازه زمانی ( $t$ ) که در اولین سه مختصهٔ دیده می‌شود، اختیاری است. مقدار پیش فرض آن صفر است. در این حالت،  $S'STABLE(0\text{ ns}) = 1$  معادل با S'EVENT است. نحوهٔ استفاده از این مختصه‌ها و همچنین مختصهٔ S'LAST\_VALUE در مثال زیر نمایش داده شده است.

<sup>۱</sup>Transition

**Example 4.2: DFF with Several Event-Based Attributes**

The code below produces a D-type flip-flop (DFF), triggered at the positive edge of the clock. To detect a positive clock edge, three equivalent synthesizable alternatives are shown in lines 13–15.

```

1 -----
2 ENTITY flipflop IS
3     PORT (d, clk, rst: IN BIT;
4             q: OUT BIT);
5 END flipflop;
6 -----
7 ARCHITECTURE example OF flipflop IS
8 BEGIN
9     PROCESS(clk, rst)
10    BEGIN
11         IF (rst='1') THEN
12             q <= '0';
13         ELSIF (clk'EVENT AND clk='1') THEN
14             --ELSIF (NOT clk'STABLE AND clk='1') THEN
15             --ELSIF (clk'EVENT AND clk'LAST_VALUE='0') THEN
16                 q <= d;
17             END IF;
18     END PROCESS;
19 END example;
20 -----

```

**مختصه‌های مربوط به اقلام نامدار<sup>۱</sup>**

این مختصه‌ها جهت تولید رشته‌ای مرتبط با نام اقلام اعلام شده استفاده می‌شوند. در شکل ۴-۵ از نماد E جهت نمایش اقلام استفاده شده است.

Name	Result TYPE	Result
E\$SIMPLE_NAME	STRING	String representing the simple name, character literal, or operator symbol of the named entity E
E\$INSTANCE_NAME	STRING	String describing the hierarchical path from the top entity or architecture down to the named entity E, including the names of instantiated design entities
E\$PATH_NAME	STRING	Same as above, but excluding the names of instantiated design entities

**Figure 4.5**  
String-related attributes.

<sup>1</sup>Named Entities

## مختصه‌های مربوط به سنتز

هدف از این مختصه‌ها ارتباط با کامپایلر است. مهمترین این مختصه‌ها عبارتند از:

- *enum\_encoding* attribute
- *chip\_pin* attribute
- *keep* attribute
- *preserve* attribute
- *noprune* attribute.

## مختصه *enum\_encoding*

استفاده از انواع داده شمارش شده در طراحی ماشینهای حالت محدود ضروری است. در این کار نیاز به کدگذاری حالتها داریم. این مختصه جهت تعیین نوع کدگذاری استفاده می‌شود. مهمترین انواع کدگذاری عبارتند از:

- Sequential encoding
- Gray encoding
- Johnson encoding
- One-hot encoding
- User-defined encoding.

مشابه با هر مختصه‌ی دیگری در اینجا نیاز به اعلان و سپس تعریف خصوصیات دارد. این کار در مثال بعدی توضیح داده شده است.

مثال: در خطوط زیر نوع شمارش شده state ایجاد می‌شود تا حالت‌های یک FSM را نمایش دهد. گزینه‌های مختلفی که در استفاده از این مختصه وجود دارد در ادامه نشان داده شده است (آخرین مورد یک مورد تعریف شده توسط کاربر است).

```

TYPE state IS (A, B, C, D);
-----  

ATTRIBUTE enum_encoding: STRING;
ATTRIBUTE enum_encoding OF state: TYPE IS "sequential";
--Result: A="00", B="01", C="10", D="11"
-----  

ATTRIBUTE enum_encoding: STRING;
ATTRIBUTE enum_encoding OF state: TYPE IS "one-hot";
--Result: A="0001", B="0010", C="0100", D="1000"
-----  

ATTRIBUTE enum_encoding: STRING;
ATTRIBUTE enum_encoding OF state: TYPE IS "11 00 10 01";
--Result: A="11", B="00", C="10", D="01"
-----  


```

**مختصه chip\_pin**

به کمک این مختصه می‌توان سیگنالهای نام برد شده در PORT را به پین‌های افزاره تخصیص داد.

مثال: خطوط زیر موجب تخصیص سیگنال کلک clk به پین N2 از افزاره مورد نظر می‌شود.

```

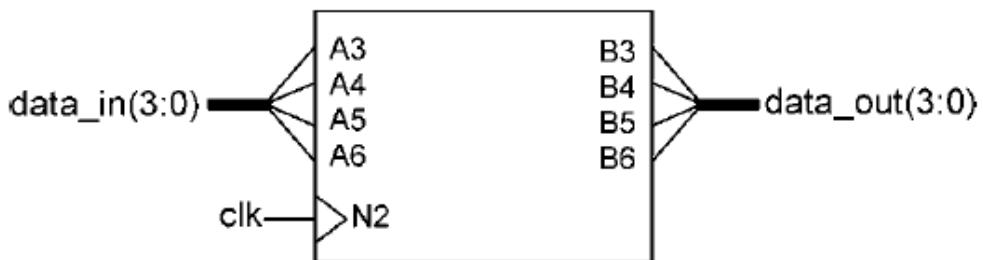
ATTRIBUTE chip_pin: STRING;
ATTRIBUTE chip_pin OF clk: SIGNAL IS "N2";

```

مثال ۳-۴: مشخص کردن پین‌های افزاره به کمک مختصه chip\_pin

در شکل زیر یک ثبات ۴ بیتی نشان داده شده است که دارای ورودی‌های data\_in و data\_out می‌باشد. این مدار را روی طراحی کنید که اتصالات خواسته شده بین پورت‌های مدار و پین‌های افزاره برقرار شود (توجه کنید که شما در طراحی خود باید پین‌هایی را انتخاب کنید که امکان استفاده از آنها به عنوان پین‌های کاربر<sup>۱</sup> فراهم باشد). برای برقراری چنین تخصیصی دو راه وجود دارد. یکی استفاده از امکانات و تنظیمات کامپایلر و دیگری استفاده از مختصه chip\_pin در داخل کد VHDL می‌باشد. در این مساله از راه حل دوم استفاده می‌کنیم.

<sup>۱</sup> User Pins



**Figure 4.6**  
4-bit register of example 4.3.

حل:

همان طور که در کد زیر ملاحظه می‌کنید از مختصه‌ی chip\_pin در بخش اعلان معماری (خطوط ۹ تا ۱۲) استفاده شده است گرچه داخل بخش اعلان موجودیت (بین خطوط ۵ و ۶) نیز می‌توانیم این کار را انجام دهیم.

```

1 -----
2 ENTITY data_register IS
3     PORT (clk: IN BIT;
4             data_in: IN BIT_VECTOR(3 DOWNTO 0);
5             data_out: OUT BIT_VECTOR(3 DOWNTO 0));
6 END ENTITY;
7 -----
8 ARCHITECTURE data_register OF data_register IS
9     ATTRIBUTE chip_pin: STRING;
10    ATTRIBUTE chip_pin OF clk: SIGNAL IS "N2";
11    ATTRIBUTE chip_pin OF data_in: SIGNAL IS "A3, A4, A5, A6";
12    ATTRIBUTE chip_pin OF data_out: SIGNAL IS "B3, B4, B5, B6";
13 BEGIN
14     PROCESS (clk)
15     BEGIN
16         IF (clk'EVENT AND clk='1') THEN
17             data_out <= data_in;
18         END IF;
19     END PROCESS;
20 END ARCHITECTURE;
21 -----

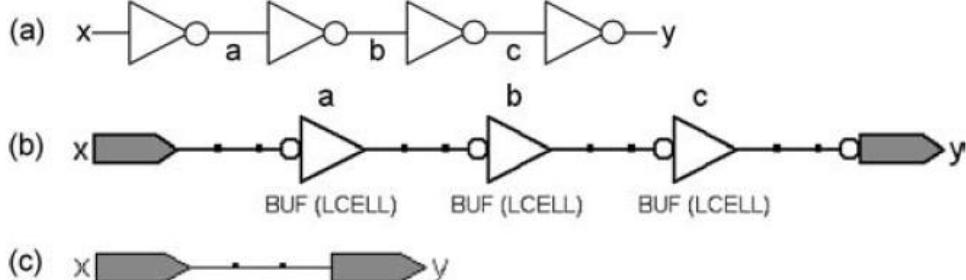
```

### مختصه‌ی keep

هدف از این مختصه (که گاهآآن را syn\_keep نیز می‌نامند) این است که جلوی کامپایلر در ساده‌سازی و کاهش گره‌های لیست شده گرفته شود. به عبارت دیگر جلوی برخی ساده‌سازی‌های

منطقی در مدارات ترکیبی گرفته می‌شود. در حین استفاده از این مختصه باید نام سیگنال‌ها (سیم‌ها) بی‌که مایل به حفظ آنها هستیم را ذکر کنیم.

**مثال ۴-۴: ساخت خط تاخیر به کمک مختصه keep**  
 یک کد VHDL جهت ساخت یک خط تاخیر به کمک چهار معکوس‌کننده بنویسید (یعنی شکل ۷-۴-الف). کد خود را یک بار با استفاده از مختصه keep و یک بار هم بدون استفاده از این مختصه کامپایل و سپس نتایج را با هم مقایسه کنید.



**Figure 4.7**  
 (a) Delay line of example 4.4; RTL viewer image (b) with and (c) without the *keep* attribute.

حل:

یک کد VHDL در زیر نشان داده شده است. در این کد به منظور جلوگیری از حذف و ساده‌سازی گره‌های a، b و c از مختصه keep در خطوط ۹ الی ۱۰ استفاده شده است. شکل ۷-۴-ب خروجی RTL Viewer<sup>۱</sup> را بعد از کامپایل این کد نشان می‌دهد. توجه کنید که در این شکل تعداد ۴ معکوس‌کننده وجود دارد که چهارمی در بافر خروجی نهفته است. همچنین اگر گره‌های x (وروودی مدار) و y (خروجی مدار) را نیز در مختصه keep ذکر کنیم موجب افزوده شدن دو بافر (یکی بعد از پایه یا پد x در شکل اخیر و دیگر قبل از پایه y) می‌شود.

اگر خطوط ۹ الی ۱۰ را از کد (با توضیح گذاری) حذف و آن را کامپایل کنیم، خروجی RTL Viewer در این حالت به صورت نشان داده شده در شکل ۷-۴-ج خواهد بود.

<sup>۱</sup> پس از اتمام موفقیت آمیز فرآیند سنتز می‌توانید از طریق منوی Synthesize-XST -> View RTL Schematic یک شماتیک RTL از طرح خود را مشاهده کنید.

```

1 -----
2 ENTITY delay_line IS
3   PORT (x: IN BIT;
4         y: OUT BIT);
5 END ENTITY;
6 -----
7 ARCHITECTURE example OF delay_line IS
8   SIGNAL a, b, c: BIT;
9   ATTRIBUTE keep: BOOLEAN;
10  ATTRIBUTE keep OF a, b, c: SIGNAL IS TRUE;
11 BEGIN
12   a <= NOT x;
13   b <= NOT a;
14   c <= NOT b;
15   y <= NOT c;
16 END ARCHITECTURE;
17 -----

```

### مختصه‌ی preserve

دو مختصه مفید که تاحدی شبیه به هم هستند عبارتند از `preserve` و `noprune`. مختصه `preserve` معادل مختصه‌ی `keep` است که برای مدارات رجیستر شده استفاده می‌شود نه ترکیبی. بنابراین این مختصه به منظور جلوگیری از حذف ثباتها (فلیپ‌فلاپ‌ها) به کار می‌رود. مثال بعدی این مختصه را توضیح بیشتری خواهد داد.

### مختصه‌ی noprune

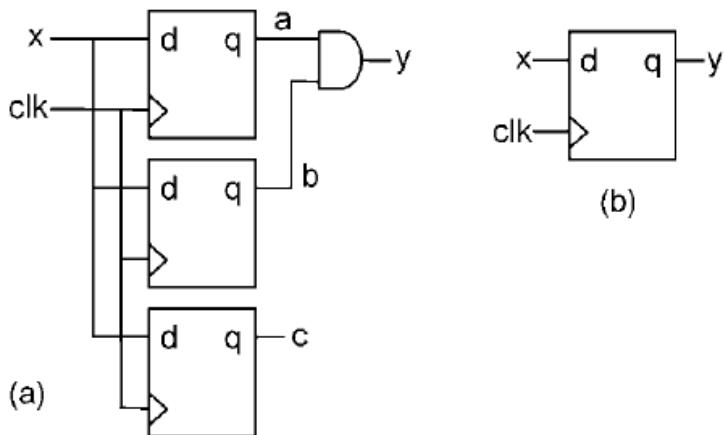
این مختصه علاوه بر رجیسترها (که وظیفه‌ی مختصه‌ی `preserve` بود) رجیسترها‌یی که مستقیماً به پورتهای موجودیت سطح رو (top-level ENTITY) متصل نباشند را نیز محافظت می‌کند (از ساده‌سازی و حذف آنها جلوگیری می‌کند). مثال بعدی دو مختصه‌ی معرفی شده‌ی اخیر را توضیح خواهد داد.

**مثال ۴-۵: حفظ رجیسترها زاید<sup>۱</sup> به کمک مختصه‌های `preserve` و `noprune`**

شکل ۴-۸-الف مداری شامل رجیسترها زاید (همپوشان یا redundant) را نشان می‌دهد. همان طور که ملاحظه می‌کنید می‌توان فلیپ‌فلاپ‌های مربوط به `a` و `b` را حذف و در یک فلیپ‌فلاپ ادغام نمود (مطابق با شکل ۴-۸-ب). همچنین توجه کنید که فلیپ‌فلاپ مربوط به `c` به جایی وصل نبوده و هیچ سیم خروجی موجودیت (مثلًا `y`) را تغذیه نمی‌کند لذا به راحتی می‌توان از آن صرفنظر کرد. یک کد VHDL جهت پیاده‌سازی مدار شکل ۴-۸-الف بنویسید.

<sup>۱</sup> Redundant

حل: یک کد VHDL در زیر نشان داده شده است. در خطوط ۹ الی ۱۲ از مختصه‌های اشاره شده استفاده شده است. اگر تنها از preserve استفاده کنیم (یعنی خطوط ۹ و ۱۰ حفظ و خطوط ۱۱ و ۱۲ را به توضیح تبدیل کنیم)، فلیپ فلاپهای a و b حفظ می‌شوند اما فلیپ فلاپ c حذف می‌شود زیرا این فلیپ فلاپ، موجودیت را تغذیه نکرده است. حال اگر تنها از مختصه‌ی noprunе استفاده کنیم (یعنی خطوط ۱۱ و ۱۲ محفوظ اما خطوط ۹ و ۱۰ به توضیح تبدیل شوند) آنگاه همه‌ی فلیپ فلاپها از جمله فلیپ فلاپ مربوط به c نیز حفظ خواهند شد.

**Figure 4.8**

(a) Circuit with redundant registers of example 4.5; (b) Equivalent simplified circuit.

```
1 -----
2 ENTITY redundant_registers IS
3     PORT (clk, x: IN BIT;
4             y: OUT BIT);
5 END ENTITY;
6 -----
7 ARCHITECTURE arch OF redundant_registers IS
8     SIGNAL a, b, c: BIT;
9     --ATTRIBUTE preserve: BOOLEAN;
10    --ATTRIBUTE preserve OF a, b, c: SIGNAL IS TRUE;
11    ATTRIBUTE noprune: BOOLEAN;
12    ATTRIBUTE noprune OF a, b, c: SIGNAL IS TRUE;
13 BEGIN
14     PROCESS (clk)
15     BEGIN
16         IF (clk'EVENT AND clk='1') THEN
17             a <= x;
18             b <= x;
19             c <= x;
20         END IF;
21     END PROCESS;
22     y <= a AND b;
23 END ARCHITECTURE;
24 -----
```

### مطالب و تغییرات جدید در VHDL 2008

تغییرات ایجاد شده در VHDL 2008 که مرتبط با مطالب این فصل هستند عبارتند از:

1) Regarding the logical operators:

For the types STD\_(U)LOGIC and STD\_(U)LOGIC\_VECTOR, additional options were included in the package *std\_logic\_1164* (part II of appendix I). For the types (UN)SIGNED, additional options were included in the package *numeric\_std* (part II of appendix J). Unary operations were also included. Logical operators were also defined for the new types UFIXED, SFIXED, and FLOAT.

2) Regarding the arithmetic operators:

For the types STD\_(U)LOGIC and STD\_(U)LOGIC\_VECTOR, arithmetic operators were defined in the new package *numeric\_std\_unsigned* (appendix N). For the types (UN)SIGNED, additional options were included in the package *numeric\_std* (part II of appendix J). Arithmetic operators were also specified for the new types UFIXED, SFIXED, and FLOAT.

3) Regarding the comparison operators:

Comparison operators were defined also for the new types BOOLEAN\_VECTOR, INTEGER\_VECTOR, UFIXED, SFIXED, and FLOAT.

4) Regarding the shift operators:

For the types STD\_(U)LOGIC\_VECTOR, some shift operators were included in the expansion of the package *std\_logic\_1164* (part II of appendix I). Other shift operators for STD\_(U)LOGIC\_VECTOR were introduced in the new package *numeric\_std\_unsigned* (appendix N). For the types (UN)SIGNED, additional shift operators were included in the expansion of the package *numeric\_std* (part II of appendix J). Shift operators were also defined for the new types BOOLEAN\_VECTOR, UFIXED, and SFIXED.

5) Regarding the matching comparison operators:

These operators (?=, ?/=, ?<, ?>, ?<=, ?>=) were all introduced in VHDL 2008. They include the types BIT, BIT\_VECTOR (partial set), BOOLEAN\_VECTOR, STD\_(U)LOGIC, STD\_(U)LOGIC\_VECTOR (whole set if proper package used), (UN)SIGNED, UFIXED, and SFIXED.

6) Others:

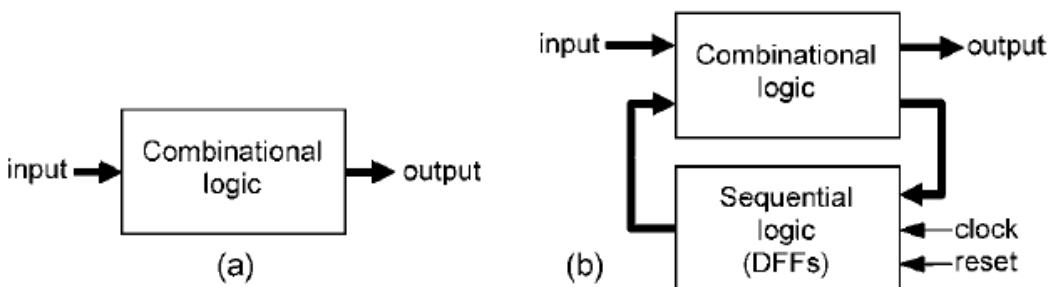
The functions MINIMUM, MAXIMUM, TO\_STRING, TO\_OSTRING, and TO\_HSTRING were also introduced in VHDL 2008, with support for nearly all VHDL types. Several new attributes for scalars, signals, etc. were included as well.



## فصل پنجم

# کد همزمان

تعریف مدارات ترتیبی و ترکیبی و تفاوت آنها



**Figure 5.1**

Models for (a) combinational and (b) sequential logic circuits.

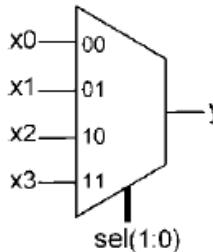
کدهای VHDL شامل دو نوع همزمان (یا موازی) و ترتیبی می‌باشند. تنها کدهایی که داخل FUNCTION، PROCEDURE، PROCESS باشند از نوع ترتیبی بوده و مابقی از نوع همزمان هستند. زبان VHDL ذاتاً زبانی موازی است. حتی کل بدنی PROCESS (و البته بدنی زیربرنامه‌ها) نیز به صورت موازی با بقیه اجزایی که خارج از فرآیند نوشته شده‌اند، اجرا می‌شود.

سه دستور کاملاً همزمان عبارتند از WHEN، SELECT و GENERATE. اینها حتماً باید خارج از بدنی یک کد ترتیبی نوشته شوند. این دستورات در این فصل بررسی می‌شوند. در مقابل نیز چهار دستور کاملاً ترتیبی عبارتند از IF، LOOP، WAIT و CASE. اینها حتماً باید خارج از بدنی یک کد همزمان نوشته شوند. این دستورات در فصل بعدی بررسی می‌شوند. کدهمzman فقط به منظور پیاده‌سازی مدارات ترکیبی استفاده می‌شود. اما از کدهای ترتیبی می‌توان برای پیاده‌سازی هر دو نوع مدار ترکیبی و ترتیبی استفاده کرد. توجه کنید که در داخل یک کد همزمان ترتیب دستورات اصلاً مهم نیست.

### ۲-۵ استفاده از عملگرها

اصولاً هر نوع مداری را می‌توان برپایه استفاده از عملگرها توصیف نمود. البته این روش تنها برای مدارات حسابی یا مدارات منطقی ساده مفید و عملی است. در مثال زیر از عملگرها برای ساخت یک مالتی پلکسر استفاده می‌شود.

مثال ۱-۵: پیاده‌سازی مالتی پلکسر با عملگرها  
تنها با استفاده از عملگرها منطقی یک مالتی پلکسر  $4 \times 1$  (شکل ۲-۵) پیاده‌سازی کنید.



**Figure 5.2**  
 $4 \times 1$  multiplexer of example 5.1.

حل: معادله‌ی منطقی خروجی چنین مالتی پلکسری به صورت زیر است:

$$y = sel'_1 \cdot sel'_0 \cdot x_0 + sel'_1 \cdot sel_0 \cdot x_1 + sel_1 \cdot sel'_0 \cdot x_2 + sel_1 \cdot sel_0 \cdot x_3$$

در معادله فوق تنها از عملگرها منطقی استفاده شده است. کد VHDL چنین مداری به صورت زیر است:

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY mux IS
6     PORT (x0, x1, x2, x3: IN STD_LOGIC;
7             sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8             y: OUT STD_LOGIC);
9 END mux;
10 -----
11 ARCHITECTURE operators_only OF mux IS
12 BEGIN
13     y <= (NOT sel(1) AND NOT sel(0) AND x0) OR
14         (NOT sel(1) AND sel(0) AND x1) OR
15         (sel(1) AND NOT sel(0) AND x2) OR
16         (sel(1) AND sel(0) AND x3);
17 END operators_only;
18 -----
19 -----

```

### ۳-۵ دستور WHEN

این دستور تقریباً معادل با دستور IF است. گرامر ساده آن به صورت زیر است:

```
assignment_expression WHEN conditions ELSE
    assignment_value WHEN conditions ELSE
    ...
;
```

### برخی مثالها

```
x <= '0' WHEN rst='0' ELSE
    '1' WHEN a='0' OR b='1' ELSE
    '-'; --don't care
y <= "00" WHEN (a AND b)="01" ELSE
    "11" WHEN (a AND b)="10" ELSE
    "ZZ"; --high impedance
```

در دستور WHEN از شرطهای چندتایی (عبارت‌های بولی) می‌توان استفاده کرد. برای این کار از عملگرهای AND، OR و یا NOT می‌توان استفاده کرد.

گرچه در دستور WHEN لازم نیست تمام حالت‌های ممکن از مقادیر ورودی را ذکر کنید اما در مورد مدارات ترکیبی (جدول صحت) توصیه می‌شود که حتماً تمام حالت‌های ممکن ورودی را ذکر کنید تا از وجود لچ‌ها در مدار پیاده‌سازی شده توسط کامپایلر جلوگیری کنید. در این حالت استفاده از کلمه کلیدی OTHERS مفید است.

یک کلمه کلیدی مفید دیگر، UNAFFECTED است. این کلمه کلیدی در موقعی استفاده می‌شود که بخواهیم در برخی حالات در خروجی تغییری رخ ندهد (یعنی حفظ حالت قبلی). بنابراین استفاده از این کلمه کلیدی معمولاً موجب تولید لچ‌هایی در مدار پیاده‌سازی شده می‌شود. لذا تنها زمانی از این کلمه کلیدی استفاده کنید که دخیل شدن عناصر حافظه در سیستم‌تان مطلوب و مورد نظرتان باشد.

مثال: هر دو کد نشان داده شده در زیر موجب پیاده‌سازی یک لچ D حساس به سطح بالا می‌شوند. اما کد سمت راست تمام حالات (یا مقادیر) ورودی را پوشش داده است و صراحتاً اعلام کرده است که مایل به وجود عناصر حافظه در مدار است اما کد سمت جپ تمام حالات را مشخص نکرده و وجود یا عدم وجود عناصر حافظه را به تصادف واگذار کرده است.

<code>q &lt;= '0' WHEN rst='1' ELSE</code>	<code>q &lt;= '0' WHEN rst='1' ELSE</code>
<code>    d WHEN clk='1';</code>	<code>    d WHEN clk='1' ELSE</code>
	<code>UNAFFECTED;</code>

**توجه:** در استاندارد VHDL 2008 اجازه‌ی استفاده از دستور WHEN در کدهای ترتیبی و تست عبارات بولی داده شده است.

#### ۴-۵ دستور SELECT

این دستور تقریباً معادل با دستور ترتیبی CASE است. گرامر ساده‌ی شده‌ای از آن به صورت زیر است:

```
WITH identifier SELECT
  assignment_expression WHEN values,
  assignment_value WHEN values,
  ...;
```

#### برخی مثالها

<pre>WITH control SELECT   y &lt;= "000" WHEN 0   1,   "100" WHEN 2 TO 5,   "Z--" WHEN OTHERS;</pre>	<pre>WITH (a AND b) SELECT   y &lt;= "00" WHEN "001",   "11" WHEN "100",   UNAFFECTED WHEN OTHERS;</pre>
--	--

همان طور که در مثال بالایی سمت چپ نشان داده شده است، در این دستور امکان استفاده از مقادیر چندگانه (به جای شرطهای چندتایی) وجود دارد. در این حالت تنها از عملگرها | (به معنای «یا») و TO (برای مشخص کردن بازه‌ای از مقادیر) باید استفاده شود. یعنی:

```
WHEN value1 | value2 | ...
WHEN value1 TO value2      --range
```

در دستور SELECT تمام مقادیر (یا حالات) ممکن ورودی باید مشخص شوند. در این کار از کلمه‌ی کلیدی مفید OTHERS می‌توان استفاده کرد.

همچنین در اینجا نیز از UNAFFECTED می‌توان استفاده نمود. اما در اینجا نیز مشابه با قبل باید به مساله استفاده از عناصر حافظه در مدار توجه نمود. به عنوان نکته آخر توجه کنید که هر تخصیص سیگنال (از جمله تخصیصهای انجام شده در این فصل) را می‌توان به همراه یک برچسب (قبل از انجام تخصیص) استفاده کرد اما این کار به ندرت استفاده می‌شود.

**توجه:** در VHDL 2008 امکان استفاده از دستور SELECT در داخل کدهای ترتیبی فراهم شده است. در این استاندارد دستور SELECT معرفی شده است که بعداً در بخش ۱۰-۵ توضیح داده خواهد شد.

#### مثال ۲-۵: پیاده‌سازی مالتی پلکسر با SELECT و WHEN

همان مالتی پلکسر مثال ۱-۵ را در اینجا به این صورت پیاده‌سازی کنید که ورودی مالتی پلکسر همان طور که در شکل ۳-۵ نشان داده شده است، N بیتی باشد و N را از طریق GENERIC پیاده‌سازی کنید. دو راه مختلف را نشان دهید: یکی بر مبنای استفاده از WHEN و دیگری بر مبنای استفاده از SELECT. نتایج شبیه‌سازی را نشان دهید.

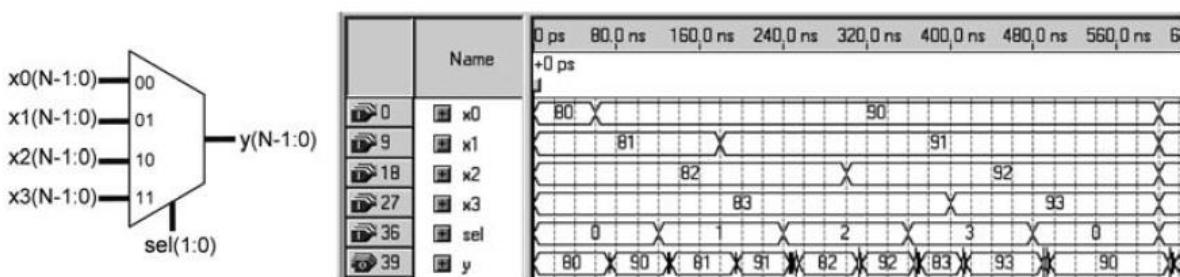


Figure 5.3  
4 × N multiplexer of example 5.2 and respective simulation results.

حل:

یک کد VHDL با دو معماری در زیر نشان داده شده است. پارامتر N به صورت GENERIC تعریف (خط ۶) و استفاده (خطوط ۷ و ۹) شده است. تنها از نوع STD\_LOGIC\_VECTOR استفاده شده است. در هر دو معماری، تمام حالات ممکن ورودی پوشش داده شده است.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY mux IS
6     GENERIC (N: INTEGER := 8);
7     PORT (x0, x1, x2, x3: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8             sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
9             y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
10 END ENTITY;
11 -----
12 ARCHITECTURE with_WHEN OF mux IS
13 BEGIN
14     y <= x0 WHEN sel="00" ELSE
15         x1 WHEN sel="01" ELSE
16         x2 WHEN sel="10" ELSE
17         x3;
18 END ARCHITECTURE;
19 -----
20 -----
21 ARCHITECTURE with_SELECT OF mux IS
22 BEGIN
23     WITH sel SELECT
24         y <= x0 WHEN "00",
25             x1 WHEN "01",
26             x2 WHEN "10",
27             x3 WHEN OTHERS;
28 END ARCHITECTURE;
29 -----
30 -----

```

نتایج شبیه‌سازی در شکل ۳-۵ نشان داده شده است. به وقوع خطای glitch توجه کنید. این اتفاق طبیعی است زیرا همزمان تعدادی سیگنال تغییر می‌کنند.

**توجه:** در کد فوق توجه کنید که برای یک موجودیت مشخص، در هر لحظه تنها یک معماری می‌تواند وجود داشته باشد. بنابراین، دو راه داریم: یکی این که در هر بار اجرای شبیه‌سازی تمام معماری‌های نوشته شده به جز معماری مد نظرمان را (با علامت --) توضیح‌گذاری کنیم. راه دوم استفاده از اعلان CONFIGURATION قبل از موجودیت و معماری‌ها است. نحوه‌ی استفاده از این اعلان برای مثال فوق در زیر نشان داده شده است:

```
-----  
CONFIGURATION which_mux OF mux IS  
  FOR with_WHEN  
    END FOR;  
END CONFIGURATION;
```

برطبق کد فوق، معماری with\_WHEN به همراه موجودیت mux استفاده شده و سایر معماری‌ها (در صورت وجود) نادیده گرفته می‌شوند. البته در این حالت کامپایلر گرامر تمام معماری‌ها (حتی معماری‌هایی که فعلاً مدنظر ما نیستند) را بررسی کرده و در صورت خطایی در هر کدام، آن را گزارش می‌دهد.

### مثال ۳-۵: ALU

این مثال مطالعه شود.

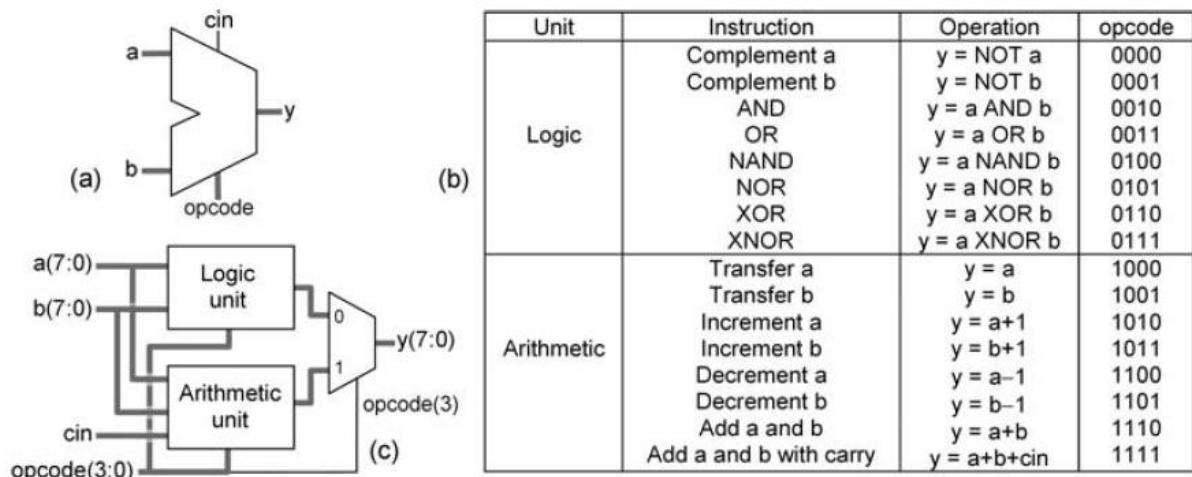


Figure 5.4  
ALU of example 5.3. (a) ALU symbol; (b) truth table; (c) a possible implementation.

An ALU (Arithmetic Logic Unit) is shown in figure 5.4(a), having  $a$ ,  $b$ ,  $cin$  (carry in), and  $opcode$  (operation code) as inputs, and  $y$  as output. The desired functionality is expressed in the truth table of figure 5.4(b), where each function is selected by a different value of  $opcode$ . Note that the upper eight instructions are logical, while the lower eight are arithmetic. Design this circuit using the concurrent statement SELECT, satisfying the following conditions:

- 1) The arithmetic operations must be *signed*.
- 2) The number of bits for inputs  $a$  and  $b$  must be *generic*.
- 3) All ports must be of type STD\_LOGIC(\_VECTOR) (industry standard).
- 4) Simulation results must also be included in the solution.

**Solution** Figure 5.4(c) shows a possible ALU implementation (among several other options). The circuit contains two main sections, called *logic* and *arithmetic* units, each controlled by the same three LSBs of  $opcode$ . The MSB of  $opcode$  is employed to control a multiplexer, letting the *logic* result out when low or the arithmetic result out if high.

A VHDL code for this circuit is presented below, under the title *alu* (line 6). The number of bits in  $a$  and  $b$  is a generic parameter (line 7), and all ports (lines 8–11) are of type STD\_LOGIC(\_VECTOR). Because the arithmetic operations were asked to be *signed*, the package *numeric\_std* (line 4) was included in the library/package declarations. The code proper is divided according to figure 5.4(c)—that is, a logic unit (lines 22–30), an arithmetic unit (lines 32–43), and a multiplexer (lines 45–47).



The implementation of the logic unit is straightforward. However, because the arithmetic unit must be *signed*, the same procedure used in the recommended solution of example 3.9 (see also recommendations in section 5.7) is adopted here; that is, the inputs are explicitly converted from STD\_LOGIC\_VECTOR to SIGNED (by type casting, in lines 32–33), they are then processed, and finally the result is converted back to STD\_LOGIC\_VECTOR (at the mux input, line 47, again by type casting).

Note also that because *cin* is STD\_LOGIC, not SIGNED, NATURAL, or INTEGER, it could not participate directly in the sum of line 43 (observe in the package *numeric\_std*, in appendix J, that the overloaded operator "+" does not contain the SIGNED + STD\_LOGIC option), so a small integer (lines 19 and 34) was created to allow the sum.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.numeric_std.all;
5 -----
6 ENTITY alu IS
7     GENERIC (N: INTEGER := 8); --word bits
8     PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
9            cin: IN STD_LOGIC;
10           opcode: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
11           y: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
12 END ENTITY;
13 -----
14 ARCHITECTURE alu OF alu IS
15     SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
16     SIGNAL y_sig: SIGNED(N-1 DOWNTO 0);
17     SIGNAL y_unsig: STD_LOGIC_VECTOR(N-1 DOWNTO 0);
18     SIGNAL small_int: INTEGER RANGE 0 TO 1;
19 BEGIN
20     -----
21     -----Logic unit:-----
22     WITH opcode(2 DOWNTO 0) SELECT
23         y_unsig <= NOT a WHEN "000",
24                         NOT b WHEN "001",
25                         a AND b WHEN "010",
26                         a OR b WHEN "011",
27                         a NAND b WHEN "100",
28                         a NOR b WHEN "101",
29                         a XOR b WHEN "110",
30                         a XNOR b WHEN OTHERS;
31     -----
32     -----Arithmetic unit:-----
33     a_sig <= SIGNED(a);
34     b_sig <= SIGNED(b);
35     small_int <= 1 WHEN cin='1' ELSE 0;

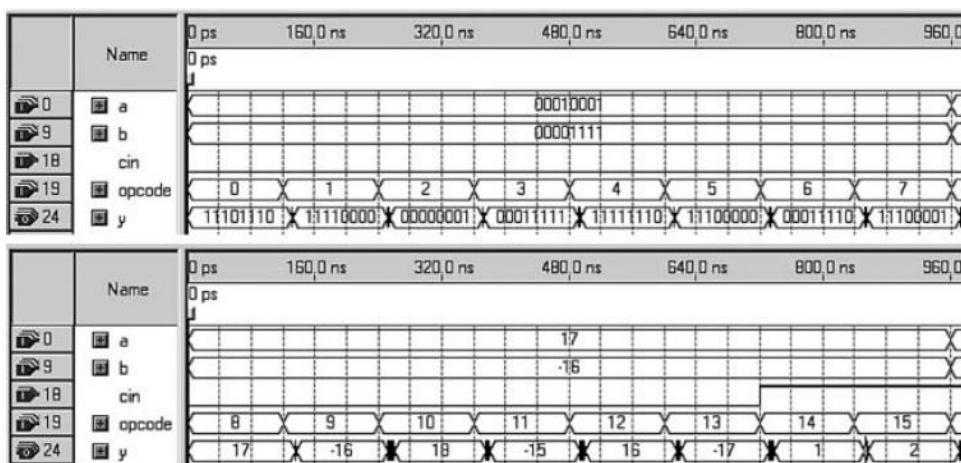
```

```

35      WITH opcode(2 DOWNTO 0) SELECT
36          y_sig <= a_sig WHEN "000",
37                      b_sig WHEN "001",
38                      a_sig + 1 WHEN "010",
39                      b_sig + 1 WHEN "011",
40                      a_sig - 1 WHEN "100",
41                      b_sig - 1 WHEN "101",
42                      a_sig + b_sig WHEN "110",
43                      a_sig + b_sig + small_int WHEN OTHERS;
44      -----Mux:-----
45      WITH opcode(3) SELECT
46          y <= y_unsig WHEN '0',
47                      STD_LOGIC_VECTOR(y_sig) WHEN OTHERS;
48  END ARCHITECTURE;
49  -----

```

Simulation results are depicted in figure 5.5. The upper graph is for logic instructions, while the lower graph exhibits results from arithmetic operations. The reader is invited to examine both to check the correct circuit operation.



**Figure 5.5**  
Simulation results from the ALU of example 5.3 (logic instruction in the upper graph, arithmetic instructions in the lower graph).



**توجه:** برای تبدیل نوع میتوان از جدول زیر استفاده کرد:

From	To	Type conversion function	Package of origin
INTEGER	STD_LOGIC_VECTOR	conv_std_logic_vector(a, cs)	std_logic_arith
	UNSIGNED	to_unsigned(a, cs) conv_unsigned(a, cs)	numeric_std std_logic_arith
	SIGNED	to_signed(a, cs) conv_signed(a, cs)	numeric_std std_logic_arith
	UFIXED	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
BIT_VECTOR	STD_LOGIC_VECTOR	to_stdlogicvector(a, cs)	std_logic_1164
STD_LOGIC_VECTOR	INTEGER	conv_integer(a, cs) conv_integer(a, cs) to_integer(a, cs)	std_logic_signed std_logic_unsigned numeric_std_unsigned
	BIT_VECTOR	to_bitvector(a, cs)	std_logic_1164
	UNSIGNED	unsigned(a) (*) unsigned(a) (*)	numeric_std std_logic_arith
	SIGNED	signed(a) (*) signed(a) (*)	numeric_std std_logic_arith
	UFIXED	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED	to_sfixed(a, cs)	fixed_generic_pkg
FLOAT	FLOAT	to_float(a, cs)	float_generic_pkg
UNSIGNED and SIGNED	INTEGER	to_integer(a, cs) conv_integer(a, cs)	numeric_std std_logic_arith
	STD_LOGIC_VECTOR	std_logic_vector(a) (*) std_logic_vector(a) (*) conv_std_logic_vector(a, cs)	numeric_std std_logic_arith std_logic_arith
	UNSIGNED	conv_unsigned(a, cs)	std_logic_arith
	SIGNED	conv_signed(a, cs)	std_logic_arith
	UFIXED (unsigned only)	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED (signed only)	to_sfixed(a, cs)	fixed_generic_pkg
FLOAT	FLOAT	to_float(a, cs)	float_generic_pkg
UFIXED and SFIXED	INTEGER	to_integer(a, cs)	fixed_generic_pkg
	STD_LOGIC_VECTOR	to_slv(a, cs)	fixed_generic_pkg
	UNSIGNED (ufixed only)	to_unsigned(a, cs)	fixed_generic_pkg
	SIGNED (sfixed only)	to_signed(a, cs)	fixed_generic_pkg
	SFIXED (ufixed only)	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
FLOAT	INTEGER	to_integer(a, cs)	float_generic_pkg
	STD_LOGIC_VECTOR	to_slv(a, cs)	float_generic_pkg
	UNSIGNED	to_unsigned(a, cs)	float_generic_pkg
	SIGNED	to_signed(a, cs)	float_generic_pkg
	UFIXED	to_ufixed(a, cs)	float_generic_pkg
	SFIXED	to_sfixed(a, cs)	float_generic_pkg

(a, cs) = (argument, conversion specifications)  
 cs may include vector size, left/right range constants, overflow and rounding specs, etc. (consult package)  
 (\*) = type casting

Figure 3.10  
 Main type-conversion options (type casting and type-conversion functions).

توجه: منظور از تغییر نقش (Type Casting) تبدیل بین انواعی است که نوع پایه در هر دو یکی است. تغییر نقش حالت خاصی از تبدیل نوع است که اشاره شد.

## Type Casting

UNSIGNED/SIGNED have the same base type (STD\_LOGIC) and indexing (NATURAL) as STD\_LOGIC\_VECTOR, so the following direct conversions using type casting are allowed:

- (UN)SIGNED(arg), where the argument is STD\_LOGIC\_VECTOR
- STD\_LOGIC\_VECTOR(arg), where the argument is SIGNED or UNSIGNED.

## Examples

```
-----
unsig <= UNSIGNED(slv);
sig <= SIGNED(slv);
slv1 <= STD_LOGIC_VECTOR(unsig);
slv2 <= STD_LOGIC_VECTOR(sig);
-----
```

**توجه:** یکی از فواید تبدیل نوع در رفع ابهام است. به مثال زیر توجه کنید:

**Example** Say that we want to perform the *signed* sum below, where *a*, *b*, and *sum* are all signed values:

```
sum <= a + b + "1000";
```

The problem is that there are two possible values for "1000": 8 if unsigned, -8 if signed. This conflict is resolved by the qualified expression below, which determines that "1000" is a signed value:

```
sum <= a + b + SIGNED'("1000");
```

## دستور GENERATE

دستور GENERATE یک دستور همزمان بوده و معادل با دستور ترتیبی LOOP است. وظیفه‌ی این دستور تکرار تعدادی دستور به تعداد معین است. داخل این دستور می‌توان از دستور IF استفاده کرد؛ در این حالت، دستور «GENERATE شرطی» گفته می‌شود. این حالت نیز مشابه با حالت ترکیب دستورات LOOP و IF است. هر یک از این دو حالت می‌توانند به صورت تو در تو داخل دیگری استفاده شوند.

دستور «GENERATE غیرشرطی» (یا دستور FOR-GENERATE هم گفته می‌شود) به منظور تکرار بلوکی از دستورات به تعداد معین استفاده می‌شود. گرامر ساده‌ی آن به صورت زیر است. در این گرامر توجه کنید که استفاده از یک برچسب ضروری بوده و استفاده از کلمه‌ی BEGIN نیز تنها در صورتی ضروری است که اعلان یا اعلانهایی صورت گرفته باشد.

```

label: FOR identifier IN range GENERATE
    [declarative_part
BEGIN]
    concurrent_statements_part
END GENERATE [label];

```

مثال:

در مثال زیر ابتدا سه سیگنال تعریف شده و سپس یک تکه کد سه بار به کمک GENERATE تکرار می‌شود. در هر سه تکرار از یک بروچسب مشترک با نام gen و از یک متغیر مشترک با نام i استفاده شده است. محدوده‌ی مقادیر متغیر حلقه نیز یا 0 To 7 یا 7 DOWNTO 0 است.

```

-----
SIGNAL a, b, x: BIT_VECTOR(7 DOWNTO 0);
-----
gen: FOR i IN 0 TO 7 GENERATE
    x(i) <= a(i) XOR b(7-i);
END GENERATE;
-----
gen: FOR i IN a'RANGE GENERATE
    x(i) <= a(i) XOR b(7-i);
END GENERATE;
-----
gen: FOR i IN a'REVERSE_RANGE GENERATE
    x(i) <= a(i) XOR b(7-i);
END GENERATE;
-----
```

دستور GENERATE شرطی (یا دستور IF-GENERATE نیز نامیده می‌شود) شامل یک دستور GENERATE در حلقه‌ی IF می‌باشد. گرامر ساده‌ای از آن به صورت زیر می‌باشد:

```

label: IF condition GENERATE
    [declarative_part
BEGIN]
    concurrent_statements_part
END GENERATE [label];

```

این نسخه از دستور GENERATE کمتر با اقبال و علاقه روبرو می‌شود. برای افزایش جذابیت و اقبال این دستور ویژگی‌های جدیدی در VHDL 2008 به آن اضافه شده است که شامل (۱) استفاده از ELSIF/ELSE، (۲) استفاده از برچسب‌های جایگزین و کلمه‌ی END قبل از CASE-GENERATE و (۳) استفاده از گزینه‌ی اختیاری GENERATE بیشتر در بخش ۱۰-۵ می‌باشد.

یک نکته‌ی مهم دیگر در مورد دستور GENERATE (و البته مشابه‌اً در مورد LOOP که بعداً بررسی خواهد شد) این است که حدود آن باید ایستا<sup>۱</sup> باشند. برای مثال اگر در تکه کد زیر، x یک ورودی (و بنابراین یک پارامتر غیرایستا) باشد، آن‌گاه کد زیر ممکن است قابل سنتز نباشد.

```
-----  
NotOK: FOR i IN 0 TO x GENERATE  
      ...  
    END GENERATE;  
-----
```

در مورد سیگنالهای چندمحرکه<sup>۲</sup> نیز این مطلب اهمیت دارد. هانطور که بعداً خواهیم دید، تخصیص چندگانه‌ی همزمان به VARIABLE مجاز است زیرا مقدار متغیرها بلافاصله به روزرسانی می‌شود اما این کار برای سیگنال‌ها مجاز نیست. این مساله در مثال زیر نشان داده شده است. در فصل ۷ یک راه برای غلبه بر این مشکل نشان داده خواهد شد.

مثال:

اولین بخش از کد زیر (پس از تعریف سیگنال‌ها) مجاز و معتبر هستند زیرا به هر بیت از بیتهاي سیگنال x تنها یک بار عمل تخصیص را انجام می‌دهند. اما بخش دوم معتبر نمی‌باشد زیرا یک مقدار چندین بار به سیگنال y تخصیص داده شده است. به طور مشابه بخش سوم نیز غیرمجاز است زیرا سیگنال z چندین مقدار را همزمان به خود گرفته است.

<sup>1</sup> Static

<sup>2</sup> Multiple-Driven Signals

```
SIGNAL a, b, x, y: BIT_VECTOR(3 DOWNTO 0);
SIGNAL z: INTEGER RANGE 0 TO 7;

-----  
OK: FOR i IN x'RANGE GENERATE
    x(i)<='1' WHEN (a(i) AND b(i))='1' ELSE '0';
END GENERATE;  
  
-----  
NotOK: FOR i IN y'LOW TO y'HIGH GENERATE
    y <="1111" WHEN (a(i) AND b(i))='1' ELSE
        "0000";
END GENERATE;  
  
-----  
NotOK: For i IN 0 TO 3 GENERATE
    z <= z + 1 WHEN a(i)='1';
END GENERATE;
```

در ادامه دو مثال کامل از نحوه طراحی مبتنی بر GENERATE آورده شده است.

مثال ۴-۵: رمزگشایی عام آدرس مبتنی بر GENERATE  
در این مثال همان رمزگشایی عام آدرس مربوط به مثال ۴-۲ را دوباره طراحی می‌کنیم اما در اینجا با استفاده از تنها نوع .STD\_LOGIC\_VECTOR

**Solution** A VHDL code for this problem is presented below. All ports are STD\_LOGIC-based (lines 8–10). The GENERATE statement is employed in lines 17–19, containing just one assignment (using WHEN, line 18). Note that *address* was converted into an integer in line 16 using a functions available in the package *std\_logic\_unsigned* (see figure 3.10). Lines 14 and 16 can obviously be suppressed if we choose to write "... WHEN i=conv\_integer(address) ..." in line 18. The size of the code is fixed, regardless of the number of input bits. Simulation results are similar to those in figure 2.7.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_unsigned.all;
5 -----
6 ENTITY address_decoder IS
7     GENERIC (N: NATURAL := 3); --number of address bits
8     PORT (address: IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
9            ena: IN STD_LOGIC;
10           word_line: OUT STD_LOGIC_VECTOR(2**N-1 DOWNTO 0));
11 END ENTITY;
12 -----
13 ARCHITECTURE decoder OF address_decoder IS
14     SIGNAL addr: NATURAL RANGE 0 TO 2**N-1;
15 BEGIN
16     addr <= conv_integer(address);
17     gen: FOR i IN word_line'RANGE GENERATE
18         word_line(i)<='0' WHEN i=addr AND ena='1' ELSE '1';
19     END GENERATE;
20 END ARCHITECTURE;
21 -----

```

یک کاربرد مفید دیگر از دستور GENERATE در هنگام نمونه‌سازی اجزاء<sup>۱</sup> و به منظور ساختن مدارات بزرگتر می‌باشد. گرچه دستور COMPONENT بعداً در فصل ۸ معرفی خواهد شد اما در اینجا به منظور آشنایی اولیه با آن و نمایش نحوه استفاده از GENERATE در این دستور مثال زیر آورده شده است.

### مثال ۵-۵: نمونه‌سازی COMPOENET به همراه GENERATE

در شکل ۵-۶ نحوی ساخت یک مالتی پلکسر بزرگتر از روی چندین نمونه از یک مالتی پلکسر ساده و پایه‌ای نشان داده شده است. در شکل (الف) یک مالتی پلکسر  $1 \times 2$  نشان داده شده است. این مالتی پلکسر در شکل (ب) سه بار نمونه‌سازی شده است تا مطابق با شکل (ج) یک مالتی پلکسر « $1 \times 2$  سه‌تایی» ایجاد شود. این مالتی پلکسر را با روش ساختاری<sup>۲</sup> (یعنی به کمک (COMPONENT) طراحی و از دستور GENERATE به منظور نمونه‌سازی استفاده کنید).

<sup>1</sup> Components

<sup>2</sup> Structural

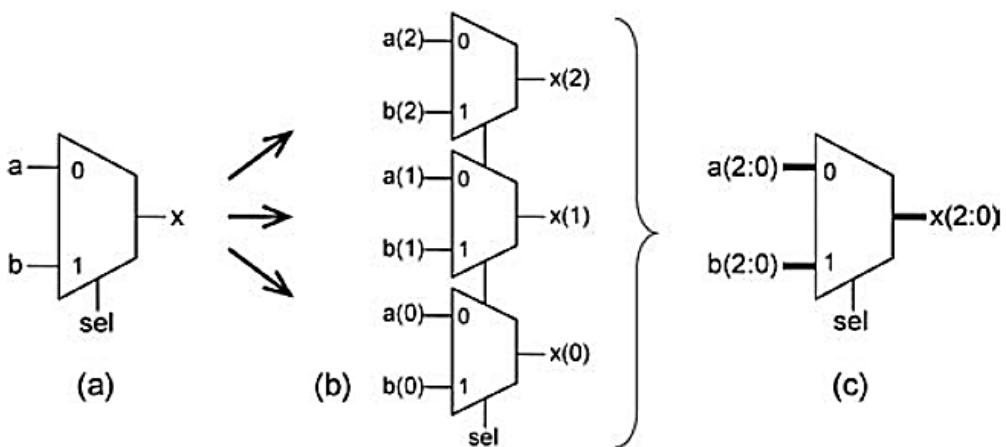


Figure 5.6

(a) A  $2 \times 1$  mux that (b) instantiated three times (c) creates a  $2 \times 3$  mux.

**Solution** A VHDL code for this circuit is shown below, consisting of two parts. The first part builds the basic unit (*mux2x1*), which is then instantiated in the second part (main code) using the GENERATE statement (lines 18–20). (Details on COMPONENT construction and usage will be seen in chapter 8.)

```

1 -----The component (mux2x1):-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY mux2x1 IS
6     PORT (a, b, sel: IN STD_LOGIC;
7             x: OUT STD_LOGIC);
8 END ENTITY;
9 -----
10 ARCHITECTURE mux2x1 OF mux2x1 IS
11 BEGIN
12     x<=a WHEN sel='0' ELSE b;
13 END ARCHITECTURE;
14 ----

1 -----Main code:-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY mux2x3 IS
6     PORT (a, b: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
7             sel: IN STD_LOGIC;
8             x: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
9 END ENTITY;
10 -----
11 ARCHITECTURE mux2x3 OF mux2x3 IS
12     ---Component declaration:---
13     COMPONENT mux2x1 IS
14         PORT (a, b, sel: IN STD_LOGIC; x: OUT STD_LOGIC);
15     END COMPONENT;
16     BEGIN
17         ---Component instantiation:---
18         generate_mux2x3: FOR i IN 0 TO 2 GENERATE
19             comp: mux2x1 PORT MAP (a(i), b(i), sel, x(i));
20         END GENERATE generate_mux2x3;
21     END ARCHITECTURE;
22 -----

```

## ۵-۶ پیادهسازی مدارات ترتیبی به کمک کد همزمان

در اصل، کدهای همزمان تنها برای پیادهسازی مدارات ترکیبی استفاده می‌شوند. اما از آنجایی که (۱) «هر مدار دیجیتال» را می‌توان تنها با استفاده از گیت NOR یا NAND ساخت، (۲) مدارات ترتیبی نیز نوعی از مدارات دیجیتال هستند، و (۳) گیت‌های NAND و NOR به کمک

کدهای همزمان پیاده‌سازی می‌شوند، بنابراین از کدهای همزمان به تنها یکی می‌توان برای ساخت مدارات ترتیبی استفاده کرد اما در حالت کلی این روش فقط برای مدارات ترتیبی ساده معقول و قابل به کارگیری است زیرا برای مدارات پیچیده‌تر، کد بسیار طولانی‌تر شده، نوشتند و خطایابی آن پیچیده‌تر و دشوارتر خواهد شد، و ردگیری و فهم آن از حالت طبیعی خارج خواهد شد. بنابراین، نتیجه این که: «استفاده از کدهای همزمان برای پیاده‌سازی مدارات ترتیبی توصیه نمی‌شود».

#### Example 5.6: DFF Implemented with Concurrent Code

Figure 5.7 shows, on the left, a pair of multiplexers (combinational circuits), whose connections emulate a DFF (a sequential circuit), shown on the right. Write a VHDL code for this circuit and examine the expressions inferred by the compiler.

**Solution** A corresponding VHDL code is presented below. The intermediate signal  $p$  is specified in line 8, and the multiplexers are implemented in lines 10–11 using the WHEN statement. Looking at the compilation report (fitter equations), the expected result is the inference of two serially connected latches, because a multiplexer with a feedback loop (as in figure 5.7) emulates a latch. In conclusion, the resulting circuit should present the correct functionality, but because of its construction, it is slower than an actual, prefabricated DFF (the latter is optimized to operate specifically as a DFF).

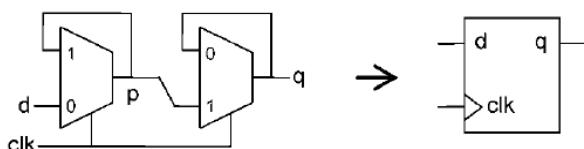


Figure 5.7  
DFF implemented with multiplexers (example 5.7).

```

1
2 ENTITY concurrent_dff IS
3     PORT (d, clk: IN BIT;
4             q: BUFFER BIT);
5 END ENTITY;
6
7 ARCHITECTURE concurrent OF concurrent_dff IS
8     SIGNAL p: BIT;
9 BEGIN
10    p <= d WHEN clk='0' ELSE p; --1st mux
11    q <= p WHEN clk='1' ELSE q; --2nd mux
12 END ARCHITECTURE;
13

```

۷-۵ پیاده‌سازی مدارات ریاضی با عملگرهای  
مدارات حسابی معمولاً تنها به کمک عملگرهای ریاضی  
 $+, -, *, /, ABS, REM, MOD$

پیاده‌سازی می‌شوند. چهار عملگر اول ذکر شده در لیست فوق متداول‌ترین عملگرهایی است که تاکنون استفاده کرده‌ایم. لذا در این بخش توجه ویژه‌ای به آنها کرده‌ایم.

مهمترین نوع داده‌ای که برای پورت (PORT) مدارات حسابی به عنوان واسط اتصال<sup>۱</sup> استفاده می‌شود، نوع INTEGER و نوع STD\_LOGIC\_VECTOR است که بهتر است از اولی برای پورت استفاده نشود؛ نوع دوم یک نوع استاندارد صنعتی است. از طرف دیگر تا جایی که به «داخل» مدارات حسابی مرتبط است، انواع SIGNED و UNSIGNED ترجیح داده می‌شوند. این انواع در دو بسته‌ی رقیب numeric\_std (بسته‌ی استاندارد و توصیه شده) و std\_logic\_arith می‌باشند.

برای استفاده از عملگرهای ریاضی فوق‌الذکر توجه به اندازه‌ی (یا تعداد بیت‌های) خروجی حاصل از عملیات ریاضی نیز مهم است. برای مثال برای چهار عملگر نام برد شده، اندازه‌ی خروجی عملگر با توجه به تعاریف این عملگرها در بسته‌ی numeric\_std به قرار زیر است. در کدهای زیر، ورودی‌ها و خروجی‌ها از نوع SIGNED در نظر گرفته شده‌اند. نمادهای R و L نیز به ترتیب عملوند‌های سمت راست و سمت چپ عملگر مورد بررسی می‌باشند.

```
FUNCTION "+" (L, R: SIGNED) RETURN SIGNED;
--Result SUBTYPE: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 DOWNTO 0)

FUNCTION "-" (L, R: SIGNED) RETURN SIGNED;
--Result SUBTYPE: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 DOWNTO 0)

FUNCTION "*" (L, R: SIGNED) RETURN SIGNED;
--Result SUBTYPE: SIGNED((L'LENGTH+R'LENGTH-1) DOWNTO 0)

FUNCTION "/" (L, R: SIGNED) RETURN SIGNED;
--Result SUBTYPE: SIGNED(L'LENGTH-1 DOWNTO 0)
```

با توجه به تعاریف فوق ملاحظه می‌کنید که:

- 1) For "+" and "-": The size of the result must be equal to the size of the largest operand.
- 2) For "\*": The size of the result must be equal to the sum of the operands' sizes.
- 3) For "/": The size of the result must be equal to the size of the numerator.

همان‌گونه که اشاره شد، در مورد مدارات ریاضی یک مساله‌ی مهم، ذات و نوع درونی آنها است که این نوع درونی می‌تواند علامت‌دار و یا بدون علامت باشد. اعداد منفی به شکل متمم ۲ نمایش داده می‌شوند. بنابراین، ذات عملیات جمع و تفریق یکی است. برای مثال برای یک سیستم

علامت‌دار ۴ بیتی:

<sup>1</sup> Interface

$$5 + 2 = "0101" + "0010" = "0111" = 7$$

$$5 - 2 = 5 + (-2) = "0101" + "1110" = "0011" = 3$$

$$5 - (-2) = 5 + 2 = "0111" = 7$$

$$-5 + 2 = "1011" + "0010" = "1101" = -3$$

ضرب و تقسیم اعداد علامتدار نیز مستلزم بهره‌گیری از سیستم نمایش متمم ۲ است. یک عدد منفی باید متمم ۲ شود تا قدر مطلق اندازه‌ی خود را بدست آورد. در پایان عملیات ضرب و تقسیم، نتیجه‌ی عملیات متناسب با علامت مورد انتظار در حالتی که علامت دو عملوند ورودی متفاوت باشد، باید متمم ۲ شود. برای مثال برای همان سیستم علامتدار ۴ بیتی:

$$5*3 = "0101" * "0011" = "00001111" = 15$$

$$-5*3 = -(5*3) = -("0101" * "0011") = -("00001111") = "11110001" = -15$$

$$5/3 = "0101" / "0011" = "0001" = 1$$

$$-5/3 = -(5/3) = -("0101" / "0011") = -("0001") = "1111" = -1$$

شکل ۸-۵ سه دیاگرام مختلف برای یک جمع‌کننده/تفریق‌کننده را نشان می‌دهد. در اولین دیاگرام، شکل (الف)، عملوندها a و b (هرکدام ۴ بیتی) بوده، بیت نقلی ورودی cin، مجموع و cout\_sub و بیت نقلی خروجی sum، خروجی تفریق و بیت نقلی خروجی نیز cout\_sum و sub می‌باشند.

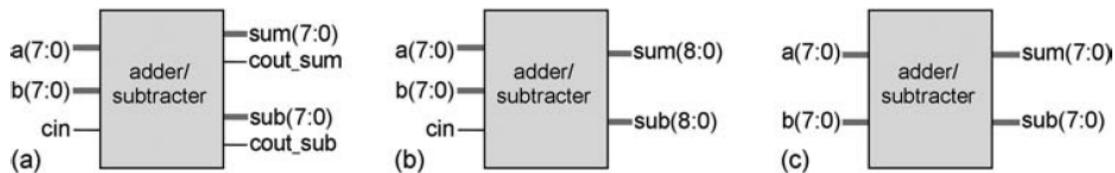


Figure 5.8

Adder/subtractor with the carry-out bits (a) separated from sum and sub, (b) grouped with sum and sub, and (c) nonexistent. The first two are equivalent and not subject to overflow, whereas in the last one overflow can occur.

یک نمایش معادل در شکل (ب) نمایش داده شده است. در این نوع نمایش، بیت‌های نقلی خروجی با sum و sub ترکیب شده‌اند (توجه کنید که حالا این ترکیب شامل ۹ بیت بوده و بیت MSB شامل cout است). در طرح شکل (ج) هیچ بیت نقلی وجود نداشته و بنابراین این مدار در معرض سرریز است.

فرض کنید که سیستم بدون علامت است. در این حالت برای محاسبه sum در شکل ۸-۵-الف می‌توان به صورت زیر عمل کرد:

```

-----  

SIGNAL a_uns, b_uns, sum: UNSIGNED(7 DOWNTO 0);  

SIGNAL sum_uns: UNSIGNED(9 DOWNTO 0);  

SIGNAL cin, cout_sum: STD_LOGIC;  

sum_uns <= ('0' & a_uns & cin) + ('0' & b_uns & '1');  

sum <= sum_uns(8 DOWNTO 1);  

cout_sum <= sum_uns(9);  

-----  


```

در کد فوق به این نکته توجه کنید که عملوند `sum_uns` دارای ۱۰ بیت است که از طریق افزودن `'1'` یا `'0'` در سمت راست و `'0'` در سمت چپ حاصل شده است. بیت `LSB` از `sum_uns` نادیده گرفته شده و بیت `MSB` از `sum_uns` نیز برابر `cout_sum` است.  
حال اگر سیستم را علامت دار در نظر بگیریم، از کد زیر می‌توانیم استفاده کنیم:

```

-----  

SIGNAL a_sig, b_sig, sum: SIGNED(7 DOWNTO 0);  

SIGNAL sum_sig: SIGNED(9 DOWNTO 0);  

SIGNAL cin, cout_sum: STD_LOGIC;  

sum_sig <= (a_sig(7) & a_sig & cin) + (b_sig(7) & b_sig & '1');  

sum <= sum_sig(8 DOWNTO 1);  

cout_sum <= sum_sig(9);  

-----  


```

تنها تفاوت این کد با کد قبلی در بیت توسعه‌ی علامت<sup>۲</sup> است. در اینجا این بیت باید یک کپی از بیت منتها الیه سمت چپ باشد.

مشکلی که در دو رویکرد فوق (سیستمهای بدون علامت و علامت‌دار) وجود دارد، عدم کارایی آن برای تفریق است. تفریق در بیشتر سیستم‌ها به ویژه سیستم‌های علامت‌دار ضروری است. لذا به رویکرد کلی تری نیاز داریم (تا تفریق را پوشش دهد). یک رویکرد در زیر نشان داده شده است. در این کد به عبارت مربوط به `sub_sig` توجه کنید که در آن تمام سه عملوند (از جمله `cin`) توسعه‌ی علامت داده شده‌اند (برای `cin` از `'0'` استفاده شده است زیرا یک عدد نامنفی است). با این رویکرد هر دو حالت جمع و تفریق قابل انجام است. طراحی کامل بعداً به زودی بررسی خواهد شد.

<sup>1</sup> Appending<sup>2</sup> Sign Extension Bit

```

-----  

SIGNAL a_sig, b_sig, sum, sub: SIGNED(7 DOWNTO 0);  

SIGNAL sum_sig, sub_sig: SIGNED(8 DOWNTO 0);  

SIGNAL cin, cout_sum, cout_sub: STD_LOGIC;  

sum_sig <= (a_sig(7) & a_sig) + (b_sig(7) & b_sig) + ('0' & cin);  

sum <= sum_sig(7 DOWNTO 0);  

cout_sum <= sum_sig(8);  

sub_sig <= (a_sig(7) & a_sig) - (b_sig(7) & b_sig) + ('0' & cin);  

sub <= sub_sig(7 DOWNTO 0);  

cout_sub <= sub_sig(8);
-----  


```

**برخی توصیه‌ها:** با توجه به مطالب و مثالهای بیان شده تا کنون، نتایج زیر به عنوان جمع‌بندی به دست می‌آیند:

- ۱) برای پورتهای تنها از نوع STD\_LOGIC\_VECTOR استفاده کنید.
- ۲) در داخل مدار، تنها از نوع SIGNED(UN) استفاده کنید.
- ۳) برای انواع SIGNED(UN) از بسته‌ی استاندارد numeric\_std (استاندارد IEEE) استفاده کنید.
- ۴) قبل از انجام هر محاسبه‌ای داده‌های خود را صراحتاً از نوع STD\_LOGIC\_VECTOR به نوع SIGNED(UN) تبدیل کنید. این کار را به کمک تغییر نقش انواع (Type Casting) انجام دهید.
- ۵) محاسبات خود را انجام دهید.
- ۶) در پایان، نتایج خود را مجدداً به نوع STD\_LOGIC\_VECTOR برگردانید. برای این کار نیز مجدداً از تغییر نقش انواع استفاده کنید.

**مثال: پیاده‌سازی جمع‌کننده/تفريق‌کننده به شیوه‌ی توصیه شده**  
یک کد VHDL برای پیاده‌سازی مدار جمع‌کننده/تفريق‌کننده شکل ۸-۵-الف یا ۸-۵-ب بنویسید (از نظر فیزیکی این دو مدار یکسان و معادل هم هستند). در طراحی خود از توصیه‌های انجام شده در پاراگراف قبلی استفاده کنید. فرض کنید که این مدار بخشی از یک سیستم علامت‌دار است.

حل: در زیر یک کد VHDL با عنوان signed\_add\_sub (خط ۶) نشان داده شده است. از آنجاکه مدار از نوع سیستم علامت‌دار است، از بسته‌ی numeric\_std در خط ۴ استفاده شده است. تعداد بیت‌های عملوندها به صورت یک پارامتر عام GENERIC (انتخاب شده است. نوع تمام پورت‌ها STD\_LOGIC\_VECTOR انتخاب شده است (خطوط ۸ الی ۱۲). پورت‌های ورودی شامل a، b و cin و پورت‌های خروجی شامل sum و sub می‌باشند.

The code proper (lines 20–33) is organized in four parts. In the first part (lines 21–22), the operands are explicitly converted to SIGNED. In the second part (lines 24–25), they are added and subtracted, with carry-in included. In the third part (lines 27–28), the signals are converted back to STD\_LOGIC\_VECTOR, with the carry-out bits grouped with *sum* and *sub*, as in figure 5.8(b). The forth part (lines 30–33) is equivalent to the third, just with the carry-out bits separated from *sum* and *sub*, as in figure 5.8(a). To have the code resemble figure 5.8(a) instead of 5.8(b), just comment out lines 10, 27, and 28 and uncomment lines 11–12 and 30–33. Simulation results are shown in figure 5.9.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.numeric_std.all;
5 -----
6 ENTITY signed_add_sub IS
7     GENERIC (N: INTEGER := 4); --number of input bits
8     PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
9             cin: IN STD_LOGIC;
10            sum, sub: OUT STD_LOGIC_VECTOR(N DOWNTO 0));
11           --sum, sub: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
12           --cout_sum, cout_sub: OUT STD_LOGIC);
13 END ENTITY;
14 -----
15 ARCHITECTURE signed_add_sub OF signed_add_sub IS
16     SIGNAL a_sig, b_sig: SIGNED(N-1 DOWNTO 0);
17     SIGNAL sum_sig, sub_sig: SIGNED(N DOWNTO 0);
18 BEGIN
19     -----convert to signed:-----
20     a_sig <= signed(a);
21     b_sig <= signed(b);
22     -----add and subtract:-----
23     sum_sig <= (a_sig(N-1) & a_sig) + (b_sig(N-1) & b_sig) + ('0' & cin);
24     sub_sig <= (a_sig(N-1) & a_sig) - (b_sig(N-1) & b_sig) + ('0' & cin);
25     -----output option #1:-----
26     sum <= std_logic_vector(sum_sig);
27     sub <= std_logic_vector(sub_sig);
28     -----output option #2:-----
29     --sum <= std_logic_vector(sum_sig(N-1 DOWNTO 0));
30     --cout_sum <= std_logic(sum_sig(N));
31     --sub <= std_logic_vector(sub_sig(N-1 DOWNTO 0));
32     --cout_sub <= std_logic(sub_sig(N));
33 END ARCHITECTURE;
34 -----

```

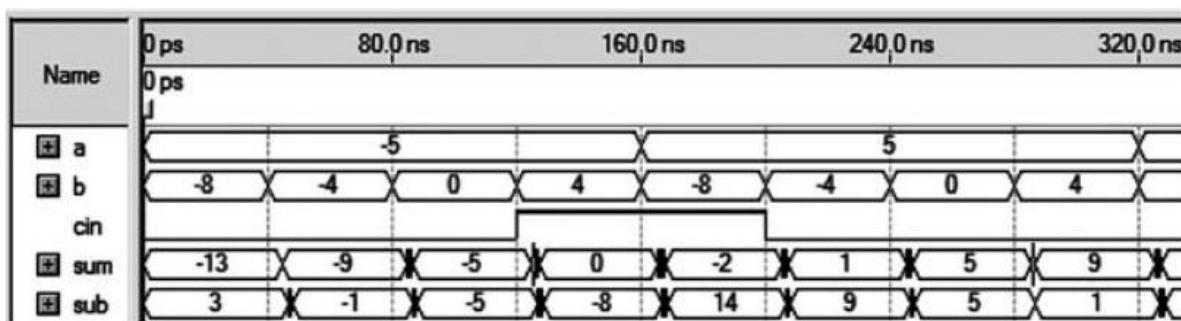


Figure 5.9  
Simulation results from the adder (or subtracter) of example 5.7.

## ۸-۵ جلوگیری از ساده شدن مدارات منطقی ترکیبی

در زیر دو مثال آورده شده که در جهت جلوگیری از ساده شدن مدارات ترکیبی، یکی به کمک مختصه‌ی `keep` و دیگری بدون استفاده از این مختصه، کار مزبور را انجام می‌دهد.

### مثال ۸-۵ تولید کننده‌ی پالس کوتاه به کمک مختصه‌ی `keep`

شکل ۱۰-۵-الف یک نمونه مدار متداول جهت تولید پالس‌های کوتاه مبتنی بر استفاده از فلیپ‌فلاب‌ها را نشان می‌دهد. توجه کنید که در این شکل، خط تاخیر شامل سه معکوس‌کننده است که اگر ما تمهدات خاصی در نظر نگیریم، کامپایلر آن را به تنها یک معکوس‌کننده ساده‌سازی خواهد کرد. این مدار را به کمک استفاده از مختصه‌ی `keep` طوری طراحی کنید که گره‌های `a`, `b` و `c` حفظ شوند. نتایج شبیه‌سازی را نشان دهید. قبل از شروع، در شکل ۱۰-۵-ب، شکل موج‌های مطلوب و مورد نظر را در گره‌های `c` و `short_clk` رسم نمایید. فرض کنید که تاخیر انتشار در هر معکوس‌کننده و نیز گیت AND حدود ۱ ns است. همچنین، این بازه‌ی زمانی را برابر با فاصله‌ی دو خط‌چین عمودی متواالی در شکل مزبور در نظر بگیرید.

حل: شکل ۱۰-۵-ب را خودتان رسم نمایید. کد VHDL مربوط به مدار اشاره شده در زیر نشان داده شده است. در خطوط ۱۰-۹ از مختصه‌ی `keep` استفاده شده است تا کامپایلر گره‌های `a`, `b` و `c` را حفظ نماید. نمایش RTL ایجاد شده توسط این کد به همراه نتایج شبیه‌سازی در شکل ۱۱-۵ نشان داده شده است. بررسی کنید که آیا ترسیم خودتان در شکل ۱۰-۵-ب با نتایج شبیه‌سازی در شکل ۱۱-۵ تطابق دارد؟

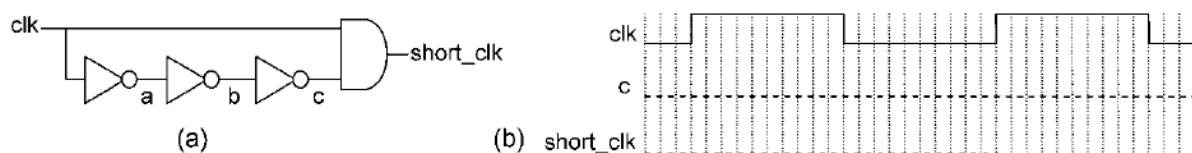


Figure 5.10  
Short-pulse generator of example 5.8.

```

1 -----
2 ENTITY short_pulse_gen IS
3     PORT (clk: IN BIT;
4             short_clk: OUT BIT);
5 END ENTITY;
6 -----
7 ARCHITECTURE short_pulse OF short_pulse_gen IS
8     SIGNAL a, b, c: BIT;
9     ATTRIBUTE keep: BOOLEAN;
10    ATTRIBUTE keep OF a, b, c: SIGNAL IS TRUE;
11 BEGIN
12     a <= NOT clk;
13     b <= NOT a;
14     c <= NOT b;
15     short_clk <= clk AND c;
16 END ARCHITECTURE;
17 -----

```

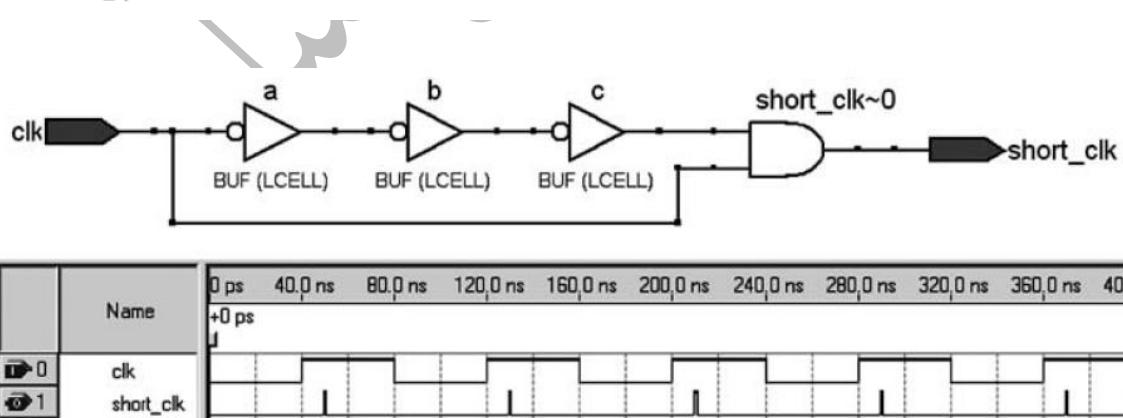


Figure 5.11  
RTL view and simulation results from the short-pulse generator of example 5.8.

یک راه حل دیگر برای طراحی فوق استفاده از «عنصر پایه‌ای»<sup>۱</sup> LCELL در نرم افزار Quartus است که در حقیقت یک جزء<sup>۲</sup> شامل یک بافر است که می‌توان آن را در مسیر سیگنال قرار داد. برای استفاده از این جزء باید آن را در کد خود به کمک دستور COMPONENT (فصل ۸) نمونه‌سازی کنیم. شکل استفاده از آن به صورت زیر است:

```
COMPONENT LCELL
  PORT (a_in: IN STD_LOGIC;
        a_out: OUT STD_LOGIC);
END COMPONENT;
```

**مثال ۹-۵: تولیدکننده پالس کوتاه به کمک LCELL**  
همان مثال ۸-۵ را این بار به کمک عنصر پایه‌ای LCELL (به جای مختصه‌ی keep) طراحی کنید.

حل: یک کد VHDL برای این مساله در زیر نشان داده شده است. در این کد، در بخش اعلان معماری از اعلان COMPONENT (خطوط ۱۵-۱۲) برای نمونه‌سازی LCELL استفاده شده است (این بزرگ-تابع<sup>۳</sup> در فایل altera\_mf\_components.vhd قرار دارد). از شش سیگنال کمکی به جای سه سیگنال، استفاده شده است زیرا هر سیگنال باید شکسته شود تا یک عنصر LCELL در آن قرار داده شود. نمونه‌سازی‌های LCELL در خطوط ۱۸، ۲۰ و ۲۲ انجام شده است. مدار حاصل از این کد مشابه همان مدار مربوط به مثال قبلی است.

<sup>1</sup> LCELL

<sup>2</sup> Component

<sup>3</sup> Megafunction

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY short_pulse_gen IS
6     PORT (clk: IN STD_LOGIC;
7             short_clk: OUT STD_LOGIC);
8 END ENTITY;
9 -----
10 ARCHITECTURE short_pulse OF short_pulse_gen IS
11     SIGNAL a1, a2, b1, b2, c1, c2: STD_LOGIC;
12     COMPONENT LCELL IS
13         PORT (a_in: IN STD_LOGIC;
14                 a_out: OUT STD_LOGIC);
15     END COMPONENT;
16 BEGIN
17     a1 <= NOT clk;
18     buffer_a: COMPONENT LCELL PORT MAP (a1, a2);
19     b1 <= NOT a2;
20     buffer_b: COMPONENT LCELL PORT MAP (b1, b2);
21     c1 <= NOT b2;
22     buffer_c: COMPONENT LCELL PORT MAP (c1, c2);
23     short_clk <= clk AND c2;
24 END ARCHITECTURE;
25 -----

```



## ۱۰-۵ ملاحظات مربوط به VHDL 2008

نکاتی که در VHDL 2008 مرتبط با مطالب مطرح شده در این فصل باید به آنها توجه نمود عبارتند از:

With respect to the material covered in this chapter, the main additions specified in VHDL 2008 are those listed below.

- 1) The concurrent WHEN and SELECT statements can be used also in sequential code. For example, WHEN can replace IF or can be used inside IF, while SELECT can replace CASE.
- 2) WHEN allows boolean tests (the only consequence of this is sometimes a slightly shorter code, at the cost of reduced code clarity). For example, the two codes below are equivalent (the traditional format is on the left).

```
x <= '0' WHEN rst='0' ELSE
    '1' WHEN a='0' OR b='1' ELSE
    '-';
x <= '0' WHEN NOT rst ELSE
    '1' WHEN NOT a OR b ELSE
    '-';
```

- 3) The matching "SELECT?" statement was introduced, which allows the use of don't care inputs. An example is shown below.

```
WITH interrupt SELECT?
priority <= 4 WHEN "1---",
            3 WHEN "01--",
            2 WHEN "001-",
            1 WHEN "0001",
            0 WHEN OTHERS;
```

- 4) In the conditional IF-GENERATE statement, the use of ELSIF/ELSE is allowed. A simplified syntax is shown below (on the left).

```
label: IF condition GENERATE
  [declarative_part
  BEGIN]
  concurrent_statements_part
[ELSIF condition GENERATE
  [declarative_part
  BEGIN]
  concurrent_statements_part]
[ELSE GENERATE
  [declarative_part
  BEGIN]
  concurrent_statements_part]
END GENERATE [label];
```

```
label: CASE expression GENERATE
WHEN condition_1 =>
  [declarative_part
  BEGIN]
  concurrent_statements_part
WHEN condition1_2 =>
  [declarative_part
  BEGIN]
  concurrent_statements_part
...
END GENERATE [label];
```

- 5) A new form of conditional GENERATE, called CASE-GENERATE, was also introduced. A simplified syntax is shown above (on the right).
- 6) The use of alternative labels for IF-GENERATE and the use of END before END GENERATE are both allowed.

## فصل ششم

### کد ترتیبی

همان طور که در فصل پنجم نیز اشاره شد، کدهای همزمان تنها برای توصیف مدارات ترکیبی و کدهای ترتیبی برای توصیف هر دو نوع مدار (ترکیبی و ترتیبی) قابل استفاده هستند. منظور از «دستورات همزمان» دستوراتی است که فقط برای توصیف مدارات ترکیبی قابل استفاده هستند. این دستورات شامل WHEN، SELECT و GENERATE می باشند. به طور معادل، «دستورات ترتیبی» نیز وجود دارند که شامل دستورات IF، LOOP، WAIT و CASE می باشند. در VHDL سه نوع کد ترتیبی وجود دارد که عبارتند از فرآیند (PROCESS)، تابع (FUNCTION) و روال (PROCEDURE) (دو تای آخر را زیربرنامه نیز می گویند). فرآیند به طور خاص برای استفاده در بدنی معماری ها در نظر گرفته شده است اما دو تای دیگر به طور خاص برای استفاده در کتابخانه ها در نظر گرفته شده اند. فرآیند در فصل فعلی و توابع و روال ها در فصل ۹ بررسی خواهند شد.

توجه: چه فرآیند و چه زیربرنامه ها در حالت کلی و از بیرون خود، به عنوان یک دستور همزمان تلقی می شوند (لذا به طور همزمان با بقیه دستوراتی که بیرون از اینها استفاده شده، اجرا می شوند).

یک نکته مهم دیگر در ارتباط با کدهای ترتیبی، توجه به تفاوت بین سیگنال ها (SIGNAL) و متغیرها (VARIABLE) است. این تفاوت آن قدر مهم است که فصل ۷ به طور کامل و ویژه به این مطلب اختصاص داده شده است. اما به دلیل نیاز این فصل به این نکته، در اینجا مهمترین نکات و ویژگی های این دو آورده شده است.

**مهمترین ویژگی های یک سیگنال:**

- یک سیگنال را تنها در «بیرون» از یک کد ترتیبی می توان اعلام کرد. اما داخل کدهای ترتیبی می توان از سیگنال ها «استفاده» نمود.
- به روز رسانی یک سیگنال فوری و لحظه ای «نمی باشد». هر گاه داخل یک کد ترتیبی مقدار یک سیگنال بخواهد تغییر کند، مقدار جدید در «پایان» آن دور اجرایی از کد ترتیبی اعمال خواهد شد.
- هرگاه در لحظه تغییر یک سیگنال، بخواهیم عمل انتساب و تخصیص مقدار به یک سیگنال دیگر داشته باشیم، این کار منجر به تولید و استفاده از ثباتها در مدار مربوطه خواهد شد.

- در تمام بدن‌های یک کد ترتیبی، برای هر سیگنال تنها یک بار می‌توان عمل تخصیص را انجام داد. حتی اگر کامپایلر به تخصیص چندگانه انجام شده داخل این کد ایرادی دارد نکند، تنها آخرین تخصیص ملاک عمل و معتبر و موثر خواهد بود؛ بنابراین، باز هم می‌توان گفت تنها یک عمل تخصیص برای هر سیگنال مجاز و موثر خواهد بود.

### مهمترین ویژگی‌های یک متغیر:

- یک متغیر را تنها در «داخل» یک کد ترتیبی (یک فرآیند یا یک زیربرنامه) می‌توان «اعلان» و «استفاده» کرد. حتی اگر از یک متغیر به اشتراک گذاشته شده<sup>۱</sup> استفاده کنیم (یعنی متغیری که اعلان آن در جایی دیگر انجام گرفته است)، تنها در داخل یک کد ترتیبی می‌توان از این متغیر استفاده نمود.
- به روز رسانی مقدار یک متغیر فوری است. لذا مقدار جدید متغیر در خط بعدی از کد ترتیبی قابل استفاده است.
- هرگاه در لحظه‌ی تغییر یک سیگنال، بخواهیم عمل انتساب و تخصیص مقدار به یک متغیر داشته باشیم، این کار منجر به تولید و استفاده از ثباتها در مدار مربوطه خواهد شد (زیرا در حالت کلی فرض می‌شود که مقدار یک متغیر قرار است روی یک سیگنال تاثیر داشته باشد).
- تخصیص‌های چندگانه به متغیرها جایز است.

## ۶-۲ فلیپ‌فلاب‌ها و لچ‌ها

از آنجا که فلیپ‌فلاب‌ها عناصری اساسی و جدایی‌ناپذیر در مدارات ترتیبی هستند و نیز لچ‌ها هم گاه‌آ در این نوع مدارات استفاده می‌شوند، در اینجا این عناصر به طور مختصر بررسی می‌شوند. دو نوع اساسی لچ داریم: لچ SR (SRL) و لچ D (DL). چهار نوع فلیپ‌فلاب نیز عبارتند از: فلیپ‌فلاب SR (SRFF)، فلیپ‌فلاب D (DFF)، فلیپ‌فلاب T (TFF) و فلیپ‌فلاب JK (JKFF). عناصر DFF و DL متداول‌ترین و رایج‌ترین در دسته‌ی خود هستند. در بین تمامی عناصر نیز DFF بیشتر از بقیه استفاده و کاربرد دارد. برای مثال در هر افزارهای FPGA هزاران DFF وجود دارد. تفاوت اساسی بین فلیپ‌فلاب‌ها و لچ‌ها در حساسیت آنها به سطح<sup>۲</sup> یا لبه<sup>۳</sup> است. لچ‌ها حساس به سطح هستند؛ یعنی برای مثال، در تمام بازه‌ی زمانی که کلاک مقدار<sup>۱</sup> داشته باشد (حساس به سطح بالا) تغییرات و مقادیر ورودی در خروجی کپی می‌شود. در غیر این لحظات، خروجی آخرین مقدار خود را حفظ کرده و به تغییرات ورودی واکنش نشان نمی‌دهد.

<sup>1</sup> Shared Variable

<sup>2</sup> Level Sensitive

<sup>3</sup> Edge Sensitive

شکل های ۱-۶-الف و ۱-۶-ب دو نوع DL یکی حساس به سطح بالا و دیگری حساس به سطح پایین را نشان می دهند. هر دو نوع دارای ورودی ریست (rst) هستند به این معنا که به محض '۱' شدن این ورودی، مقدار خروجی صفر می شود (ورودی آسنکرون).

در شکل های ۱-۶-ج تا ۱-۶-و چهار نوع DFF نمایش داده شده است. دو شکل اول به ترتیب حساس به لبه بالارونده و حساس به لبه پایین رونده هستند. دو شکل اخر نیز همین ترتیب را دارند. تفاوت بین دو شکل اول و دو شکل آخر در نوع ریست آنها است. در دو شکل اول، ریست با علامت  $rst$  بوده و از نوع آسنکرون است. اما در دو شکل آخر ریست از نوع سنکرون بوده و نام آن  $clr$  است (ورودی «پاک کردن»). در این کتاب، همیشه از نام ریست برای اشاره به نوع آسنکرون استفاده شده است.

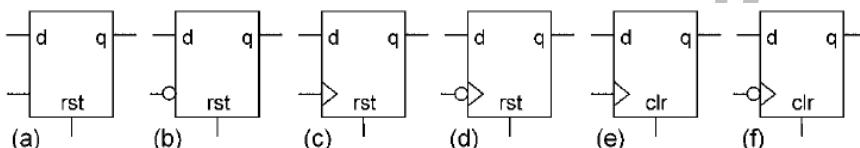
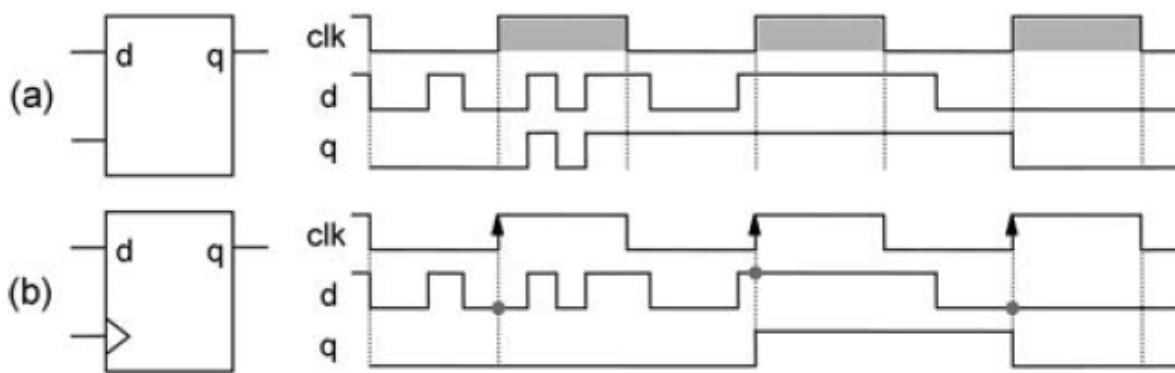


Figure 6.1

Latches and flip-flops. (a) Positive-level DL with reset; (b) Negative-level DL with reset; (c) Positive-edge DFF with reset; (d) Negative-edge DFF with reset; (e) Positive-edge DFF with clear; (f) Negative-edge DFF with clear.

شکل ۲-۶ عملکرد مدارات DFF و DL را نشان می دهد (این نوع نمایش، تنها تحلیل تابعی<sup>۱</sup> است زیرا از تاخیرهای انتشار درونی صرفنظر شده است). از یک کلاک و یک نوع داده برای هر دو مدار استفاده شده است. در هر دو مدار، خروجی  $q$  نامیده شده است. مقدار یا «حالت» اولیه در هر دو مدار به صورت یکسان '۱'  $q$  در نظر گرفته شده است. برای تسهیل تحلیل، آن «بازههای» زمانی که DL شفاف است با سایهای خاکستری روی سیگنال کلاک مشخص شده است. در مورد DFF آن «لحظاتی» که فلیپ فلاب شفاف است، با علامت پیکان روی سیگنال کلاک مشخص شده است. ملاحظه می کنید که علیرغم یکسان بودن شکل موج ورودی  $d$  در هر دو مدار، خروجی های بسیار متفاوتی تولید شده است.

<sup>۱</sup> Functional Analysis



**Figure 6.2**  
(a) DL and (b) DFF operation examples.

### ۳-۶ فرآیند

داخل فرآیند تنها دستورات «ترتیبی» شامل IF، WAIT، LOOP و CASE می‌توان استفاده کرد. البته از عملگرها در هر نوع کدی همواره می‌توان استفاده کرد. گرامر ساده شده‌ای از آن به صورت زیر است:

```
[label:] PROCESS [(sensitivity_list)] [IS]
    [declarative_part]
BEGIN
    sequential_statements_part
END PROCESS [label];
```

گذاشتن برچسب اختیاری بوده و به منظور بهبود خوانایی کدهای طولانی استفاده می‌شود. لیست حساسیت اجباری است اما در صورت استفاده از دستور WAIT غیرمجاز و ممنوع می‌باشد. دستورات داخل فرآیند هر زمان که یکی از سیگنالهای مشخص شده در لیست حساسیت تغییر کنند (یا شرط ذکر شده همراه با WAIT برقرار شود) یک بار اجرا می‌شوند. بخش اعلان فرآیند می‌تواند شامل اقلام زیر باشد:

- اعلان زیربرنامه
- بدنه‌ی زیربرنامه
- اعلان نوع
- اعلان زیرنوع
- اعلان ثابت

- اعلان متغیر
  - اعلان فایل
  - اعلان همپوشان (alias)
  - اعلان مختصه (attribute specification)
  - عبارت use
  - اعلان کلیشه گروه
  - اعلان گروه
- اعلان سیگنال در فرآیند مجاز نیست.

In VHDL 2008, the following is allowed in the declarative part of PROCESS besides the items already listed above: subprogram instantiation declaration, package declaration, package body, and package instantiation declaration. Additionally, the keyword ALL was introduced for the sensitivity list (to reduce errors when implementing combinational circuits with sequential code). See other details in section 6.10.

مثال:

**Example** The (partial) process below is executed whenever *clk* or *rst* changes. It contains three variable declarations (*a*, *b*, *c*), the first two specified as INTEGER, the last one as BIT\_VECTOR. Only for *c* a default value (optional) was entered.

```
PROCESS (clk, rst)
  VARIABLE a, b: INTEGER RANGE 0 TO 255;
  VARIABLE c: BIT_VECTOR(7 DOWNTO 0) := "00001111";
BEGIN
  ...
END PROCESS;
```

#### ۴-۶ دستور IF

گرامر ساده‌شده‌ای از این دستور در زیر نشان داده شده است. در قسمت شرایط، به طور اختیاری می‌توانید از پرانتزها استفاده نمایید.

```
[label:] IF conditions THEN  
    assignments;  
ELSIF conditions THEN  
    assignments;  
...  
ELSE  
    assignments;  
END IF [label];
```

مثال:

### Example

```
IF (x<y) THEN  
    temp:= "00001111";  
ELSIF (x=y AND w='0') THEN  
    temp:= "11110000";  
ELSE  
    temp:=(OTHERS => '0');  
END IF;
```

در VHDL 2008 در داخل IF می‌توان از دستورات همزمان WHEN و SELECT استفاده نمود.  
همچنین در دستور IF می‌توان از گزاره‌های بولی استفاده کرد.

مثال ۱-۶ :

**Example 6.1: DFFs with Reset and Clear**

Employing the IF statement, write a code that implements the DFFs of figures 6.1c and 6.1e.

**Solution** A code for this circuit is shown below. The inputs are  $d1$ ,  $clk$ , and  $rst$  for the first flip-flop, and  $d2$ ,  $clk$ , and  $clr$  for the second, while the outputs are  $q1$  for the first and  $q2$  for the second DFF. Even though both DFFs could be designed with just one process, two processes were employed to make the code easier to inspect (this does not affect the inferred circuit).

The first DFF is in the process of lines 13–20, under the (optional) label *with\_reset*. Note that  $clk$  and  $rst$  are in the sensitivity list (line 13), so if any of them changes the process is run. Note also that  $rst$  has precedence over  $clk$  in the IF statement (lines 15–19). To detect a clock edge, the 'EVENT attribute (line 17), seen in section 4.4, is used, which returns TRUE when an event occurs on  $clk$  ( $clk'EVENT$ ) and this event is an upward transition (AND  $clk = '1'$ ).

The second process is in lines 22–31, under the label *with\_clear*. Only  $clk$  is in the sensitivity list, so the process is run only when  $clk$  changes (in this particular example, the presence of  $clr$  in the sensitivity list would not affect the result). The synchronism of  $clr$  is established in the IF statements of lines 24–30, because  $clr$  is only tested (line 25) when a positive clock edge occurs (line 24).

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY flipflops IS
6     PORT (d1, d2, clk, rst, clr: IN STD_LOGIC;
7             q1, q2: OUT STD_LOGIC);
8 END ENTITY;
9 -----
10 ARCHITECTURE flipflops OF flipflops IS
11 BEGIN
12     ---DFF of Figure 6.1(c):---
13     with_reset: PROCESS (clk, rst)
14     BEGIN
15         IF (rst='1') THEN
16             q1 <= '0';

```

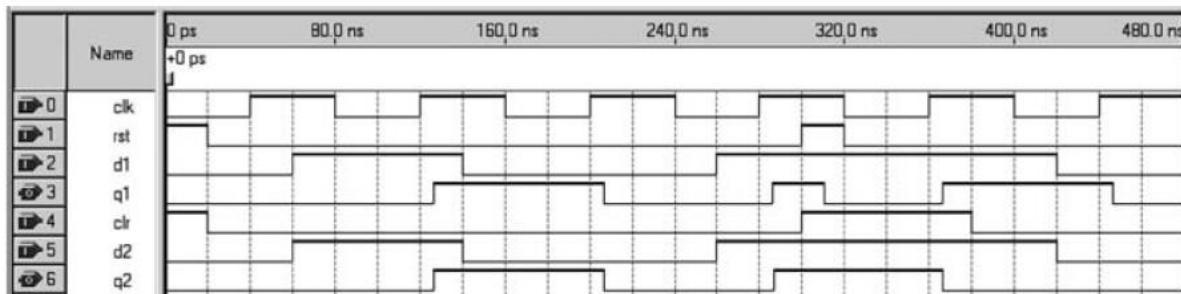


```

17      ELSIF (clk'EVENT AND clk='1') THEN
18          q1 <= d1;
19      END IF;
20  END PROCESS with_reset;
21  ---DFF of Figure 6.1(e):---
22  with_clear: PROCESS (clk)
23  BEGIN
24      IF (clk'EVENT AND clk='1') THEN
25          IF (clr='1') THEN
26              q2 <= '0';
27          ELSE
28              q2 <= d2;
29          END IF;
30      END IF;
31  END PROCESS with_clear;
32 END ARCHITECTURE;
33 -----

```

Simulation results are displayed in figure 6.3. The reader is invited to examine the plots to check the (correct) operation of both DFFs (observe particularly the effects of *rst* and *clr*).

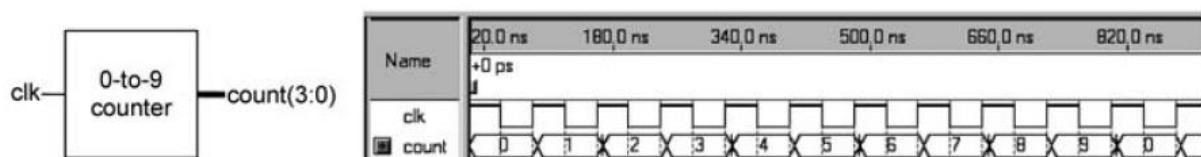


**Figure 6.3**  
Simulation results from the DFFs of example 6.1.

: ۲-۶ مثال

### Example 6.2: Basic Counter

Figure 6.4 shows, on the left, a diagram for a regular binary 0-to-9 counter. Write a VHDL code that implements this circuit.



**Figure 6.4**  
Counter (with simulation results) of example 6.2.

**Solution** A VHDL code for this counter is presented below, with input *clk* (line 3) and output *count* (line 4). A PROCESS (lines 9–19), with the IF statement playing the central role, is used to construct the circuit. In it, a VARIABLE, called *temp* (line 10), is employed, whose value is eventually passed to the actual output, *count* (line 18). Because a variable is updated *immediately*, the comparison in line 14 must be against 10 instead of 9 (state 10 actually never occurs because the variable never reaches line 18 with that value), so the actual range of the counter is from 0 to 9. Simulation results are included in figure 6.4.

```

1 -----
2 ENTITY counter IS
3     PORT (clk: IN BIT;
4             count: OUT INTEGER RANGE 0 TO 9 );
5 END ENTITY;
6 -----
7 ARCHITECTURE counter OF counter IS
8 BEGIN
9     PROCESS(clk)
10    VARIABLE temp: INTEGER RANGE 0 TO 10;
11    BEGIN
12        IF (clk'EVENT AND clk='1') THEN
13            temp := temp + 1;
14            IF (temp=10) THEN
15                temp := 0;
16            END IF;
17        END IF;
18        count <= temp;
19    END PROCESS;
20 END ARCHITECTURE;
21 -----

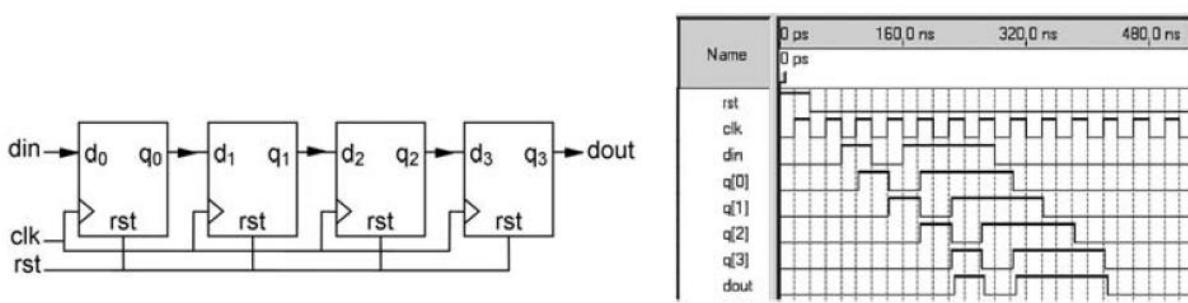
```



مثال :۳-۶

### Example 6.3: Shift Register

Figure 6.5 shows a shift register, which consists of a string of serially connected DFFs. The input is *din* and the output is either *q<sub>0</sub>* to *q<sub>3</sub>* or just *dout*, depending on the application. For example, the former can be used to convert data from serial to parallel form, while the latter can be used to implement a delay line. Design this circuit using VHDL. The number of stages should be *generic*.



**Figure 6.5**  
Shift register (with simulation results) of example 6.3.

**Solution** A VHDL code for this circuit is presented below. The number of stages ( $N$ ) is generic (line 6). The data input and output ports are *din* (line 7) and *dout* (line 8), respectively. A process is used to implement the circuit (lines 13–22), with *clk* and *rst* in the sensitivity list (line 13). Note that the shift register is obtained by simply shifting the whole vector *q* one position to the right at every positive clock transition, with the rightmost value discarded and the leftmost position taken by *din*. Simulation results (for  $N = 4$ ) are included in figure 6.5. As can be seen, the whole data vector does move one position to the right at every rising edge of the clock.

دانشگاه شهید بهشتی

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY shift_register IS
6     GENERIC (N: INTEGER := 4); --number of stages
7     PORT (din, clk, rst: IN STD_LOGIC;
8             dout: OUT STD_LOGIC);
9 END ENTITY;
10 -----
11 ARCHITECTURE shift_register OF shift_register IS
12 BEGIN
13     PROCESS (clk, rst)
14         VARIABLE q: STD_LOGIC_VECTOR(0 TO N-1);
15     BEGIN
16         IF (rst='1') THEN
17             q := (OTHERS => '0');
18         ELSIF (clk'EVENT AND clk='1') THEN
19             q := din & q(0 TO N-2);
20         END IF;
21         dout <= q(N-1);
22     END PROCESS;
23 END ARCHITECTURE;
24 -----

```

## 5-۶ دستور WAIT

این دستور ترتیبی دارای سه شکل است. از این سه شکل، دو شکل برای سنتز و سومی برای شبیه‌سازی استفاده می‌شود. اگر از این دستور در یک فرآیند استفاده شود، دیگر فرآیند نباید دارای لیست حساسیت باشد. گرامر ساده شده‌ی این سه شکل به صورت زیر است:

[label:] WAIT UNTIL condition;

[label:] WAIT ON sensitivity\_list;

[label:] WAIT FOR time\_expression;

### دستور WAIT UNTIL

این دستور موجب توقف موقتی یک فرآیند یا زیربرنامه تا زمان تحقق یک شرط می‌شود. در مثال زیر دو فرآیند معادل با هم جهت پیاده‌سازی یک DFF با پاک کردن سنکرون نشان داده شده است. در یک فرآیند از IF و در دیگری از WAIT UNTIL استفاده شده است. توجه کنید که در حالت استفاده از WAIT فرآیند مربوطه قادر لیست حساسیت است.

```
---DFF process with IF:-----
PROCESS (clk)
BEGIN
  IF (clk'EVENT AND clk='1') THEN
    IF (clr='1') THEN
      q <= '0';
    ELSE
      q <= d;
    END IF;
  END IF;
END PROCESS;
```

```
---DFF process with WAIT UNTIL:-----
PROCESS
BEGIN
  WAIT UNTIL (clk'EVENT AND clk='1');
  IF (clr='1') THEN
    q <= '0';
  ELSE
    q <= d;
  END IF;
END PROCESS;
```

### دستور WAIT ON

این دستور نیز موجب توقف فرآیند یا زیرنامه تا زمان تغییر حداقل یکی از سیگنال‌های موجود در لیست دستور WAIT می‌شود. در مثال زیر دستور WAIT ON سیگنال کلاک را رصد (مانیتور) می‌کند. استفاده از دستور WAIT ON در ابتدا یا انتهای یک فرآیند قادر لیست حساسیت معادل با استفاده از یک فرآیند دارای همان لیست حساسیت قبلی است؛ بنابراین، دو فرآیند زیر معادل با هم

بوده و هر دو یک DFF دارای پاک کردن سنکرون را پیاده‌سازی می‌کنند. مجلد ااضن توجه کنید که در حالت استفاده از WAIT فرآیند مربوطه قادر لیست حساسیت است.

```
---DFF process with IF:-----
PROCESS (clk)
BEGIN
  IF (clk'EVENT AND clk='1') THEN
    IF (clr='1') THEN
      q <= '0';
    ELSE
      q <= d;
    END IF;
  END IF;
END PROCESS;
```

```
---DFF process with WAIT ON:-----
PROCESS
BEGIN
  IF (clk'EVENT AND clk='1') THEN
    IF (clr='1') THEN
      q <= '0';
    ELSE
      q <= d;
    END IF;
  END IF;
  WAIT ON clk;
END PROCESS;
```

### دستور WAIT FOR

این دستور به منظور شبیه‌سازی استفاده شده و در فصل ۱۰ به طور دقیق بررسی خواهد شد. اعلان زیر موجب تولید شکل موج کلکی با دوره‌ی تناوب ۸۰ ns می‌شود.

```
WAIT FOR 40ns;
clk <= NOT clk;
```

### ۶-۶ دستور LOOP

دستور LOOP زمانی استفاده می‌شود که تکه‌ای از کد می‌باشد بارها تکرار شود. این دستور معادل با دستور همزمان GENERATE است. این دستور نیز فقط مخصوص استفاده در کدهای ترتیبی است.

دستور LOOP در پنج حالت قابل استفاده است:

- غیرشرطی
- به همراه FOR
- به همراه WHILE
- به همراه EXIT
- به همراه NEXT

گرامر ساده شده‌ی این پنج حالت در زیر آورده شده است:

### Unconditional LOOP:

```
[label:] LOOP  
    sequential_statements  
END LOOP [label];
```

### LOOP with FOR:

```
[label:] FOR identifier IN range LOOP  
    sequential_statements  
END LOOP [label];
```

### LOOP with WHILE:

```
[label:] WHILE condition LOOP  
    sequential_statements  
END LOOP [label];
```

### LOOP with EXIT:

```
[loop_label:] [FOR identifier IN range] LOOP  
    ...  
    [exit_label:] EXIT [loop_label] [WHEN condition];  
    ...  
END LOOP [loop_label];
```

### LOOP with NEXT:

```
[loop_label:] [FOR identifier IN range] LOOP
    ...
    [next_label:] NEXT [loop_label] [WHEN condition];
    ...
END LOOP [loop_label];
```

مثالها:

حالت غیرشرطی

Example of unconditional LOOP:

```
LOOP
    WAIT UNTIL clk='1';
    count := count + 1;
END LOOP;
```

حالت FOR-LOOP

Example of LOOP with FOR: In the code below, which employs the FOR-LOOP version of LOOP, the loop is repeated unconditionally until  $i$  reaches 5 (that is, 6 times).

```
FOR i IN 0 TO 5 LOOP
    x(i) <= a(i) AND b(5-i);
    y(0, i) <= c(i);
END LOOP;
```

An important remark regarding FOR-LOOP (similar to that made for GENERATE in chapter 5) is that both range bounds are normally required to be static. Thus a declaration of the type "FOR  $i$  IN 0 TO  $x$  LOOP", where  $x$  is an input (therefore a nonstatic parameter), is generally not synthesizable.

حالت WHILE-LOOP

Example of LOOP with WHILE: The loop below will keep repeating while  $i < 10$ .

```
WHILE (i<10) LOOP
    WAIT UNTIL clk'EVENT AND clk='1';
    ...
END LOOP;
```

**حالت LOOP-EXIT**

Example of LOOP with EXIT: The loop will be terminated if a value different from '0' is found in *data*.

```
FOR i IN data'RANGE LOOP
  CASE data(i) IS
    WHEN '0' => count:=count+1;
    WHEN OTHERS => EXIT;
  END CASE;
END LOOP;
```

**حالت LOOP-NEXT**

Example of LOOP with NEXT: NEXT will cause LOOP to skip one iteration if *i = skip* occurs.

```
FOR i IN 0 TO 15 LOOP
  NEXT WHEN i=skip;
  ...
END LOOP;
```

در VHDL 2008 از عبارات بولی نیز به عنوان شرط می‌توان استفاده کرد. برای مثال، به جای .WHILE ena LOOP می‌توان نوشت: WHILE ena='1' LOOP

مثال ۴-۶ استفاده از حالت FOR-LOOP را که متداول‌ترین حالت استفاده از این دستور است، نشان می‌دهد. این مثال، همچنین نحوه‌ی ایجاد یک مدار کاملاً ترکیبی را به کمک کدهای ترتیبی نشان می‌دهد.

**مثال ۴-۶: جمع‌کننده ریپل رقم‌نقلی**

**Example 6.4: Carry-Ripple Adder**

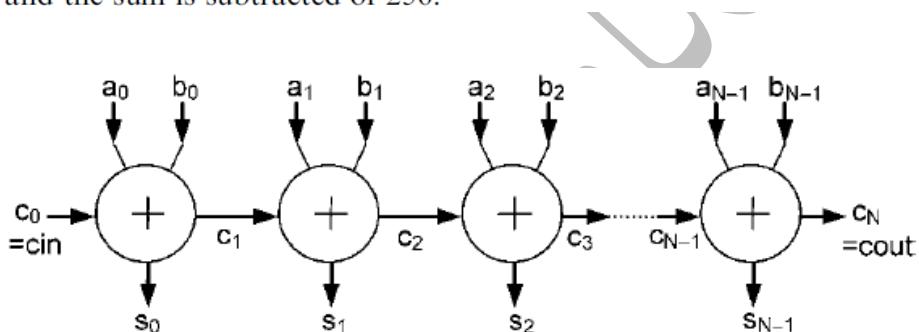
Adders are combinational circuits. The simplest multibit architecture, shown in figure 6.6, is called *carry-ripple adder* (Pedroni 2008). Each individual cell, called *full-adder* (FA), was seen in section 1.5 (figure 1.2). Write a VHDL code from which this adder can be inferred. Employ sequential code and enter the number of bits (stages) as a *generic* parameter, so the code can be easily adjusted to any adder size. Assume that the adder is unsigned.

**Solution** Each FA cell computes the sum and carry-out bits according with:

$$\text{Sum: } s_k = a_k \oplus b_k \oplus c_k$$

$$\text{Carry: } c_{k+1} = a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k$$

These two expressions appear in lines 20–22 of the code below. The FOR-LOOP statement (lines 19–23) is employed to instantiate the expressions  $N$  times. The number of bits is entered using a GENERIC declaration (line 6). Simulation results, for  $N = 8$ , are displayed in figure 6.7. As expected, whenever the sum is higher than 255, the carry-out bit is asserted and the sum is subtracted of 256.

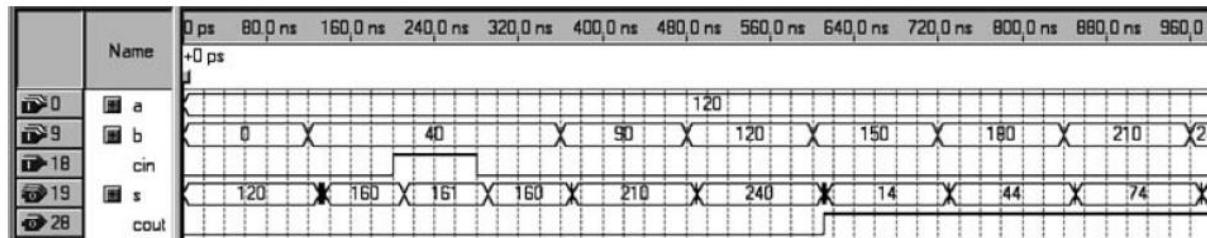


**Figure 6.6**  
Carry-ripple adder.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY carry_ripple_adder IS
6     GENERIC (N : INTEGER := 8); --number of bits
7     PORT (a, b: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8             cin: IN STD_LOGIC;
9             s: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0);
10            cout: OUT STD_LOGIC);
11 END ENTITY;
12 -----
13 ARCHITECTURE structure OF carry_ripple_adder IS
14 BEGIN
15     PROCESS(a, b, cin)
16         VARIABLE c: STD_LOGIC_VECTOR(N DOWNTO 0);
17     BEGIN
18         c(0) := cin;
19         FOR i IN 0 TO N-1 LOOP
20             s(i) <= a(i) XOR b(i) XOR c(i);
21             c(i+1) := (a(i) AND b(i)) OR (a(i) AND c(i)) OR
22                         (b(i) AND c(i));
23         END LOOP;
24         cout <= c(N);
25     END PROCESS;
26 END ARCHITECTURE;
27 -----

```



مثال ۶-۵ کاربردی از دستور LOOP-EXIT را نشان می‌دهد.

### Example 6.5: Leading Zeros

Design a circuit that counts the number of leading zeros in a binary vector, starting from its left end (MSB).

**Solution** A VHDL code for this problem is shown below. LOOP is in lines 16–21, and can be repeated up to  $N = 8$  times. If a '0' is found in the data vector, then EXIT (line 19) will terminate the loop. Simulation results are displayed in figure 6.8, illustrating the correct operation of the inferred circuit.

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY leading_zeros IS
6      PORT (data: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7             zeros: OUT INTEGER RANGE 0 TO 8);
8  END ENTITY;
9  -----
10 ARCHITECTURE behavior OF leading_zeros IS
11 BEGIN
12     PROCESS (data)
13         VARIABLE count: INTEGER RANGE 0 TO 8;
14     BEGIN
15         count := 0;
16         FOR i IN data'RANGE LOOP
17             CASE data(i) IS
18                 WHEN '0' => count := count + 1;
19                 WHEN OTHERS => EXIT;
20             END CASE;
21         END LOOP;
22         zeros <= count;
23     END PROCESS;
24 END ARCHITECTURE;
25 -----

```

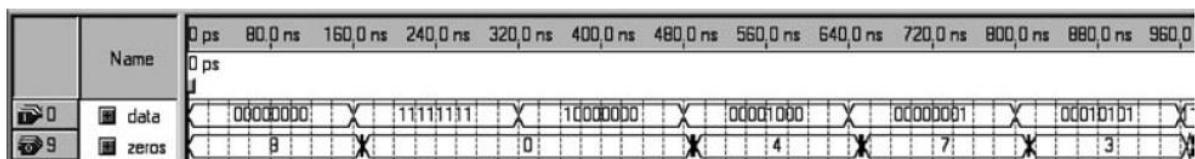


Figure 6.8  
Simulation results from the leading-zeros counter of example 6.5.

## 7-۶ دستور CASE

این دستور نیز جزو دستورات ترتیبی است. گرامر ساده شده‌ای از آن به همراه مثال در زیر آورده شده است:

```
[label:] CASE expression IS
    WHEN value => assignments;
    WHEN value => assignments;
    ...
END CASE;
```

### Example

```
CASE control IS
    WHEN "000" => x<=a; y<=b;
    WHEN "000" | "111" => x<=b; y<= '0';
    WHEN OTHERS => x<='0'; y<='1';
END CASE;
```

در این دستور از نمادهای | و TO می‌توان برای تعیین مقادیر مختلف استفاده کرد:

```
WHEN value1 | value2 | ...          --value1 or value2 or ...
WHEN value1 TO value2              --range (for enumerated types only)
```

در دستور CASE، مشابه با دستور SELECT، می‌بایست تمام مقادیر ممکن ورودی پوشش داده شوند (جدول صحت به طور کامل توصیف شود) لذا کلمه‌ی کلیدی OTHERS ممکن است مفید واقع شود. یک کلمه‌ی کلیدی مفید دیگر، NULL است که معادل با UNAFFECTED در دستور SELECT است. در استفاده از این کلمه‌ی کلیدی به تولید ثباتها و فلیپ‌فلاب‌ها توجه داشته باشید.

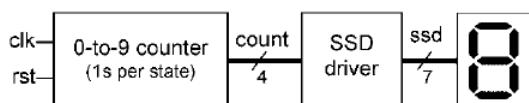
گرچه دستور CASE اساساً فقط در کدهای همزمان قابل استفاده است اما کاربرد اصلی آن در پیاده‌سازی مدارات ترکیبی بدون نیاز به خروج از فرآیند (PROCESS) یا زیربرنامه است. به

عبارت دیگر کاربرد اصلی این دستور بسیار مشابه با دستور SELECT است. مثال ۶-۶ استفاده از این دستور را نشان می‌دهد.

در VHDL 2008 امکان استفاده از دستور WHEN در داخل دستور CASE فراهم شده است. همچنین دستور انطباق شرطی ? CASE به منظور فراهم آوردن امکان استفاده از حالات بی‌همیت برای ورودی در نظر گرفته شده است. در این استاندارد، امکان استفاده از UNAFFECTED در کدهای ترتیبی نیز فراهم شده است.

### Example 6.6: Slow 0-to-9 Counter with SSD

Add an SSD (Seven-Segment Display—described in section 12.1, see figure 12.2) to the output of the 0-to-9 counter designed in example 6.2, such that the state of the counter can be visually inspected. This arrangement is depicted in figure 6.9. Besides the counter and the SSD, an SSD driver is also shown, which is a *combinational* circuit that converts the 4-bit output from the counter (called *count*) into a 7-bit signal (called *ssd*) to feed the seven segments of the display. Assume that the clock frequency is 50 MHz, which should be entered as a *generic* parameter, and that the counter must remain one second in each state. Write a VHDL code that implements such a circuit.



**Figure 6.9**  
One-second-per-state 0-to-9 counter with SSD of example 6.6.

**Solution** A VHDL code for this circuit is presented below. The inputs are *clk* and *rst* (line 4), while *ssd* is the output (line 5). The clock frequency, *fclk*, was entered using a GENERIC declaration (line 3).

A process (lines 10–42) is employed to build the circuit. In its first part (lines 15–27), the counter is built using the EVENT attribute (line 18) combined with the IF statement. Note that in fact two counters are constructed, for which two variables (lines 11–12) are used. The first counter sends an enable signal (lines 20–22) to the second counter after every *fclk* clock pulses, which lasts just one clock period, hence allowing the second counter to be incremented every one second. In the second part of the process (lines 29–41), the CASE statement is employed to build the SSD driver. Note that the signal *counter2* in the code corresponds to the signal *count* in figure 6.9.

Some simulation results are presented in figure 6.10, where the digit changes after every four clock pulses (to ease the inspection of the results). Note that this design (like many others that will come) is interesting to be tested in an actual device (FPGA board).

```
1 -----
2 ENTITY slow_counter IS
3     GENERIC (fclk: INTEGER := 50_000_000); --50MHz
4     PORT (clk, rst: IN BIT;
5             ssd: OUT BIT_VECTOR(6 DOWNTO 0));
6 END ENTITY;
7 -----
8 ARCHITECTURE counter OF slow_counter IS
9 BEGIN
10    PROCESS (clk, rst)
11        VARIABLE counter1: NATURAL RANGE 0 TO fclk := 0;
12        VARIABLE counter2: NATURAL RANGE 0 TO 10 := 0;
13    BEGIN
14        -----counter-----
15        IF (rst='1') THEN
16            counter1 := 0;
17            counter2 := 0;
18        ELSIF (clk'EVENT AND clk='1') THEN
19            counter1 := counter1 + 1;
20            IF (counter1=fclk) THEN
21                counter1 := 0;
22                counter2 := counter2 + 1;
```



```

23      IF (counter2=10) THEN
24          counter2 := 0;
25      END IF;
26  END IF;
27  -----SSD driver:-----
28 CASE counter2 IS
29     WHEN 0 => ssd<="0000001"; --"0" on SSD
30     WHEN 1 => ssd<="1001111"; --"1" on SSD
31     WHEN 2 => ssd<="0010010"; --"2" on SSD
32     WHEN 3 => ssd<="0000110"; --"3" on SSD
33     WHEN 4 => ssd<="1001100"; --"4" on SSD
34     WHEN 5 => ssd<="0100100"; --"5" on SSD
35     WHEN 6 => ssd<="0100000"; --"6" on SSD
36     WHEN 7 => ssd<="0001111"; --"7" on SSD
37     WHEN 8 => ssd<="0000000"; --"8" on SSD
38     WHEN 9 => ssd<="0000100"; --"9" on SSD
39     WHEN OTHERS => ssd<="0110000"; --"E"rror
40 END CASE;
41 END PROCESS;
42 END ARCHITECTURE;
43 -----
44 -----

```

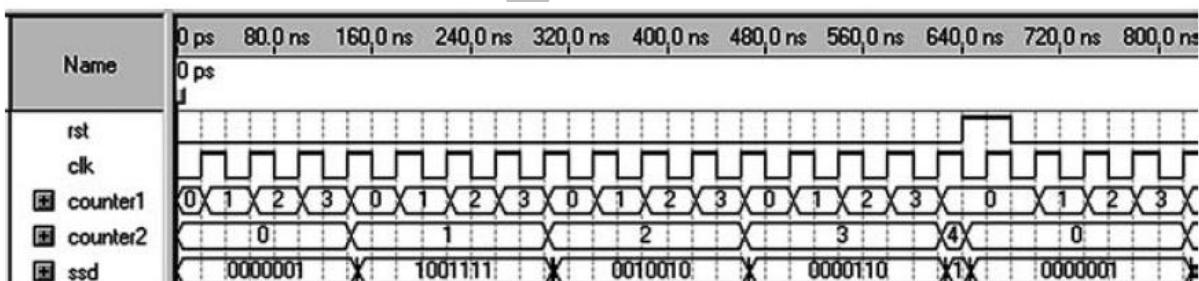


Figure 6.10  
Simulation results from the counter of example 6.6.

## ۸-۶ مقایسه SELECT و CASE

دستورات SELECT و CASE بسیار شبیه به هم هستند اما اولی برای کدهای همزمان و دومی برای کدهای ترتیبی قابل استفاده‌اند. مهمترین شباهتها و تفاوتها در جدول زیر آورده شده است.

	SELECT	CASE
Statement type	Concurrent (*)	Sequential
Location	Outside sequential code (*)	Inside sequential code
Input values that must be tested	All	All
Number of tests per entry	One test, multiple values	One test, multiple values
Number of assignments per test	One	Any
No-action keyword	UNAFFECTED (*)	NULL (*)
Matching statement	SELECT? (introd. in VHDL 2008)	CASE? (introd. in VHDL 2008)
(*) See Section 6.10 for extended options in VHDL 2008.		

Figure 6.11  
Comparison between SELECT and CASE.

مثال: کدهای زیر معادل با هم هستند:

----With SELECT:-----

```
WITH sel & ena SELECT
  x <= a WHEN "00" | "11",
  b WHEN "01",
  c WHEN OTHERS;
WITH sel & ena SELECT
  y <= "0000" WHEN "11",
  "1--1" WHEN OTHERS;
```

----With CASE:-----

```
CASE sel & ena IS
  WHEN "00" => x<=a; y<="1--1";
  WHEN "01" => x<=b; y<="1--1";
  WHEN "11" => x<=a; y<="0000";
  WHEN OTHERS => x<=c; y<="1--1";
END CASE;
```

## ۹-۶ پیاده‌سازی مدارات ترکیبی با کدهای ترتیبی

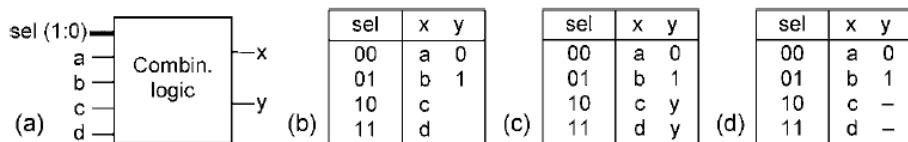
قبل‌اً اشاره شد که کدهای ترتیبی برای پیاده‌سازی هر دو نوع مدارات ترکیبی و ترتیبی قابل استفاده‌اند. در حین توصیف مدارات ترتیبی استفاده از ثباتها ضروری است اما در مدارات ترکیبی باید از تولید ثباتها اجتناب کرد. لذا باید به نکات زیر توجه شود:

- ۱- جدول صحیح مدار را به طور کامل توصیف کرده و تمام مقادیر ممکن را پوشش دهید.
- ۲- تمام سیگنال‌های مورد استفاده در یک فرآیند (PROCESS) را حتماً در لیست فرآیند نیز وارد کنید.

عدم توجه به نکته‌ی (۱) موجب تولید احتمالی ثباتها در مداراتان و عدم توجه به نکته‌ی (۲) موجب هشدار کامپایلر مبنی بر عدم استفاده از سیگنالی که در حال استفاده در داخل فرآیند است، خواهد شد.

**Example 6.7: Incomplete Combinational Design**

Consider a circuit whose top-level diagram is that in figure 6.12a, for which the specifications in figure 6.12b were given, saying that  $x$  should behave as a multiplexer—that is, should be equal to the input selected by  $sel$ , while  $y$  should be equal to '0' when  $sel = "00"$  or '1' if  $sel = "01"$ . Design such a circuit using VHDL.

**Figure 6.12**

(a) Top-level diagram for the circuit of example 6.7; (b) Specifications provided; (c) Implemented truth table; (d) Correct approach.

**Solution** This is a combinational circuit for which only partial specifications were provided for  $y$  (figure 6.12b). Using just those specifications, the code could be as follows.

After compiling this code, the compiler will report that (as expected) no registers (flip-flops) were inferred. However, when we look at the simulation results (figure 6.13), we notice something peculiar about  $y$ . Observe that, for the same value of the input ( $sel = 3$ ), two different results are obtained for  $y$  (when  $sel = 3$  is preceded by  $sel = 0$ ,  $y = '0'$  results, while  $y = '1'$  occurs when  $sel = 3$  is preceded by  $sel = 1$ ). This means that some sort of memory was implemented. If we now inspect the actual *equations* implemented by the fitter, a latch will be found. In summary, extra (unnecessary) logic was inferred, resulting the truth table of figure 6.12c. In this kind of situation, the 'don't care' value ("-") should be used to complete the specifications for  $y$  (see figure 6.12d).



```

1 -----Poor design-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY poor_design IS
6     PORT (a, b, c, d: IN STD_LOGIC;
7             sel: IN INTEGER RANGE 0 TO 3;
8             x, y: OUT STD_LOGIC);
9 END ENTITY;
10 -----
11 ARCHITECTURE example OF poor_design IS
12 BEGIN
13     PROCESS (a, b, c, d, sel)
14     BEGIN
15         IF (sel=0) THEN x<=a; y<='0';
16         ELSIF (sel=1) THEN x<=b; y<='1';
17         ELSIF (sel=2) THEN x<=c;
18         ELSE x<=d;
19         END IF;
20     END PROCESS;
21 END ARCHITECTURE;
22 -----

```

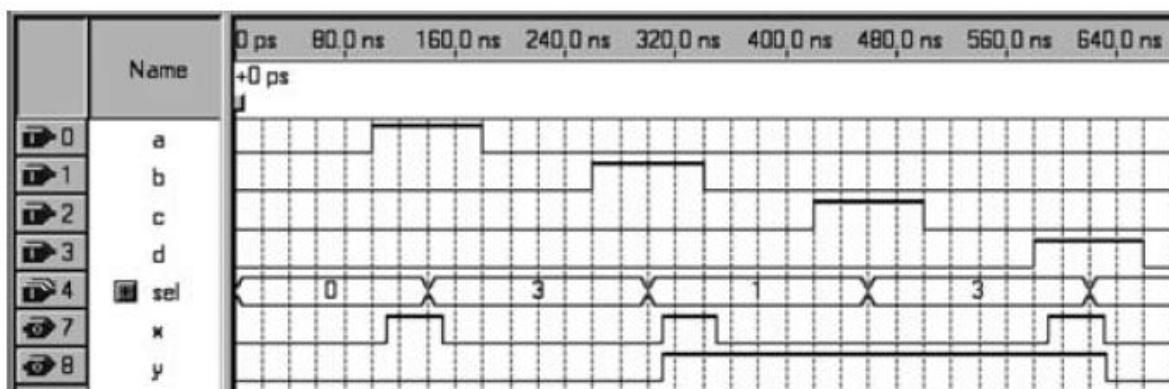


Figure 6.13  
Simulation results from the circuit of example 6.7.

## 6.10 VHDL 2008

With respect to the material covered in this chapter, the main additions specified in VHDL 2008 are those listed below.

- 1) In the declarative part of PROCESS, the following additional declarations are allowed: subprogram instantiation declaration, package declaration, package body, and package instantiation declaration.
- 2) The keyword ALL is allowed in the PROCESS's sensitivity list (to reduce errors when implementing combinational circuits with sequential code). For example:

```
PROCESS (ALL)
BEGIN
...
END PROCESS;
```

- 3) The concurrent statements WHEN and SELECT are allowed in sequential code (the only consequence of this is a slightly shorter code). An example with IF and WHEN is shown below (the traditional format is on the left).

<pre>IF (clk'EVENT AND clk='1') THEN   IF (clr='1') THEN q &lt;= '0';   ELSE q &lt;= d;   END IF; END IF;</pre>	<pre>IF (clk'EVENT AND clk='1') THEN   q &lt;= '0' WHEN clr='1' ELSE d; END IF;</pre>
---	---

- 4) IF allows boolean tests (the only consequence of this is sometimes a slightly shorter code, at the cost of reduced code clarity). For example, the two lines below are equivalent (the traditional format is in the first).

```
IF (a='0' AND b='0') OR c='1' THEN ...
IF (NOT a AND NOT b) OR c THEN ...
```

- 5) The matching CASE? statement was introduced, which has the same purpose of SELECT?—that is, to allow the use of don't care inputs. Two equivalent codes are presented.

<pre>WITH interrupt SELECT?   priority &lt;= 4 WHEN "1---",   3 WHEN "01--",   2 WHEN "001-",   1 WHEN "0001",   0 WHEN OTHERS;</pre>	<pre>CASE? interrupt IS   WHEN "1---" =&gt; priority &lt;= 4;   WHEN "01--" =&gt; priority &lt;= 3;   WHEN "001-" =&gt; priority &lt;= 2;   WHEN "0001" =&gt; priority &lt;= 1;   WHEN OTHERS =&gt; priority &lt;= 0;</pre>
---	---

- 6) The UNAFFECTED keyword was extended to sequential code (IF and CASE statements).
- 7) Some tests associated with LOOP statements can be boolean. For example, "WHILE ena LOOP" can be used instead of "WHILE ena = '1' LOOP".

## فصل هفتم

### سیگنال و متغیر

#### ۲-۷ سیگنال

سیگنال‌ها به حمل و نقل مقادیر به بیرون یا درون یک مدار و همچنین نقل و انتقالات درونی آن مدار کمک می‌کنند. به بیان دیگر، سیگنال‌ها معرف اتصالات داخلی مدار هستند. برای مثال، تمام پورتهای یک مدار از نوع سیگنال هستند.

اعلان یک سیگنال را می‌توان در بخش‌های اعلان موجودیت (ENTITY)، معماری GENERATE (ARCHITECTURE)، بسته PACKAGE، بلوک BLOCK، و دستور انجام داد. گرامر ساده‌ای از اعلان یک سیگنال به همراه تعدادی مثال در زیر آورده شده است:

```
SIGNAL signal_name: signal_type [range] [:= default_value];
```

#### Example

```
-----  
SIGNAL flag: STD_LOGIC := '0';  
SIGNAL address: NATURAL RANGE 0 TO 2**N-1;  
SIGNAL data: BIT_VECTOR(15 DOWNTO 0);  
-----
```

#### سیگنال‌های تعیین تکلیف شده (Resolved Signals)

یک ویژگی مختص به سیگنال‌ها این است که می‌توان برای آنها یک تابع «تعیین تکلیف»<sup>۱</sup> متناظر کرد. هدف از تعریف این تابع برای یک سیگنال، رفع ابهام («تعیین تکلیف») در مواردی است که بیش از یک محرک (یا «درایور»)، سیگنال مزبور را تغذیه (یا «درایو») می‌کنند (مشابه با آنچه که در مورد نوع STD\_LOGIC در بخش ۶-۳ دیده شد و اشاره شد که این نوع، حالت «تعیین تکلیف» شده‌ای از نوع STD\_ULOGIC است). گرامر ساده‌شده‌ای از اعلان سیگنال که شامل تابع تعیین تکلیف نیز باشد، به صورت زیر است:

<sup>1</sup> Resolution Function

```
SIGNAL signal_name: resolution_function signal_type [:= def_value];
```

### Example

```
SIGNAL x: my_resolution_function my_data_type;
```

### سیگنال‌های محافظت شده (Guarded Signals)

هرگاه نیاز به پشتیبانی از مقدار امپدانس بالا ('Z') در مدار خود داشته باشیم، از نوع STD\_LOGIC\_VECTOR می‌توانیم استفاده کنیم. در VHDL برای انواع دیگر نیز یک راه غیرمستقیم جهت پشتیبانی از این مقدار فراهم شده است. در این راه، اتصال محرک‌های انتخاب شده (به سیگنال مورد نظر)، به کمک کلمه‌ی کلیدی NULL، قطع می‌شود. سیگنالی که دارای چنین قابلیتی باشد را یک سیگنال محافظت شده می‌نامیم. در اعلان چنین سیگنالی از کلمات کلیدی REGISTER و یا BUS استفاده می‌کنیم:

```
SIGNAL signal_name: ... signal_type [REGISTER | BUS] [:= def_value];
```

فقط از سیگنال‌های تعیین تکلیف شده می‌توان محافظت نمود. تفاوت بین حالت‌های BUS و REGISTER زمانی مشخص می‌شود که «تمام» محرک‌های متصل به سیگنال قطع شده باشند. سیگنالی که از نوع REGISTER باشد، آخرین مقداری که قبل از قطع اتصال داشته است را حفظ می‌کند. اما سیگنالی که از طریق BUS اعلان شده باشد، مقدار خود در این حالت را از طریق تابع تعیین تکلیف که برای آن تعریف کرده‌ایم، می‌گیرد. در طراحی‌های بزرگ در حالت کلی از نوع STD\_LOGIC\_VECTOR استفاده شده و معمولاً نیازی به استفاده از سیگنال‌های محافظت شده نمی‌باشد.

## ۳-۷ متغیر

یک متغیر یک شیء بسیار ارزشمند در مدارات ترتیبی است. متغیرها حامل اطلاعات محلی هستند زیرا تنها در داخل کد ترتیبی که متغیر را تعریف کرده‌ایم، قابل استفاده هستند (در این حالت، تنها استثناء موجود، متغیرهای به اشتراک گذاشته شده هستند). متغیرها را می‌توان در بخش اعلان فرآیند، روال، تابع، بسته، و بدنی بسته (PACKAGE BODY) اعلان نمود. گرامر ساده‌ی اعلان یک متغیر به صورت زیر است:

```
VARIABLE variable_name: variable_type [range] [:= default_value];
```

### Example

```
-----
VARIABLE flag: STD_LOGIC := '0';
VARIABLE address: NATURAL RANGE 0 TO 2**N-1;
VARIABLE data: BIT_VECTOR(15 DOWNTO 0);
-----
```

### متغیرهای به اشتراک گذاشته شده (Shared Variables)

هرگاه متغیری از نوع به اشتراک گذاشته شده اعلان شود، آن‌گاه بیش از یک کد ترتیبی می‌توانند از آن متغیر استفاده کنند. در این حالت حتی کدهای همزمان نیز می‌توانند از آن متغیر استفاده نمایند. البته فقط کدهای ترتیبی قادر به تصحیح و تغییر مقدار متغیر هستند. نکته‌ی دیگر این است که مقدار یک متغیر به اشتراک گذاشته شده را می‌توان از طریق عمل تخصیص (انتساب) در «بیرون» از کد ترتیبی به یک سیگنال منتقل کرد. در بخش‌های زیر می‌توان چنین متغیری را اعلان نمود:

- ENTITY
- ARCHITECTURE
- BLOCK
- GENERATE
- PACKAGE

در مورد آخری توجه کنید که بسته نباید داخل یک فرآیند یا زیربرنامه قرار داشته باشد.

مثال ۷-۱ نحوه استفاده از چنین متغیرهایی را نشان می‌دهد.

### Example 7.1: Counter with SHARED VARIABLE

Design a 00-to-99 counter employing shared variables for both digits.

**Solution** The code below implements a circuit that counts from 00 to 99 and then automatically restarts from 00. Even though a single process would do, two were employed in order to illustrate the use of shared variables (*temp1* and *temp2*, declared in line 8). Note that each variable is modified by only one process (*proc1* for *temp1*, *proc2* for *temp2*) and that the passing of their values to signals can be done outside the processes (lines 35–36).

```

1 -----
2 ENTITY counter_with_sharedvar IS
3     PORT (clk: IN BIT;
4             digit1, digit2: OUT INTEGER RANGE 0 TO 9);
5 END ENTITY;
6 -----
7 ARCHITECTURE counter OF counter_with_sharedvar IS
8     SHARED VARIABLE temp1, temp2: INTEGER RANGE 0 TO 9;
9 BEGIN
10    -----
11    proc1: PROCESS (clk)
12        BEGIN
13            IF (clk'EVENT AND clk='1') THEN
14                IF (temp1=9) THEN
15                    temp1 := 0;
16                ELSE
17                    temp1 := temp1 + 1;
18                END IF;
19            END IF;
20        END PROCESS proc1;
21    -----
22    proc2: PROCESS (clk)
23        BEGIN
24            IF (clk'EVENT AND clk='1') THEN
25                IF (temp1=9) THEN
26                    IF (temp2=9) THEN
27                        temp2 := 0;
28                    ELSE
29                        temp2 := temp2 + 1;
30                    END IF;
31                END IF;
32            END IF;
33        END PROCESS proc2;
34    -----
35        digit1 <= temp1;
36        digit2 <= temp2;
37    END ARCHITECTURE;
38 -----

```

## ۴-۷ سیگنال در مقایسه با متغیر

انتخاب بین سیگنال و متغیر همیشه کار سرراست و ساده‌ای نیست. در زیر شش قانون به منظور تبیین تفاوت‌های بین این دو و نحوه استفاده از هر کدام معرفی شده است.

### قانون ۱: محل اعلان

#### Rule 1: Local of Declaration

SIGNAL: Can be declared in the declarative part of ENTITY, ARCHITECTURE, PACKAGE, BLOCK, or GENERATE.

VARIABLE: Can only be declared in sequential units (PROCESS and subprograms). The only exception is for *shared* variables, which can be declared in the same places as SIGNAL, but should only be modified by one sequential unit.

### قانون ۲: حوزه (محل استفاده)

#### Rule 2: Scope (Local of Use)

SIGNAL: Can be global (seen and modified in the whole code, including in sequential units).

VARIABLE: Always local (seen and modified only inside the sequential unit where it was created). To leave that unit, its value must be passed directly or indirectly to a signal. The only exception is for a *shared* variable, which can be global (seen by more than one sequential unit and also by concurrent statements, though it should be modified only by one sequential unit).

### قانون ۳: به روز رسانی

#### Rule 3: Update

SIGNAL: A new value is only available after the conclusion of the present run of the process or subprogram.

VARIABLE: Updated immediately, so its new value is ready to be used in the next line of code.

### قانون ۴: عملگر تخصیص

#### Rule 4: Assignment Operator

SIGNAL: Values are assigned using " $<=$ " (example: `sig <= 5;`).

VARIABLE: Values are assigned using " $:=$ " (example: `var := 5;`).

### قانون ۵: تخصیص‌های همزمان (چندگانه)

### Rule 5: Multiple Assignments

SIGNAL: Only one effective assignment is allowed in the whole code.

VARIABLE: Because its update is immediate, multiple assignments are fine.

### قانون ۶: تولید ثباتها

### Rule 6: Inference of Registers

SIGNAL: Flip-flops are inferred when an assignment to a signal occurs at the transition of another signal.

VARIABLE: Flip-flops are inferred when an assignment to a variable occurs at the transition of another signal and this variable's value is eventually passed directly or indirectly to a signal.

## خلاصه قوانین

Rule	SIGNAL	VARIABLE
1. Local of declaration	ENTITY, ARCHITECTURE, BLOCK, GENERATE, and PACKAGE (declaration in sequential code is forbidden)	Only in sequential units (PROCESS and subprograms), except shared variables (declared in ENTITY, ARCHITECTURE, BLOCK, GENERATE, PACKAGE)
2. Scope (local of use and of modification)	Can be global (used and modified anywhere in the code)	Local (used and modified only inside its own sequential unit), except shared variables (can be global, but modified by only one sequential unit)
3. Update	New value available only at the end of the current cycle	Updated immediately (new value ready to be used in the next line of code)
4. Assignment operator	Values are assigned using "<=" Example: sig<=5;	Values are assigned using ":=" Example: var:=5;
5. Multiple assignments	Only one assignment is allowed	Multiple assignments are fine (because update is immediate)
6. Inference of registers	Flip-flops are inferred when an assignment to a signal occurs at the transition of another signal	Flip-flops are inferred when an assignment to a variable occurs at the transition of a signal and this variable's value eventually affects a signal's value

Figure 7.1  
Summary of SIGNAL versus VARIABLE comparison.



**Example 7.2: SIGNAL versus VARIABLE Usage**

Consider the section of code shown below, which contains a SIGNAL (*sig*, declared in the declarative part of the architecture, line 4) and a VARIABLE (*var*, declared in the declarative part of the process, line 7). Check whether any of the six rules above was violated.

```

1  ...
2 -----
3  ARCHITECTURE example OF example IS
4      SIGNAL sig: INTEGER RANGE -8 TO 7;
5  BEGIN
6      PROCESS (clock)
7          VARIABLE var INTEGER RANGE -8 TO 7;
8      BEGIN
9          sig <= 0;
10         var := 0;
11         IF (clock'EVENT AND clock='1') THEN
12             sig <= sig + 1;
13             var := var + 1;
14             IF (sig=a) THEN ...
15             ELSIF (var=b) THEN ...
16             END IF;
17         END IF;
18         ...
19     END PROCESS;
20 END example;
21 -----

```



### Solution

Rule 1: Was not violated because both *sig* and *var* were declared in right places (lines 4 and 7).

Rule 2: Is fine as well because *var* was used only inside the process (*sig* can be used anywhere in the architecture code, including in the process).

Rule 3: The assignment in line 12 increments *sig*, but the new value will only be ready at the conclusion of the present process cycle, so the test in line 14 indeed compares *a* to an outdated value of *sig*. If that was not done on purpose (although not a recommended practice, one might want to actually compare *sig* to  $a - 1$ ), then an incorrect circuit will be inferred. Regarding *var*, being a variable, its update is immediate, so the value assigned in line 13 is already available to be used in line 15.

Rule 4: Was not violated because all assignments to *sig* and *var* (lines 9–10 and 12–13) employed the proper operators.

Rule 5: Was violated because lines 9 and 12 together make multiple assignments to *sig*. However, because this is a sequential code, such repetitions might be accepted by the compiler, but only the last assignment will survive, thus producing an incorrect circuit. Regarding *var*, no violation occurred because for variables multiple assignments (lines 10, 13) are fine.

Rule 6: This is not exactly a rule, but rather a consequence. In this example, *sig* will cause DFFs to be inferred because a value is assigned to it (line 12) at the transition of another signal (*clk*, line 11). The assignment to *var* in line 13 will infer registers if *var* affects a signal that leaves the process (which is the case in regular codes).



**Example 7.3: Counters with SIGNAL and VARIABLE**

Write a VHDL code that implements a regular 0-to-9 binary counter. Develop two solutions: with a signal and with a variable. If the same tests and assignments are made in both codes, will the resulting counters have the same counting range?

**Solution** A VHDL code for these counters is presented below. Counter 1 employs a signal, while counter 2 uses a variable. Even though they could be designed with just one process, two processes were used to make the code easier to inspect (this does not affect the inferred circuit).

The only input is *clk* (line 3), while the outputs are *count1* and *count2* (line 4), with a range from 0 to 9. The purpose of the range is just for the compiler to know the number of bits that it should use to represent each object, so 0-to-9 and 0-to-15, for example, are equivalent.

The process for counter 1 is in lines 11–20, under the (optional) label *with\_sig*. It employs a signal named *temp1*, specified in the declarative part of the architecture (line 8). Every time a rising clock edge occurs (line 13), *temp1* is incremented (line 14). When *temp1* reaches 10 (line 15), it is zeroed (line 16). The value of *temp1* is eventually passed to the *count1* output (line 19).

The process for counter 2 is in lines 22–32, under the (optional) label *with\_var*. Instead of a signal, a variable, called *temp2*, is now used. Its specification is made in the declarative part of the process (line 23). Every time a rising clock edge occurs (line 25), *temp2* is incremented (line 26). When *temp2* reaches 10 (line 27), it is zeroed (line 28). The value of *temp2* is eventually passed to the *count2* output (line 31).

As requested, the same assignments and tests were made in both processes, so now we need to determine the expected results. Based on rule 3, we know that a variable is updated immediately, while the transaction scheduled for a signal will only be concluded at the end of the process cycle. Consequently,  $\text{temp1} = 10$  will only occur in the eleventh process run, meaning that counter 1 will count from 0 to 10 instead of from 0 to 9. Counter 2, on the other hand, will have the correct range. This can be observed in the simulation results of figure 7.2.



```

1 -----
2 ENTITY counter IS
3     PORT (clk: IN BIT;
4             count1, count2: OUT INTEGER RANGE 0 TO 9;
5     END ENTITY;
6 -----
7 ARCHITECTURE dual_counter OF counter IS
8     SIGNAL temp1: INTEGER RANGE 0 TO 10;
9 BEGIN
10    -----counter 1: with signal:-----
11    with_sig: PROCESS(clk)
12    BEGIN
13        IF (clk'EVENT AND clk='1') THEN
14            temp1 <= temp1 + 1;
15            IF (temp1=10) THEN
16                temp1 <= 0;
17            END IF;
18        END IF;
19        count1 <= temp1;
20    END PROCESS with_sig;
21    -----counter 2: with variable:-----
22    with_var: PROCESS(clk)
23        VARIABLE temp2: INTEGER RANGE 0 TO 10;
24    BEGIN
25        IF (clk'EVENT AND clk='1') THEN
26            temp2 := temp2 + 1;
27            IF (temp2=10) THEN
28                temp2 := 0;
29            END IF;
30        END IF;
31        count2 <= temp2;
32    END PROCESS with_var;
33 END ARCHITECTURE;
34 -----

```

## ۵-۷ تولید ثباتها (The Inference of Registers)

در این بخش راجع به تعداد فلیپ فلاپهایی که کامپایلر از یک کد استنباط می‌کند، بحث می‌شود. همان طور که در قانون ۶ اشاره شد، هرگاه در لحظه‌ی تغییر یک سیگنال، مقداری به یک سیگنال دیگر تخصیص داده شود، این کار موجب تولید ثبات و اضافه شدن آن به مدار می‌شود. چنین تخصیصی، که طبیعتاً سنکرون است، باید داخل بخشی از یک کد «ترتیبی» انجام شده باشد. همچنین چنین تخصیصی معمولاً بعد از استفاده از اعلانی به صورت‌های زیر انجام می‌شود:

1. "IF clock'EVENT ..."
2. "WAIT UNTIL clock'EVENT ...".

یک متغیر نیز می‌تواند موجب تولید ثبات شود؛ هر گاه در لحظه‌ی تغییر یک سیگنال، مقداری به متغیر نسبت داده شود و آن متغیر بعداً روی سیگنالی بخواهد اثر بگذارد، این کار موجب تولید ثبات در مدار خواهد شد.

مثال: اعلانهای زیر را در نظر بگیرید:

```
SIGNAL clk: BIT;
SIGNAL sig1: BIT_VECTOR( 7 DOWNTO 0 );
SIGNAL sig2: INTEGER RANGE 0 TO 7;
VARIABLE var: BIT_VECTOR(3 DOWNTO 0);
```

در این صورت در کد زیر تمام سه شیء موجب تولید ثبات می‌شوند؛ یا به عبارت دیگر، ذخیره (رجیستر) می‌شوند زیرا در لحظه‌ی تغییر یک سیگنال (clk) به هر کدام از این سه شیء یک تخصیص نسبت داده شده است. در کل به تعداد  $15 = 8 + 3 + 4$  فلیپ فلاپ DFF تولید می‌شود.

```
IF (clk'EVENT AND clk='1') THEN
    sig1 <= x;
    sig2 <= y;
    var := z;
END IF;
```

در کد زیر تنها sig2 و var ذخیره شده و به تعداد  $7 = 4 + 3$  فلیپ فلاپ تولید خواهد شد.

```

PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        sig2 <= y;
        var := z;
    END IF; ...
    sig1 <= x;
END PROCESS;

```



#### Example 7.4: DFF with $q$ and $qbar$

Figure 7.3 shows four implementations involving DFFs with  $q$  and  $qbar$  outputs. The circuits in (a) and (b) are equivalent, with the only difference that the latter does not have a built-in  $qbar$  output, so an inverter is needed to construct it. On the other hand, the circuit in (c) is only *functionally* equivalent to (a)–(b) and that in (d) is not equivalent at all ( $qbar$  is one clock period behind  $q$ ). Examine the codes below and determine which of these circuits each code implements. Then compile the codes and check in the RTL viewer and/or fitter equations if the inferred circuits match your answers.

```

1 -----
2 ENTITY flipflop IS
3 PORT (d, clk: IN BIT;
4         q: BUFFER BIT;
5         qbar: OUT BIT);
6 END ENTITY;
7 -----
8 ARCHITECTURE arch OF flipflop IS
9 BEGIN
10 proc1: PROCESS (clk)
11 BEGIN
12     IF clk'EVENT AND clk='1' THEN
13         q <= d;
14         qbar <= NOT d;
15     END IF;
16 END PROCESS proc1;
17 END ARCHITECTURE;
18 -----
10 proc2: PROCESS (clk)
11 BEGIN
12 IF clk'EVENT AND clk='1' THEN
13     q <= d;
14     qbar <= NOT q;
15 END IF;
16 END PROCESS proc2;
-----
10 proc3: PROCESS (clk)
11 BEGIN
12 IF clk'EVENT AND clk='1' THEN
13     q <= d;
14 END IF;
15 END PROCESS proc3;
16 qbar <= NOT q;
-----
```

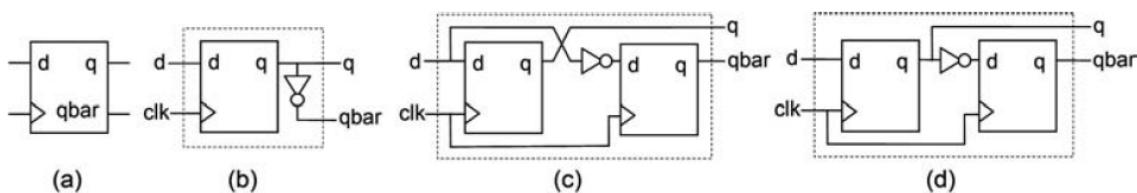


Figure 7.3  
Flip-flops of example 7.4.

**Solution** From the code of *proc1*, DFFs are inferred for *q* and *qbar* because values are assigned to both (lines 13–14) at the transition of another signal (*clk*, line 12). Since both assignments are related to *d*, the circuit of figure 7.3c is expected to result. Applying the same reasoning to *proc2*, again two DFFs are inferred; however, the assignment to *qbar* (line 14) is related to *q*, thus causing the circuit of figure 7.3d to be inferred. Finally, in the third code, the assignment to *qbar* (line 16) is no longer under the IF *clk'EVENT...* test, so *qbar* is not registered, resulting the circuit of figure 7.3b. It is important to mention that, depending on the compiler being used and the target CPLD/FPGA device, during the optimization phase the compiler might be able to simplify *proc1* to produce the same circuit as *proc3* (but the approach of *proc3* is always recommended).

### Example 7.5: Over-registered Counter

Consider the counter designed in example 6.6.

- How many flip-flops are needed to implement it?
- What happens if line 27 (END IF;) of that code is moved to the position between lines 41 and 42?

**Solution**

- To obtain the 1 Hz signal (*counter1*),  $\lceil \log_2 50M \rceil = 26$  DFFs are needed. For the 0-to-9 counter (*counter2*), four DFFs are required. Hence the total is 30 DFFs.
- This causes the CASE statement to be under the IF *clk'EVENT...* test, so *ssd* will be registered. In summary, in figure 6.9 an extra block, containing seven DFFs, would be included between the SSD driver block and the display (then totaling 37 flip-flops). Note that these DFFs are not necessary because the input to the SSD driver is already registered (this is a very popular mistake). The reader is invited to compile the code to check these answers (exercise 7.6).

## ۶-۷ مدارات دو لبه<sup>۱</sup>

فرض کنید که در یک کاربرد خاص به فلیپ فلاب‌های دو لبه (یعنی فلیپ فلاب‌هایی که در هر دو لبه‌ی کلاک داده را ذخیره می‌کنند) نیاز داشته باشیم. اما از آن طرف اگر افزاره‌ی

<sup>1</sup> Dual-Edge Circuits

تنها مجهر به فلیپ فلاپ‌های تک لبه باشد (که در تقریباً تمام افزارهای موجود چنین مطلبی صادق است) آن‌گاه کامپایلر قادر به سنتز مستقیم کد شما روی یک فلیپ فلاپ نخواهد بود. در این حالت دو راه که ممکن است به ذهن برسد در زیر نشان داده شده است. فرآیند واقع در سمت چپ، شامل شمارنده‌ای است که می‌بایست در هر دو لبه‌ی کلاک عمل شمارش (افزایش) را انجام دهد لذا دو بار از `clk'EVENT` (در این فرآیند) استفاده شده است. فرآیند سمت راست همان هدف فرآیند قبلی را دارد اما از راه نسبتاً متفاوتی عمل کرده است. در این فرآیند، کلاک در لیست حساسیت قرار داده شده لذا هر زمان که کلاک تغییر کند فرآیند اجرا می‌شود و با توجه به این که همراه با `clk'EVENT` از هیچ شرط دیگری استفاده نشده است، این طور تصور می‌شود که هر دو لبه‌ی کلاک منجر به افزایش شمارنده می‌شوند. در تکنولوژی‌های مبتنی بر فلیپ فلاپ‌های تک لبه، هر دو کد زیر منجر به توقف کامپایلر می‌شوند؛ البته در مورد کد سمت راست برعکس کامپایلرها به طور خودکار و پیش فرض شرط AND را در نظر گرفته و کد را سنتز می‌کنند اما در هر حال در این صورت نیز همان حالت تک لبه رخ داده و به منظور خود نخواهیم رسید.

-----Not OK:-----

```
PROCESS (clk)
  VARIABLE count: INTEGER RANGE 0 TO 15;
BEGIN
  IF (clk'EVENT AND clk='1') THEN
    count := count + 1;
  ELSIF (clk'EVENT AND clk='0') THEN
    count := count + 1;
  END IF;
  ...
END PROCESS;
```

-----Not OK:-----

```
PROCESS (clk)
BEGIN
  IF (clk'EVENT) THEN
    count <= count + 1;
  END IF;
  ...
END PROCESS;
```

مطلوب فوق به این معنا نیست که نمی‌توان یک فرآیند یا زیربرنامه را در هر دو لبه‌ی کلاک اجرا کرد. برای مثال کد زیر معتبر است زیرا در آن از دو متغیر «مجزا» (یا سیگنال هم می‌تواند مورد استفاده قرار بگیرد) استفاده شده است که یکی تنها در لبه‌های بالاروندهای کلاک و دیگری در لبه‌های پایین‌روندهای آن کار می‌کند.

```
-----OK-----
PROCESS (clk)
    VARIABLE count1, count2: INTEGER RANGE ...;
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        count1 := count1 + 1;
    ELSIF (clk'EVENT AND clk='0') THEN
        count2 := count2 + 1;
    END IF;
    ...
END PROCESS;
```

### Example 7.6: Dual-Edge Flip-Flop

Write a VHDL code from which a circuit that resembles a dual-edge flip-flop can be inferred (assuming that no dual-edge DFFs are available in the target device).

**Solution** One alternative to construct a dual-edge DFF is shown in figure 7.4 (Pedroni 2008), which employs two D-type latches (DLs) connected in parallel, followed by a multiplexer (in figure 7.4, the DLs too are implemented with multiplexers, with a feedback loop).

A code for this circuit is presented below, employing sequential code with the IF statement (one for each mux). Note that although a flip-flop will result from this code, a fully concurrent code could have been employed because the individual units (muxes) are combinational circuits, in which case the WHEN statement would render a shorter code.

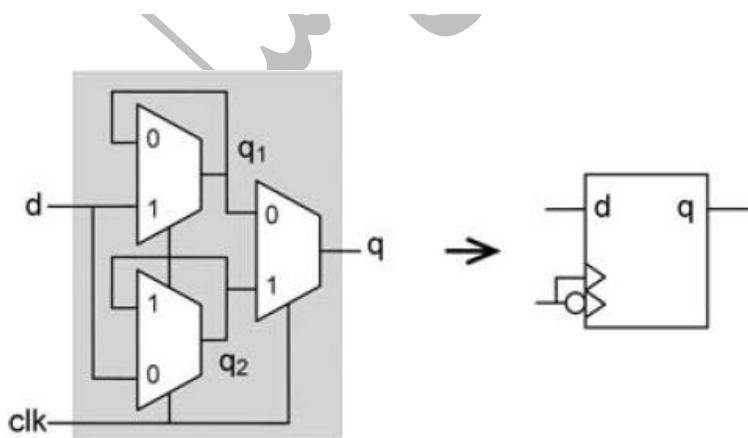


Figure 7.4

Dual-edge DFF implemented with multiplexers (the first two operate as DLs).

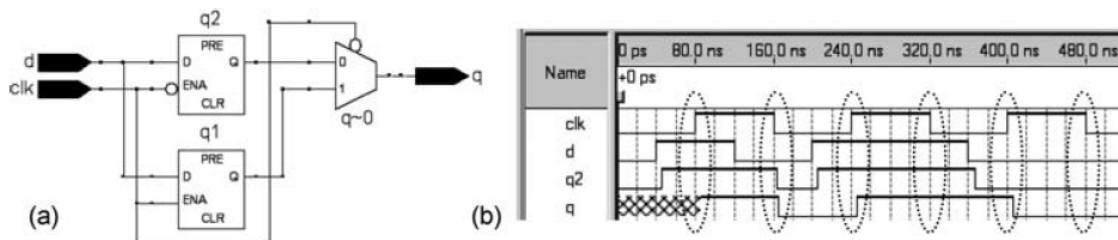
```

1 -----
2 ENTITY dual_edge_dff IS
3 PORT (d, clk: IN BIT;
4         q: OUT BIT);
5 END ENTITY;
6 -----
7 ARCHITECTURE structure OF dual_edge_dff IS
8     SIGNAL q1, q2: BIT;
9 BEGIN
10 PROCESS(clk, d)
11     BEGIN
12         ---mux for q1:-----
13         IF (clk='0') THEN q1 <= q1;
14         ELSE q1 <= d;
15         END IF;
16         ---mux for q2:-----
17         IF (clk='0') THEN q2 <= d;
18         ELSE q2 <= q2;
19         END IF;
20         ---mux for q:-----
21         IF (clk='0') THEN q <= q1;
22         ELSE q <= q2;
23         END IF;
24     END PROCESS;
25 END ARCHITECTURE;
26 -----

```



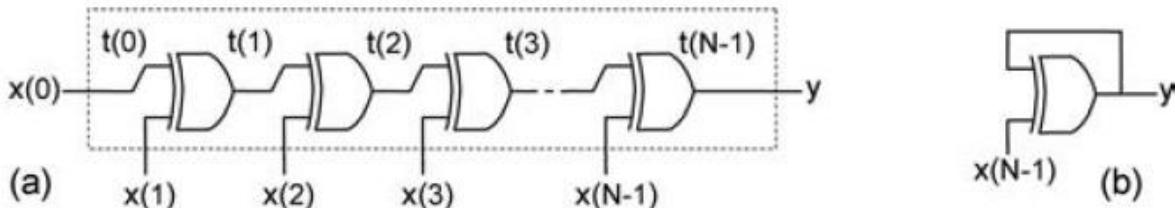
The RTL view produced by the compiler is shown in figure 7.5a, which matches our circuit perfectly. Simulation results are depicted in figure 7.5b, where it can be observed that both clock transitions (highlighted) are indeed active.



**Figure 7.5**  
Dual-edge DFF of example 7.6. (a) RTL view; (b) Simulation results showing that both clock edges are active.

## ۷-۷ تخصیص‌های همزمان روی سیگنال

با توجه به قانون ۵ از شکل ۱-۷ در سرتاسر کد تنها یک بار روی یک سیگنال مفروض می‌توان عمل تخصیص را انجام داد. برای مثال طراحی یک آشکارساز توازن مطابق با شکل ۷-۶-الف را در نظر بگیرید. این مدار، تابع  $y = x_0 \oplus x_1 \oplus x_2 \oplus \dots \oplus x_{N-1}$  را پیاده‌سازی می‌کند. هرگاه تعداد '۱'‌های موجود در ورودی عددی فرد باشد، مقدار خروجی این مدار برابر '۱' می‌شود. در زیر دو نمونه کد برای طراحی چنین مداری نشان داده شده است.



**Figure 7.6**  
Parity detector.

```

1 -----
2 ENTITY parity_det IS
3   GENERIC (N: POSITIVE := 8);
4   PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5         y: OUT BIT);
6 END ENTITY;
7 -----
8 ARCHITECTURE not_ok OF parity_det IS
9   SIGNAL temp: BIT;
10 BEGIN
11   temp <= x(0);
12   gen: FOR i IN 1 TO N-1 GENERATE
13     temp <= temp XOR x(i);
14   END GENERATE;
15   y <= temp;
16 END ARCHITECTURE;
17 -----
7 -----
8 ARCHITECTURE not_ok OF ...
9 SIGNAL temp: BIT;
10 BEGIN
11 PROCESS (x)
12 BEGIN
13   temp <= x(0);
14   FOR i IN 1 TO N-1 LOOP
15     temp <= temp XOR x(i);
16   END LOOP;
17   y <= temp;
18 END PROCESS;
19 END ARCHITECTURE;
20 -----

```

اولین کد از نوع همزمان بوده و مبتنی بر GENERATE است. دومین کد از نوع ترتیبی و مبتنی بر LOOP و PROCESS می‌باشد. در هر دو کد temp یک سیگنال (خط ۹) بوده در هر دو کد روی این سیگنال تخصیص همزمان انجام می‌دهند (خطوط ۱۱ و ۱۳ در اولین کد و خطوط ۱۳ و ۱۵ در کد دوم). بنابراین هیچ یک از دو مدار قادر به پیاده‌سازی صحیح هدف مورد نظرمان نخواهد بود.

در مورد اولین کد، کامپایلر پیغام خطا (مبنی بر تخصیص همزمان به یک سیگنال) داده و از فرآیند کامپایل خارج خواهد شد. اما کد دوم قابل سنتز است زیرا دو عمل تخصیص پشت سر هم ظاهر

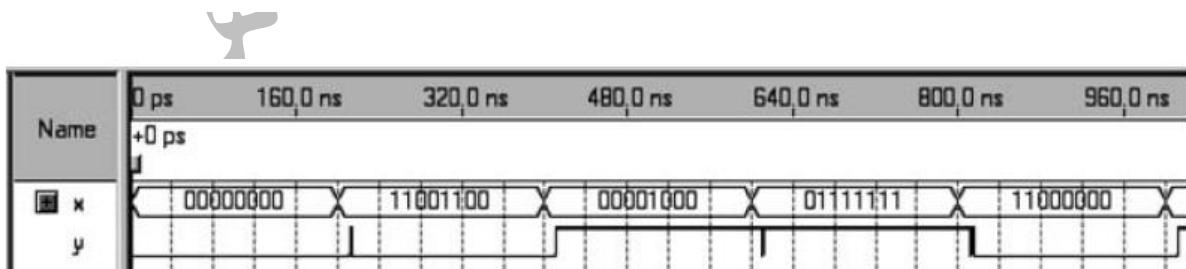
شده‌اند اما از آنجائی که مقدار جدید یک سیگنال تنها در پایان دور اجرایی از کد ترتیبی اعمال خواهد شد، تنها آخرین تخصیص، یعنی  $y = y \oplus x_{N-1}$  معتبر و موثر خواهد بود؛ بنابراین، این کد منجر به مدار نشان داده شده در شکل ۷-۶-ب خواهد شد. طبیعتاً این مدار همان مدار مورد نظر ما نیست. نکته‌ی دیگر در مورد این مدار این است که اگر  $x_{N-1} = 1$  انتخاب شود، مدار تبدیل به یک نوسان‌ساز خواهد شد.

حل مساله‌ی مطرح شده در اینجا در زیر نشان داده شده است. در این روش، از سیگنالی استفاده شده که بعد آن یکی بیشتر از ابعاد سیگنال محاسبه شده می‌باشد. در این مثال  $y$  تک بیتی (یا اسکالر) بوده و بنابراین باید از سیگنالی یک بعدی (1D) که در خط ۹ نشان داده شده است استفاده شود. بنابراین، به جای  $temp$  از  $temp(N-1, 0)$  که در شکل ۷-۶-الف یک بردار داخلی بوده و با  $t$  نمایش داده شده، استفاده شده است. حال توجه کنید که هر عمل تخصیص شامل «بخش متفاوتی» از  $temp$  بوده و بنابراین، قانون ۵ نقض نمی‌شود! لذا مدار صحیح و مورد انتظارمان ایجاد شده است. نتایج شیوه‌سازی برای این مدار در شکل ۷-۷ نشان داده شده است.

```

7 -----
8 ARCHITECTURE ok OF parity_det IS
9     SIGNAL temp: BIT_VECTOR(N-1 DOWNTO 0);
10 BEGIN
11     temp(0) <= x(0);
12     gen: FOR i IN 1 TO N-1 GENERATE
13         temp(i) <= temp(i-1) XOR x(i);
14     END GENERATE;
15     y <= temp(N-1);
16 END ARCHITECTURE;
17 -----

```



**Figure 7.7**  
Simulation results from the parity detector of figure 7.6.

به این نکته‌ی مهم توجه داشته باشید که این نوع مشکل (که در اینجا بررسی شد) در صورت استفاده از متغیرها (VARIABLE) رخ نخواهد داد زیرا مقدار متغیرها بلافاصله به روز رسانی شده و بنابراین امکان استفاده از تخصیص‌های همزمان وجود دارد!.

**Example 7.7: Generic Hamming Weight with Concurrent Code**

Exercise 5.7 asked to design a circuit that computes the number of '1's in a binary vector using the GENERATE statement. Exercise 6.6 asked the same thing, but using the LOOP statement. While the latter is simple, the former requires some additional effort. That exercise is an excellent opportunity to illustrate the extra dimension required in the auxiliary signal in order to allow multiple assignments to be made to it, because now a *different part* of the signal is involved in each iteration. Design such a circuit using only concurrent code (operators, WHEN, SELECT, GENERATE).

**Solution** Two solutions are presented below, both with  $N$  (size of the input vector) declared as a generic parameter (line 3). In the first solution, *temp* is declared as an integer in the 0 to  $N$  range, because that is the range of *y* (the value of *temp* must be passed to *y* eventually—line 15). For the reasons explained above, this solution is not fine (it contains multiple assignments to *temp*—lines 11 and 13). The second solution adopts the described approach, which increases the dimension of *temp* by one unit (from 1D to 1D  $\times$  1D). This is done in lines 9–10. Now the assignments in lines 12 and 14 are no longer to the same portions of *temp*. Note that *y* is still 1D, so only the last value of *temp* is passed to *y* (line 16).

```

1 -----
2 ENTITY hamm_weight IS
3   GENERIC (N: POSITIVE := 8);
4   PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5         y: OUT INTEGER RANGE 0 TO N);
6 END ENTITY;
7 -----
8 ARCHITECTURE not_ok OF hamm_weight IS
9   SIGNAL temp: INTEGER RANGE 0 TO N;
10 BEGIN
11   temp <= 0;
12   temp(0) <= 0;
13   gen: FOR i IN 1 TO N-1 GENERATE
14     temp(i) <= temp(i-1) + 1 WHEN x(i)='1' ELSE temp(i-1);
15   END GENERATE;
16   y <= temp(N-1);
17 END ARCHITECTURE;
18 -----

```

## فصل هشتم

### بسته و جزء

هدف از این فصل و فصل بعدی معرفی چهار واحد VHDL است که هدف اصلی آنها

- بخش‌بندی کرد،
- به اشتراک گذاری کرد،
- استفاده‌ی مجدد از کد

می‌باشد. این بخش‌ها عبارتند از: COMPONENT، PACKAGE، PROCEDURE و FUNCTION.

این واحدها عمدتاً در جایی خارج از کد اصلی و در داخل کتابخانه قرار داده می‌شوند لذا به آنها واحدهای سطح سیستمی<sup>۱</sup> هم گفته می‌شود.

شکل ۸-۸ رابطه‌ی بین واحدهای سطح سیستمی و کد اصلی را نشان می‌دهد. اگر کتابخانه‌ی خاصی در کد اصلی اعلان شود آن گاه بخش‌های این کتابخانه در کد اصلی قابل استفاده خواهد بود. یک کتابخانه می‌تواند شامل بسته‌ها (هر بسته می‌تواند شامل توابع، روال‌ها و چندین نوع اعلان باشد) و دیگر طراحی‌ها (این طراحی‌ها را می‌توان داخل کد اصلی به کمک کلمه‌ی کلیدی COMPONENT نمoneه‌سازی کرد) باشد. طراحی‌های انجام شده در این فصل و فصل بعدی مستلزم ایجاد و استفاده از کتابخانه است.

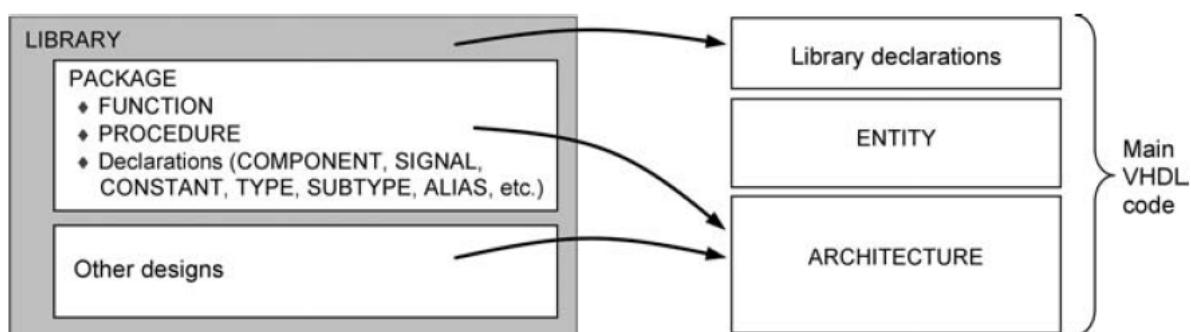


Figure 8.1

Relationship between the main code and the units intended mainly for system-level design (libraries).

### (PACKAGE) ۲-۸ بسته

<sup>1</sup> System-Level Units

برای ساختن یک بسته به (حداکثر) دو بخش از کد به نامهای PACKAGE و BODY نیاز داریم (یعنی «بسته» و «بدنه‌ی» آن). گرامر ساده شده‌ی این دو بخش در زیر نشان داده شده است.

```

PACKAGE package_name IS
    declarative_part
END [PACKAGE] [package_name];
-----
[PACKAGE BODY package_name IS
    [subprogram_body]
    [deferred_constant_specifications]
END [PACKAGE BODY] [package_name]];

```

اولین بخش (PACKAGE) فقط باید شامل اعلان‌ها باشد. این اعلان‌ها می‌توانند شامل اعلان زیربرنامه، اعلان نوع، اعلان زیرنوع، اعلان ثابت، اعلان سیگنال، اعلان متغیرهای به اشتراک گذاشته شده، اعلان همپوشان (alias)، اعلان جزء (COMPONENT)، اعلان مختصه، توصیف مختصه (Attribute Specification)، توصیف قطع اتصال<sup>۱</sup>، عبارت use، اعلان کلیشه یا قالب گروه<sup>۲</sup>، و اعلان گروه باشند.

دومین بخش (PACKAGE BODY) تنها زمانی نیاز است که در بخش اول اعلانی مربوط به یک زیربرنامه یا یک ثابت به تعویق افتاده<sup>۳</sup> (یعنی مقدار آن تعریف نشده است) وجود داشته باشد. در این صورت بدنه‌ی کامل زیربرنامه یا توصیف کامل ثابت باید در بخش دوم آورده شود. سرآیند کامل زیربرنامه<sup>۴</sup> به طور دقیق، همان طور که در PACKAGE آمده، در بخش دوم نیز باید تکرار شود. بخش دوم شامل همان اعلان‌های انجام شده در بخش اول است به جز موارد زیر:

- اعلان سیگنال،
- اعلان جزء،
- اعلان مختصه،
- توصیف مختصه،
- توصیف قطع اتصال.

<sup>1</sup> Disconnection Specification

<sup>2</sup> Group Template Declaration

<sup>3</sup> Deferred Constant

<sup>4</sup> Full Subprogram Header

**Example** The PACKAGE below is called *my\_package* and contains only TYPE, SIGNAL, and (complete) CONSTANT declarations (thus PACKAGE BODY is not necessary).

```
-----
PACKAGE my_package IS
    TYPE matrix IS ARRAY (1 TO 3, 1 TO 3) OF BIT;
    SIGNAL x: matrix;
    CONSTANT max1, max2: INTEGER := 255;
END PACKAGE;
```

در VHDL 2008 یک سرآیند به PACKAGE به منظور فراهم آوردن امکان اعلان ثابت‌های عام (GENERIC) افروده شده است. در بخش اعلان، اعلان‌های زیر مجاز شده‌اند:

- اعلان نمونه‌سازی زیربرنامه،
- اعلان بسته،
- اعلان نمونه‌سازی بسته،
- اعلان PSL.

همچنان بدنی بسته (PACKAGE BODY) محلی برای ساخت بدنی زیربرنامه‌ها و مشخص کردن ثابت‌های به تعویق افتاده می‌باشد.

### Example 8.1: PACKAGE with FUNCTION and Deferred CONSTANT

The code below shows, in lines 1–9, a PACKAGE called *my\_package*, which contains a deferred constant (called *flag*) and also a function (called *down\_edge*), so in this case a PACKAGE BODY is needed. Knowing that the constant value must be '1' and that this function should be equivalent to "clk'EVENT AND clk = '0'", write a code for the PACKAGE BODY.

```
1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 PACKAGE my_package IS
6     CONSTANT flag: STD_LOGIC;
7     FUNCTION down_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
8 END my_package;
9 -----
10 PACKAGE BODY my_package IS
11     CONSTANT flag: STD_LOGIC := '1';
12     FUNCTION down_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
13     BEGIN
14         RETURN (s'EVENT AND s='0');
15     END down_edge;
16 END my_package;
17 -----
```

**Solution** The code was included in lines 10–16, with the deferred constant specified in line 11 and the function body in lines 12–15 (this function returns TRUE when a negative clock edge occurs). Note that the function header (line 12) is an exact copy of the function declaration (line 7). Observe also that the package *std\_logic\_1164* was included in lines 2–3 because the data type STD\_LOGIC was used in the code.

### ۳-۸ جزء (COMPONENT)

یک جزء در حقیقت یک کد معمولی است (یعنی شامل «اعلان کتابخانه/بسته + موجودیت + معماری» است). اما اعلان این کد به صورت «جزء» امکان استفاده مجدد آن و نیز ایجاد طراحی‌های «سلسله‌مراتبی»<sup>۱</sup> را فراهم می‌کند. زیرسیستم‌های دیجیتالی متداول مانند جمع‌کننده‌ها، ضرب‌کننده‌ها، مالتی‌پلکسرها و مشابه آن معمولاً از طریق این تکنیک در طراحی‌ها استفاده و کامپایل می‌شوند. طراحی‌های مبتنی بر استفاده از اجزاء را معمولاً «طراحی‌های ساختاری»<sup>۲</sup> می‌گویند.

در بخش‌های زیر می‌توان یک جزء را اعلان نمود:

- ARCHITECTURE
- PACKAGE
- GENERATE
- BLOCK

برای استفاده از اجزاء به دو بخش از کد نیازمندیم: یکی برای اعلان جزء و دیگری برای نمونه‌سازی آن. هر دو کار در گرامرها ساده شده زیر نشان داده شده است:

COMPONENT declaration:

```
COMPONENT component_name [IS]
  [GENERIC (
    const_name: const_type := const_value;
    const_name: const_type := const_value;
    ...);
  PORT (
    port_name: port_mode signal_type;
    port_name: port_mode signal_type;
    ...);
  END COMPONENT [component_name];
```

<sup>1</sup> Hierarchical Designs

<sup>2</sup> Structural Designs

## COMPONENT instantiation:

```
label: [COMPONENT] component_name
        [GENERIC MAP (generic_list)]
        PORT MAP (port_list);
```

اولین گرامر مربوط به «اعلان جزء» بوده و باید به صورت یک کپی دقیق و کامل از موجودیت طراحی مورد نظرمان (که قرار است نمونه‌سازی شود) باشد. از توصیف GENERIC تنها در صورتی باید حتماً استفاده شود که جزء شامل یک لیست<sup>1</sup> GENERIC بوده و حداقل یک یا چند مقدار متعلق به این لیست توسط طراحی که نمونه‌سازی را انجام می‌دهد تغییر داده خواهد شد (این تغییر توسط GENERIC MAP انجام خواهد شد).

گرامر دوم مربوط به «نمونه‌سازی جزء» بوده و حتماً باید با یک برچسب شروع شود. پس از این برچسب اقلام زیر خواهد آمد:

- کلمه‌ی اختیاری COMPONENT،
- نام جزء (همان نام موجودیت استفاده شده در گرامر اول)،
- اعلان اختیاری GENERIC MAP،
- اعلان PORT MAP. این اعلان تناظری<sup>2</sup> (یا نگاشتی<sup>2</sup>) بین نام پورت‌های استفاده شده در «طراحی اصلی» (انجام شده در گرامر اول) و «طراحی جدید» (که عمل نمونه‌سازی را انجام می‌دهد)، برقرار می‌کند.

استفاده از GENERIC MAP زمانی ضروری است که جزء اصلی در سرآیند موجودیت خود دارای توصیف (یا لیست<sup>1</sup> GENERIC) بوده و حداقل یکی از مقادیر ذکر شده در این لیست توسط طراحی جدید باید تغییر کند.

<sup>1</sup> Association

<sup>2</sup> Mapping

**Example** Say that a 3-input NAND gate, called *nand3*, has been previously designed and we now want to use it as part of a new design. This is illustrated in the code below. The first part contains the *component declaration*, which must be exactly as in the original entity. The second part shows the *component instantiation*, labeled *nand\_gate*, with the current *x1*, *x2*, *x3*, and *y* ports assigned to the original *a1*, *a2*, *a3*, and *b* ports, respectively. This is called *positional mapping*, because the first signal in one list corresponds to the first signal in the other, the second in one to the second in the other, and so on.

```
-----COMPONENT declaration-----
COMPONENT nand3 IS
    PORT (a1, a2, a3: IN STD_LOGIC; b: OUT STD_LOGIC);
END COMPONENT;

-----COMPONENT instantiation-----
nand_gate: nand3 PORT MAP (x1, x2, x3, y);
```

### گزینه‌های نگاشت

کلمه‌ی PORT MAP در واقع لیستی است که پورتهای مدار واقعی را به پورتهای مدار ار قبل طراحی شده (طراحی اصلی) «ربط» می‌دهد. این رابطه یا نگاشت به دو صورت مکانی<sup>۱</sup> و نامی<sup>۲</sup> قابل انجام است. مثال زیر این مطلب را توضیح می‌دهد. در این مثال از مدار *nand3* (که در مثال قبلی آن را معرفی کردیم) استفاده مجدد می‌شود.

```
----- Component instantiation: -----
nand3_1: nand3 PORT MAP (x1, x2, x3, y); --positional mapping
nand3_2: nand3 PORT MAP (a1=>x1, a2=>x2, a3=>x3, b=>y); --nominal mapping
nand3_3: nand3 PORT MAP (x1, x2, x3, OPEN); --positional mapping
nand3_4: nand3 PORT MAP (a1=>x1, a2=>x2, a3=>x3, b=>OPEN); --nominal mapping
```

دو نمونه‌سازی اول با هم معادل هستند تنها تفاوت آنها در انتخاب روش برای انجام نگاشت است. دو نمونه‌سازی آخر نیز با هم معادل هستند. در این دو نمونه‌سازی از کلمه‌ی OPEN به منظور تأکید بر لزوم عدم اتصال یکی از خروجی‌های مدار استفاده شده است.

### گزینه‌های اعلان جزء

شکل ۲-۸ دو راه متقابل جهت استفاده از اجزاء را نشان می‌دهد. در شکل (الف) اجزاء در کتابخانه قرار داشته و کد اصلی شامل اعلان‌های اجزاء به همراه نمونه‌سازی آنها می‌باشد. در شکل (ب) اعلان‌ها نیز در یک کتابخانه (در داخل یک بسته‌ی مشخص) قرار دارند؛ بنابراین در این روش، در کد اصلی تنها به نمونه‌سازی نیاز می‌باشد.

<sup>1</sup> Positional

<sup>2</sup> Nominal

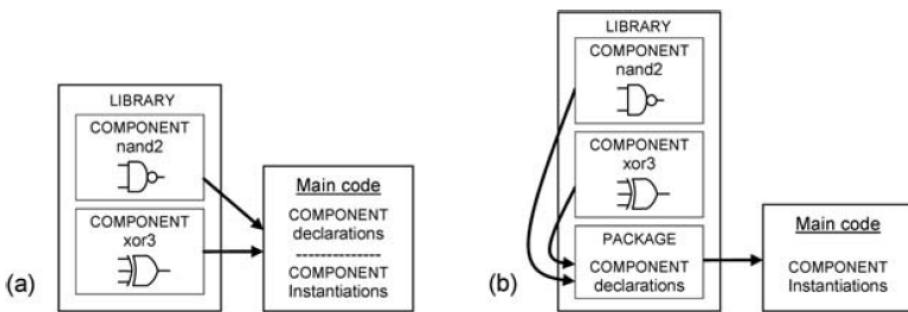


Figure 8.2

Typical COMPONENT usage: (a) With the declarations and instantiations in the main code; (b) With the declarations in a separate package, thus with only the instantiations in the main code.

### گزینه‌های اسambil کردن فایل/پروژه

کدی که شامل `COMPONENT` باشد را به چند طریق می‌توان وارد فرآیند کامپایل نمود:

روش اول: کد به طور کامل داخل یک فایل قرار داده می‌شود. نام فایل باید همان نام موجودیت مدار (طراحی) اصلی باشد. در این روش، اعلان‌های اجزاء نیز داخل کد اصلی قرار داده می‌شوند (مطابق با شکل ۲-۸-الف). بنابراین در مجموع می‌توان گفت در این روش، تمام مطالب و اطلاعات داخل یک فایل (بزرگ) قرار داده می‌شوند.

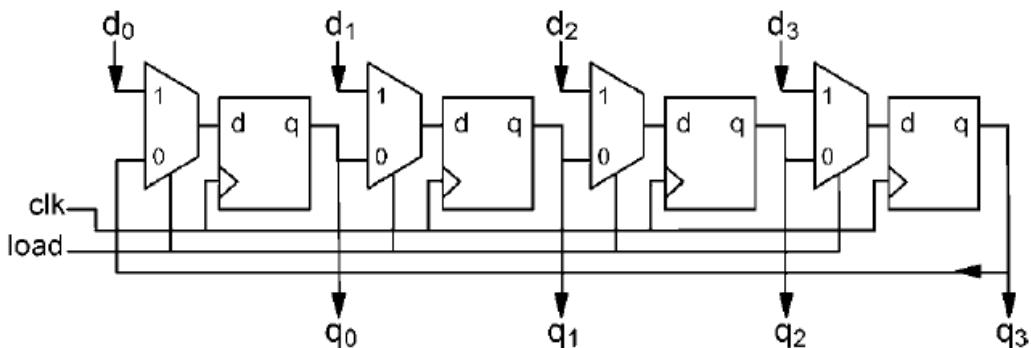
روش دوم: در این روش هر جزء قبلاً در یک پروژه‌ی جداگانه طراحی شده و پس از کامپایل داخل کتابخانه کاری (work library) خودش یا کتابخانه‌ی کاری فعلی قرار داده می‌شود. اگر در کتابخانه‌ی خودش قرار داده شود، آن گاه در کد اصلی باید از عبارت `use` استفاده کرد تا به کتابخانه‌ی مربوطه اشاره شود. در این روش اعلان‌های اجزاء در کد اصلی (که قرار است عمل نمونه‌سازی را انجام دهد) قرار داده می‌شوند (مطابق با شکل ۲-۸-الف).

روش سوم: این روش مشابه با روش دوم است اما با این تفاوت که اعلان‌های اجزاء در یک بسته‌ی جداگانه (مانند شکل ۲-۸-ب) قرار داده می‌شوند. در این روش باید در کد اصلی به کمک عبارت `use` به بسته‌ی مزبور اشاره نمود.

مثال زیر نحوه‌ی استفاده از روش اول را نشان می‌دهد.

### Example 8.2: Circular Shift Register with COMPONENT

Figure 8.3 shows a circular shift register with a programmable rotating sequence. The inputs are  $clk$ ,  $load$ , and  $d = d_0d_1d_2d_3$ , with the latter employed to load the desired sequence into the flip-flops when  $load = '1'$ . The output is  $q = q_0q_1q_2q_3$ , which displays the instantaneous value of the rotating sequence. Note that all cells in this circuit are alike and composed of a multiplexer plus a DFF. Design this circuit using COMPONENT to instantiate the multiplexers and DFFs. Adopt the following approach: use method 1 of section 8.3 to enter the code, with *positional mapping* in it.



**Figure 8.3**

Programmable circular shift register of example 8.2.

**Solution** A corresponding VHDL code is presented below. It consists of just one VHDL file (called *circular\_shift.vhd* in this example), having all component declarations included in the main code (lines 11–17), hence complying with method 1. The labels chosen for the component instances are *mux1*, *mux2*, and so on for the multiplexers (lines 20–23), and *dff1*, *dff2*, and so on for the DFFs (lines 24–27). An internal signal called *i(0:3)* was declared in line 9 to provide interface between the muxes and the flip-flops.

```

1  -----Multiplexer-----
2  ENTITY mux IS
3      PORT (a, b, sel: IN BIT;
4             x: OUT BIT);
5  END ENTITY;
6  -----
7  ARCHITECTURE mux OF mux IS
8  BEGIN
9      x <= a WHEN sel='0' ELSE b;
10 END ARCHITECTURE;
11 -----
```

```

1 -----Flip-flop-----
2 ENTITY flipflop IS
3   PORT (d, clk: IN BIT;
4         q: OUT BIT);
5 END ENTITY;
6 -----
7 ARCHITECTURE flipflop OF flipflop IS
8 BEGIN
9   PROCESS (clk)
10  BEGIN
11    IF (clk'EVENT AND clk='1') THEN
12      q <= d;
13    END IF;
14  END PROCESS;
15 END ARCHITECTURE;
16 ----

1 -----Main code-----
2 ENTITY circular_shift IS
3   PORT (clk, load: IN BIT;
4         d: IN BIT_VECTOR(0 TO 3);
5         q: BUFFER BIT_VECTOR(0 TO 3));
6 END ENTITY;
7 -----
8 ARCHITECTURE structural OF circular_shift IS
9   SIGNAL i: BIT_VECTOR(0 TO 3);
10  -----
11  COMPONENT mux IS
12    PORT (a, b, sel: IN BIT; x: OUT BIT);
13  END COMPONENT;
14  -----
15  COMPONENT flipflop IS
16    PORT (d, clk: IN BIT; q: OUT BIT);
17  END COMPONENT;
18  -----
19 BEGIN
20   mux1: mux PORT MAP (q(3), d(0), load, i(0));
21   mux2: mux PORT MAP (q(0), d(1), load, i(1));
22   mux3: mux PORT MAP (q(1), d(2), load, i(2));
23   mux4: mux PORT MAP (q(2), d(3), load, i(3));
24   dff1: flipflop PORT MAP (i(0), clk, q(0));
25   dff2: flipflop PORT MAP (i(1), clk, q(1));
26   dff3: flipflop PORT MAP (i(2), clk, q(2));
27   dff4: flipflop PORT MAP (i(3), clk, q(3));
28 END ARCHITECTURE;
29 -----

```

Simulation results are shown in figure 8.4. Note that the input sequence is  $d = "0100"$ , which is loaded into the shift register when  $clk$  goes up with  $load = '1'$ , and then rotates, moving one position to the right at every positive clock transition.

In VHDL 2008, expressions are allowed in PORT MAP assignments. See other details in section 8.8.

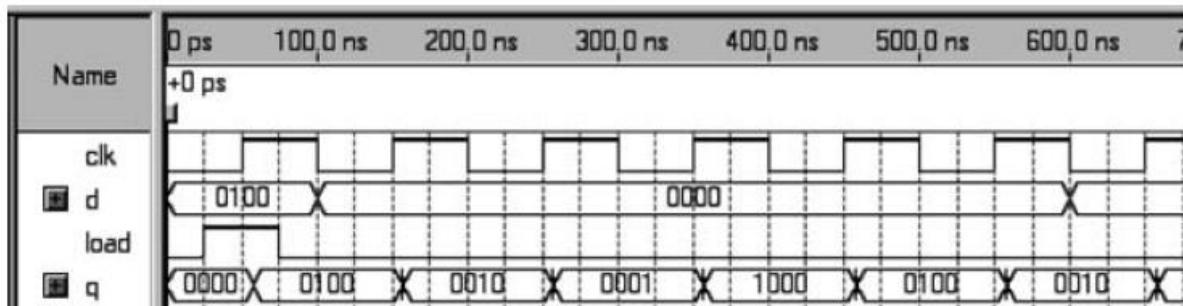


Figure 8.4

Simulation results from the circular shift register of example 8.2.

## GENERIC MAP ۴-۸

با استفاده از اعلان‌های GENERIC می‌توان به کدهای کاملاً عام (برای مثال برای استفاده در کتابخانه‌ها) دست یافت. در زمان نمونه‌سازی اجزاءی که دارای پارامترهای عام هستند، می‌توان مقادیر اولیه‌ی این پارامترها را تغییر داد. برای این کار در زمان نمونه‌سازی از GENERIC MAP استفاده می‌شود.

**Example** A component called *and\_gate* is declared below, which has a generic parameter called *inputs*, whose (optional) default value is 8. The two instantiations shown next (*a1*, *a2*) are equivalent. In the first, the mapping is *positional*, while in the second, it is *nominal*. In either case, the default value of *inputs* (8) is replaced with 16.

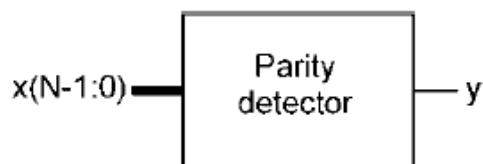
```
-----Component declaration-----
COMPONENT and_gate IS
    GENERIC (inputs: POSITIVE := 8); --see Note below
    PORT (a: IN BIT_VECTOR(1 TO inputs);
          b: OUT BIT);
END COMPONENT;
-----Component instantiation-----
a1: and_gate GENERIC MAP (16) PORT MAP (x, y);
a2: and_gate GENERIC MAP (inputs=>16) PORT MAP (a=>x, b=>y);
-----
```

Note: GENERIC does not need to be included in the component declaration above if GENERIC MAP is not used in the component instantiation and the GENERIC values specified in the main code coincide with those in the component code (this is fine even if they do not have the same name). Also, in the component code and in the component declaration, the GENERIC values can be left unspecified, but then GENERIC MAP is obviously required in order to specify them.

نحوه‌ی استفاده از GENERIC MAP در مثال زیر نشان داده شده است.

### Example 8.3: Parity Detector with COMPONENT and GENERIC MAP

The generic  $N$ -bit parity detector of figure 8.5 must produce  $y = '1'$  when the number of '1's in  $x$  is odd, or  $y = '0'$  otherwise (an implementation for this kind of circuit was seen in section 7.7). Write a VHDL code to solve this problem, where the parity detector is entered as a COMPONENT that employs a GENERIC declaration to define  $N$ . Include a GENERIC MAP declaration in your code to overwrite the original value of  $N$ . Adopt the following approach: use method 1 of section 8.3 to enter the code, with *nominal* mapping in it.



**Figure 8.5**  
Generic parity detector of example 8.3.

**Solution** A VHDL code for this circuit is presented below. First, the component (*par\_detector*) is shown, which has a generic parameter called *bits* (line 3) that defines the size of the input vector (left unspecified in this example). In the main code (*parity\_detector*), this component is declared in lines 10–14 and then used in lines 17–18, with GENERIC MAP (line 17) defining the actual value of *bits*. Note in lines 17–18 that nominal mappings were utilized.

```

1 -----The component-----
2 ENTITY par_detector IS
3     GENERIC (bits: POSITIVE);
4     PORT (input: IN BIT_VECTOR(bits-1 DOWNTO 0);
5             output: OUT BIT);
6 END par_detector;
7 -----
8 ARCHITECTURE behavior OF par_detector IS
9 BEGIN
10    PROCESS(input)
11        VARIABLE temp: BIT;
12    BEGIN
13        temp := '0';
14        FOR i IN input'RANGE LOOP
15            temp := temp XOR input(i);
16        END LOOP;
17        output <= temp;
18    END PROCESS;
19 END behavior;
20 -----

```



```

1 -----Main code-----
2 ENTITY parity_detector IS
3   GENERIC (N: POSITIVE := 8);
4   PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5         y: OUT BIT);
6 END parity_detector;
7 -----
8 ARCHITECTURE structural OF parity_detector IS
9 -----
10  COMPONENT par_detector IS
11    GENERIC (bits: POSITIVE);
12    PORT (input: IN BIT_VECTOR(bits-1 DOWNTO 0);
13          output: OUT BIT);
14  END COMPONENT;
15 -----
16 BEGIN
17   det: par_detector GENERIC MAP (bits=>N)
18     PORT MAP(input=>x, output=>y);
19 END structural;
20 -----

```

## 5-8 نمونه‌سازی اجزاء به کمک GENERATE

همان طور که اشاره شد یکی از جاهایی که میتوان از COMPONENT (جهت نمونه‌سازی) استفاده کرد، در حلقه‌های ایجاد شده توسط دستور GENERATE است. این روش به ویژه زمانی که تعداد نمونه‌سازی‌های یک جزء خاص زیاد باشد بسیار مفید است. در این حالت، نحوه استفاده به صورت زیر است. در این گرامر، نامهای gen و comp برچسب‌هایی هستند که به ترتیب برای دستور GENERATE و جزء نمونه‌سازی شده به کار گرفته شده‌اند.

```

gen: FOR i IN 0 TO max GENERATE
      comp: my_component PORT MAP (x(i), y(i), z(i));
END GENERATE gen;
-----
```

یک نمونه از کاربرد دستور GENERATE در نمونه‌سازی را قبلاً در فصل پنجم (مثال ۵-۵) دیده بودیم. در اینجا مثال دیگری آورده شده است.

### Example 8.4: Shift Register with COMPONENT and GENERATE

Figure 8.6 shows a truly generic  $M \times N$  shift register with data-load capability (Pedroni 2008) (a related circuit was described in example 8.2). The inputs are  $clk$ ,  $load$ ,  $x$ , and  $d$  (the flip-flops are synchronously loaded with the values of  $d$  when  $load = '1'$ ), while the (only) output is  $y$ . As in example 8.2, design this circuit using COMPONENT to instantiate the multiplexers and flip-flops. Adopt the following approach: use method 3 of section 8.3 to enter the code, with *positional mapping* in it.

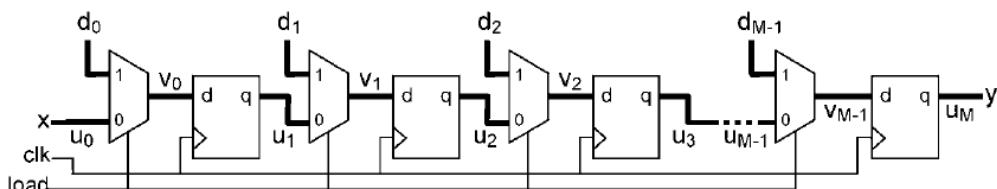


Figure 8.6  
Generic shift register of example 8.4.

**Solution** Note in figure 8.6 that the circuit has  $M$  stages with  $N$  bits each, hence totaling  $M \times N$  mux-DFF pairs. A VHDL code for this circuit is presented below. The multiplexer and flip-flop circuits are those seen in example 8.2, here compiled in separate projects (because method 3 is to be used to enter the design). As can be seen in the code below, the component declarations are in a separate package (called *my\_declarations*), so a USE clause pointing to it was included in line 2 of the main code. Note that the package contains also a TYPE declaration (*twoD*, line 3) to create a 2D array of BIT elements, which is used in the main code to specify the  $d$  input (line 9) and also the internal signals  $u$  and  $v$  (lines 14–15). The code proper contains two sections, both with the GENERATE statement. The first section of code (lines 18–21) transfers  $x$  to the first slice of  $u$  and the last slice of  $u$  to  $y$  (if one prefers, the transfer of  $y$  can be done in a separate section at the end of the code). The second section of code (lines 23–28) computes the values for the internal 2D arrays  $u$  and  $v$ .

```

1 -----Package-----
2 PACKAGE my_declarations IS
3     TYPE twoD IS ARRAY (NATURAL RANGE <>, NATURAL RANGE <>) OF BIT;
4     -----
5     COMPONENT mux IS
6         PORT (a, b, sel: IN BIT; x: OUT BIT);
7     END COMPONENT;
8     -----
9     COMPONENT flipflop IS
10        PORT (d, clk: IN BIT; q: OUT BIT);
11    END COMPONENT;
12    -----
13 END PACKAGE;
14 -----

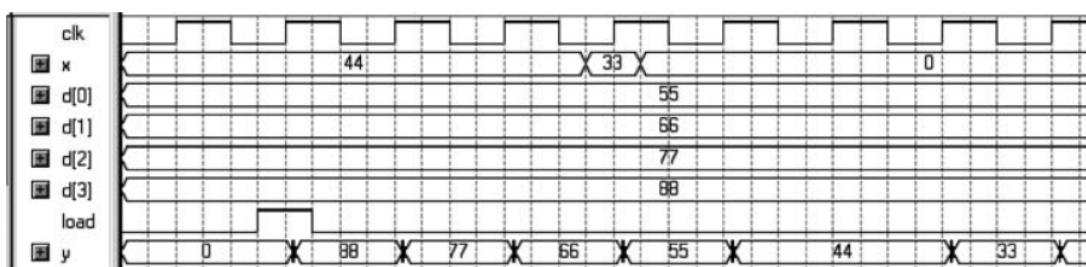
```

```

1 -----Main code-----
2 USE work.my_declarations.all;
3 -----
4 ENTITY shift_register IS
5     GENERIC (M: POSITIVE := 4;
6                 N: POSITIVE := 8);
7     PORT (clk, load: IN BIT;
8             x: IN BIT_VECTOR(N-1 DOWNTO 0);
9             d: IN twoD(0 TO M-1, N-1 DOWNTO 0);
10            y: OUT BIT_VECTOR(N-1 DOWNTO 0));
11 END ENTITY;
12 -----
13 ARCHITECTURE structural OF shift_register IS
14     SIGNAL u: twoD(0 TO M, N-1 DOWNTO 0);
15     SIGNAL v: twoD(0 TO M-1, N-1 DOWNTO 0);
16 BEGIN
17     ----Transfer x->u and u->y:-----
18     gen1: FOR i IN N-1 DOWNTO 0 GENERATE
19         u(0,i) <= x(i);
20         y(i) <= u(M,i);
21     END GENERATE gen1;
22     ----Update internal array:-----
23     gen2: FOR i IN 0 TO M-1 GENERATE
24         gen3: FOR j IN N-1 DOWNTO 0 GENERATE
25             mux1: mux PORT MAP (u(i,j), d(i,j), load, v(i,j));
26            dff1: flipflop PORT MAP (v(i,j), clk, u(i+1,j));
27         END GENERATE gen3;
28     END GENERATE gen2;
29 END ARCHITECTURE;
30 -----

```

Simulation results are displayed in figure 8.7 (for  $N = 8$  and  $M = 4$ ). Note that  $d$  is loaded into the shift register at the first positive clock transition after *load* is asserted. Observe also that each individual value of  $d$  lies in the 0-to-255 range (because  $N = 8$ ) and that a new input value (33, for example, in this simulation) reaches the output at the fourth positive clock transition (because  $M = 4$ ).



**Figure 8.7**  
Simulation results from the shift register of example 8.4.

## CONFIGURATION ۶-۸

این دستور اتصالی بین یک موجودیت (از یک سو) و معماری مورد نظرمان (از سوی دیگر) برقرار می‌کند. این قابلیت در پروژه‌هایی شامل چند معماری (به ازاء یک موجودیت مشخص) و یا در طراحی‌های سلسله‌مراتبی مفید است. گرامر ساده شده‌ای مربوط به هریک از این دو حالت در زیر نشان داده شده است.

Direct binding:

```
CONFIGURATION config_name OF entity_name IS
    FOR arch_name
    END FOR;
END [CONFIGURATION] [config_name];
```

Binding in component instantiations:

```
CONFIGURATION config_name OF entity_name IS
    FOR arch_name
        FOR label: component_name
        --or FOR OTHERS/ALL: component_name
            USE ENTITY entity_name [(arch_name)];
        END FOR;
    END FOR;
END [CONFIGURATION] [config_name];
```

در حالت یا پیکره‌بندی اول که بین یک موجودیت و یک معماری رابطه‌ای مستقیم ایجاد می‌شود ابتدا کلمه‌ی CONFIGURATION و سپس اقلام زیر ظاهر می‌شوند:

- نامی برای مرحله‌ی پیکره‌بندی،
- نام موجودیت (نام پروژه) که قرار است معماری به آن متصل شود،
- در خط دوم، کلمه‌ی FOR نام معماری مورد نظرمان را مشخص می‌کند. کد مربوط به پیکره‌بندی باید خارج از هر موجودیت یا معماری قرار داده شود.

**Example** The code below contains an entity (called *test*) and two architectures (*arch1* and *arch2*). The configuration declaration (called *config1*) defines the binding as *test-arch1* (that is, *arch1* is to be used with *test*).

```
-----
ENTITY test ...
END test;
-----
ARCHITECTURE arch1 ...
END arch1;
-----
ARCHITECTURE arch2 ...
END arch2;
-----
CONFIGURATION config1 OF test IS
  FOR arch1
    END FOR;
  END CONFIGURATION;
```

گرامر یا پیکربندی دوم امکان اتصال پیچیده‌تری را فراهم می‌کند. برای مثال در اینجا یک جزء را می‌توان با هر نام دلخواهی در داخل کد خود اعلان یا نمونه‌سازی کنیم.

**Example** The code below is from example 8.3. Note, however, that the name of the component in the component declaration (lines 10–14) and component instantiation (line 17) was changed to *detector*, which does not correspond to any entity available in our design. However, a configuration declaration was also included in the code (lines 21–27), saying the following: its name is *my\_config* and it relates to the entity *parity\_detector* (line 21); the architecture named *structural* (line 22) of that entity must employ, for the component instantiation labeled *det* of component *detector* (line 23), the entity *par\_detector* with architecture *behavior* (line 24), both available in the current *work* library (project's directory).

```

1 -----The component:-----
2 ENTITY par_detector IS
.. ... (see Example 8.3)
19 END behavior;
20 -----

1 -----Main code:-----
2 ENTITY parity_detector IS
3   GENERIC (N: POSITIVE := 8);
4     PORT (x: IN BIT_VECTOR(N-1 DOWNTO 0);
5           y: OUT BIT);
6   END parity_detector;
7 -----

8 ARCHITECTURE structural OF parity_detector IS
9   -----
10  COMPONENT detector IS
11    GENERIC (bits: POSITIVE);
12      PORT (input: IN BIT_VECTOR(bits-1 DOWNTO 0);
13          output: OUT BIT);
14  END COMPONENT;
15  -----

16 BEGIN
17   det: detector GENERIC MAP (N) PORT MAP(x, y);
18 END structural;
19 -----

20 -----
21 CONFIGURATION my_config OF parity_detector IS
22   FOR structural
23     FOR det: detector
24       USE ENTITY work.par_detector(behavior);
25     END FOR;
26   END FOR;
27 END my_config;
28 -----

```

It is important to mention that in regular codes (with just one architecture for each entity, and with component instantiations fully specified using the syntax of section 8.3) there is no need to use CONFIGURATION.



## BLOCK ۷-۸

دستور BLOCK یک دستور همزمان است که هدف اصلی آن فراهم کردن راهی جهت بخش‌بندی کد (لذا انجام طراحی سطح سیستمی) می‌باشد. از BLOCK در داخل بدنهٔ معماری جهت گروه‌بندی بخش‌های مرتبط با هم از کد استفاده می‌شود تا بدین ترتیب خوانایی کل کد بیشتر و مدیریت آن راحت‌تر شود (این امر در طراحی‌های بزرگ حائز اهمیت است).

دستور BLOCK با توجه به همزمان بودن آن، قابل استفاده در یک کد ترتیبی نمی‌باشد اما داخل آن می‌توان از فرآیند (PROCESS) استفاده نمود زیرا یک فرآیند از بیرون یک دستور همزمان محاسب می‌شود. گرامر ساده‌شده‌ای از دستور BLOCK در زیر نشان داده شده است.

```
label: BLOCK [ (guard_expression) ] [ IS ]
        [ declarative_part ]
        BEGIN
        concurrent_statements_part
        END BLOCK [label];
```

مطابق با گرامر فوق، استفاده از برجسب ضروری است اما استفاده از عبارت محافظه<sup>۱</sup> (بعد از توضیح داده خواهد شد) اختیاری است. بخش اعلان می‌تواند شامل اعلان‌های زیر باشد:

- GENERIC
- GENERIC MAP
- PORT
- PORT MAP

علاوه بر اقلام فوق، تمام اعلان‌هایی که داخل بخش اعلان یک معماری مجاز هستند نیز در اینجا قابل استفاده می‌باشند. دستورات بلوکی را می‌توان داخل یکدیگر به صورت تو در تو استفاده کرد تا بتوان ساختارهای پیچیده‌ای از بلوک‌ها ایجاد کرد. مثال زیر ساده‌ترین کاربرد ممکن از دستور BLOCK را نشان می‌دهد (در اینجا تنها یک بخش‌بندی ساده‌ی کد است).

```
-----  
ARCHITECTURE example OF ...  
BEGIN  
    ...  
    controller: BLOCK  
    BEGIN  
        ...  
    END BLOCK controller;  
    ...  
END example;
```

یک بلوک محافظت شده<sup>۱</sup> بلوکی است که در آن از عبارت محافظ استفاده شود. در این بلوک، دستورات موجود در بلوک تنها زمانی اجرا می‌شوند که مقدار عبارت محافظ TRUE باشد. عبارت محافظ موجب ایجاد یک مکانیزم سخت‌افزاری جهت اتصال یا قطع اتصال می‌شود.

---

<sup>۱</sup> Guard Expression

مثال ۸-۵ نحوه تاثیر عبارت محافظت می‌داند. توجه کنید که این مثال تنها یک مثال نمایشی بوده و روش طراحی توصیه شده‌ای برای لچ‌ها یا دیگر مدارات نمی‌باشد زیرا هدف اصلی عبارت محافظت اتصال/عدم اتصال محرک‌ها است.

#### Example 8.5: Latch Implemented with a Guarded BLOCK

The code below implements a D-type latch. The whole architecture body is included in a block called *blk*, which contains the guard expression *clk = '1'* (line 12). When this condition is fulfilled, the block is evaluated. Because the output is defined as *q = GUARDED d* (line 14), *q* receives the value of *d* when the guard expression is TRUE. In other words, *q = d* results while *clk = '1'*, which is the logic equation for a D-type latch.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY latch IS
6     PORT (d, clk: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8 END ENTITY;
9 -----
10 ARCHITECTURE block_latch OF latch IS
11 BEGIN
12     blk: BLOCK (clk='1')
13     BEGIN
14         q <= GUARDED d;
15     END BLOCK blk;
16 END ARCHITECTURE;
17 -----
```




---

<sup>1</sup> Guarded Block

## 8.8 VHDL 2008

With respect to the material covered in this chapter, the main additions specified in VHDL 2008 are those listed below.

- 1) The declarative part of a PACKAGE can contain these additional kinds of declarations: subprogram instantiation declaration, package declaration, package instantiation declaration, and PSL declaration.
- 2) The use of GENERIC (section 2.6) in the PACKAGE header is allowed, as indicated in the simplified syntax below. An example is shown subsequently.

```
PACKAGE package_name IS
  [GENERIC (generic_list);]
  declarative_part
END [PACKAGE] [package_name];
```

```
PACKAGE generic_type IS
  GENERIC (CONSTANT words: NATURAL;
            TYPE: word_type);
  TYPE gen_type IS ARRAY 1 TO words OF word_type;
END PACKAGE;
```

- 3) A package with a generic list is called an *uninstantiated package*, which must be instantiated with a *package instantiation* declaration, shown in the simplified syntax below. An example of instantiation for the package *generic\_type* above is also presented below.

```
PACKAGE package_name IS NEW uninstantiated_package_name
  GENERIC MAP (instantiation_list);
```

```
LIBRARY ieee;
USE ieee.std_logic_1164;
PACKAGE memory_array IS NEW work.generic_type
  GENERIC MAP (words => 256, word_type => STD_LOGIC_VECTOR(15 DOWNTO 0);
```

- 4) Expressions are allowed in PORT MAP assignments, as in the example below.

```
cir: my_circuit PORT MAP(inp => a AND b, outp => c);
```

## فصل نهم

### تابع و روال

برخلاف فرآیندها که برای بدنه‌ی معماری طراحی شده‌اند، زیربرنامه‌ها (تابع و روال‌ها) را می‌توان در

- PACKAGE
- ENTITY
- ARCHITECTURE
- PROCESS

ایجاد کرد. البته متداول‌ترین محل برای ایجاد زیربرنامه‌ها، بسته‌ها هستند لذا زیربرنامه‌ها را (به همراه بسته‌ها و اجزاء) جزو واحدهای سطح سیستمی می‌شناسیم.

#### ۲-۹ دستور ASSERT

دستور ASSERT برای چک کردن ورودی‌های یک زیربرنامه مفید است. این دستور در شبیه‌سازی نیز مفید است. این دستور در هر دو نوع کد همزمان و ترتیبی قابل استفاده است. هدف این دستور ایجاد مدار نیست بلکه چک کردن برخی شرایط و ملزومات در حین سنتز یا شبیه‌سازی است. گرامر آن به صورت زیر است.

```
[label:] ASSERT boolean_expression
      [REPORT string_expression]
      [SEVERITY severity_level];
```

در گرامر فوق عبارت رشته‌ای می‌تواند شامل یک ثابت یا سیگنالی از نوع STRING باشد. از عملگر تجمعی (&) در اینجا می‌توان استفاده نمود.

**Example** Say that *s* is a string whose value is *idle*. Then the statement below

```
REPORT "Attention: s=" & s & "!"
```

will cause the message *Attention: s = idle!* to be printed on the screen. Other cases will be presented ahead.

سطح شدت (SEVERITY) می‌تواند یکی از مقادیر زیر را به خود بگیرد:

- NOTE : برای انتقال اطلاعات از کامپایلر/ شبیه‌ساز
- WARNING : برای اطلاع دادن این که امری غیرعادی رخ داده است
- ERROR : برای اطلاع دادن این که شرایط غیرعادی جدی رخ داده است
- FAILURE : یک شرط کاملاً غیرقابل قبول رخ داده است

پیام در صورتی که شرط استفاده شده نادرست (FALSE) باشد نمایش داده می‌شود.

**Example** Say that a certain function receives two vectors, called *a* and *b*, which must have the same size. Then the following test could be done (the use of parentheses is optional):

```
ASSERT (a'LENGTH=b'LENGTH)
REPORT "Signals a and b do not have the same length!"
SEVERITY FAILURE;
```

گزینه‌ی فوق که متداول‌ترین گزینه است، ASSERT شرطی نامیده می‌شود. نسخه‌ی غیرشرطی این دستور نیز به صورت زیر وجود دارد:

```
[label:] ASSERT FALSE
[REPORT string_expression]
[SEVERITY severity_level];
```

دستور فوق موجب الزام نمایش پیام می‌شود. این حالت زمانی که بخواهیم بررسی کنیم آیا کامپایلر به نقطه‌ی خاصی رسیده است یا خیر مفید است (در این حالت معمولاً از مقدار NOTE برای سطح شدت استفاده می‌شود). این دستور با دستور IF قابل ترکیب است تا شرطی را بررسی کرده و مجددًا حالت شرطی به خود بگیرد (مثال ۳-۹).

برای ساختن حالتهای پیچیده‌تری از دستور ASSERT می‌توان از مختصه‌ی T'IMAGE(X) استفاده نمود (در بخش ۴-۴ معرفی شد). این مختصه برای مقدار X که از نوع T است، نوع نمایش رشته (STRING) را تعیین می‌کند. این مختصه به این دلیل مفید است که قسمت REPORT از دستور ASSERT فقط رشته‌ها را قبول می‌کند. توجه داشته باشید که T محدود به انواع عددی، شمارش شده، و فیزیکی می‌باشد (در مورد انواع فیزیکی، نوع TIME متداول‌ترین

نوع می باشد). مثالی در زیر نشان داده شده است که مربوط به کدی برای شبیه سازی است (فصل .(۱۰)

```
-----
USE ieee.std_logic_unsigned.all;
...
SIGNAL x, y: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL n: INTEGER RANGE 0 TO 255;
SIGNAL t: TIME RANGE 0ns TO 200ns;
...
ASSERT (x=y AND n=ref)
  REPORT "Mismatch at t=" & TIME'IMAGE(t) &
    " (for n=" & INTEGER'IMAGE(n) &
    ", x=" & INTEGER'IMAGE(conv_integer(x)) &
    ", y=" & INTEGER'IMAGE(conv_integer(y)) & "."
  SEVERITY FAILURE;
...
-----
```

In the code above, *x* and *y* are STD\_LOGIC\_VECTOR signals, *n* is INTEGER, and *t* is TIME. For them to be used in the REPORT section of ASSERT, they must first be converted to the type STRING. Such conversion can be made within the REPORT statement itself, using the TIMAGE(X) attribute. In the first line, a direct conversion for *t*, from TIME to STRING, is made. In the second line, another direct conversion is made, for *n*, now from INTEGER (a numeric type) to STRING. However, in the cases of *x* and *y*, a direct conversion is not possible because STD\_LOGIC\_VECTOR is not allowed in the TIMAGE(X) attribute. Consequently, a conversion to INTEGER is made first, using the function *conv\_integer()*, available in the *std\_logic\_unsigned* package (see the package declaration at the beginning of the code), so such integers can then be converted to STRING. If anyone of the two boolean conditions is not satisfied, the following message will be issued (assuming *x* = "0011", *y* = "1111", *n* = 8, and *t* = 5ns): *Mismatch at t = 5000 ps (for n = 8, x = 3, y = 15)*.

The use of ASSERT will be illustrated in the examples with FUNCTION in section 9.3. It will also be used in the next chapter, which deals with simulation.

In VHDL 2008, the TO\_STRING type-conversion function was introduced, which can be used instead of the TIMAGE(X) attribute. The advantage of TO\_STRING is that it supports a wider set of data types: BOOLEAN, BIT, BIT\_VECTOR, INTEGER, NATURAL, POSITIVE, CHARACTER, STD\_(U)LOGIC\_VECTOR, (UN)SIGNED, REAL, TIME, SFIXED, UFIXED, and FLOAT.

### ۳-۹ تابع

تابع بخشی از کدهای ترتیبی در VHDL است که هدف آن امکان ایجاد راهکارهایی برای مسائل شناخته شده که معمولاً به آنها برخورد می‌کنیم و ذخیره‌ی آنها در کتابخانه است. برای مثال تبدیل نوع و عملیات منطقی و حسابی را می‌توان نام برد.

تابع مشابه با فرآیند است. این تشابه از این جهت است که هر دو ترتیبی بوده و فقط قادر به استفاده از دستورات ترتیبی (IF، LOOP، WAIT و CASE) و البته حالت‌های توسعه یافته مربوط به دستورات WHEN و SELECT در بخش ۷-۹ VHDL 2008 را نیز ملاحظه کنید) هستند. شباهت دیگر این است که همان اقلامی که در بخش اعلان یک فرآیند می‌توانیم اعلان کنیم در اینجا نیز می‌توان آنها را اعلان کرد (بخش ۳-۶؛ در اینجا هم اعلان سیگنال مجاز نیست). گرامر ساده‌ای از ایجاد یک تابع در زیر نشان داده شده است:

```
[PURE | IMPURE] FUNCTION function_name [(input_list)]
RETURN return_value_type IS
[declarative_part]
BEGIN
statement_part
[label:] RETURN expression;
END [FUNCTION] [function_name];
```

گرامر فوق با اعلان PURE و یا IMPURE شروع می‌شود (بعداً توضیح داده خواهد شد). اگر آن را اعلام نکنید، پیش‌فرض PURE استفاده خواهد شد.

لیست ورودی می‌تواند شامل هر تعداد پارامتر (حتی تعداد صفر) باشد که همگی دارای حالت IN (یعنی همه‌ی پارامترها «ورودی» تابع) هستند. این لیست فقط می‌تواند شامل اشیاء VARIABLE (پیش‌فرض است)، SIGNAL و CONSTANT (شیء FILE غیرمجاز است) بوده و به صورت زیر اعلان شوند:

```
[CONSTANT] constant_name: constant_type;
SIGNAL signal_name: signal_type;
```

یک تابع همیشه مستقل از تعداد پارامترهای ورودی، تنها یک پارامتر خروجی را برمی‌گرداند. نوع این تابع باید بعد از کلمه‌ی RETURN در سرآیند تابع مشخص شود.

**Example** The function below, named *positive\_edge*, receives a signal called *s*, returning TRUE when a positive transition occurs on *s*.

```
-----  
FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS  
BEGIN  
    RETURN (s'EVENT AND s='1');  
END FUNCTION positive_edge;  
-----
```

در محلهای زیر می‌توان یک تابع را (به کمک گرامر فوق) ساخت:

- PACKAGE
- ENTITY
- ARCHITECTURE
- PROCESS
- BLOCK
- هر زیربرنامه‌ی دیگر

از فصل ۸ به خاطر داریم که هرگاه داخل یک بسته، یک زیربرنامه ساخته شود آن‌گاه استفاده از بدنی بسته (PACKAGE BODY) ضروری است (در این حالت، اعلان زیربرنامه در قسمت PACKAGE و بدنی زیربرنامه در PACKAGE BODY قرار داده می‌شود). این مطلب در زیر نشان داده شده است:

```
-----Package:  
PACKAGE my_subprograms IS  
    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN;  
END PACKAGE;  
-----Package body:  
PACKAGE BODY my_subprograms IS  
    FUNCTION positive_edge (SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS  
    BEGIN  
        RETURN (s'EVENT AND s='1');  
    END FUNCTION positive_edge;  
END PACKAGE BODY;  
-----
```

### فراخوانی تابع

اساساً یک تابع را می‌توان در هر محلی فراخوانی (اعم از کدهای ترتیبی و همزمان، درون زیربرنامه‌ها، داخل GENERATE، وغیره) نمود. همان طور که در مثالها نشان داده شده است، فراخوانی یک تابع همیشه باید بخشی از یک عبارت باشد.

**Example** Say that the function *positive\_edge* seen above is part of our design. Then the first line below, which contains a call to that function, is equivalent to the second line.

```
IF positive_edge(clk) THEN ...
IF clk'EVENT AND clk='1' THEN...
```

### نگاشت مکانی در مقابل نگاشت نامی

مشابه با PORT MAP در حین نمونه‌سازی یک جزء، در اینجا هم باید یک نگاشت بین فراخوانی تابع و اعلان پارامترها (در حین تعریف تابع) انجام شود. این نگاشت می‌تواند نامی و یا مکانی باشد.

**Example** Three equivalent function calls are shown below.

```
-----Function declaration:-----
FUNCTION my_function (SIGNAL a, b: BIT) RETURN BIT;
-----Equivalent function calls:-----
y <= my_function (x1, x2);           --positional mapping
y <= my_function (a=>x1, b=>x2);   --nominal mapping
y <= my_function (b=>x2, a=>x1);   --nominal mapping
```

### توابع خالص (Pure) در مقابل توابع ناخالص (Impure)

یک تابع را خالص می‌نامیم هرگاه تنها قادر به تصحیح و تغییر متغیرهای خود باشد. بنابراین اگر تابعی را با پارامترهای مشخصی بارها و بارها فراخوانی کنیم همان نتایج یکسان قبل به دست خواهد آمد. اما تابع ناخالص علاوه بر امکان تغییر متغیرهای خود، قادر به تصحیح سیگنال‌ها و متغیرهای مربوط به معماری، فرآیند، و یا زیربرنامه‌ای که تابع مزبور در آن اعلان شده، می‌باشد. بنابراین، در این حالت اگر تابع را با پارامترهای مشخصی چندبار در زمان‌های مختلف فراخوانی کنیم هربار ممکن است نتایج متفاوتی در برداشته باشد.

در VHDL 2008 امکان استفاده از یک لیست GENERIC در درون یک تابع فراهم شده است. جزئیات بیشتر در بخش ۷-۹ آمده است.

در ادامه چند مثال کامل مربوط به ساخت و استفاده از توابع آورده شده است.

**Example 9.1: FUNCTION *max* in an ARCHITECTURE**

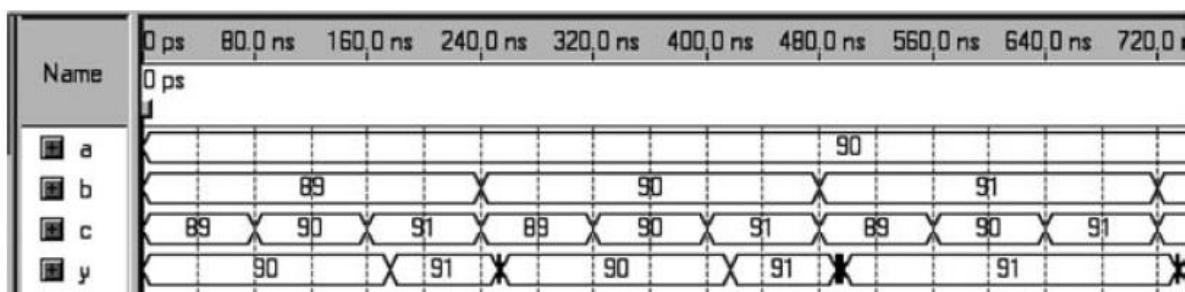
Write a function that returns the largest of three integers. Assume that all in/out signals are required to have the same range (use the ASSERT statement to check this). Construct the function directly in the ARCHITECTURE of the main code (in its declarative part).

**Solution** A code for this circuit is shown below, under the title *comparator* (line 2). The FUNCTION, called *max*, is in lines 8–23, located in the declarative part of the ARCHITECTURE. Note that it is divided into two parts. The first part (lines 11–14) contains an ASSERT statement that checks whether the sizes of *a*, *b*, *c*, and *y* are all equal, printing the message “Signal sizes are not all equal!” on the screen if the result of the assertion test is FALSE, which also causes the compilation to stop (due to the chosen severity level, FAILURE). The second part (lines 16–22) determines the largest of the three inputs. A function call is made in the main code (line 25), which passes the largest input to the output (*y*). Positional versus nominal mapping is illustrated in lines 25–26. Simulation results are depicted in figure 9.1.

```

1 -----
2 ENTITY comparator IS
3     PORT (a, b, c: IN INTEGER RANGE 0 TO 255;
4             y: OUT INTEGER RANGE 0 TO 255);
5 END ENTITY;
6 -----
7 ARCHITECTURE comparator OF comparator IS
8     FUNCTION max (in1, in2, in3: INTEGER) RETURN INTEGER IS
9     BEGIN
10         -----Check in-out signals-----
11         ASSERT (y'LEFT=a'LEFT AND y'LEFT=b'LEFT AND y'LEFT=c'LEFT
12             AND y'RIGHT=a'RIGHT AND y'RIGHT=b'RIGHT AND y'RIGHT=c'RIGHT)
13             REPORT "Signal sizes are not all equal!"
14             SEVERITY FAILURE;
15         -----Find maximum-----
16         IF (in1>=in2 AND in1>=in3) THEN
17             RETURN in1;
18         ELSIF (in2>=in1 AND in2>=in3) THEN
19             RETURN in2;
20         ELSE
21             RETURN in3;
22         END IF;
23     END FUNCTION;
24 BEGIN
25     y <= max(a, b, c); --positional mapping
26     --y <= max(in1=>a, in2=>b, in3=>c); --nominal mapping
27 END ARCHITECTURE;
28 -----

```



**Figure 9.1**

Simulation results from the code of example 9.1.

### Example 9.2: FUNCTION *order\_and\_fill* in a PACKAGE

Write a function that reorganizes a binary word, such that the indexing is always descending and ending in zero, regardless of the original specification (for example:  $a(5:2) \rightarrow b(3:0)$ ,  $a(1:4) \rightarrow b(3:0)$ ,  $a(3:0) \rightarrow b(3:0)$ , etc.). Moreover, the vector must be filled with zeros (on the left) until a predefined size is attained. Include the ASSERT statement in your solution to assure that the size of the input word is not bigger than the size wanted for the final vector (after filling). Construct your function in a PACKAGE (most common option).

**Solution** The FUNCTION, called *order\_and\_fill*, is shown below, constructed in a PACKAGE called *my\_package*. Recall that when a subprogram is constructed in a package, only the declaration goes in the package, because the subprogram body must go in the PACKAGE BODY. That can be seen below, with the function declaration in lines 6–7 and the function body in lines 11–36. The function receives two parameters, called *input* (of type UNSIGNED, which is the vector to be reorganized and filled) and *bits* (of type NATURAL, which is the size wanted for *input* after reorganization and filling). Note that the function body (in the package body) is broken into three parts. The first part (lines 17–19) contains an ASSERT statement that checks whether the size of *input* is not larger than *bits*, printing the message “Improper input size!” on the screen if the result of the assertion test is FALSE, which also causes the compilation to stop (due to the chosen severity level, FAILURE). The second part (lines 21–27) reorders the vector, such that its indexing always becomes “*input\_length – 1 DOWNTO 0*”. Finally, the third part (lines 29–34) fills the vector with zeros (on the left) to attain a final size equal to *bits*. The result is returned to the calling expression in line 35.

An example of application is also included below (main code). The function is used (called) in line 14, causing the input vector *x* to be reorganized and filled with zeros, producing an output *y* with a total of *size* bits.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.numeric_std.all;
4 -----
5 PACKAGE my_package IS
6     FUNCTION order_and_fill (input: UNSIGNED; bits: NATURAL)
7         RETURN UNSIGNED;
8 END PACKAGE;
9 -----
10 PACKAGE BODY my_package IS
11     FUNCTION order_and_fill (input: UNSIGNED; bits: NATURAL)
12         RETURN UNSIGNED IS
13         VARIABLE a: UNSIGNED(input'LENGTH-1 DOWNTO 0);
14         VARIABLE result: UNSIGNED(bits-1 DOWNTO 0);
15     BEGIN
16         -----Check input size:-----
17         ASSERT (input'LENGTH <= bits)
18             REPORT "Improper input size!"
19             SEVERITY FAILURE;
20         -----Organize input:-----
21         IF (input'LEFT>input'RIGHT) THEN
22             a := input;
23         ELSE
24             FOR i IN a'RANGE LOOP
25                 a(i) := input(input'LEFT + i);
26             END LOOP;
27         END IF;
28         -----Fill with zeros:-----
29         IF (a'LENGTH < bits) THEN
30             result(bits-1 DOWNTO a'LENGTH) := (OTHERS => '0');
31             result(a'LENGTH-1 DOWNTO 0) := a;
32         ELSE
33             result:=a;
34         END IF;
35         RETURN result;
36     END FUNCTION;
37 END PACKAGE BODY;
38 -----
1 -----Main code:-----
2 LIBRARY ieee;
3 USE ieee.numeric_std.all;
4 USE work.my_package.all;
5 -----

```

```

6 ENTITY organizer IS
7     GENERIC (size: NATURAL := 5);
8     PORT (x: IN UNSIGNED(2 TO 5);
9             y: OUT UNSIGNED(size-1 DOWNTO 0));
10    END ENTITY;
11  -----
12 ARCHITECTURE organizer OF organizer IS
13 BEGIN
14     y <= order_and_fill(x, size);
15 END ARCHITECTURE;
16  -----

```

### Example 9.3: FUNCTION *slv\_to\_integer* in an ENTITY

Write a FUNCTION that converts a signal of type STD\_LOGIC\_VECTOR to type INTEGER. Include an ASSERT statement in your code to ensure that no symbols other than '0', 'L' (both synthesized as '0'), '1' or 'H' (both synthesized as '1') are present at the input. This time, construct the function directly in the code's ENTITY.

**Solution** A VHDL code that solves this problem is shown below. The FUNCTION, called *slv\_to\_integer*, was constructed in the code's ENTITY (after PORT, lines 7–24). An ALIAS (section 4.8), called *ss*, was used in line 9 to normalize the range of *s* to "1 TO *s*'LENGTH", which then simplifies the writing of the function. The ASSERT statement in lines 17–20 checks if only the proper values occur at the input (if any of the other five STD\_LOGIC values occurs, an error message is issued and the compilation is interrupted). Note that an unconditional ASSERT is employed in lines 18–20, but because it is associated to the IF statement in line 17, the overall test is still conditional.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY ...
6   PORT (....)
7   FUNCTION slv_to_integer (SIGNAL s: STD_LOGIC_VECTOR)
8     RETURN INTEGER IS
9     ALIAS ss: STD_LOGIC_VECTOR(1 TO s'LENGTH) IS s;
10    VARIABLE result: INTEGER RANGE 0 TO 2**s'LENGTH-1;
11   BEGIN
12     result := 0;
13     FOR i IN 1 TO s'LENGTH LOOP
14       result := result * 2;
15       IF (ss(i)='1' OR ss(i)='H') THEN
16         result := result + 1;
17       ELSIF (ss(i)='0' AND ss(i)='L') THEN
18         ASSERT FALSE
19           REPORT "There is an invalid input!"
20           SEVERITY FAILURE;
21       END IF;
22     END LOOP;
23     RETURN result;
24   END FUNCTION slv_to_integer;
25 -----
26 ARCHITECTURE ...
27 -----

```

#### ۴-۹ روال

هدف، نحوه‌ی ایجاد و نحوه‌ی استفاده از روال‌ها مشابه با توابع است با این تفاوت اساسی که روال‌ها قادر به بازگرداندن بیش از یک مقدار خروجی هستند. گرامری از نحوه‌ی ایجاد یک روال در زیر نشان داده شده است:

```

PROCEDURE procedure_name (input_output_list) IS
    [declarative_part]
BEGIN
    statement_part
END [PROCEDURE] [procedure_name]

```

لیست ورودی-خروجی می‌تواند شامل VARIABLE و CONSTANT و SIGNAL باشد. حالت‌های آنها نیز می‌توانند شامل IN، OUT و inout باشند. اگر از حالت IN استفاده شود، از CONSTANT به طور پیش فرض استفاده می‌شود اما برای حالت‌های OUT و inout از VARIABLE به طور پیش فرض استفاده می‌شود. اعلان SIGNAL، CONSTANT و VARIABLE به صورت زیر است:

```

CONSTANT constant_name: mode constant_type;
SIGNAL signal_name: mode signal_type;
VARIABLE variable_name: mode variable_type;

```

توابع و روال‌ها کدهای ترتیبی بوده و بنابراین فقط دستورات ترتیبی در داخل آنها مجاز می‌باشند (البته حالتهای توسعه یافته‌ی WHEN و SELECT در VHDL 2008 را در بخش ۷-۹ نیز ملاحظه کنید). هر دو را به یک شکل می‌توان در یک بسته (که متدائلترین محل است) و بدنه‌ی آن ایجاد و استفاده کرد.

مشابه با توابع، فراخوانی روال‌ها را در هر محلی می‌توان انجام داد. اما برخلاف توابه که فراخوانی آنها باید بخشی از یک عبارت باشد، فراخوانی یک روال خود یک دستور جدا و مستقل است. مثالهایی از فراخوانی چند روال در زیر نشان داده شده است:

```

-----
sort (a1, a2, a3, b1, b2, b3);
-----
divide (dividend, divisor, quotient, remainder);
-----
IF (x>y) THEN get_max (x1, x2, x3, x4, y1, y2);
-----
```

در روال‌ها هم باید نگاشت به یکی از دو صورت نامی و مکانی انجام شود.

In VHDL 2008, a GENERIC list is allowed in a PROCEDURE and the OUT mode can be read by the containing PROCEDURE.

#### Example 9.4: PROCEDURE *min\_max* in a PACKAGE

A diagram for a 3-input, 2-output circuit (called *min\_max*) is depicted in the top-left corner of figure 9.2. The circuit must detect the smallest and largest values among *a*, *b*, and *c*, and assign them to *min* and *max*, respectively. Write a PROCEDURE capable of implementing such a functionality. Construct it in a PACKAGE, then write a main code with a call to this procedure to test it (with negative numbers allowed).

**Solution** A VHDL code for this problem is shown below. The procedure (called *min\_max*) is declared in lines 3–4 of a PACKAGE (called *my\_package*) and its body (lines 8–30) is constructed in the corresponding PACKAGE BODY. The code is based on the flowchart included in figure 9.2. A call to the PROCEDURE is made in line 11 of the main code (note that it is a statement on its own).

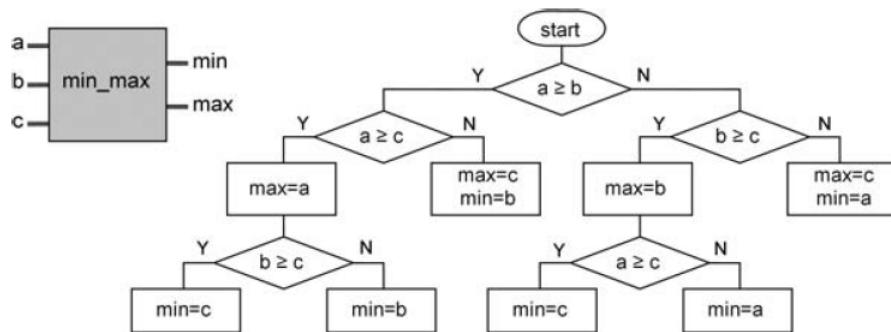


Figure 9.2

Top-level diagram and respective flowchart for the procedure *min\_max* of example 9.4.



```

1 -----Package-----
2 PACKAGE my_package IS
3     PROCEDURE min_max (SIGNAL a, b, c: IN INTEGER;
4                         SIGNAL min, max: OUT INTEGER);
5 END PACKAGE;
6 -----
7 PACKAGE BODY my_package IS
8     PROCEDURE min_max (SIGNAL a, b, c: IN INTEGER RANGE 0 TO 255;
9                         SIGNAL min, max: OUT INTEGER RANGE 0 TO 255) IS
10    BEGIN
11        IF (a>=b) THEN
12            IF (a>=c) THEN max <= a;
13                IF (b>=c) THEN min <= c;
14                ELSE min <= b;
15                END IF;
16            ELSE
17                max <= c;
18                min <= b;
19                END IF;
20            ELSE
21                IF (b>=c) THEN max <= b;
22                    IF (a>=c) THEN min <= c;
23                    ELSE min <= a;
24                    END IF;
25                ELSE
26                    max <= c;
27                    min <= a;
28                    END IF;
29                END IF;
30        END PACKAGE BODY;
31 END my_package;
32 -----Main code-----
3 USE work.my_package.all;
4 -----
5 ENTITY comparator IS
6     PORT (a, b, c: IN INTEGER RANGE -256 TO 255;
7             min, max: OUT INTEGER RANGE -256 TO 255);
8 END ENTITY;
9 -----
10 ARCHITECTURE comparator OF comparator IS
11 BEGIN
12     min_max(a, b, c, min, max);
13 END ARCHITECTURE;
14 -----

```

## ۵-۹ جمع‌بندی تابع و روال

- Location: Both can be located in the declarative part of an ENTITY, ARCHITECTURE, or of another subprogram. The most usual location, however, is in a PACKAGE (in which case a PACKAGE BODY is also needed).
- Call: Both can be called basically anywhere (in sequential as well concurrent code, in subprograms, etc.). However, the first is called as part of an expression, while the second is a statement on its own.
- Statements: Only sequential statements are allowed (IF, WAIT, LOOP, CASE).
- In/Out parameters for functions: Any number of input parameters are allowed, but only (exactly) one value is returned. The in/out objects can only be CONSTANT (default) or SIGNAL.
- In/Out parameters for procedures: Any number of in/out parameters are allowed, which can be CONSTANT (default for input), SIGNAL, or VARIABLE (default for output).

## ۶-۹ سربارگذاری

منظور از یک عملگر سربارگذاری شده (همان طور که در بخش ۳-۱۷ دیده شد) عملگری است که بیش از یک وظیفه‌ی ورودی-خروجی برای آن وجود دارد. برای مثال در بسته‌ی std numeric \_ (ضمیمه‌ی J) شش نسخه از '+' به صورت زیر وجود دارد (منظور از L و R عملوندهای به ترتیب چپ و راست می‌باشد).

```
FUNCTION "+" (L, R: UNSIGNED) RETURN UNSIGNED;
FUNCTION "+" (L, R: SIGNED) RETURN SIGNED;
FUNCTION "+" (L: UNSIGNED; R: NATURAL) RETURN UNSIGNED;
FUNCTION "+" (L: NATURAL; R: UNSIGNED) RETURN UNSIGNED;
FUNCTION "+" (L: INTEGER; R: SIGNED) RETURN SIGNED;
FUNCTION "+" (L: SIGNED; R: INTEGER) RETURN SIGNED;
```

بسته به نوع داده‌هایی که عملگر روی آنها کار می‌کند، کامپایلر تابع مناسب را انتخاب و استفاده می‌کند.

در بخش ۳-۴ مثالی از سربارگذاری (توسط کاربر) نشان داده شد. در اینجا مثال ۵-۹ جزئیات بیشتری را نشان می‌دهد.

**Example 9.5: Overloaded "+" Operator**

According to section 4.2 and figure 4.1, arithmetic operators were not originally defined for the type STD\_LOGIC\_VECTOR (see package *std\_logic\_1164* in appendix I). Write a FUNCTION that further overloads the "+" (addition) operator, such that STD\_LOGIC\_VECTOR inputs are also supported, returning a value of the same type.

**Solution** The requested "+" function is shown below. Its declaration is in line 6 of a PACKAGE, and its body is in lines 10–22 of the corresponding PACKAGE BODY. ALIAS declarations (section 4.8) were employed in lines 11–12 to normalize the range of *a* and *b* to "1 TO *a*'LENGTH" and "1 TO *b*'LENGTH", which helps write the function. Notice that, contrary to example 9.3, invalid STD\_LOGIC inputs are not tested in this example. Note also that the logic operations in lines 17–19 require that the vector sizes be equal, which is not tested either (see exercise 9.1). An example of call to this FUNCTION is illustrated in line 14 of the main code succeeding the package, which adds several objects of type STD\_LOGIC\_VECTOR.

```

1 -----Package-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 PACKAGE my_package IS
6     FUNCTION "+" (a, b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
7 END PACKAGE;
8 -----
9 PACKAGE BODY my_package IS
10    FUNCTION "+" (a, b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
11        ALIAS aa: STD_LOGIC_VECTOR(1 TO a'LENGTH) IS a;
12        ALIAS bb: STD_LOGIC_VECTOR(1 TO b'LENGTH) IS b;
13        VARIABLE result: STD_LOGIC_VECTOR(1 TO a'LENGTH);
14        VARIABLE carry: STD_LOGIC := '0';
15    BEGIN
16        FOR i IN result'REVERSE_RANGE LOOP
17            result(i) := aa(i) XOR bb(i) XOR carry;

```



```

18      carry := (aa(i) AND bb(i)) OR (aa(i) AND carry) OR (bb(i) AND carry);
19          OR (bb(i) AND carry);
20      END LOOP;
21      RETURN result;
22  END FUNCTION "+";
23 END PACKAGE BODY;
24 -----
1 -----Main code:-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE work.my_package.all;
5 -----
6 ENTITY add_stdlogic IS
7     PORT (x: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
8           y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9 END ENTITY;
10 -----
11 ARCHITECTURE adder OF add_stdlogic IS
12     CONSTANT const: STD_LOGIC_VECTOR(7 DOWNTO 0) := "00001111";
13 BEGIN
14     y <= x + const + "01111111"; --overloaded "+" operator
15 END ARCHITECTURE;
16 -----

```

این فصل را با مثالی راجع به ساخت توابع به پایان می‌بریم. این مثال تا حدی برخلاف مثال قبلی است. مثال قبلی یک عملگر را با تعریف تابعی جدید برای آن سربارگذاری کرد. مثال بعدی از تابعی معادل با تابعی که قبلاً وجود دارد استفاده می‌کند اما عملگر را سربارگذاری نمی‌کند.

#### Example 9.6: Non-overloaded "AND" Operator

According to section 4.2 and figure 4.1, logical operators were already defined for the type STD\_LOGIC\_VECTOR in its package of origin (*std\_logic\_1164*, appendix I). Write a FUNCTION that computes the AND function and, contrary to the previous example, does not overload the AND operator.

**Solution** Not overloading an operator is trivial: just give the function a different name (like *my\_and*, used in the code below). The FUNCTION was again declared in a PACKAGE (line 6) and then constructed in the corresponding PACKAGE BODY (lines 25–34). Note the specification of a new data type, called *stdlogic\_table* (line 10), followed by a CONSTANT, called *and\_table* (lines 11–24—see the PACKAGE BODY of the *std\_logic\_1164* package in the VHDL libraries that accompany your VHDL compiler), which conforms with that data type. Observe that this constant is a table that works as a resolution function for the AND function, which is accessed using *enumerated* indexing (section 3.12). Two ALIAS declarations (section 4.8) appear in lines 26–27, with the purpose of normalizing the range of *a* and *b* to "1 TO *a'LENGTH*" and "1 TO *b'LENGTH*", respectively, which helps write the function. The function proper is in the LOOP statement of lines 30–32, which accesses the *and\_table* for each bit of *aa* and *bb*. Note that *a* and *b* must have the same length, a condition that was not tested in this example (see exercise 9.2). A main code, with a call to this function, is also included in the code below.

```

1 -----Package-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 PACKAGE my_package IS
6     FUNCTION my_and (a, b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
7 END PACKAGE;
8 -----
9 PACKAGE BODY my_package IS
10    TYPE stdlogic_table IS ARRAY(STD_ULOGIC, STD_ULOGIC) OF STD_ULOGIC;
11    CONSTANT and_table: stdlogic_table := (
12        -----
13        -- U   X   0   1   Z   W   L   H   -
14        -----
15        ('U', 'U', '0', 'U', 'U', '0', 'U', 'U'), --| U |
16        ('U', 'X', '0', 'X', 'X', '0', 'X', 'X'), --| X |
17        ('0', '0', '0', '0', '0', '0', '0', '0'), --| 0 |
18        ('U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'), --| 1 |
19        ('U', 'X', '0', 'X', 'X', '0', 'X', 'X'), --| Z |
20        ('U', 'X', '0', 'X', 'X', '0', 'X', 'X'), --| W |
21        ('0', '0', '0', '0', '0', '0', '0', '0'), --| L |
22        ('U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'), --| H |
23        ('U', 'X', '0', 'X', 'X', '0', 'X', 'X')); --| - |
24    -----
25    FUNCTION my_and (a, b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
26        ALIAS aa: STD_LOGIC_VECTOR(1 TO a'LENGTH) IS a;
27        ALIAS bb: STD_LOGIC_VECTOR(1 TO b'LENGTH) IS b;
28        VARIABLE result: STD_LOGIC_VECTOR(1 TO a'LENGTH);
29    BEGIN
30        FOR i IN result'RANGE LOOP
31            result(i) := and_table (aa(i), bb(i));
32        END LOOP;
33        RETURN result;
34    END FUNCTION;
35 END PACKAGE BODY;
36 -----

```



```

1 -----Main code-----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE work.my_package.all;
5 -----
6 ENTITY myand IS
7     PORT (x1, x2: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
8             y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
9 END ENTITY;
10 -----
11 ARCHITECTURE myand OF myand IS
12 BEGIN
13     y <= my_and(x1, x2);
14 END ARCHITECTURE;
15 -----

```



## 9.7 VHDL 2008

With respect to the material covered in this chapter, the main additions specified in VHDL 2008 are listed below.

- 1) The TO\_STRING type-conversion function was introduced, which eases the construction of ASSERT statements because it supports a much wider set of types than TIMAGE(X). The types supported by TO\_STRING are BOOLEAN, BIT, BIT\_VECTOR, INTEGER, NATURAL, POSITIVE, CHARACTER, STD\_(U)LOGIC\_VECTOR, (UN)SIGNED, REAL, TIME, SFIXED, UFIXED, and FLOAT.
- 2) GENERIC lists are allowed in FUNCTION and PROCEDURE. The corresponding syntaxes are depicted below.

```

[PURE | IMPURE] FUNCTION
function_name
[GENERIC (generic_list)]
[(input_list)]
RETURN return_value_type IS
    [declarative_part]
BEGIN
    statements_part
    [label:] RETURN expression;
END [FUNCTION] [function_name];

```

```

PROCEDURE procedure_name
[GENERIC (generic_list)]
(input_output_list) IS
    [declarative_part]
BEGIN
    statements_part
END [PROCEDURE] [procedure_name]

```

- 3) In PROCEDURE, an OUT mode object can be read internally.
- 4) Recall that in VHDL 2008 the WHEN and SELECT statements can also be used inside sequential code.

## یادآوری از بخش ۳-۴ برای ALIAS

## 4.8 ALIAS

An ALIAS declaration defines an alternate name for an existing named entity (not to be confused with ENTITY). A simplified syntax is shown below.

```
ALIAS new_name [: specifications] IS original_name [signature];
```

ALIAS can be applied to nearly all named entities, except for labels, loop parameters, and generate parameters. It is divided into two groups: *object alias* (for VHDL objects; that is, CONSTANT, SIGNAL, VARIABLE, and FILE) and *non-object alias* (for all remaining entities; that is, TYPE, COMPONENT, operators, user subprograms, etc.). One difference between these two groups is that a signature cannot be used in the former, while in the latter it is required if the entity is a subprogram.

The most common places for ALIAS declarations are the declarative parts of architectures and of subprograms, and the most commonly aliased entities are subprograms, SIGNAL, and TYPE.

The use of ALIAS can be seen, for example, in the package body of the package *std\_logic\_1164* that accompanies the VHDL compiler.

**Example** This example illustrates how an ALIAS declaration can be applied to a VHDL object (a SIGNAL, in this example) to do any of the following: (i) change its name; (ii) create a name for a section of it; (iii) change its range; (iv) create a new range for a section of it.

---

```
--This is the object to which ALIAS will be applied:
SIGNAL data_bus: STD_LOGIC_VECTOR(31 DOWNTO 0);

--bus1 is a new name for data_bus:
ALIAS bus1 IS data_bus;

--bus2 is a new name for data_bus, but with a modified range:
ALIAS bus2: STD_LOGIC_VECTOR(32 DOWNTO 1) IS data_bus;

--bus3 is another name for data_bus, with an ascending range:
ALIAS bus3: STD_LOGIC_VECTOR(1 TO 32) IS data_bus;

--upper_bus1 is a new name for the upper half of data_bus
ALIAS upper_bus1 IS data_bus(31 DOWNTO 16);

--upper_bus2 is a new name for the upper half of data_bus, but
--with a modified range:
ALIAS upper_bus2: STD_LOGIC_VECTOR(17 TO 32) IS data_bus(31 DOWNTO 16);

--lower_bus1 is a new name for the lower half of data_bus
ALIAS lower_bus1 IS data_bus(15 DOWNTO 0);
```

---

```
--lower_bus2 is a new name for the lower half of data_bus, but
--with a modified range:
ALIAS lower_bus2: STD_LOGIC_VECTOR(1 TO 16) IS data_bus(15 DOWNTO 0);
-----
```

As indicated in the syntax of ALIAS, the specification of the original name allows the inclusion of a *signature*. Its purpose is the same as that seen in the study of attributes (section 4.5); that is, to make possible the identification among overloaded items (with the same name). Signatures can be used in ALIAS declarations for subprograms (that is, FUNCTION and PROCEDURE).

**Example** This example illustrates how an ALIAS declaration can be applied to a VHDL subprogram (a PROCEDURE, in this case), which also includes the use of a signature. Let us consider again the *sort* procedure seen at the end of section 4.5, which assigns the smaller and larger of *a* and *b* to *x* and *y*, respectively. However, assume that two versions of *sort* were written, one for ports of type INTEGER, the other for ports of type BIT\_VECTOR. That is:

```
PROCEDURE sort (a, b: IN INTEGER; x, y: OUT INTEGER) IS ...
PROCEDURE sort (a, b: IN BIT_VECTOR; x, y: OUT BIT_VECTOR) IS ...
```

The ALIAS declarations below define alternative names for *sort* (*sort\_int* for the former, *sort\_bv* for the latter), thus allowing the procedures to be easily identified.

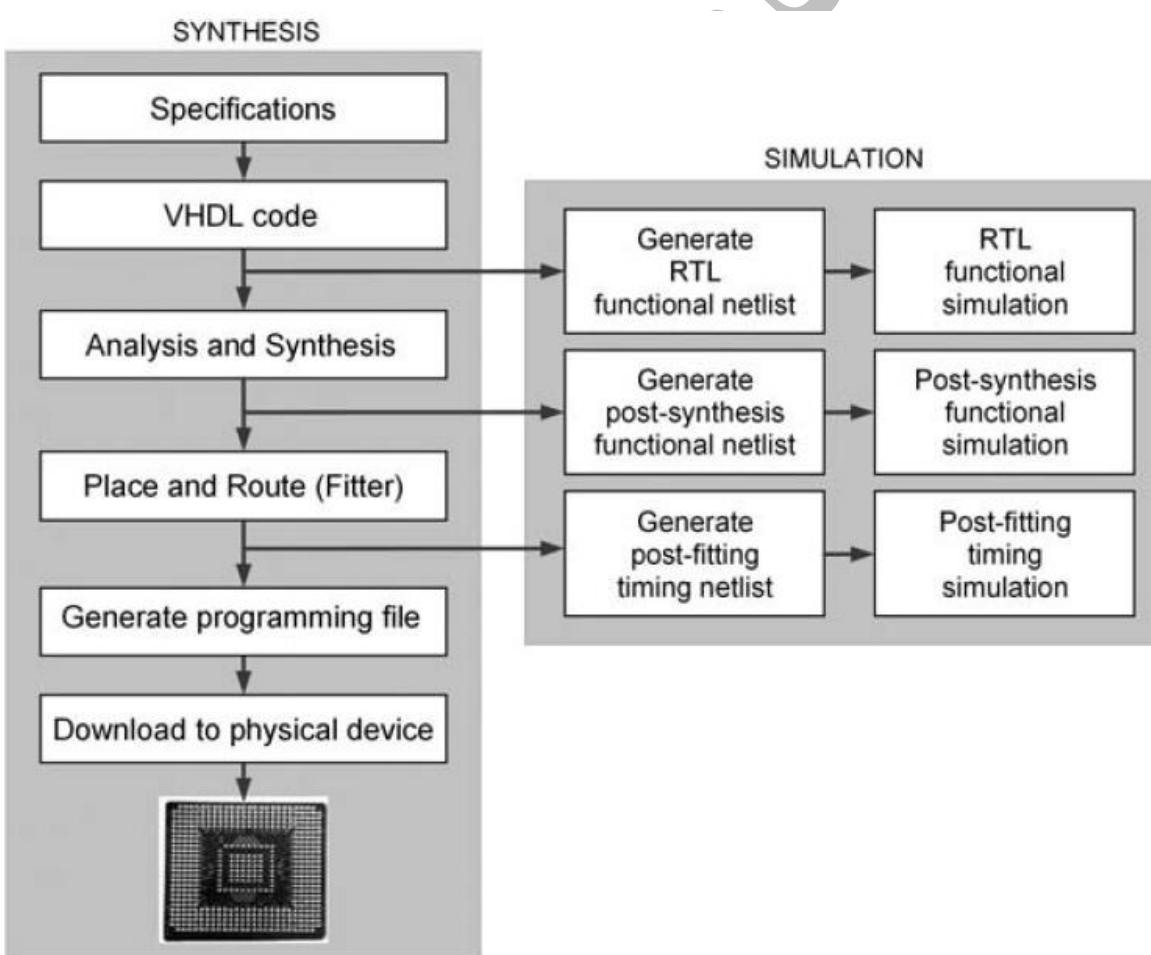
```
ALIAS sort_int IS sort [INTEGER, INTEGER, INTEGER, INTEGER];
ALIAS sort_bv IS sort [BIT_VECTOR, BIT_VECTOR, BIT_VECTOR, BIT_VECTOR];
```



## فصل دهم

# شبیه‌سازی با بستر آزمایش در VHDL

تمام مطالب موجود در این کتاب به جز فصل ۱۰، مرتبط با سنتز می‌باشد.  
شکل ۱-۱۰ (از فصل اول مجدداً آورده شده است) روند ساده‌ای از طراحی در VHDL را نشان می‌دهد. مراحل سنتز در سمت چپ نشان داده شده‌اند حال آن که گزینه‌های شبیه‌سازی در سمت راست نشان داده شده‌اند. ملاحظه می‌کنید که حداقل در سه نقطه نیاز به شبیه‌سازی وجود دارد.



**Figure 10.1**  
Simplified VHDL design flow.

شبیه سازی RTL<sup>۱</sup> مبتنی بر کد VHDL است. در این مرحله اطلاعات زمان بندی داده نشده و نیز از اطلاعات افزارهی مورد نظر استفاده نمی شود؛ بلکه تنها کارایی طراحی بررسی و ارزیابی می شود. به این نوع شبیه سازی، «شبیه سازی عملکردی»<sup>۲</sup> نیز گفته می شود. شبیه سازی بعدی نیز عملکردی (هنوز هم اطلاعات زمان بندی داده نمی شود) بوده و این مرحله بعد از سنتز اجرا می شود. هدف این مرحله این است که آیا پس از سنتز نیز همچنان همان عملکرد به قوت و اعتبار خود باقی است؟ آخرین مرحلهی شبیه سازی پس از جانمایی<sup>۳</sup> انجام شده و شامل اطلاعات سلولهای داخلی و تاخیر مسیرها می باشد لذا بر افزارهی فیزیکی واقعی منطبق است. با توجه به شامل شدن اطلاعات زمایی به این نوع شبیه سازی، «شبیه سازی زمان بندی»<sup>۴</sup> گفته می شود. برای مثال، تمام طراحی های انجام شده در فصول بعدی از نوع زمان بندی است.

اطلاعات زمانی در یک فایل SDF<sup>۵</sup> ذخیره می شود که منطبق بر استانداردهای VITAL می باشد. اطلاعات ذخیره شده در این فایلها کمک می کنند تا شبیه سازی در دو حالت بهترین و بدترین شرایط انجام شود. در Xilinx از این اطلاعات با نمادهای MIN، TYP و MAX نام برده شده و کاربر قادر به انتخاب یکی از آنها است اما در Altera در حالت عادی از بدترین شرایط در انجام شبیه سازی استفاده می شود؛ بهترین حالت نیز در شبیه سازی از نوع سریع<sup>۶</sup> انجام می شود.

لیستی از شبیه سازهای معروف در زیر لیست شده است (در مثالهای انجام شده در این فصل از شبیه ساز ModelSim استفاده شده است).

- From Altera: Quartus II graphical simulator
- From Xilinx: ISE simulator
- From Mentor Graphics: ModelSim
- From Synopsys: VCS
- From Cadence: NC-Sim
- From Aldec: Active-HDL

در زیر برخی توصیه های ساده اما مفید در زمینهی انجام شبیه سازی آورده شده است:

<sup>1</sup> Register Transfer Level

<sup>2</sup> Functional Simulation

<sup>3</sup> Fitting

<sup>4</sup> Timing Simulation

<sup>5</sup> Standard Delay Format

<sup>6</sup> Fast Simulation

- 1) In-system testing might not be enough to catch all design problems, so also include timing simulation, at the least.
- 2) Keep the projects that are for simulation separate from those that are for synthesis (in the same way that the material in this chapter was kept apart from that in all the other chapters).
- 3) Remember that the fundamental time-related statements and function are AFTER, WAIT FOR, and NOW. The attribute 'LAST\_EVENT' is also helpful sometimes.
- 4) Never use these statements and function in code that is for synthesis.
- 5) Use GENERIC to enter arbitrary design/simulation parameters whenever appropriate.
- 6) Do not hesitate to use type-conversion functions whenever convenient (see figure 3.10).
- 7) To convert an INTEGER type into a TIME type, just multiply the former by one unit of the desired time scale. For example,  $time\_value = int\_value * 1ns$ .
- 8) Make sure to feed the same clock to all units that belong to the same clock domain.
- 9) In the simulation of large, complex systems, use files to enter/store data (see sections 10.3, 10.4, and 10.13).
- 10) Do not display unimportant data.
- 11) Before starting a simulation, make sure that you have the correct design code and that you have understood its functionalities.
- 12) Finally, always provide a means for the simulation to end. For example, close the code with a WAIT FOR statement with an additional time interval such that the total simulation time (in the code) is larger than the total time set up in the simulation software, so the latter will be able to close the simulation unconditionally when its time limit is reached (this can be done, for example, with a simple "WAIT;" at the end of the code).

## ۲-۱۰ انواع شبیه‌سازی

شکل ۲-۱۰ شش نوع شبیه‌سازی را نشان می‌دهد. در سمت چپ این شکل ورودی طراحی مشخص شده که به دو صورت گرافیکی و یا کد VHDL می‌تواند باشد. در وسط، طراحی تحت آزمایش (DUT<sup>1</sup>) نشان داده شده است که همیشه با VHDL این طراحی انجام می‌شود (تاخیرهای انتشار ممکن است در نظر گرفته شده باشد یا نشده باشد). و بالاخره در سمت راست نیز پاسخ مدار نمایش داده شده است. این پاسخ یا به صورت گرافیکی ارزیابی می‌شود و یا به کمک کد VHDL ارزیابی و بررسی می‌شود (تصدیق خودکار).

در شکلهای ۲-۱۰-الف و ۲-۱۰-ب هم ورودی و هم خروجی به بصورت گرافیکی هستند (کاربر به کمک یک ابزار گرافیکی شکل موج ورودی را رسم کرده و سپس شبیه‌سازی شکل موج‌های خروجی را محاسبه و رسم می‌کند؛ حال کاربر به صورت بصری خروجی را ارزیابی می‌کند) لذا به آنها شبیه‌سازی گرافیکی<sup>2</sup> گفته می‌شود. در شکل (الف) تاخیرهای انتشار در نظر

<sup>1</sup> Design Under Test

<sup>2</sup> Graphical Simulation

گرفته نشده است اما در شکل (ب) این تاخیرها لحاظ شده‌اند؛ لذا به اولی شبیه‌سازی عملکردی<sup>۱</sup> و به دومی شبیه‌سازی زمان‌بندی<sup>۲</sup> گفته می‌شود.

در شکل‌های ۱۰-۲-ج و ۱۰-۲-د محرک‌ها<sup>۳</sup> توسط یک کد VHDL تعیین و تولید شده‌اند. اما خروجی‌ها همچنان گرافیکی هستند. از آنجایی که تنها تهییی ورودی‌ها (یا محرک‌ها) به صورت خودکار انجام شده است، به این دو نوع شبیه‌سازی، شبیه‌سازی محرک-تنها<sup>۴</sup> یا شبیه‌سازی دستی<sup>۵</sup> گفته می‌شود. مجدداً در اینجا نیز دو حالت شبیه‌سازی عملکردی و زمان‌بندی مطرح هستند که به آنها به ترتیب شبیه‌سازی‌های نوع I<sup>۶</sup> و نوع II<sup>۷</sup> گفته می‌شود.

و بالاخره در شکل‌های ۱۰-۲-۰ و ۱۰-۲-۰ و هر دو ورودی و خروجی توسط کد VHDL مدیریت می‌شوند (یعنی محرک‌ها را یک کد VHDL تولید کرده و ارزیابی صحت خروجی‌ها را نیز یک کد VHDL انجام می‌دهد؛ این ارزیابی به کمک مقایسه نتایج با خروجی‌های از قبل مشخص و مورد انتظار انجام می‌شود). با توجه به این که در این حالت هم ورودی‌ها (محرک‌ها) را یک کد تولید کرده و هم ارزیابی نتایج را یک کد VHDL بر عهده دارد (یعنی کارها خودکار انجام می‌شود)، این دو نوع شبیه‌سازی را «شبیه‌سازی خودکار»<sup>۸</sup> می‌نامند. در این جا هم یکی از این شبیه‌سازی‌ها را عملکردی و دیگری را زمان‌بندی می‌نامند. به این دو نوع شبیه‌سازی عملکردی و زمان‌بندی به ترتیب شبیه‌سازی نوع III و IV گفته می‌شود.

خلاصه‌ی چهار نوع شبیه‌سازی به صورت زیر جمع‌بندی شده است:

- Type I testbench (*manual functional simulation*): The DUT's internal delays are not considered and the output is manually verified (normally by visual inspection). This is the simplest kind of VHDL code for simulation.
- Type II testbench (*manual timing simulation*): The DUT's internal delays are taken into account, but the output is still manually verified.
- Type III testbench (*automated functional simulation*): The DUT's internal delays are not considered, but the output is automatically verified by the simulator (the data for comparison can be included, for example, in the test file itself or in a separate file).
- Type IV testbench (*automated timing simulation*, also called *full bench*): The DUT's internal delays are taken into account and the output is automatically verified by the simulator. This is obviously the most complete and also the most complex type of simulation with testbenches.

<sup>1</sup> Functional Simulation

<sup>2</sup> Timing Simulation

<sup>3</sup> Stimuli

<sup>4</sup> Stimuli-Only

<sup>5</sup> Manual Simulation

<sup>6</sup> Type I

<sup>7</sup> Type II

<sup>8</sup> Automated Simulation

### ۱۰-۳ نوشتن داده‌ها در فایل

فایلها برای ذخیره‌ی داده‌های حاصل از شبیه سازی اهمیت زیادی دارند. مهمترین روالهای VHDL (برگرفته از بسته‌ی textio در پیوست M و بسته‌ی standard در پیوست H) جهت نوشتن داده‌ها در فایلها در زیر لیست شده‌اند. در این روالها این انواع داده به کار گرفته می‌شوند:

```
TYPE LINE IS ACCESS STRING;
TYPE TEXT IS FILE OF STRING;
TYPE SIDE IS (left, right);
SUBTYPE WIDTH IS NATURAL;
```

۱- برای باز کردن یک فایل در حالت «نوشتن» که در آن f یک شناسه‌ی فایل<sup>۱</sup> است (هر نامی جهت نام فایل، قابل استفاده است):

```
FILE f: TEXT OPEN WRITE_MODE IS "file_name";
```

۲- برای نوشتن یک مقدار val در یک متغیر l از نوع LINE (دوگزینه وجود دارد) که در آن 'INTEGER، 'BIT\_VECTOR، 'BIT، 'BOOLEAN می‌تواند شامل data\_type باشد: CHARACTER، TIME، REAL

```
PROCEDURE WRITE(l: INOUT LINE; val: IN data_type);
```

```
PROCEDURE WRITE(l: INOUT LINE; val: IN data_type;
```

```
justified: IN SIDE := right; field: IN WIDTH := 0);
```

۳- برای نوشتن خط l به داخل فایل مشخص شده با شناسه‌ی f :

```
PROCEDURE WRITELINE(FILE f: TEXT; l: INOUT LINE);
```

۴- حالت‌های فایل و پرچم‌های خطاب عبارتند از:

```
TYPE file_open_kind IS (read_mode, write_mode, append_mode);
```

```
TYPE file_open_status IS (open_ok, status_error, name_error, mode_error);
```

استفاده از روال‌های فوق در مثال زیر نشان داده شده است.

<sup>1</sup> File Identifier

**Example 10.1: Writing Values to a File**

Write a VHDL code that generates a clock with a 100ns period and writes, at every positive clock transition, a value to a table in a text file containing in each line the time value at the clock transition followed by an integer (pointer)  $i$  from 0 to 7, as shown below.

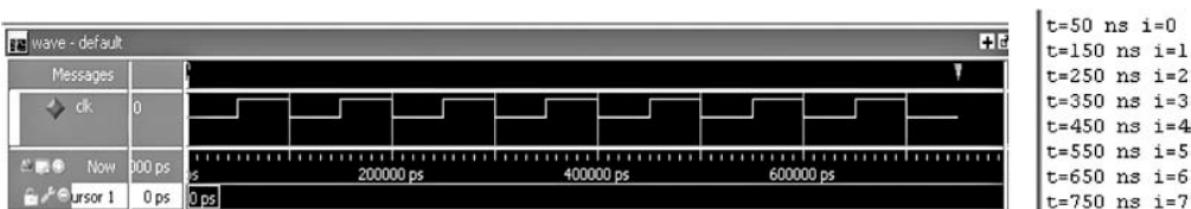
```
t=50ns i=0
t=150ns i=1
t=250ns i=2
t=350ns i=3
t=450ns i=4
t=550ns i=5
t=650ns i=6
t=750ns i=7
```

**Solution** A VHDL code for this exercise is presented below. Note the presence of the *textio* package in line 2. As with all simulation codes, the entity (lines 4–5) is empty (optionally, GENERIC can be used).

The declarative part of the architecture (lines 8–10) contains a constant representing the desired clock period (100ns), a signal declaration for the clock (with initial value '0'), and finally a file declaration for the output file. Such a file, named *test\_file.txt* and identified by  $f$ , is declared to be opened in *write* mode.

In the code proper (lines 11–28), a process is employed to create the clock and the file. The declarative part of the process (lines 13–17) contains two constants of type STRING to provide the proper fixed characters for the table. It also contains a variable  $l$ , of type LINE, to represent the file's lines, and a variable  $t$ , or type TIME, to represent the time (must include a time unit; ns in this case). Finally, it contains a variable  $i$ , of type NATURAL, to represent the requested 0-to-7 pointer. The process body (lines 18–27) produces the desired clock (lines 19–20 plus 25–26), the pointer (line 24), and the file (line 22 creates a file line, which is written to the actual file in line 23). Before starting the writing procedure, the file is automatically cleared.

If simulated with ModelSim (appendix D), for example, the waveform and file contents depicted in figure 10.3 will be obtained (in the file, the time unit, ns, and the space between



**Figure 10.3**  
Simulation results from example 10.1.

it and the time value are automatically inserted by the simulator). To perform such a simulation, follow the procedure in sections D.1 to D.3 of appendix D, but using just one file.

```

1 -----
2 USE std.textio.all;
3 -----
4 ENTITY write_to_file IS
5 END ENTITY;
6 -----
7 ARCHITECTURE write_to_file OF write_to_file IS
8     CONSTANT period: TIME := 100ns;
9     SIGNAL clk: BIT := '0';
10    FILE f: TEXT OPEN WRITE_MODE IS "test_file.txt";
11 BEGIN
12    PROCESS
13        CONSTANT str1: string(1 TO 2) := "t=";
14        CONSTANT str2: string(1 TO 3) := " i=";
15        VARIABLE l: LINE;
16        VARIABLE t: TIME RANGE 0ns TO 800ns;
17        VARIABLE i: NATURAL RANGE 0 TO 7 := 0;
18    BEGIN
19        WAIT FOR period/2;
20        clk <= '1';
21        t := period/2 + i*period;
22        WRITE(l, str1); WRITE(l, t); WRITE(l, str2); WRITE(l, i);
23        WRITELINE(f, l);
24        i := i + 1;
25        WAIT FOR period/2;
26        clk <='0';
27    END PROCESS;
28 END ARCHITECTURE;
29 -----

```

## ۴- خواندن داده‌ها از فایلها

مهمترین روالهای VHDL (برگرفته از بسته `textio` در پیوست M و بسته `H`) جهت خواندن داده‌ها از فایلها در زیر لیست شده‌اند. در این روالها این انواع داده به کار گرفته می‌شوند:

```

        TYPE LINE IS ACCESS STRING;
        TYPE TEXT IS FILE OF STRING;
        TYPE SIDE IS (left, right);
        SUBTYPE WIDTH IS NATURAL;

```

۱- برای بازکردن فایلها در حالت «خواندن»، که در آن `f` یک شناسه است (از هر نام دلخواهی می‌توان استفاده کرد)

```
FILE f: TEXT OPEN READ_MODE IS "file_name";
```

۲- برای خواندن یک خط از یک فایل (با شناسه‌ی فایل *f*) و تخصیص آن به متغیر ۱:

```
PROCEDURE READLINE(FILE identifier: TEXT; l: OUT LINE);
```

۳- برای خواندن یک مقدار از خط ۱ و تخصیص آن به متغیر *val* (دو راه وجود دارد)، که در آن *data\_type* می‌تواند شامل *BIT\_VECTOR*، *BIT*، *BOOLEAN*، *INTEGER*، *CHARACTER*، *TIME*، *REAL* باشد:

```
PROCEDURE READ(l: INOUT LINE; val: OUT data_type);
```

```
PROCEDURE READ(l: INOUT LINE; val: OUT data_type; good: OUT BOOLEAN);
```

تست *good* در آخرین روال، در صورتی که نوع مقدار خوانده شده از فایل با نوع شیء‌ای که چنین مقداری به آن نسبت داده شده، تطابق نداشته باشد، مقدار FALSE را برمی‌گرداند.

۴- نمونه روش متدالو برای بررسی رسیدن به انتهای فایل:

```
WHILE NOT ENDFILE (f) LOOP ...
```

گرچه دستور ENDFILE یک دستور واقعی VHDL نمی‌باشد اما هر شبیه‌سازی از آن پشتیبانی می‌کند.

۵- از دستور ASSERT (بخش ۲-۹) برای مقایسه نتایج با مقادیر مورد انتظار و نیز برای شناسایی مقادیر نامناسب (و غیرمنطبق بر انتظاراتان) در حین خواندن از فایل‌ها استفاده کنید.

استفاده از روال‌های فوق در مثال زیر نشان داده شده است.

#### Example 10.2: Reading Values from a File

Write a VHDL code that generates a clock with a 100ns period and reads, at every positive clock transition, one line from the file *test\_file.txt* written by the code of example 10.1. Display in the waveforms the signals *t* (time) and *i* (pointer) read from that file.

**Solution** A VHDL code for this example is presented below. Note again the presence of the *textio* package (line 2) and that the entity (lines 4–5) is empty.

The declarative part of the architecture (lines 8–11) contains a file declaration for the input file, followed by three signal declarations concerning the signals that must be displayed in the waveforms. The input file is named *test\_file.txt*, identified by *f*, now opened in *read* mode.

In the code proper (lines 12–31), a process is employed to read the file and produce the three output signals. The declarative part of the process (lines 14–18) contains essentially the same objects seen in the previous example. The process body (lines 19–30) produces the desired clock (lines 20–21 plus 28–29) and reads the file. While it is not end-of-file (line 22), a line is read (line 23) after every positive clock transition, with the four parameters assigned to proper variables (line 24), of which *t* and *i* are assigned to *t\_out* (line 25) and *i\_out* (line 26), respectively.

```

1 -----
2 USE std.textio.all;
3 -----
4 ENTITY read_from_file IS
5 END ENTITY;
6 -----
7 ARCHITECTURE read_from_file OF read_from_file IS
8     FILE f: TEXT OPEN READ_MODE IS "test_file.txt";
9     SIGNAL clk: BIT := '0';
10    SIGNAL t_out: TIME RANGE 0ns TO 800ns;
11    SIGNAL i_out: NATURAL RANGE 0 TO 7;
12 BEGIN
13     PROCESS
14         VARIABLE l: LINE;
15         VARIABLE str1: string(1 TO 2);
16         VARIABLE str2: string(1 TO 3);
17         VARIABLE t: TIME RANGE 0ns TO 800ns;
18         VARIABLE i: NATURAL RANGE 0 TO 7;
19     BEGIN
20         WAIT FOR 50ns;
21         clk <= '1';
22         IF NOT ENDFILE(f) THEN
23             READLINE(f, l);
24             READ(l, str1); READ(l, t); READ(l, str2); READ(l, i);
25             t_out <= t;
26             i_out <= i;
27         END IF;
28         WAIT FOR 50ns;
29         clk <= '0';
30     END PROCESS;
31 END ARCHITECTURE;
32 -----

```

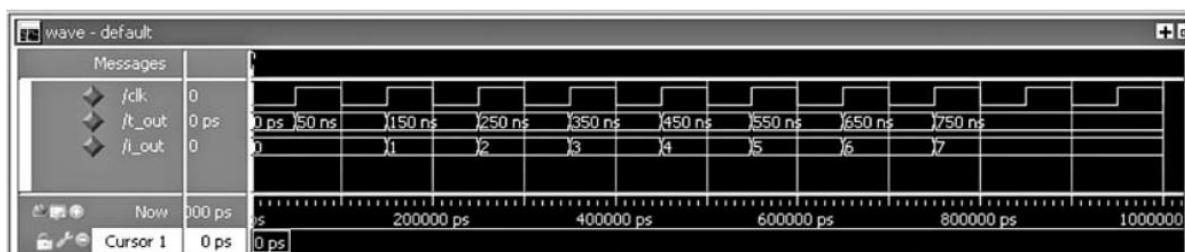


Figure 10.4  
Simulation results from example 10.2.

یک راه دیگر:

A more sophisticated code is shown below, which includes a verification for the second parameter (*t*) read from the file. If its type does not match the type specified for *t\_out*, the message “Bad value at *i* = 5!” (assuming that the previous value of *i* is 4) is issued and the software quits reading the file (see details about ASSERT and IMAGE in section 9.2). The same check could obviously also be included for the other values read from the file.

```

22      IF NOT ENDFILE(f) THEN
23          READ(l, str1);
24          READ(l, t, good_value);
25          ASSERT good_value
26              REPORT "Bad value at i=" & INTEGER'IMAGE(i+1) & "!"
27              SEVERITY FAILURE;
28          READ(l, str2); READ(l, i);
29          t_out <= t;
30          i_out <= i;
31      END IF;
32      WAIT FOR 50ns;
33      clk <= '0';
34  END PROCESS;
35 END ARCHITECTURE;
36 -----

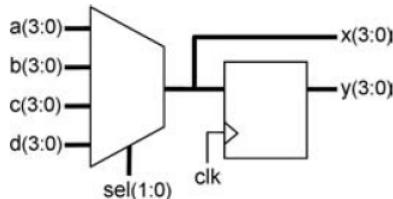
```

If simulated with ModelSim (appendix D), for example, the waveforms of figure 10.4 will be obtained, which display *clk*, *t\_out*, and *i\_out*. To perform such a simulation, follow the procedure in sections D.1 to D.3 of appendix D, but using just one file.

بخش بعدی راجع به شبیه‌سازی گرافیکی بحث کرده که چندان مورد استفاده ما در این درس سیستم‌های FPGA نمی‌باشد).

## 10.5 Graphical Simulation (Preparing the Design)

Figure 10.5 shows a registered  $4 \times 4$  multiplexer (note that the registered output is  $y$ , while  $x$  is unregistered). This circuit will be used in the examples with VHDL testbenches in later sections, so before proceeding it will be designed in this section. It will also be simulated here using *graphical simulation* (no VHDL simulation yet) in both *functional* and *timing* modes.



**Figure 10.5**  
Registered multiplexer.

A VHDL code for this circuit is presented below, under the name *reg\_mux* (line 5). Note that all ports (lines 6–9) are of type STD\_LOGIC\_VECTOR (industry standard). The  $4 \times 4$  multiplexer (lines 15–20) constitutes the combinational part of the circuit, while the 4-bit register (lines 21–26) is the sequential part.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY reg_mux IS
6     PORT (a, b, c, d: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
7             sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
8             clk: IN STD_LOGIC;
9             x, y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
10 END ENTITY;
11 -----
12 ARCHITECTURE reg_mux OF reg_mux IS
13     SIGNAL mux: STD_LOGIC_VECTOR(3 DOWNTO 0);
14 BEGIN
15     mux <= a WHEN sel="00" ELSE

```



```

16      b WHEN sel="01" ELSE
17      c WHEN sel="10" ELSE
18          d;
19
20      x <= mux;
21  PROCESS (clk)
22  BEGIN
23      IF (clk'EVENT AND clk='1') THEN
24          y <= mux;
25      END IF;
26  END PROCESS;
27 END ARCHITECTURE;
28 -----

```

Say that the waveforms plotted for  $clk$ ,  $a$ ,  $b$ ,  $c$ ,  $d$  and  $sel$  in figure 10.6 are the desired test inputs, from which the simulator must determine the outputs  $x$  and  $y$ . Say also that we are using Quartus II (the conclusions would be exactly the same with any other simulator); then the setups for functional and timing simulations are those described below.

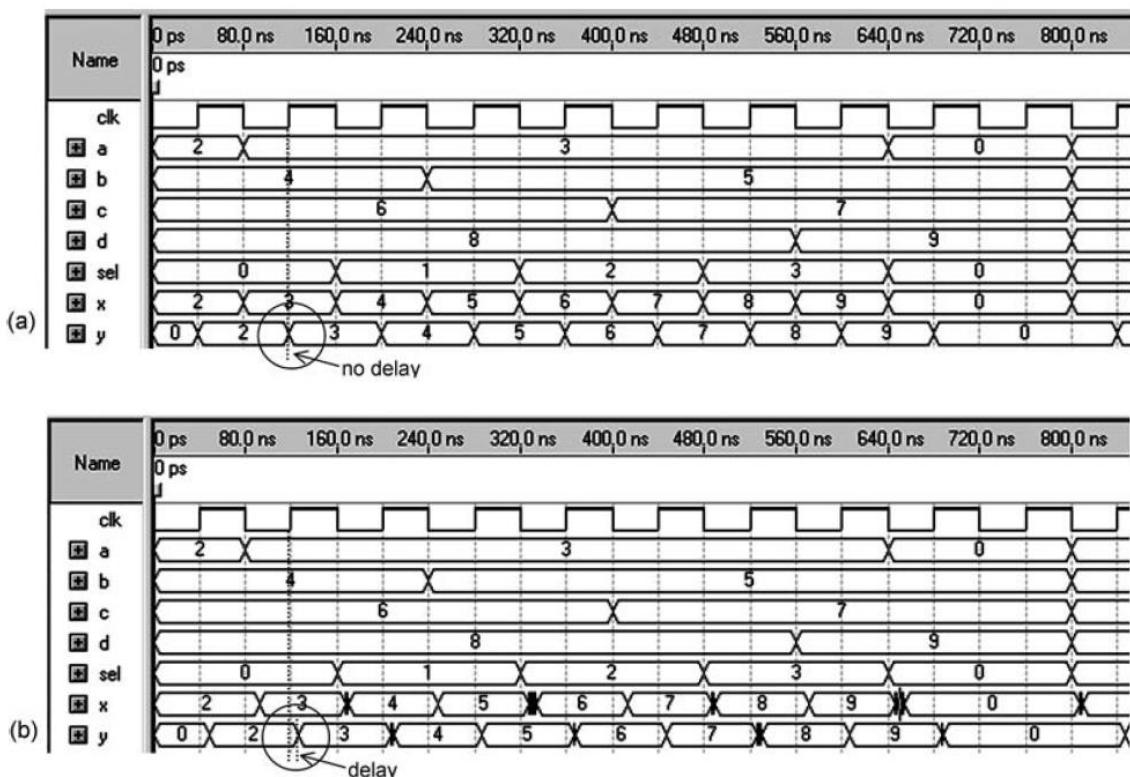


Figure 10.6  
(a) Functional and (b) timing simulation results from the registered multiplexer of figure 10.5.

در ادامه نحوه انجام دو نوع شبیه سازی عملکردی و زمان بندی در نرم افزار (ظاهراً ModelSim) توضیح داده شده است.

### Functional Simulation

Select Processing > Generate Functional Simulation Netlist. After the program finishes producing the netlist, select Assignments > Settings > Simulator Settings. In the Simulation Mode list, choose Functional. Finally, in the Simulation Input box, enter the name of the waveform file (*reg\_mux.vwf*) and click OK. Recompile the code and run the simulation. The results for *reg\_mux* are those displayed in figure 10.6a. Notice that there are no propagation delays (the output changes immediately when the clock changes).

### Timing Simulation

Select Assignments > Settings > Simulator Settings. In the Simulation Mode list, choose Timing. In the Simulation Input box, enter the name of the waveform file (*reg\_mux.vwf*) and click OK. Recompile the code and run the simulation. The results are now those displayed in figure 10.6b. Observe the propagation delay between the clock edge and the settling of *y*.

*Note:* If the timing simulation is performed using a third-party software (ModelSim, for example) instead of the Quartus II simulator, postsynthesis and SDF files specific for that simulator must be generated by the synthesizer, hence requiring appropriate setups (described in the ModelSim tutorial, appendix D).

In the example above, the simulation results were *visually* inspected. This approach is fine for individual system units or for small systems, but not for large systems. As described ahead, the use of VHDL testbenches allows automated verification to be performed. It also allows the testing code to be reused and the testing procedure to be more effectively documented.

روش ورودی گرافیکی به منظور ارزیابی در طراحی‌های بزرگ چندان موثر و مفید نمی‌باشد. در ادامه از کد VHDL به منظور نوشتتن بستر آزمایش<sup>۱</sup> و تامین ورودی‌ها استفاده شده است.

## ۱۰-۶ تولید محرك‌ها

تولید محرك یکی از بخش‌های اساسی هر بستر آزمایش است. برای تولید هر بستر آزمایش معمولاً از NOW و WAIT FOR، AFTER دستور استفاده می‌شود. دستور AFTER یک دستور همزمان است اما دستور WAIT FOR یک دستور ترتیبی است. بنابراین دستور اول باید خارج از یک دستور ترتیبی و دستور دوم باید داخل یک دستور ترتیبی (یعنی داخل فرآیند، تابع و روال) استفاده شود. با استفاده از این دو دستور، کدهای معادل با هم می‌توان نوشت.

دستور NOW معرف زمان کنونی شبیه‌سازی است لذا برای بررسی یک نقطه‌ی خاص از شبیه‌سازی مفید است. برای مثال دستور IF در زیر

<sup>1</sup> Testbench

```

IF (NOW<50ns) THEN
    WAIT FOR 50ns-NOW;
    y <= x;
END IF;

```

باعث می‌شود که فرآیند تا لحظه‌ی 50 ns متوقف شده و سپس پس از رسیدن به این زمان، مقدار x را به y تخصیص می‌دهد.

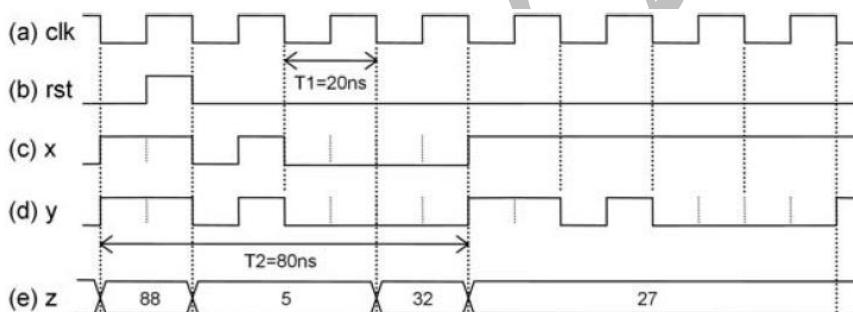
مختصه‌ی LAST EVENT نیز گاه‌هاً مفید واقع می‌شود. این مختصه، بازه‌ی زمانی سپری شده از آخرین تغییر انجام شده روی یک سیگنال یا متغیر را نشان می‌دهد. برای مثال می‌توان نوشت:

```

IF (s'LAST_EVENT>20ns) THEN ...

```

به کمک شکل زیر فرآیند تولید محرک را توضیح می‌دهیم. این شکل پنج شکل موج محرک را نشان می‌دهد که انواع متداوی محرک هستند. شکل موج (a) شکلی موجی منظم و متناوب است لذا نماینده‌ی سیگنال کلاک می‌تواند باشد. شکل موج (b) یک تک‌پالس است لذا نماینده‌ی مناسبی از ریست است. شکل موج (c) نامنظم و محدود است؛ شکل موج (d) نیز نامنظم اما متناوب است. و بالاخره، شکل موج (e) نیز یک شکل موج چندبیتی است.



**Figure 10.7**  
Typical stimuli: (a) Periodic signal (clock); (b) Single-pulse signal (reset); (c) Irregular finite signal; (d) Irregular periodic signal; (e) Multibit irregular signal.

نحوه‌ی تولید شکل موج‌های فوق به کمک دستورهای AFTER و WAIT FOR در زیر نشان داده شده است. کد کامل در مثال ۱۰-۳ نشان داده خواهد شد.

Generation of *clk* (figure 10.7a):

```
-----  
SIGNAL clk: STD_LOGIC := '0';  
---option 1:  
clk <= NOT clk AFTER 10ns;  
---option 2:  
WAIT FOR 10ns;  
clk <= NOT clk;  
---option 3:  
WAIT FOR 10ns;  
clk <= '1';  
WAIT FOR 10ns;  
clk <= '0';  
-----
```

Generation of *rst* (figure 10.7b):

```
-----  
SIGNAL rst: STD_LOGIC := '0';  
---option 1:  
rst <= '0', '1' AFTER 10ns, '0' AFTER 20ns;  
---option 2:  
rst <= '1' AFTER 10ns, '0' AFTER 20ns;  
---option 3:  
rst <= '0'; --optional  
WAIT FOR 10ns;  
rst <= '1';  
WAIT FOR 10ns;  
rst <= '0';  
WAIT;  
-----
```

Generation of  $x$  (figure 10.7c):

```

-----
SIGNAL x: STD_LOGIC := '1';
---option 1:-----
x <= '1', --optional
    '0' AFTER 20ns,
    '1' AFTER 30ns,
    '0' AFTER 40ns,
    '1' AFTER 80ns;
---option 2:-----
x <= '1'; --optional
WAIT FOR 20ns;
x <= '0';
WAIT FOR 10ns;
x <= '1';
WAIT FOR 10ns;
x <= '0';
WAIT FOR 40ns;
x <= '1';
WAIT;
---option 3:-----
CONSTANT template: STD_LOGIC_VECTOR(1 TO 9)
    := "110100001";
FOR i IN template'RANGE LOOP
    x <= template(i);
    WAIT FOR 10ns;
END LOOP;
WAIT;
-----
```



Generation of  $y$  (figure 10.7d); See exercise 10.1.

Generation of  $z$  (figure 10.7e):

```
-----  
SIGNAL z: NATURAL RANGE 0 TO 255 := 88;  
---option 1:-----  
z <= 88, --optional  
      5 AFTER 20ns,  
      32 AFTER 60ns,  
      27 AFTER 80ns;  
---option 2:-----  
z <= 88; --optional  
WAIT FOR 20ns;  
x <= 5;  
WAIT FOR 40ns;  
x <= 32;  
WAIT FOR 20ns;  
x <= 27;  
WAIT;  
-----
```

پی شامگرد

### Example 10.3: Stimuli Generation

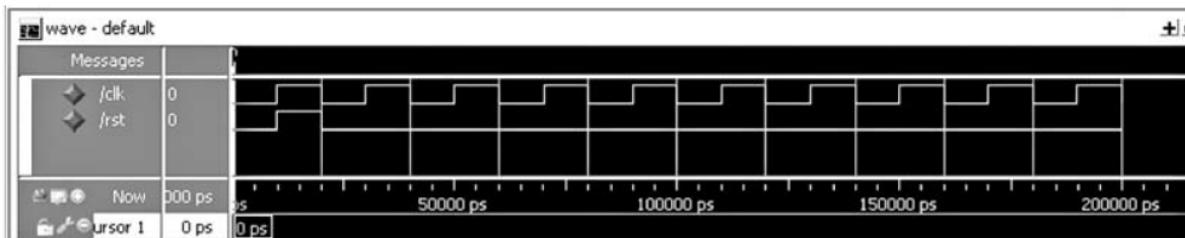
Write a complete VHDL code that generates the signals *clk* and *rst* of figure 10.7. Employ AFTER for the former and WAIT FOR for the latter.

**Solution** A VHDL code for this exercise is shown below. As with all codes for simulation, the entity (lines 5–6) has no PORT declarations. The declarative part of the architecture (lines 9–10) contains declarations relative to the two signals to be generated, with respective initial values. In the architecture body (lines 11–24), *clk* is generated using AFTER in line 13 (outside the process, because AFTER is concurrent), while *rst* is created in lines 15–22. Note that because WAIT FOR is sequential a process is needed, causing the code to be much longer than that for *clk* (the whole process could be replaced with just line 23). Simulation results from this code, compiled with ModelSim (appendix D), are depicted in figure 10.8. To check them, follow the procedure in sections D.1 to D.3 of appendix D, but using just one file.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY testbench IS
6 END ENTITY;
7 -----
8 ARCHITECTURE testbench OF testbench IS
9     SIGNAL clk: STD_LOGIC := '0';
10    SIGNAL rst: STD_LOGIC := '0';
11 BEGIN
12     --Generation of clk with AFTER:
13     clk <= NOT clk AFTER 10ns;
14     --Generation of rst with WAIT FOR:
15     PROCESS
16     BEGIN
17         WAIT FOR 10ns;
18         rst <= '1';
19         WAIT FOR 10ns;
20         rst <= '0';
21         WAIT;
22     END PROCESS;
23     --rst <= '1' AFTER 10ns, '0' AFTER 20ns;
24 END ARCHITECTURE;
25 -----

```



**Figure 10.8**  
Simulation results from example 10.3.

## ۷-۱۰ قالب‌های کلی برای بستر آزمایش

تعمیمی از قالب یک بستر آزمایش در شکل ۹-۱۰ نشان داده شده است. همان طور که در سمت چپ این کد نیز مشخص شده است، قالب این بستر آزمایش مانند یک کد معمولی VHDL است؛ یعنی اعلان کتابخانه، موجودیت و معماری.

ویرگی موجودیت در این قالب کلی این است که خالی است (البته استثناءً می‌تواند شامل GENERIC باشد). ویرگی معماری این قالب کلی نیز این است که به منظور سنتز نوشته نشده بلکه منحصراً به منظور شبیه‌سازی است.

در بخش اعلان معماری (خطوط ۱۰ الی ۱۸) طراحی تحت بررسی (یا DUT<sup>۱</sup>) به همراه سیگنالهای لازم برای تست و آزمایش آن اعلان می‌شوند. در این شکل، این سیگنالها به نامهای `a_tb` و `b_tb` (ورودی‌ها) و `y_tb` (خروجی) می‌باشند. در بدنهٔ معماری (خطوط ۱۹ الی ۳۲) ابتدا DUT نمونه‌سازی می‌شود (خطوط ۲۱-۲۲)، سپس محرک‌ها تولید می‌شوند (خطوط ۲۵-۲۶) و در انتهای یک بخش اختیاری کد نشان داده شده است (خطوط ۲۷ الی ۳۱) که زمانی استفاده می‌شود که کاربر بخواهد شبیه‌ساز مقادیر به دست آمده برای `y_tb` را با مقادیر مورد انتظار برای آن مقایسه کند (یعنی تصدیق خودکار<sup>۲</sup>). توجه کنید که ورودی‌ها به کمک دستور AFTER تولید شده‌اند (خطوط ۲۴ و ۲۵) اما به جای آن از دستور WAIT FOR نیز می‌توان استفاده کرد. در برخی مواقع، تولید محرک‌ها و بررسی و تست مقادیر خروجی به طور همزمان و توامان انجام می‌شود (همان طور که بعداً در مثال‌های ۷-۱۰ و ۸-۱۰-۹-۱۰ انجام خواهد شد).

در مابقی بخش‌های این فصل از قالب کلی نشان داده شده در شکل ۹-۱۰ استفاده خواهد شد.

<sup>1</sup> Design Under Test

<sup>2</sup> Automated Verification

```

1  -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  -----
5  ENTITY mycircuit_tb IS
6      GENERIC (... );
7  END ENTITY;
8  -----
9  ARCHITECTURE testbench OF mycircuit_tb IS
10   ----DUT declaration:-----
11     COMPONENT mycircuit IS
12         PORT (a, b: IN STD_LOGIC;
13                 y: OUT STD_LOGIC);
14     END COMPONENT;
15   ----Signal declarations:-----
16     SIGNAL a_tb: STD_LOGIC := '1';
17     SIGNAL b_tb: STD_LOGIC := '0';
18     SIGNAL y_tb: STD_LOGIC;
19 BEGIN
20   ----DUT instantiation:-----
21     dut: mycircuit
22         PORT MAP (a=>a_tb, b=>b_tb, y=>y_tb);
23   ----Stimuli generation:-----
24     a_tb <= '0' AFTER 25ns, ...;
25     b_tb <= '1' AFTER 40ns, ...;
26   ----Output verification (optional):
27     PROCESS
28     BEGIN
29         WAIT FOR ...
30         ASSERT (y_tb=y) ...
31     END PROCESS;
32 END ARCHITECTURE;
33 -----

```

Architecture

Component and signal declarations

Component instantiation

Stimulus generation (with AFTER or WAIT FOR)

Output verification (optional)

**Figure 10.9**  
VHDL template for testbenches.

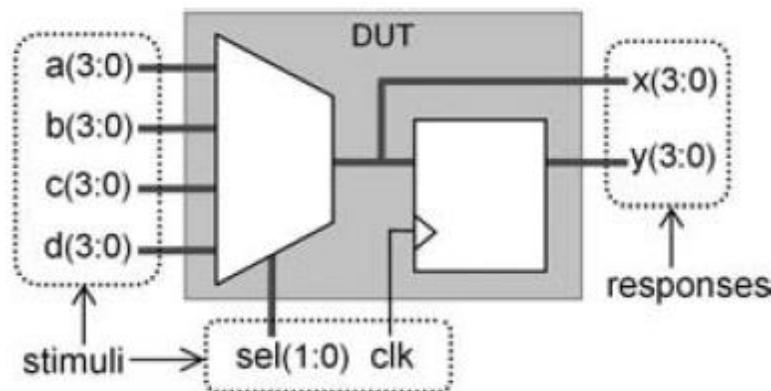
## ۱۰-۱ بستر آزمایش از نوع I ( شبیه‌سازی عملکردی دستی<sup>۱</sup> )

در این بخش نحوه تولید یک بستر آزمایش از نوع I به طور کامل نشان داده می‌شود. از شکل ۱۰-۲-ج به یاد بیاورید که در حالت نوع I تحلیل از نوع «عملکردی» (یعنی تاخيرهای مدار در نظر گرفته نمی‌شود) و «دستی» (یعنی پاسخ را دستی و از طریق بصری ارزیابی می‌کنیم) است.

### Example 10.4: Type I Testbench for a Registered Mux

This example concerns the development of a *manual functional simulation*. The design to be tested is that seen in section 10.5 (repeated in figure 10.10, with the stimuli and responses highlighted). Develop a Type I testbench to test this circuit using the same waveforms of figure 10.6a.

<sup>1</sup> Manual Functional Simulation

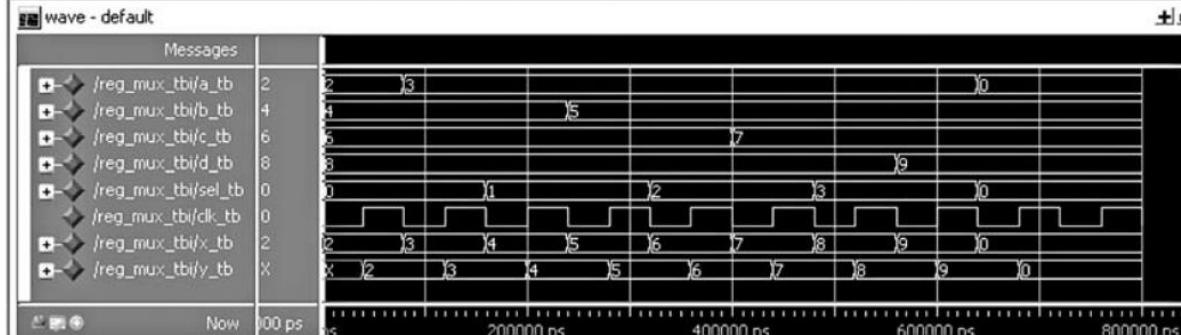


**Figure 10.10**  
Registered multiplexer of example 10.4.

**Solution** The *design* file for this circuit (*reg\_mux.vhd*) was already prepared in section 10.5. We need now to write the *test* file (to be saved as *reg\_mux\_tb.vhd*), after which the simulation can be performed.

A VHDL code for the test file is presented below. As usual, the entity (lines 5–6) is empty (GENERIC was not needed in this example). The declarative part of the architecture (lines 9–24) contains a declaration for the design to be tested (DUT = *reg\_mux.vhd* of section 10.5) and for the stimuli. Finally, the architecture body (lines 25–46) contains the DUT instantiation and the stimuli generation.

Simulation results, using ModelSim, are depicted in figure 10.11 (just follow the procedure in sections D.1 to D.3 of appendix D). Note that the waveforms are the same as those in figure 10.6a. Just to ease the inspection, the waveforms were displayed using integers instead of binary values.



**Figure 10.11**  
Simulation results from example 10.4.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 -----
5 ENTITY reg_mux_tb IS
6 END ENTITY;
7 -----
8 ARCHITECTURE testbench OF reg_mux_tb IS
9     ----DUT declaration:-----
10    COMPONENT reg_mux IS
11        PORT (a, b, c, d: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
12                sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
13                clk: IN STD_LOGIC;
14                x, y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
15    END COMPONENT;
16    ----Signal declarations:-----
17    SIGNAL a_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0010";
18    SIGNAL b_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0100";
19    SIGNAL c_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
20    SIGNAL d_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "1000";
21    SIGNAL sel_tb: STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";
22    SIGNAL clk_tb: STD_LOGIC := '0';
23    SIGNAL x_tb: STD_LOGIC_VECTOR(3 DOWNTO 0);
24    SIGNAL y_tb: STD_LOGIC_VECTOR(3 DOWNTO 0);
25 BEGIN
26     ----DUT instantiation:-----
27     dut: reg_mux PORT MAP (
28         a => a_tb,
29         b => b_tb,
30         c => c_tb,
31         d => d_tb,
32         clk => clk_tb,
33         sel => sel_tb,
34         x => x_tb,
35         y => y_tb);
36     ----Stimuli generation:-----
37     clk_tb <= NOT clk_tb AFTER 40ns;
38     a_tb <= "0011" AFTER 80ns, "0000" AFTER 640ns;
39     b_tb <= "0101" AFTER 240ns;
40     c_tb <= "0111" AFTER 400ns;
41     d_tb <= "1001" AFTER 560ns;
42     sel_tb <= "01" AFTER 160ns,
43             "10" AFTER 320ns,
44             "11" AFTER 480ns,
45             "00" AFTER 640ns;
46 END ARCHITECTURE;
47 -----

```

## ۱۰-۹ بستر آزمایش از نوع II

نوع II مشابه با نوع I است اما با این تفاوت که تاخیر انتشارهای داخلی DUT نیز در نظر گرفته می‌شود. با توجه به این که این شبیه‌سازی دربرگیرنده زمان بوده اما ارزیابی نتایج همچنان دستی انجام می‌شود، این نوع شبیه‌سازی را «شبیه‌سازی دستی زمان‌بندی»<sup>۱</sup> گویند.

### Example 10.5: Type II Testbench for a Registered Mux

Modify the Type I simulation of example 10.4 in order to produce a Type II simulation.

**Solution** As explained in section 10.2, besides the two files prepared by the user (*reg\_mux.vhd* and *reg\_mux\_tb.vhd*), two other files (postsynthesis and SDF), produced by the synthesizer, are also needed now.

For example, if using Quartus II to synthesize the design and ModelSim to simulate it, follow the procedure in sections D.1, D.2, and D.4 of appendix D. After the simulation is finished, the waveforms of figure 10.12 will be displayed. Note that the only difference with respect to the simulation results in figure 10.11 is the inclusion of propagation delays. For example, the cursor shows a transition at 368.34ns, in response to a clock transition at 360ns, hence with a propagation delay of 8.34ns.

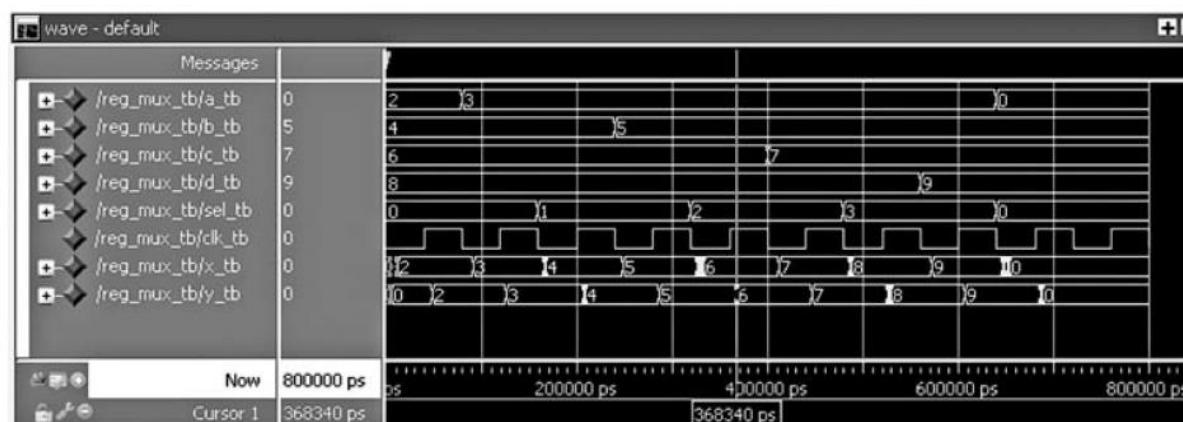


Figure 10.12  
Simulation results from example 10.5.

## ۱۰-۱۰ بستر آزمایش از نوع III

در این نوع بستر آزمایش نتایج به طور خودکار توسط یک برنامه VHDL با مقادیر مورد انتظار مقایسه می‌شوند. همچنین تاخیرهای داخلی مدار در نظر گرفته نمی‌شوند. لذا به آن «شبیه‌سازی عملکردی خودکار»<sup>۲</sup> گفته می‌شود. لذا نوع III بخشی از نوع IV محسوب می‌شود همان‌طور که نوع I بخشی از نوع II بود. در نتیجه، بعد از فهم نوع IV، نوشتن و فهم نوع III سرراست خواهد بود.

<sup>1</sup> Manual Timing Simulation

<sup>2</sup> Automated Functional Simulation

## ۱۱-۱۱ بستر آزمایش از نوع IV

این نوع کاملترین و الیته پیچیده ترین نوع بستر آزمایش است. به این نوع، « شبیه‌سازی خودکار زمان بندی »<sup>۱</sup> و یا « بستر آزمایش کامل »<sup>۲</sup> نیز گفته می‌شود.

### Example 10.6: Type IV Testbench for a Registered Mux

Develop a Type IV testbench to test the registered output ( $y$ ) of the multiplexer seen in section 10.5. Assume that the expected maximum propagation delay between the clock and  $y$  ( $t_{pCQ}$ ) is 10ns. Use the same stimuli employed in the previous examples (from figure 10.6), so the expected output values are those depicted in figure 10.13, where the propagation delays are highlighted by gray shades.

**Solution** A VHDL code for a full bench to test this circuit is presented below. The only major difference with respect to the test file in example 10.4 is the inclusion of the optional code section that automatically checks the output values against expected results (lines 42–68). The expected output signal (called *expected*) was declared in line 27 and then constructed in lines 44–52. The comparison is made in the process of lines 54–68 by means of the ASSERT statement (see details in section 9.2), and occurs every 10ns (line 56). Note in line 7 that the acceptable propagation delay was specified as a generic parameter.

The IF statement in lines 57–67 causes the process to be run until the time reaches 800ns. If a mismatch is detected, the first ASSERT (lines 58–62) terminates the simulation and issues the message “Mismatch at  $t = \text{xxx}$   $y_{\text{tb}} = \text{xxx}$   $y_{\text{exp}} = \text{xxx}$ ”, where  $\text{xxx}$  represents the corresponding actual value. If no errors are found, the simulation is terminated when the time reaches 800ns, with the second ASSERT (lines 64–66) forcing the message “No error found ( $t = 800000$  ps)” to be issued.

Looking at the simulation results obtained in example 10.5 (figure 10.12), delays under 10ns are observed, so we do not expect this simulation to find any data mismatches. However, if the value of the parameter in line 7 is reduced, a problem will eventually occur, which is one way of estimating the limiting speed of this circuit.

To simulate this design (with ModelSim), just follow the procedure in sections D.1, D.2, and D.4 of appendix D. Try to play with  $t_p$  and *expected* in order to better understand the automated comparison process.



<sup>1</sup> Automated Timing Simulation

<sup>2</sup> Full bench

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_unsigned.all;
5 -----
6 ENTITY reg_mux_tb IS
7     GENERIC (tp: TIME := 10ns);
8 END ENTITY;
9 -----
10 ARCHITECTURE testbench OF reg_mux_tb IS
11     ----DUT declaration:-----
12     COMPONENT reg_mux IS
13         PORT (a, b, c, d: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
14                 sel: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
15                 clk: IN STD_LOGIC;
16                 x, y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
17     END COMPONENT;
18     ----Signal declarations:-----
19     SIGNAL a_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0010";
20     SIGNAL b_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0100";
21     SIGNAL c_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
22     SIGNAL d_tb: STD_LOGIC_VECTOR(3 DOWNTO 0) := "1000";
23     SIGNAL sel_tb: STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";
24     SIGNAL clk_tb: STD_LOGIC := '0';
25     SIGNAL x_tb: STD_LOGIC_VECTOR(3 DOWNTO 0);
26     SIGNAL y_tb: STD_LOGIC_VECTOR(3 DOWNTO 0);
27     SIGNAL expected: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
28 BEGIN
29     ----DUT instantiation:-----
30     dut: reg_mux PORT MAP (a_tb, b_tb, c_tb, d_tb, sel_tb,
31                             clk_tb, x_tb, y_tb);
32     ----Stimuli generation:-----
33     clk_tb <= NOT clk_tb AFTER 40ns;
34     a_tb <= "0011" AFTER 80ns, "0000" AFTER 640ns;
35     b_tb <= "0101" AFTER 240ns;
36     c_tb <= "0111" AFTER 400ns;
37     d_tb <= "1001" AFTER 560ns;
38     sel_tb <= "01" AFTER 160ns,
39                 "10" AFTER 320ns,
40                 "11" AFTER 480ns,

```

```

41           "00" AFTER 640ns;
42   ---Output verification:-----
43   ---(i)Generate template:
44   expected <= "0010" AFTER 40ns+tp,
45           "0011" AFTER 120ns+tp,
46           "0100" AFTER 200ns+tp,
47           "0101" AFTER 280ns+tp,
48           "0110" AFTER 360ns+tp,
49           "0111" AFTER 440ns+tp,
50           "1000" AFTER 520ns+tp,
51           "1001" AFTER 600ns+tp,
52           "0000" AFTER 680ns+tp;
53   ---(ii)Make comparison:
54   PROCESS
55   BEGIN
56       WAIT FOR tp;
57       IF (NOW<800ns) THEN
58           ASSERT (y_tb=expected)
59               REPORT "Mismatch at t=" & TIME'IMAGE(NOW) &
60                   " y_tb=" & INTEGER'IMAGE(conv_integer(y_tb)) &
61                   " y_exp=" & INTEGER'IMAGE(conv_integer(expected))
62               SEVERITY FAILURE;
63       ELSE
64           ASSERT FALSE
65               REPORT "No error found (t=" & TIME'IMAGE(NOW) & ")"
66               SEVERITY NOTE;
67       END IF;
68   END PROCESS;
69 END ARCHITECTURE;
70 -----

```

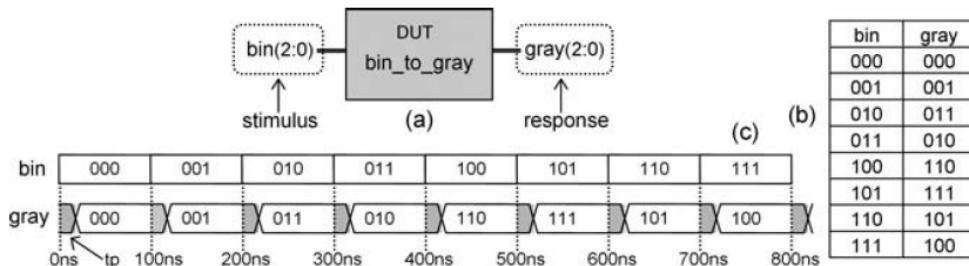
## ۱۰-۱۲ بستر آزمایش با استفاده از نوع RECORD

نوع RECORD برای تست و آزمایش کدهایی مناسب است که مقادیر ورودی-خروجی آنها به صورت یک جدول جستجو (البته نه چندان بزرگ) در داخل کد مشخص شده باشد. برای استفاده از داده‌های با حجم زیاد استفاده از فایل توصیه می‌شود.

**Example 10.7: Type IV Testbench with a Record Type**

This example concerns the development of a full bench for a binary-to-gray converter (figure 10.14a) with the input stimuli and expected results stored in a table in the test file itself, constructed with the help of RECORD. Recall that a gray code is one in which neighboring words differ by just one bit. To ease the analysis, just three bits will be used. Their values are listed in figure 10.14b. This exercise is divided into three parts:

- 1) Write a VHDL code for the *design* file. In it, use a closed-form expression to make the binary-to-gray conversion.
- 2) Write a VHDL code for the *test* file. In it, to illustrate the use of tables and RECORD, enter the data (stimuli plus expected results) using a lookup table instead of an expression.
- 3) Simulate the design, for *timing* analysis. Change the input data (*bin*) every 100ns, then give some time ( $t_p$ ) for the output (*gray*) to settle. Next, make the comparison between the actual and the expected results. This approach is illustrated in figure 10.14c, where the gray areas highlight the output settling period (maximum propagation delay allowed).



**Figure 10.14**  
Binary-to-gray converter.

**Solution** Part (1): A VHDL code for the design file, under the title *bin\_to\_gray* (line 5), is shown below. Only concurrent code is employed (lines 13–16), with closed-form expressions used to make the conversion.

```

1 -----
2 LIBRARY ieee;
3 USE ieee. std_logic_1164.all;
4 -----
5 ENTITY bin_to_gray IS
6   GENERIC (N: NATURAL := 3);
7   PORT (bin: IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
8         gray: OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0));
9 END ENTITY;
10 -----
11 ARCHITECTURE bin_to_gray OF bin_to_gray IS
12 BEGIN
13   gray(N-1) <= bin(N-1);
14   gen: FOR i IN 1 TO N-1 GENERATE
15     gray(N-1-i) <= bin(N-i) XOR bin(N-1-i);
16   END GENERATE;
17 END ARCHITECTURE;
18 -----

```

Part (2): A full testbench for the binary-to-gray converter is presented next, under the title *bin\_to\_gray\_tb* (line 5). As before, the entity (lines 5–8) has no PORT declarations; it contains only generic parameters representing the distance (*period* = 100ns) between the data samplings and the waiting time interval (*tp* = 15ns) before the output is read.

The declarative part of the architecture (lines 11–26) contains a component (DUT) declaration followed by signal, type, and constant declarations. A RECORD (section 3.14), called *data\_pair* and with two STD\_LOGIC\_VECTOR values named *col1* and *col2*, is created in lines 19–22. Next, in line 23, a type called *table* is declared as having eight such pairs. Finally, a constant called *templates*, of type *table*, is specified in lines 24–26, containing the values listed in figure 10.14b.

The code proper (lines 27–45) contains two parts. The first part (line 29) consists of the DUT instantiation, while the second part (lines 31–44) is a process responsible for generating the stimuli and also for checking the results. Observe that, contrary to the previous example, here the stimulus generation and the output verification are done together (a stimulus is applied in line 34, then some time is given in line 35 for the output to settle, and finally a comparison is made by the ASSERT statement in lines 36–38). If a mismatch is detected, the simulation is terminated and the following message is issued (assuming *i* = 5): “Mismatch at iteration *i* = 5”. On the other hand, if no mismatch occurs, the simulator eventually reaches line 41, whose ASSERT statement forces the message “No error found!” to be displayed. (For details on ASSERT and the usage of IMAGE, see section 9.2.)

```

1 -----
2 LIBRARY ieee;
3 USE ieee. std_logic_1164.all;
4 -----
5 ENTITY bin_to_gray_tb IS
6     GENERIC (period: TIME := 100ns;
7               tp: TIME := 15ns);
8 END ENTITY;
9 -----
10 ARCHITECTURE testbench OF bin_to_gray_tb IS
11     ----DUT declaration:-----
12     COMPONENT bin_to_gray IS
13         PORT (bin: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
14                 gray: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
15     END COMPONENT;
16     ----Signal declarations:-----
17     SIGNAL b: STD_LOGIC_VECTOR(2 DOWNTO 0); --binary in
18     SIGNAL g: STD_LOGIC_VECTOR(2 DOWNTO 0); --gray out
19     TYPE data_pair IS RECORD
20         col1: STD_LOGIC_VECTOR(2 DOWNTO 0);
21         col2: STD_LOGIC_VECTOR(2 DOWNTO 0);
22     END RECORD;

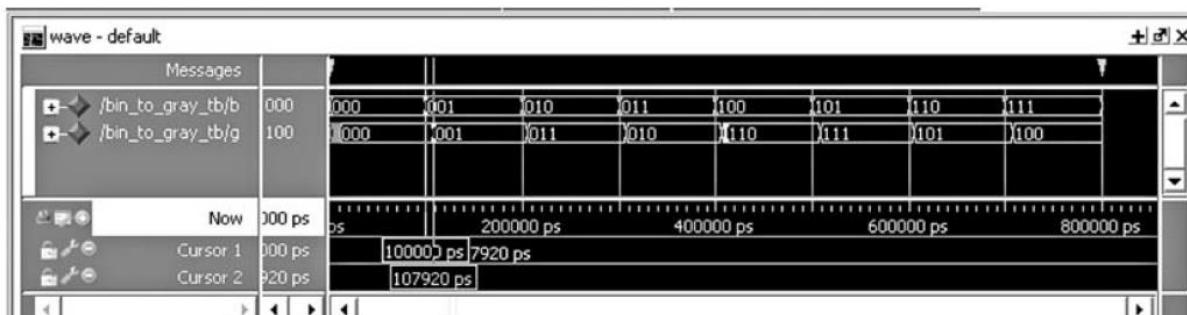
```

```

23      TYPE table IS ARRAY (1 TO 8) OF data_pair;
24      CONSTANT templates: table := (
25          ("000", "000"), ("001", "001"), ("010", "011"), ("011", "010"),
26          ("100", "110"), ("101", "111"), ("110", "101"), ("111", "100"));
27      BEGIN
28          ---DUT instantiation:-----
29          dut: bin_to_gray PORT MAP (bin => b, gray => g);
30          ---Stimuli generation and comparison:-----
31      PROCESS
32      BEGIN
33          FOR i IN table'RANGE LOOP
34              b <= templates(i).col1;
35              WAIT FOR tp;
36              ASSERT g=templates(i).col2
37                  REPORT "Mismatch at iteration=" & INTEGER'IMAGE(i)
38                  SEVERITY FAILURE;
39              WAIT FOR period-tp;
40          END LOOP;
41          ASSERT FALSE
42              REPORT "No error found!"
43              SEVERITY NOTE;
44      END PROCESS;
45  END ARCHITECTURE;
46  -----

```

Part (3): Since it is a timing simulation, we must include in the design the postsynthesis and SDF files generated by the synthesizer. This procedure was already described in the previous example. If using ModelSim, just follow the procedure in sections D.1, D.2, and D.4 of appendix D. Simulation results (from ModelSim) are depicted in figure 10.15. Observe at the cursors' feet that the input-output propagation delay is 7.92ns (the device used in this design was a Cyclone II FPGA).



**Figure 10.15**  
Simulation results from example 10.7.

### ۱۰- ۱۳- بستر آزمایش با استفاده از فایل داده

در صورت استفاده از داده‌هایی با حجم زیاد در بستر آزمایش، استفاده از فایلها توصیه می‌شود. در این بخش نشان داده می‌شود که چگونه یک کد VHDL می‌تواند در طول شبیه‌سازی نوع IV با یک فایل کار کند.

دانشگاه صنعتی شاهرود

**Example 10.8: Type IV Testbench with a Data File**

This example concerns the development of a full bench with the in-out data stored in a file. Redo example 10.7, this time with the stimuli (*bin*) and expected values (*gray*) stored in a file (call it *template.txt*). The design file is obviously still the same.

**Solution** A test file for a Type IV testbench for this exercise is presented below. Because a data file will now be used, the *textio* package (line 4) was included in the package declarations. The entity (lines 6–9) is similar to that in the previous example.

As usual, the declarative part of the architecture (lines 12–20) contains the DUT declaration followed by signal declarations. In the latter, not only three signals (*b*, *g*, and *gtest*, needed to represent *bin* and *gray*) are declared (lines 18–19), but also a file (line 20), called *template.txt*, identified by *f* and opened in *read* mode.

Again, as in the previous design, the architecture body (lines 21–55) consists of the DUT instantiation (line 23) followed by a process (lines 25–54) for stimulus generation plus output verification. Note that the overall file-reading process is relatively similar to that seen earlier in section 10.4.

In the declarative part of the process (lines 26–29), five variables are specified. The first (*l*) is used to store a file line during the file-read procedure. The second (*good\_value*) is used to check whether the type of the read value matches the type of the object to which the value is being assigned. The third (*space*) is just to suppress the line space between *bin* and *gray*. Finally, the last two (*bfile*, *gfile*) receive the values of *bin* and *gray* read from the file.

In the process body (lines 30–54), a file line is read and assigned to *l* (line 32), then the first value is separated from *l* and assigned to *bfile* (line 33), and is subsequently tested using ASSERT (lines 34–36). Because the procedure READ does not accept the type STD\_LOGIC\_VECTOR (see section 10.4), a type-conversion function was employed to convert *bfile* (BIT\_VECTOR) into *b* (STD\_LOGIC\_VECTOR) (see type-conversion functions in the table of figure 3.10). If an improper value is found in the file, the message “Improper value for ‘bin’ in file!” is issued (line 35) and the simulation is terminated (line 36). A similar construction is used for the value assigned to *gtest*. Finally, some time is given (line 44) for the output to settle, after which the expected value, *gtest*, is compared against the value attained from the circuit, *g*. If they do not match, the message “Data mismatch!” is issued (line 46) and the simulation is terminated (line 47). Only when no errors are found does the simulator reach line 50, where the last ASSERT forces the message “No errors found!” to be issued, concluding the simulation.

A final remark regards the *good\_value* tests. If, for example, the check in lines 34–36 is deleted, and an improper value is found in the file, the simulator will still report the problem. But because that will cause just the loop to be terminated, lines 50–52 will still be executed, giving the user the false information that no error was found (indeed, the circuit might be correct; the problem is that the tests were incomplete).

Simulation results, using ModelSim, are depicted in figure 10.16. To check them, just follow the procedure in sections D.1, D.2, and D.4 of appendix D.

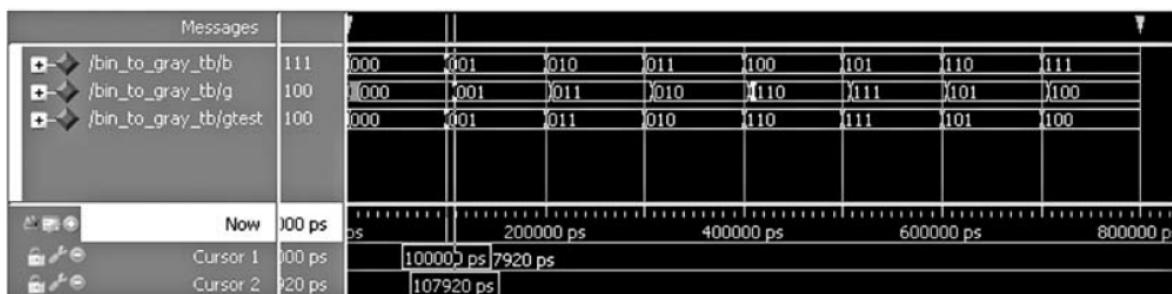


Figure 10.16  
Simulation results from example 10.8.

```

1 -----
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE std.textio.all;
5 -----
6 ENTITY bin_to_gray_tb IS
7     GENERIC (period: TIME := 100ns;
8             tp: TIME := 15ns);
9 END ENTITY;
10 -----
11 ARCHITECTURE testbench OF bin_to_gray_tb IS
12     ----DUT declaration:-----
13     COMPONENT bin_to_gray IS
14         PORT (bin: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
15                 gray: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
16     END COMPONENT;
17     ----Signal declarations:-----
18     SIGNAL b: STD_LOGIC_VECTOR(2 DOWNTO 0);
19     SIGNAL g, gtest: STD_LOGIC_VECTOR(2 DOWNTO 0);
20     FILE f: TEXT OPEN READ_MODE IS "template.txt";
21 BEGIN
22     ----DUT instantiation:-----
23     dut: bin_to_gray PORT MAP (bin => b, gray => g);
24     ----Output verification:-----
25     PROCESS
26         VARIABLE l: LINE;
27         VARIABLE good_value: BOOLEAN;
28         VARIABLE space: CHARACTER;
29         VARIABLE bfile, gfile: BIT_VECTOR(2 DOWNTO 0);
30     BEGIN
31         WHILE NOT ENDFILE (f) LOOP
32             READLINE(f, l);

```

```

33      READ(l, bfile, good_value);
34      ASSERT (good_value)
35          REPORT "Improper value for 'bin' in file!"
36          SEVERITY FAILURE;
37      b <= to_stdlogicvector(bfile);
38      READ(l, space);
39      READ(l, gfile, good_value);
40      ASSERT (good_value)
41          REPORT "Improper value for 'gray' in file!"
42          SEVERITY FAILURE;
43      gtest <= to_stdlogicvector(gfile);
44      WAIT FOR tp;
45      ASSERT (gtest=g)
46          REPORT "Data mismatch!"
47          SEVERITY FAILURE;
48      WAIT FOR period-tp;
49      END LOOP;
50      ASSERT FALSE
51          REPORT "No errors found!"
52          SEVERITY NOTE;
53      WAIT;
54  END PROCESS;
55 END ARCHITECTURE;
56 -----

```



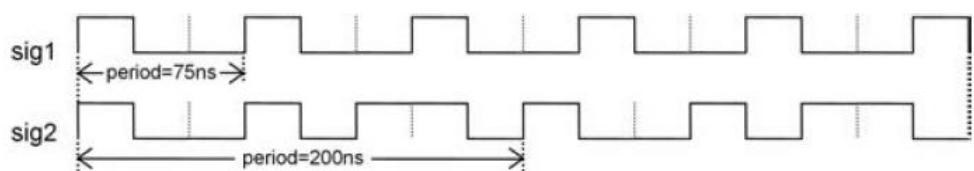


Figure 10.17

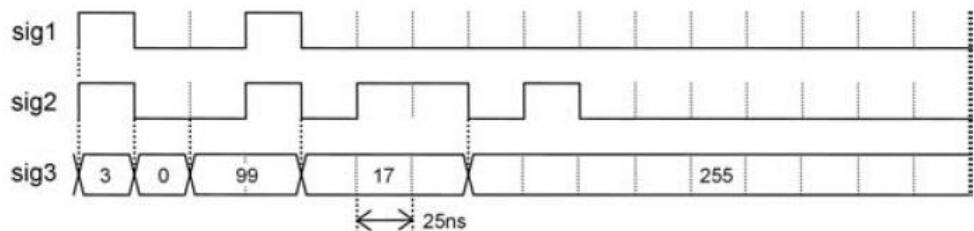


Figure 10.18

time	inp	outp
0ns	1	100
50ns	2	110
100ns	0	011
150ns	3	111
200ns	4	010
250ns	6	101
300ns	7	000
350ns	5	001

Figure 10.19

پ سامانه

مطالب کمکی مربوط به فصل ۳ و مفید در فصل فعلی:

### 3.20 ACCESS Types

All data types described so far have a well-defined, known structure. However, in models for simulation with a high level of abstraction, the data structure might sometimes not be known in advance or might not be static (varying size). To deal with such situations, VHDL provides a type class called *access types*, which acts as a *pointer* to the location in memory where the data is located rather than representing the data directly (so unknown and dynamic data structures can be used).

Access types can only be used in sequential code, and only variables can be of that type. In the example below, an access type called *int\_pointer* is created, which points to objects of type INTEGER. Next, a variable of that type, called *pointer*, is declared.

```
TYPE int_pointer IS ACCESS INTEGER;
VARIABLE pointer: int_pointer := NULL;
-----
pointer := NEW INTEGER'(16);
pointer.ALL := 0;
```

The variable *pointer* above points to integers stored in memory. However, its initial (default) value is NULL (that is, it points nowhere), which must be modified by the code. In the third line, the predefined function NEW is called to provide memory space for 16 integers, returning an access value (a pointer) to the allocated memory, which is assigned to *pointer*. Finally, in the fourth line, a value (0, in this example) is assigned to the object pointed to by *pointer*.

This description is just to give an idea on how access types work. Even though very complex models can be built with such a pointing mechanism, its regular use is for simulation, and only for very particular data structures, so a hardware designer might never need it.

### 3.21 FILE Types

File types were briefly described in section 3.2. Because files are particularly important for simulation, further details will be given in chapter 10, which deals specifically with that subject.

### 3.22 VHDL 2008

With respect to the material covered in this chapter, the main additions specified in VHDL 2008 are those seen in sections 3.3, 3.5, 3.6, and 3.8. Just as a reminder, the following occurred with the standardized VHDL packages:

برای سلامتی رهبر معظم انقلاب و  
تعجیل در ظهور ولی عصر (عج)  
صلوات

دانشگاه صنعتی شاهمرد