

آموزش VHDL

درس طراحی مدارات FPGA

دانشگاه صنعتی شاهرود

دکتر هادی گرایلو

۱۳۹۷

بخش دوم

فصل نهم

مدل‌سازی ساختاری در VHDL

همان طور که قبلاً اشاره شد، معمولاً سه رویکرد نوشتن کد VHDL وجود دارد: مدل‌سازی جریان داده، مدل‌سازی رفتاری، و مدل‌سازی ساختاری. تا به حال دو رویکرد مدل‌سازی جریان داده و مدل‌سازی رفتاری را بررسی کرده‌ایم. در این فصل به رویکرد مدل‌سازی ساختاری می‌پردازیم.

با پیچیده‌تر شدن طراحی‌های دیجیتال، احتمال این که بتوان طراحی را به کمک هر کدام از سه مدل VHDL پیاده‌سازی کرد، کمتر می‌شود. ما قبلاً این مطلب و ویژگی را در مورد کارکردن با FSM‌ها که دستورات فرآیند (مدل‌سازی رفتاری) را با دستورات تخصیص انتخابی سیگنال ترکیب کردیم، دیده بودیم؛ نتیجه، یک مدل VHDL ترکیبی (هیبرید) بود. مدل‌سازی ساختاری شبیه به یک مدل ترکیبی است. اکثر طراحی‌های پیچیده را می‌توان از نوع مدل‌های ساختاری در نظر گرفت.

مدل‌سازی ساختاری معادل با همان رویکرد ماثولار در کدنویسی با زبانهای برنامه‌نویسی سطح بالا است. رویکرد طراحی ماثولار در VHDL مستقیماً از طراحی سلسه‌مراتبی که در طراحی مدارات پیچیده استفاده می‌شود، پشتیبانی می‌کند. مزایای طراحی ماثولار در VHDL معادل و مشابه با مزایای طراحی ماثولار (یا طراحی شی‌ءگرایی) برای زبانهای برنامه‌نویسی سطح بالا است. طراحی‌های ماثولار با بسته‌بندی توابع سطح پایین در ماثولارها، موجب افزایش قابلیت فهم می‌شود. این ماثولارها قابل استفاده مجدد در دیگر طراحی‌ها بوده و بنابراین، موجب صرفه‌جویی در وقت طراح می‌شوند زیرا دیگر نیازی به «اختراع و آزمایش مجدد چرخ» نخواهند داشت!.

۱-۹ ماثولار شدن VHDL به کمک کامپونت‌ها

مهمترین ابزار در زبانهای سطح بالا (مانند C) برای تحقق ماثولار شدن، تابع است. در دیگر زبانهای برنامه‌نویسی نیز از ابزارهایی مانند متدها، روال‌ها و سابروتین‌ها استفاده می‌شود. مراحل انجام کار در زبان C عبارتند از: (۱) تابعی که قصد نوشتن آن را دارید، نامگذاری کنید (اعلان تابع). (۲) وظیفه و کار تابع را کدنویسی کنید (بدنه‌ی تابع). (۳) اجازه دهید که برنامه از وجود و قابلیت استفاده از تابع مطلع شود (الگو^۱). (۴) تابع را از داخل بدنه‌ی اصلی برنامه «فراخوانی» کنید.

^۱ Prototype

در VHDL مارژولار شدن از طریق استفاده از بسته‌ها^۱، کامپوننت‌ها^۲، و توابع^۳ تحقق می‌یابد. در بخش‌های بعدی، نحوه استفاده از کامپوننت‌ها نشان داده خواهد شد.

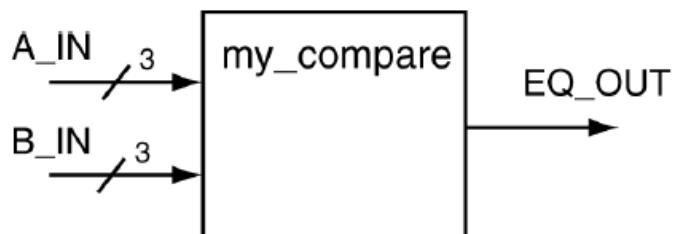
مراحل استفاده از کامپوننت‌ها در VHDL بدین ترتیب است: (۱) مارژول را که تصمیم به توصیف آن دارد، نامگذاری کنید («موجودیت»). (۲) کاری که مارژول انجام می‌دهد را توصیف کنید («معماری»). (۳) اجازه دهید که برنامه از وجود مارژول مطلع شود («اعلان کامپوننت»). (۴) مارژول را در کد خود استفاده کنید («نمونه‌سازی»^۴ یا نگاشت^۵). شباختهایی که بین دو رویکرد اخیر (در زبانهای C و VHDL) بیان شد، در جدول ۹-۱ فهرست شده است.

| C programming language | VHDL |
|--|------------------------------------|
| Describe function interface | The entity |
| Describe what the function does (coding) | The architecture |
| Provide a function prototype to main program | Component declaration |
| Call the function from main program | Component instantiation or mapping |

Table 9.1: Similarities between modules in C and VHDL.

حال اجازه دهید که این اصول را در قالب یک مثال عملی استفاده کنیم.

مثال ۲۲: با استفاده از یک مدل ساختاری VHDL یک مقایسه‌گر سه‌بیتی طراحی کنید. اتصال (یا واسطه‌ی) این مدار در دیاگرام زیر نشان داده شده است.



¹ Packages

² Components

³ Functions

⁴ Instantiation

⁵ Mapping

حل: مقایسه‌گر یکی از مدارات ترکیبی کلاسیک است که هر مهندس طراح دیجیتال باید در جایی از طول تحصیلات خود آن را طراحی و تحلیل کند. در اینجا نسخه‌ای از این مدار به کمک گیتهای گسته پیاده‌سازی و ارائه می‌شود. این مدار در شکل ۹-۱ نشان داده شده است. راه حلی که در اینجا ارائه می‌شود، اساساً مثالی از یک مدل ساختاری VHDL بوده و لزوماً بیانگر بهترین راه حل نیست.

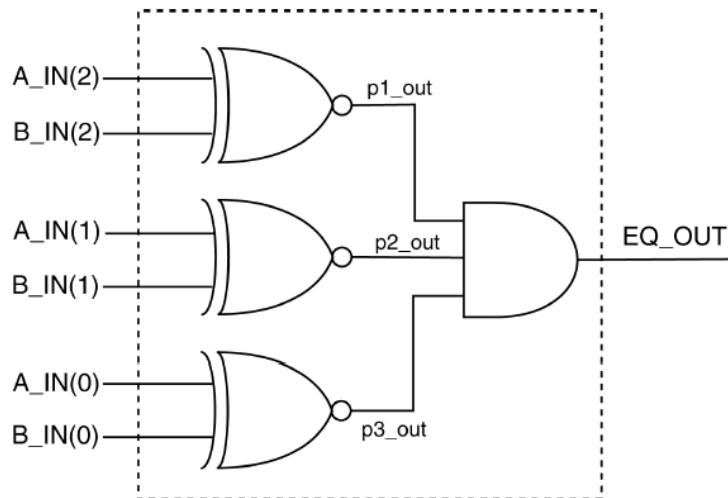


Figure 9.1: Discrete gate implementation of a 3-bit comparator.

رویکردی که در این راه حل در پیش گرفته شده است، مدل کردن هر یک از گیتهای گسته به عنوان یک بلوک مستقل و جداگانه است؛ درست است که این بلوک‌ها یک گیت ساده هستند اما قواعد و ملزمات مربوط به اتصال همانهایی است که در حالت یک مدار پیچیده باید در نظر گرفت (یعنی مستقل از سادگی یا پیچیدگی مدار داخل هر بلوک است).

مدار نشان داده شده در شکل ۹-۱ شامل برخی اطلاعات اضافه مرتبط با پیاده‌سازی ساختاری VHDL است. اول این که مرز خط‌چین نشان‌دهنده‌ی «موجودیت سطح رو»^۱ است؛ بنابراین، سیگنالهایی که این مرز را قطع می‌کنند باید در قسمت اعلان موجودیت ذکر شوند. دوم این که هر یک از سیگنالهای داخلی نامی داده شده است؛ سیگنال داخلی سیگنالی است که مرز خط‌چین را قطع نکرده است. نام گذاری سیگنالهای داخلی در پیاده‌سازی ساختاری VHDL، کاری لازم و ضروری است زیرا این سیگنالها را باید به زیرماژول‌های مختلف موجود در درون طرح (جایی داخل معماری) تخصیص داد (لذا به نام آنها نیاز داریم).

¹ Top-level entity

اولین بخش از برنامه‌ی جواب به پیاده‌سازی موجودیت و معمازی گیت‌های مختلف نشان داده شده در شکل ۹-۱ می‌پردازد. ما باید حداقل یک گیت XOR و یک گیت AND سه-ورودی را تعریف کنیم. درست است که در داخل دیاگرام از سه گیت XOR استفاده شده است اما کافی است تنها یک بار این گیت را تعریف کنیم. رویکرد مژولار در VHDL این امکان را فراهم می‌آورد که از یک تعریف داده شده برای یک مدار چندین بار بتوان استفاده‌ی مجدد کرد. تعاریف مورد نیاز در برنامه‌ی ۹-۱ نشان داده شده است.

Listing 9.1: Entity and Architecture definitions for discrete gates.

```
-- Description of XNOR function --
-----
entity big_xnor is
    Port ( A,B : in  std_logic;
           F : out std_logic);
end big_xnor;

architecture ckt1 of big_xnor is
begin
    F <= not ( (A and (not B)) or ( (not A) and B) );
end ckt1;
-----
-- Description of 3-input AND function --
-----
entity big_and3 is
    Port ( A,B,C : in  std_logic;
           F : out std_logic);
end big_and3;

architecture ckt1 of big_and3 is
begin
    F <= ( A and B and C );
end ckt1;
```

برنامه‌ی ۹-۱ جزئیات جدیدی از نقطه نظر VHDL ندارد. اما از این نظر که چگونه از عناصر مداری لیست شده در شکل ۹-۱ به عنوان کامپوننت در یک مدار بزرگتر استفاده می‌شود، حاوی اطلاعات جدیدی است. مراحل پیاده‌سازی یک طرح ساختاری VHDL در گامهای زیر خلاصه شده است. در این قدمها فرض شده است که اعلانات موجودیتهای مربوط به مژولهای داخلی، قبل‌اً انجام شده است.

قدم اول: اعلان موجودیت سطح رو را تولید کنید.

قدم دوم: واحدهای سطح پایین‌تر مورد استفاده در طراحی را اعلان کنید.

قدم سوم: سیگنالهای داخلی مورد نیاز برای اتصال واحدهای طراحی را اعلان کنید.

قدم چهارم: واحدهای طراحی را نمونه‌سازی کنید.

قدم اول: اولین قدم در یک پیاده‌سازی ساختاری مشابه با رویکرد استانداردی است که برای پیاده‌سازی سایر مدارات VHDL استفاده کردیم: (یعنی) «موجودیت». اعلان موجودیت مستقیماً از روی جعبه‌ی خط‌چین نشان داده شده در شکل ۹-۱ قابل استخراج بوده و در برنامه‌ی ۹-۲ نشان داده شده است. به بیان دیگر، سیگنالهایی که با خطوط خط‌چین تقاطع دارند، سیگنالهایی هستند که برای دنیای بیرون از موجودیت شناخته شده و معلوم هستند و (بنابراین) باید در اعلان موجودیت استفاده شوند.

Listing 9.2: Entity declaration for 3-bit comparator.

```
-- Interface description of 3-bit comparator --
entity my_compare is
    Port ( A_IN : in  std_logic_vector(2 downto 0);
           B_IN : in  std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_compare;
```

قدم دوم: قدم بعدی، اعلان آن دسته از واحدهای طراحی است که در مدار استفاده شده‌اند. در ادبیات VHDL منظور از اعلان کردن، فراهم ساختن یک واحد طراحی خاص جهت استفاده در یک طراحی خاص است. توجه کنید که عمل اعلان کردن یک واحد طراحی، برطبق تعریف، مدار شما را به یک طرح سلسله‌مراتبی تبدیل می‌کند. اعلان یک واحد طراحی باعث می‌شود آن واحد آماده‌ی جایگذاری شدن در سلسله‌مراتب طرح شود. واحدهای طراحی در واقع، مازولهایی هستند که در سطوح پایین طراحی مستقر می‌شوند. ما در طراحی خود، نیاز به اعلان دو واحد طراحی مجزا داریم: گیت XOR و یک گیت AND سه ورودی.

در اعلان یک واحد طراحی، دو مساله مطرح است: (۱) چگونه این کار را انجام دهیم، و (۲) آن را کجا قرار دهیم. یک اعلان کامپوننت را می‌توان به منزله‌ی اصلاحیه‌ای بر اعلان موجودیت آن دانست؛ تفاوت در این جا این است که کلمه‌ی کلیدی entity با کلمه‌ی component جایگزین شده و برای خاتمه دادن به کار اعلان باید از end component استفاده کرد. بهترین راه برای انجام این کار، کپی و پیست کردن اعلان موجودیت اولیه و سپس اصلاح آن است. این اعلان کامپوننت باید در بخش اعلان معماری، یعنی بعد از خط مربوط به architecture و قبل از خط begin، قرار داده شود. دو اعلان کامپوننت به همراه اعلان‌های موجودیت متناظر در برنامه‌ی بعدی نشان داده است. برنامه‌ی ۹-۳ اعلان‌های کامپوننت را در کد اصلی VHDL نشان می‌دهد.

```
entity big_xnor is
  Port ( A,B : in std_logic;
          F : out std_logic);
end big_xnor;
```

```
component big_xnor
  Port ( A,B : in std_logic;
          F : out std_logic);
end component;
```

```
entity big_and3 is
  Port ( A,B,C : in std_logic;
          F : out std_logic);
end big_and3;
```

```
component big_and3
  Port ( A,B,C : in std_logic;
          F : out std_logic);
end component;
```

قدم سوم: قدم بعدی اعلان سیگنالهای داخلی مورد استفاده در طراحیتان است. سیگنالهای داخلی مورد نیاز در این طرح، سیگنالهایی هستند که خط‌چین نشان داده شده در شکل ۱-۹ را قطع نکرده‌اند. این سه سیگنال مشابه با متغیرهای محلی مورد استفاده در زبانهای برنامه‌نویسی سطح بالا هستند؛ در این دسته از زبانهای برنامه‌نویسی ابتدا باید متغیرها را اعلان و سپس در برنامه استفاده کرد. **عملأ وظیفه** این سیگنالها، ارتباط‌دهی بین واحدهای طراحی مختلفی است که در طراحی نهایی نمونه‌سازی شده‌اند. در طراحی حال حاضر، سه سیگنال مورد نیاز بوده و به عنوان خروجی‌های گیت‌های XOR و همچنین، ورودی‌هایی به گیت AND استفاده می‌شوند. اعلان سیگنالهای داخلی، مانند همین سه سیگنالی که اشاره شد، در همان بخشی کامپوننت‌ها اعلان می‌شوند (یعنی بعد از خط architecture و قبل از خط begin)، قرار داده می‌شود. توجه کنید که اعلان سیگنالهای میانی مشابه با اعلان سیگنالی است که در بدنی موجودیت (entity body) انجام می‌شود؛ تنها تفاوت این است که اعلان سیگنالهای میانی نیازی به مشخص کردن حالت (یا مُد) ندارد. ما قبلاً در دیگر بخش‌های این کتاب با سیگنالهای میانی کار کرده بودیم. در برنامه‌ی ۹-۳، اعلان سیگنالهای میانی آورده شده است.

قدم چهارم: آخرین قدم، ایجاد (یا تولید) نمونه‌هایی از ماثولهای مورد نیاز و «نگاشت»^۱ این نمونه‌های مختلف از کامپوننت‌ها در داخل بدنی معماری است؛ به این کار، «نمونه‌سازی»^۲ می‌گوییم؛ بنابراین، منظور ما از «نمونه‌سازی»، تولید تعداد دلخواه از نمونه‌های یک یا چند کامپوننت در داخل بدنی معماری (یعنی بعد از خط begin) و اتصال‌دهی مناسب بین این نمونه‌های تولید شده است. فرآیند نگاشت در واقع، اتصال بین نمونه‌های تولید شده را فراهم می‌کند. هر عمل نگاشت، شامل یک نام منحصر‌بفرد به همراه نام موجودیت اصلی مربوط به خود است. اطلاعات اصلی و مهم عمل نگاشت، بعد از کلمه‌ی کلیدی port map می‌آیند. برنامه‌ی ۹-۳ این کار را نشان می‌دهد.

¹ Mapping² Instantiation

یک نکته‌ی کلیدی و مهم و قابل توجه در فرآیند نمونه‌سازی، استفاده از برچسب برای تمام واحدهای طراحی نمونه‌سازی شده است. همیشه باید از برچسب‌ها به عنوان بخشی از فرآیند نمونه‌سازی واحد طراحی استفاده کرد زیرا استفاده از برچسب‌های مناسب، خوانایی و قابلیت فهم کد VHDL شما را افزایش می‌دهد. به بیان دیگر، استفاده از برچسب‌های مناسب، طبیعتِ خود-توضیحی طراحی شما را بهبود داده و در برنامه‌نویسی VHDL، یک رویکرد خوب و توصیه شده شناخته می‌شود.

دانشگاه حسنی شاهزاد

Listing 9.3: VHDL code for the design hierarchy for the 3-bit comparator.

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_compare is
    Port ( A_IN : in std_logic_vector(2 downto 0);
           B_IN : in std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_compare;
-- architecture
architecture ckt1 of my_compare is

    -- XNOR gate -----
    component big_xnor 
        Port ( A,B : in std_logic;
               F : out std_logic);
    end component;

    -- 3-input AND gate -----
    component big_and3 
        Port ( A,B,C : in std_logic;
               F : out std_logic);
    end component;

    -- intermediate signal declaration
    signal p1_out,p2_out,p3_out : std_logic;

begin
    x1: big_xnor port map (A => A_IN(2),
                           B => B_IN(2),
                           F => p1_out);

    x2: big_xnor port map (A => A_IN(1),
                           B => B_IN(1),
                           F => p2_out);

    x3: big_xnor port map (A => A_IN(0),
                           B => B_IN(0),
                           F => p3_out);

    a1: big_and3 port map (A => p1_out,
                           B => p2_out,
                           C => p3_out,
                           F => EQ_OUT);
end ckt1;

```

توجه به این نکته سودمند است که جواب ارائه شده در برنامه‌ی ۹-۳ تنها رویکرد ممکن برای تحقق فرآیند نگاشت نیست. روش استفاده شده در برنامه‌ی ۹-۳، «نگاشت مستقیم»^۱ کامپوننت‌ها است. در

¹ Direct mapping

این روش، هر یک از سیگنالهای مورد نیاز در اتصال دادن واحدهای طراحی، فهرست شده و مستقیماً توسط عملگر \Rightarrow ، به سیگنالهایی که قرار است در طراحی سطح بالاتر متصل شوند، تناظر داده می‌شوند. این روش دارای چند مزیت است: صریح و شفاف است؛ کامل است؛ سیگنالها را می‌توان با هر ترتیبی داخل فهرست ذکر کرد. تنها مشکل یا عیب آن این نکته می‌تواند باشد که فضای نسبتاً زیادی داخل کد VHDL اشغال می‌کند.

یک راه دیگر برای تحقق فرآیند نگاشت، «نگاشت ضمنی»^۱ است. در این روش، اتصال بین سیگنالهای خارجی واحدهای طراحی با سیگنالهای واحد طراحی به همان ترتیبی انجام می‌شود که در دستور نگاشت ظاهر شده است. تفاوت این روش با روش قبلی («نگاشت مستقیم») این است که در اینجا تنها سیگنالهای مربوط به طراحی سطح بالاتر در دستور نگاشت ظاهر می‌شوند. تناظر بین سیگنالهای متعلق به واحدهای طراحی و طراحی سطح بالاتر به همان ترتیب رعایت شده در بخش اعلان موجودیت یا اعلان کامپوننت انجام می‌شود. این روش فضای کمتری در کد VHDL اشغال می‌کند؛ اما در مقابل باید ترتیب ذکر و استفاده از سیگنالها را رعایت کرد. برنامه‌ی ۴-۹ همان برنامه‌ی ۳-۹ را اما این بار به کمک روش نگاشت ضمنی نشان می‌دهد.

برای انجام موققیت‌آمیز شبیه‌سازی و سنتز طراحی نشان داده شده در برنامه‌ی ۹-۳، باید برنامه‌ی ۹-۱ را نیز در پروژه‌ی VHDL خود داخل کنید. معمولاً این دو برنامه در دو فایل جداگانه ذخیره و نگهداری می‌شوند؛ همین مطلب در مورد برنامه‌ی ۹-۴ نیز صادق است.

نگاشت ضمنی

¹ Implied mapping

Listing 9.4: Alternative architecture for Example 22 using implied mapping.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_compare is
    Port ( A_IN : in std_logic_vector(2 downto 0);
           B_IN : in std_logic_vector(2 downto 0);
           EQ_OUT : out std_logic);
end my_compare;
-- architecture
architecture ckt2 of my_compare is
    component big_xnor 
        Port ( A,B : in std_logic;
               F : out std_logic);
    end component;
    component big_and3 
        Port ( A,B,C : in std_logic;
               F : out std_logic);
    end component;
    signal p1_out,p2_out,p3_out : std_logic;
begin
    x1: big_xnor port map (A_IN(2),B_IN(2),p1_out);
    x2: big_xnor port map (A_IN(1),B_IN(1),p2_out);
    x3: big_xnor port map (A_IN(0),B_IN(0),p3_out);
    a1: big_and3 port map (p1_out,p2_out,p3_out,EQ_OUT);
end ckt2;
```

ممکن است به نظر برسد در طراحی‌های ساختاری، همانم بودن سیگنال‌ها مشکل ایجاد می‌کند؛ برای مثال، اغلب طراحی‌ها واحدهای طراحی مختلف مربوط به سطوح طراحی مختلف، دارای سیگنال کلاک هستند که ممکن است در آنها از یک نام مشترک استفاده شده باشد. در هنگام نمونه‌سازی تکرار نام مشترک مشکلی ایجاد نمی‌کند زیرا کامپایلر می‌داند که هر نام متعلق به کدام سطح طراحی است. برای مثال، نمونه‌سازی کامپونتی که در برنامه‌ی ۹-۵ نشان داده شده است، معتبر و فاقد اشکال است؛ در واقع تصادم نام که ظاهراً در این جا رخ داده مشکلی ایجاد نمی‌کند زیرا کامپایلر می‌داند که سیگنالی که در سمت چپ عملگر $=>$ استفاده شده است، مربوط به داخلی کامپوننت و سیگنالی که (با همان نام CS یا CLK) در سمت راست این عملگر آمده متعلق به سطح طراحی بالاتر است.

Listing 9.5: Example of the same signal name crossing hierarchical boundaries.

```
x5: some_component port map (CLK => CLK,
                               CS => CS);
```

Generic Map ۲-۹

همان طور که در بخش قبلی دیدیم، استفاده از کلمه‌ی کلیدی component به ما اجازه می‌دهد یک ماثول VHDL را به منظور نمونه‌سازی به تعداد دفعات دلخواه، اعلان کنیم.

اغلب تمایل داریم کدی بنویسیم که عام^۱ و قابل توسعه باشد؛ برای مثال، برنامه‌ای که یک کار مشخص را روی یک آرایه‌ی ورودی با طول قابل تنظیم انجام دهد. فرض کنیم می‌خواهیم قطعه‌ای از کد مربوط به یک نوع برنامه‌ی وارسی^۲ توازن^۳ را پیاده‌سازی کنیم که در آن اگر آرایه‌ی ورودی به اندازه‌ی N شامل یک عدد زوج باشد، مقدار خروجی^۴ '۱' و اگر آرایه‌ی ورودی شامل عدد فردی باشد، مقدار خروجی^۵ '۰' برگرداند. برنامه‌ی ۶-۹ چنین برنامه‌ای را نشان می‌دهد.

Listing 9.6: Parity check implementation with generic input array size.

```

1 -- library declarations
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 -- entity
5 entity gen_parity_check is
6     generic ( n: positive);
7     port      ( x: in std_logic_vector(n-1 downto 0);
8                  y: out std_logic);
9 end gen_parity_check;
10 -- architecture
11 architecture arch of gen_parity_check is
12 begin
13     process(x)
14         variable temp: std_logic;
15     begin
16         temp:='0';
17         for i in x'range loop
18             temp := temp XOR x(i);
19         end loop;
20         y <= temp;
21     end process;
22 end arch;
```

برنامه‌ی ۷-۹ نشان می‌دهد که چگونه با استفاده از روش component می‌توان برنامه‌ی ۶-۹ را در کد خود اعلان و نمونه‌سازی کنیم. به ویژه در برنامه‌ی ۷-۹، از برنامه‌ی عام ۹-۶ در قالب یک ماثول وارسی^۴ توازن^۵-بیتی استفاده شده است.

برای دستیابی به این قابلیت کدنویسی عام، کلمه‌ی کلیدی generic هم در داخل میدان entity مربوط به کد ۹-۶ و هم در داخل میدان component در زمان اعلان شدن در داخل کد ۹-۷ استفاده شده است.

¹ Generic

² Parity

از میدان generic (در داخل component و در زمان اعلان -م) برای فراهم آوردن امکان کنترل تمام متغیرهای generic استفاده می‌شود.

توجه کنید که چگونه در زمان نمونه‌سازی (برنامه‌ی ۹-۷، خط ۲۰ را ملاحظه کنید) از کلمه‌ی کلیدی generic به همراه کلمه‌ی کلیدی port map برای تعریف متغیرهای عام استفاده شده است.

Listing 9.7: Use of generic for the construct of a generic parity check code.

```

1 -- library declaration
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 -- entity
5 entity my_parity_chk is
6     Port ( input    : in  std_logic_vector(3 downto 0);
7             output   : out std_logic);
8 end my_parity_chk;
9
10 -- architecture
11 architecture arch of my_parity_chk is
12     ----- component declaration -----
13     component gen_parity_check
14         generic ( std_logic : positive);
15         port      ( input    : in  std_logic_vector(N-1 downto 0);
16                     output   : out std_logic);
17     end component;
18 begin
19     ----- component instantiation -----
20     cpl: my_parity_chk generic map (4) port map (input, output);
21 end arch;
```

مجددأً یادآوری می‌شود که به منظور انجام موفقیت‌آمیز شبیه‌سازی و سنتز برنامه‌ی ۹-۶ نیز به پروژه شما الحاق (یا اضافه) شده باشد.

۹-۳ نکات مهم

- مدل‌سازی ساختاری در VHDL از مفاهیم طراحی سلسله‌مراتبی پشتیبانی می‌کند. توانایی بسط (و توسعه‌ی) مدارات دیجیتال به سطوح بالاتر، کلید فهم و طراحی مدارات دیجیتال پیچیده است.
- امروزه دیگر طراحی شماتیکی و ترسیمی مدارات دیجیتال، روشی قدیمی و ناکارآمد است.
- مدل ساختاری VHDL از قابلیت استفاده‌ی مجدد از واحدهای طراحی پشتیبانی می‌کند. این قابلیت شامل واحدهای طراحی که شما قبل آنها را طراحی کرده‌اید و نیز مأذول‌های از قبل تعریف شده در کتابخانه‌ها می‌شود.

- اگر شما از یک ابزار توسعه‌ی نرم‌افزاری متعلق به یکی از شرکت‌های سازنده‌ی FPGA استفاده می‌کنید، می‌توانید بلوک‌های دیجیتالی را که آن شرکت قبلاً برای شما طراحی و آماده کرده است، اعلان و سپس استفاده کنید. در این صورت نیازی به اعلان موجودیت مشابه با برنامه‌ی ۲-۹ ندارید، بلکه به سادگی کافی است مانند مثال زیر، یک کتابخانه‌ی مناسب را در داخل کد خود وارد کنید.

```
library UNISIM;
use UNISIM.VComponents.all;
```

تمام بلوک‌های دیجیتالی که داخل کتابخانه‌ی فوق آماده و در دسترس شما قرار داده شده‌اند، در مستندسازی‌های^۱ مربوط به ابزار توسعه‌ی نرم‌افزاری FPGA (برای مثال نرم‌افزار Vivado متعلق به شرکت Xilinx) معرفی و توضیح داده شده‌اند.

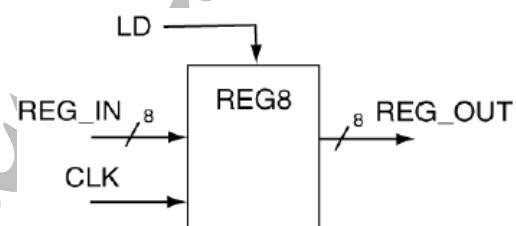
^۱ Documentation

فصل دهم

ثباتها و سطح انتقال ثباتی (RTL)

به طور ساده، یک ثبات (رجیستر) در VHDL یک نسخه‌ی برداری از یک فلیپ فلاپ D است که در آن تمام عملیات‌ها به طور همزمان با هم اجرا می‌شوند. «سطح انتقال ثباتی» یا RTL نوعی از طراحی است که اساساً روی این مطلب تمرکز دارد که چه موقع و چگونه داده‌ها در یک سیستم دیجیتال بین ثباتهای مختلف انتقال داده می‌شوند. طراحی سطح RTL اغلب با طراحی‌های «مسیر داده» متناظر گرفته می‌شود که در آن نیاز به کنترل و زمان‌بندی دقیق داده‌های انتقال یافته بین ثباتها وجود دارد. حتی در انتقال داده بین دو ثبات ساده نیز نیاز به کنترل دقیق توسط یک موجودیت بیرونی است تا «دبنه‌ی داده» را کنترل کند. در این موارد، دنباله‌ی صحیح انتقال داده‌ها توسط یک FSM کنترل می‌شود.

مثال ۲۳: با استفاده از مدل سازی رفتاری در VHDL، یک ثبات ۸-بیتی دارای سیگنال بارگذاری موازن سنکرون فعال-بالا طراحی کنید. بارگذاری ثبات را سنکرون با لبه‌های بالارونده‌ی کلاک در نظر بگیرید.



حل: برنامه‌ی جواب مربوط به یک ثبات ۸-بیتی تا حد زیادی شبیه به مدل یک فلیپ فلاپ D است. جواب کامل این مثال، در برنامه‌ی ۱۰-۱ نشان داده شده است. چند نکته وجود دارد که توجه به آنها مفید است:

Listing 10.1: Solution to Example 23.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity reg8 is
    Port ( REG_IN : in std_logic_vector(7 downto 0);
            LD, CLK : in std_logic;
            REG_OUT : out std_logic_vector(7 downto 0));
end reg8;
-- architecture
architecture reg8 of reg8 is
begin
    reg: process(CLK)
    begin
        if (rising_edge(CLK)) then
            if (LD = '1') then
                REG_OUT <= REG_IN;
            end if;
        end if;
    end process;
end reg8;
```

- توجه کنید که یک دستور if در این برنامه وجود دارد که فاقد دستور else است؛ بنابراین، این کار موجب تولید حافظه می‌شود (که در اینجا مطلوب است – م). در این مثال، تعداد ۸ عنصر حافظه‌ی بیتی (یعنی فلیپ فلاپ) وجود خواهد داشت؛ در اینجا، این فلیپ فلاپ‌ها از نوع D هستند. این عناصر حافظه مربوط به سیگنال گروهی REG_OUT هستند. سادگی استفاده از D در تولید فلیپ فلاپ‌های D به این شیوه، باعث شده است فلیپ فلاپ‌های D متداول‌ترین و رایج‌ترین نوع فلیپ فلاپ در مدارات دیجیتال باشند.

- در کد، برای ورودی و خروجی از سیگنال گروهی استفاده شده است. در VHDL، تخصیص یک سیگنال گروهی به سیگنال گروهی دیگر، همان طور که در کد نشان داده شده است، سرراست و ساده است. در بسیاری از مواقع، مانند آن چه که در این کد ملاحظه می‌کنید، در این تخصیص نیازی به استفاده از عملگر دسترسی گروهی نیست.

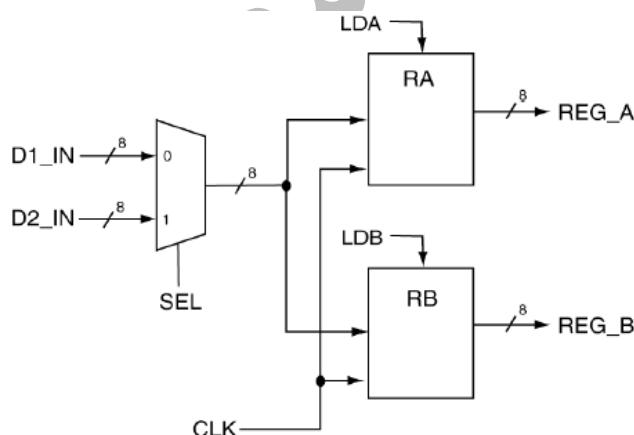
- تخصیص ورودی به خروجی، بستگی به وضعیت لبه‌ی کلاک و نیز سیگنال LD دارد. رویکردی که در برنامه ۱۰ - ۱ در پیش گرفته شده است، استفاده از یک if جداگانه برای هر دو سیگنال CLK و LD است. می‌توانستیم هر دو شرط (مربوط به هر یک از دو دستور if) را با هم در یک دستور if تجمعی کنیم؛ اما این کار در هنگام کدنویسی VHDL و کار با عناصر سنکرون، روش خوب و توصیه شده‌ای نیست. به بیان دیگر، شما باید همیشه سعی کنید شرایط مربوط به سیگنال کلاک‌دهی را از شرایط مربوط به کاری که می‌خواهید انجام بگیرد، مجزا کنید. در دنیای VHDL

سیگنالهای مربوط به کلک، خاص و ویژه هستند؛ شما باید با احتیاط و ظرفت با آنها رفتار کنید.

- با توجه به این که سیگنالهای IN REG و LD باید سنکرون باشند، این سیگنالها درون لیست حساسیت فرآیند قرار داده نشده‌اند.

مدار مربوط به مثال بعدی اندکی پیچیده‌تر از غالب مداراتی است که تاکنون در مثالها بررسی شده‌اند. به علاوه به خاطر داشته باشید که برای یک مساله‌ی خاص، جوابهای زیاد و مختلف وجود دارد. این امر در VHDL طبیعی است؛ در واقع، در اکثر موقع نمی‌توان «بهترین» جواب را برای پیاده‌سازی یک مدار معین تعیین کرد. مثالهایی که در ادامه می‌آیند، در واقع مربوط به یک مثال مشترک هستند که با روش‌های مختلف پیاده‌سازی شده‌اند (اما از نظر عملکرد، یکسان و معادل هستند).

مثال ۲۴: از مدل‌سازی رفتاری در VHDL برای طراحی مدار نشان داده در زیر استفاده کنید. هر دو سیگنال بارگذاری را فعال—بالا فرض کنید. همچنین، فرض کنید که مدار بالهی بالاروندهی سیگنال کلک سنکرون است.



حل: مدار نشان داده شده شامل دو ثبات ۸-بیتی و یک مالتی‌پلکسر ۲ به ۱ است. این مدار مثالی از انتقال داده مبتنی بر گذرگاه در خروجی مالتی‌پلکسر است که به ورودی‌های ثباتها متصل شده است (در واقع این جا یک گذرگاه ساده پیاده‌سازی شده است – م).

هریک از دو ثبات، ورودی کنترل بارگذاری مخصوص به خود را دارد. جواب به این مثال در برنامه‌ی ۱۰-۲ نشان داده شده است. نکات قابل توجه در خصوص این برنامه، عبارتند از:

Listing 10.2: Solution to Example 24.

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity ckt_rtl is
    port (D1_IN,D2_IN : in std_logic_vector(7 downto 0);
          CLK,SEL : in std_logic;
          LDA,LDB : in std_logic;
          REG_A,REG_B : out std_logic_vector(7 downto 0));
end ckt_rtl;
-- architecture
architecture rtl_behavioral of ckt_rtl is
    -- intermediate signal declaration -----
    signal s_mux_result : std_logic_vector(7 downto 0);
begin
    ra: process(CLK) -- process
    begin
        if (rising_edge(CLK)) then
            if (LDA = '1') then
                REG_A <= s_mux_result;
            end if;
        end if;
    end process;

    rb: process(CLK) -- process
    begin
        if (rising_edge(CLK)) then
            if (LDB = '1') then
                REG_B <= s_mux_result;
            end if;
        end if;
    end process;

    with SEL select
        s_mux_result <= D1_IN when '1',
                           D2_IN when '0',
                           (others => '0') when others;
end rtl_behavioral;

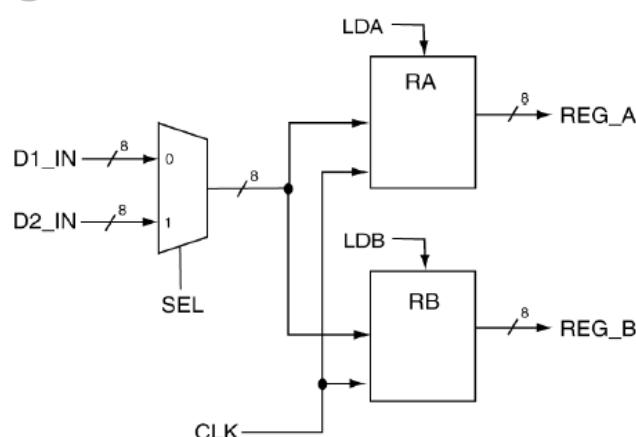
```

- در این برنامه، سه دستور همزمان وجود دارد که دو تای آنها از مدل رفتاری و یکی از مدل جریان داده استفاده می‌کند.
- برای هر ثبات از یک فرآیند مجزا استفاده شده است. گرچه می‌توانستیم برای هر دو ثبات از یک فرآیند استفاده کنیم، اما این کار موجب پیچیده‌تر شدن کد و کاهش قابلیت فهم آن می‌شد.
 - همیشه رویکرد بهتر در VHDL، شکستن کارها به کارهای کوچکتر و منطقاً مجزا و سپس استفاده از تکنیک‌های مختلف مدل‌سازی VHDL (برای پیاده‌سازی کارها و) حفظ سادگی و

تمایز آنها است. واقعیت این است که اگر مدل‌های ساده در اختیار سنترکننده قرار دهید، این ابزار دوست شما خواهد شد! کمیت کدی که برای توصیف یک مدار خاص استفاده می‌کنید اهمیتی ندارد. پیچیدگی هر مدل داده شده به حجم و کمیت کد آن مدل بستگی ندارد بلکه توسط پیچیده‌ترین قطعه کد موجود در آن مدل تعیین می‌شود؛ بنابراین در VHDL، سادگی همیشه خوب و حتی بهتر است (زیرا باعث می‌شود کل مدل، ساده شود).

- تمام سیگنال‌هایی که در مثال ۲۴ نشان داده شده‌اند، اتصال و وابستگی خارجی دارند (یعنی توسط دنیای بیرون تعیین می‌شوند یا این که به دنیای بیرون منتقل می‌شوند) به جز خروجی مالتی‌پلکسر که به ورودی هر دو ثبات متصل شده است. رویکرد نهایی مورد استفاده در این مثال، متداول است: تعداد زیادی فرآیند که با یکدیگر از طریق سیگنال‌های به اشتراک گذاشته شده ارتباط برقرار می‌کنند. در این مثال، تنها یک سیگنال به اشتراک گذاشته شده وجود دارد زیرا برنامه، برنامه‌ی ساده‌ای است. در مدارات پیچیده‌تر نیز از همین مدل برای ارتباط فرآیندها با هم استفاده می‌شود.
- مدل مربوط به مالتی‌پلکسر ۲ به ۱ از عبارت `'0' > others => 1` استفاده کرده است. این عبارت یک میانبر برای تولید تعدادی مقدار `'0'` (برای تخصیص به خروجی) است. لذا مزیت استفاده از این روش این است که نیازی ندارید تعداد دقیق `'0'`‌ها را بدانید؛ بنابراین، هرگاه عرض سیگنال گروهی مورد نظر تغییر کند، نیازی به تغییر این خط از کد نیست.

مثال ۲۵: از مدل‌سازی ساختاری در VHDL برای طراحی مدار نشان داده شده در زیر استفاده کنید. هر دو سیگنال بارگذاری را فعال-بالا فرض کنید. همچنین، مدار را باللهای بالاروندهی سیگنال کلاک سنکرون فرض کنید.



حل: جواب این مثال در برنامه‌ی ۱۰-۳ نشان داده شده است. این مثال، در واقع مثال واقعی‌تری نسبت به مثال‌های قبلی، در زمینه‌ی استفاده از مدل ساختاری است. نکات درخور ر توجه در اینجا عبارتند از:

- مهمترین نکته در خصوص برنامه‌ی ۱۰-۳ این است که علیرغم کمیت نسبتاً زیاد آن، از ساختار خوبی برخوردار است. اگر با این ساختار به خوبی آشنا شده باشید، به راحتی می‌توانید برنامه‌ی جواب را بفهمید و حتی مطلب مهم‌تر و بهتر این که قادر خواهید بود کدهای متعدد و پیچیده‌ی VHDL را بنویسید.

- برنامه‌ی ۱۰-۳ از قالب و شکل زیبایی برخوردار است. به ویژه این که این کد به خوبی تودرتو گذاری^۱ شده و بنابراین، اطلاعات را به خوبی منتقل می‌کند. فهم این نوع کدها آسان است. همچنین، دیگران با یک نگاه مختصر، سریعتر به منظور و عملکرد کد شما پی می‌برند.

فیضی شاهزاد

¹ Indent

Listing 10.3: Solution to Example 25 using a structural modeling approach.

```

entity mux2t1 is                                     --- ENTITY
  port ( A,B : in  std_logic_vector(7 downto 0);
         SEL : in  std_logic;
         M_OUT : out std_logic_vector(7 downto 0));
end mux2t1;
architecture my_mux of mux2t1 is                  --- ARCHITECTURE
begin
  with SEL select
    M_OUT <= A when '1',
                B when '0',
                (others => '0') when others;
end my_mux;
entity reg8 is                                     --- ENTITY
  Port ( REG_IN : in  std_logic_vector(7 downto 0);
         LD,CLK : in  std_logic;
         REG_OUT : out std_logic_vector(7 downto 0));
end reg8;
architecture reg8 of reg8 is                      --- ARCHITECTURE
begin
  reg: process(CLK)
  begin
    if (rising_edge(CLK)) then
      if (LD = '1') then
        REG_OUT <= REG_IN;
      end if;
    end if;
  end process;
end reg8;
entity ckt_rtl is                                 --- ENTITY
  port ( D1_IN,D2_IN : in  std_logic_vector(7 downto 0);
         CLK,SEL : in  std_logic;
         LDA,LDB : in  std_logic;
         REG_A,REG_B : out std_logic_vector(7 downto 0));
end ckt_rtl;
architecture rtl_structural of ckt_rtl is          --- ARCHITECTURE
-- component declaration
component mux2t1
  port ( A,B : in  std_logic_vector(7 downto 0);
         SEL : in  std_logic;
         M_OUT : out std_logic_vector(7 downto 0));
end component;
component reg8
  Port ( REG_IN : in  std_logic_vector(7 downto 0);
         LD,CLK : in  std_logic;
         REG_OUT : out std_logic_vector(7 downto 0));
end component;
-- intermediate signal declaration
signal s_mux_result : std_logic_vector(7 downto 0);

```

```

begin
  ra: reg8
  port map ( REG_IN => s_mux_result,
              LD => LDA,
              CLK => CLK,
              REG_OUT => REG_A );
  rb: reg8
  port map ( REG_IN => s_mux_result,
              LD => LDB,
              CLK => CLK,
              REG_OUT => REG_B );
  m1: mux2t1
  port map ( A => D1_IN,
              B => D2_IN,
              SEL => SEL,
              M_OUT => s_mux_result);
end rtl_structural;

```

۱-۱۰ نکات مهم

- از VHDL می‌توان به راحتی برای پیاده‌سازی مدارات در سطح انتقال ثبات (RTL) استفاده کرد. مدل‌های VHDL که در این کار استفاده می‌شوند قابل پیاده‌سازی به صورت ساختاری یا کاملاً رفتاری هستند.
- مدل‌های VHDL سطح RTL باید به دنبال حفظ سادگی باشند. اگر مدل‌های رفتاری در طراحی RTL پیچیده شوند، شанс این که مداراتان به خوبی کار کند، تا حد زیادی کاهش می‌یابد زیرا حالا باید یک مدار پیچیده (و نه ساده) سنتز شود.

فصل یازدهم

اشیاء داده‌ای (Data Objects)

اشیاء در VHDL، اقلامی هستند که دارای یک نام (شناسه) و یک نوع مشخص هستند. چهار نوع شیء و تعداد زیادی نوع داده‌ای^۱ در VHDL وجود دارد. تا به حال ما با شیء داده‌ای^۲ signal و با نوع داده‌ای آشنا شده‌ایم. در این فصل دو شیء داده‌ای جدید و چندین نوع داده‌ای جدید معرفی می‌شوند.

۱۱-۱ انواع اشیاء داده‌ای

در VHDL چهار نوع «شیء داده‌ای» وجود دارد: سیگنال (signal)، متغیر (variable)، ثابت (constant)، و فایل (file). شیء داده‌ای فایل که صرفاً در شبیه‌سازی استفاده می‌شود، در این فصل بررسی نمی‌شود.

۱۱-۲ اعلان اشیاء داده‌ای

اولین نکته در مورد اشیاء داده‌ای، شباهت اعلان آنها است. قالب اشیاء داده‌ای که راجع به آنها بحث خواهیم کرد، در جدول ۱۱-۱ نشان داده شده است. در این جدول، کلماتی که به صورت توپر نشان داده شده‌اند، کلمات کلیدی هستند.

| VHDL data object | Declaration form |
|------------------|--|
| Signal | signal sig_name : sig_type:=initial_value; |
| Variable | variable var_name : var_type:=initial_value; |
| Constant | constant const_name : const_type:=initial_value; |

Table 11.1: Data object declaration forms.

در جدول فوق توجه کنید که تخصیص مقدار اولیه، کاری دلخواه است. معمولاً در زمان اعلان یک «شیء سیگنال» از مقداردهی اولیه استفاده نمی‌شود اما در مقابل، برای اشیاء ثابت معمولاً همیشه

¹ Data type

² Data object

مقداردهی اولیه انجام می‌شود. مقداردهی اولیه روی سیگنال‌ها را نمی‌توان روی سیلیکون پیاده‌سازی کرد بلکه این مقادیر اولیه توسط ابزارهای سنتز جهت شبیه‌سازی استفاده می‌شوند. مثالهایی از اعلان این سه نوع شیء داده‌ای در جدول ۱۱-۲ نشان داده شده است. در این مثالها از انواع داده‌ای جدیدی استفاده شده است که بعداً بحث و بررسی می‌شوند.

| Data object | Declaration form |
|-------------|---|
| Signal | <pre>signal sig_var1 : std_logic := '0'; signal tmp_bus : std_logic_vector(3 downto 0):="0011"; signal tmp_int : integer range -128 to 127 := 0; signal my_int : integer;</pre> |
| Variable | <pre>variable my_var1, my_var2 : std_logic; variable index_a : integer range (0 to 255) := 0; variable index_b : integer := -34;</pre> |
| Constant | <pre>constant sel_val : std_logic_vector(2 downto 0):="001"; constant max_cnt : integer := 12;</pre> |

Table 11.2: Example declarations for signal, variable and constant data objects.

۱۱-۳ متغیرها و عملگر تخصیص “:=”

گرچه متغیرها مشابه با سیگنال‌ها هستند اما به دلایل متعددی که در این فصل اشاره شده است، چندان عملیاتی نیستند. متغیرها را تنها می‌توان داخل فرآیندها، توابع^۱ و روال‌ها^۲ اعلان کرد (توابع و روال‌ها در این فصل بررسی نمی‌شوند). بنابراین، دستورات تخصیص متغیرها از نوع ترتیبی هستند زیرا تمام دستوراتی که داخل بدنه‌ی یک فرآیند (و توابع و روال‌ها) قابل استفاده هستند، تنها می‌توانند از نوع ترتیبی باشند. یکی از اشتباهاتی که برنامه‌نویسان VHDL در ابتدای کار خود مرتکب می‌شوند، استفاده از متغیرها در بیرون از فرآیند است.

برای انتقال مقدار به یک متغیر باید از عملگر تخصیص (=) استفاده کرد (در مقابل سیگنال‌ها که برای انتقال مقدار یک سیگنال به یک سیگنال دیگر از عملگر => استفاده می‌شود). نکته‌ی دیگر این است که برای دادن مقدار اولیه به هر یک از سه نوع شیء داده‌ای از عملگر = استفاده می‌شود (یعنی این عملگر، فرآبارگذاری^۳ شده است).

¹ Functions

² Procedures

³ Overload

۱۱-۴ سیگنال‌ها در مقایسه با متغیرها

استفاده از سیگنال‌ها و متغیرها، به دلیل تشابهاتی که با هم دارند، تا حدی گیج‌کننده و ابهام‌آور است. اگر بخواهیم کلی صحبت کنیم، سیگنال‌ها را می‌توان به عنوان سیم یا نوعی اتصال فیزیکی موجود در یک طرح سخت‌افزاری دانست. بنابراین، سیگنال‌ها روشی برای اتصال مازول‌های VHDL با دنیای بیرونی خود محسوب می‌شوند. در حوزه‌ی شبیه‌سازی مداری، این سیگنال‌ها هستند که می‌توان آنها را طوری برنامه‌ریزی کرد که در زمانهای معینی، مقادیر معینی بگیرند؛ حال آن که این قابلیت در مورد متغیرها وجود ندارد. بنابراین، رخدادها را تنها برای سیگنال‌ها، و نه متغیرها، می‌توان برنامه‌ریزی کرد.

در مدارات نسبتاً ساده، سیگنال‌ها به تنها یکی کافی بوده و معمولاً نیازی به استفاده از متغیرها نداریم. با پیچیده‌تر شدن مدارات، ممکن است شما نیاز به اعمال کنترل بیشتری روی مدل خود داشته باشید که خارج از توان سیگنال‌ها باشد. مهمترین ویژگی سیگنال‌ها که موجب تا حدی محدود شدن آنها شده است، زمان و نحوه اعمال برنامه‌ریزی^۱ روی آنها است؛ به بیان دیگر، «تخصیص انجام شده روی یک سیگنال و در داخل یک فرآیند، تا انتهای آن فرآیند کامل نخواهد شد و تنها پس از اتمام فرآیند است که عمل تخصیص واقعاً روی سیگنال اعمال می‌شود». به همین دلیل است که در طول اجرای یک فرآیند، امکان تخصیص همزمان روی یک سیگنال مشخص قابل انجام بوده و هیچ خطای سنتزی گزارش نخواهد شد؛ در این حالت، تنها آخرین تخصیص انجام شده ملاک عمل قرار گرفته و در پایان فرآیند، روی سیگنال اعمال می‌شود (توجه کنید که در بیرون از یک فرآیند، امکان اعمال همزمان چند تخصیص روی یک سیگنال وجود ندارد و پیام خطای صادر نخواهد شد -م). بنابراین، مشکلی که ممکن است برای شما به وجود آید این است که امکان استفاده از مقدار جدید تخصیص یافته به سیگنال، در داخل فرآیند وجود ندارد (زیرا این مقدار در انتهای فرآیند روی سیگنال اعمال می‌شود).

تخصیص متغیر درون فرآیند به شیوه متفاوتی نسبت به سیگنال انجام می‌شود؛ تخصیص متغیر داخل فرآیند به صورت آنی و بلافصله انجام شده و مقدار جدید، بلافصله در ادامه‌ی فرآیند قابل استفاده است. به بیان دیگر، برخلاف سیگنال، تخصیص سیگنال به حالت تعویق یافته و برنامه‌ریزی شده نیست. این تفاوت (بین سیگنال و متغیر)، بسیار مهم بوده و اثرات و عواقب بسیار مهمی هم در شبیه‌سازی و هم در سنتز دارد.

متغیرها لزوماً همیشه به عنوان «سیم» در یک مدار مدل نمی‌شوند. همچنین، متغیرها مفهوم حافظه را با خود به همراه ندارند زیرا قادر به ذخیره‌ی رویدادها نیستند. ممکن است از خود بپرسید که پس محل مناسب برای استفاده و به کارگیری متغیرها کجاست؟ جواب این است که متغیرها تنها باید یا به عنوان شمارنده‌ی تعداد تکرار در حلقه‌ها یا به عنوان مقادیر موقتی در حین اجرای الگوریتمی که مشغول

¹ Schedule

عملیات محاسبه است، استفاده شوند. گرچه می‌توان در خارج از این دو حوزه نیز از متغیرها استفاده کرد اما توصیه می‌شود از این کار خودداری کنید.

قبل‌اشاره شد که دستورات داخل یک فرآیند به صورت ترتیبی اجرا می‌شوند؛ این مطلب باید فکر شما را به این سمت سوق دهد که محیط یک فرآیند مشابه با قطعه‌ای از یک کد C است. به خاطر داشته باشید که هر خط از یک کد C نیاز به چند سیکل کلاک دارد تا اجرای آن کامل شود؛ اما در مقابل، دستورات VHDL باید در یک زمان بسیار کوتاه، کمتر از یک سیکل کلاک، اجرا شوند. هزینه‌ای که در مقابل این اجرای بسیار سریع پرداخته می‌شود این است که تخصیص سیگنال انجام شده در داخل یک فرآیند، در پایان آن فرآیند کامل و عملیاتی می‌شود. بنابراین توصیه می‌شود که «فرآیندهای خود را ساده و کوتاه بنویسید».

۱۱-۵ انواع داده‌ای استاندارد

در VHDL نه تنها انواع داده‌ای از قبل تعریف شده‌ی زیادی وجود دارد، بلکه به کاربر نیز اجازه داده شده است که انواع داده‌ای مورد نیاز خود را تعریف و ایجاد کند. در اینجا ما تنها با تعدادی از انواع داده‌ای که بسیار متداول و رایج هستند، کار می‌کنیم. از بین انواع داده‌ای متداول موجود در VHDL، می‌توان به انواع داده‌ای زیر اشاره کرد:

bit: یک نوع شمارش شده دومقداره^۱ است که جای آن را نوع std_logic گرفته است.

bit_vector: با نوع قدرتمند std_logic_vector جایگزین شده است.

boolean: یک نوع شمارش شده دومقداره است.

boolean_vector: شکل برداری نوع boolean است. به بخش «انواع صحیح» مراجعه کنید.

natural: یک زیرنوع از نوع integer است زیرا این نوع، نوع صحیح نامنفی است.

positive: یک زیرنوع از نوع integer است زیرا این نوع، نوع صحیح مثبت است.

integer_vector: شکل برداری نوع integer است.

character: یک نوع شمارش شده ۲۵۶-نمادی است.

string: شکل برداری نوع character است.

در بخش‌های بعدی چند نوع متداول و مفید معرفی می‌شود.

¹ Two-value enumerated

² Sub-type

۱۱-۶ انواع تعریف شده توسط کاربر^۱

در VHDL به شما اجازه داده شده است نوع داده خودتان را تعریف کنید. برای مثال، تعریف نوع صحیح سفارشی شده^۲ و استفاده از آن به صورت زیر است:

```
type my_type is range 0 to 100;
constant my_const : my_type := 31;
```

طبعاً امکان تعریف ساختارهای داده‌ای پیچیده‌تر فراهم است. برای مثال اگر بخواهید یک حافظه ROM را در VHDL پیاده‌سازی کنید، روش متداول این است که از یک نوع داده‌ای سفارشی شده استفاده کنید:

```
-- typical custom data type for a 20-byte ROM
type memory is array (0 to 19) of std_logic_vector(7 downto 0);
constant my_rom : memory := (
    1 => "11111111"
    2 => "11110111"
    5 => "11001111"
    12 => "10110101"
    18 => "10001101"
    others => "00000000");
```

۱۱-۷ انواع رایج و پراستفاده

انواعی که در فصلهای قبلی معرفی شدند به همراه دو نوع جدید دیگر در جدول ۱۱-۳ فهرست شده‌اند. ما قبلاً انواع std_logic_vector و std_logic را بسیار استفاده کردیم. این دو نوع پیچیده‌تر از حدی هستند که تاکنون معرفی شده‌اند و در این فصل بیشتر توضیح داده می‌شوند. نوع شماره شده (enumerated) در فصل مربوط به ماشینهای حالت استفاده شد. نوع integer به طور مختصر و موجز قبلاً اشاره شد اما در این فصل به همراه نوع boolean توضیح داده می‌شود.

¹ User-defined types

² Custom integer

| Type | Example | Usage |
|-------------------|---|--------------|
| std_logic | signal my_sig : std_logic; | all examples |
| std_logic_vectors | signal busA : std_logic_vector(3 downto 0); | all examples |
| enumerated | type state_type is (ST0, ST1, ST2, ST3); | Example 18 |
| boolean | variable my_test : boolean := false; | None |
| integer | signal iter_cnt : integer := 0; | Example 26 |

Table 11.3: Some popular data types already introduced in previous chapters.

۱۱-۸ انواع صحیح (integer)

استفاده از نوع صحیح در طراحی کدهای VHDL که وظیفه و ذات محاسباتی و الگوریتمی دارند، مفید است. این نوع کدهای VHDL امکان توصیف رفتار مدارات دیجیتال پیچیده را فراهم می‌کنند. همزمان که در مطالعه‌ی سیستمهای دیجیتال جلو می‌روید، احساس نیاز به داشتن ابزارهای توصیفی پیچیده‌تر در VHDL خواهدید کرد. انواع داده‌ای مختلف مانند integer در واقع پاسخی به این نیاز هستند. محدوده‌ی تغییرات نوع صحیح (integer) از -2^{31} تا $2^{31}-1$ است. این محدوده در واقع نمایش ۳۲-بیتی یک عدد علامتدار است: از -2^{31} تا $2^{31}-1$. انواع دیگری که مشابه با نوع صحیح (integer) باشند، شامل انواع positive و natural هستند. این انواع در واقع همان نوع صحیح هستند که محدوده‌ی مقادیر آنها (نسبت به محدوده‌ی مقادیر نوع صحیح) شیفت داده شده‌اند. برای مثال نوع positive و natural به ترتیب از ۰ و ۱ شروع شده و تا محدوده‌ی کامل اعداد ۳۱-بیتی ادامه می‌یابند. مثالهایی از اعلان نوع صحیح در زیر نشان داده شده است:

```
-- integer declarations
signal my_int      : integer range 0 to 255 := 0;
variable max_range : integer := 255;
constant start_addr : integer := 512;
```

گرچه می‌توانید از اعلان نوع پایه‌ای integer استفاده کنید، VHDL به شما اجازه می‌دهد نوع داده‌ای خودتان را که در آن محدوده را شما تعیین کرده‌اید، استفاده کنید. تا می‌توانید از این امکان در جهت افزایش خوانایی کدتان استفاده کنید. در این کار از سازه‌ی range به همراه کلمات کلیدی to و downto استفاده می‌شود. چند مثال از اعلان‌های مبتنی بر نوع integer در زیر نشان داده شده است:

```
-- integer type declarations
type scores is range 0 to 100;
type years is range -3000 to 3000;
type apples is range 0 to 15;
type oranges is range 0 to 15;
```

گرچه هر کدام از انواعی که در مثالهای اخیر معرفی شده‌اند، مبتنی بر نوع پایه‌ای `integer` هستند اما عملاً نوع جدیدی محسوب شده و قابل تخصیص به یکدیگر نیستند (زیرا نوع متفاوتی نسبت به هم محسوب می‌شوند). با در نظر گرفتن تعریفهای انجام شده در مثال اخیر، دستورات نشان داده شده در زیر، غیرمجاز هستند:

```
-- Illegal assignment statements
signal score1 : scores := 100;
signal my_apple : apples := 0;
signal my_orange : oranges := 0;

my_apple <= my_orange; -- different types
my_orange <= 24;      -- out of range
my_score <= 110;       -- out of range
```

۱۱-۹- انواع علامدار (`signed`) و بدون علامت (`unsigned`)

برای فراهم شدن امکان استفاده از انواع داده‌ای `signed` و `unsigned` باید بسته‌ی استاندارد `ieee.numeric_std` متعلق به کتابخانه‌ی IEEE را اعلان کنید. البته توجه داشته باشد که بسته‌ی استانداردنشده `std_logic_arith` نیز این دو نوع داده را تعریف کرده است؛ استفاده از کتابخانه‌های استانداردنشده به هیچ وجه توصیه نمی‌شود.

مقادیر مربوط به نوع داده‌ای `signed` در بازه‌ی -2^{N-1} تا $2^{N-1}-1$ و مقادیر مربوط به نوع داده‌ای `unsigned` در بازه‌ی ۰ تا 2^N-1 قرار دارند که در آن، N تعداد بیتها است.

از انواع داده‌ای `signed` و `unsigned` می‌توان برای متغیرهای داخلی و نیز پورتهای مدار استفاده کرد. همچنین کتابخانه‌های `ieee.numeric_unsigned` و `ieee.numeric_signed` توابع ریاضی و نیز توابع تبدیل نوع را برای کار با این انواع فراهم کرده‌اند.

از برخی نظرها، به ویژه از نظر نحوه اعلان، انواع `signed` و `unsigned` را می‌توان مشابه با نوع `std_logic_vector` دانست. بنابراین ممکن است برای شما این سوال پیش بیاید که پس چرا مانیاز داریم (گاهی اوقات) از انواع `signed` و `unsigned` به جای نوع `std_logic_vector` استفاده کنیم؟ جواب به این سوال در برنامه‌ی ۱-۱۱، به ویژه در خط شماره ۱۷ و ۱۸ نشان داده شده است. ما نباید از نوع

برای تعریف متغیر یا سیگنالی که قرار است از نظر عددی معنادار باشد (یعنی با آن مانند یک عدد و به منظور استفاده در انجام محاسبات رفتار می‌شود)، استفاده کنیم. از نوع std_logic_vector فقط باید برای تعریف «گروهی از بیت‌ها» استفاده شود. هر زمان این گروه از بیت‌ها دیگر نباید صرفاً یک «گروه» از بیتها باشند بلکه با یکدیگر «یک عدد عالمدار یا بدون علامت» تشکیل می‌دهند، دیگر نباید از نوع std_logic_vector استفاده کنیم بلکه باید از انواعی مانند signed، unsigned و حتی integer استفاده شود.

Listing 11.1: Use of unsigned types in your code.

```

1 -- library declaration
2 library IEEE;
3   use IEEE.std_logic_1164.all; -- defines std_logic_vector type
4   use IEEE.numeric_std.all;    -- defines signed and unsigned types
5 -- entity
6 entity double_sum is
7   Port (
8     in1      : in  std_logic_vector (7 downto 0);
9     in2      : in  std_logic_vector (7 downto 0);
10    out1     : out std_logic_vector (7 downto 0));
11    unsig_in : in  unsigned(7 downto 0);
12    unsig_out: out unsigned(7 downto 0));
13  end double_sum;
14 -- architecture
15 architecture arch of sum is
16 begin
17   out1 <= in1 + 1;  -- ILLEGAL OPERATION, 1 is an integer
18   out1 <= in1 + in2;-- ILLEGAL OPERATION, addition is not defined
19   unsig_out <= unsig_in + 1;                      -- legal operation
20   unsig_out <= unsigned(in1) + 1;                  -- legal operation
21   out1 <= std_logic_vector(unsigned(in1) + 1);-- legal operation
22 end arch;

```

به عنوان آخرین نکته در اینجا این مطلب قابل توجه است که اگر در برنامه‌ی ۱۱-۱ از کتابخانه و بسته‌ی استانداردنشده‌ی std_logic_arith استفاده کنیم دیگر دستوری مانند $out1 \leq in1 + 1$ مجاز و قابل استفاده است و بنابراین، خیلی کارها ساده‌تر و راحت‌تر می‌شود. اما باز هم تاکید می‌کنیم که استفاده از کتابخانه‌های استانداردنشده توصیه نمی‌شود.

۱۱-۱۰ انواع std_logic

در این کتاب تاکنون برای نمایش سیگنال‌های دیجیتال از نوع std_logic استفاده کردہ‌ایم. اما نوعی که شبیه به std_logic بوده و تاکنون نه معرفی شده و نه استفاده شده، نوع bit است. نوع bit تنها قادر به داشتن مقادیر '۰' و '۱' است. به ظاهر همین دو مقدار برای طراحی مدارات دیجیتال کفایت می‌کنند اما

در عمل این دو مقدار کمی محدود کننده هستند. نوع `std_logic` به دلیل تنوع مقادیر و جامع بودن خود نسبت به نوع `bit` ترجیح داده می‌شود. نوع `std_logic` در بسته‌ی `ieee.std_logic_1164` تعریف شده است.

نوع `std_logic` رسماً یک نوع شمارش شده است که دو مقدار از مجموعه‌ی مقادیر تعریف شده برای آن مقادیر '0' و '1' هستند. تعریف واقعی این نوع در برنامه‌ی ۱۱-۲ نشان داده شده است. نوع `std_logic` نوع تعیین تکلیف شده^۱ است. منظور از «تعیین تکلیف» این است که برخلاف حالتِ نوع `std_ulogic`, این اجازه به شما داده شده است که چند مقدار (یا «درایور») مختلف به سیگنالی از نوع `std_logic` اعمال کنید و در این صورت ابهامی پیش نمی‌آید که نتیجه، چه خواهد شد؟ (کامپایلر ایراد و خطایی صادر نمی‌کند).

Listing 11.2: Declaration of the `std_logic` enumerated type.

```
type std_logic is ( 'U', -- uninitialised
                     'X', -- forcing unknown
                     '0', -- forcing 0
                     '1', -- forcing 1
                     'Z', -- high impedance
                     'W', -- weak unknown
                     'L', -- weak 0
                     'H', -- weak 1
                     '-' -- unspecified (do not care)
                );

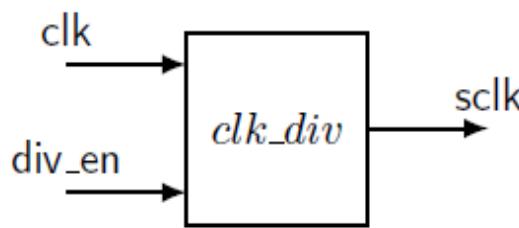
```

نوع `std_logic` مشابه با نوع `character` از علامت '' برای تعریف مقادیر خود استفاده می‌کند. اگر چه نوع `std_logic` قادر است ۹ مقدار مختلف داشته باشد، اما در این کتاب تنها از مقادیر '0', '1', 'Z', 'W', 'L', 'H', '-' استفاده می‌شود. عموماً مقدار 'Z' در هنگام کار با ساختارهای گذرگاهی استفاده می‌شود. اگر سیگنالی به حالت امپدانس بالا (یعنی مقدار 'Z') برود به این معنا است که دیگر به منبعی متصل نبوده و عملاً از مدار کنار گذاشته شده است. نکته‌ی آخر در اینجا این است که مقادیر مجاز در نوع `std_logic` به صورت کاراکتری تعریف شده‌اند لذا باید عیناً مطابق با آن چه در برنامه‌ی ۱۱-۲ آمده، استفاده شوند (یعنی فقط به صورت حروف بزرگ نوشته شوند؛ استفاده از حروف کوچک موجب بروز خطا می‌شود).

مثال ۲۶: یک مدار تقسیم‌کننده‌ی کلاک طراحی کنید که فرکانس سیگنال ورودی را بر ۶۴ تقسیم کند. این مدار همان طور که در دیاگرام نشان داده شده است، دارای دو ورودی است: ورودی `div_en`

^۱ Resolved

برافراشته^۱ شود (یعنی '1، شود)، اجازه می‌دهد که مدار کار تقسیم فرکانسی را انجام دهد؛ یعنی خروجی فرکانسی برابر $1/64$ فرکانس سیگنال ورودی clk داشته باشد. اگر div_en برافراشته نباشد، خروجی sclk در سطح پایین (یعنی مقدار '0)، قرار می‌گیرد.



حل: همان طور که قبل^۲ گفته شد برای مداراتی که پیچیده و پیچیده‌تر شوند، روش‌های زیادی برای حل و نوشتند کد VHDL وجود خواهد داشت. یکی از این راه‌حل‌ها که نشان دهنده‌ی چند نکته از نکاتی که تاکنون معرفی کردہ‌ایم، است در برنامه‌ی ۱۱-۳ نشان داده شده است. چند نکته‌ی مهم در این برنامه عبارتند از:

- اعلان نوع برای my_count در بدنی معماری و قبل از دستور begin آمده است.
- برای متغیر max_count از نوع ثابت استفاده شده است. این کار به منظور تنظیم سریع فرکانس کلک انجام شده است. البته اهمیت این مطلب در این مثال چندان مشخص نیست زیرا متغیر max_count تنها یک بار استفاده شده است (اگر در جاهای متعددی از کد از این متغیر استفاده می‌شود آن‌گاه تنها با یک بار تغییر مقدار ثابت همه‌ی این موارد به طور خودکار تنظیم و تصحیح می‌شوند -۳).
- متغیر در داخل بدنی فرآیند و قبل از دستور begin مربوط به فرآیند، اعلان شده است.

پیاده‌سازی تقسیم‌کننده‌ی فرکانسی که سیگنال کلک مشخص اولیه را دریافت کرده و سیگنال کلک ثانویه با فرکانس بالاتر یا پایین‌تر از فرکانس اولیه را در خروجی تولید می‌کند، کاری متداول و رایج در VHDL است. البته در حالت معمولی و در عمل، این کار به کمک بلوک‌های مدیریت کلک از قبل ساخته شده و موجود در FPGAها انجام می‌شود. مثالهایی از این بلوک‌ها عبارتند از مدیر کلک دیجیتال (DCM)^۴، مدیر کلک حالت ترکیبی (MMCM)^۳، و حلقه‌ی قفل شده در فاز (PLL)^۴.

¹ Asserted

² Digital Clock Manager

³ Mixed Mode Clock Manager

⁴ Phase Locked Loop

Listing 11.3: Solution to Example 26.

```

1 -- library declaration
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 use IEEE.numeric_std.all;
5
6
7 -- entity
8 entity clk_div is
9 Port (
10     clk      : in  std_logic;
11     div_en   : in  std_logic;
12     sclk     : out std_logic);
13 end clk_div;
14
15 -- architecture
16 architecture my_clk_div of clk_div is
17     type my_count is range 0 to 100;      -- user-defined type
18     constant max_count : my_count := 31; -- user-defined constant
19     signal tmp_sclk : std_logic;         -- intermediate signal
20 begin
21     my_div: process (clk,div_en)
22         variable div_count : my_count := 0;
23     begin
24         if (div_en = '0') then
25             div_count := 0;
26             tmp_sclk <= '0';
27         elsif (rising_edge(clk)) then
28             -- divider enabled
29             if (div_count = max_count) then
30                 tmp_sclk <= not tmp_sclk;    -- toggle output
31                 div_count := 0;           -- reset count
32             else
33                 div_count := div_count + 1; -- count
34             end if;
35         end if;
36     end process my_div;
37     sclk <= tmp_sclk;                   -- final assignment
38 end my_clk_div;

```

The VHDL implementation of frequency divider that takes a certain clock

۱۱- نکات مهم

- هر زمان که «گروهی از بیت‌ها» دیگر تنها مجموعه‌ای از بیت‌ها نبوده و به عنوان یک عدد علامدار یا بدون علامت و یا عدد صحیح ایفای نقش می‌کنند، استفاده از انواع signed و unsigned توصیه می‌شود. یک نمونه مثال، متغیری است که در یک شمارنده استفاده شده و طبعاً هیچ توجیهی برای استفاده از آن به عنوان یک نوع std_logic_vector وجود ندارد. در این خصوص به خطوط ۱۷، ۲۲، و ۳۳ از برنامه‌ی ۱۱-۳ مراجعه کنید.

- هر زمان بخواهید از انواع signed و unsigned استفاده کنید، باید کتابخانه‌ی استاندارد ieee.numeric_std را در کد خود فراخوانی کنید. این کتابخانه‌ی استاندارد IEEE همواره نسبت به کتابخانه‌ی استاندارد نشده‌ی std_logic_arith ترجیح داده می‌شود.
- استفاده از کتابخانه‌های استاندارد نشده‌ی std_logic_unsigned و std_logic_signed و std_logic_arith است. مربوط به شرکت Synopsys توصیه نمی‌شود.
- شما نمی‌توانید یک سیگنال از نوع std_logic_vector را مستقیماً افزایش/کاهش دهید. برای این کار لازم است ابتدا آن را به یکی از انواع unsigned، signed و یا integer تبدیل کنید:

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all

signal val1, val2 : std_logic_vector( 31 downto 0 );
val2 <= val1 + 1;      -- ILLEGAL OPERATION
val2 <= std_logic_vector( unsigned(val1) + 1 );
```

لسته شامخورد

فصل دوازدهم

ساختارهای حلقه‌نویسی

با پیچیده‌تر شدن مدارات دیجیتال در طراحی تان، انتظار شما هم از VHDL در زمینه‌ی قابلیت انعطاف و قدرت عملیاتی بیشتر و بیشتر می‌شود. یکی از این قابلیت‌ها امکان حلقه‌نویسی است که در این فصل به آن پرداخته می‌شود.

در VHDL دو نوع حلقه وجود دارد: حلقه‌های `for` و حلقه‌های `while`. این دو نوع دستور حلقه از نوع ترتیبی بوده و بنابراین، فقط داخل یک فرآیند قابل استفاده هستند.

۱۲-۱- حلقه‌های `for` و `while`

هدف از یک سازه‌ی حلقه برقراری امکان تکرار و وقوع پی در پی تعدادی دستورات کدنویسی است. برای برقراری این امکان دو نوع حلقه وجود دارد که مشابه با سایر زبانهای برنامه‌نویسی سطح بالا، از یک نوع حلقه می‌توان با اندکی تغییرات و تصحیحات به نوع دیگر رسید؛ البته در VHDL شما باید به خوانایی و قابل فهم بودن توجه جدی کنید. با این نکته‌ی مهم، به بررسی تفاوت عملیاتی بین دستورات حلقه‌نویسی `for` و `while` می‌پردازیم. برای بررسی این تفاوت، دو کد نشان داده شده در برنامه‌ی ۱۲-۱ را ملاحظه کنید.

Listing 12.1: The basic structure of the `for` and `while` loops.

| | |
|--|---|
| <pre>-- for loop my_label: for index in a_range loop sequential statements... end loop my_label;</pre> | <pre>-- while loop my_label: while (condition) loop sequential statements... end loop my_label;</pre> |
|--|---|

تفاوت اساسی بین دو نوع حلقه، تعداد دفعاتی است که حلقه تکرار می‌شود. به طور معادل، تفاوت در این است که کدام شرط باعث می‌شود مدار از تکرار خارج شود. اگر شما از قبل تعداد دفعات تکرار را می‌دانید، بهتر است از دستور `if` استفاده کنید زیرا در این دستور، تعداد دفعات تکرار به صراحت مشخص می‌شود. در غیر این صورت بهتر است از دستور `while` استفاده کنید؛ در اینجا تکرار زمانی خاتمه می‌یابد که شرط معینی برقرار شود. هر حلقه دارای یک برچسب است که گذاشتن و استفاده از آن



اختیاری است لذا در برنامه‌ی ۱۲-۱ به صورت خمیده^۱ نشان داده شده است. شما بهتر است همیشه از این برچسب‌ها به منظور روشن کردن هدف و منظورتان استفاده کنید. به ویژه در حلقه‌های تودرتو و در دستورات کنترل حلقه از برچسب‌ها استفاده کنید.

۱۲-۱-۱ حلقه‌های for

شکل پایه‌ای حلقه‌ی for در برنامه‌ی ۱۲-۱ نشان داده شد. این حلقه از یک نوع مقدار اندیس برای تکرار و تغییر در یک بازه‌ی مشخص استفاده می‌کند. دو گزینه برای تعیین بازه‌ی تغییرات این اندیس در اختیار داریم: (۱) تعیین حدود بازه در دستور for، (۲) استفاده از یک بازه‌ی از قبل اعلان شده. مثالی از این روشها در زیر نشان داده شده است:

```
for cnt_val in 0 to 24 loop
    -- sequential_statements
end loop;
```

```
for cnt_val in 24 downto 0 loop
    -- sequential_statements
end loop;
```

```
type my_range is range 0 to 24;
--
for cnt_val in my_range loop
    -- sequential_statements
end loop;
```

```
type my_range is range 24 downto 0;
--
for cnt_val in my_range loop
    -- sequential_statements
end loop
```

متغیر اندیس استفاده شده در دستور for ویژگی‌های عجیبی دارد که در ادامه معرفی می‌شوند. اگر به نکاتی که در ادامه گفته می‌شود دقت و توجه کافی داشته باشید، جلوی بسیاری از خطاهای صادر شده از سوی ابزار سنتز گرفته می‌شود و از سوی دیگر، در وقت شما هم صرفه‌جویی می‌شود. نکته‌ی بعدی این که توجه کنید که بدنی‌ی حلقه به صورت تورفته^۲ نوشته شود تاکد شما خواناتر شود. همیشه به دنبال افزایش و بهبود خوانایی کد خودتان باشید.

- متغیر اندیس نیازی به اعلان ندارد زیرا به طور ضممنی همیشه این کار انجام می‌شود.
- نمی‌توان روی متغیر حلقه هیچ تخصیصی انجام داد. اما از آن می‌توان در محاسبات و در داخل بدنی‌ی حلقه استفاده کرد.
- متغیر اندیس فقط با گامی به اندازه‌ی واحد می‌تواند در محدوده‌ی تعریف شده برای آن تغییر کند.

¹ Italic

² Indented

- شناسه‌ای که برای متغیر اندیس انتخاب می‌شود می‌تواند همنام با هر سیگنال یا متغیر استفاده شده در سرتاسر کد بوده و هیچ خطای ناشی از هم‌پوشانی نامها رخ نخواهد داد؛ البته این کار اصلاً توصیه نشده و نشان‌دهنده‌ی برنامه‌نویسی بد است.
 - بازه‌ی مشخص شده برای متغیر اندیس (در حالتی که بیرون از اعلان حلقه مشخص شده باشد)، می‌تواند از هر نوع شمارش شده‌ای باشد.
- وبالاخره این که همان طور که در برنامه‌ی اخیر نشان داده شده است، حلقه‌های `for` را با گزینه‌ی `downto` نیز می‌توان پیاده‌سازی کرد.

۱۲-۱-۲ حلقه‌های while

حلقه‌های `while` تا حدی ساده‌تر از حلقه‌های `loop` هستند زیرا فاقد یک متغیر اندیس هستند. تفاوت اساسی بین حلقه‌های `while` و `for` در این است که اعلان حلقه‌ی `for` شامل یک شرط درون-ساخته جهت تعیین زمان خاتمهٔ حلقه است. اولین نکته‌ای که شما باید در مورد حلقه‌های `while` به خاطر بسپارید این است که کد شما باید به طریقی زمان خاتمه دادن به حلقه و خروج از آن را مشخص کرده باشد. چند مثال از حلقه‌ی `while` در برنامه‌ی زیر نشان داده شده است. در مورد مثال سمت راست، نیازی به تذکر این نکته نیست که این حلقه بهتر است با دستور `for` پیاده‌سازی شود زیرا در حقیقت، تعداد دفعات تکرار حلقه در این مثال کاملاً مشخص است!.

```
constant max_fib : integer := 2000;
variable fib_sum : integer := 1;
variable tmp_sum : integer := 0;

while (fib_sum < max_fib) loop
    fib_sum := fib_sum + tmp_sum;
    tmp_sum := fib_sum;
end loop;
--
```

```
constant max_num : integer := 10;
variable fib_sum : integer := 1;
variable tmp_sum : integer := 0;
variable int_cnt : integer := 0;

while (int_cnt < max_num) loop
    fib_sum := fib_sum + tmp_sum;
    tmp_sum := fib_sum;
    int_cnt := int_cnt + 1;
end loop;
```

۱۲-۱-۳ کنترل حلقه: دستورات `next` و `exit`

مشابه با زبانهای برنامه‌نویسی سطح بالا، در VHDL نیز چند گزینه‌ی اضافه برای اعمال کنترل بیشتر روی حلقه در نظر گرفته شده است. این گزینه‌ها شامل دو دستور `next` و `exit` است. این دو دستور مشابه با معادلهای خود در دیگر زبانهای برنامه‌نویسی هستند. این دو دستور را در هر یک از حلقه‌های `while` و `for` می‌توان به کار گرفت.

دستور next

دستور **next** موجب نادیده‌گرفته شدن مابقی دستورات تا انتهای حلقه شده و باعث می‌شود دور بعدی حلقه بلا فاصله شروع شود؛ طبیعتاً در حلقه‌ی **for**، قبل از شروع دور بعدی تکرار، متغیر اندیس یکی اضافه خواهد شد. در حلقه‌ی **while** نیز به کدنویسی کاربر بستگی دارد که کدنویسی خود را طوری انجام داده باشد که با وقوع دستور **next** از ادامه و خاتمه‌ی مناسب کد اطمینان حاصل شود. دستور **next** دو شکل استفاده دارد که هر دو شکل در برنامه‌ی بعدی نشان داده شده‌اند. در این برنامه، دو مثال آورده شده است که از دستور **next** استفاده کرده‌اند و لزوماً نشان‌دهنده‌ی یک برنامه‌نویسی خوب یا حتی یک کار معنادار نیستند.

```
variable my_sum : integer := 0;
--  
for cnt_val in 0 to 50 loop
    if (my_sum = 20) then
        next;
    end if;
    my_sum := my_sum + 1;
end loop;
```

```
variable my_sum : integer := 0;
--  
while (my_sum < 300) loop
    next when (my_sum = 20);
    my_sum := my_sum + 1;
end loop;
--  
--
```

دستور exit

دستور **exit** موجب خاتمه‌ی فوری حلقه و خروج از آن می‌شود. این دستور در هر دو حلقه‌ی **for** و **while** قابل استفاده است. با برخورد به این دستور، کنترل به اولین دستور پس از کلمه‌ی کلیدی **end** (منتظر با حلقه‌ی مزبور) منتقل می‌شود. در حلقه‌های تودرتو نیز می‌توان از دستور **exit** استفاده کرد. دستور **exit** نیز مانند دستور **next** دو شکل دارد که مشابه با هم هستند. مثالهایی از این دو شکل در برنامه‌ی زیر نشان داده شده است.

```
variable my_sum : integer := 0;
--  
for cnt_val in 0 to 50 loop
    if (my_sum = 20) then
        exit;
    end if;
    my_sum := my_sum + 1;
end loop;
```

```
variable my_sum : integer := 0;
--  
while (my_sum < 300) loop
    exit when (my_sum = 20);
    my_sum := my_sum + 1;
end loop;
--  
--
```

فصل سیزدهم

مدارات دیجیتال استاندارد در VHDL

همان طور که می‌دانید (یا همان طور که خواهید دانست)، حتی پیچیده‌ترین مدارات دیجیتال نیز از مجموعه‌ی نسبتاً کوچکی از مدارات دیجیتال استاندارد به همراه تعدادی سیگنال کنترلی تشکیل شده‌اند. این مدارات دیجیتال استاندارد شامل مداراتی مانند مالتی‌پلکسراها (MUX)، دیکدرها، شمارنده‌ها، مقایسه‌گرها، ثباتها (رجیسترها)، و غیره هستند. هنر طراحی دیجیتال به کمک VHDL روی انتخاب مناسب و اتصال‌دهی مناسب این مدارات دیجیتال استاندارد متمرکز شده است.

موثرترین رویکرد برای بهره‌گیری از مدارات دیجیتال استاندارد، استفاده از کدهای موجود برای این مدارات و اصلاح آنها مطابق با نیازهای ما در یک طراحی خاص است. بنابراین شما می‌توانید با اطمینان از کارکرد صحیح مدارات دیجیتال پایه، به طراحی مدارات پیچیده‌تر پردازید. در ادامه، برخی مدارات دیجیتال استاندارد به همراه کد VHDL هر کدام نشان داده می‌شود. این مدارات در اندازه‌ها و عرضهای مختلف قابل استفاده هستند. توجه کنید که کدهای VHDL نشان داده شده، تنها نمونه‌ای از پیاده‌سازی بوده و به این معنا نیست که تنها کد ممکن برای پیاده‌سازی هستند. حتی این کدها می‌توانند نقطه‌ی شروعی برای شما باشند تا با اصلاح این کدها به کدهای بهتری منطبق بر نیازهایتان برسید.

13.1 RET D Flip-flop - Behavioral Model

```
-- D flip-flop: RET D flip-flop with single output
--
-- Required signals:
-----
-- CLK,D: in std_logic;
-- Q:      out std_logic;
-----

process (CLK)
begin
  if (rising_edge(CLK)) then
    Q <= D;
  end if;
end process;
```

13.2 FET D Flip-flop with Active-low Asynchronous Preset - Behavioral Model

```
-- D flip-flop: FET D flip-flop with asynchronous preset. The
-- preset input takes precedence over the synchronous input.
--
-- Required signals:
-----
-- CLK,D,S: in std_logic;
-- Q:      out std_logic;
-----

process (CLK,S)
begin
  if (S = '0') then
    Q <= '1';
  elsif (falling_edge(CLK)) then
    Q <= D;
  end if;
end process;
```

13.3 8-Bit Register with Load Enable - Behavioral Model

```
-- Register: 8-bit Register with load enable.  
--  
-- Required signals:  
-----  
-- CLK,LD: in std_logic;  
-- D_IN:   in std_logic_vector(7 downto 0);  
-- D_OUT:  out std_logic_vector(7 downto 0);  
-----  
process (CLK)  
begin  
    if (rising_edge(CLK)) then  
        if (LD = '1') then    -- positive logic for LD  
            D_OUT <= D_IN;  
        end if;  
    end if;  
end process;  
--
```



13.4 Synchronous Up/Down Counter - Behavioral Model

```

1
2 -----
3 -- Counter: synchronous up/down counter with asynchronous
4 -- reset and synchronous parallel load.
5 -----
6 -- library declaration
7 library IEEE;
8 use IEEE.std_logic_1164.all;
9 use IEEE.numeric_std.all;
10
11 entity COUNT_8B is
12     port ( RESET,CLK,LD,UP : in  std_logic;
13             DIN : in  std_logic_vector (7 downto 0);
14             COUNT : out std_logic_vector (7 downto 0));
15 end COUNT_8B;
16 architecture my_count of COUNT_8B is
17     signal t_cnt : unsigned(7 downto 0); -- internal counter signal
18 begin
19     process (CLK, RESET)
20     begin
21         if (RESET = '1') then
22             t_cnt <= (others => '0'); -- clear
23         elsif (rising_edge(CLK)) then
24             if (LD = '1') then      t_cnt <= unsigned(DIN); -- load
25             else
26                 if (UP = '1') then t_cnt <= t_cnt + 1; -- incr
27                 else                t_cnt <= t_cnt - 1; -- decr
28             end if;
29         end if;
30     end process;
31     COUNT <= std_logic_vector(t_cnt);
32 end my_count;
33 --

```



13.5 Shift Register with Synchronous Parallel Load - Behavioral Model

```

-----
-- Shift Register: unidirectional shift register with synchronous
-- parallel load.
--
-- Required signals:
-----
-- CLK, D_IN:    in std_logic;
-- P_LOAD:       in std_logic;
-- P_LOAD_DATA:  in std_logic_vector(7 downto 0);
-- D_OUT:        out std_logic;
--
-- Required intermediate signals:
signal REG_TMP: std_logic_vector(7 downto 0);
-----
process (CLK)
begin
  if (rising_edge(CLK)) then
    if (P_LOAD = '1') then
      REG_TMP <= P_LOAD_DATA;
    else
      REG_TMP <= REG_TMP(6 downto 0) & D_IN;
    end if;
  end if;
  D_OUT <= REG_TMP(7);
end process;
--
```



13.6 8-Bit Comparator - Behavioral Model

```
-- Comparator: Implemented as a behavioral model. The outputs
-- include equals, less than and greater than status.
--
-- Required signals:
-----
-- CLK:          in  std_logic;
-- A_IN, B_IN:   in  std_logic_vector(7 downto 0);
-- ALB, AGB, AEB: out std_logic
-----
process(CLK)
begin
    if ( A_IN < B_IN ) then ALB <= '1';
    else ALB <= '0';
    end if;

    if ( A_IN > B_IN ) then AGB <= '1';
    else AGB <= '0';
    end if;

    if ( A_IN = B_IN ) then AEB <= '1';
    else AEB <= '0';
    end if;
end process;
```



13.7 BCD to 7-Segment Decoder - Data-Flow Model

```
-- BCD to 7-Segment Decoder: Implemented as combinatorial circuit.
-- Outputs are active low; Hex outputs are included. The SSEG format
-- is ABCDEFG (segA, segB etc.)
--
-- Required signals:
--
-- BCD_IN: in std_logic_vector(3 downto 0);
-- SSEG:    out std_logic_vector(6 downto 0);
--
with BCD_IN select
  SSEG <= "0000001" when "0000",      -- 0
  "1001111" when "0001",      -- 1
  "0010010" when "0010",      -- 2
  "0000110" when "0011",      -- 3
  "1001100" when "0100",      -- 4
  "0100100" when "0101",      -- 5
  "0100000" when "0110",      -- 6
  "0001111" when "0111",      -- 7
  "0000000" when "1000",      -- 8
  "0000100" when "1001",      -- 9
  "0001000" when "1010",      -- A
  "1100000" when "1011",      -- b
  "0110001" when "1100",      -- C
  "1000010" when "1101",      -- d
  "0110000" when "1110",      -- E
  "0111000" when "1111",      -- F
  "1111111" when others;     -- turn off all LEDs
--
```



13.8 4:1 Multiplexer - Behavioral Model

```
-- A 4:1 multiplexer implemented as behavioral model using case
-- statement.

--
-- Required signals:
--

-- SEL:      in std_logic_vector(1 downto 0);
-- A, B, C, D: in std_logic;
-- MUX_OUT:   out std_logic;

process (SEL, A, B, C, D)
begin
    case SEL is
        when "00" => MUX_OUT <= A;
        when "01" => MUX_OUT <= B;
        when "10" => MUX_OUT <= C;
        when "11" => MUX_OUT <= D;
        when others => (others => '0');
    end case;
end process;
```

13.9 4:1 Multiplexer - Data-Flow Model

```
-- A 4:1 multiplexer implemented as data-flow model using a
-- selective signal assignment statement.

--
-- Required signals:
--

-- SEL:      in std_logic_vector(1 downto 0);
-- A, B, C, D: in std_logic;
-- MUX_OUT:   out std_logic;

with SEL select
    MUX_OUT <= A when "00",
                  B when "01",
                  C when "10",
                  D when "11",
                  (others => '0') when others;
```

13.10 Decoder

```
-- Decoder: 3:8 decoder with active high outputs implemented as
-- combinatorial circuit with selective signal assignment statement
--
-- Required signals:
--
-- D_IN: in std_logic_vector(2 downto 0);
-- FOUT: out std_logic_vector(7 downto 0);
--
with D_IN select
    F_OUT <= "00000001" when "000",
                "00000010" when "001",
                "00000100" when "010",
                "00001000" when "011",
                "00010000" when "100",
                "00100000" when "101",
                "01000000" when "110",
                "10000000" when "111",
                "00000000" when others;
```

دانشگاه صنعتی شاهرود

پایان کتاب اول

بر محمد (ص) و آل پاک او صلوات
دینه حسینی شاهزاد