

# آموزش VHDL

درس طراحی مدارات FPGA

دانشگاه صنعتی شاهرود

دکتر هادی گرایلو

۱۳۹۷

بخش اول

# فصل اول

## مقدمه‌ای بر VHDL

حروف V در کلمه‌ی VHDL مخفف VHSIC: Very High-Speed Integrated Circuit و HDL نیز مخفف Hardware Description Language است.

برخلاف اکثر زبانهای برنامه‌نویسی که تاکنون دیده‌اید، زبان VHDL به دنبال توصیف یک سخت‌افزار است. زبان‌های برنامه‌نویسی مزبور ذاتی ترتیبی دارند اما VHDL این گونه نیست. VHDL یک زبان همزمان<sup>۱</sup> است. به این معنا که دستورات VHDL در حالت معمول، به طور «همزمان» شروع به اجرا شدن می‌کنند (مستقل از اندازه‌ی پیاده‌سازی شما).

زبان VHDL مدلی از مدار دیجیتال ارائه می‌دهد که توسط ابزارهای نرم‌افزاری در طی فرآیندی به نام سنتز به قالبی ترجمه می‌شود که به راحتی از آن می‌توان برای ایجاد مدارات واقعی استفاده کرد. به فرآیند تفسیر کد VHDL و تولید یک پیاده‌سازی فیزیکی از مدار به منظور برنامه‌ریزی یک افزاره مانند FPGA، «سنتز» گفته می‌شود.

### ۱- قوانین طلایی VHDL

چند مورد را فراموش نکنید.

**VHDL یک زبان طراحی سخت‌افزار است.** در هنگام کار کردن با VHDL شما در حال برنامه‌نویسی نیستید بلکه در حال طراحی سخت‌افزار هستید. معنای این جمله چیست؟ معنایش این است که به استثناء برخی موقع خاص، خطوط دستوری موجود در کد شما همگی تقریباً در یک زمان شروع به اجرашدن می‌کنند. اگر کد VHDL شما خیلی شبیه به یک زبان برنامه‌نویسی سطح بالا است، احتمالاً آن را بد نوشه‌اید. این مطلب بسیار مهم است.

<sup>۱</sup> Concurrent language

حتماً ذهنیتی در این مورد داشته باشید که ساخت افزار شما شبیه به چه چیزی باید باشد. درست است که VHDL بسیار قدرتمند است اما اگر فهمی از سازه‌های اساسی دیجیتال<sup>۱</sup> نداشته باشید، احتمالاً قادر به تولید مدارات دیجیتال نخواهید بود. علت این امر این است که طراحی دیجیتال شبیه به برنامه‌نویسی زبان سطح بالا است؛ از این جهت که حتی پیچیده‌ترین برنامه‌های نوشته شده در هر سطحی قابل شکسته شدن و تجزیه به سازه‌های پایه‌ای برنامه‌نویسی هستند. در طراحی دیجیتال نیز حتی پیچیده‌ترین مدارات دیجیتال قابل توصیف بر حسب سازه‌های پایه‌ای دیجیتال هستند. بنابراین می‌توان گفت اگر شما نتوانید مدار دیجیتال مدنظرتان را به سازه‌های پایه‌ای دیجیتال تفکیک و تجزیه کنید، از زبان VHDL نیز احتمالاً به نحو بد و نامناسب استفاده خواهید کرد.

# دانشگاه صنعتی شاهرود

<sup>1</sup> Basic digital construct

## فصل دوم

### VHDL مبانی

#### ۲-۱ حساسیت به بزرگی/کوچکی حروف

VHDL به بزرگی/کوچکی حروف حساس نیست. بنابراین، برای مثال دو دستور زیر یکسان و معادل هم هستند. توجه کنید که اینها صرفاً مثال بوده و هدف ما نشان دادن یک کد خوب نیست.

Listing 2.1: An example of VHDL case insensitivity.

Dout <= A and B;	doUt <= a And b;
------------------	------------------

#### ۲-۲ فاصله (فضای خالی)

VHDL به فاصله (فضای خالی)<sup>۱</sup> (شامل کاراکترهای فاصله و تب) موجود در سند منبع حساس نیست. دو دستور زیر معنای یکسانی دارند. توجه کنید که اینها صرفاً مثال بوده و هدف ما نشان دادن یک کد خوب نیست. در مثال زیر مجدداً روی حساس نبودن به بزرگی/کوچکی تاکید شده است.

Listing 2.2: An example showing VHDL's indifference to white space.

nQ <= In_a or In_b;	nQ <=in_a OR in_b;
---------------------	--------------------

#### ۲-۳ توضیحات

توضیحات در VHDL با نماد “--” (یعنی دو کاراکتر خط تیره<sup>۲</sup>ی متوالی) شروع می‌شوند. کامپایلر VHDL هر متنی که بعد از این نماد باشد را تا انتهای خط مربوطه نادیده می‌گیرد. دو نوع شیوه‌ی توضیح‌گذاری در مثال زیر نشان داده شده است. متأسفانه در VHDL، توضیح‌گذاری بلوكی<sup>۳</sup> (یعنی نمادهایی جهت مشخص کردن ابتدا و انتهای یک بلوك از توضیحات) وجود ندارد.

<sup>1</sup> White space

<sup>2</sup> Dash

<sup>3</sup> Block-style comment

Listing 2.3: Two typical uses of comments.

```
-- This next section of code is used to blah-blah
-- This type of comment is the best fake for block-style commenting.
PS_reg <= NS_reg; -- Assign next_state value to present_state
```

استفاده‌ی مناسب از توضیحات، خوانایی و قابل فهم بودن کد شما را افزایش می‌دهد. قاعده‌ی کلی این است که هر خط یا بخشی از کد را که از نظر شما برای خواننده واضح و روشن نیست، بهتر است توضیح‌گذاری کنید. گفتن چیزی که واضح است، ارزش و ضرورتی ندارد. فهم و تصور کدی که توضیحات بسیار کمی دارد، مشکل است بنابراین خجالت را کنار بگذارید: از توضیحات کافی و مناسب استفاده کنید. تحقیقات نشان داده است که استفاده از توضیحات زیاد و مناسب نشان‌دهنده‌ی هوش و زیرکی بالا است.

## ۲-۴ پرانتزها

در استفاده از پرانتزها چندان سخت‌گیر نیست. مانند دیگر زبانهای برنامه‌نویسی، در اینجا هم برخی قوانین اولویت‌دهی بین عملگرهای موجود در VHDL وجود دارد. گرچه می‌توانید تمام این قوانین را حفظ و در کدتان اعمال کنید اما با استفاده از پرانتزها می‌توانید مطمئن شوید که اولویت مدنظرتان رعایت شده است و خواننده نیز به راحتی کد شما را خواننده و خوب می‌فهمد. دو دستور نشان داده شده در مثال زیر معادل و یکسان هستند اما توجه کنید که در دستور دوم، استفاده از کarakترهای فضای خالی و پرانتز موجب واضح‌تر شدن کد شده است.

Listing 2.4: Example of parentheses that can improve clarity.

```
if x = '0' and y = '0' or z = '1' then
    blah; -- some useful statement
    blah; -- some useful statement
end if;
if ( ((x = '0') and (y = '0')) or (z = '1') ) then
    blah; -- some useful statement
    blah; -- some useful statement
end if;
```

## ۲-۵ دستورات VHDL

هر دستور VHDL به نقطه‌ویرگول<sup>۱</sup> ختم می‌شود.

<sup>1</sup> Semicolon

## ۶- دستورات if، case و loop

همیشه قوانین بیان شده در زیر را در هنگام نوشتن یا خطایابی کد خود به خاطر داشته باشید تا در زمان صرفه‌جویی زیادی کنید.

- هر دستور if یک جزء then به همراه خود دارد.
- هر دستور if با یک end if; خاتمه می‌یابد.
- اگر از سازه‌ی else if می‌خواهید استفاده کنید، معادل آن در VHDL کلمه‌ی elsif است.
- هر دستور case با یک end case; خاتمه می‌یابد.
- هر دستور loop با یک end loop; خاتمه می‌یابد.

## ۷- شناسه‌ها

منظور از یک شناسه<sup>۱</sup>، نام داده شده به اقلام مختلف موجود در VHDL است. مثالهایی از شناسه‌ها در دیگر زبانهای برنامه‌نویسی عبارتند از نام متغیرها و نام توابع. مثالهایی از شناسه‌ها در VHDL عبارتند از نام متغیرها، نام سیگنالها، و نام پورتها (تمام اینها به زودی بحث می‌شوند). در زیر برخی قوانین اجباری و اختیاری مربوط به استفاده از شناسه‌ها آورده شده است:

- بهتر است شناسه‌ها خود-توصیف<sup>۲</sup> باشند؛ یعنی شناسه‌ها گویای استفاده و هدف اقلامی باشند که شناسه آنها را نمایش داده است.
- شناسه‌ها هر طول دلخواهی<sup>۳</sup> می‌توانند داشته باشند (یعنی شامل کاراکترهای زیادی باشند).
- نامهای کوتاه‌تر خواناتر هستند و نامهای طولانی حاوی اطلاعات بیشتری هستند. انتخاب طول یک شناسه به دلخواه و انتخاب برنامه‌نویس است.
- شناسه‌ها فقط می‌توانند ترکیبی از حروف الفبا (A-Z و a-z)، ارقام (0-9)، و کاراکتر زیرخط<sup>۴</sup> ("\_") باشند.
- شناسه‌ها باید با یک کاراکتر حرفی<sup>۵</sup> شروع شوند.
- شناسه‌ها نباید به کاراکتر زیرخط ختم شوند و نیز نباید شامل دو کاراکتر زیرخط متوالی باشند.
- یک نمونه نام‌گذاری مناسب: بهترین شناسه برای تابعی که موقعیت زمین را محاسبه می‌کند، می‌تواند CalcEarthPosition یا calc\_earth\_position است. همواره از یک شیوه ثابت و مشخص استفاده کنید.

<sup>1</sup> Identifier

<sup>2</sup> Self-describing

<sup>3</sup> Underscore

<sup>4</sup> Alphabetic

- یا در مثالی دیگر، بهترین نام برای متغیری که سن خودروی شما را نشان دهد، AgeMyCar یا age\_my\_car است. مجدداً این که سعی کنید به یک شیوه عمل کنید.

به خاطر داشته باشید که انتخابهای هوشمندانه برای شناسه‌ها، کد شما را خواناتر، قابل فهم‌تر، و برای دیگران گیراتر خواهد کرد. در زیر تعدادی انتخاب خوب و بد برای نام شناسه‌ها آورده شده است.

Listing 2.5: Valid identifiers.

```
data_bus      --descriptive name
WE           --classic write enable
div_flag     --real winner
port_A       --provides some info
in_bus       --input bus
clk          --classic clock
clk_in       --
clk_out      --
mem_read_data
--
```

Listing 2.6: Invalid identifiers.

```
3Bus_val    -- begins with a number
DDD         -- not self commenting
mid_$num   -- illegal character
last__val  -- consec. underscores
str_val_   -- ends with underscore
in          -- uses VHDL reserved word
@#$%       -- total garbage
it_sucks   -- try to avoid
Big_vAlUe  -- valid but ugly
pa          -- possibly lacks meaning
sim-val    -- illegal character (dash)
DDE_SUX    -- no comment
```

## ۲-۸ کلمات رزرو شده

در VHDL برعی کلمات هستند که معنای خاصی دارند. اینها «کلمات رزرو شده» نامیده شده و در کدهای VHDL نباید از آنها به عنوان شناسه استفاده کرد. برعی از این کلمات رزرو شده در زیر فهرست شده‌اند. لیست کامل این کلمات در پیوست آورده شده است.

Listing 2.7: A short list of VHDL reserved words.

access	after	alias	all	attribute	block
body	buffer	bus	constant	exit	file
for	function	generic	group	in	is
label	loop	mod	new	next	null
of	on	open	out	range	rem
return	signal	shared	then	to	type
until	use	variable	wait	while	with

## ۹-۲ شیوه‌ی کدنویسی VHDL

منظور از شیوه‌ی کدنویسی<sup>۱</sup>، ظاهر<sup>۲</sup> کد منبع VHDL است. طبعاً برخی آزادی‌ها مانند حساس نبودن به بزرگی/کوچکی حروف، بی‌تفاوتی نسبت به کاراکتر فاصله (جای خالی)، و قوانین نسبتاً آسان و سهل استفاده از پرانتزها درجاتی از بی‌نظمی را در کد VHDL پدید می‌آورند. بنابراین، در کدنویسی تاکید روی خوانایی کد است. البته مtasفانه خوانایی کد امری نسبی و سلیقه‌ای است.

# دستگاه حسنهٔ شامرود

<sup>۱</sup> Coding style

<sup>۲</sup> Appearance

## فصل سوم

### VHDL واحدهای طراحی

توصیفات VHDL از مدارات مبتنی بر رویکرد جعبه‌ی سیاه است. دو قسمت اصلی هر طرح سلسله‌مراتبی یکی جعبه‌ی سیاه و دیگری چیزهایی است که درون جعبه‌ی سیاه قرار می‌گیرد (که البته خود می‌تواند شامل جعبه‌ی سیاه دیگری باشد). در VHDL به این جعبه‌ی سیاه، موجودیت (Entity) و به چیزهایی که درون جعبه‌ی سیاه قرار می‌گیرد، معماری (Architecture) گفته می‌شود. به همین دلیل، موجودیت و معماری در VHDL خیلی به هم مرتبط هستند. همان طور که شما احتمالاً بتوانید تصور کنید، ایجاد موجودیت نسبتاً ساده است حال آن که بخش عمده‌ی زمانی کدنویسی VHDL صرف نوشتن صحیح معماری می‌شود. رویکرد ما در معرفی مقدمه‌ای بر نوشتن کد VHDL مبتنی بر توصیف موجودیت و سپس حرکت به سوی جزئیات نوشتن معماری است. آشنا شدن با موجودیت، به شما در یادگیری تکنیکهای توصیف معماری کمک می‌کند.

#### ۱-۳ موجودیت

سازه‌ی «موجودیت» در VHDL روشی برای بسط عملکرد یک توصیف مداری به یک سطح بالاتر را فراهم می‌کند. موجودیت در واقع یک بسته و لفافه برای مدار سطح پایین است و چگونگی ارتباط و تعامل جعبه‌ی سیاه با دنیای بیرون را توصیف می‌کند. با توجه به این که VHDL مدارات دیجیتال را توصیف می‌کند، پس موجودیت به سادگی ورودی‌ها و خروجی‌های این مدار دیجیتال را فهرست می‌کند. بر حسب اصطلاحات و تعاریف VHDL، جعبه‌ی سیاه به کمک «اعلان موجودیت»<sup>۱</sup> توصیف می‌شود. گرامر اعلان موجودیت به صورت زیر است:

<sup>1</sup> Entity declaration

Listing 3.1: The entity declaration in VHDL.

```
entity my_entity is
port(
    port_name_1 : in    std_logic ;
    port_name_2 : out   std_logic;
    port_name_3 : inout std_logic ); --do not forget the semicolon
end my_entity; -- do not forget this semicolon either
```

نام موجودیت my\_entity را تعریف می‌کند. بخش بعدی شامل لیست سیگنالهایی است که از مدار مربوطه در اختیار دنیای بیرون قرار داده می‌شوند. به این لیست معمولاً مشخص کردن اتصال<sup>۱</sup> گفته می‌شود. port\_name\_x شناسه‌ای است که برای ایجاد تمایز و تفاوت بین سیگنالهای مختلف استفاده شده است. کلمه‌ی کلیدی بعدی (یعنی، in) تعیین کننده‌ی جهت سیگنال نسبت به موجودیت است؛ در واقع، سیگنال‌ها یا می‌توانند وارد موجودیت شوند، یا از آن خارج شوند، یا هر دو کار را انجام دهند. این نوع سیگنالهای ورودی و خروجی به ترتیب با کلمات کلیدی in، out، و inout<sup>۲</sup> مشخص می‌شوند. کلمه‌ی کلیدی بعدی (یعنی std\_logic) اشاره به نوع داده‌ای دارد که پورت حمل می‌کند. چندین نوع داده در VHDL وجود دارد اما ما بیشتر با نوع std\_logic و انواع مشتق شده از آن کار می‌کنیم. اطلاعات بیشتر راجع به انواع داده بعداً ارائه خواهد شد.

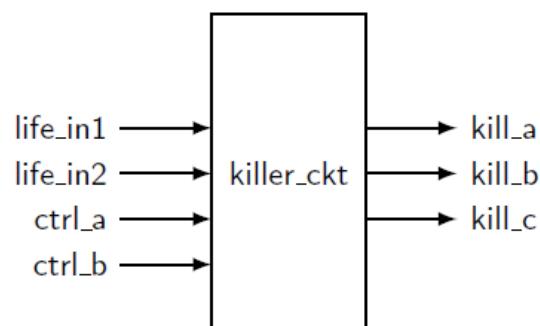
هرگاه بخواهید کد VHDL نسبتاً پیچیده‌ای بنویسید، نیاز دارید که خود را به فایلها، توابع، و بسته‌های مختلفی بشکنید تا به شما در مدیریت بهتر کد کمک کنند. در زیر مثالی از یک جعبه‌ی سیاه و کد مربوط به توصیف آن مشاهده می‌شود.

Listing 3.2: VHDL entity declaration.

```
-- interface description --
-- of killer_ckt --
```

---

```
entity killer_ckt is
port (
    life_in1      : in  std_logic;
    life_in2      : in  std_logic;
    ctrl_a, ctrl_b : in  std_logic;
    kill_a        : out std_logic;
    kill_b, kill_c : out std_logic);
end killer_ckt;
```

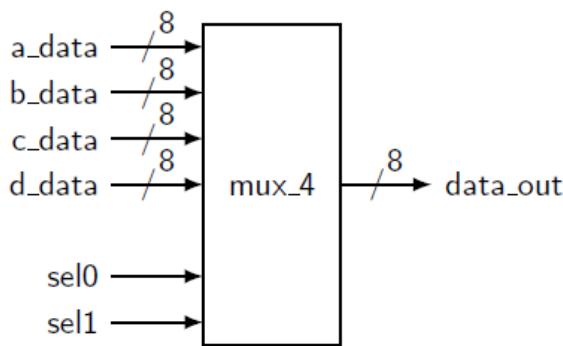
<sup>1</sup> Interface specification<sup>2</sup> حالت inout بعداً در همین کتاب بررسی می‌شود

در ارتباط با کد VHDL اخیر چند نکته‌ی زیر قابل تأمل است. اکثر این نکات مرتبط با خوانایی و قابل فهم بودن این کد هستند:

- نام هر پورت یکتا بوده و دارای یک حالت و نوع داده است. این مطلب یک امر ضروری است.
- کامپایلر VHDL به شما اجازه می‌دهد که چند نام پورت را در یک خط قرار دهید. نام پورتها با ویرگول از هم جدا می‌شوند. همیشه خوانایی کد را در نظر داشته باشید.
- معمولاً به جهت افزایش خوانایی سعی می‌شود نام پورتها در خطوط منظمی تنظیم شوند. این کار اجباری نیست بلکه به جهت افزایش خوانایی است. به خاطر داشته باشید که فضاهای خالی توسط کامپایلر نادیده گرفته می‌شوند.
- یک توضیح راجع به کاری که موجودیت انجام می‌دهد، گذاشته شده است.
- یک دیاگرام جعبه‌ی سیاه نیز از مدار نشان داده شده است. کشیدن جعبه‌ی سیاه به نوشتن کد VHDL کمک می‌کند.

با توجه به این که اصطلاح «گذرگاه» (bus) در مدارات دیجیتال معنای خاصی دارد برای این که در VHDL با مجموعه‌ای از سیگنالها که نام مشترکی بر آنها گذاشته شده، اشتباہ گرفته نشوند از کلمه‌ی گروه سیگنال (bundle) استفاده می‌کنیم.

چند مثال در شکل ۳-۳ نشان داده شده است. در این مثالها توجه کنید که حالت (مُد) ثابت مانده است اما نوع تغییر کرده است. نوع داده‌ی std\_logic\_vector با نوع std\_logic جایگزین شده است تا نشان داده شود هر نام سیگنال شامل بیش از یک سیگنال است. برای ارجاع و نام بردن از هر یک از اعضای یک گروه سیگنال راههایی وجود دارد که بعداً به آن خواهیم پرداخت.



Listing 3.3: Entity declaration with bundles.

```

-----
-- Unlike the other examples, this is actually an interface
-- for a MUX that selects one of four bus lines for the output.
-----
entity mux4 is
port ( a_data      : in      std_logic_vector(0 to 7);
       b_data      : in      std_logic_vector(0 to 7);
       c_data      : in      std_logic_vector(0 to 7);
       d_data      : in      std_logic_vector(0 to 7);
       sel0,sel1  : in      std_logic;
       data_out   : out     std_logic_vector(7 downto 0));
end mux4;

```

همان طور که از شکل ۳-۳ می توان دید دو روش ممکن برای توصیف سیگنالهای واقع در یک گروه وجود دارد. این دو روش در لیست آرگومانها بعد از اعلان نوع داده نشان داده شده‌اند. سیگنالهای واقع در یک گروه به دو طریق قابل مرتب شدن هستند که برای آنها از کلمات کلیدی `to` و `downto` استفاده می‌شود. اگر بخواهید که با ارزش‌ترین بیت (MSB) از گروه سیگنال متناظر با اولین بیت از سمت چپ باشد، از کلمه‌ی کلیدی `downto` استفاده کنید.

در جعبه‌ی سیاه نمایش داده شده در شکل ۳-۳ از علامت اسلش به همراه یک عدد استفاده شده است. علامت اسلش نشان‌دهنده‌ی وجود یک گروه سیگنال و عدد متناظر نیز معرف تعداد سیگنالهای موجود در آن گروه سیگنال است. در شکل ۳-۳ می‌توانستیم خطوط ورودی `sel0` و `sel1` را نیز در یک گروه سیگنال ادغام کنیم.

نوع داده‌ی `std_logic` و `std_logic_vector` در بسته‌ی IEEE 1164 از کتابخانه‌ی IEEE تعریف شده‌اند. این نوع علاوه بر دو مقدار دیجیتال متداول ۰ و ۱ شامل هفت مقدار دیگر (یعنی جمعاً ۹ مقدار مختلف) است که عبارتند از: U، L، Z، W، X، H، –.

برای استفاده از نوع داده‌ی `std_logic` باید در ابتدای کد خود از اعلان

library IEEE;

```
use IEEE.std_logic_1164.all;
```

استفاده کنید.

یک نوع ساده‌تر و معادل با نوع `std_logic`, نوع `bit` است که تنها شامل 0 و 1 است.

## ۲-۳ کتابخانه‌های استاندارد VHDL

به منظور بهره بردن از مهمترین ویژگی‌های قابل پیاده‌سازی VHDL باید دو بسته‌ی اصلی از کتابخانه‌ی IEEE را که در خطوط ۲ تا ۴ از شکل زیر نشان داده شده است، مشخص و استفاده کنید.

Listing 3.4: Typical inclusions of IEEE standard libraries.

```

1 -- library declaration
2 library IEEE;
3 use IEEE.std_logic_1164.all; -- basic IEEE library
4 use IEEE.numeric_std.all;   -- IEEE library for the unsigned type and
5                           -- various arithmetic operators
6
7 -- WARNING: in general try NOT to use the following libraries
8 --           because they are not IEEE standard libraries
9 -- use IEEE.std_logic_arith.all;
10 -- use IEEE.std_logic_unsigned.all;
11 -- use IEEE.std_logic_signed
12
13 -- entity
14 entity my_ent is
15     port ( A,B,C : in std_logic;
16            F      : out std_logic);
17 end my_ent;
18 -- architecture
19 architecture my_arch of my_ent is
20     signal v1,v2 : std_logic_vector (3 downto 0);
21     signal u1    : unsigned (3 downto 0);
22     signal i1    : integer;
23 begin
24     u1 <= "1101";
25     i1 <= 13;
26     v1 <= std_logic_vector(u1);                      -- = "1101"
27     v2 <= std_logic_vector(to_unsigned(i1, v2'length)); -- = "1101"
28
29 -- "4" could be used instead of "v2'length", but the "length"
30 -- attribute makes life easier if you want to change the size of v2
31
32     F <= NOT (A AND B AND C);
33 end my_arch;

```

با داول کردن این بسته‌ها شما به مجموعه‌ی بزرگی از قابلیت‌ها دسترسی خواهید داشت که برخی از آنها عبارتند از: چند نوع داده، عملگرهای سربارگذاری شده، توابع تبدیل مختلف، توابع ریاضی، و غیره.

برای مثال، با داول کردن بسته‌ی `numeric_std.all` شما امکان استفاده از نوع داده‌ی `unsigned` و تابع `to_unsigned` که در شکل ۳-۴ نشان داده شده‌اند را پیدا می‌کنید.

### ۳-۳ معماری

اعلان موجودیت در VHDL در واقع اتصال یا نمایش خارجی مدار را توصیف می‌کند. معماری، کاری را که مدار واقعاً انجام می‌دهد، توصیف می‌کند. به بیان دیگر، معماری در VHDL، پیاده‌سازی داخلی موجودیت متناظر با خود را توصیف می‌کند.

متناظر با یک موجودیت، چندین معماری معادل با هم می‌توانند وجود داشته باشند که همگی انها عملکرد داخلی موجودیت را توصیف می‌کنند. شیوه‌ی کدنویسی داخل بدنی معماری تاثیر زیادی روی نحوه‌ی سنتز شدن مدار (یعنی نحوه‌ی پیاده‌سازی مدار داخل یک افزارهای سیلیکون واقعی) دارد. این امر به برنامه‌نویس VHDL قدرت انعطافی در زمینه‌ی طراحی سیستمهایی با ویژگی‌های مثبت و منفی مانند اندازه‌ی فیزیکی مشخص (برحسب تعداد عناصر پایه‌ای دیجیتال) یا سرعت عملکرد قرار می‌دهد. یک معماری را می‌توان به کمک سه تکنیک مدل‌سازی یا ترکیبی از این مدل‌ها نوشت: مدل جریان داده<sup>۱</sup>، مدل رفتاری<sup>۲</sup>، مدل ساختاری<sup>۳</sup>، و مدل ترکیبی<sup>۴</sup>. این مدل‌ها در طول کتاب توضیح داده خواهند شد. شکل ۳-۵ یک پیش‌نمایش کلی از یک نمونه کد ساده اما کامل VHDL را نشان می‌دهد.

<sup>1</sup> Data-flow model

<sup>2</sup> Behavioral model

<sup>3</sup> Structural model

<sup>4</sup> Hybrid model

Listing 3.5: Example of a simple VHDL block.

```

1 ----- FILE: my_sys.vhd -----
2 -- library declaration
3 library ieee;
4 use ieee.std_logic_1164.all;
5
6 -- the ENTITY
7 entity circuit1 is
8     port (A,B,C : in std_logic;
9             F,G    : out std_logic);
10 end circuit1;
11
12 -- the ARCHITECTURE
13 architecture circuit1_arc of circuit1 is
14     signal sig_1 : std_logic;           -- signal definition
15 begin
16     process (a,b,c)
17         variable var_1 : integer;       -- variable definition
18     begin
19         F <= not (A and B and C);      -- signal assignment
20         sig_1 <= A;                  -- another signal assignment
21         var_1 := 34;                 -- variable assignment
22     end process;
23
24     G <= not (A and B);            -- concurrent assignment
25 end circuit1_arc;

```

### ۴-۳ تخصیص‌های سیگنال و متغیر

در VHDL چندین نوع شیء وجود دارد. برخی از متداول‌ترین این انواع شیء عبارتند از نوع شیء سیگنال (signal)، نوع شیء متغیر (variable)، و نوع شیء ثابت (constant). نوع سیگنال، نمایش نرم‌افزاری یک سیم است. نوع متغیر، همانند زبان‌های C و Java، برای ذخیره‌سازی اطلاعات محلی استفاده می‌شود. نوع ثابت مانند نوع شیء متغیر است اما با این تفاوت که مقدار آن قابل تغییر نیست. یک شیء سیگنال می‌تواند انواع مختلفی داشته باشد. برای مثال، قبلًا دیدیم که یک شیء سیگنال می‌تواند از نوع std\_logic و یا دیگر انواع مانند integer و نوع‌های سفارشی<sup>1</sup> باشد. همین مطالب در مورد شیء متغیر نیز برقرار است.

قبل از استفاده از یک سیگنال یا متغیر باید آن را اعلان کنید. سیگنال‌ها در بخش بالایی بدنی معماری و قبل از کلمه‌ی کلیدی begin اعلان می‌شوند. متغیرها باید داخل سازه‌ی «فرآیند» اعلان شوند؛ متغیرها محلی<sup>2</sup> (ونه سرتاسری) هستند. یک مثال در خط ۱۴ و ۱۷ از شکل ۳-۵ نشان داده شده است.

همان طور که در خط ۱۹ و ۲۰ از شکل ۳-۵ نشان داده شده است، زمانی که بخواهید مقدار جدیدی به یک شیء از نوع سیگنال تخصیص دهید، از عملگر " $=>$ " استفاده می‌کنید. معادلاً، همان طور که در

<sup>1</sup> Custom types

<sup>2</sup> Local

**خط ۲۱ از شکل ۳-۵** نشان داده شده است، زمانی که بخواهید مقدار جدیدی به یک شیء از نوع متغیر تخصیص دهید، از عملگر `=` استفاده می‌کنید.

درک تفاوت بین سیگنال و متغیر مهم است به ویژه زمانی که مقدار آنها تغییر می‌کند. بعد از انجام تخصیص متغیر، مقدار آن متغیر بلا فاصله تغییر می‌کند. اما در مورد سیگنال، پس از ارزیابی عبارت تخصیص، مقداری زمان می‌گذرد تا مقدار جدید سیگنال اعمال شود. این ویژگی عاقب و تاثیرات مهمی از حیث به روزرسانی سیگنالها و متغیرها دارد. این ویژگی بیان می‌دارد که پس از انجام تخصیص روی یک سیگنال هرگز نباید تصور کنید که به روزرسانی بلا فاصله انجام می‌شود. همچنین بیان می‌دارد که هرگاه در داخل یک فرآیند، اقدام به پیاده‌سازی یک شمارنده کردید و یا نیاز به ذخیره مقادیر داشتید، می‌توانید از متغیر استفاده کنید.

برای استفاده از متغیرها باید از یک سازه‌ی «فرآیند» استفاده کنیم که تا بحال شما با این سازه آشنایی پیدا نکرده‌اید. بعدها در این کتاب خواهیم دید که هرگاه نیاز به یک محیط اجرای غیرهمزمان<sup>۱</sup> (یعنی محیطی که در آن خطوط، مانند زبان C یا Java، به ترتیب و یکی پس از دیگری اجرا می‌شوند) داشتید، می‌توانیم از سازه‌ی فرآیند برای تحقق چنین محیطی استفاده کنیم. داخل یک فرآیند، تمام دستورات پشت سر هم و به ترتیب از بالا به پایین اجرا می‌شوند. تمام بدنی فرآیند به عنوان یک دستور تلقی می‌شود که به صورت موازی با بقیه‌ی کد (در مثال اخیر، یعنی به موازات دستور موجود در خط ۲۴) اجرا می‌شود. به خاطر داشته باشید که اجرای فرآیند کاملاً به موازات خط شماره‌ی ۲۴ انجام می‌شود.

### ۳-۵ خلاصه

- اعلان موجودیت، ورودی‌ها و خروجی‌های مدار شما را توصیف می‌کند. این مجموعه از سیگنالها را غالباً اتصال<sup>۲</sup> با مدار شما می‌نامند زیرا سیگنالهایی هستند که مدار خارج از موجودیت، از آنها جهت ارتباط و تعامل با مدار شما استفاده می‌کنند.
- سیگنالهای توصیف شده در بخش اعلان موجودیت دارای یک توصیف‌گر حالت (یا مُد<sup>۳</sup>) و یک توصیف‌گر نوع هستند. حالت می‌تواند شامل `in` و یا `out` (و یا همان طور که بعدها خواهد دید، `inout`) باشد. نوع نیز می‌تواند شامل `std_logic_vector` یا `std_logic` باشد.
- استفاده از کلمه‌ی گروه (bundle) بهتر از کلمه‌ی گذرگاه (bus) در هنگام کار و نام بردن از گروهی از سیگنالها که هدف مشترکی را دنبال می‌کنند، است. کلمه‌ی گذرگاه در دنیای دیجیتال معنا و کاربرد مهم دیگری دارد که ممکن است استفاده از آن در VHDL باعث سردرگمی و گمراهی شود.

<sup>1</sup> Non-concurrent execution environment

<sup>2</sup> Interface

<sup>3</sup> Mode specifier

- گروهی از سیگنالها که هدف مشترکی دارند، باید با استفاده از نوع `std_logic_vector` به صورت یک «گروه سیگنال» اعلان شوند. کارکردن یا گروه سیگنال راحت‌تر از سیگنال‌های تکی و منفرد است.
- وظیفه‌ی معماری، توصیف رفتار و کاری است که مدار شما انجام می‌دهد. در VHDL، یک رفتار مشخص را به چندین روش (یا مدل) می‌توان پیاده‌سازی کرد. این مدل‌ها عبارتند از: مدل جریان داده، مدل رفتاری، مدل ساختاری، و ترکیبی از هر یک از این سه مدل به نام مدل ترکیبی.

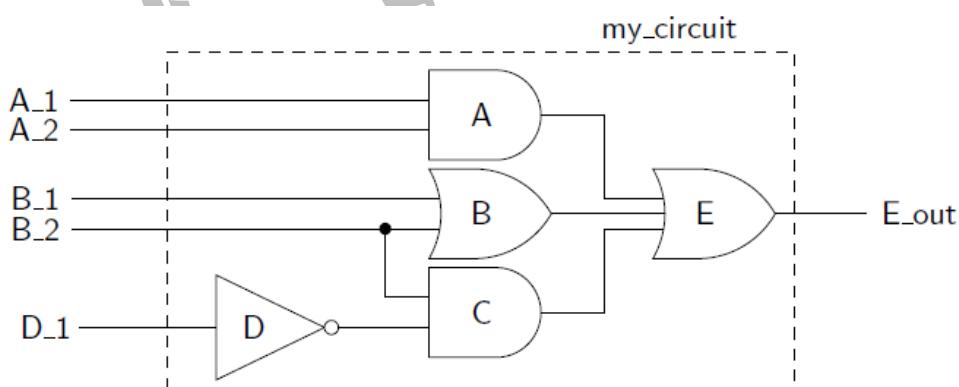
فصل چهارم

## الگوی برنامه نویسی VHDL

۴- دستورات همزمان

برنامه‌نویسی VHDL متفاوت از اکثر زبانهای برنامه‌نویسی متداول است از این جهت که قادر به اجرای دستورات به صورت موازی است؛ این امر به دلیل ذات موازی کاری سخت‌افزارهای دیجیتال است و زبان VHDL نیز برای توصیف یک سخت‌افزار دیجیتال طراحی شده است.

شکل ۴-۱ یک مثال ساده از یک مداری را نشان می‌دهد که به صورت موازی کار می‌کند. همان‌طور که می‌دانید، خروجی گیت‌ها تابعی از ورودی این گیت‌ها است. هر زمان که یکی از ورودی‌های گیت تغییر کند این امکان وجود دارد که پس از مدت زمان (تأخير) مشخصی، خروجی گیت نیز تغییر کند. این ویژگی در مورد هر یک از گیت‌های شکل ۴-۱ صادق است. هر زمان که ورودی‌ها تغییر کنند، وضعیت مدار ارزیابی مجدد شده و متعاقباً خروجی‌ها ممکن است تغییر کنند.



**Figure 4.1:** Some common circuit that is well known to execute parallel operations.

گرچه شکل ۴-۱ شامل تعداد محدودی گیت است اما ایده‌ی عملکرد همزمان عناصر موجود در مدار در تمام مدارات دیجیتال، کوچک یا بزرگ، برقرار و حاکم است.

الگوی برنامه‌نویسی VHDL مبتنی بر توصیف متغیر موازی کاری و همزمانی است. قلب برنامه‌نویسی VHDL، دستورات همزمان<sup>۱</sup> هستند. ظاهر این نوع دستورات مشابه با دستورات موجود در زبانهای برنامه‌نویسی الگوریتمی است اما عملاً بسیار متفاوت از آنها هستند زیرا عملیات‌های موازی را توصیف می‌کنند.

برنامه‌ی ۴-۱ کد مربوط به پیاده‌سازی مدار نشان داده شده در شکل ۴-۱ را نشان می‌دهد. این کد، چهار دستور تخصیص همزمان سیگنال را نشان می‌دهد. همان طور که قبلًاً بیان شد، سازه‌ی “=>” معرف عملگر تخصیص سیگنال است. گرچه نمی‌توانیم این چهار دستور را همزمان با هم بنویسیم اما می‌توانیم اینها را چهار دستوری در نظر بگیریم که همزمان با هم در حال اجرا شدن هستند. به خاطر بسپارید که این مفهوم همزمانی یک مفهوم کلیدی در VHDL است.

Listing 4.1: VHDL code for the circuit of Figure 4.1.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;

-- entity
entity my_circuit is
    port ( A_1,A_2,B_1,B_2,D_1 : in std_logic;
           E_out                  : out std_logic);
end my_circuit;

-- architecture
architecture my_circuit_arc of my_circuit is
    signal A_out, B_out, C_out : std_logic;
begin
    A_out <= A_1 and A_2;
    B_out <= B_1 or B_2;
    C_out <= (not D_1) and B_2;
    E_out <= A_out or B_out or C_out;
end my_circuit_arc;
```

با توجه به ذات موازی کاری در دستورات همزمان، قبول دارید که سه قطعه کد زیر دقیقاً معادل با کد نشان داده شده در برنامه‌ی ۴-۱ هستند. در حقیقت، ترتیب دستورات در این قطعه کدها تاثیری روی عملکرد نهایی کد ندارد. در حالت کلی، توصیف مدار به شیوه‌ی انجام شده در برنامه‌ی ۴-۱ بهترین شکل است زیرا به نوعی گویای سازمان‌طبعی دستورات است.

<sup>۱</sup> Concurrent statements

Listing 4.2: Equivalent VHDL code for the circuit of Figure 4.1.

```
C_out <= (not D_1) and B_2;
A_out <= A_1 and A_2;
B_out <= B_1 or B_2;
E_out <= A_out or B_out or C_out;
```

Listing 4.3: Equivalent VHDL code for the circuit of Figure 4.1.

```
A_out <= A_1 and A_2;
E_out <= A_out or B_out or C_out;
B_out <= B_1 or B_2;
C_out <= (not D_1) and B_2;
```

Listing 4.4: Equivalent VHDL code for the circuit of Figure 4.1.

```
B_out <= B_1 or B_2;
A_out <= A_1 and A_2;
E_out <= A_out or B_out or C_out;
C_out <= (not D_1) and B_2;
```

#### ۴-۲ عملگر تخصیص سیگنال ”=>

در VHDL برای انجام تخصیص سیگنال از عملگری به صورت ”=>“ استفاده می‌شود. بنابراین دستور  $G <= A \text{ AND } B;$  در برنامه‌ی ۴-۱ به این معنا است که مقدار سیگنال G با انجام عمل منطقی AND بین مقادیر دو سیگنال A و B به دست می‌آید.

چهار نوع دستور همزمان وجود دارد که در این فصل به آنها پرداخته می‌شود. یکی از آنها همین عملگر تخصیص سیگنال است. سه نوع دستور همزمان دیگر عبارتند از: دستور process، تخصیص شرطی سیگنال، و تخصیص انتخاب شده‌ی سیگنال.

#### ۴-۳ دستورات تخصیص همزمان سیگنال

شکل کلی یک دستور تخصیص همزمان سیگنال در برنامه‌ی ۴-۵ نشان داده شده است. در اینجا مقصد (هدف) سیگنالی است که مقادیر عبارت<sup>۱</sup> را دریافت می‌کند. یک «عبارت» می‌تواند شامل ثابت‌ها، سیگنال‌ها، یا مجموعه‌ای از عملگرها باشد.

Listing 4.5: Syntax for the concurrent signal assignment statement.

```
<target> <= <expression>;
```

<sup>1</sup> Expression

مثالهایی از عبارتهای مورد استفاده در VHDL در قالب مثالهایی که در ادامه می‌آیند، نشان داده شده است.

**مثال ۱.** یک کد VHDL جهت پیاده‌سازی یک گیت NAND سه ورودی بنویسید. سه سیگنال ورودی A، B، و C نام داشته و نام سیگنال خروجی نیز F است.

حل: همیشه سعی کنید یک دیاگرام از مداری که قصد طراحی آن را دارید، رسم کنید. برای رسک این دیاگرام بهتر است از رویکرد جعبه‌ی سیاه استفاده کنید؛ یعنی اگر چه در اینجا می‌توانید از نماد مداری معروف گیت NAND استفاده کنید اما از دیاگرام کلی‌تر (جعبه‌ی سیاه) بهتر است استفاده شود. جواب این سوال در برنامه‌ی ۴-۶ نشان داده شده است.

Listing 4.6: Solution of Example 1.

```

1 -- library declaration
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 -- entity
5 entity my_nand3 is
6   port ( A,B,C : in std_logic;
7         F      : out std_logic);
8 end my_nand3;
9 -- architecture
10 architecture exa_nand3 of my_nand3 is
11 begin
12   F <= NOT(A AND B AND C);
13 end exa_nand3;
```



این مثال شامل برخی ایده‌های نسبتاً جدید است:

- شما باید از فایلهای سرآیند و فایلهای کتابخانه‌ای در کد VHDL خود استفاده کنید تا کدتان به درستی کامپایل شود. خطوط کد مربوط به این فایلهای در قسمت بالایی کد نشان داده شده در برنامه‌ی ۴-۶ آورده شده‌اند. البته این خطوط بیشتر از حد مورد نیاز برای این مثال هستند اما در مثالهای بعدی مورد نیاز هستند. برای صرفه‌جویی در مکان، این خطوط در بعضی مثالها حذف می‌شوند (نشان داده نمی‌شوند).

- این مثال استفاده از برخی عملگرهای منطقی را نشان داده است. عملگرهای منطقی موجود در VHDL عبارتند از AND، OR، NOR، NAND، XOR و XNOR. عملگر NOT گرچه از نظر فنی یک عملگر نیست، اما جزو عملگرهای VHDL آورده شده است. نکته‌ی دیگر این که این عملگرها «باینری» هستند به این معنا که روی دو عملوند موجود در دو طرف خود اعمال

می‌شوند. از این جهت، عملگر NOT عملگری یکانی<sup>۱</sup> است و روی عملوند موجود در سمت راست خود اعمال می‌شود.

- در راه حل ارائه شده برای این مثال، موجودیت تنها دارای یک معماری است. در بیشتر طراحی‌های مبتنی بر VHDL این کار امری رایج و متداول است.

مثال ۱ نحوه‌ی استفاده از تخصیص سیگنال همزمان (CSA)<sup>۲</sup> را نشان می‌دهد (خط ۱۲ از برنامه‌ی ۴–۶). اما با توجه به این که تنها از یک دستور CSA استفاده شده است، مفهوم همزمانی به خوبی مشخص نشده است. ایده‌ی موجود پشت هر دستور همزمان در VHDL این است که هر زمان که یکی از ورودی‌ها تغییر کند، خروجی نیز تغییر کند. به بیان دیگر، هر زمان که یک سیگنال موجود در عبارت ورودی تغییر کند، خروجی F نیز ارزیابی مجدد شده و احتمالاً تغییر می‌کند. این مطلب، یک مفهوم کلیدی در درک صحیح VHDL است. مفهوم همزمانی به کمک مثال ۲ بیشتر و واضح‌تر نشان داده شده است.

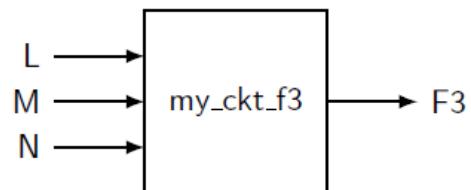
مثال ۲. یک کد VHDL برای پیاده‌سازی تابعی که با معادله‌ی منطقی زیر مشخص شده است، بنویسید.

$$F3 = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$$

حل: دیاگرام جعبه‌ی سیاه به همراه کد VHDL مربوطه در برنامه‌ی ۴–۷ نشان داده شده است.

Listing 4.7: Solution of Example 2.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
           F3      : out std_logic);
end my_ckt_f3;
-- architecture
architecture f3_2 of my_ckt_f3 is
begin
    F3<= ((NOT L) AND (NOT M) AND N) OR (L AND M);
end f3_2;
```



این مثال پیاده‌سازی تک خطی از معادله‌ی منطقی داده شده را نشان می‌دهد.

<sup>1</sup> Unary operator

<sup>2</sup> Concurrent Signal Assignment (CSA)

یک راه حل دیگر برای مثال ۲ در برنامه‌ی ۴-۸ نشان داده شده است. این مثال، یک مفهوم مهم در VHDL را نشان می‌دهد. راه حل نشان داده شده در برنامه‌ی ۴-۸ از برخی دستورات خاص جهت پیاده‌سازی مدار استفاده کرده است. این دستورات خاص به منظور تولید آن چه که نتایج میانی<sup>۱</sup> نامیده می‌شوند، استفاده شده‌اند. این رویکرد معادل با اعلان متغیرهای بیشتر در یک زبان برنامه‌نویسی الگوریتمی به منظور ذخیره‌ی نتایج میانی است. برای پوشش نیاز به نتایج میانی، باید چند سیگنال اضافه به نام سیگنالهای میانی<sup>۲</sup> اعلان شوند. در برنامه‌ی ۴-۸ دقت کنید که اعلان سیگنالهای میانی مشابه با اعلان پورت است که در بخش اعلان موجودیت آمده است با این تفاوت که در این جا نیازی به مشخص کردن حالت (in، out، inout) نیست.

سیگنالهای میانی باید در داخل بدنه‌ی معماری اعلان شوند زیرا این نوع سیگنالها هیچ راه ارتباطی به دنیای بیرون ندارند و بنابراین نباید در بخش اعلان موجودیت آورده شوند. این نوع سیگنالها داخل بدنه‌ی معماری و قبل از دستور begin اعلان می‌شوند.

Listing 4.8: Alternative solution of Example 2.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
           F3      : out std_logic);
end my_ckt_f3;
-- architecture
architecture f3_1 of my_ckt_f3 is
    signal A1, A2 : std_logic; -- intermediate signals
begin
    A1 <= ((NOT L) AND (NOT M) AND N);
    A2 <= L AND M;
    F3 <= A1 OR A2;
end f3_1;
```

برخلاف ظاهر کار که معماری‌های f3\_2 و f3\_1 مربوط به برنامه‌های ۴-۷ و ۴-۸ متفاوت به نظر می‌رسند، اینها از نظر عملکرد معادل با هم هستند زیرا تمام دستورات استفاده شده از نوع دستورات تخصیص همزمان سیگنال هستند. حتی اگر چه معماری f3\_1 شامل سه CSA است اما این دستورات از نظر عملکرد، معادل با دستورات CSA در معماری f3\_2 هستند زیرا هر یک از این سه دستور عمل<sup>۳</sup> موازی با هم اجرا می‌شوند.

<sup>1</sup> Intermediate results

<sup>2</sup> Intermediate signals

اگرچه برای این مثال، رویکرد استفاده از سیگنالهای میانی ضروری و اجباری نیست، اما استفاده از این رویکرد نکات خوبی برای ما به همراه دارد. اول این که استفاده از سیگنالهای میانی در اکثر مدلهاي VHDL رایج و مورد استفاده است. در برنامه‌ی ۴-۸ استفاده از سیگنالهای میانی به این دلیل اختیاری بود که مدار ساده‌ای را توصیف می‌کرد. با پیچیده‌تر شدن مدارات، موقعی پیش می‌آید که استفاده از سیگنالهای میانی ضروری و لازم است. دوم این که استفاده از سیگنالهای میانی در حکم ابزاری است که شما اغلب در مدلهاي VHDL خود به آن نیاز دارید. استفاده از سیگنالهای میانی به شما اجازه می‌دهد مدارات دیجیتال را ساده‌تر مدل کنید بدون این که سخت‌افزار نهایی پیچیده‌تر شود.

#### ۴-۴ تخصیص شرطی سیگنال به کمک when

در دستورات تخصیص سیگنال همزمان یک هدف و یک عبارت داشتیم اما در اینجا چند عبارت ولی باز هم یک هدف داریم. برای هر عبارت، یک شرط نیز وجود دارد. شرطها به ترتیب بررسی می‌شوند، اولین شرطی که صحیح بود، عبارت متاظر آن به هدف تخصیص می‌یابد. گرامر تخصیص شرطی سیگنال در برنامه‌ی ۴-۹ نشان داده شده است. در اینجا، هدف (target) نام یک سیگنال است. هر شرط می‌تواند مبتنی بر وضعیت برخی سیگنالهای دیگر موجود در مدار بیان شود. در هر دستور شرطی نهایتاً فقط یک تخصیص انجام می‌شود.

Listing 4.9: The syntax for the conditional signal assignment statement.

```
<target> <= <expression> when <condition> else
          <expression> when <condition> else
          <expression>;
```

احتمالاً راحت‌ترین راه برای فهم دستور تخصیص شرطی سیگنال، استفاده از یک مدار است. اجازه دهید مثال ۲ را مجدداً انجام دهیم اما این بار به جای تخصیص همزمان سیگنال، از تخصیص شرطی سیگنال استفاده کنیم.

مثال ۳. یک کد VHDL برای پیاده‌سازیتابع منطقی بیان شده در مثال ۲ بنویسید. تنها از دستورات تخصیص شرطی سیگنال در کد خود استفاده کنید.

حل: اعلان موجودیت در اینجا نسبت به مثال ۲، تغییری نمی‌کند؛ بنابراین، تنها نیازمند نوشتن یک معماری جدید هستیم. با در نظر گفتن معادله منطقی مربوط به مثال ۲، جواب مثال ۳ به صورت برنامه‌ی ۴-۱۰ خواهد بود.

Listing 4.10: Solution of Example 3.

```

architecture f3_3 of my_ckt_f3 is
begin
    F3 <= '1' when (L = '0' AND M = '0' AND N = '1') else
    '1' when (L = '1' AND M = '1') else
    '0';
end f3_3;

```

چند نکته‌ی جالب در مورد این راه حل وجود دارد:

- این راخ حل بهبود چندانی نسبت به راه حل مثال ۲ (تخصیص سیگنال همزمان) ایجاد نکرده است. حتی از نظر تعداد دستورات، کارایی کمتری نسبت به راه حل مثال ۲ دارد.
- در اینجا یک هدف و چندین عبارت و شرط وجود دارد. هر عبارت در اینجا یک رقم است که با دو علامت نقل قول تکی<sup>۱</sup> احاطه شده است. هر شرط پس از کلمه‌ی کلیدی when آمده است. تنها از یک عملگر تخصیص استفاده شده است.
- آخرین عبارت در دستور تخصیص سیگنال، شرط «گرفتن همه»<sup>۲</sup> نامیده می‌شود. اگر هیچیکی از شرایطی که در بالای این عبارت آخر وجود دارند، برقرار و صحیح نباشند، این عبارت آخر به هدف نسبت داده می‌شود.
- این راه حل از عملگرهای نسبت<sup>۳</sup> استفاده کرده است. در واقع، شش عملگر نسبت مختلف در VHDL وجود دارد؛ از این بین، دو عملگر متداول و رایج عبارتند از ”=” و ”=/“ که معنای آنها به ترتیب «مساوی با» و «نامساوی با» است. در بخش‌های بعدی، عملگرها با جزئیات بیشتری بحث خواهند شد.

یکی از استفاده‌های کلاسیک استفاده از دستور تخصیص شرطی سیگنال، پیاده‌سازی مالتی‌پلکسر (MUX) است. مثال بعدی، چنین کاری را نشان می‌دهد.

**مثال ۴.** یک کد VHDL برای پیاده‌سازی یک MUX 4:1 با استفاده از تنها یک دستور تخصیص شرطی سیگنال بنویسید. ورودی‌های MUX، ورودی‌های داده D0، D1، D2، و D3 به همراه یک گذرگاه کنترل دو ورودی SEL است. تنها خروجی نیز MX\_OUT نام دارد.

<sup>1</sup> Single quote

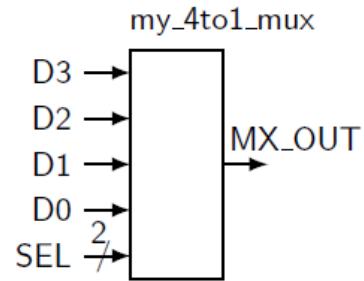
<sup>2</sup> Catch-all condition

<sup>3</sup> Relational operators

حل: در این مثال، لازم است از ابتدا شروع کنیم. لذا از دیاگرام جعبه‌ی سیاه و دستور اعلان موجودیت استفاده می‌کنیم. کد VHDL جواب در برنامه‌ی ۴-۱۱ نشان داده شده است.

Listing 4.11: Solution of Example 4.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_4t1_mux is
    port(D3,D2,D1,D0 : in std_logic;
          SEL : in std_logic_vector(1 downto 0);
          MX_OUT : out std_logic);
end my_4t1_mux;
-- architecture
architecture mux4t1 of my_4t1_mux is
begin
    MX_OUT <= D3 when (SEL = "11") else
                  D2 when (SEL = "10") else
                  D1 when (SEL = "01") else
                  D0 when (SEL = "00") else
                  '0';
end mux4t1;
```



نکات جالب مرتبط با این جواب به صورت زیر است:

- در مقایسه با جواب مبتنی بر دستورات CSA و در نظر گرفتن مقدار منطق استفاده شده، راه حل ارائه شده در این جا کارتر است. کد VHDL ارائه شده در اینجا به نظر خوب بوده و چشم‌نوای است (نکاتی که روی خوانایی کد اثر دارند).
- از عملگر نسبت “=” به همراه یک گروه سیگنال استفاده شده است. برای دسترسی به مقادیر گروه سیگنال SEL از علامت نقل قول مضاعف<sup>1</sup> حول مقدار مورد نظر استفاده شده است. به عبارت دیگر، از علامت نقل قول تکی برای توصیف مقادیر سیگنالهای تکی و از علامت نقل قول مضاعف برای توصیف مقادیر سیگنالهای چندتایی (یا همان «گروه سیگنال») استفاده می‌شود.
- به منظور رعایت کامل بودن کد، ما در اینجا تمام شرط‌های ممکن را برای سیگنال SEL به همراه دستور else (دستور گرفتن همه) در نظر گرفته‌ایم. می‌توانستیم خطی را که شامل '0' است به D0 تغییر داده و خط متناظر با مقدار "00" از شرط SEL را حذف کنیم. نتیجه‌ی این کار از نظر عملکرد معادل با راه حل نشان داده شده است اما در این صورت، ظاهر آن چندان چشمگیر و

<sup>1</sup> Pleasing to the eye

<sup>2</sup> Double quotes

تأثیرگذار نخواهد شد. اگر بخواهیم کلی صحبت کنیم، شما بهتر است تمام گزینه‌های ممکن را در کد خود آورده و به یک دستور گرفتن همه به منظور انجام تخصیص سیگنال اکتفا نکنید.

به خاطر داشته باشید که یک دستور تخصیص شرطی سیگنال نوعی دستور همزمان است؛ در اینجا، دستور تخصیص شرطی سیگنال زمانی اجرا می‌شود که تغییری روی سیگنالهای شرطی (یعنی سیگنالهای موجود در عبارات سمت راست عملگر تخصیص سیگنال) رخ دهد. این کار شبیه به دستور تخصیص همزمان سیگنال است که در آن، هرگاه تغییری روی یکی از سیگنالهای موجود در سمت راست عملگر تخصیص سیگنال رخ می‌داد، این دستور اجرا می‌شد. دستور تخصیص شرطی سیگنال، از نظر عملکرد، معادل با سازه‌ی if-else در زبانهای برنامه‌نویسی الگوریتمی است.

مفهوم کار با گروه‌های سیگنال در VHDL بسیار مهم است. در حالت کلی می‌توان گفت اگر می‌توانید به جای سیگنالهای تکی، با گروه سیگنال کار کنید حتماً همین کار را انجام دهید. در این حالت، اغلب نیاز به دسترسی به یکی از سیگنالهای موجود در گروه سیگنال دارید. برای این کار از گرامر خاص خود (برای مثال، SEL(1)) استفاده کنید. توجه کنید که کد نشان داده شده در برنامه‌ی ۱۲-۴ معادل با کد مربوط به برنامه‌ی ۱۱-۴ است اما به روشنی و وضوح آن نمی‌رسد. به شباهت‌ها و تفاوت‌های بین این دو برنامه دقت کنید.

Listing 4.12: Alternative solution to Example 4 accessing individual signals.

```
-- entity and architecture of 4:1 Multiplexer implemented using
-- conditional signal assignment. The conditions access the
-- individual signals of the SEL bundle in this model.
-----
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_4t1_mux is
    port (D3,D2,D1,D0 : in std_logic;
          SEL           : in std_logic_vector(1 downto 0);
          MX_OUT        : out std_logic);
end my_4t1_mux;
-- architecture
architecture mux4t1 of my_4t1_mux is
begin
    MX_OUT <= D3 when (SEL(1) = '1' and SEL(0) = '1') else
                  D2 when (SEL(1) = '1' and SEL(0) = '0') else
                  D1 when (SEL(1) = '0' and SEL(0) = '1') else
                  D0 when (SEL(1) = '0' and SEL(0) = '0') else
                  '0';
end mux4t1;
```

#### ۴-۵ تخصیص انتخاب شده سیگنال به کمک دستور **with select**

دستورات تخصیص انتخاب شده سیگنال، سومین نوع از تخصیص سیگنال است که اکنون آن را بررسی می‌کنیم. در اینجا هم مشابه با تخصیص شرطی سیگنال، تنها از یک عملگر تخصیص استفاده می‌شود. تفاوت دستورات تخصیص انتخاب شده سیگنال با دستورات تخصیص شرطی این است که تمام تخصیص‌ها مبتنی بر تنها یک عبارت است. گرامر دستور تخصیص انتخاب شده سیگنال در برنامه‌ی ۴-۱۳ نشان داده شده است.

**Listing 4.13:** Syntax for the selected signal assignment statement.

```
with <choose_expression> select
    target <= <expression> when <choices>,
    <expression> when <choices>;
```

مثال ۵. یک کد VHDL برای پیاده‌سازی تابع توصیف شده با معادله‌ی منطقی زیر بنویسید. در کد خود تنها از دستورات تخصیص انتخاب شده سیگنال استفاده کنید.

$$F3 = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$$

حل: در اینجا نسخه‌ی دیگری از مثال f3\_ckt\_my\_ckt\_f3 که اولین بار در مثال ۲ دیدیم، نشان داده می‌شود. برنامه‌ی ۴-۱۴ این جواب را نشان می‌دهد.

**Listing 4.14:** Solution of Example 5.

```
-- yet another solution to the previous example
architecture f3_4 of my_ckt_f3 is
begin
    with ((L = '0' and M = '0' and N = '1') or (L = '1' and M = '1')) select
        F3 <= '1' when '1',
        '0' when '0',
        '0' when others;
end f3_4;
```

یک نکته در مورد برنامه‌ی ۴-۱۴ استفاده از عبارت when others به عنوان آخرین بخش از دستور تخصیص انتخاب شده سیگنال است. در عمل، عبارت وسطی '0' when '0' را از کد می‌توان حذف کرد بدون این که معنای دستور تغییری کند. در حالت کلی، در کدنویسی VHDL، کار خوبی است که تمام موارد مورد انتظار را در دستور تخصیص انتخاب شده سیگنال در نظر گرفته و در انتهای نیز از عبارت when others استفاده کنیم.

**مثال ۶.** یک کد VHDL برای پیاده‌سازی یک MUX 4:1 با استفاده از یک دستور تخصیص انتخاب شده‌ی سیگنال بنویسید. ورودی‌های این مالتی‌پلکسر، داده‌های ورودی D0، D1، D2، D3 و همراه یک گذرگاه کنترل دو ورودی SEL بوده و تنها خروجی نیز MX\_OUT است.

حل: این مثال، تکرار مثال ۴ است با این تفاوت که به جای عملگر تخصیص شرطی سیگنال، باید از عملگر تخصیص انتخاب شده‌ی سیگنال استفاده شود. جواب این مثال در برنامه‌ی ۴-۱۵ نشان داده شده است. دیاگرام جعبه‌ی سیاه مربوط به این مثال عیناً همان دیاگرام قبل بوده و بنابراین، در این جا تکرار نشده است.

Listing 4.15: Solution of Example 6.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_4t1_mux is
    port (D3,D2,D1,D0 : in std_logic;
          SEL           : in std_logic_vector(1 downto 0);
          MX_OUT        : out std_logic);
end my_4t1_mux;
-- architecture
architecture mux4t1_2 of my_4t1_mux is
begin
    with SEL select
        MX_OUT <= D3 when "11",
                      D2 when "10",
                      D1 when "01",
                      D0 when "00",
                      '0' when others;
end mux4t1_2;
```

چند نکته‌ی قابل توجه در خصوص این مثال عبارت است از:

- کد VHDL این راه حل شباهت زیادی به کد VHDL مربوط به مثال ۵ دارد. ظاهر کلی هر دو کد یکی است. هر دو کد بسیار چشم‌نوازتر از کد مربوط به استفاده از دستورات تخصیص همزمان سیگنال هستند.
- مجدداً در اینجا هم از عبارت when others استفاده شده است. در این جا اگر هیچ یک از شرایط لیست شده در قسمت chooser\_expression (از گرامر کلی -م) برآورده نشده باشد، مقدار ثابت '0' به خروجی تخصیص می‌یابد.

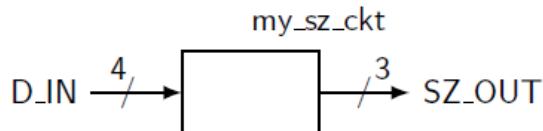
- در اینجا تمام شرایط مربوط به `chooser_expression` در بدنی دستور تخصیص انتخاب شده‌ی سیگنال در نظر گرفته شده است. البته این کار لازم و اجباری نیست. تنها کار لازم و اجباری در اینجا، استفاده از کلمه‌ی کلیدی `when others` در آخرین خط از دستور است.

**مثال ۷.** یک کد VHDL برای پیاده‌سازی مدار زیر بنویسید. این مدار شامل یک گروه سیگنال ورودی متشکل از چهار سیگنال بوده و گروه سیگنال خروجی نیز متشکل از سه سیگنال است. گروه سیگنال ورودی، `D_IN`، بیانگر یک عدد بازیگر چهار بیتی است. گروه سیگنال خروجی، `SZ_OUT`، بیانگر اندازه‌ی عدد چهار بیتی ورودی است. رابطه‌ی بین ورودی و خروجی در جدول زیر داده شده است. از یک دستور تخصیص انتخاب شده‌ی سیگنال در کد خود استفاده کنید.

range of D_IN	SZ_OUT
$0000 \rightarrow 0011$	100
$0100 \rightarrow 1001$	010
$1010 \rightarrow 1111$	001
unknown value	000

حل: این مدار، مثالی از یک دیکدر عام<sup>۱</sup> است. جواب این مثال در برنامه‌ی ۴-۱۶ نشان داده شده است.

<sup>۱</sup> Generic decoder



Listing 4.16: Solution of Example 7.

```

-- A decoder-type circuit using selected signal assignment --
library IEEE;
use IEEE.std_logic_1164.all;
entity my_sz_ckt is
    port ( D_IN : in std_logic_vector(3 downto 0);
           SZ_OUT : out std_logic_vector(2 downto 0));
end my_sz_ckt;
architecture spec_dec of my_sz_ckt is
begin
    with D_IN select
        SZ_OUT <= "100" when "0000"|"0001"|"0010"|"0011",
                           "010" when "0100"|"0101"|"0110"|"0111"|"1000"|"1001",
                           "001" when "1010"|"1011"|"1100"|"1101"|"1110"|"1111",
                           "000" when others;
end spec_dec;

```

تنها توضیح مربوط به برنامه‌ی ۴-۱۶ است که در دستور تخصیص انتخاب شده‌ی سیگنال، از علامت خط عمودی به عنوان کاراکتر انتخاب در بخش انتخابها استفاده شده است. این کار موجب افزایش خوانایی کد می‌شود.

مجدداً یادآوری می‌شود که دستور تخصیص انتخاب شده‌ی سیگنال یک شکل از دستور همزمان است.

دستور تخصیص انتخاب شده‌ی سیگنال هر بار که یکی از سیگنالهای موجود در **chooser\_expression** که در خط اول از این دستور لیست شده‌اند، تغییر کند، ارزیابی می‌شود. ارزیابی مجدد نیز زمانی اتفاق می‌افتد که یک سیگنال شرطی واقع در سمت راست عملگر تخصیص سیگنال تغییر کند.

آخرین نکته در مورد دستور تخصیص انتخاب شده‌ی سیگنال مشابه با آخرین نکته در مورد دستور تخصیص شرطی سیگنال است. شکل کلی دستور تخصیص انتخاب شده‌ی سیگنال مشابه با شکل کلی دستور **switch** در دیگر زبانهای برنامه‌نویسی الگوریتمی مانند C یا Java است.

**مثال ۸.** یک کد VHDL برای پیاده‌سازیتابع بیان شده با معادله‌ی منطقی زیر بنویسید.

$$F3 = \overline{L} \cdot \overline{M} \cdot N + L \cdot M$$

حل: این همان مساله‌ای است که قبلاً دیدیم. مشکل راه حل قبلی در ارتباط با این مثال این بود نیاز به پاده‌سازی کاربر قبل از پیاده‌سازی داشت. عبارت منطقی معادل برای تابع  $F_3$  به همراه جدول  نو آن در زیر نشان داده شده است. حل مثال ۸ در برنامه‌ی ۴–۱۷ نشان داده شده است.

$$F_3(L, M, N) = \sum(1, 6, 7)$$

L	M	N	$F_3$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Listing 4.17: Solution of Example 8.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_ckt_f3 is
    port ( L,M,N : in std_logic;
           F3      : out std_logic);
end my_ckt_f3;
-- architecture
architecture f3_8 of my_ckt_f3 is
    signal t_sig : std_logic_vector(2 downto 0); -- local bundle
begin
    t_sig <= (L & M & N); -- concatenation operator

    with (t_sig) select
        F3 <= '1' when "001" | "110" | "111",
                    '0' when others;
end f3_8;
```



#### ۴-۶ دستور فرآیند (process)

دستور فرآیند آخرین نوع از تخصیص سیگنال است که نگاهی به آن می‌اندازیم. قبل از این کار باید برخی مقدمات و مفاهیم تشریح شود. دستور فرآیند دستوری است که شامل تعداد مشخصی از دستورات بوده و هر زمان که دستور فرآیند بخواهد اجرا شود، این دستورات به صورت ترتیبی اجرا می‌شوند. به بیان دیگر هرگاه بخواهید تعدادی دستور به صورت ترتیبی (از بالا به پایین) اجرا شوند، می‌توانید از دستور

فرآیند استفاده کنید. فراموش نکنید که دستور فرآیند خود یک دستور همزمان بوده و به موازات بقیه دستورات همزمان موجود در بدنهٔ معماری اجرا می‌شود.

#### ۷-۴ جمع‌بندی

- زوج موجودیت/معماری توصیف اتصال و توصیف رفتار یک مدار دیجیتال هستند.
- مهمترین ملاحظات طراحی در مدل‌سازی VHDL از این واقعیت که مدارات دیجیتال موازی کار هستند، پشتیبانی می‌کنند. به بیان دیگر، واحدهای مختلف طراحی در یک طراحی دیجیتال، اطلاعات را مستقل از هم پردازش و ذخیره‌سازی می‌کنند. این مطلب تفاوت اصلی بین VHDL و زبانهای برنامه‌نویسی سطح بالای کامپیوتری هستند.
- مهمترین انواع تخصیص سیگنال در VHDL عبارتند از: تخصیص همزمان سیگنال، تخصیص شرطی سیگنال، تخصیص انتخاب‌شده سیگنال، و دستورات فرآیند. هر دستور همزمان ف همزمان و موازی با بقیه دستورات همزمان اجرا می‌شود.
- دستور فرآیند یک دستور همزمان و شامل تعدادی دستور است که به صورت ترتیبی اجرا می‌شوند. برنامه‌نویس از این دستور زمانی استفاده می‌کند که بخواهد دستوراتی به ترتیب و پشت سر هم اجرا شوند.
- بدنهٔ معماری می‌تواند شامل هر تعداد از این انواع دستورات همزمان باشد.
- سیگنالهایی که به عنوان خروجی در قسمت اعلان موجودیت، اعلان شده‌اند، نمی‌توانند در سمت راست یک عملگر تخصیص سیگنال ظاهر شوند. برای جلوگیری از وقوع این حالت، از اعلان سیگنالهای میانی استفاده کنید. سیگنالهای میانی مشابه با سیگنالهای اعلان شده در موجودیت هستند اما با این تفاوت که برای آنها حالت (مُد) مشخص نمی‌شود. سیگنالهای میانی در داخل بدنهٔ معماری و قبل از دستور begin اعلان می‌شوند.
- در حالت کلی برای مدل‌سازی یک مدار دیجیتال، چندین رویکرد وجود دارد. به بیان دیگر، از انواع مختلفی از دستورات همزمان می‌توان برای توصیف یک مدار مشخص استفاده کرد. طراح باید به دنبال واضح‌سازی در مدل‌سازی دیجیتال بوده و به ابزار سنتز VHDL اجازه‌ی مرتب کردن جزئیات را بدهد.

## فصل پنجم

### مدلهای استاندارد در معماری‌های VHDL

در نوشن<sup>۱</sup> معماری‌های VHDL سه رویکرد مختلف وجود دارد که عبارتند از: شیوهی جریان داده<sup>۲</sup>، شیوهی ساختاری<sup>۳</sup>، و شیوهی رفتاری<sup>۴</sup>. رویکرد استاندارد برای یادگیری VHDL شامل معرفی هر یک از این شیوه‌های معماری به صورت جداگانه و سپس طراحی تعدادی مدار مبتنی بر هر شیوه است. گرچه این شیوه خوب است اما گاهی اوقات موجب گمراهی می‌شود زیرا در مدارات پیچیده‌تر VHDL، در حالت کلی، از ترکیبی از این شیوه‌ها استفاده می‌شود. در بحث‌های بعدی که راجع به این شیوه‌ها خواهد آمد، به این نکته توجه داشته باشید. البته ما قسمت عمدهی تلاش و تمرکز خود را روی معماری‌های جریان داده و رفتاری می‌گذاریم. مدل ساختاری در واقع روشی برای ترکیب مجموعه‌های موجود از مدل‌های VHDL است. به بیان دیگر، مدل سازی ساختاری از اتصال جعبه‌های سیاه پشتیبانی می‌کند اما قادر به توصیف توابع منطقی مورد استفاده برای مدل سازی عملکرد مدار نمی‌باشد؛ به همین دلیل، اکثرآ نمی‌توان آن را یک روش طراحی نامید بلکه بیشتر روشی جهت اتصال مأذول‌های قبل<sup>۵</sup> طراحی شده است.

تا به این لحظه تمام مداراتی که دیدیم به شیوهی جریان داده پیاده‌سازی شدند. اکنون زمان پرداختن به توصیف معماری‌ها به شیوهی رفتاری رسیده است. این شیوه حول یک دستور همزمان دیگر به نام دستور فرآیند<sup>۶</sup> بنا شده است.

#### ۱-۱ معماری به شیوهی جریان داده

در رویکرد جریان داده، برای توصیف مدارات از روابط ورودی-خروجی حاکم بر کامپونت‌های درون-ساخته‌ی مختلف موجود در زبان VHDL بهره گرفته می‌شود. کامپونت‌های درون-ساخته‌ی VHDL عبارتند از عملگرهایی مانند AND، OR، XOR و غیره. سه شکل دستورات همزمان که تا

<sup>1</sup> Data-flow style

<sup>2</sup> Structural style

<sup>3</sup> Behavioral style

بحال دیدیم (تخصیص همزمان سیگنال، تخصیص شرطی سیگنال، و تخصیص انتخاب شدهی سیگنال) همگی متعلق به معماری‌های مربوط به شیوهی جریان داده بودند. به عبارت دیگر، اگر شما در مدل‌های VHDL خود منحصرًا از دستورات همزمان، تخصیص شرطی و انتخاب شدهی سیگنال استفاده کنید، در واقع از یک مدل جریان داده استفاده کرده‌اید. مدل جریان داده برای مدارات کوچک و نسبتاً ساده مناسب و خوب است. اما با پیچیده‌تر شدن مدار اغلب بهتر است سراغ مدل‌های شیوهی رفتاری برویم.

## ۲-۵ معماری شیوهی رفتاری

معماری شیوهی رفتاری، برخلاف معماری شیوهی جریان داده، هیچ اطلاعاتی راجع به نحوی پیاده‌سازی سخت‌افزاری طرح (در زمان سنتز) نمی‌دهد. در اینجا رفتار و واکنش کلی طرح (یعنی رابطه‌ی کلی حاکم بین خروجی و ورودی) تعیین شده و جزئیات کار (پیاده‌سازی سخت‌افزاری) به ابزار سنتز سپرده می‌شود. به همین دلیل، مدل سازی رفتاری یک سطح بالاتر از مدل سازی جریان داده در حوزه‌ی تجرید مدار<sup>۱</sup> است. به بیان دیگر، مدل سازی شیوهی رفتاری را می‌توان معادل با رویکرد «جمعیه سیاه» در طراحی مدارات دانست.

قلب معماری شیوهی رفتاری، دستور فرآیند است. این دستور، چهارمین نوع از دستورات همزمان است و بعدها با آن کار خواهیم کرد. همان طور که خواهیم دید، این دستور از جنبه‌های متعددی متفاوت از سه دستور همزمان دیگر است.

## ۳-۵ دستور فرآیند (process)

دستور فرآیند خود یک دستور همزمان است که علائم مشخصه‌ی آن شامل یک برجسب، لیست حساسیت، ناحیه‌ی اعلان، و ناحیه‌ی begin-end دستوراتی که به صورت ترتیبی اجرا می‌شوند، است. از دستور process در برنامه‌ی ۵-۱ نشان داده شده است.

Listing 5.1: Syntax for the process statement.

```
-- this is my first process
my_label: process(sensitivity_list) is
  <item_declarations>
begin
  <sequential_statements>
end process my_label;
```

<sup>۱</sup> Circuit abstraction

مهمترین نکته در مورد دستور process این است که بدنی آن شامل دستورات ترتیبی است. مهمترین نفاوت بین دستورات تخصیص همزمان سیگنال و دستورات process همین دستورات ترتیبی است. قبل از تشریح تفاوتها، اجازه دهید شباهتها را معرفی و بررسی کنیم. برچسب فرآیند، مانند آن چه در برنامه‌ی ۱-۵ نشان داده شده است، اختیاری است اما بهتر همواره به جهت خودتوضیحی کدتان آن را استفاده کنید.

برنامه‌های ۲-۵ و ۳-۵ یک معماری شیوه‌ی جریان داده و یک معماری شیوه‌ی رفتاری را به ترتیب و برای یک موجودیت XOR مشترک نشان می‌دهند. مهمترین تفاوت بین این دو معماری حضور دستور process در کد است.

اجازه دهید به یاد بیاوریم که دستور تخصیص همزمان سیگنال در توصیف جریان داده به این صورت کار می‌کند که هر زمان تغییری در یکی از سیگنالهای واقع در سمت راست عملگر تخصیص سیگنال رخ دهد، سیگنال واقع در سمت چپ این عملگر باز-ارزیابی<sup>۱</sup> (یا اجرای مجدد) می‌شود. در مورد توصیف معماری رفتاری، هر زمان که تغییری در یکی از سیگنالهای موجود در لیست حساسیت رخ دهد، تمام دستورات (ترتیبی) موجود در بدنی دستور process باز-ارزیابی می‌شوند. (بنابراین، در اینجا) ارزیابی (و اجرای) دستور process توسط سیگنالهای موجود در لیست حساسیت کنترل می‌شود. دو رویکرد توضیح داده شده اساساً و عملاً یکسان بوده فقط این که گرامر آنها از اساس متفاوت است.

Listing 5.2: Data-flow architecture.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_xor is
port ( A,B : in std_logic;
       F : out std_logic);
end my_xor;
-- architecture
architecture dataflow of my_xor is
begin
    F <= A XOR B;
end dataflow;
--
```

Listing 5.3: Behavioral architecture.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_xor is
port ( A,B : in std_logic;
       F : out std_logic);
end my_xor;
-- architecture
architecture behav of my_xor is
begin
    xor_proc: process(A,B) is
    begin
        F <= A XOR B;
    end process xor_proc;
end behav;
```

<sup>1</sup> Re-evaluate

این جا همان جایی که است عجیب می‌شود. اگر چه هر دو معماری نشان داده شده در برنامه‌های ۵-۲ و ۳-۵ دارای یک دستور مشترک تخصیص سیگنال ( $F \leq A \text{ XOR } B$ ) هستند، اما اجرای این دستور در معماری شیوه‌ی رفتاری منوط به تغییر یکی از سیگنالهای موجود در «لیست حساسیت» (نه لزوماً یکی از سیگنالهای موجود در سمت راست این عبارت تخصیص -م) است. در مقابل، اجرای این دستور در مدل جریان داده، منوط به تغییر یکی از سیگنالهای A یا B است. این تفاوت، یک تفاوت عملکردی است نه یک تفاوت صوری.

دستور process خود یک دستور همزمان است؛ بنابراین هر گاه دو دستور در بدنی معماری از دو دستور process استفاده کنید، اجرای این دو همزمان خواهد بود. در برنامه‌ی ۴-۵ شما یک نمونه دستور process کامل را می‌توانید ببینید. دقت کنید که تمام متغیرهای تعریف شده در بدنی فرآیند، تنها در داخل بدنی فرآیند قابل مشاهده هستند. همچنین، توجه کنید که دستور واقع در خط ۲۴ در داخل بدنی معماری اما خارج از بدنی فرآیند قرار گرفته است؛ بنابراین، اجرای این دستور به موازات و همزمان با دستور process صورت می‌گیرد.

Listing 5.4: Use of the process statement.

```

1 -- library declaration
2 library IEEE;
3 use IEEE.std_logic_1164.all;
4 -- entity
5 entity my_system is
6 port ( A,B,C : in std_logic;
7        F,Q : out std_logic);
8 end my_system;
9 -- architecture
10 architecture behav of my_system is
11   signal A1 : std_logic;
12 begin
13   some_proc: process(A,B,C) is
14     variable x,y : integer;
15   begin
16     x:=74;
17     y:=67;
18     A1 <= A and B and C;
19     if x>y then
20       F <= A1 or B;
21     end if;
22   end process some_proc;
23   -- we are outside the process body
24   Q <= not A;
25 end behav;
```

## ۴-۵ دستورات ترتیبی

فرآیند ابزاری است که امکان اجرای برخی دستورات ترتیبی را برای ما فراهم می‌کند. سه نوع دستور ترتیبی وجود دارد. اولین نوع، دستور تخصیص سیگنال، "`=>`"، است که آن را قبلاً در مدل جریان داده دیده بودیم. دو نوع دستور دیگر عبارتند از دستور `if` و دستور `case`. خوبشخтанه این دو ساختار را قبلاً در زبانهای برنامه‌نویسی الگوریتمی دیده‌اید. در اینجا هم، این ساختارها بسیار شبیه به ساختارهای مربوط به زبانهای برنامه‌نویسی الگوریتمی است.

#### ۴-۱ دستور تخصیص سیگنال

شیوه‌ی ترتیبی یک دستور از نظر شکل ظاهری شبیه به دستور تخصیص همزمان سیگنال است. یک راه برای تمیز دادن این دو این است که اگر یک دستور تخصیص سیگنال در داخل یک فرآیند ظاهر شود از نوع ترتیبی و گرنه از نوع همزمان است. برای مثال، دستور تخصیص سیگنال "F <= A XOR B" در معماری شیوه‌ی جریان داده‌ی برنامه‌ی ۲-۵ یک دستور تخصیص همزمان سیگنال و همین دستور در معماری شیوه‌ی رفتاری مربوط به برنامه‌ی ۳-۵، یک دستور تخصیص ترتیبی سیگنال است.

#### ۴-۲ دستور `if`

از دستور `if` برای ایجاد شاخه در روند اجرای دستورات ترتیبی استفاده می‌شود. بسته به شرایط لیست شده در بدنه‌ی دستور `if`، یا دستورات متناظر با یکی از شاخه‌ها اجرا می‌شود یا این که دستورات هیچ شاخه‌ای اجرا نمی‌شود. شکل کلی دستور `if` در برنامه‌ی ۵-۵ نشان داده شده است.

Listing 5.5: Syntax of the `if` statement.

```
if (condition) then
  <statements>
elsif (condition) then
  <statements>
else
  <statements>
end if;
```

شکل ظاهری و نیز نحوه‌ی عملکرد دستور `if` در VHDL مشابه دستورات `if` موجود در اکثر زبانهای برنامه‌نویسی الگوریتمی است (البته گرامر آن اندکی متفاوت است). دستور `if` معادل ترتیبی دستور همزمان تخصیص شرطی سیگنال است. این دو دستور اساساً یک کار انجام می‌دهند اما دستور `if` ترتیبی بوده و فقط داخل بدنه‌ی یک فرآیند قابل استفاده است اما دستور تخصیص شرطی سیگنال نوعی دستور همزمان تخصیص سیگنال است.

چند نکته در مورد گرامر دستور `if` عبارتند از:

- پرانتزهای گذاشته شده در دو طرف عبارتهای شرطی اختیاری هستند. برای افزایش خوانایی سعی کنید همیشه از این پرانتزها استفاده کنید.
- هر دستور if یک کلمه‌ی کلیدی else then به همراه خود دارد. آخرین عبارت else قادر کلمه‌ی کلیدی then متناظر با خود است.
- همان طور که در برنامه‌ی ۵-۵ نیز نوشته شده است، عبارت else (آخر) یک دستور «گرفتن همه»<sup>۱</sup> است. اگر هیچ یک از شرطهایی که قبل از این عبارت نوشته شده‌اند، برقرار نباشند آن گاه دنباله‌ی دستوراتی که متناظر با عبارت else (آخر) هستند، اجرا می‌شوند. شکلی از دستور if که در برنامه‌ی ۵-۵ نشان داده شده است، تضمین می‌کند که حداقل یکی از دنباله‌های دستورات اجرا خواهند شد.
- عبارت else (آخر) اختیاری است. البته اگر این عبارت را استفاده نکنید این امکان وجود دارد که هیچ یک از دنباله‌های دستورات در دستور if ارزیابی و اجرا نشوند. این امر تبعات جدی در پی دارد که بعداً بحث و بررسی خواهد شد.

حال در ادامه مثالهایی بررسی می‌شوند تا نحوه استفاده از دستور if بهتر درک شود.

**مثال ۹.** یک کد VHDL بنویسید که با استفاده از یک دستور if تابع منطقی زیر را پیاده‌سازی کند.

$$F_{OUT}(A, B, C) = A\overline{B}\overline{C} + BC$$

حل: اگر چه در صورت سوال صراحتاً اشاره نشده اما در اینجا باید از یک معماری رفتاری استفاده کنیم زیرا در سوال خواسته شده است که از دستور if استفاده شود. کد VHDL مربوط به جواب این سوال در برنامه‌ی ۵-۶ نشان داده شده است.

<sup>۱</sup> Catch-all statement

Listing 5.6: Solution to Example 9.

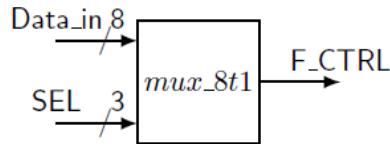
```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_ex is
port (A,B,C : in std_logic;
      F_OUT : out std_logic);
end my_ex;
-- architecture
architecture silly_example of my_ex is
begin
  proc1: process(A,B,C) is
  begin
    if (A = '1' and B = '0' and C = '0') then
      F_OUT <= '1';
    elsif (B = '1' and C = '1') then
      F_OUT <= '1';
    else
      F_OUT <= '0';
    end if;
  end process proc1;
end silly_example;
```

جواب اخیر احتمالاً بهترین راه برای پیاده‌سازی یک تابع منطقی نباشد اما در هر صورت کاربرد دستور if را نشان می‌دهد. یک معماری جواب دیگر برای این سوال در برنامه‌ی ۷-۵ نشان داده شده است.

Listing 5.7: Alternative solution to Example 9.

```
-- architecture
architecture bad_example of my_ex_7 is
begin
  proc1: process(A,B,C)
  begin
    if (A='1' and B='0' and C='0') or (B='1' and C='1') then
      F_OUT <= '1';
    else
      F_OUT <= '0';
    end if;
  end process proc1;
end bad_example;
```

**مثال ۱۰.** یک کد VHDL برای پیاده‌سازی 8:1 MUX زیر با استفاده از یک دستور if بنویسید.



حل: جواب در برنامه‌ی ۵-۸ نشان داده شده است.

Listing 5.8: Solution to Example 10.

```

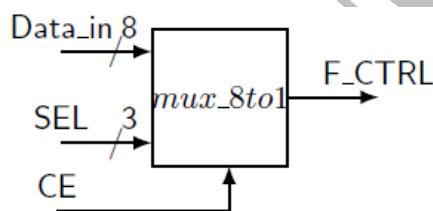
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity mux_8t1 is
    port ( Data_in : in std_logic_vector (7 downto 0);
           SEL : in std_logic_vector (2 downto 0);
           F_CTRL : out std_logic);
end mux_8t1;
-- architecture
architecture mux_8t1_arc of mux_8t1 is
begin
    my_mux: process (Data_in,SEL)
    begin
        if      (SEL = "111") then F_CTRL <= Data_in(7);
        elsif (SEL = "110") then F_CTRL <= Data_in(6);
        elsif (SEL = "101") then F_CTRL <= Data_in(5);
        elsif (SEL = "100") then F_CTRL <= Data_in(4);
        elsif (SEL = "011") then F_CTRL <= Data_in(3);
        elsif (SEL = "010") then F_CTRL <= Data_in(2);
        elsif (SEL = "001") then F_CTRL <= Data_in(1);
        elsif (SEL = "000") then F_CTRL <= Data_in(0);
        else F_CTRL <= '0';
        end if;
    end process my_mux;
end mux_8t1_arc;
  
```

در برنامه‌ی ۵-۸ از برخی گرامرها جدید استفاده شده است. در موجودیت از گروههای سیگنال استفاده شده است. در معماری برای دسترسی به عناصر این گروههای سیگنال از عملگر اندیس گذرگاه<sup>۱</sup> استفاده شده است. برای این کار از یک عدد داخل یک جفت پرانتز (برای مثال، (Data\_in(7)، استفاده شده است. عملگرهای اندیس گذرگاه به طور وسیعی در VHDL استفاده می‌شوند.

<sup>۱</sup> Bus index operator

یک نکته‌ی دیگر که در برنامه‌ی ۵-۸ باید به آن توجه کرد این است که تمام ترکیب‌های ممکن از متغیر انتخاب در کد در نظر گرفته شده است. می‌توان آخرین دستور `elsif` را در کد حذف کرده و دستور تخصیص سیگنال مربوط به آن را در عبارت `else` قرار داد؛ اما این کار، کار درستی نبوده و باید از انجام آن اجتناب کرد زیرا خوانایی کد را کاهش می‌دهد.

**مثال ۱۱.** یک کد VHDL برای پیاده‌سازی MUX 8:1 زیر بنویسید. از تعداد کافی از دستورات `if` که لازم می‌دانید، استفاده کنید. در دیاگرام جعبه‌سیاه نشان داده شده، ورودی `CE` یک فعال‌ساز<sup>۱</sup> چیپ<sup>۱</sup> (تراسه) است. هرگاه  $CE = 1'$  باشد، خروجی مانند مالتی‌پلکس‌ر مربوط به مثال ۱۰ کار خواهد کرد. هرگاه  $CE = 0'$  باشد، خروجی مالتی‌پلکس‌ر برابر  $0'$  خواهد شد.



حل: جواب این مثال تا حد بسیار زیادی شبیه به جواب مثال ۱۰ است. توجه کنید که در این جواب، دستور `if` را می‌توان به صورت تودرتو نوشت. برنامه‌ی ۵-۹ جواب را نشان می‌دهد.

<sup>۱</sup> Chip enable

Listing 5.9: Solution to Example 11.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity mux_8to1_ce is
    port ( Data_in : in std_logic_vector (7 downto 0);
           SEL : in std_logic_vector (2 downto 0);
           CE : in std_logic;
           F_CTRL : out std_logic);
end mux_8to1_ce;
-- architecture
architecture mux_8to1_ce_arch of mux_8to1_ce is
begin
    my_mux: process (Data_in, SEL, CE)
    begin
        if (CE = '0') then
            F_CTRL <= '0';
        else
            if (SEL = "111") then F_CTRL <= Data_in(7);
            elsif (SEL = "110") then F_CTRL <= Data_in(6);
            elsif (SEL = "101") then F_CTRL <= Data_in(5);
            elsif (SEL = "100") then F_CTRL <= Data_in(4);
            elsif (SEL = "011") then F_CTRL <= Data_in(3);
            elsif (SEL = "010") then F_CTRL <= Data_in(2);
            elsif (SEL = "001") then F_CTRL <= Data_in(1);
            elsif (SEL = "000") then F_CTRL <= Data_in(0);
            else F_CTRL <= '0';
            end if;
        end if;
    end process my_mux;
end mux_8to1_ce_arch;
```

### ۳-۴-۵ دستور case

دستور **case** تا حدی شبیه به دستور **if** است از این جهت که در آن، در صورت برقراری و درست بودن یک عبارت شرط، دنباله‌ای از دستورات اجرا می‌شوند. تفاوت این دستور با دستور **if** در این است که در اینجا **انتخاب بستگی** به مقدار تنها یک عبارت کنترل دارد. در هر اجرای دستور **case**، تنها یک مجموعه از دستورات ترتیبی اجرا می‌شوند. گرامر این دستور در برنامه‌ی ۵-۱۰ نشان داده شده است.

Listing 5.10: Syntax for the *case* statement.

```
case (expression) is
    when choices =>
        <sequential statements>
    when choices =>
        <sequential statements>
    when others =>
        <sequential statements>
end case;
```

دستور *case* معادل ترتیبی دستور همزمان تخصیص انتخاب شده سیگنال است. در این دستور، خط *when others* اختیاری است اما توصیه می شود حتماً از آن استفاده کنید.

مثال ۱۲. یک کد VHDL برای پیاده سازیتابع زیر با استفاده از دستور *case* بنویسید.

$$F_{OUT}(A, B, C) = A\bar{B}\bar{C} + BC$$

حل: در اولین قدم باید تابع منطقی داده شده را بر حسب مجموع مینترم ها توصیف کنیم. بنابراین می توانیم بنویسیم:

$$F_{OUT}(A, B, C) = A\bar{B}\bar{C} + BC$$

$$F_{OUT}(A, B, C) = A\bar{B}\bar{C} + BC(A + \bar{A})$$

$$F_{OUT}(A, B, C) = A\bar{B}\bar{C} + ABC + \bar{A}BC$$

حال، جواب در برنامه‌ی ۱۱-۵ نشان داده شده است. یک ویژگی جالب این که، گروه‌بندی سه سیگنال ورودی است که باعث شده به راحتی بتوان از آن در دستور *case* استفاده کرد. برای این کار نیاز به اعلان سیگنال میانه‌ای داریم که نام مناسبی مانند ABC برای آن انتخاب شده است. خاطر نشان می شود که شاید این جواب (استفاده از دستور *case*) موثرترین راه برای پیاده سازی یک تابع منطقی نباشد اما قدرت استفاده از دستور *case* را نشان می دهد و نیز نشان می دهد که باید در هر شرایطی، خلاق بوده و بتوان از منابع موجود استفاده کرد.

Listing 5.11: Solution to Example 12.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_example is
port (A,B,C : in std_logic;
      F_OUT : out std_logic);
end my_example;
-- architecture
architecture my_soln_exam of my_example is
    signal ABC: std_logic_vector(2 downto 0);
begin
    ABC <= A & B & C; -- group signals for case statement
    my_proc: process (ABC)
    begin
        case (ABC) is
            when "100" => F_OUT <= '1';
            when "011" => F_OUT <= '1';
            when "111" => F_OUT <= '1';
            when others => F_OUT <= '0';
        end case;
    end process my_proc;
end my_soln_exam;
```

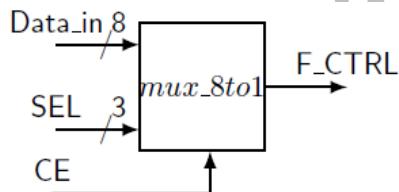
یک رویکرد مشابه دیگر برای جواب به سوال مثال ۱۲، استفاده از ویرگی حالت بی‌اهمیت (don't care) که به صورت درون-ساخته در VHDL وجود دارد، است. یک معماری متفاوت دیگر برای جواب به مثال ۱۲ در برنامه‌ی ۱۲-۵ نشان داده شده است. یک اشکال استفاده از ویرگی حالت بی‌اهمیت این است که برخی شبیه‌سازها و ابزارهای سنتر نمی‌توانند به خوبی آن را مدیریت کنند. (لذا به شخصه من از این روش دوری می‌کنم)

Listing 5.12: Alternative solution to Example 12.

```
architecture my_soln_exam2 of my_example is
    signal ABC: std_logic_vector(2 downto 0);
begin
    ABC <= A & B & C; -- group signals for case statement
    my_proc: process (ABC)
    begin
        case (ABC) is
            when "100" => F_OUT <= '1';
            when "-11" => F_OUT <= '1';
            when others => F_OUT <= '0';
        end case;
    end process my_proc;
end my_soln_exam2;
```

یکی از نکات مهمی که در هر کد VHDL باید به آن توجه داشت، خوانایی کد است. در مساله‌ی بعدی، مثال ۱۱ را مجدداً انجام می‌دهیم اما این بار به جای دستور if، از دستور case استفاده می‌کنیم.

**مثال ۱۳.** یک کد VHDL برای پیاده‌سازی MUX 8:1 نشان داده شده در زیر با استفاده از دستور case بنویسید. در دیاگرام جعبه‌ی سیاه نشان داده شده، ورودی CE یک فعال‌ساز چیپ (تراسه) است. هرگاه  $CE = '1'$  باشد، خروجی مانند مالتی‌پلکسر مربوط به مثال ۱۰ کار خواهد کرد. هرگاه  $CE = '0'$  باشد، خروجی مالتی‌پلکسر برابر  $'0'$  خواهد شد.



حل: جواب در برنامه‌ی ۱۳-۵ نشان داده شده است. برای راحتی، اعلان موجودیت نیز نشان داده شده است. در اینجا دستور case داخل بدنی دستور if استفاده شده است؛ از نظر فنی به این کار، حالت تودرتو گفته می‌شود. ملاحظه می‌کنید که با پیچیده‌تر شدن یک مدار، تعداد راههای پیاده‌سازی آن نیز بیشتر می‌شود.

Listing 5.13: Solution to Example 13.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity mux_8to1_ce is
    port ( Data_in : in std_logic_vector (7 downto 0);
           SEL : in std_logic_vector (2 downto 0);
           CE : in std_logic;
           F_CTRL : out std_logic);
end mux_8to1_ce;
-- architecture
architecture my_case_ex of mux_8to1_ce is
begin
    my_mux: process (SEL,Data_in,CE)
    begin
        if (CE = '1') then
            case (SEL) is
                when "000" => F_CTRL <= Data_in(0);
                when "001" => F_CTRL <= Data_in(1);
                when "010" => F_CTRL <= Data_in(2);
                when "011" => F_CTRL <= Data_in(3);
                when "100" => F_CTRL <= Data_in(4);
                when "101" => F_CTRL <= Data_in(5);
                when "110" => F_CTRL <= Data_in(6);
                when "111" => F_CTRL <= Data_in(7);
                when others => F_CTRL <= '0';
            end case;
        else
            F_CTRL <= '0';
        end if;
    end process my_mux;
end my_case_ex;
```

یک نکته‌ی بسیار مهم در ارتباط با مثال ۱۳، گنجاندن دستور case داخل سازه‌ی if است که به آن تودرتو کردن<sup>۱</sup> گفته می‌شود. در مدل‌های رفتاری، دستورات ترتیبی تودرتو رایج بوده و اغلب از این تکنیک استفاده می‌شود. در واقع، این روش یکی از تکنیکهایی است که موجب قدرتمندتر شدن مدل‌سازی رفتاری نسبت به مدل‌سازی جریان داده شده است. واقعیت این است که دستورات تخصیص شرطی و انتخاب شده‌ی سیگنال را نمی‌توان (نسبت به هم) تودرتو کرد (اما در اینجا در مدل‌سازی رفتاری این کار را می‌توان انجام داد – م).

...

<sup>1</sup> Nesting

در VHDL، بهترین رویکرد این است که هر یک از دستورات `process` را حول یک عملکرد خاص متمرکز کرده و از چندین فرآیند مختلف که با هم ارتباط دارند، استفاده کنید. با این کار ابزارسترنز مداری ارائه می‌دهد که هم ساده است و هم کار می‌کند.

رویکرد بد این است که یک فرآیند بزرگ داشته و هر کاری را داخل آن انجام دهید. با این کار ممکن است بتوانید به مداری برسید که کار کند و حتی از نظر معیارهایی مانند سرعت بهینه باشد، اما این امکان نیز وجود دارد که چنین چیزی اتفاق نیفتد.

برخلاف زبانهای برنامه‌نویسی الگوریتمی سطح بالا که حجم کوچکی از کد اغلب موجب تولید کد موثر و کارایی می‌شود، در VHDL برای تولید یک کد کارآمد و موثر باید از بخش‌بندی ساده و فشرده (بسته و متناسب با سازه‌های سخت‌افزاری موجود) استفاده کرد. به بیان دیگر مدل‌های ساده‌ی VHDL بهتر هستند اما سادگی از طریق بخش‌بندی مناسب کد و توصیف مناسب مدل به دست می‌آید. لذا هیچ وقت به دنبال رقابت با دوستان خود برای هرچه کوتاه‌تر نوشتن کد VHDL خود نباشید!.

## ۶-۵ جمع‌بندی

اجازه دهید برخی مفاهیم مهمی که در این فصل معرفی شدند را با هم مرور کنیم:

- سه شیوه‌ی اصلی مدل‌سازی در VHDL عبارتند از جریان داده، رفتاری، و ساختاری.
- مدل‌های رفتاری، برطبق تعریف خود، از دستورات فرآیند استفاده می‌کنند.
- مدل‌های جریان داده، بر طبق تعریف خود، از تخصیص همزمان سیگنال، تخصیص شرطی سیگنال، و/یا تخصیص انتخاب شده‌ی سیگنال استفاده می‌کنند.
- دستور `process` یک دستور همزمان است. دستوراتی که داخل بدنی دستور `process` استفاده می‌شوند، دستورات ترتیبی هستند.
- دستور `if` شباهت مستقیمی با دستور تخصیص شرطی سیگنال، استفاده شده در مدل‌سازی جریان داده دارد.
- دستور `case` شباهت مستقیمی با دستور تخصیص انتخاب شده‌ی سیگنال، استفاده شده در مدل‌سازی جریان داده دارد.
- هر دو دستور `case` و `if` را می‌توان به صورت تودرتو استفاده کرد. اما دستورات تخصیص همزمان سیگنال، تخصیص شرطی سیگنال، و تخصیص انتخاب شده‌ی سیگنال را نمی‌توان به صورت تودرتو استفاده کرد.
- ساده‌ترین دستور همزمان، دستور تخصیص همزمان سیگنال (مانند "`F <= A`") است. معادل ترتیبی این دستور، دستور تخصیص ترتیبی سیگنال است که ظاهر این دو کاملاً یکی است.

## فصل ششم

### VHDL عملگرهای

تا بحال به صورت غیرمستقیم برخی عملگرهای موجود در VHDL را معرفی کرده بودیم. در این فصل لیست کامل این عملگرها به همراه تعدادی مثال ارائه می‌شود. جدول ۶-۱ لیست کامل عملگرهای موجود در VHDL را نشان می‌دهد. گرچه ممکن شما نیاز فوری به این عملگرها پیدا نکنید اما آشنایی با آنها مفید است. اطلاعات خیلی زیادی در این فصل راجع به عملگرها داده نمی‌شود.

Operator type						
logical	and	or	nand	nor	xor	xnor
relational	=	/=	<	<=	>	>=
shift	sll	srl	sla	sra	rol	ror
adding	+	-	&			
sign	+	-				
multiplying	*	/	mod	rem		
miscellaneous	**	abs	not			

Table 6.1: VHDL operators.

عملگرها در VHDL به هفت نوع مختلف دسته‌بندی می‌شوند: منطقی (Logical)، نسبی/نسبتی (Relational)، شیفت (Shift)، جمع (Adding)، علامت (Sign)، ضرب (Multiplying)، و دم دستی (Miscellaneous). ترتیبی که در اینجا برای عملگرها استفاده شده مهم است زیرا بیانگر اولویت آنها نسبت به هم است (اولویت‌ها در اینجا نسبت به هم در حال افزایش هستند؛ بنابراین، عملگرهای منطقی دارای پایین‌ترین اولویت هستند). توصیه می‌شود به این اولویت‌گذاری زیاد تکیه نکنید بلکه از پرانتزها برای تعیین قطعی اولویت‌های مد نظر خود استفاده کنید. بین عملگرهایی که در هر سطر از جدول ۶-۱ آورده شده است، اولویت خاصی وجود ندارد و به همان ترتیبی که روی عملوندها استفاده شده‌اند، به همان ترتیب نیز ارزیابی می‌شوند.

## ۲-۶ عملگرهای منطقی

عملگرهای منطقی کاملاً شناخته شده هستند. تنها نکته قابل توجه در اینجا این است که عملگر `not` از نظر فنی یک عملگر منطقی نیست و در زمرة عملگرهای دم دسته‌بندی می‌شود؛ بنابراین، اولویت بالایی دارد.

## ۳-۶ عملگرهای رابطه‌ای

این‌ها هم عملگرهای کاملاً شناخته شده‌ای بوده و نیاز به توضیح بیشتر ندارند. لیست کامل عملگرهای رابطه‌ای در جدول ۲-۶ نشان داده شده است.

Operator	Name	Explanation
<code>A = B</code>	equivalence	is A equivalent to B?
<code>A /= B</code>	non-equivalence	is A not equivalent to B?
<code>A &lt; B</code>	less than	is A less than B?
<code>A &lt;= B</code>	less than or equal	is A less than or equal to B?
<code>A &gt; B</code>	greater than	is A greater than B?
<code>A &gt;= B</code>	greater than or equal	is A greater than or equal to B?

Table 6.2: VHDL relational operators with brief explanations.

## ۳-۶ عملگر شیفت

در بسته `ieee.numeric_std` یا بسته `ieee.numeric_bit` سه نوع عملگر شیفت تعریف شده است: شیفت منطقی، شیفت حسابی، و شیفت چرخشی. عملگرهای شیفت در جدول ۳-۶ فهرست شده‌اند.

Operator	Name		Example	Result
logical	sll	shift left logical	<code>result &lt;= "10010101" sll 2</code>	"01010100"
	srl	shift right logical	<code>result &lt;= "10010101" srl 3</code>	"00010010"
arithmetic	sla	shift left arithmetic	<code>result &lt;= "10010101" sla 3</code>	"10101111"
	sra	shift right arithmetic	<code>result &lt;= "10010101" sra 2</code>	"11100101"
rotate	rol	rotate left	<code>result &lt;= "10100011" rol 2</code>	"10001110"
	ror	rotate right	<code>result &lt;= "10100011" ror 2</code>	"11101000"

Table 6.3: VHDL shift operators with examples.

تفاوتها بین این عملگرهای شیفت به شرح زیر است:

- هر دو عملگر شیفت منطقی، جاهای خالی حاصل از شیفت دادن را با صفر پر می‌کنند. تفاوت بین شیفت منطقی و شیفت حسابی در این است که در شیفت حسابی، بیت علامت هرگز تغییر نمی‌کند. در شیفت حسابی به راست، بیت علامت جاهای خالی ایجاد شده در سمت چپ را پر می‌کند. در شیفت حسابی به چپ، از سمت راست صفر جاهای خالی تولید شده را پر می‌کند.
- عملگرهای شیفت چرخشی، بیت خارج شده از یک طرف را به جای خالی ایجاد شده در طرف دیگر انتقال می‌دهد.

#### ۶-۴ دیگر عملگرها

گروههای دیگر از عملگرها معمولاً به همراه (یا روی) انواع عددی استفاده می‌شود. در اینجا جزئیات عملیات عددی توضیح داده نمی‌شود لذا تنها لیست این عملگرها در جدول ۶-۴ نشان داده شده است. در بین این عملگرها، عملگرهای `&`, `mod` و `rem` از جایگاه و توجه ویژه‌ای برخوردار هستند؛ این عملگرها نیز محدود به کار با انواع خاصی هستند که در اینجا ذکر نمی‌شود.

Operator		Name	Comment
adding	+	addition	
	-	subtraction	
	&	concatenation	can operate only on specific types
sign	+	identity	unary operator
	-	negation	unary operator
multiplying	*	multiplication	
	/	division	often limited to powers of two
	mod	modulus	can operate only on specific types
	rem	remainder	can operate only on specific types
miscellaneous	**	exponentiation	often limited to powers of two
	abs	absolute value	

Table 6.4: All the other VHDL operators not listed so far.

#### ۶-۵ عملگر تجمعی

عملگر تجمعی (`&`) اغلب در مدارات دیجیتال، عملگر مفیدی است. موقع زیادی پیش می‌آید که نیاز دارید دو مقدار مختلف را در کنار هم به صورت یک واحد فشرده درآورید. عملگر تجمعی<sup>۱</sup> در برخی

<sup>1</sup> Concatenation operator

مثالهایی که قبلاً دیدیم، استفاده شده بود. برنامه‌ی ۶-۱ برخی مثالهای بیشتر از این عملگر را نشان می‌دهد.

Listing 6.1: Examples of the concatenation operator.

```
signal A_val, B_val : std_logic_vector(3 downto 0);
signal C_val : std_logic_vector(5 downto 0);
signal D_val : std_logic_vector(7 downto 0);
-----
C_val <= A_val & "00";
C_val <= "11" & B_val;
C_val <= '1' & A_val & '0';
D_val <= "0001" & C_val(3 downto 0);
D_val <= A_val & B_val;
```

## ۶-۶ عملگرهای باقیمانده (Reminder و Modulus)

هر دو عملگر باقیمانده‌ی `rem` و `mod` روی انواع داده‌ای صحیح اعمال شده و هر دو یک مقدار صحیح برمی‌گردانند. تعریف دقیق نحوه عمل این دو عملگر در جدول ۶-۵ آورده شده است. جدول ۶-۶ نیز چند مثال از استفاده از این عملگرهای باقیمانده را نشان می‌دهد. یک قانون کلی که برنامه‌نویسان معمولاً آن را رعایت می‌کنند این است که اگر با اعداد منفی کار می‌کنید، از عملگر `mod` استفاده نکنید. همان طور که از برخی مثالهای اخیر می‌توانید ملاحظه کنید، برخی جوابها ممکن است با انتظار شما تطابق نداشته باشند.

Operator	Name	Satisfies this Conditions
<code>rem</code>	remainder	1. sign of ( $X \text{ rem } Y$ ) is the same as $X$ 2. $\text{abs}(X \text{ rem } Y) < \text{abs}(Y)$ 3. $(X \text{ rem } Y) = (X - (X / Y) * Y)$
<code>mod</code>	modulus	1. sign of ( $X \text{ mod } Y$ ) is the same as $Y$ 2. $\text{abs}(X \text{ mod } Y) < \text{abs}(Y)$ 3. $(X \text{ mod } Y) = (X * (Y - N))$ for some integer $N$

Table 6.5: Definitions of `rem` and `mod` operators. ( $\text{abs} = \text{absolute value}$ )

rem	mod
$8 \text{ rem } 5 = 3$	$8 \text{ mod } 5 = 3$
$-8 \text{ rem } 5 = -3$	$-8 \text{ mod } 5 = 2$
$8 \text{ rem } -5 = 3$	$8 \text{ mod } -5 = -2$
$-8 \text{ rem } -5 = -3$	$-8 \text{ mod } -5 = -3$

Table 6.6: Example of rem and mod operators.

## 6.7 Review of Almost Everything Up to Now

VHDL is a language used to design, test and implement digital circuits. The basic design units in VHDL are the entity and the architecture which exemplify the general hierarchical approach of VHDL. The entity represents the black-box diagram of the circuit or the interface of the circuit to the outside world while the architecture encompasses all the other details of how the circuit behaves.

The VHDL architecture is made of statements that describe the behavior of the digital circuit. Because this is a hardware description language, statements in VHDL are primarily considered to execute concurrently. The idea of concurrency is one of the main themes of VHDL as one would expect since a digital circuit can be modeled as a set of logic gates that operate concurrently.

The main concurrent statement types in VHDL are the concurrent signal assignment statement, the conditional signal assignment statement, the selected signal assignment statement and the process statement. The process statement is a concurrent statement which is constituted of sequential statements exclusively. The main types of sequential statements are the signal assignment statement, the if statement and the case statement. The if statement is a sequential version of conditional signal assignment statement while the case statement is a sequential version of the selected signal assignment statement.

Coding styles in VHDL fall under the category of data-flow, behavioral and structural models. Exclusive use of process statements indicates a behavioral model. The use of concurrent, conditional and selective signal assignment indicate the use of a data-flow model. VHDL code describing more complex digital circuits will generally contain both features of all of these types of modeling.

Since you should make no effort whatsoever to memorize VHDL syntax, it is recommended that a cheat sheet always be kept next to you as you perform VHDL modeling. Developing a true understanding of VHDL is what is going to make you into a good hardware designer. The ability to memorize VHDL syntax proves almost nothing.

## فصل هفتم

### استفاده از VHDL برای مدارات ترتیبی

تمام مداراتی که تاکنون بررسی کردیم از نوع ترکیبی بودند؛ یعنی قادر به ذخیره‌ی اطلاعات نبودند (زیرا فاقد عناصر ذخیره‌کننده مانند فلیپ‌فلاب‌ها بودند). در این بخش، برخی روش‌های توصیف مدارات ترتیبی معرفی می‌شوند. ما بحث خود را به بررسی مدل‌های رفتاری محدود می‌کنیم. البته این امکان وجود دارد و حتی گاه‌آ بهتر این است که از مدهای جریان داده استفاده کنیم اما استفاده از مدل‌های رفتاری بسیار ساده‌تر است.

#### ۷-۱ عناصر ذخیره‌سازی ساده در VHDL

رویکرد عمومی و متداول برای یادگیری نحوه‌ی پیاده‌سازی عناصر ذخیره‌سازی<sup>۱</sup>، مطالعه‌ی ویژگی‌های یک «سلول تزویج متقابل پایه»<sup>۲</sup> یا همان لچ<sup>۳</sup> است. به منظور افزایش کنترل‌پذیری این عنصر، سیگنال کلاک نیز به این عنصر افزوده می‌شود. علاوه بر این؛ برخی مدارات نازک‌سازی<sup>۴</sup> پالس<sup>۵</sup> نیز به مدار حاصل اضافه می‌شود تا در نهایت به مداری به نام «فلیپ‌فلاب»<sup>۶</sup> بررسیم. یک فلیپ‌فلاب چیزی جز یک عنصر ذخیره‌ساز بیت (از نوع) حساس به لبه<sup>۷</sup> (یا تریگر شونده در لبه<sup>۸</sup>) نیست. مثال بعدی به پیاده‌سازی یک فلیپ‌فلاب D حساس به لبه می‌پردازد.

**مثال ۱۴:** یک کد VHDL برای توصیف یک فلیپ‌فلاب D (نشان داده شده در زیر) بنویسید. در توصیف خود، از یک مدل رفتاری استفاده کنید.

<sup>1</sup> Storage elements

<sup>2</sup> Basic cross-coupled cell

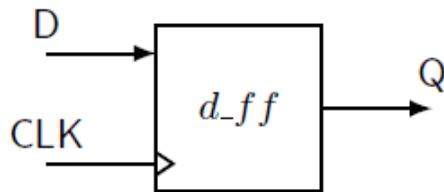
<sup>3</sup> Latch

<sup>4</sup> Pulse narrowing circuitry

<sup>5</sup> Flip Flop

<sup>6</sup> Edge-sensitive

<sup>7</sup> Edge-triggered



حل: جواب این مثال در برنامه‌ی ۷ نشان داده شده است. مهمترین نکات قابل توجه در این کد عبارتند از:

- بدنی معماری استفاده شده، my\_dff نام دارد که گویای این است که توصیف my\_dff از موجودیت d\_ff را ارائه کرده است.
- با توجه به این که باید از یک مدل رفتاری استفاده شود، در بدنی معماری از دستور process استفاده شده است. دستورات داخل بدنی فرآیند (process) به صورت متوالی اجرا می‌شوند (نه موازی). بدنی فرآیند هر زمان که تغییری در یکی از سیگنالهای موجود در لیست حساسیت رخ دهد، اجرا می‌شود. در این جا لیست حساسیت شامل تنها یک سیگنال، سیگنال CLK، است.
- استفاده از سازه‌ی rising\_edge () در دستور if به منظور نشان دادن این مطلب است که خروجی rising\_edge () در لبه‌های بالاروندهای سیگنال ورودی CLK امکان تغییر کردن دارد. سازه‌ی rising\_edge () در واقع تابعی است که در یکی از کتابخانه‌های اعلان شده تعریف شده است. این شیوه‌ای که کد VHDL نوشته شده است، موجب می‌شود تمام مدار، سنکرون شود؛ (به این معنا که) تمام تغییرات در خروجی‌های مدار با لبه‌ی بالاروندهای سیگنال کلک هماهنگ و سنکرون شده‌اند. در این مثال، کاری (یا تغییری) که انجام می‌شود، شامل انتقال مقدار منطقی ورودی D به خروجی Q است.
- یک عبارت معادل برای تابع rising\_edge(CLK) استفاده از عبارت متداول و رایج CLK'event است که در آن از مختصه‌ی<sup>۱</sup> event استفاده شده است. لطفاً این مطلب را به خاطر بسپارید.
- فرآیند دارای برچسبی به نام dff است. استفاده از این برچسب در VHDL ضروری و لازم نیست اما استفاده از آن موجب خود-توصیفگر<sup>۲</sup> شدن کد و در نتیجه، افزایش خوانایی و قابلیت درک آن می‌شود.

<sup>1</sup> Attribute

<sup>2</sup> Self-describing

Listing 7.1: Solution to Example 14.

```

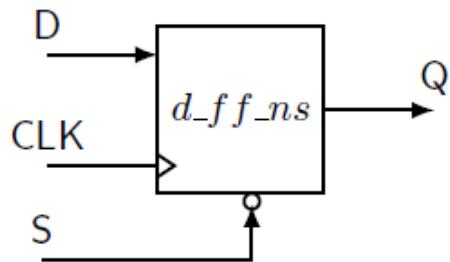
-----
-- Model of a simple D Flip-Flop --
-----
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity d_ff is
    port ( D, CLK : in  std_logic;
           Q :      out std_logic);
end d_ff;
-- architecture
architecture my_d_ff of d_ff is
begin
    dff: process(CLK)
    begin
        if (rising_edge(CLK))      then
--or      if (CLK'event and CLK='1') then
            Q <= D;
        end if;
    end process dff;
end my_d_ff;

```

کدی که در برنامه‌ی ۷-۱ استفاده شده است، به وضوح و روشنی نشان نمی‌دهد که مدار مربوطه قادر به ذخیره سازی یک بیت است. قابلیت ذخیره‌سازی یک مدار به نحوی تفسیر کد VHDL مربوطه توسط کامپایلر بستگی دارد. در کد اخیر، مشخص نشده است که اگر شرط استفاده شده در دستور if برآورده نشود، چه اتفاقی می‌افتد و مدار باید چه کاری انجام دهد. در چنین موقعی که تکلیف عدم رخداد برخی شرایط مشخص نشده است، فرض بر حفظ مقدار قبلی است؛ بنابراین، در کد اخیر، اگر شرط دستور if برقرار نشود، مقدار قبلی خروجی باید حفظ شود و برای این کار باید از عناصر ذخیره‌ساز بیت استفاده شود. مداراتی که دارای قابلیت حفظ مقدار فعلی (یا همان حالت مدار) هستند، طبعاً از ویژگی حافظه برخوردارند.

**مثال ۱۵:** یک کد VHDL برای توصیف فلیپ فلاب D نشان داده شده در زیر بنویسید. در توصیف خود از یک مدل رفتاری استفاده کنید. ورودی S در این مدار یک ورودی فعال-پایین<sup>۱</sup> و سنکرون است که موجب سیت شدن (یعنی گرفتن مقدار ۱ در) خروجی می‌شود.

<sup>۱</sup> Active-low



حل: جواب به این مثال در برنامه ۷-۲ نشان داده شده است. در اینجا به چند نکته‌ی جالب توجه کنید:

- ورودی S از نوع سنکرون است زیرا در لبه‌ی بالارونده‌ی کلاک است که اجازه دارد روى رفتار فلیپ فلاب اثر بگذارد.
- در هنگام وقوع لبه‌ی بالارونده‌ی کلاک، ورودی S اولویت بالاتری (نسبت به ورودی D) دارد زیرا در دستور if قبل از ورودی D مورد بررسی و ارزیابی قرار گرفته است.

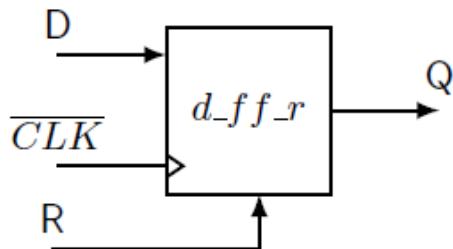
Listing 7.2: Solution to Example 15.

```

-- RET D Flip-flop model with active-low synchronous set input. --
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity d_ff_ns is
    port ( D,S :  in  std_logic;
           CLK :  in  std_logic;
           Q :  out std_logic);
end d_ff_ns;
-- architecture
architecture my_d_ff_ns of d_ff_ns is
begin
    dff: process (CLK)
    begin
        if (rising_edge(CLK)) then
            if (S = '0') then
                Q <= '1';
            else
                Q <= D;
            end if;
        end if;
    end process dff;
end my_d_ff_ns;

```

**مثال ۱۶:** یک کد VHDL برای توصیف فلیپ فلاب D نشان داده شده در زیر بنویسید. در توصیف خود از یک مدل رفتاری استفاده کنید. ورودی R یک ورودی فعال-بالا و آسنکرون است که هر زمان که فعال شود موجب ریست شدن (یعنی تولید مقدار صفر در) خروجی می‌شود.



حل: جواب این مثال در برنامه‌ی ۷-۳ نشان داده شده است. در اینجا به چند نکته‌ی جالب زیر توجه کنید:

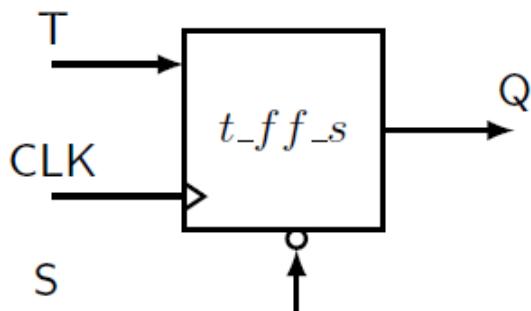
- ورودی ریست (R) مستقل از کلک بوده و اولویت بالاتری نسبت به آن دارد. این اولویت‌گذاری به این صورت برقرار شده است که در دستور if ابتدا وضعیت ورودی R بررسی می‌شود؛ اگر این ورودی مقدار '1' نداشته باشد، آن‌گاه بقیه‌ی ورودی‌ها و شرایط بررسی می‌شوند.
- از تابع falling\_edge برای تولید یک فلیپ فلاب حساس به لبه‌ی پایین‌رونده استفاده شده است. این تابع نیز در کتابخانه‌ای که اعلان شده، تعریف شده است.

Listing 7.3: Solution to Example 16.

```
-- FET D Flip-flop model with active-high asynchronous reset input. --
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity d_ff_r is
    port (    D,R :  in  std_logic;
              CLK :  in  std_logic;
              Q :  out std_logic);
end d_ff_r;
-- architecture
architecture my_d_ff_r of d_ff_r is
begin
    dff: process (R,CLK)
    begin
        if (R = '1') then
            Q <= '0';
        elsif (falling_edge(CLK)) then
            Q <= D;
        end if;
    end process dff;
end my_d_ff_r;
```

دو مثال ۱۵ و ۱۶ نحوهی مدیریت ورودی‌های سنکرون و آسنکرون را نشان می‌دهند.

**مثال ۱۷:** یک کد VHDL برای توصیف فلیپ فlap T نشان داده شده در زیر نویسید. در توصیف خود از یک مدل رفتاری استفاده کنید. ورودی S پایین-فعال و آسنکرون بوده و در صورت فعال شدن موجب ست شدن (یعنی تولید مقدار ۱ در) خروجی می‌شود.



**حل:** برنامه‌ی ۴-۷ جواب این مثال را نشان می‌دهد. در این مثال، از چند تکنیک بسیار مهم استفاده شده است که توجه به آنها خالی از فایده نیست:

- در اینجا یک تفاوت اساسی بین فلیپ فلاپ D و فلیپ فلاپ T مشاهده می‌شود. خروجی یک فلیپ فلاپ D تنها به مقدار ورودی D بستگی داشته و تابعی از مقدار فعلی خروجی خود نیست اما خروجی یک فلیپ فلاپ از نوع T هم به ورودی T و هم به مقدار فعلی خروجی این فلیپ فلاپ بستگی دارد. این امر موجب پیچیده‌تر شدن مدل فلیپ فلاپ T در مقایسه با فلیپ فلاپ D می‌شود. در برنامه‌ی ۷-۴ از یک سیگنال موقتی جهت استفاده از حالت فعلی فلیپ فلاپ در ورودی استفاده شده است. با توجه به این که Q به عنوان یکی از پورتهای موجودیت تعریف شده است پس باید یک حالت (مُد) به آن تخصیص داد. در اینجا از مُد خروجی (out) برای این پورت استفاده شده است. سیگنال‌هایی که به عنوان خروجی اعلان می‌شوند را نمی‌توان در سمت راست یک عملگر تخصیص سیگنال استفاده کرد. رویکرد استاندارد برای رفع این (ظاهرً) محدودیت، استفاده از سیگنال‌های میانی<sup>۱</sup> است؛ این سیگنال دیگر نیاز و الزامی به داشتن حالت (مُد) ندارند و بنابراین، در بدنهٔ معماری، هم به عنوان ورودی و هم به عنوان خروجی (یعنی در هر دو طرف عملگر تخصیص سیگنال) می‌توانند استفاده شوند. رویکرد در اینجا این است که سیگنال‌های میانی را در بدنهٔ معماری دستکاری کنیم ولی همزمان از یک دستور «تخصیص همزمان سیگنال» برای تخصیص سیگنال میانی به خروجی مناسب استفاده کنیم. به دستور کلیدی موجود در برنامه‌ی ۷-۴ توجه کنید که در آن سیگنال میانی در هر دو طرف عملگر تخصیص سیگنال حضور دارد. این کار یک رویکرد متداول در VHDL است: لطفاً به اندازهٔ کافی برای فهم این کار وقت بگذارید. نکته‌ی دیگر در این بخش این است که علیرغم وجود حالت (مُد) buffer در VHDL برای دورزدن مشکل مطرح شده در این بخش، توصیه می‌شود که هرگز از این حالت برای توصیف پورتهای یک مدار استفاده نکنید. راه حلی که در برنامه‌ی ارائه شده در اینجا استفاده شد، یک راه حل خوب و مناسب است.
- برنامه‌ای که در اینجا ارائه شده است از معادلهٔ مشخصه‌ی یک فلیپ فلاپ T برای پیاده‌سازی آن استفاده کرده است.
- در VHDL غالباً فلیپ فلاپ‌های D گزینه‌ی مناسب و متداول جهت استفاده به عنوان عنصر ذخیره‌ساز هستند. شما اگر دلیل خاصی برای استفاده از انواع دیگر فلیپ فلاپ‌ها ندارید، بهتر است از آنها استفاده نکنید.

<sup>1</sup> Intermediate signals

Listing 7.4: Solution to Example 17.

```
-- RET T Flip-flop model with active-low asynchronous set input. --
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity t_ff_s is
    port ( T,S,CLK :  in  std_logic;
           Q :  out std_logic);
end t_ff_s;
-- entity
architecture my_t_ff_s of t_ff_s is
    signal t_tmp : std_logic; -- intermediate signal declaration
begin
    tff: process (S,CLK)
    begin
        if (S = '0') then
            t_tmp <= '1';
        elsif (rising_edge(CLK)) then
            t_tmp <= T XOR t_tmp; -- temp output assignment
        end if;
    end process tff;
    Q <= t_tmp; -- final output assignment
end my_t_ff_s;
```

## ۷-۲ استنباط حافظه: مدل جریان داده در مقابل مدل رفتاری

بخش عمده‌ای از طراحی دیجیتال با مدارات ترتیبی سروکار دارد. اگر بخواهیم کلی صحیت کنیم، اکثر طراحی‌های مدارات ترتیبی قصد سنکرون‌سازی وقایع را با یک لبه‌ی کلاک دارند؛ به بیان دیگر، تغییرات خروجی تنها در لبه‌های کلاک رخ می‌دهند. این توصیف از عناصر حافظه که در اینجا مطرح شد ممکن است به خواننده القا کند که عناصر حافظه در VHDL تنها با مدل‌های رفتاری مرتبط هستند، در صورتی که این طور نیست. همین مفهوم تولید (یا استنباط<sup>1</sup>) حافظه در مدل‌سازی جریان داده نیز برقرار و صادق است: اگر تکلیف مقدار یک خروجی را برای تمام شرایط مختلف ورودی تعیین نکنید، یک لچ (یعنی یک عنصر حافظه) تولید (و استنباط) خواهد شد. بنابراین یکی از وظایف طراح دیجیتال، بررسی به منظور جلوگیری از تولید ناخواسته‌ی عناصر حافظه است زیرا این عناصر حافظه‌ی ناخواسته موجب افزایش غیرضروری پیچیدگی مدار تولید شده‌ی نهایی می‌شوند. اگر چه می‌توان فلیپ‌فلاب‌ها را به کمک مدل‌سازی جریان داده نیز تولید کرد، اما اکثر افراد صاحب‌نظر در بررسی چنین کدی نمی‌توانند

<sup>1</sup> Induce

به سرعت به هدف کد شما پی ببرند (بنابراین بهتر است در تولید فلیپ‌فلاب‌ها از مدل‌سازی جریان داده استفاده نکنید – م).

### ۷-۳ نکات مهم

- مشخص نگردن مقدار خروجی برای تمام شرایط و حالات ورودی موجب تولید عناصر حافظه می‌شود.
- معمولاً تولید ناخواسته‌ی عناصر حافظه توسط ابزار سنتزر تحت عنوان «تولید لچ» گزارش می‌شود. مجدداً تاکید می‌شود که لچ زمانی تولید می‌شود که یک حالت و وضعیت ورودی مدار وجود دارد که تکلیف خروجی برای آن حالت معین نشده است.
- تولید (ناخواسته‌ی – م) عناصر حافظه در هر دو مدل‌سازی رفتاری و جریان داده ممکن است اتفاق بیفتد.
- اگر در بخش اعلان موجودیت، سیگنالی با حالت (یا مُد) خروجی مشخص شده باشد، دیگر نمی‌توان آن سیگنال را در سمت راست یک عملگر تخصیص سیگنال به کار برد. برای رفع این محدودیت، می‌توان از سیگنال‌های میانی در یک عملیات تخصیص استفاده کرده و سپس آنها را به کمک دستور تخصیص همزمان سیگنال به سیگنال‌ها (یا همان پورتها)ی خروجی تخصیص داد.
- نباید از حالت (مُد) buffer استفاده کرد بلکه بهتر است از روش سیگنال‌های میانی بهره برد.

## فصل هشتم

### طراحی ماشین حالت محدود (FSM) به کمک VHDL

ماشین‌های حالت محدود (FSM<sup>۱</sup>) توصیفات ریاضی برای حل دسته‌ی وسیعی از مشکلات مانند خودکارسازی طراحی الکترونیکی<sup>۲</sup> و طراحی پروتکل‌های ارتباطی هستند. شما ممکن است تاکنون ماشین‌های حالت زیادی را روی کاغذ طراحی کرده باشید. حال در موقعیتی قرار دارید که بتوانید این ماشین‌های را به طور واقعی روی سخت‌افزار پیاده‌سازی و آزمایش کنید. اولین قدم در این کار، آشنایی با نحوه‌ی استفاده از FSM‌ها در VHDL است.

طراحی‌های ساده‌ی FSM تنها به منزله‌ی برداشتن یک قدم بیشتر در طراحی مدارات ترتیبی است که در فصل قبلی بررسی شد. شما در فصل فعلی، تکنیک‌هایی یاد می‌گیرید که به کمک آنها قادر به طراحی FSM‌های نسبتاً پیچیده خواهید شد.

دیاگرام بلوکی مربوط به یک FSM استاندارد از نوع مور<sup>۳</sup> در شکل ۸-۱ نشان داده شده است. این دیاگرام نسبتاً معمولی و شناخته شده است اما برخی بلوکهای آن نام خاص و متفاوتی به خود گرفته‌اند.

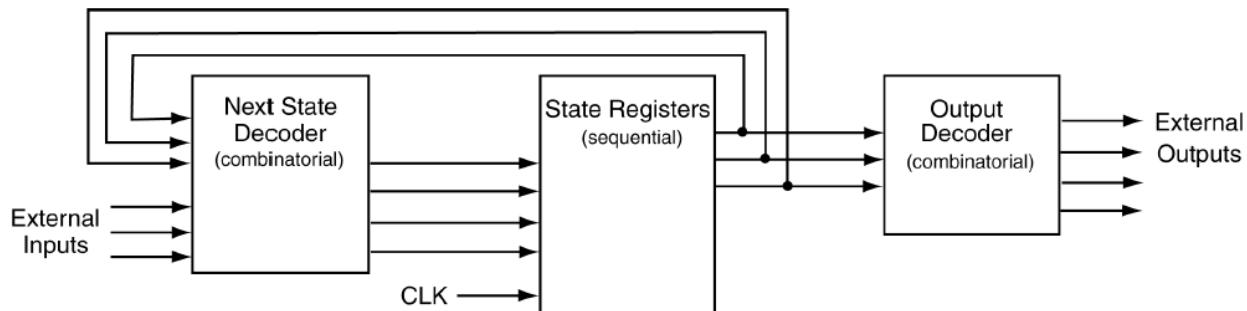


Figure 8.1: Block diagram for a Moore-type FSM.

<sup>1</sup> Finite State machine

<sup>2</sup> Electronic Design Automation

<sup>3</sup> Moore-type FSM

блок Next State Decoder یک блок منطق ترکیبی است که از ورودی‌های خارجی فعلی و حالت فعلی برای تعیین و تصمیم‌گیری در مورد حالت بعدی FSM استفاده می‌کند. به بیان دیگر، ورودی‌های بلک Next State Decoder رمزگشایی (یا دیکد) می‌شوند تا یک خروجی که نمایشگر حالت بعدی FSM است را تولید کنند. در حالت کلی، مدار داخلی بلک Next State Decoder (شامل) معادلات تحریک عناصر حافظه (یعنی فلیپ فلاب‌ها)ی موجود در بلک State Registers هستند. حالت بعدی FSM پس از وقوع و اعمال مجدد کلاک ورودی به بلک State Registers تبدیل به حالت فعلی FSM می‌شوند. بلک State Registers یک عنصر حافظه است که حالت فعلی ماشین را ذخیره می‌کند. از ورودی‌های بلک Output Decoder برای تولید خروجی‌های خارجی مطلوب استفاده می‌شود. این ورودی‌ها به کمک یک منطق ترکیبی رمزگشایی (دیکد) شده تا خروجی‌های خارجی را تولید کنند. علت این که این FSM، از نوع مور نامگذاری شده این است که خروجی‌های خارجی تنها به حالت فعلی ماشین بستگی دارند.

مدل FSM نشان داده شده در شکل ۸-۱ متدائل‌ترین مدل برای توصیف یک FSM از نوع مور است. ما در اینجا می‌خواهیم از دیدگاه VHDL به FSM نگاه کنیم. قدرت واقعی VHDL اینجا با مدیریت موثر FSM‌ها شروع می‌شود. همان طور که خواهید دید، تنوع و قدرت مدل‌سازی‌های رفتاری VHDL موجب می‌شود نیازی به انجام طراحی‌های بزرگ و بدون انتهای نقشه‌ی کارنو<sup>۱</sup> و نیز عناصر منطقی متعدد روی کاغذ نداشته باشیم.

برای مدل کردن FSM در VHDL روش‌های مختلف زیادی وجود دارد. این روش‌ها حاصلِ تنوع‌پذیری وسیع زبان VHDL هستند. روشی که ما در این فصل معرفی و استفاده می‌کنیم، احتمالاً واضح‌ترین و سرراست‌ترین رویکرد برای پیاده‌سازی FSM است. دیاگرام این رویکرد پیاده‌سازی FSM در شکل ۸-۲ نشان داده شده است.

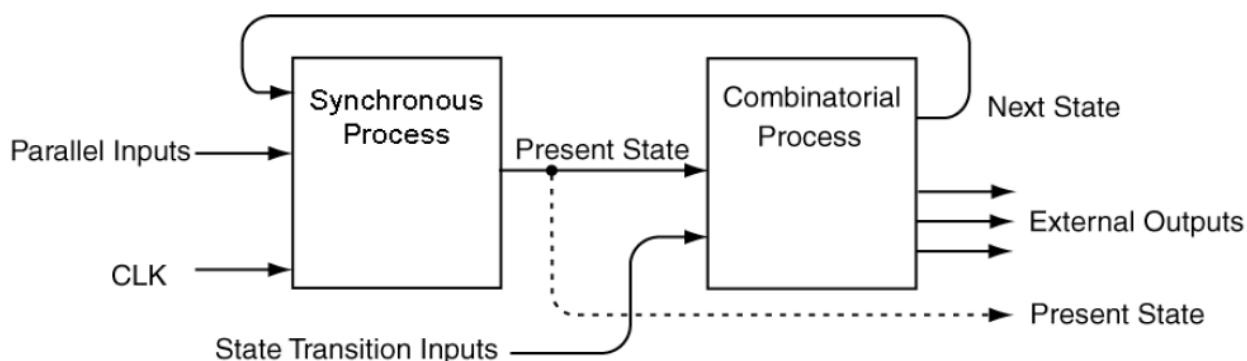


Figure 8.2: Model for VHDL implementations of FSMs.

<sup>1</sup> K-map

گرچه دیاگرام بلوکی اخیر ممکن است در ابتدا چندان واضح و روشن نباشد اما به زودی خواهد دید که مدل FSM نشان داده شده در شکل ۲-۸ یک روش سرراست برای پیاده‌سازی FSM است. رویکردی که ما استفاده خواهیم کرد، FSM را به دو فرآیند VHDL تقسیم می‌کند. یک فرآیند که از آن با نام «فرآیند ترتیبی» (Synchronous Process) یاد می‌شود، تمام کارهای مربوط به کلاک‌دهی و نیز کارهای کنترلی مربوط به عنصر حافظه را مدیریت می‌کند. فرآیند دیگر که «فرآیند ترکیبی» نام دارد، تمام کارهای مربوط به بلوك‌های Output Decoder و Next State Decoder از شکل ۲-۱ را انجام می‌دهد؛ توجه کنید که این دو بلوك از منطق کاملاً ترکیبی تشکیل شده‌اند.

در توصیف سیگنال‌های مربوط به شکل ۲-۸ از برخی علائم و نمادها استفاده می‌شود که به صورت زیر خلاصه شده‌اند:

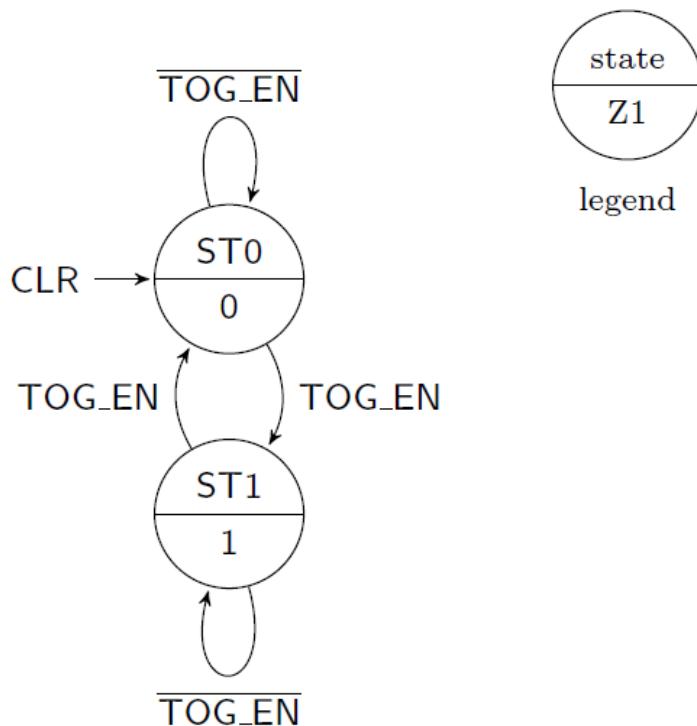
- از ورودی‌هایی که برچسب Parallel Inputs دارند، برای مشخص کردن ورودی‌هایی که به صورت موازی با هم روی هر یک از عناصر حافظه عمل می‌کنند، استفاده می‌شود. این ورودی‌های عبارتند از فعال‌سازها، ست‌کننده‌ها (presets)، پاک‌کننده‌ها (clears) و غیره.
- ورودی‌هایی که برچسب State Transition Inputs دارند، ورودی‌های خارجی هستند که گذرهای حالت را کنترل می‌کنند. علاوه بر این، این ورودی‌ها شامل ورودی‌های خارجی مورد استفاده در دیکد کردن خروجی‌های خارجی نوع میلی<sup>۱</sup> نیز هستند.
- سیگنال‌های (با برچسب Present State) توسط جعبه‌ی Combinatorial Process برای دیکد کردن حالت بعدی و نیز دیکد کردن خروجی استفاده می‌شوند. همچنین، دیاگرام شکل ۲-۸ نشان می‌دهد که متغیرهای Present State می‌توانند در نقش خروجی‌های FSM نیز ظاهر شوند اما نیازی به آنها نیست.

و اما نکته‌ی آخر قبل از این که شروع کنیم: روش‌های مختلف زیادی برای توصیف FSM در VHDL وجود دارد. با فهمیدن روشی که ما در این فصل ارائه می‌دهیم، فهمیدن سایر روش‌ها چندان مشکل نخواهد بود.

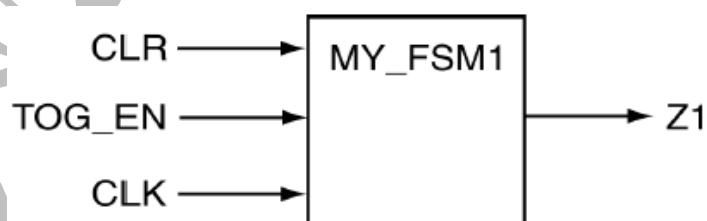
## ۲-۸ نمایش رفتاری FSM در VHDL

مثال ۱۸: یک کد VHDL برای پیاده‌سازی FSM نشان داده شده در زیر بنویسید.

<sup>۱</sup> Mealy-type



حل: این مساله نمایشگر پیاده‌سازی یک FSM پایه است. به منظور آموزش بهتر از دیاگرام جعبه‌ی سیاه استفاده می‌کنیم تا موجودیت مدار را توصیف کنیم. اصولاً در مسائل FSM بهتر است از ترسیم دیاگرام جعبه‌ی سیاه استفاده کنیم. اغلب در مسائل FSM تمیز دادن بین ورودی‌ها و خروجی‌ها چالش‌برانگیز است. ترسیم یک دیاگرام موجل حل این مشکل و چالش می‌شود. دیاگرام جعبه‌ی سیاه و جواب مثال ۱۸ در برنامه‌ی ۱-۸ نشان داده شده است.



Listing 8.1: Solution to Example 18.

```
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_fsm1 is
    port ( TOG_EN : in std_logic;
           CLK,CLR : in std_logic;
           Z1 : out std_logic);
end my_fsm1;
-- architecture
architecture fsm1 of my_fsm1 is
    type state_type is (ST0,ST1);
    signal PS,NS : state_type;
begin
    sync_proc: process(CLK,NS,CLR)
    begin
        -- take care of the asynchronous input
        if (CLR = '1') then
            PS <= ST0;
        elsif (rising_edge(CLK)) then
            PS <= NS;
        end if;
    end process sync_proc;

    comb_proc: process(PS,TOG_EN)
    begin
        Z1 <= '0';          -- pre-assign output
        case PS is
            when ST0 =>    -- items regarding state ST0
                Z1 <= '0';  -- Moore output
                if (TOG_EN = '1') then NS <= ST1;
                else NS <= ST0;
                end if;
            when ST1 =>    -- items regarding state ST1
                Z1 <= '1';  -- Moore output
                if (TOG_EN = '1') then NS <= ST0;
                else NS <= ST1;
                end if;
            when others => -- the catch-all condition
                Z1 <= '0';  -- arbitrary; it should never
                NS <= ST0;  -- make it to these two statements
        end case;
    end process comb_proc;
end fsm1;
```



در این جواب، نکات جالب زیادی وجود دارد که در زیر به آنها اشاره شده است:

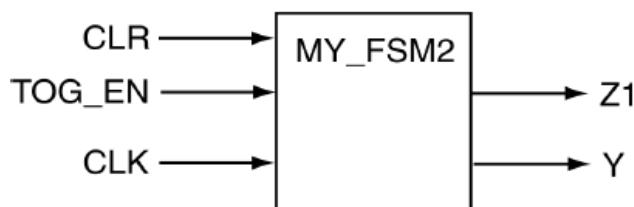
- نوع خاصی را در VHDL به نام state\_type اعلام کرده‌ایم تا نمایشگر حالتها در این FSM باشد.  
این مثال نحوه‌ی استفاده از نوع شمارش شده<sup>1</sup> را در VHDL نشان می‌دهد. مشابه با دیگر زبانهای برنامه‌نویسی، در اینجا هم برای هر حالت لیست شده، یک معادل یا نمایش عددی به صورت داخلی تعریف و استفاده می‌شود؛ گرچه ماتنها با معادل یا نام نمادین این حالتها سروکار داریم. در این مثال، نوعی که ایجاد کرده‌ایم را state\_type نامگذاری کرده و دو متغیر (PS و NS) از این نوع اعلام کرده‌ایم. دقت کنید که state\_type نوعی است که ما ایجاد کرده‌ایم و جزو انواع ذاتی موجود در زبان VHDL نیست.
- فرآیند سنکرون (در اینجا) از نظر شکل و عملکرد معادل با یک فلیپ فلاب D است که قبلاً در مورد مدارات ترتیبی دیده بودیم. تنها تفاوت این است که به جای D و Q به ترتیب از PS و NS استفاده کرده‌ایم. نکته‌ای که باید در اینجا ذکر شود این است که عنصر ذخیره‌ساز در اینجا تنها متناظر با سیگنال PS است. توجه کنید که مقدار PS برای هر ترکیب ممکن از ورودی‌ها مشخص نشده است.
- گرچه این مثال یکی از ساده‌ترین FSM‌هایی است که ممکن است دیده باشید، اما کد آن تا حدی پیچیده است. اما اگر این کد را با دقت بررسی کنید ملاحظه خواهید کرد که هر چیزی که قبلاً در مورد ساختار FSM گفته شد در اینجا به خوبی متناظرسازی شده است. دو فرآیند وجود دارد؛ فرآیند سنکرون، ریست آسنکرون و نیز تخصیص یک حالت جدید در لحظه‌ی ورود کلاک سیستم را مدیریت می‌کند. فرآیند ترکیبی نیز خروجی‌هایی را که در فرآیند سنکرون مدیریت نشده بودند را مدیریت و تعیین تکلیف می‌کند.
- دو فرآیند اشاره شده به صورت موازی و همزمان کار می‌کنند. تغییرات سیگنال NS که نشات گرفته از فرآیند ترکیبی است، به زور موجب ارزیابی فرآیند سنکرون می‌شود. بعد از این که تغییرات در لبه‌ی بالاروندهی بعدی کلاک در فرآیند سنکرون اعمال شدند، تغییرات روی سیگنال PS موجب ارزیابی اجباری فرآیند ترکیبی می‌شود. این ترتیب و اثر متقابل همین طور ادامه پیدا می‌کند.
- دستور case در فرآیند ترکیبی شامل یک عبارت when برای هر حالت FSM است. همچنین از عبارت when others نیز استفاده شده است که تخصیص سیگنال استفاده شده در این بخش اختیاری است زیرا قاعده‌ای که هیچگاه نباید وارد این بخش شود؛ علت استفاده از این کد، کامل کردن ظاهر دستور و خوب جلوه دادن کدنویسی VHDL است.
- خروجی مور تنها تابعی از حالت فعلی است. این امر در کد VHDL این گونه تحقق یافته است که تخصیص خروجی Z1 بدون هیچ قید و شرطی در هر عبارت when از دستور case متعلق

<sup>1</sup> Enumerated type

به فرآیند ترکیبی ارزیابی شده است. به بیان دیگر، متغیر  $Z_1$  درون عبارت `when` اما بیرون از دستور `if` متعلق به عبارت `when` است. علت انجام این کار این است که خروجی‌های مور تنها تابعی از حالتها و نه تابعی از ورودی‌های خارجی هستند. توجه کنید که این ورودی‌های خارجی هستند که تعیین می‌کنند **FSM** در هر حالت فعلی به کدام حالت بعدی گذر کند. بعدها خواهید دید که خروجی‌های میلی، بنا به ذات عمومی خود، درون دستور `if` تخصیص داده می‌شوند.

- به عنوان اولین قدم در فرآیند ترکیبی، خروجی  $Z_1$  پیش-تخصیص<sup>۱</sup> داده می‌شود. این پیش-تخصیص موجب جلوگیری از تولید ناخواسته‌ی لچ برای سیگنال  $Z_1$  می‌شود. در کارکردن با **FSM**‌ها همیشه این تمایل و احتمال طبیعی برای طراح وجود دارد که از تخصیص مقدار برای خروجی  $Z_1$  در هر یک از حالتها غافل شود. پیش-تخصیص موجب جلوگیری از تولید لچ‌ها و تولید کدی تمیز می‌شود. پیش-تخصیص موجب تغییر رفتار و هدف مطلوب کد VHDL نمی‌شود زیرا اگر چند تخصیص همزمان درون یک کد وجود داشته باشد، تنها آخرین تخصیص ملاک عمل قرار خواهد گرفت (ما بقی تخصیص‌ها تاثیری نخواهند داشت – م). به بیان دیگر، در لحظه‌ی خاتمه و اتمام فرآیند، تنها آخرین تخصیص در نظر گرفته می‌شود.

در ارتباط با مثال ۱۸ یک نکته‌ی دیگر نیز قابل تأمل و توجه است. به منظور حفظ سادگی مثال، از مقادیر دیجیتال متغیرهای حالت صرفنظر کردیم. این مطلب در دیاگرام جعبه‌ی سیاه برنامه‌ی ۱-۸ این طور نشان داده شده است که تنها خروجی **FSM**، سیگنال  $Z_1$  است. این کار در صورتی منطقی است که تنها یک سیگنال برای کنترل افزارهای مدارات دیگر لازم باشد. متغیر حالت به صورت داخلی نمایش داده شده و نمایش دقیق آن مهم نیست زیرا متغیر حالت به عنوان یک خروجی در نظر گرفته نشده است. در برخی طراحی‌های **FSM**، متغیرهای حالت به عنوان خروجی در نظر گرفته می‌شوند. برای نمایش این وضعیت، جوابی را برای مثال ۱۸ ارائه می‌دهیم که در آن متغیرهای حالت به عنوان خروجی در نظر گرفته شده‌اند. دیاگرام جعبه‌ی سیاه و کد VHDL مربوط به این وضعیت در برنامه‌ی ۲-۸ نشان داده شده است.



<sup>1</sup> Pre-assign

Listing 8.2: Solution to Example 18 that include state variable as output.

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_fsm2 is
    port (    TOG_EN : in  std_logic;
              CLK,CLR : in  std_logic;
                  Y,Z1 : out std_logic);
end my_fsm2;
-- architecture
architecture fsm2 of my_fsm2 is
    type state_type is (ST0,ST1);
    signal PS,NS : state_type;
begin
    sync_proc: process(CLK,NS,CLR)
    begin
        if (CLR = '1') then
            PS <= ST0;
        elsif (rising_edge(CLK)) then
            PS <= NS;
        end if;
    end process sync_proc;

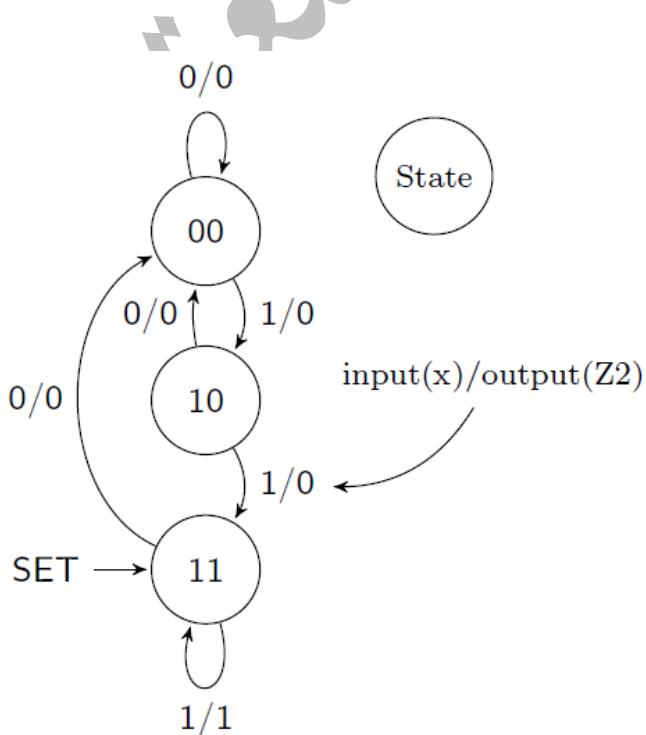
    comb_proc: process(PS,TOG_EN)
    begin
        Z1 <= '0';
        case PS is
            when ST0 =>          -- items regarding state ST0
                Z1 <= '0';      -- Moore output
                if (TOG_EN = '1') then NS <= ST1;
                else   NS <= ST0;
                end if;
            when ST1 =>          -- items regarding state ST1
                Z1 <= '1';      -- Moore output
                if (TOG_EN = '1') then NS <= ST0;
                else   NS <= ST1;
                end if;
            when others =>       -- the catch-all condition
                Z1 <= '0';      -- arbitrary; it should never
                NS <= ST0;       -- make it to these two statements
        end case;
    end process comb_proc;

    -- assign values representing the state variables
    with PS select
        Y <= '0' when ST0,
                  '1' when ST1,
                  '0' when others;
end fsm2;

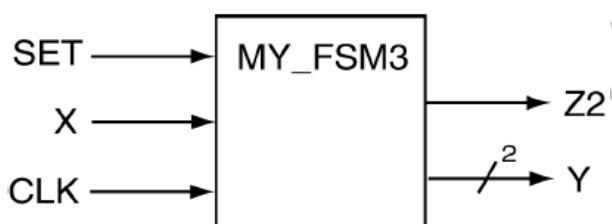
```

توجه کنید که برنامه‌ی ۸-۲ تنها از دو جنبه با برنامه‌ی ۸-۱ متفاوت است. اولین جنبه، اصلاح اعلان موجودیت به گونه‌ای است که خروجی  $Z$  مربوط به متغیر حالت نیز گنجانده شود. دومین جنبه، داخل کردن دستور تخصیص انتخابی سیگنال است که بسته به شرایط و وضعیت متغیر حالت، مقداری را به خروجی متغیر حالت  $Z$  تخصیص می‌دهد. هر زمان که تغییری در سیگنال PS آشکارسازی شود، دستور تخصیص انتخابی سیگنال ارزیابی می‌شود. مجدداً ملاحظه می‌شود که با توجه به این که یک نوع شمارش شده برای متغیرهای حالت اعلان کرده‌ایم، هیچ راهی برای کسب اطلاع دقیق از این که ابزار سنتز چه تصمیمی در مورد نمایش متغیر حالت می‌گیرد، نداریم. دستور تخصیص انتخابی سیگنال در برنامه‌ی ۸-۲ تنها این جنبه را برای ما روشن و نمایان می‌کند که تنها یک متغیر حالت وجود داشته و حالتها با یک '1' و یک '0' نمایش داده می‌شوند. در واقعیت و عمل، روش‌های وجود دارد که به کمک آنها می‌توانیم نحوه نمایش حالتها را کنترل کنیم؛ به زودی با روش‌ها سروکار خواهیم داشت. و آخرین مطلب این که سه دستور همزمان در برنامه‌ی ۸-۲ وجود دارد: دو دستور فرآیند و یک دستور تخصیص انتخابی سیگنال.

**مثال ۱۹:** یک کد VHDL برای پیاده‌سازی FSM نشان داده شده در شکل زیر بنویسید. متغیرهای حالت را به عنوان خروجی‌های FSM در نظر بگیرید.



حل: دیاگرام حالت نشان داده شده برای این مساله نشان می‌دهد که یک FSM سه حالته با یک خروجی خارجی از نوع میلی و یک ورودی خارجی داریم. چون سه حالت وجود دارد، حداقل به دو متغیر حالت برای مدیریت این حالتها نیاز داریم. دیاگرام جعبه‌ی سیاه این جواب در برنامه‌ی ۳-۸ نشان داده شده است. توجه کنید که دو متغیر حالت به صورت یک سیگنال گروهی مدیریت شده‌اند.



Listing 8.3: Solution to Example 19.

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_fsm3 is
    port ( X,CLK,SET : in std_logic;
           Y : out std_logic_vector(1 downto 0);
           Z2 : out std_logic);
end my_fsm3;
-- architecture
architecture fsm3 of my_fsm3 is
    type state_type is (ST0,ST1,ST2);
    signal PS,NS : state_type;
begin
    sync_proc: process(CLK,NS,SET)
    begin
        if (SET = '1') then
            PS <= ST2;
        elsif (rising_edge(CLK)) then
            PS <= NS;
        end if;
    end process sync_proc;

    comb_proc: process(PS,X)
    begin
        Z2 <= '0'; -- pre-assign FSM outputs
        case PS is
            when ST0 => -- items regarding state ST0
                Z2 <= '0'; -- Mealy output always 0
                if (X = '0') then NS <= ST0;
                else NS <= ST1;
                end if;
            when ST1 => -- items regarding state ST1
                Z2 <= '0'; -- Mealy output always 0
                if (X = '0') then NS <= ST0;
                else NS <= ST2;
                end if;
            when ST2 => -- items regarding state ST2
                -- Mealy output handled in the if statement
                if (X = '0') then NS <= ST0; Z2 <= '0';
                else NS <= ST2; Z2 <= '1';
                end if;
            when others => -- the catch all condition
                Z2 <= '1'; NS <= ST0;
        end case;
    end process comb_proc;

    -- faking some state variable outputs
    with PS select
        Y <= "00" when ST0,
        "10" when ST1,
        "11" when ST2,
        "00" when others;
end fsm3;

```

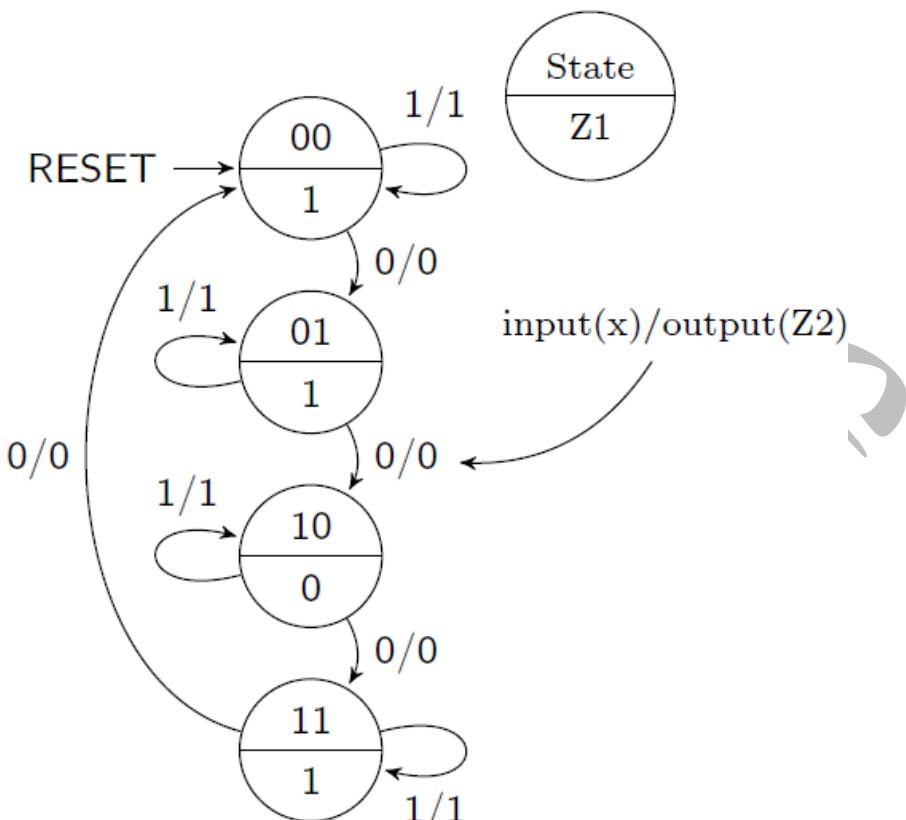
مطابق معمول به چند نکته در ارتباط با مثال ۱۹ توجه کنید. مهمتر از همه این که به شباهت‌های بین این کد و کد مربوط به مثال قبلی توجه کنید.

- FSM دارای یک خروجی از نوع میلی است. کد VHDL در دو عبارت when اول مربوط به دستور case، ناچاراً با این خروجی به عنوان یک خروجی از نوع مور رفتار می‌کند. در آخرین عبارت when، خروجی Z2 در هر دو بخش از دستور if ظاهر شده است. این واقعیت که خروجی Z2 در ارتباط با حالت ST2 متفاوت است، موجب شده است که این خروجی از نوع میلی شده و بنابراین، FSM نیز از نوع میلی شود.

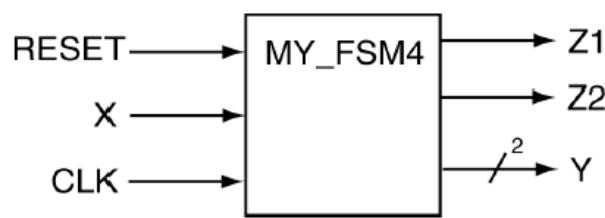
- در ظاهرسازی خروجی‌های متغیر حالت (به خاطر داشته باشید که متغیرهای حالت واقعی به کمک انواع شمارش شده نمایش داده می‌شوند)، دو سیگنال نیاز داریم زیرا دیاگرام حالت شامل بیش از دو حالت (و البته کمتر از پنج حالت) است. در راهکاری که استفاده کردہ‌ایم، این خروجی‌ها به صورت یک گروه<sup>۱</sup> نمایش داده شده‌اند و همین کار باعث شده شکل ظاهری دستور تخصیص انتخاب شده‌ی سیگنال که در انتهای بدنه‌ی معماری قرار دارد، اندکی تغییر کند.

**مثال ۲۰:** یک کد VHDL برای پیاده‌سازی FSM نشان داده شده در شکل زیر بنویسید. متغیرهای حالت لیست شده را به عنوان خروجی در نظر بگیرید.

<sup>۱</sup> Bundle



حل: دیاگرام حالت نشان می‌دهد که پیاده‌سازی آن به چهار حالت، یک ورودی خارجی، و دو خروجی خارجی نیاز دارد. این FSM یک ترکیبی است که در آن دستور if شامل یک خروجی میلی و یک خروجی از نوع مور است اما در اینجا FSM را از نوع میلی در نظر گرفته‌ایم. دیاگرام جعبه‌ی سیاه و کد جواب در برنامه‌ی ۸-۴ نشان داده شده است.



Listing 8.4: Solution to Example 20.

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_fsm4 is
    port ( X,CLK,RESET : in std_logic;
           Y : out std_logic_vector(1 downto 0);
           Z1,Z2 : out std_logic);
end my_fsm4;
-- architecture
architecture fsm4 of my_fsm4 is
    type state_type is (ST0,ST1,ST2,ST3);
    signal PS,NS : state_type;
begin
    sync_proc: process(CLK,NS,RESET)
    begin
        if (RESET = '1') then PS <= ST0;
        elsif (rising_edge(CLK)) then PS <= NS;
        end if;
    end process sync_proc;

    comb_proc: process(PS,X)
    begin
        -- Z1: the Moore output; Z2: the Mealy output
        Z1 <= '0'; Z2 <= '0'; -- pre-assign the outputs
        case PS is
            when ST0 =>      -- items regarding state ST0
                Z1 <= '1'; -- Moore output
                if (X = '0') then NS <= ST1; Z2 <= '0';
                else NS <= ST0; Z2 <= '1';
                end if;
            when ST1 =>      -- items regarding state ST1
                Z1 <= '1'; -- Moore output
                if (X = '0') then NS <= ST2; Z2 <= '0';
                else NS <= ST1; Z2 <= '1';
                end if;
            when ST2 =>      -- items regarding state ST2
                Z1 <= '0'; -- Moore output
                if (X = '0') then NS <= ST3; Z2 <= '0';
                else NS <= ST2; Z2 <= '1';
                end if;
            when ST3 =>      -- items regarding state ST3
                Z1 <= '1'; -- Moore output
                if (X = '0') then NS <= ST0; Z2 <= '0';
                else NS <= ST3; Z2 <= '1';
                end if;
            when others => -- the catch all condition
                Z1 <= '1'; Z2 <= '0'; NS <= ST0;
        end case;
    end process comb_proc;

    with PS select
        Y <= "00" when ST0,
        "01" when ST1,
        "10" when ST2,
        "11" when ST3,
        "00" when others;
    end fsm4;

```

شاید تاکنون متوجه شده باشید که در VHDL، پیاده‌سازی FSM‌ها به کمک مدل رفتاری بسیار سرراست است. در عمل به ندرت پیش می‌آید که لازم باشد FSM را از ابتدا بنویسیم. کافی است از قالب FSM ای که قبلاً نوشته‌ایم، شروع کرده و آن را اصلاح کنیم، البته چندان به مدل‌سازی رفتاری FSM‌ها ساده‌انگار و کم‌توجه نباشد؛ لذت واقعی در تولید FSM‌ای است که برای یک مساله‌ی خاص (و متناسب با مساله اصلاح و) نوشته شده باشد.

## ۲-۸ رمزگذاری One-Hot FSM‌ها

روشهای بسیار زیادی برای رمزگذاری متغیرهای حالت وجود دارد (در اینجا منظور از رمزگذاری، تخصیص یک الگوی منحصر‌بفرد از ۱‌ها و ۰‌ها به هر یک از حالت‌ها و به گونه‌ای است که هر حالت قابل تشخیص و تمایز از بقیه‌ی حالت‌ها شود). اگر برای شما شکل دقیق نمایش مهم است، لازم است کارهای مربوط به کنترل دقیق نحوه‌ی نمایش متغیرهای حالت توسط ابزار سنتز را انجام دهید. برای کنترل این نمایش دو رویکرد وجود دارد. اولین رویکرد، دادن اجازه به ابزار سنتز برای در دست گرفتن جزئیات کار است. با توجه به این که این FSM‌هایی که تا به حال در مثال‌ها بررسی کرده‌ایم همگی از نوع شمارش شده برای نمایش متغیرهای حالت بهره برده‌اند، ابزار سنتز قادر است به انتخاب خود یک طرح رمزگذاری را انتخاب و استفاده کند. واقعیت این است که این ابزارها معمولاً این اختیار و گزینه را در اختیار ما قرار می‌دهند که ما خودمان بتوانیم طرح رمزگذاری مطلوبمان را انتخاب کنیم. اشکال این رویکرد این است که ممکن است شما طرح رمزگذاری خاصی مد نظر داشته باشید که جزو گزینه‌های ابزار سنتز نباشد. رویکرد دوم این است که خودمان در داخل کد VHDL نحوه‌ی رمزگذاری متغیرهای حالت را تعیین کنیم. در این فصل از این رویکرد استفاده شده است.

در طرح رمزگذاری one-hot برای هر حالت متعلق به FSM یک بیت مخصوص در ثبات حالت<sup>۱</sup> در نظر گرفته می‌شود. (بنابراین برای مثال)، برای رمزگذاری یک FSM شامل ۱۶ حالت به کمک روش رمزگذاری one-hot، به ۱۶ فلیپ فلابپ نیاز است. این در حالی است که اگر بخواهیم از روش رمزگذاری باینتری استفاده کنیم، تنها به چهار فلیپ فلابپ نیاز خواهیم داشت. حسن روش رمزگذاری one-hot در این است که منطق و نیز اتصالات بین منطق کلی را ساده‌تر می‌کند. این طرح رمزگذاری معمولاً منجر به تولید FSM‌های کوچک‌تر و سریع‌تر می‌شود.

رویکردی که در مثال‌های قبلی استفاده می‌شد استفاده از روش رمزگذاری کامل<sup>۲</sup> (همان روش رمزگذاری باینتری) برای متغیرهای حالت بود. این روش موجب می‌نمایم شدن تعداد عناصر ذخیره‌ساز (یعنی

<sup>1</sup> State register

<sup>2</sup> Full encoding

فلیپ‌فلاپ‌های لازم برای ذخیره‌ی متغیرهای حالت می‌شود. در این روش، رابطه‌ی بین تعداد

فلیپ‌فلاپ‌های ذخیره‌ساز لازم و تعداد حالت‌های FSM به صورت زیر است:

$$\#(\text{flip-flops}) = \lceil \log_2(\#\text{states}) \rceil \quad (8.1)$$

علامت شبیه به کروشه در رابطه‌ی ۸-۱ نشان دهنده‌ی گردکردن رو به بالا است.

در روش رمزگذاری one-hot تعداد فلیپ‌فلاپ‌های مورد نیاز جهت پیاده‌سازی یک FSM برابر با تعداد حالت‌های آن FSM است:

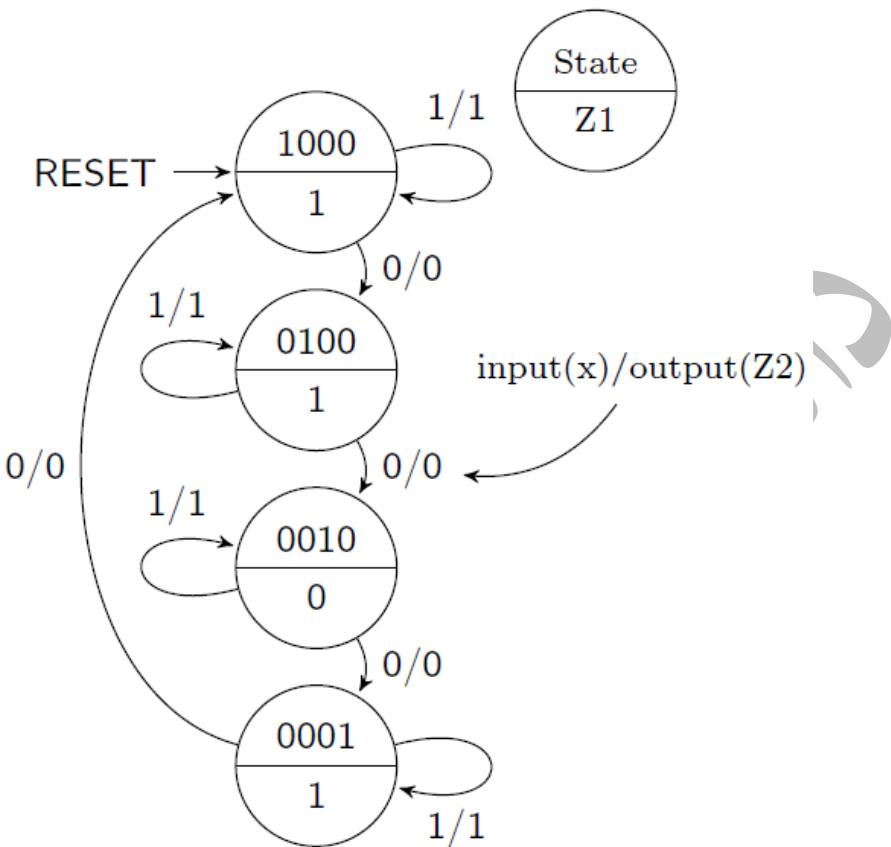
$$\#(\text{flip-flops}) = \#\text{states} \quad (8.2)$$

**مثال ۲۱:** یک کد VHDL برای پیاده‌سازی FSM نشان داده شده در شکل زیر بنویسید. متغیرهای حالت لیست شده را به عنوان خروجی در نظر بگیرید. از روش رمزگذاری one-hot برای متغیرهای حالت استفاده کنید. این مثال عین مثال ۲۰ است اما از رمزگذاری one-hot برای متغیرهای حالت استفاده کرده است.

فصلی شامگاه

<sup>۱</sup> Ceiling function

The ceiling function  $y = \lceil x \rceil$  assigns  $y$  to the smallest integer that is greater or equal to  $x$ .



حل: دیاگرام حالت نشان دهندهٔ چهار حالت، یک ورودی خارجی  $X$ ، و دو خروجی خارجی  $Z_1$  و  $Z_2$  است که خروجی  $Z_2$  یک خروجی میلی است. این FSM از نوع میلی بوده و باید از روش رمزگذاری one-hot برای رمزگذاری متغیرهای حالت استفاده کنیم. پیاده‌سازی این FSM را تکه به تکه به پیش خواهیم برد.

برنامه‌ی ۵-۸ اصلاحات لازم الاجرا روی بخش اعلان موجودیت<sup>۱</sup> مثال ۲۰ را به منظور تبدیل روش رمزگذاری کامل به روش رمزگذاری شبیه‌سازی one-hot<sup>۲</sup> نشان می‌دهد.

برنامه‌ی ۶-۸ اصلاحات لازم الاجرا روی تخصیص خروجی متغیر حالت به منظور حرکت و تغییر از انواع شمارش شده<sup>۳</sup> به شکل خاصی از انواع تخصیص یافته<sup>۴</sup> را نشان می‌دهد. به منظور اعمال اجبار در نمایش متغیرهای حالت به کمک رمزگذاری one-hot واقعی به این دو خط کد اضافه (که در برنامه‌ی ۶-۸ نشان داده شده است)، نیازمندیم. این دو خط کد ابزار سنتز VHDL را وادر می‌کند هر حالت از FSM را با عنصر ذخیره‌ساز خودش نمایش دهد (یعنی روش رمزگذاری one-hot واقعی -م). به بیان دیگر، اجباراً هر حالت با رشته‌ی اصلاحگری که در کد آمده، نمایش داده می‌شود. بدین ترتیب برای

<sup>1</sup> Pseudo one-hot encoding

<sup>2</sup> Enumeration types

<sup>3</sup> Assigned types

پیاده‌سازی FSM باید از چهار بیت برای هر حالت استفاده شود؛ بنابراین به چهار فلیپ فلاب نیاز خواهد بود. در برنامه‌ی ۷-۸ توجه کنید که حتی در قسمت پیش‌فرض مربوط به دستور case از یک مقدار معتبر مربوط به کد one-hot استفاده شده است (به جای مقدار متداول تمام-صفر). سه شکل (یا برنامه‌ی) زیر را با دقت زیاد با هم مقایسه کنید. جواب نهایی در برنامه‌ی ۸-۸ نشان داده شده است.

Listing 8.5: Modifications to convert Example 20 to one-hot encoding.

```
-- full encoded approach
entity my_fsm4 is
    port ( X,CLK,RESET : in std_logic;
           Y : out std_logic_vector(1 downto 0);
           Z1,Z2 : out std_logic);
end my_fsm4;
-----
-- one-hot encoding approach
entity my_fsm4 is
    port ( X,CLK,RESET : in std_logic;
           Y : out std_logic_vector(3 downto 0);
           Z1,Z2 : out std_logic);
end my_fsm4;
```

Listing 8.6: Modifications to convert state variables to use one-hot encoding.

```
-- the approach for enumeration types
type state_type is (ST0,ST1,ST2,ST3);
signal PS,NS : state_type;
-----
-- the approach used for explicitly specifying state bit patterns
type state_type is (ST0,ST1,ST2,ST3);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
signal PS,NS : state_type;
```



Listing 8.7: Modifications to convert state output to pseudo one-hot encoding.

```
-- fake full encoded approach
with PS select
    Y <= "00" when ST0,
    "01" when ST1,
    "10" when ST2,
    "11" when ST3,
    "00" when others;
end fsm4;

-----  
-- one-hot encoded approach
with PS select
    Y <= "1000" when ST0,
    "0100" when ST1,
    "0010" when ST2,
    "0001" when ST3,
    "1000" when others;
end fsm4;
```

Listing 8.8: The final solution to Example 21.

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity my_fsm4_oh is
    port ( X,CLK,RESET : in std_logic;
           Y : out std_logic_vector(3 downto 0);
           Z1,Z2 : out std_logic);
end my_fsm4_oh;
-- architecture
architecture fsm4_oh of my_fsm4_oh is
    type state_type is (ST0,ST1,ST2,ST3);
    attribute ENUM_ENCODING: STRING;
    attribute ENUM_ENCODING of state_type: type is "1000 0100 0010 0001";
    signal PS,NS : state_type;
begin
    begin
        sync_proc: process(CLK,NS,RESET)
        begin
            if (RESET = '1') then PS <= ST0;
            elsif (rising_edge(CLK)) then PS <= NS;
            end if;
        end process sync_proc;

        comb_proc: process(PS,X)
        begin
            -- Z1: the Moore output; Z2: the Mealy output
            Z1 <= '0'; Z2 <= '0'; -- pre-assign the outputs
            case PS is
                when ST0 => -- items regarding state ST0
                    Z1 <= '1'; -- Moore output
                    if (X = '0') then NS <= ST1; Z2 <= '0';
                    else NS <= ST0; Z2 <= '1';
                    end if;
                when ST1 => -- items regarding state ST1
                    Z1 <= '1'; -- Moore output
                    if (X = '0') then NS <= ST2; Z2 <= '0';
                    else NS <= ST1; Z2 <= '1';
                    end if;
                when ST2 => -- items regarding state ST2
                    Z1 <= '0'; -- Moore output
                    if (X = '0') then NS <= ST3; Z2 <= '0';
                    else NS <= ST2; Z2 <= '1';
                    end if;
                when ST3 => -- items regarding state ST3
                    Z1 <= '1'; -- Moore output
                    if (X = '0') then NS <= ST0; Z2 <= '0';
                    else NS <= ST3; Z2 <= '1';
                    end if;
                when others => -- the catch all condition
                    Z1 <= '1'; Z2 <= '0'; NS <= ST0;
            end case;
        end process comb_proc;

        -- one-hot encoded approach
        with PS select
            Y <= "1000" when ST0,
            "0100" when ST1,
            "0010" when ST2,
            "0001" when ST3,
            "1000" when others;
    end fsm4_oh;

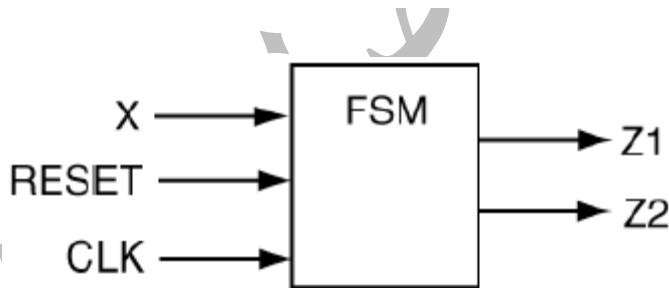
```

### ۳-۸ نکات مهم

- مدل‌سازی FSM‌ها از روی یک دیاگرام حالت در VHDL و به کمک مدل‌سازی رفتاری فرآیندی سرراست است.
- به دلیل تنوع‌پذیری وسیع و عمومی VHDL رویکردهای بسیار زیادی برای مدل‌سازی FSM‌ها در VHDL وجود دارد. رویکردی که در این فصل معرفی شد، یکی از سرراست‌ترین روش‌ها است.
- اگر از انواع شمارش شده در کد VHDL استفاده شود، رمزگذاری متغیرهای حالت FSM در عمل به ابزار سنترستگی دارد. اگر روش رمزگذاری خاصی را برای متغیرهای حالت مد نظر دارید، می‌توانید از روشی که در این فصل مطرح شد استفاده کنید.

### ۴-۸ تمرین‌ها: مدل‌سازی رفتاری FSM‌ها

**EXERCISE 1.** Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and completely label everything.

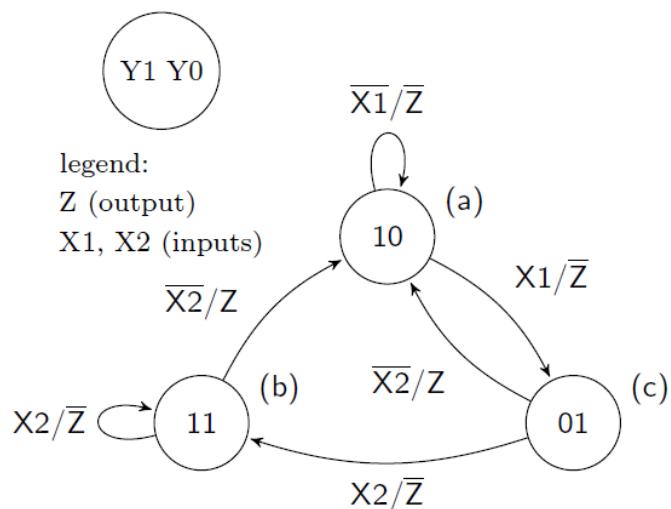


```

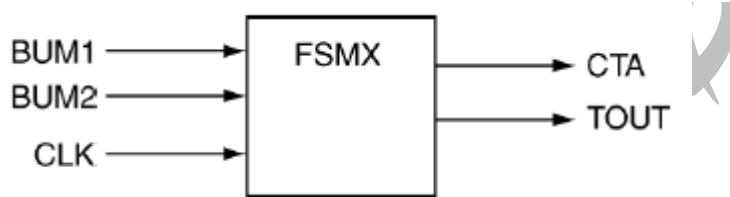
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsm is
    port ( X,CLK : in std_logic;
           RESET : in std_logic;
           Z1,Z2 : out std_logic;
end fsm;
-- architecture
architecture fsm of fsm is
    type state_type is (A,B,C);
    signal PS,NS : state_type;
begin
    sync_proc: process(CLK,NS,RESET)
    begin
        if (RESET = '0') then PS <= C;
        elsif (rising_edge(CLK)) then PS <= NS;
        end if;
    end process sync_proc;
    comb_proc: process(PS,X)
    begin
        case PS is
            Z1 <= '0';      Z2 <= '0';
        when A =>
            Z1 <= '0';
            if (X='0') then NS<=A; Z2<='1';
            else NS <= B; Z2 <= '0';
            end if;
        when B =>
            Z1 <= '1';
            if (X='0') then NS<=A; Z2<='0';
            else NS <= C; Z2 <= '1';
            end if;
        when C =>
            Z1 <= '1';
            if (X='0') then NS<=B; Z2<='1';
            else NS <= A; Z2 <= '0';
            end if;
        when others =>
            Z1 <= '1'; NS<=A; Z2<='0';
        end case;
    end process comb_proc;
end fsm;

```

**EXERCISE 2.** Write a VHDL behavioral model that could be used to implement the state diagram as shown on the right. The state variables should be encoded as listed and also provided as outputs of the FSM.



**EXERCISE 3.** Draw the state diagram associated with the following VHDL code. Be sure to provide a legend and remember to label everything.



دانشگاه صنعتی شاهرود

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsmx is
    Port ( BUM1,BUM2 : in  std_logic;
           CLK : in  std_logic;
           TOUT,CTA : out std_logic);
end fsmx;
-- architecture
architecture my_fsmx of fsmx is
    type state_type is (S1,S2,S3);
    signal PS,NS : state_type;
begin
    sync_p: process (CLK,NS)
    begin
        if (rising_edge(CLK)) then
            PS <= NS;
        end if;
    end process sync_p;

    comb_p: process (CLK,BUM1,BUM2)
    begin
        case PS is
            when S1 =>
                CTA <= '0';
                if (BUM1 = '0')  then
                    TOUT <= '0';
                    NS <= S1;
                elsif (BUM1 = '1') then
                    TOUT <= '1';
                    NS <= S2;
                end if;

            when S2 =>
                CTA <= '0';
                TOUT <= '0';
                NS <= S3;

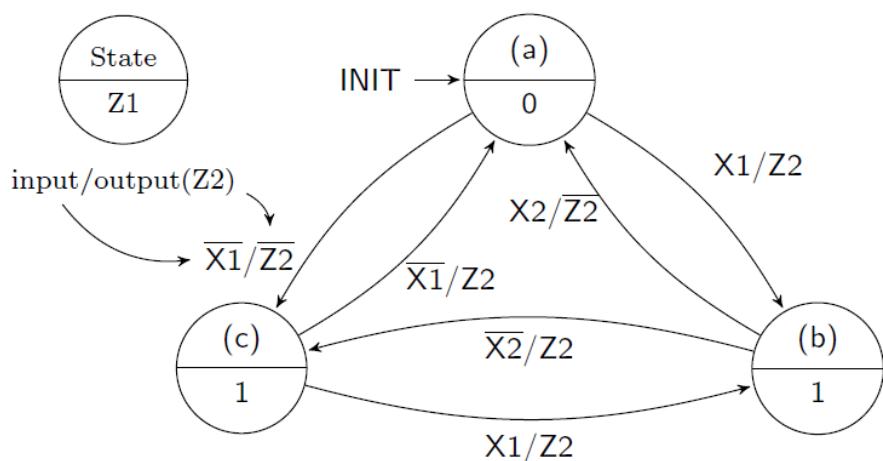
            when S3 =>
                CTA <= '1';
                TOUT <= '0';
                if (BUM2 = '1')  then
                    NS <= S1;
                elsif (BUM2 = '0') then
                    NS <= S2;
                end if;

            when others =>   CTA <= '0';
                                TOUT <= '0';
                                NS <= S1;
        end case;
    end process comb_p;
end my_fsmx;

```

**EXERCISE 4.**

Write the VHDL behavioral model code that could be used to implement the state diagram shown on the right.



**EXERCISE 5.** Draw the state diagram associated with the following VHDL code. Consider the outputs Y to be representative of the state variables. Be sure to provide a legend. Indicate the states with both state variables and their symbolic equivalents.

دانشگاه صنعتی شاهرود

```

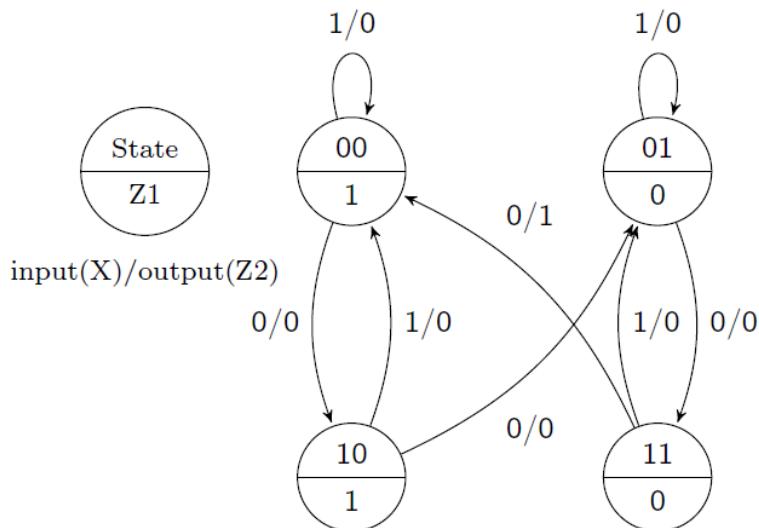
-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsm is
port (    X,CLK : in  std_logic;
            RESET : in  std_logic;
            Z1,Z2 : out std_logic;
            Y : out std_logic_vector(2 downto 0));
end fsm;
-- architecture
architecture my_fsm of fsm is
type state_type is (A,B,C);
attribute ENUM_ENCODING: STRING;
attribute ENUM_ENCODING of state_type: type is "001 010 100";
signal PS,NS : state_type;
begin
begin

sync_proc: process(CLK,NS,RESET) -- process
begin
    if (RESET = '0') then PS <= C;
    elsif (rising_edge(CLK)) then PS <= NS;
    end if;
end process sync_proc;
comb_proc: process(PS,X) -- process
begin
    case PS is
        when A =>
            Z1 <= '0';
            if (X = '0') then NS <= A; Z2 <= '1';
            else NS <= B; Z2 <= '0';
            end if;
        when B =>
            Z1 <= '1';
            if (X = '0') then NS <= A; Z2 <= '0';
            else NS <= C; Z2 <= '1';
            end if;
        when C =>
            Z1 <= '1';
            if (X = '0') then NS <= B; Z2 <= '1';
            else NS <= A; Z2 <= '0';
            end if;
        when others =>
            Z1 <= '1'; NS <= A; Z2 <= '0';
    end case;
end process comb_proc;

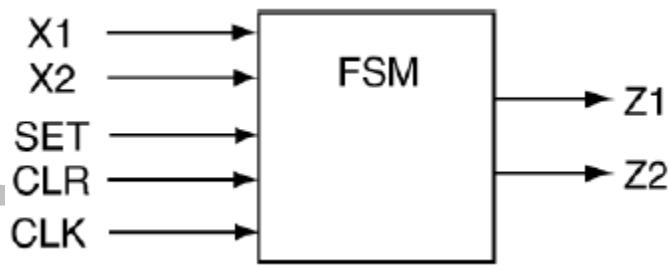
with PS select
Y <= "001" when A,
      "010" when B,
      "100" when C,
      "001" when others;
end my_fsm;

```

**EXERCISE 6.** Write a VHDL behavioral model code that can be used to implement the state diagram shown on the right. All state variables should be encoded as listed and also provided as outputs of the FSM.



**EXERCISE 7.** Draw the state diagram that corresponds to the following VHDL model and state whether the FSM is a Mealy machine or a Moore machine. Be sure to label everything.



شاهرود

```

-- library declaration
library IEEE;
use IEEE.std_logic_1164.all;
-- entity
entity fsm is
    Port ( CLK,CLR,SET,X1,X2 : in std_logic;
           Z1,Z2 : out std_logic);
end fsm;
-- architecture
architecture my_fsm of fsm is
    type state_type is (sA,sB,sC,sD);
    attribute ENUM_ENCODING: STRING;
    attribute ENUM_ENCODING of state_type: type
        is "1000 0100 0010 0001";
    signal PS,NS : state_type;
begin
    sync_p: process (CLK,NS,CLR,SET) -- process
    begin
        if (CLR = '1' and SET = '0') then
            PS <= sA;
        elsif (CLR = '0' and SET = '1') then
            PS <= sD;
        elsif (rising_edge(CLK)) then
            PS <= NS;
        end if;
    end process sync_p;
    comb_p: process (X1,X2,PS) -- process
    begin
        case PS is
            when sA =>
                if ( X1 = '1') then
                    Z1 <= '0'; Z2 <= '0';
                    NS <= sA;
                else
                    Z1 <= '0'; Z2 <= '0';
                    NS <= sB;
                end if;
            when sB =>
                if ( X2 = '1') then
                    Z1 <= '1'; Z2 <= '1';
                    NS <= sC;
                else
                    Z1 <= '1'; Z2 <= '0';
                    NS <= sB;
                end if;
            when sC =>
                if ( X2 = '1') then
                    Z1 <= '0'; Z2 <= '0';
                    NS <= sB;
                else
                    Z1 <= '0'; Z2 <= '1';
                    NS <= sC;
                end if;
            when sD =>
                if ( X1 = '1') then
                    Z1 <= '1'; Z2 <= '1';
                    NS <= sD;
                else
                    Z1 <= '1'; Z2 <= '0';
                    NS <= sC;
                end if;
        end case;
    end process comb_p;
end my_fsm;

```