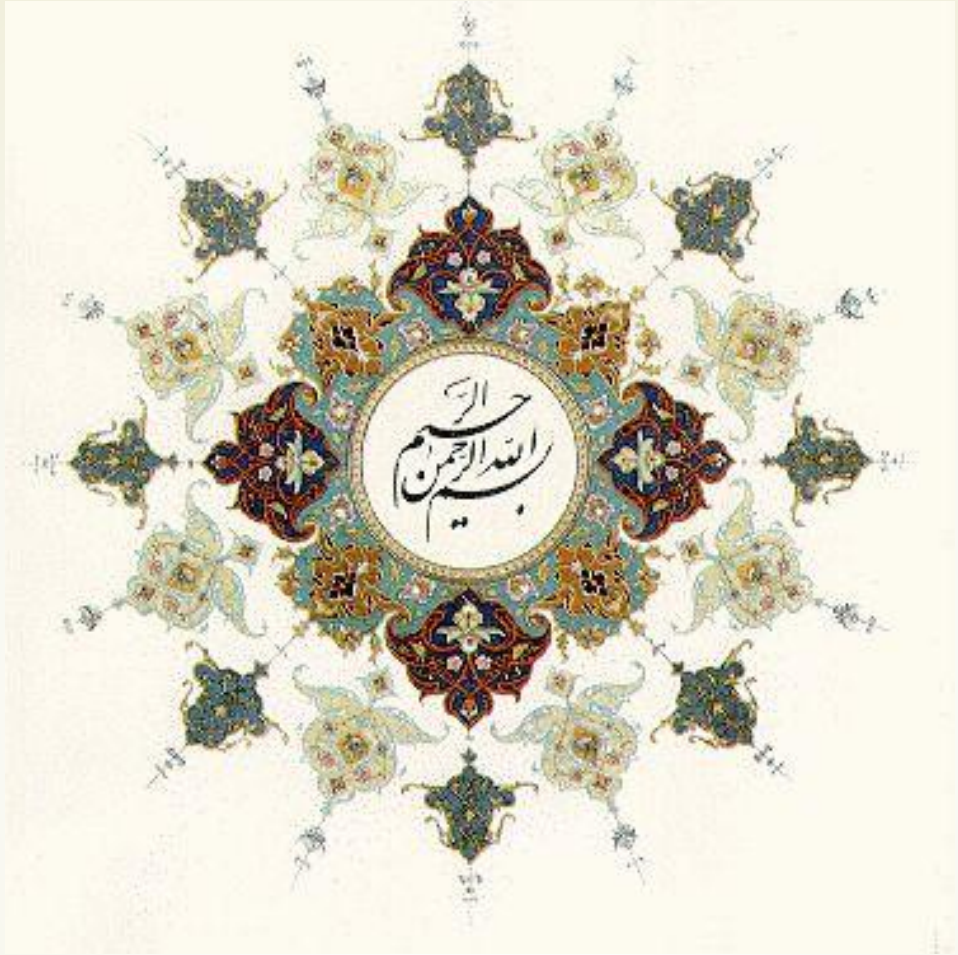


طراحی کامپیوتری سیستم های دیجیتال

References

[1] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*. SunSoft Press, 2nd ed. 2003.

[2] V. A. Pedroni, *Circuit Design with VHDL*. MIT Press, 2011.

[3] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Elsevier Pub., 2004.

۱. مقدمه

۱. ۱ در مورد VHDL

۱.۲ روند طراحی

۱.۳ ابزارهای EDA

۱.۴ تبدیل کد VHDL در یک مدار

۱.۵ نمونه های طراحی

۲. ساختار کد

۲.۱ واحدهای VHDL پایه

۲. ۲ اعلان‌های Library

۲. ۳: ENTITY (نهاد)

۲.۴ Architecture

۲.۵ مثال‌های مقدماتی

۳. انواع داده ها

۱. ۳ انواع داده‌های از پیش تعیین شده

۳.۲ انواع داده‌های تعریف شده کاربر

۳. ۳ زیر نوع ها (subtype)

۳.۴ آرایه‌ها (Arrays)

۳.۵ آرایه پورت

۳. ۶ رکوردها

۳.۷ انواع داده‌های signed و unsigned (علامت دار و بدون علام

۳.۸ تبدیل داده

۳.۹ خلاصه

۳.۱۰ مثال‌های بیشتر

۴. عملگرها و قیدها

۴.۱ عملگرها

۴. ۲ قیدهای از پیش تعریف شده

۴.۳ قیدهای تعریف شده کاربر

۴. ۴ گرانبار کردن عملگر

۴.۵ Generic (ژنریک)

۴.۶ مثال‌ها

۴.۷ خلاصه

۵. کد همروند

۵. ۱ کدهای همروند در برابر کدهای ترتیبی

۵.۲ استفاده از عملگرها

۵.۳ WHEN (ساده و منتخب)

۵.۴ GENERATE

۶. کد ترتیبی

۶.۱ process

۶.۲ signalها و variableها

۶.۳ IF

۶.۴ WAIT

۶.۵ CASE

۶.۶ LOOP (حلقه)

۶.۷ CASE در برابر IF

۶.۸ CASE در برابر WHEN

۶.۹ کلاکینگ نادرست

۶.۱۰ استفاده از کد ترتیبی در طراحی مدارهای ترکیبی

۷. سیگنال‌ها و متغیرها

۷.۱ constant

۷.۲ سیگنال (Signal)

۷.۳ VARIABLE (متغیر)

۷.۴ signal در مقابل variable

۸. ماشین‌های حالت

۸.۱ مقدمه

۸.۲ روش طراحی ۱ #

۸.۳ روش طراحی 2# (خروجی ذخیره شده

۸.۴ روش رمزگذاری: از باینری تا وان‌هات (one Hot)

ارزیابی

گزارش و ارائه	۳ نمره
تمرین	۴نمره
میان ترم	۵ نمره
پایان ترم	۸ نمره
حضور غیاب	۲ نمره

۱.۱ در مورد VHDL

VHDL یک زبان توصیف سخت افزار است. رفتار یک مدار یا سیستم الکترونیکی را توصیف می‌کند، که از آن یک مدار یا سیستم فیزیکی حاصل می‌شود.

VHDL مخفف زبان توصیف سخت افزار VHSIC است.

VHSIC خود مخفف عبارت مدارهای مجتمع خیلی سریع است که با سرمایه‌گذاری اولیه وزارت دفاع ایالات متحده در دهه ۱۹۸۰ منجر به ایجاد زبان برنامه نویسی VHDL شد. اولین نسخه آن ۸۷ VHDL بود و بعداً نسخه به روزرسانی شده آن ۹۳ VHDL نام گرفت. VHDL مبدا و اولین زبان توصیف سخت افزار بود که توسط موسسه مهندسین برق و الکترونیک با استاندارد ۱۰۷۶ IEEE، استانداردسازی شد. بعدها یک استاندارد دیگر به نام ۱۱۶۴ IEEE برای معرفی سیستم منطقی چند مقداره به آن اضافه شد.

VHDL یکی از زبانهای توصیف سخت افزار (**Hardware Description Language**) است که در طراحی مدارهای الکترونیکی مورد استفاده قرار می گیرد که اغلب برای توصیف مدار های یکپارچه (**IC**) دیجیتال یا ترکیبی آنالوگ و دیجیتال مورد استفاده قرار میگیرد . **VHDL** سرنام کلمات **VHSIC Hardware Description Language** است که **VHSIC** خود گرفته شده از سرنام کلمات **Very High Speed Integrated Circuit** می باشد.

VHDL علاوه بر شبیه سازی، برای سنتز مدار نیز به کار می رود. اما همه سازه‌های VHDL که قابل شبیه سازی هستند قابل سنتز نیستند. تاکید ما بیشتر بر روی سازه‌هایی است که قابل سنتز هستند.

انگیزه اصلی استفاده از VHDL (یا نظیر آن، Verilog) این است که VHDL یک استاندارد است، یک زبان مستقل و غیر وابسته به تکنولوژی یا شرکت فروشنده است و بنابراین قابل انتقال و استفاده مجدد است. دو کاربرد اصلی و ویژه VHDL در حوزه ابزارهای منطقی قابل برنامه‌ریزی (شامل CPLDها – ابزارهای منطقی قابل برنامه‌ریزی پیچیده و FPGA ها – آرایه های گیت قابل برنامه‌ریزی میدانی) و در حوزه ASIC (مدارهای مجتمع کاربری ویژه) است. زمانی که کدهای VHDL نوشته شود، می‌تواند هم برای پیاده‌سازی مدار در یک ابزار قابل برنامه‌ریزی (از Altera، Xilinx، Atmel و ...) استفاده شود

می‌تواند در یک کارخانه ذوب فلز برای ساخت پردازنده ASIC آن ثبت گردد. در حال حاضر، پردازنده‌های باارزش پیچیده زیادی (به عنوان مثال، میکروکنترلرها) به این شیوه ساخته می‌شوند.

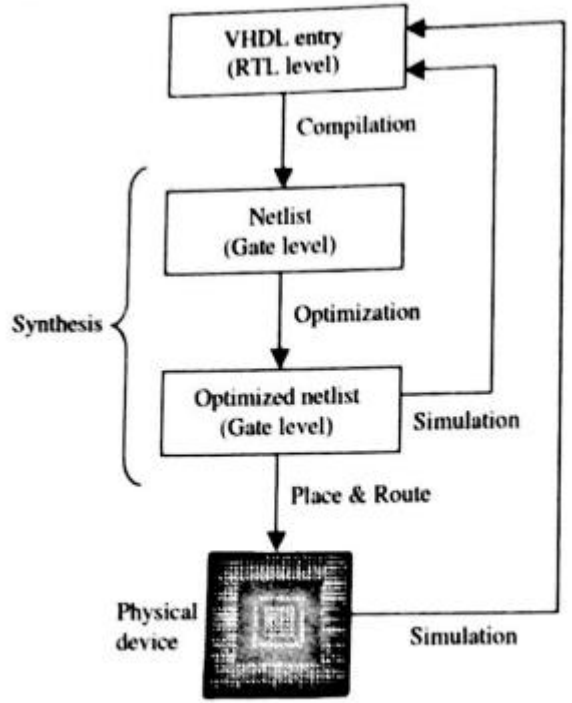
نکته آخر راجع به VHDL این است که برخلاف برنامه‌های کامپیوتری عادی که به طور ترتیبی هستند، اعلان‌های آن ذاتاً همروند هستند (موازی). به این دلیل به VHDL اغلب به جای برنامه به عنوان کد اشاره می‌شود. در VHDL تنها اعلان‌هایی که در یک Process، Function یا Procedure قرار گرفته‌اند به طور ترتیبی اجرا می‌شوند.

۱.۲ روند طراحی

همانطور که گفته شد، یکی از مزایای VHDL این است که امکان سنتز شدن یک مدار یا سیستم در یک ابزار قابل برنامه‌ریزی (PLD یا FPGA) یا در یک ASIC را فراهم می آورد.

مراحل انجام چنین پروژه ای در شکل ۱. ۱ توضیح داده شده است. روند طراحی با نوشتن کد VHDL آغاز می‌شودکه در فایلی با پسوند Vhd. و نامی مشابه نام Entity آن ذخیره شده است. اولین گام در روند سنتز کمپایل کردن است.

کمپایل کردن همان تبدیل زبان VHDL سطح بالا به یک فایل لیست در سطح درگاه است که مدار را در سطح انتقال ثبات (RTL) توصیف می‌کند. گام دوم بهینه سازی است که در لیست سطح درگاه با هدف بهینه سازی سرعت یا حجم اجرا می‌شود. در این مرحله، طراحی می‌تواند شبیه سازی شود. نهایتاً یک سخت افزار جایگزاری، لایه خارجی فیزیکی برای پردازنده PLD یا FPGA یا پوشش های یک ASIC را تولید خواهد کرد.



شکل ۱. ۱: نمایش روال طراحی VHDL

مدارهای مجتمع با کاربرد خاص (*به انگلیسی : Application-specific integrated circuit*) (به اختصار ASIC یا ایسیک) مدارهای مجتمعی هستند که به منظور انجام عملیات خاص، طراحی و بهینه سازی شده‌اند . به عنوان مثال یک پردازنده ویژه که در گوشی موبایل مورد استفاده قرار می گیرد نمونه ای از این نوع تراشه ها می باشند . استفاده از تراشه های ASIC به افزایش کارایی سیستم منتهی می شود اما طرح ایجاد شده از انعطاف پذیری لازم برخوردار نیست . در مقابل، پردازنده ها با وجود انعطاف پذیری از قابلیت های لازم برخوردار نیستند . FPGA ها راه حلی برای ایجاد یک سیستم با انعطاف پذیری بالا و کارایی مورد نیاز می باشند . در ASICهای اولیه از فن‌اوری آرایه گیت استفاده می‌شد. اولین برنامه‌ی تجاری موفق در سال‌های ۱۹۸۱ و ۱۹۸۲ در رایانه‌های ۸ بیتی سری ZX اتفاق افتاد.

۱.۳ ابزارهای EDA

چندین ابزار EDA (اتوماسیون طراحی الکترونیکی) برای سنتز، اجرا و شبیه سازی مدار با استفاده از VHDL وجود دارد.

بعضی از ابزارها (مثلاً Place and route) به عنوان بخشی از روال طراحی بکار برده می‌شوند (خصوصاً Quartus II برای Altera که امکان سنتز کد VHDL در پردازنده‌های CPLD/FPGA شرکت Altera را دارد یا سری ISE زایلینکس که برای پردازنده‌های CPLD / FPGA زایلینکس این امکان را فراهم می‌کند).

ابزارهای دیگر (مثلاً سنتزکننده ها) علاوه بر اینکه به عنوان بخشی از طراحی پیشنهاد می‌شوند همچنین می‌توانند توسط شرکت‌های متخصصEDA نیز تولید شوند. (synopsis – mentor graphics – synplicity و...)

یک نمونه از این مورد Leonardo spectrum است. (سنتزکننده‌ای از شرکت mentor graphics).

synplify (سنتزکننده‌ای از شرکت synplicity) و modelsim (یک شبیه‌ساز از model technology از شرکت mentor graphics).

طراحی‌هایی که در این ارائه شده است در ابزارهای CPLD/FPGA شرکت‌های Altera یا Xilinx سنتز شده‌اند

پروژه یک گروه

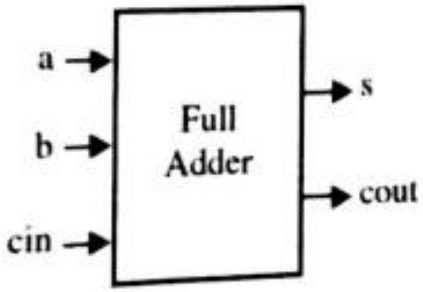
Appendix B: Xilinx ISE + ModelSim Tutorial

Appendix C: Altera MaxPlus II + Advanced Synthesis Software Tutorial

Appendix D: Altera Quartus II Tutorial

۱.۴ تبدیل کد VHDL در یک مدار

شکل ۱.۲ یک واحد تمام جمع کننده را نمایش و شرح می‌دهد. در آن a و b بیت‌های ورودی هستند که باید جمع شوند، cin بیت کری ورودی است، S بیت حاصل جمع است و Cout بیت کری خروجی است. همان طور که در جدول صحت می بینید هرگاه تعداد بیت‌های ۱ ورودی فرد باشد S باید ۱ شود. در حالی که هرگاه دو ورودی یا بیشتر ۱ باشند Cout، ۱ می‌شود.



a b cin	s cout
0 0 0	0 0
0 1 0	1 0
1 0 0	1 0
1 1 0	0 1
0 0 1	1 0
0 1 1	0 1
1 0 1	0 1
1 1 1	1 1

شکل ۲. ۱: دیاگرام تمام جمع کننده و جدول صحت

همانطور که دیده می‌شود متشکل از یک Entity است که توصیف پایه‌های (پورت ها) مدار است و یک Architecture که چگونگی عملکرد مدار را توصیف می‌کند. در شکل دوم می‌بینیم که بیت حاصل جمع از عبارت s=a+b+ Cin بدست می‌آید در حالی که Cout نتیجه عبارت cout=b.a+cin.a+cin.b می‌باشد

```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
       s, cout: OUT BIT);
END full_adder;
---------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```

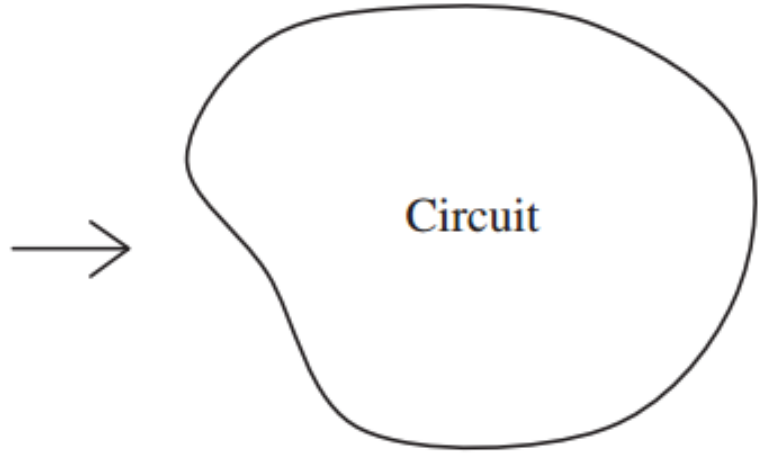


Figure 1.3
Example of VHDL code for the full-adder unit of figure 1.2.

هنگام آزمایش، شکل موج هایی مشابه آنچه در شکل ۱.۵ دیده می‌شود توسط شبیه ساز بدست خواهد آمد. در حقیقت، شکل ۱.۵ حاوی نتایج شبیه سازی مدار سنتز شده کد VHDL شکل ۱.۳ است که واحد تمام جمع کننده شکل ۱.۲ را اجرا می‌کند.

پایه‌های ورودی (توسط یک پیکان به سمت داخل با علامت I از داخل مشخص شده‌اند) و پایه‌های خروجی (توسط یک پیکان به سمت خارج با علامت O از داخل مشخص شده‌اند) در Entity شکل ۱.۳ لیست شده‌اند. می‌توانیم بدون هیچ محدودیتی هر مقداری را به سیگنال‌های ورودی (a و b و cin در این مورد) بدهیم و شبیه ساز، سیگنال‌های خروجی (s و cout) را محاسبه و رسم می نماید. خروجی ها در شکل ۱.۵ مطابق انتظار هستند.

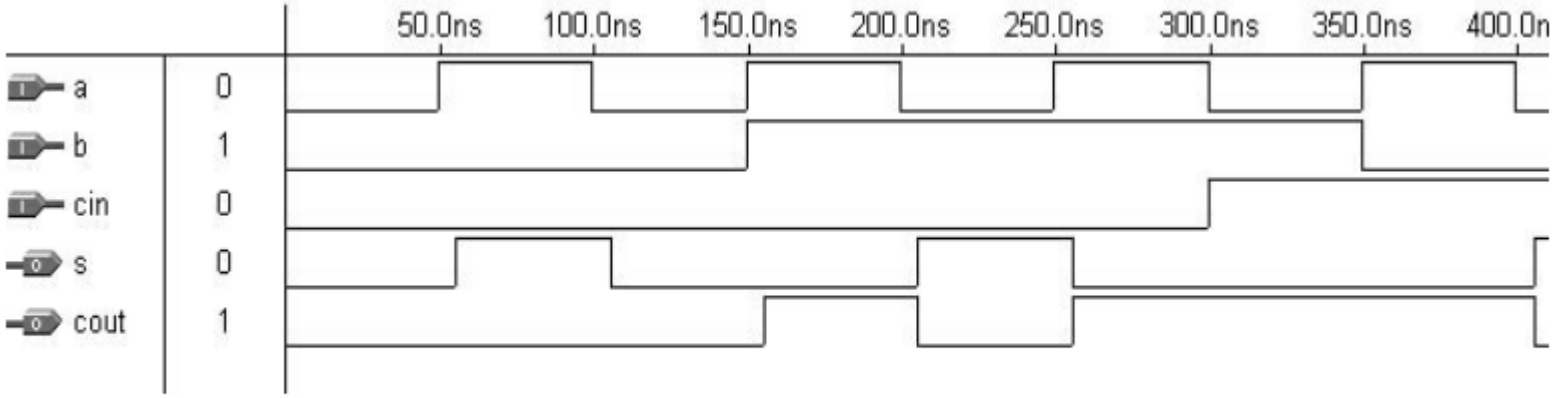


Figure 1.5
Simulation results from the VHDL design of figure 1.3.

۲ ساختار کد

در این فصل به توصیف بخش‌های پایه‌ای خواهیم پرداخت که شامل قسمت‌هایی از کد VHDL است: اعلان‌های LIBRARY، ENTITY و ARCHITECTURE

۲.۱ واحدهای پایه VHDL

همانگونه که در شکل ۲.۱ دیده می‌شود یک بخش مستقل کد VHDL شامل حداقل سه بخش پایه می‌باشد:

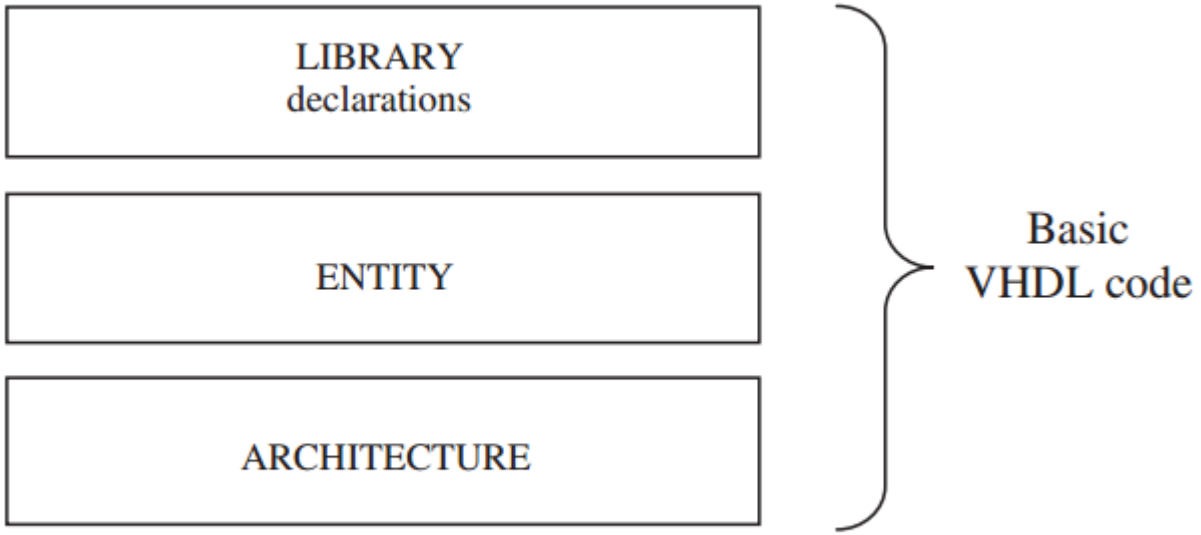


Figure 2.1
Fundamental sections of a basic VHDL code.

- **اعلان‌های LIBRARY (کتابخانه)**: حاوی لیست تمام کتابخانه های بکار رفته در طراحی می‌باشد مثلاً ieee، std، work و ...
- **Entity**: پایه‌های ورودی خروجی I/O مدار را مشخص می‌کند.
- **Architecture**: حاوی کد VHDL ای است که توصیف می‌کند مدار چگونه باید کار کند.

یک Library مجموعه ای از کدهایی است که استفاده آن‌ها رایج است. قرار دادن این کدها در داخل Library، استفاده مجدد و تسهیم آن‌ها را در طراحی‌های دیگرامکان پذیر می نماید.

یک نمونه از ساختار Library در شکل ۲. ۲ نشان داده شده است. معمولاً کد در غالب Functions، Procedures یا Components نوشته می‌شود، که در داخل Package (بسته ها) قرار دارند و سپس در Library مقصد کمپایل می‌شوند.

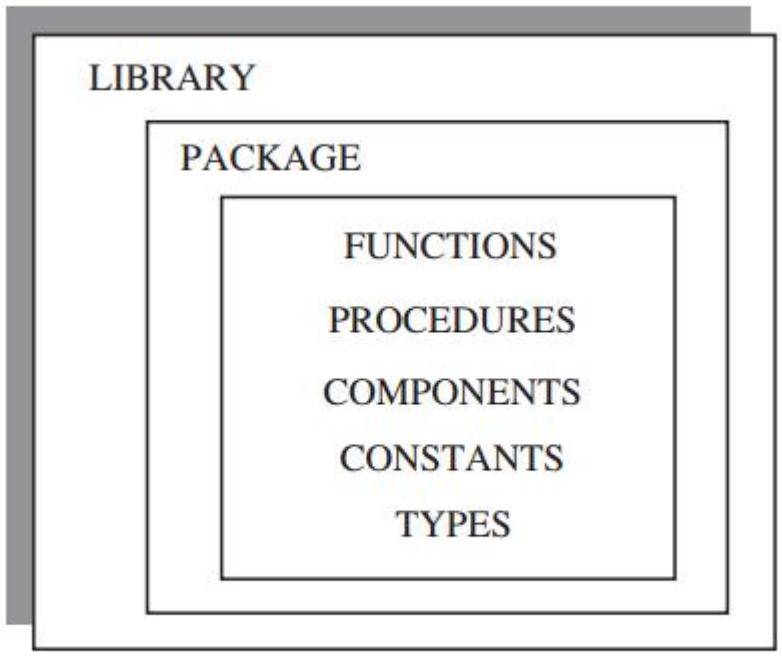


Figure 2.2
Fundamental parts of a LIBRARY.

۲.۲ اعلان‌های Library

برای اعلان یک Library (که آن را برای یک طراحی هویدا می‌کند) دو خط کد لازم است، یکی حاوی نام Library و دیگری عبارت use (کاربری)، مطابق آنچه در ترکیب زیر دیده می‌شود.

```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

معمولاً در یک طراحی حداقل سه بسته از سه کتابخانه مختلف نیاز است:

- ieee.std_logic_1164 (از کتابخانه ieee)
- standard (از کتابخانه std)
- work (از کتابخانه work)

اعلان آن‌ها مطابق زیر می‌باشد:

```
LIBRARY  ieee;                          -- A semi-colon (; ) indicates
USE ieee.std_logic_1164.all;            -- the end of a statement or
LIBRARY std;                            -- declaration, while a double
USE std.standard.all;                   --dash (--) indicates a comment.
LIBRARY work;
USE work.all;
```

کتابخانه های std و work به طور پیش فرض هویدا شده‌اند، بنابراین احتیاجی به اعلان آن‌ها نیست و تنها کتابخانه ieee باید به وضوح نوشته شود. البته مورد آخر فقط زمانی لازم است که از داده های نوع std_logic (یا std_ulogic) در طراحی استفاده شده باشد. (انواع داده به تفصیل در فصل بعد مورد مطالعه قرار خواهد گرفت).

هدف از به کارگیری سه بسته یا کتابخانه فوق به شرح زیر است :

بسته std_logic_1164 کتابخانه ieee یک سیستم منطقی چند مرحله ای را تعریف می‌کند؛ std یک کتابخانه منبع (انواع داده، i/o متنی و ...) برای محیط طراحی VHDL می‌باشد؛ و کتابخانه work مکانی است که طراحی را در آنجا ذخیره می‌کنیم (فایل vhd. به علاوه تمام فایل هایی که کمپایلر، شبیه ساز و ... ایجاد کرده‌اند).

در حقیقت کتابخانه ieee حاوی چندین بسته است، شامل:

- Std_logic_1164: سیستم های منطقی چند مرحله ای STD_Logic (۸ مرحله‌ای) و STD_Ulogic (۹ مرحله ای) را تعریف می‌کند.
- Std_logic_arith: انواع داده Signed (علامت دار) و Unsigned (بدون علامت) و ریاضیات مربوطه و عملیات مقایسه را تعریف می‌کند. همچنین حاوی چندین مبدل داده است که امکان تبدیل یک نوع داده به نوع دیگر را فراهم می نماید: conv_integer (p) و (p,b) conv_unsigned، conv_signed (p,b)، conv_std_logic_vector (p,b)
- Std_logic_signed: حاوی توابعی است که امکان کار با داده نوع std_logic_vector را فراهم می‌سازد به نحوی که داده از نوع Signed اجرا شود.
- Std_logic_unsigned: حاوی توابعی است که امکان کار با داده نوع Std_logic_vector را به فرم unsigned فراهم می‌سازد.

۳. ۲: ENTITY (نهاد)

Entity یک لیست است که پایه‌های ورودی خروجی (پورت ها) مدار را تعیین می نماید. ترکیب آن در

```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

mode سیگنال می‌تواند IN، OUT، INOUT و یا BUFFER باشد. همانند شکل ۳. ۲، IN و out پایه‌های یک طرفه هستند، در حالی که INOUT یک پایه دو طرفه است. از سوی دیگر BUFFER وقتی که باید از داخل استفاده شود (خوانده شود) به کار می رود.

Type سیگنال می‌تواند BIT، std_logic , Integer و ... باشد. جزئیات انواع داده‌ها در فصل ۳ بحث خواهد شد.

نهایتاً اسم entity اساساً می‌تواند هر اسمی غیر از کلمات رزرو شده VHDL باشد. (کلمات رزرو شده VHDL در پیوست E لیست شده است).

مثال: گیت NAND تصویر ۲.۴ را در نظر بگیرید. entity آن اینگونه تعریف می‌شود.

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

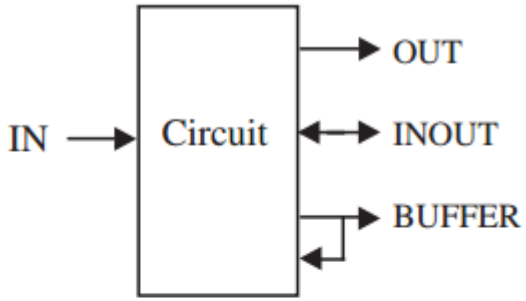


Figure 2.3
Signal *modes*.

Figure 2.4
NAND gate.

معنی Entity فوق این است: این مدار سه پایه I/O دارد، دو ورودی (a , b، مد in) و یک خروجی (x، مد out). هر سه سیگنال از نوع BIT هستند. اسم entity را nand_gate انتخاب کردیم.

Architecture ۲.۴

Architecture توصیف می‌کند که مدار چگونه باید کار کند. ترکیب آن به قرار زیر است:

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

مانند فوق، یک Architecture دو بخش دارد: یک بخش اعلان‌ها (اختیاری)، که سیگنال ها و ثابت ها در آن اعلان می‌شوند، و بخش کد (code)(از Begin تا آخر). درست مثل entity، نام Architecture نیز اساساً می‌تواند هر اسمی‌باشد، حتی شامل اسم مشابه entity.

مثال: دوباره گیت NAND شکل ۲.۴ را در نظر می گیریم.

- معنی Architecture : مدار باید عمل NAND را بین ورودی a و b اجرا کند و نتیجه را بخروجی X اطلاق نماید

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

مثال ۲.۱: DFF با reset آسنکرون (غیر همروند)

شکل ۲.۵ دیاگرام یک فلیپ فلاپ نوع D را نشان می‌دهد (DFF)، که در لبه بالارونده سیگنال ساعت (clk) و با یک ورودی Reset (rst) تحریک می‌شود. زمانی که rst=1 است خروجی باید بدون در نظر گرفتن clk صفر شود. در غیر این صورت خروجی باید ورودی را در زمانی که clk از ۰ به ۱ تغییر می‌کند کپی کند (q <= d) (این زمانی است که در clk یک لبه بالا رونده رخ می‌دهد).

چندین روش برای اجرای DFF شکل ۲.۵ وجود دارد که یکی از آن‌ها در زیر آورده شده است. به هر حال، نکته ای که باید به خاطر داشت این است که VHDL ذاتاً همروند است (برخلاف برنامه‌های کامپیوتری معمولی که ترتیبی هستند)، از این رو برای اجرای هر مدار دارای پالس ساعت (مثلاً فلیپ فلاپ ها) باید فلیپ فلاپ را اجباراً ترتیبی کنیم. این کار از طریق Process امکان پذیر است. مانند زیر:

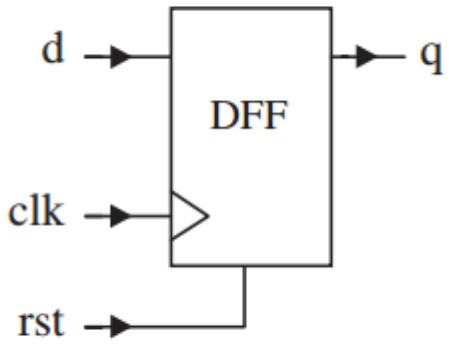


Figure 2.5
DFF with asynchronous reset.

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

توضیحات:

خط ۳-۲: اعلان‌های کتابخانه (نام کتابخانه و عبارت کاربری کتابخانه).

یادآوری کنیم که دو کتابخانه لازم دیگر (std و work) به طور پیش فرض و خودکار فراخوانی شده‌اند.

خط ۸-۵: Entity DFF

خط ۲۰-۱۰: عملکرد Architecture

خط ۶: پایه‌های ورودی (مد ورودی فقط می‌تواند IN باشد). در این مثال همه سیگنال‌های ورودی std_logic هستند.

(مد خروجی می‌تواند out، INOUT و یا BUFFER باشد). در اینجا خروجی نیز از نوع std_logic می‌باشد.

خط ۱۹-۱۱: قسمت کد Architecture (از کلمه Begin شروع می‌شود).

خط‌های ۱۹-۱۲: یک process است. (در آن کد به صورت ترتیبی اجرا می‌شود).

خط ۱۲: هرگاه یکی از سیگنال‌هایی که در لیست حساسیت قرار دارد تغییر کند process اجرا می‌شود.

در این مثال، هرگاه rst یا clk تغییر می‌کند process اجرا می‌شود.

خط‌های ۱۵-۱۴: هرگاه rst یک (۱) می‌شود خروجی بدون توجه به clk باز نشانی (Reset) می‌شود.

(reset آسنکرون).

خطهای ۱۷-۱۶: اگر rst فعال نباشد، به علاوه clk تغییر کند (یک EVENT در clk رخ دهد)، به علاوه event یک لبه بالا رونده باشد (clk = ۱) آنگاه سیگنال ورودی (d) در فلیپ فلاپ ذخیره می‌شود (q <= d).

خطهای ۱۵ و ۱۷: عملگر " <= " برای اطلاق یک مقدار به یک سیگنال استفاده می‌شود. در مقابل " := " برای یک متغیر استفاده می‌شود. همه پایه ها به طور پیش فرض، سیگنال هستند.

خطهای ۱، ۴، ۹ و ۲۱: توضیح می‌دهد (یادآور می‌شویم که "--" اعلانگر توضیح است) فقط برای سازماندهی بهتر طراحی می‌باشد.

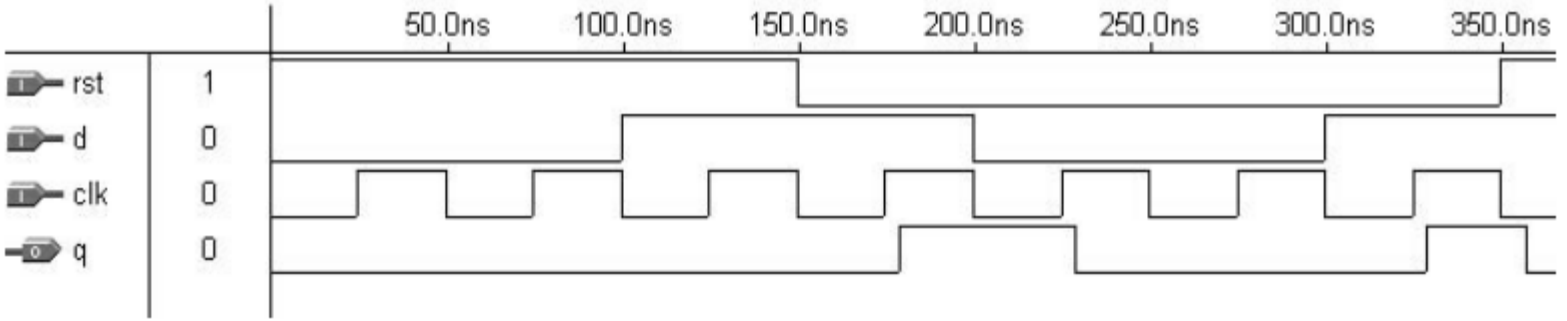


Figure 2.6
Simulation results of example 2.1.

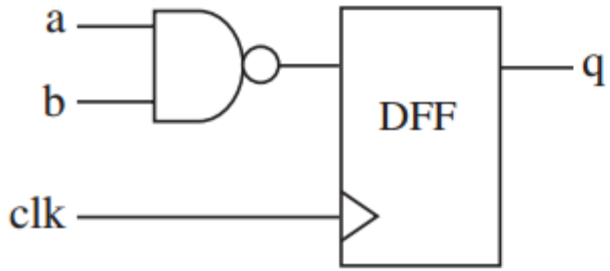


Figure 2.7
DFF plus NAND gate.

توجه کنید که پیکان‌هایی که به rst، d و clk مربوطند به سمت داخل هستند و در داخل حاوی حرف I (ورودی) هستند در حالی که q به سمت خارج است و در داخل حاوی حرف o (خروجی) است. ستون دوم مقدار هر سیگنالی را در مکانی که نشانگر عمودی قرار دارد نشان می‌دهد. در این مورد، نشانگر در 0ns است که سیگنال به ترتیب ۱، ۰، ۰، ۰ است. در این مثال، مقادیر خیلی ساده ۰ یا ۱ هستند، اما وقتی از بردارها استفاده می‌شود مقادیر به فرم باینری، دهدهی یا شانزده شانزدهی نشان داده می‌شوند. ستون دوم شبیه سازی را نشان می‌دهد. سیگنال‌های ورودی (rst، d و clk) بدون محدودیت انتخاب می‌شوند و شبیه ساز، خروجی (q) مناسب را تعیین می‌کند.

تمرین های آخر فصل ۲

تحویل دو هفته اینده

فصل سوم: انواع داد ها

VHDL حاوی یک سری از انواع داده‌های از پیش تعیین شده است که از طریق استانداردهای IEEE 1076 و IEEE 1164 قابل دسترسی هستند. تعاریف این نوع داده‌ها در بسته‌ها / کتابخانه‌های زیر به طور دقیق تر و تخصصی‌تر دیده می‌شود.

- بسته standard کتابخانه std داده‌های نوع BIT، BOOLEAN، INTEGER و REAL را تعریف می‌کند.
- بسته std_logic_1164 کتابخانه ieee داده‌های نوع Std_Logic و Std_Ulogic را تعریف می‌کند.

- بسته std_logic_arith کتابخانه ieee داده‌های نوع signed و unsigned به علاوه چندین تابع تبدیل داده مثل conv_Integer (p)، conv_unsigned (p,b)، conv_signed (p,b) و conv_std_logic_vector (p,b) را تعریف می‌کند.
- بسته‌های std_logic_signed و std_logic_unsigned کتابخانه ieee حاوی توابعی هستند که امکان انجام عملیات با داده نوع std_logic_vector به ترتیب از نوع signed و unsigned را دارند.

- BIT (and BIT_VECTOR): 2-level logic (‘0’, ‘1’).

Examples:

```
SIGNAL x: BIT;
-- x is declared as a one-digit signal of type BIT.

SIGNAL y: BIT_VECTOR (3 DOWNTO 0);
-- y is a 4-bit vector, with the leftmost bit being the MSB.

SIGNAL w: BIT_VECTOR (0 TO 7);
-- w is an 8-bit vector, with the rightmost bit being the MSB.
```

بر اساس سیگنال‌های فوق، مقداردهی‌های زیر مجاز هستند. (برای اطلاق یک مقدار به یک سیگنال از عملگر "= >" استفاده می‌شود).

x <= ' ۱ ';

X یک سیگنال تک سطحی است (مانند فوق) که مقدار آن " ۱ " است. دقت کنید که مقداردهی تکی (' ') برای یک تک بیت به کار می‌رود.

y <= " ۰۱۱۱ ";

Y یک سیگنال ۴ بیتی است (مانند فوق) که مقدار آن " ۰۱۱۱ " است (MSB = ۰). دقت کنید که کتیشن (" ") برای بردار به کار می‌رود.

```
w <= "01110001";
-- w is an 8-bit signal, whose value is "01110001" (MSB='1').
```

• Std_Logic (و std_logic_vector): سیستم منطقی ۸ مقداره که در استاندارد IEEE 1164 معرفی شده است.

X نامعلوم قوی (نامعلوم قابل سنتز)	W نامعلوم ضعیف
0 پایین قوی (0 منطقی قابل سنتز)	L پایین ضعیف
1 بالای قوی (1 منطقی قابل سنتز)	H بالای ضعیف
Z امپدانس بالا (بافر سه حالته قابل سنتز)	'–' توجه نکنید.

مثال:

```
Signal x = std_logic;
```

x به عنوان یک سیگنال تک مقداره (اسکالر) از نوع std_logic تعریف می‌شود.

```
Signal y: Std_Logic_Vector (3 to 0) := " 0001 ";
```

Y به صورت یک بردار ۴ بیتی که بیت سمت چپ آن MSB است تعریف می‌شود. مقدار اولیه y (اختیاری) " ۰۰۰۱ " است. توجه کنید که عملگر " := " برای مقداردهی اولیه استفاده می‌شود.

بیشتر سطوح std_logic فقط برای شبیه‌سازی استفاده می‌شوند. با این حال " ۰ "، " ۱ " و " z " بدون هیچ محدودیتی قابل سنتز هستند.

Table 3.1
Resolved logic system (STD_LOGIC).

	X	0	1	Z	W	L	H	–
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
–	X	X	X	X	X	X	X	X

Std_ulogic (و std_ulogic_vector): سیستم منطقی ۹ مقداره که در استاندارد IEEE 1164 معرفی شده است (_، L، H، W، Z، 0، 1، X، U). در حقیقت سیستم std_logic زیر مجموعه‌ای از std_ulogic است.

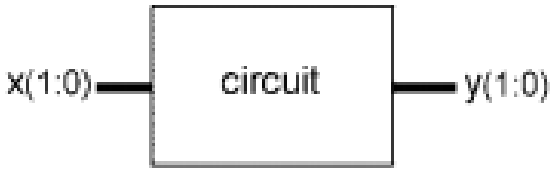
مورد دوم شامل یک مقدار منطقی اضافی، " u " به معنی تصمیم‌گیری نشده می‌باشد. بنابراین، برخلاف std_logic، در اینجا تضاد سطوح منطقی به طور خودکار برطرف نمی‌شود، از این رو سیم‌های خروجی در پیاده‌سازی فیزیکی هرگز نباید مستقیماً به هم وصل شوند. به این ترتیب، با فرض اینکه سیم‌های خروجی هرگز به هم وصل نمی‌شوند، می‌توان از چنین سیستمی برای ردیابی خطاهای طراحی استفاده نمود.

- BOOLEAN: True, False.
- INTEGER: 32-bit integers (from −2,147,483,647 to +2,147,483,647).
- NATURAL: Non-negative integers (from 0 to +2,147,483,647).
- REAL: Real numbers ranging from −1.0E38 to +1.0E38. Not synthesizable.
- Physical literals: Used to inform physical quantities, like time, voltage, etc. Useful in simulations. Not synthesizable.
- Character literals: Single ASCII character or a string of such characters. Not synthesizable.

لیترال‌های فیزیکی: برای اطلاع‌رسانی مقادیر فیزیکی مثل زمان، ولتاژ و ... به کار می‌رود. در شبیه‌سازی‌ها کارایی دارند اما قابل سنتز نیستند.

لیترال‌های کاراکتر: کاراکتر ASCII تکی یا یک رشته از این کاراکترها، غیر قابل سنتزاند.

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



(a)

(b)

x_1	x_0	y_1	y_0
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	0

(c)

x_1	x_0	y_1	y_0
0	0	0	0
0	1	1	0
1	0	0	1
1	1	–	–

Figure 3.4
Circuit with "don't care" outputs of example 3.2.

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

Signed و Unsigned: انواع داده‌هایی که در بسته std_logic_arith کتابخانه ieee تعریف شده است. ظاهر آن‌ها مثل std_logic_vector است با این تفاوت که عملیات ریاضی نیز بر روی آنها قابل اجرا است که نمونه‌هایی از انواع داده Integer هستند.

Examples:

```
x0 <= '0';              -- bit, std_logic, or std_ulogic value '0'
x1 <= "00011111";       -- bit_vector, std_logic_vector,
                        -- std_ulogic_vector, signed, or unsigned
x2 <= "0001_1111";      -- underscore allowed to ease visualization
x3 <= "101111"          -- binary representation of decimal 47
```

```
x4 <= B"101111"          -- binary representation of decimal 47
x5 <= O"57"              -- octal representation of decimal 47
x6 <= X"2F"              -- hexadecimal representation of decimal 47
n <= 1200;               -- integer
m <= 1_200;              -- integer, underscore allowed
IF ready THEN...         -- Boolean, executed if ready=TRUE
y <= 1.2E-5;             -- real, not synthesizable
q <= d after 10 ns;      -- physical, not synthesizable
```

عملیات های مجاز و غیر مجاز

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
```

```
a <= b(5);     -- legal (same scalar type: BIT)
b(0) <= a;     -- legal (same scalar type: BIT)
c <= d(5);     -- legal (same scalar type: STD_LOGIC)
d(0) <= c;     -- legal (same scalar type: STD_LOGIC)
a <= c;        -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d;        -- illegal (type mismatch: BIT_VECTOR x
               -- STD_LOGIC_VECTOR)
```

```
e <= b;          -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d;          -- illegal (type mismatch: INTEGER x
                 -- STD_LOGIC_VECTOR)
```

3.2 User-Defined Data Types

VHDL also allows the user to define his/her own data types. Two categories of user-defined data types are shown below: *integer* and *enumerated*.

- User-defined *integer* types:

```
TYPE integer IS RANGE -2147483647 TO +2147483647;
-- This is indeed the pre-defined type INTEGER.

TYPE natural IS RANGE 0 TO +2147483647;
-- This is indeed the pre-defined type NATURAL.
```

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

```
TYPE my_integer IS RANGE -32 TO 32;
-- A user-defined subset of integers.

TYPE student_grade IS RANGE 0 TO 100;
-- A user-defined subset of integers or naturals.
```

- User-defined *enumerated* types:

```
TYPE bit IS ('0', '1');
-- This is indeed the pre-defined type BIT

TYPE my_logic IS ('0', '1', 'Z');
-- A user-defined subset of std_logic.
```

```
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
-- This is indeed the pre-defined type BIT_VECTOR.
-- RANGE <> is used to indicate that the range is unconstrained.
-- NATURAL RANGE <>, on the other hand, indicates that the only
-- restriction is that the range must fall within the NATURAL
-- range.
```

```
TYPE state IS (idle, forward, backward, stop);
-- An enumerated data type, typical of finite state machines.

TYPE color IS (red, green, blue, white);
-- Another enumerated data type.
```

- کد گزاری انواع enumerated به صورت ترتیبی می باشد مگر اینکه کاربر خود از قبل مشخص کند.

example, for the type *color* above, two bits are necessary (there are four states), being "00" assigned to the first state (red), "01" to the second (green), "10" to the next (blue), and finally "11" to the last state (white).

subtype یا زیر نوع ها

A SUBTYPE is a TYPE with a constraint. The main reason for using a subtype rather than specifying a new type is that, though operations between data of different types are not allowed, they are allowed between a subtype and its corresponding base type.

```
SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- As expected, NATURAL is a subtype (subset) of INTEGER.
```

```
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Recall that STD_LOGIC=('X','0','1','Z','W','L','H','-').
-- Therefore, my_logic=('0','1','Z').
```

```
SUBTYPE my_color IS color RANGE red TO blue;
-- Since color=(red, green, blue, white), then
-- my_color=(red, green, blue).
```

```
SUBTYPE small_integer IS INTEGER RANGE -32 TO 32;
-- A subtype of INTEGER.
```

Example: Legal and illegal operations between types and subtypes.

```
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO '1';
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: my_logic;
...
b <= a;     -- illegal (type mismatch: BIT versus STD_LOGIC)
b <= c;     -- legal (same "base" type: STD_LOGIC)
```

3.4 Arrays

آرایه‌ها مجموعه‌ای از عوامل مشابه یک نوع هستند. می‌توانند یک بعدی (1D)، دو بعدی (2D) و یا یک بعد در یک بعدی (1D × 1D) باشند.

همچنین می‌توانند دارای ابعاد بالاتری نیز باشند اما آنگاه دیگر قابل سنتز نیستند.

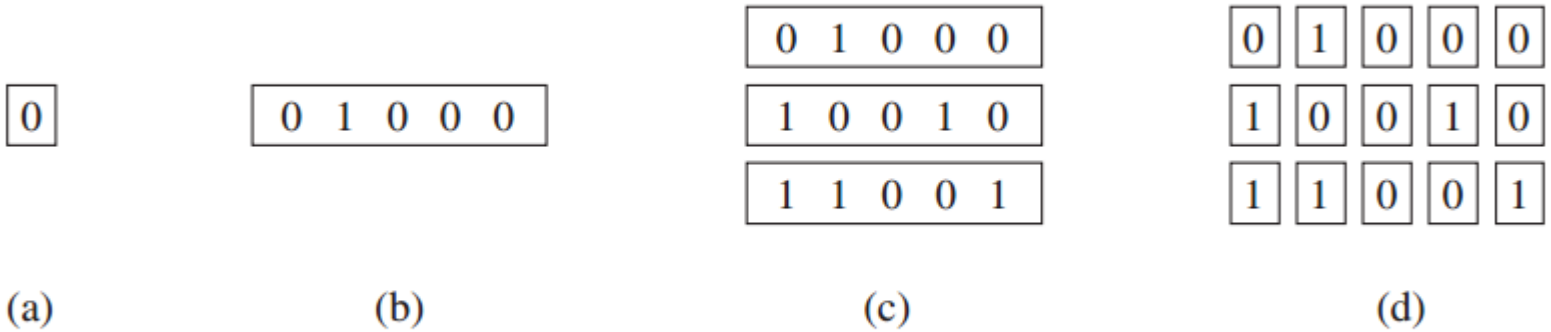


Figure 3.1
Illustration of (a) scalar, (b) 1D, (c) 1Dx1D, and (d) 2D data arrays.

در حقیقت، انواع داده از پیش تعریف شده VHDL فقط شامل اسکالر (تک بیت) و بردار (آرایه یک بعدی بیت‌ها) می‌باشد. انواع قابل سنتز از پیش تعریف شده در هر کدام از این دسته‌ها عبارتند از:

- Scalars: BIT, STD_LOGIC, STD_ULOGIC, and BOOLEAN.
- Vectors: BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, INTEGER, SIGNED, and UNSIGNED.

As can be seen, there are no pre-defined 2D or 1Dx1D arrays, which, when necessary, must be specified by the user. To do so, the new TYPE must first be defined, then the new SIGNAL, VARIABLE, or CONSTANT can be declared using that data type. The syntax below should be used.

To specify a new array type:

```
TYPE type_name IS ARRAY (specification) OF data_type;
```

To make use of the new array type:

```
SIGNAL signal_name: type_name [:= initial_value];
```

- در ترکیب فوق یک signsl اعلان شد. می توانست یک constant یا variable نیز باشد!!

Example: 1Dx1D array.
Say that we want to build an array containing four vectors, each of size eight bits. This is then an 1Dx1D array (see figure 3.1). Let us call each vector by *row*, and the complete array by *matrix*. Additionally, say that we want the leftmost bit of each vector to be its MSB (most significant bit),

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;     -- 1D array
TYPE matrix IS ARRAY (0 TO 3) OF row;            -- 1Dx1D array
SIGNAL x: matrix;                                -- 1Dx1D signal
```

Example: Another 1Dx1D array.

Another way of constructing the 1Dx1D array above would be the following:

```
TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Example: 2D array.

The array below is truly two-dimensional. Notice that its construction is not based on vectors, but rather entirely on scalars.

```
TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
-- 2D array
```

Example: Array initialization.
As shown in the syntax above, the initial value of a SIGNAL or VARIABLE is optional. However, when initialization is required, it can be done as in the examples below.

```
... :="0001";                                          -- for 1D array
... :=('0','0','0','1')                                -- for 1D array
... :=(('0','1','1','1'), ('1','1','1','0'));         -- for 1Dx1D or
                                                       -- 2D array
```

Example: Legal and illegal array assignments.
The assignments in this example are based on the following type definitions and signal declarations:

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
                                                    -- 1D array
TYPE array1 IS ARRAY (0 TO 3) OF row;
                                                    -- 1Dx1D array
TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
                                                    -- 1Dx1D
TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
                                                    -- 2D array

SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;
```

```
--------- Legal scalar assignments: ---------------
-- The scalar (single bit) assignments below are all legal,
-- because the "base" (scalar) type is STD_LOGIC for all signals
-- (x,y,v,w).
x(0) <= y(1)(2);        -- notice two pairs of parenthesis
                        -- (y is 1Dx1D)
x(1) <= v(2)(3);        -- two pairs of parenthesis (v is 1Dx1D)
x(2) <= w(2,1);         -- a single pair of parenthesis (w is 2D)
y(1)(1) <= x(6);
y(2)(0) <= v(0)(0);
y(0)(0) <= w(3,3);
w(1,1) <= x(7);
w(3,0) <= v(0)(3);
```

```
--------- Vector assignments: ---------------------
x <= y(0);                      -- legal (same data types: ROW)
x <= v(1);                      -- illegal (type mismatch: ROW x
                                -- STD_LOGIC_VECTOR)
x <= w(2);                      -- illegal (w must have 2D index)
x <= w(2, 2 DOWNTO 0);          -- illegal (type mismatch: ROW x
                                -- STD_LOGIC)
v(0) <= w(2, 2 DOWNTO 0);       -- illegal (mismatch: STD_LOGIC_VECTOR
                                -- x STD_LOGIC)
v(0) <= w(2);                   -- illegal (w must have 2D index)
y(1) <= v(3);                   -- illegal (type mismatch: ROW x
                                -- STD_LOGIC_VECTOR)
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0);      -- legal (same type,
                                        -- same size)
v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO 0);   -- legal (same type,
                                        -- same size)
w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO 0);   -- illegal (type mismatch)
```

3.5 Port Array

we might need to specify the ports as arrays of vectors.

- در موجودیت نمیتوان چنین تعریفی را ارائه داد.type

راه حل pakage

```
1   -----Package:-----------------------------------------
2   PACKAGE my_data_types IS
3      TYPE oneDoneD IS ARRAY (0 TO 3) OF BIT_VECTOR(7 DOWNTO 0);
4   END my_data_types;
5   -----------------------------------------------------

1   -----Main code: --------------------
2   USE work.my_data_types.all;
3   -----------------------------------------------------
4   ENTITY mux IS
5   PORT (x: IN oneDoneD;
6         sel: INTEGER RANGE 0 TO 3;
7         y: OUT BIT_VECTOR(7 DOWNTO 0));
8   END ENTITY;
9   -----------------------------------------------------
10  ARCHITECTURE mux OF mux IS
11  BEGIN
12     y <= x(sel);
13  END ARCHITECTURE;
14  -----------------------------------------------------
```

3.6 Records

Records are similar to arrays, with the only difference that they contain objects of *different* types.

Example:

```
TYPE birthday IS RECORD
   day: INTEGER RANGE 1 TO 31;
   month: month_name;
END RECORD;
```

3.7 Signed and Unsigned Data Types

defined in the *std_logic_arith* package of the *ieee* library.

Examples:

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

is similar to that of STD_LOGIC_VECTOR,

3.8 Data Conversion

If the data are *closely* related (that is, both operands have the same *base* type, despite being declared as belonging to two different type classes), then the *std_logic_1164* of the *ieee* library

Example: Legal and illegal operations with subsets.

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x : short;
SIGNAL y : long;
...
y <= 2*x + 5;           -- error, type mismatch
y <= long(2*x + 5);     -- OK, result converted into type long
```

Several data conversion functions can be found in the *std_logic_arith* package of the *ieee* library.

- conv_integer(p) : Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER value. Notice that STD_LOGIC_VECTOR is not included.

- conv_unsigned(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED value with size b bits.

- conv_signed(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to a SIGNED value with size b bits.

- conv_std_logic_vector(p, b): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_LOGIC to a STD_LOGIC_VECTOR value with size b bits.

Example 3.3: Adder

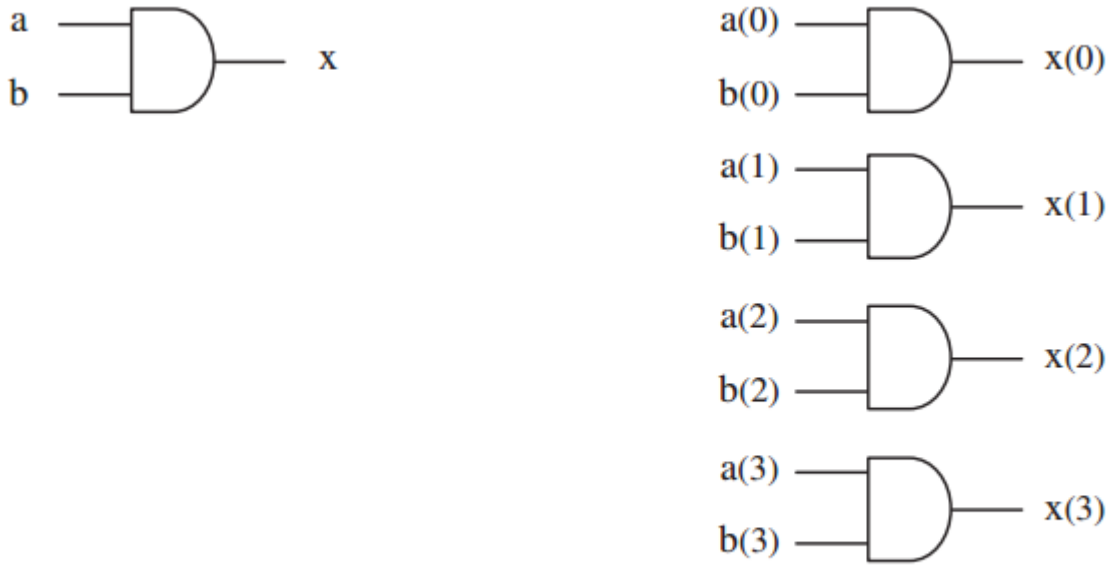


Figure 3.2
Circuits inferred from the codes of example 3.2.

```
1  ----- Solution 1: in/out=SIGNED ----------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;
5  ---------------------------------------
6  ENTITY adder1 IS
7     PORT ( a, b : IN SIGNED (3 DOWNTO 0);
8            sum : OUT SIGNED (4 DOWNTO 0));
9  END adder1;
10 ---------------------------------------
11 ARCHITECTURE adder1 OF adder1 IS
12 BEGIN
13    sum <= a + b;
14 END adder1;
15 ---------------------------------------
```

```
1  ------ Solution 2: out=INTEGER ----------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all;
5  ---------------------------------------
6  ENTITY adder2 IS
7     PORT ( a, b : IN SIGNED (3 DOWNTO 0);
8            sum : OUT INTEGER RANGE -16 TO 15);
9  END adder2;
10 ---------------------------------------
11 ARCHITECTURE adder2 OF adder2 IS
12 BEGIN
13    sum <= CONV_INTEGER(a + b);
14 END adder2;
15 ---------------------------------------
```

تمرین

- مسئله های ۲،۳،٤ و ۵ فصل سوم
- میانترم ۳۰ آبان ۵ نمره

فصل ۴: عملگر ها و قید ها

- عملگرهای مقداردهی
- عملگرهای منطقی
- عملگرهای محاسباتی
- عملگرهای رابطه ای
- عملگرهای انتقال
- عملگرهای الحاقی

Assignment Operators

<= Used to assign a value to a SIGNAL.

:= Used to assign a value to a VARIABLE, CONSTANT, or GENERIC. Used also for establishing initial values.

=> Used to assign values to individual vector elements or with OTHERS.

```
x <= '1';          -- '1' is assigned to SIGNAL x using "<="
y := "0000";       -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000";                    -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0');       -- LSB is '1', the others are '0'
```

Logical Operators

Used to perform logical operations. The data must be of type BIT, STD_LOGIC, or STD_ULOGIC (or, obviously, their respective extensions, BIT_VECTOR, STD_LOGIC_VECTOR, or STD_ULOGIC_VECTOR). The logical operators are:

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

Examples:

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

Arithmetic Operators

INTEGER,

SIGNED, UNSIGNED, or REAL

if the *std_logic_signed* or the *std_logic_unsigned* package of the *ieee*

then STD_LOGIC_VECTOR can also be employed directly

+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation
MOD	Modulus
REM	Remainder
ABS	Absolute value

- ضرب
- تقسیم
- به توان رساندن
- ضریب(خارج قسمت)
- باقیمانده
- قدر مطلق

Comparison Operators

=	Equal to
/=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Shift Operators

- sll Shift left logic – positions on the right are filled with ‘0’s
- srl Shift right logic – positions on the left are filled with ‘0’s

- Shift left arithmetic (SLA): Rightmost bit is replicated on the right.
- Shift right arithmetic (SRA): Leftmost bit is replicated on the left.

x is a BIT_VECTOR signal with value x = "01001".

```
y <= x SLL 2;    --y<="00100" (y <= x(2 DOWNTO 0) & "00";)
y <= x SLA 2;    --y<="00111" (y <= x(2 DOWNTO 0) & x(0) & x(0);)
```

```
-------------------------------------------------------------------
CONSTANT v: BIT :='1';
CONSTANT x: STD_LOGIC :='Z';
SIGNAL y: BIT_VECTOR(1 TO 4);
SIGNAL z: STD_LOGIC_VECTOR(7 DOWNTO 0);
y <= (v & "000");              --result: "1000"
y <= v & "000";                --same as above (parentheses are optional)
z <= (x & x & "11111" & x);    --result: "ZZ11111Z"
z <= ('0' & "011111" & x);     --result: "00111111Z"
-------------------------------------------------------------------
```

Data Attributes

- d'LOW: Returns lower array index
- d'HIGH: Returns upper array index
- d'LEFT: Returns leftmost array index
- d'RIGHT: Returns rightmost array index
- d'LENGTH: Returns vector size
- d'RANGE: Returns vector range
- d'REVERSE_RANGE: Returns vector range in reverse order

Example

```
SIGNAL d : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Then:

```
d'LOW=0, d'HIGH=7, d'LEFT=7, d'RIGHT=0, d'LENGTH=8,
d'RANGE=(7 downto 0), d'REVERSE_RANGE=(0 to 7).
```

If the signal is of *enumerated* type, then:

- d'VAL(pos): Returns value in the position specified
- d'POS(value): Returns position of the value specified
- d'LEFTOF(value): Returns value in the position to the left of the value specified
- d'VAL(row, column): Returns value in the position specified; etc.

صفت های سیگنال

- s'EVENT: Returns true when an event occurs on s
- s'STABLE: Returns true if no event has occurred on s
- s'ACTIVE: Returns true if s = '1'
- s'QUIET ⟨time⟩: Returns true if no event has occurred during the time specified
- s'LAST_EVENT: Returns the time elapsed since last event
- s'LAST_ACTIVE: Returns the time elapsed since last s = '1'
- s'LAST_VALUE: Returns the value of s before the last event; etc.

```
IF (clk'EVENT AND clk='1')...          -- EVENT attribute used
                                       -- with IF
IF (NOT clk'STABLE AND clk='1')...     -- STABLE attribute used
                                       -- with IF
WAIT UNTIL (clk'EVENT AND clk='1');    -- EVENT attribute used
                                       -- with WAIT
```

4.3 User-Defined Attributes

```
ATTRIBUTE attribute_name: attribute_type;
```

```
ATTRIBUTE attribute_name OF target_name: class IS value;
```

attribute_type: any data type (BIT, INTEGER, STD_LOGIC_VECTOR, etc.)

class: TYPE, SIGNAL, FUNCTION, etc.

value: '0', 27, "00 11 10 01", etc.

Example: Enumerated encoding.

enum_encoding attribute.

```
TYPE color IS (red, green, blue, white);
```

red = "00", green = "01", blue = "10", and white = "11".

```
ATTRIBUTE enum_encoding OF color: TYPE IS "11 00 10 01";
```

```
-------------------------------------------------------------------
SIGNAL number_of_pins: POSITIVE;            --signal declaration
ATTRIBUTE pins: POSITIVE;                   --attribute declaration
ATTRIBUTE pins OF nand3: COMPONENT IS 4;    --attribute specification
number_of_pins <= nand3'pins;               --attribute call (tick needed)
-------------------------------------------------------------------
```

```
FUNCTION function_name [<parameter list>] RETURN data_type IS
    [declarations]
BEGIN
    (sequential statements)
END function_name;
```

4.4 Operator Overloading

```
-------------------------------------
FUNCTION "+" (a: INTEGER, b: BIT) RETURN INTEGER IS
BEGIN
   IF (b='1') THEN RETURN a+1;
   ELSE RETURN a;
   END IF;
END "+";
-------------------------------------
```

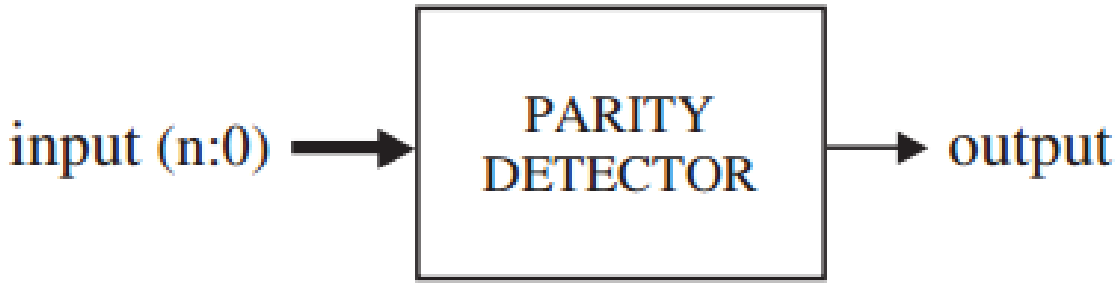
```
-------------------------------
SIGNAL inp1, outp: INTEGER RANGE 0 TO 15;
SIGNAL inp2: BIT;
    (...)
outp <= 3 + inp1 + inp2;
    (...)
-------------------------------
```

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

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

```
GENERIC (n: INTEGER := 8; vector: BIT_VECTOR := "00001111");
```

Example 4.2: Generic Parity Detector



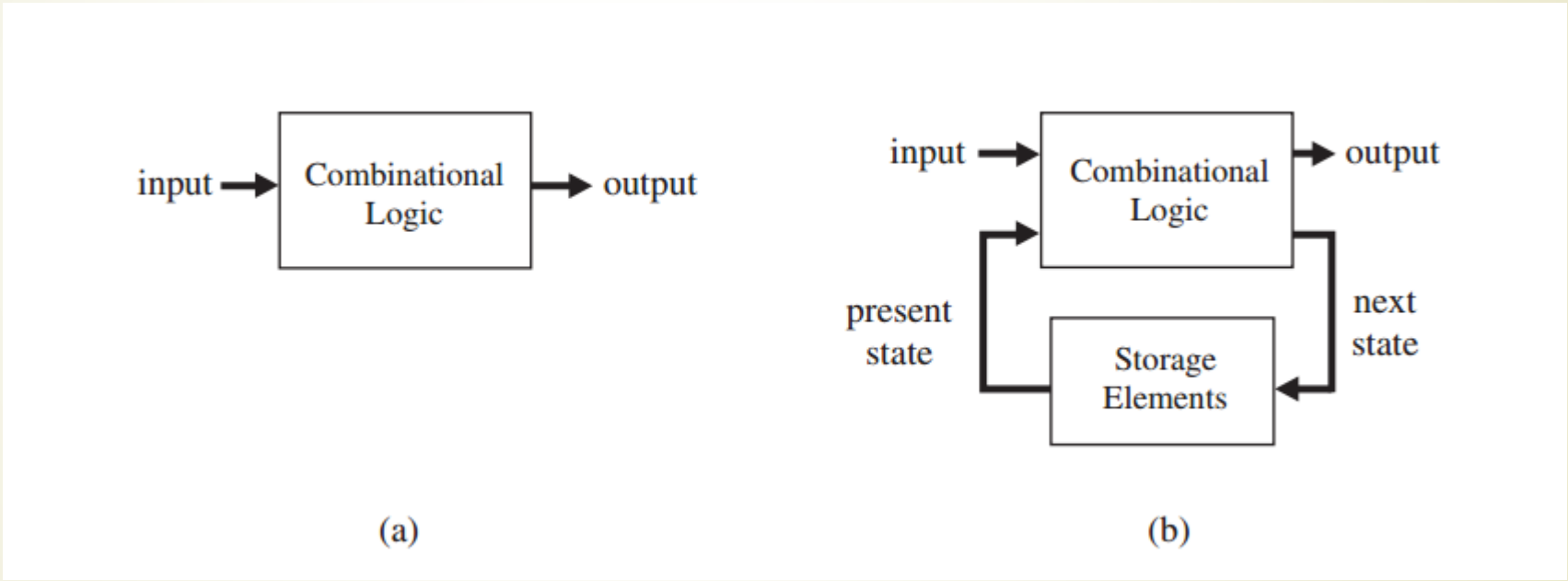
تعداد یک های بردار ورودی زوج باشد
خروجی ۰ وگر نه خروجی۱

```
2  ENTITY parity_det IS
3     GENERIC (n : INTEGER := 7);
4     PORT ( input: IN BIT_VECTOR (n DOWNTO 0);
5            output: OUT BIT);
6  END parity_det;
7  ----------------------------------------
8  ARCHITECTURE parity OF parity_det IS
9  BEGIN
10    PROCESS (input)
11       VARIABLE temp: BIT;
12    BEGIN
13       temp := '0';
14       FOR i IN input'RANGE LOOP
15          temp := temp XOR input(i);
16       END LOOP;
17       output <= temp;
18    END PROCESS;
19 END parity;
20 ----------------------------------------
```

تمرین فصل ۴

تمرینهای ۱و ۲

5 Concurrent Code



WHEN and GENERATE.

- کد همروند در برابر کد ترتیبی
- Process
- Function
- Procedure

- مدارات پیچیده معمولا با کد ترتیبی راحت تر نوشته می شوند

Table 5.1
Operators.

Operator type	Operators	Data types
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, –, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (,,,)	Same as for logical operators, plus SIGNED and UNSIGNED

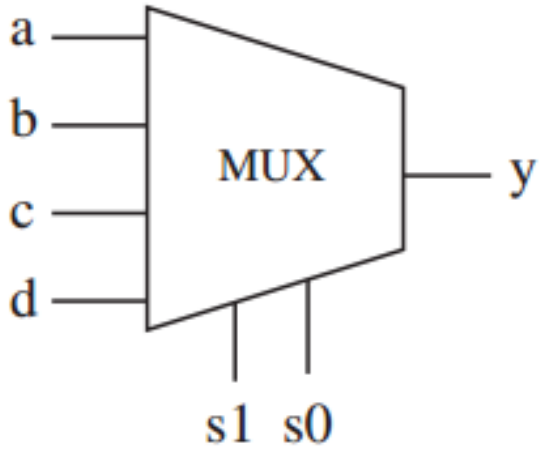


Figure 5.3
Multiplexer of example 5.1.

Example 5.1: Multiplexer #1

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
7            y: OUT STD_LOGIC);
8  END mux;
9  ----------------------------------------
10 ARCHITECTURE pure_logic OF mux IS
11 BEGIN
12    y <=  (a AND NOT s1 AND NOT s0) OR
13          (b AND NOT s1 AND s0) OR
14          (c AND s1 AND NOT s0) OR
15          (d AND s1 AND s0);
16 END pure_logic;
17 ----------------------------------------
```

5.3 WHEN (Simple and Selected)

WHEN / ELSE:

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH / SELECT / WHEN:

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

Whenever WITH / SELECT / WHEN is used, all permutations must be tested, so the keyword OTHERS is often useful. Another important keyword is UNAFFECTED, which should be used when no action is to take place.

Example:

```
------ With WHEN/ELSE ------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
        "001" WHEN ctl='1' ELSE
        "010";

---- With WITH/SELECT/WHEN --------------------
WITH control SELECT
   output <= "000" WHEN reset,
             "111" WHEN set,
             UNAFFECTED WHEN OTHERS;
-----------------------------------------------
```

```
WHEN value                      -- single value
WHEN value1 to value2           -- range, for enumerated data types
                                -- only
WHEN value1 | value2 |...       -- value1 or value2 or ...
```

Example 5.2: Multiplexer #2

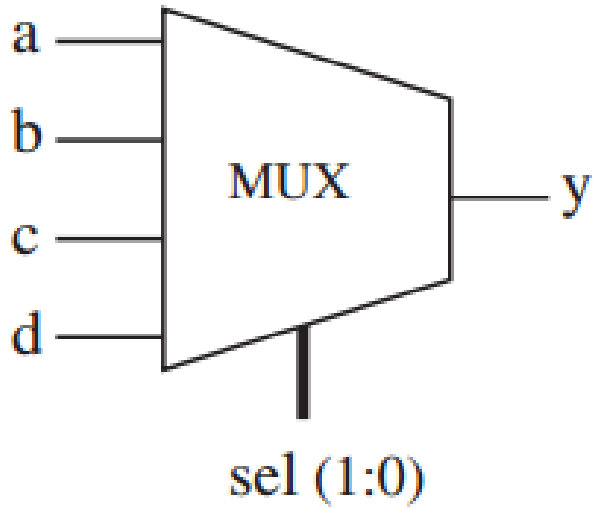


Figure 5.5
Multiplexer of example 5.2.

```
1  ------- Solution 1: with WHEN/ELSE --------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ----------------------------------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13    y <=  a WHEN sel="00" ELSE
14          b WHEN sel="01" ELSE
15          c WHEN sel="10" ELSE
16          d;
17 END mux1;
18 ----------------------------------------
```

```
1  --- Solution 2: with WITH/SELECT/WHEN -----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
8            y: OUT STD_LOGIC);
9  END mux;
10 ----------------------------------------
11 ARCHITECTURE mux2 OF mux IS
12 BEGIN
13    WITH sel SELECT
14       y <=  a WHEN "00",      -- notice "," instead of ";"
15             b WHEN "01",
16             c WHEN "10",
17             d WHEN OTHERS;    -- cannot be "d WHEN "11" "
18 END mux2;
19 ----------------------------------------
```

```
1  ------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------
5  ENTITY mux IS
6     PORT ( a, b, c, d: IN STD_LOGIC;
7            sel: IN INTEGER RANGE 0 TO 3;
8            y: OUT STD_LOGIC);
9  END mux;
10 ---- Solution 1: with WHEN/ELSE --------------
11 ARCHITECTURE mux1 OF mux IS
12 BEGIN
13    y <=  a WHEN sel=0 ELSE
14          b WHEN sel=1 ELSE
15          c WHEN sel=2 ELSE
16          d;
17 END mux1;
18 -- Solution 2: with WITH/SELECT/WHEN  --------
19 ARCHITECTURE mux2 OF mux IS
20 BEGIN
21    WITH sel SELECT
22       y <=  a WHEN 0,
23             b WHEN 1,
24             c WHEN 2,
25             d WHEN 3;    -- here, 3 or OTHERS are equivalent,
26 END mux2;                -- for all options are tested anyway
27 ------------------------------------------
```

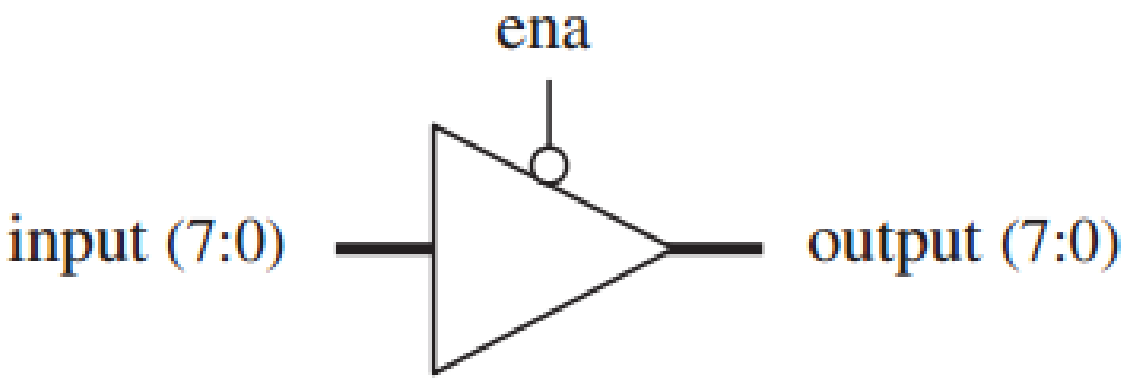


Figure 5.6
Tri-state buffer of example 5.3.

Example 5.3: Tri-state Buffer

This is another example that illustrates the use of WHEN. The 3-state buffer of figure 5.6 must provide output = input when ena (enable) is low, or output = "*ZZZZZZZZ*" (high impedance) otherwise.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ---------------------------------------
4  ENTITY tri_state IS
5     PORT ( ena: IN STD_LOGIC;
6            input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            output: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
8  END tri_state;
9  ---------------------------------------
10 ARCHITECTURE tri_state OF tri_state IS
11 BEGIN
12    output <= input WHEN (ena='0') ELSE
13              (OTHERS => 'Z');
14 END tri_state;
15 ---------------------------------------
```

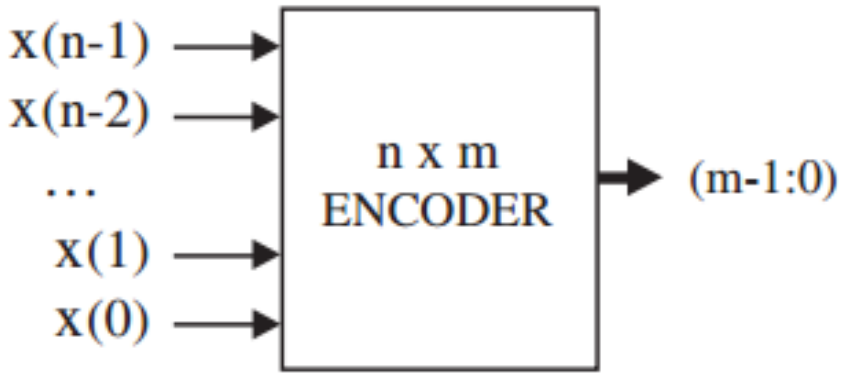
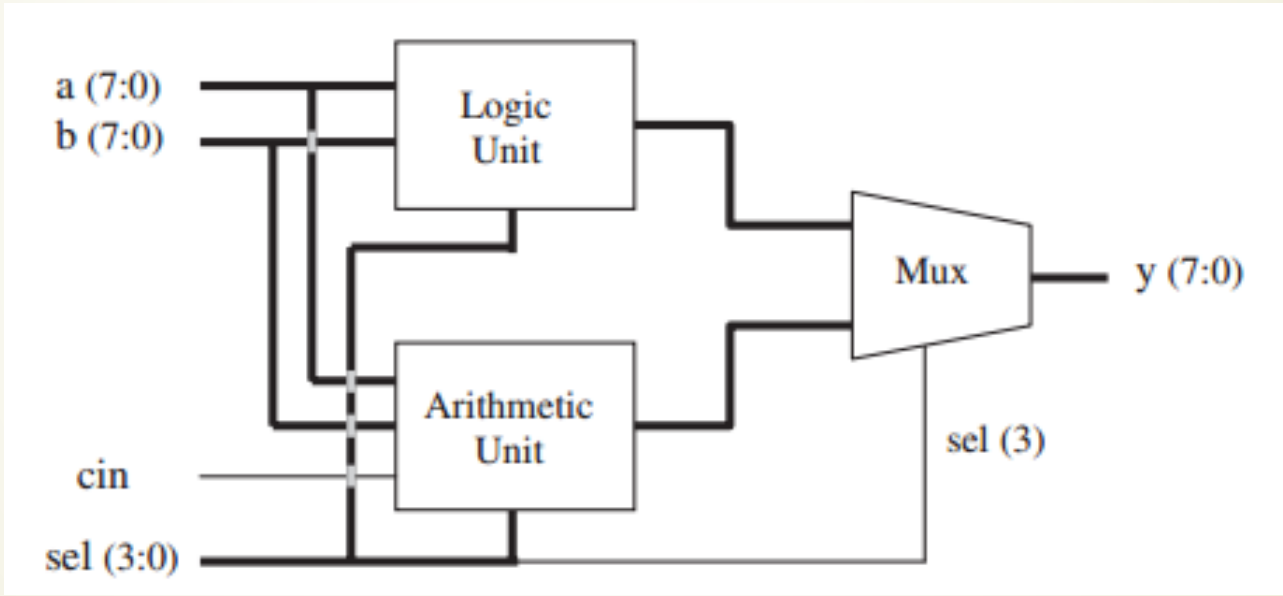


Figure 5.8
Encoder of example 5.4.

```
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY encoder IS
6     PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8  END encoder;
9  ---------------------------------------
10 ARCHITECTURE encoder1 OF encoder IS
11 BEGIN
12    y <=   "000" WHEN x="00000001" ELSE
13           "001" WHEN x="00000010" ELSE
14           "010" WHEN x="00000100" ELSE
15           "011" WHEN x="00001000" ELSE
16           "100" WHEN x="00010000" ELSE
17           "101" WHEN x="00100000" ELSE
18           "110" WHEN x="01000000" ELSE
19           "111" WHEN x="10000000" ELSE
20           "ZZZ";
21 END encoder1;
```

```
1  ---- Solution 2: with WITH/SELECT/WHEN ------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY encoder IS
6     PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8  END encoder;
9  ---------------------------------------
10 ARCHITECTURE encoder2 OF encoder IS
11 BEGIN
12    WITH x SELECT
13       y <=   "000" WHEN "00000001",
14              "001" WHEN "00000010",
15              "010" WHEN "00000100",
16              "011" WHEN "00001000",
17              "100" WHEN "00010000",
18              "101" WHEN "00100000",
19              "110" WHEN "01000000",
20              "111" WHEN "10000000",
21              "ZZZ" WHEN OTHERS;
22 END encoder2;
```

Example 5.5: ALU



sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

```
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_unsigned.all;
5  ---------------------------------------
6  ENTITY ALU IS
7     PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
8                sel: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
9                cin: IN STD_LOGIC;
10               y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
11 END ALU;
12 ---------------------------------------
13 ARCHITECTURE dataflow OF ALU IS
14    SIGNAL arith, logic: STD_LOGIC_VECTOR (7 DOWNTO 0);
15 BEGIN
16    ----- Arithmetic unit: ------
17    WITH sel(2 DOWNTO 0) SELECT
18       arith <=  a WHEN "000",
19                 a+1 WHEN "001",
20                 a-1 WHEN "010",
21                 b WHEN "011",
22                 b+1 WHEN "100",
```

```
23                      b-1 WHEN "101",
24                      a+b WHEN "110",
25                      a+b+cin WHEN OTHERS;
26      ----- Logic unit: -----------
27      WITH sel(2 DOWNTO 0) SELECT
28         logic <=  NOT a WHEN "000",
29                   NOT b WHEN "001",
30                   a AND b WHEN "010",
31                   a OR b WHEN "011",
32                   a NAND b WHEN "100",
33                   a NOR b WHEN "101",
34                   a XOR b WHEN "110",
35                   NOT (a XOR b) WHEN OTHERS;
36      -------- Mux: --------------
37      WITH sel(3) SELECT
38         y <=  arith WHEN '0',
39               logic WHEN OTHERS;
40  END dataflow;
```

FOR / GENERATE:

```
label: FOR identifier IN range GENERATE
  (concurrent assignments)
END GENERATE;
```

- GENERATE یکی از
- اعلان های همروند است بهمراه عملگر ها و WHEN و معادل LOOP
- امکان تکرار بخشی از کد را فراهم میکند

IF/GENERATE can be nested inside FOR/GENERATE (

IF / GENERATE nested inside FOR / GENERATE:

```
label1: FOR identifier IN range GENERATE
    ...
    label2: IF condition GENERATE
       (concurrent assignments)
    END GENERATE;
    ...
END GENERATE;
```

Example:

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);

...
G1: FOR i IN x'RANGE GENERATE
    z(i) <= x(i) AND y(i+8);
END GENERATE;
```

- ابتدا و انتها باید استاتیک باشد

```
NotOK: FOR i IN 0 TO choice GENERATE
    (concurrent statements)
END GENERATE;
```

- سیگنالهایی که از چند مسیر مقدار گرفته اند باید مطلع باشیم.

```
OK: FOR i IN 0 TO 7 GENERATE
   output(i)<='1' WHEN (a(i) AND b(i))='1' ELSE '0';
END GENERATE;
```

```
NotOK: FOR i IN 0 TO 7 GENERATE
   accum <="11111111" WHEN (a(i) AND b(i))='1' ELSE "00000000";
END GENERATE;
```

5.5 BLOCK

There are two kinds of BLOCK statements: *Simple* and *Guarded*.

- برای تقسیم بندی موضعی کد استفاده می شود.
- افزایش قابلیت خواندن و مدیریت آن ها

```
label: BLOCK
  [declarative part]
BEGIN
  (concurrent statements)
END BLOCK label;
```

```
-------------------------
ARCHITECTURE example ...
BEGIN
    ...
    block1: BLOCK
    BEGIN
        ...
    END BLOCK block1
    ...
    block2: BLOCK
    BEGIN
        ...
    END BLOCK block2;
    ...
END example;
-------------------------
```

Example:

```
b1: BLOCK
    SIGNAL a: STD_LOGIC;
BEGIN
    a <= input_sig    WHEN ena='1' ELSE 'Z';
END BLOCK b1;
```

A BLOCK (simple or guarded) can be nested inside another BLOCK. The corresponding syntax is shown below.

```
label1: BLOCK
  [declarative part of top block]
BEGIN
  [concurrent statements of top block]
label2: BLOCK
  [declarative part nested block]
BEGIN
  (concurrent statements of nested block)
END BLOCK label2;
  [more concurrent statements of top block]
END BLOCK label1;
```

Guarded BLOCK

A *guarded* BLOCK is a special kind of BLOCK, which includes an additional expression, called *guard* expression. A guarded statement in a guarded BLOCK is executed only when the guard expression is TRUE.

Guarded BLOCK:

```
label: BLOCK (guard expression)
  [declarative part]
BEGIN
  (concurrent guarded and unguarded statements)
END BLOCK label;
```

```
1  ------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------
5  ENTITY latch IS
6     PORT (d, clk: IN STD_LOGIC;
7           q: OUT STD_LOGIC);
8  END latch;
9  ------------------------------
10 ARCHITECTURE latch OF latch IS
 11 BEGIN
 12    b1: BLOCK (clk='1')
 13    BEGIN
 14       q <= GUARDED d;
 15    END BLOCK b1;
 16 END latch;
 17 ------------------------------
```

- یک فیلیپ فلاپ نوع D حساس به لبه بالا رونده clk با reset

DFF Implemented with a Guarded BLOCK

```
1  ------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------
5  ENTITY dff IS
6     PORT ( d, clk, rst: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END dff;
9  ------------------------------
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12    b1: BLOCK (clk'EVENT AND clk='1')
13    BEGIN
14       q <= GUARDED '0' WHEN rst='1' ELSE d;
15    END BLOCK b1;
16 END dff;
17 ------------------------------
```

تمرین های فصل ۵

- از ۸ مسئله ۴ مسئله به دلخواه حل کنید

6 Sequential Code

PROCESSES,

FUNCTIONS

PROCEDURES

WAIT, IF, CASE, and LOOP.

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

6.2 Signals and Variables

VARIABLES are also restricted to be used in sequential code only (that is, inside a PROCESS, FUNCTION, or PROCEDURE). Thus, contrary to a SIGNAL, a VARIABLE can never be global, so its value can not be passed out directly.

We will concentrate on PROCESSES here. FUNCTIONS and PROCEDURES

VHDL has two ways of passing non-static values around: by means of a SIGNAL or by means of a VARIABLE. A SIGNAL can be declared in a PACKAGE, ENTITY or ARCHITECTURE (in its declarative part), while a VARIABLE can only be declared inside a piece of sequential code (in a PROCESS, for example). Therefore, while the value of the former can be global, the latter is always local.

The value of a VARIABLE can never be passed out of the PROCESS directly; if necessary, then it must be assigned to a SIGNAL. On the other hand, the update of a VARIABLE is immediate, that is, we can promptly count on its new value in the next line of code. That is not the case with a SIGNAL (when used in a PROCESS), for its new value is generally only guaranteed to be available *after* the conclusion of the present run of the PROCESS.

6.3 IF

As mentioned earlier, IF, WAIT, CASE, and LOOP are the statements intended for sequential code. Therefore, they can only be used inside a PROCESS, FUNCTION, or PROCEDURE.

The natural tendency is for people to use IF more than any other statement. the syn thesizer will optimize the structure and avoid the extra hardware

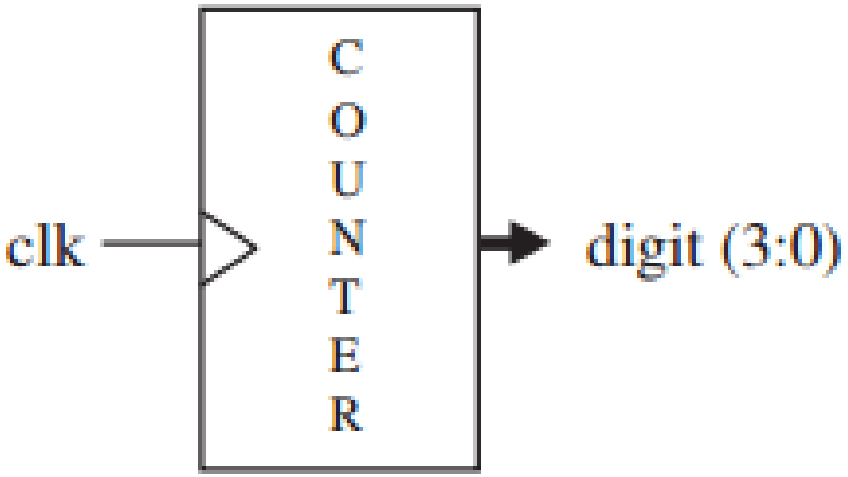
```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

Example:

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

Example 6.2: One-digit Counter #1

The code below implements a progressive 1-digit decimal counter ($0 \rightarrow 9 \rightarrow 0$). A top-level diagram of the circuit



```
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY counter IS
6     PORT (clk : IN STD_LOGIC;
7           digit : OUT INTEGER RANGE 0 TO 9);
8  END counter;
9  ---------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12    count: PROCESS(clk)
13       VARIABLE temp : INTEGER RANGE 0 TO 10;
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          temp := temp + 1;
17          IF (temp=10) THEN temp := 0;
18          END IF;
19       END IF;
20       digit <= temp;
21    END PROCESS count;
22 END counter;
```

6.4 WAIT

```
WAIT UNTIL signal_condition;
```

```
WAIT ON signal1 [, signal2, ... ];
```

```
WAIT FOR time;
```

Example: 8-bit register with synchronous reset.

```
PROCESS                 -- no sensitivity list
BEGIN
    WAIT UNTIL (clk'EVENT AND clk='1');
    IF (rst='1') THEN
        output <= "00000000";
    ELSIF (clk'EVENT AND clk='1') THEN
        output <= input;
    END IF;
END PROCESS;
```

```
PROCESS
BEGIN
    WAIT ON clk, rst;
    IF (rst='1') THEN

        output <= "00000000";
      ELSIF (clk'EVENT AND clk='1') THEN
        output <= input;
      END IF;
  END PROCESS;
```

Example 6.4: DFF with Asynchronous Reset #2

```
1  ------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY dff IS
6     PORT (d, clk, rst: IN STD_LOGIC;
7           q: OUT STD_LOGIC);
8  END dff;
9  ------------------------------------
10 ARCHITECTURE dff OF dff IS
11 BEGIN
12    PROCESS
13    BEGIN
14       WAIT ON rst, clk;
15       IF (rst='1') THEN
16          q <= '0';
17       ELSIF (clk'EVENT AND clk='1') THEN
18          q <= d;
19       END IF;
20    END PROCESS;
21 END dff;
22 ------------------------------------
```

Example 6.5: One-digit Counter #2

```
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY counter IS
6     PORT (clk : IN STD_LOGIC;
7           digit : OUT INTEGER RANGE 0 TO 9);
8  END counter;
9  ---------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12    PROCESS          -- no sensitivity list
13       VARIABLE temp : INTEGER RANGE 0 TO 10;
14    BEGIN
15       WAIT UNTIL (clk'EVENT AND clk='1');
16       temp := temp + 1;
17       IF (temp=10) THEN temp := 0;
18       END IF;
19       digit <= temp;
20    END PROCESS;
21 END counter;
```

6.5 CASE

CASE is another statement intended exclusively for sequential code (along with IF, LOOP, and WAIT). Its syntax is shown below.

```
CASE identifier IS
    WHEN value => assignments;
    WHEN value => assignments;
    ...
END CASE;
```

Example:

```
CASE control IS
    WHEN "00" => x<=a; y<=b;
    WHEN "01" => x<=b; y<=c;
    WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

```
WHEN value                      -- single value
WHEN value1 to value2           -- range, for enumerated data types
                                -- only
WHEN value1 | value2 |...       -- value1 or value2 or ...
```

Example 6.6: DFF with Asynchronous Reset #3

```
2  LIBRARY ieee;                        -- Unnecessary declaration,
3                                       -- because
4  USE ieee.std_logic_1164.all;         -- BIT was used instead of
5                                       -- STD_LOGIC
6  ----------------------------------------
7  ENTITY dff IS
8     PORT (d, clk, rst: IN BIT;
9           q: OUT BIT);
10 END dff;
11 ----------------------------------------
12 ARCHITECTURE dff3 OF dff IS
13 BEGIN
14    PROCESS (clk, rst)
15    BEGIN
16       CASE rst IS
```

```
17            WHEN '1' => q<='0';
18            WHEN '0' =>
19               IF (clk'EVENT AND clk='1') THEN
20                  q <= d;
21               END IF;
22            WHEN OTHERS => NULL;     -- Unnecessary, rst is of type
23                                     -- BIT
24         END CASE;
25      END PROCESS;
26 END dff3;
```

6.6 LOOP

FOR / LOOP: The loop is repeated a fixed number of times.

```
[label:] FOR identifier IN range LOOP
  (sequential statements)
END LOOP [label];
```

WHILE / LOOP: The loop is repeated until a condition no longer holds.

```
[label:] WHILE condition LOOP
  (sequential statements)
END LOOP [label];
```

EXIT: Used for ending the loop.

```
[label:] EXIT [label] [WHEN condition];
```

NEXT: Used for skipping loop steps.

```
[label:] NEXT [loop_label] [WHEN condition];
```

Example of FOR / LOOP:

```
FOR i IN 0 TO 5 LOOP
    x(i) <= enable AND w(i+2);
    y(0, i) <= w(i);
END LOOP;
```

Example of WHILE / LOOP: In this example, LOOP will keep repeating while i < 10.

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

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

Example with NEXT: In the example below, NEXT causes LOOP to skip one iteration when i = skip.

```
FOR i IN 0 TO 15 LOOP
    NEXT WHEN i=skip;      -- jumps to next iteration
        (...)
END LOOP;
```

Function & component

- مطالعه شود.