دانشگاه صنعتی شاهرود
دانشکده برق

عنوان درس

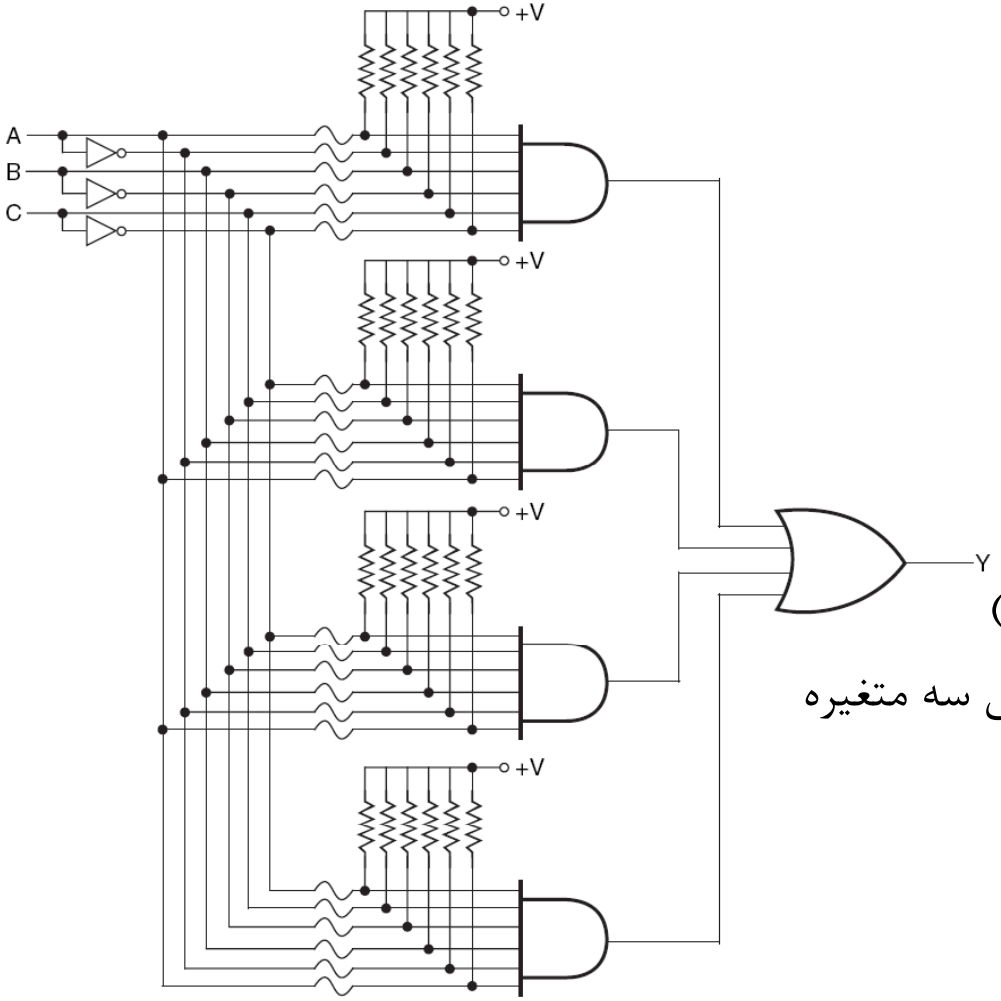
Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

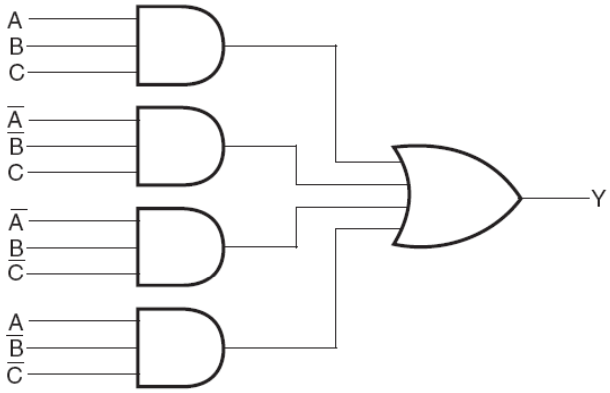
استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

- اجزای یک سیستم الکترونیک دیجیتال
  - حافظه ها
  - میکروپروسسورها
  - **عناصر منطقی**

- دو خانواده بزرگ از عناصر منطقی بر حسب نوع منطق استفاده شده در آنها
  - منطق ثابت (Fixed Logic)
  - منطق برنامه پذیر (Programmable Logic)

مثالی از منطق ثابت و منطق برنامه پذیر





- استفاده از آنتی فیوز (antifuse)
- مناسب پیاده سازی هر تابع بولی سه متغیره شامل حداکثر چهار مینترم

- **مقایسه دو منطق**
  - مدت زمان لازم از مرحله طراحی تا مرحله تولید و ساخت
  - مدت زمان و هزینه فرآیند طراحی و اعمال تغییرات و بررسی عملکرد مدار در شرایط واقعی
  - قدرت تغییر مدار توسط کاربر یا حتی خود مدار
  - کاربردهایی که نیاز به تولید انبوه دارند یا بالاترین میزان کارایی مورد نیاز است
- **مهمترین انواع PLDها** (بر حسب نوع معماری- ظرفیت منطقی- برنامه پذیر بودن)
  - ROM های برنامه پذیر (یا PROM)
  - آرایه منطقی برنامه پذیر (PLA)
  - منطق آرایه برنامه پذیر (PAL)
  - منطق آرایه عمومی (GAL)
  - ادوات منطقی برنامه پذیر پیچیده (CPLD)
  - آرایه گیتهای قابل برنامه ریزی در زمان استفاده (FPGA)

PROMها

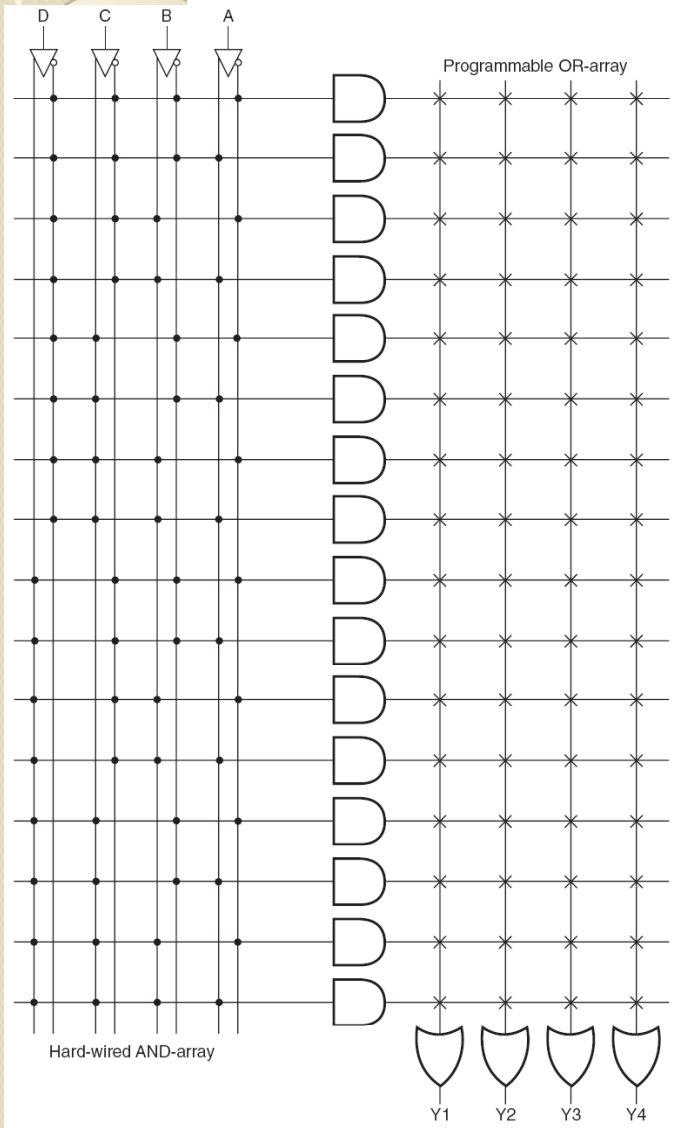
- در حکم اجداد PLD ها می باشند
- PROM ها را میتوان به کمک پروگرامرهای استاندارد برنامه-ریزی کرد
- دو شکل استفاده

حافظه

- دارای n خط ورودی (به نام خطوط آدرس) و m خط خروجی (به نام خطوط داده) جهت ذخیره 2^n کلمه m بیتی میباشد

ادوات PLD

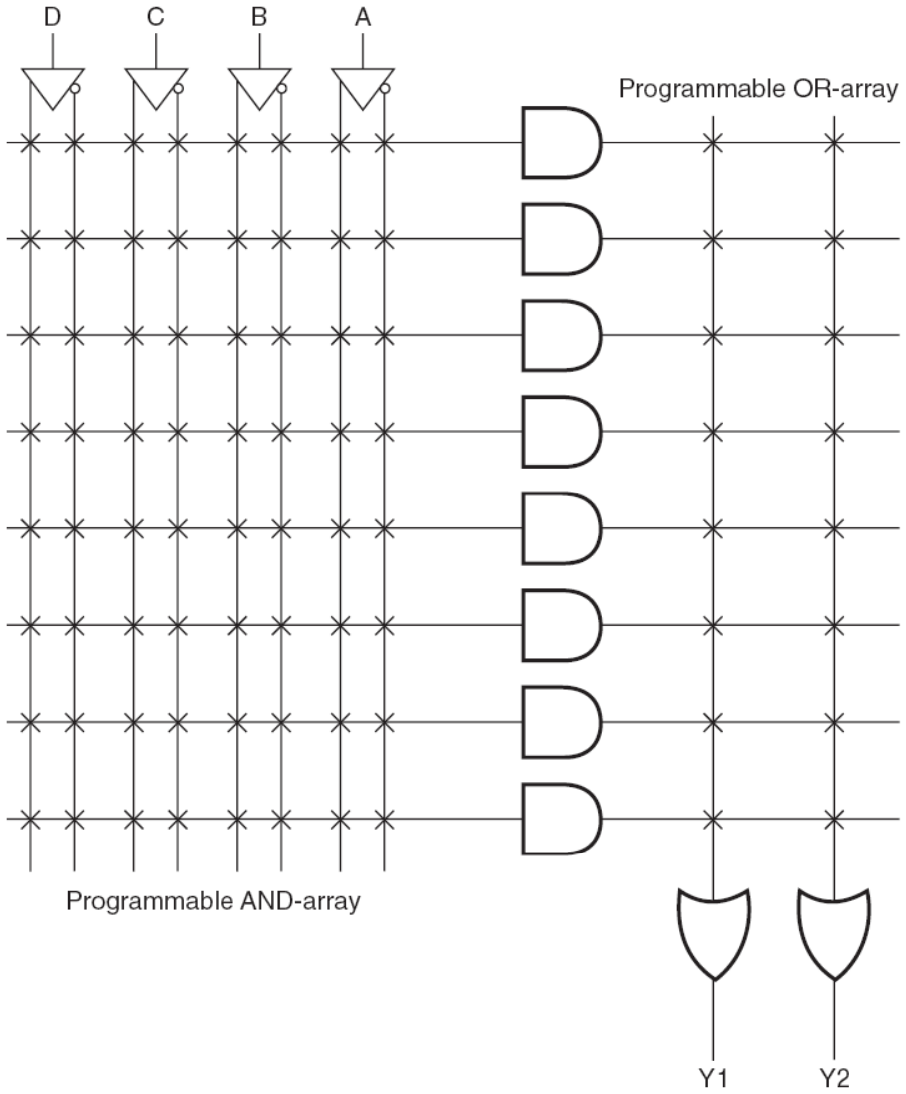
- دارای n خط ورودی و m خط خروجی است که میتوان جهت پیاده سازی m تابع ترکیبی مختلف استفاده کرد که هر تابع شامل n متغیر است
- **مثال:** استفاده از آرایه سیم بندی سخت شده در ورودی و سیم بندی قابل برنامه ریزی در خروجی: هر علامت ضربدر نشاندهنده یک اتصال برنامه ریزی نشده و مانند یک فیوز است.



- مهمترین معایب PROM ها
  - عدم استفاده بهینه از ظرفیت منطقی آنها
  - مصرف توان بالا
  - عدم توانایی در پیاده سازی مدارات ترتیبی

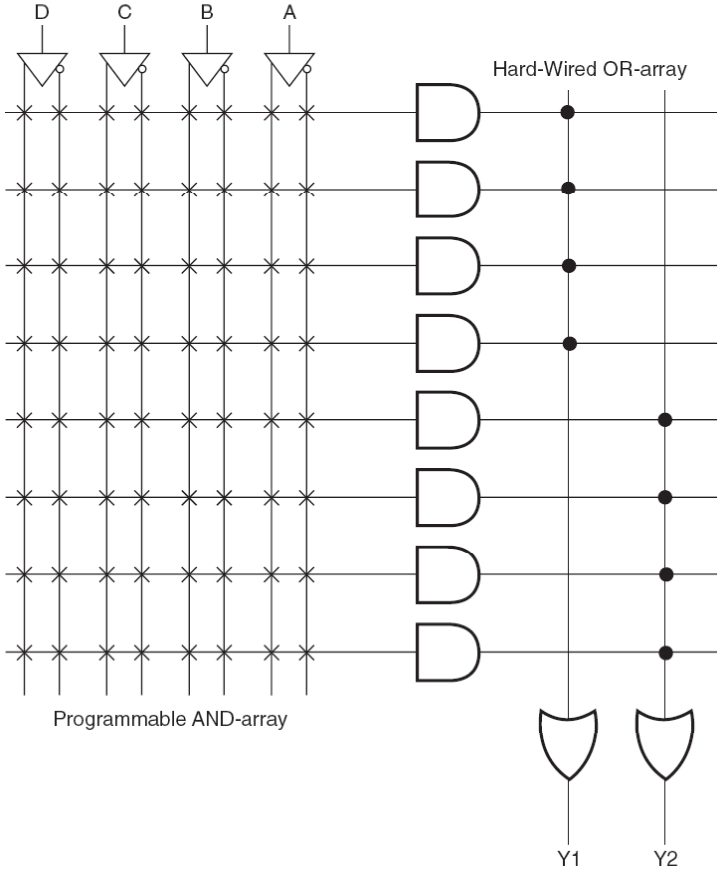
- آرایه منطقی برنامه پذیر یا PLA
  - شامل یک آرایه AND قابل برنامه ریزی در ورودی و یک آرایه OR قابل برنامه ریزی در خروجی میباشد
  - در PROM دارای m ورودی تعداد گیتهای AND **همواره** برابر 2^n است اما در PLA تعداد گیتهای AND معمولا بسیار کمتر است
  - در PLA از ظرفیت منطقی موثرتر از PROM استفاده می شود
  - مهمترین عیب PLA داشتن دو مجموعه از فیوزهای قابل برنامه ریزی است که موجب دشوارتر شدن ساخت و برنامه ریزی و تست آن میشود

- مثالی از یک PLA چهارمتغیره و دو خروجی



منطق آرایه برنامه پذیر یا PAL

- شامل یک آرایه AND قابل برنامه ریزی در ورودی و یک آرایه OR ثابت در خروجی است
- در PAL نیز مشابه با PLA تعداد گیتهای AND قابل برنامه ریزی معمولا کمتر از تعداد لازم برای تولید تمام مینترمهای ممکن از ورودیها می باشد
- مثالی از یک PAL چهار ورودی و دو خروجی

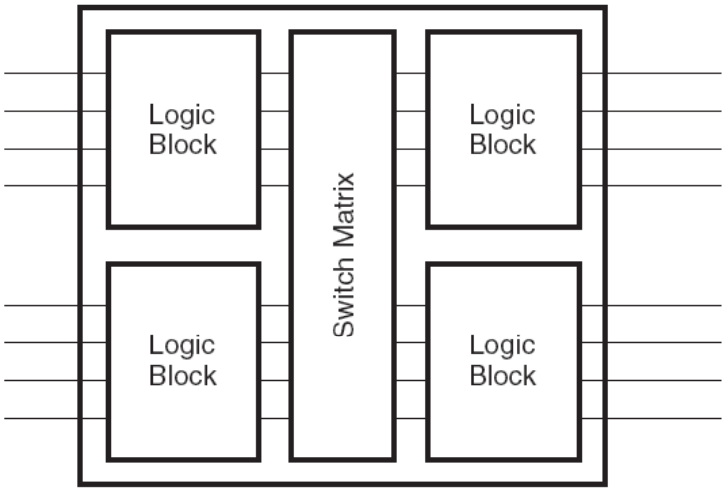


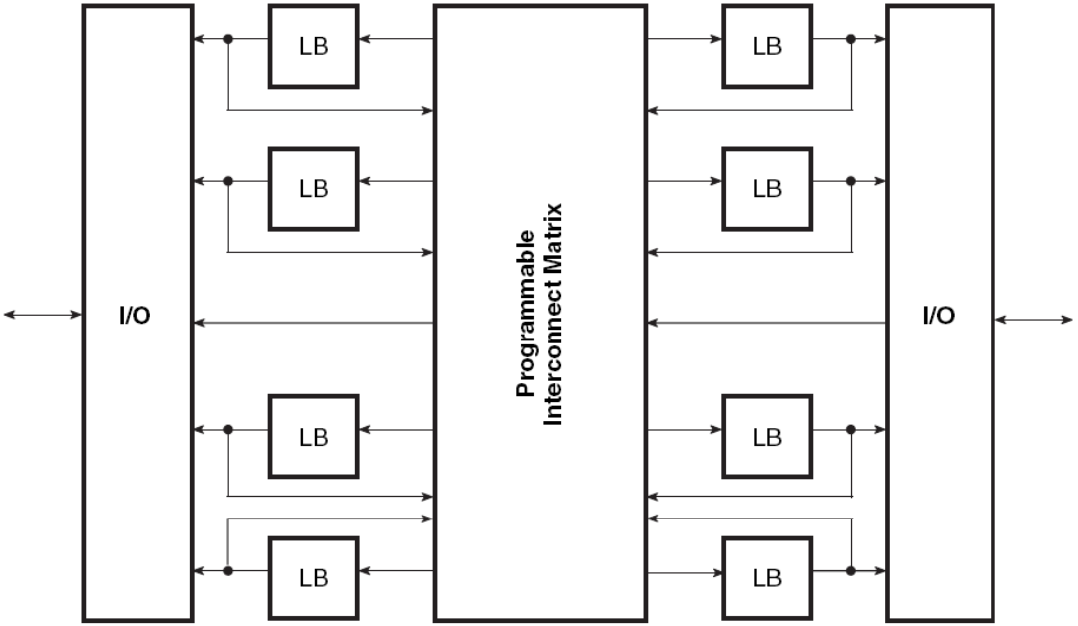
- منطق آرایه عمومی یا GAL
  - مشابه PAL بوده و توسط شرکت Lattice Semiconductor اختراع شده است
  - تفاوت GAL با PAL در این است که در GAL میتوان هم آرایه AND ورودی و هم آرایه OR خروجی را پاک و مجددا برنامه ریزی کرد که حسن بزرگی محسوب می شود
  - یک افزاره دیگر مشابه با GAL به نام PEEL (Programmable Electrically Erasable Logic) می باشد که محصول شرکت ICT می باشد

ادوات منطقی قابل برنامه ریزی پیچیده یا CPLD

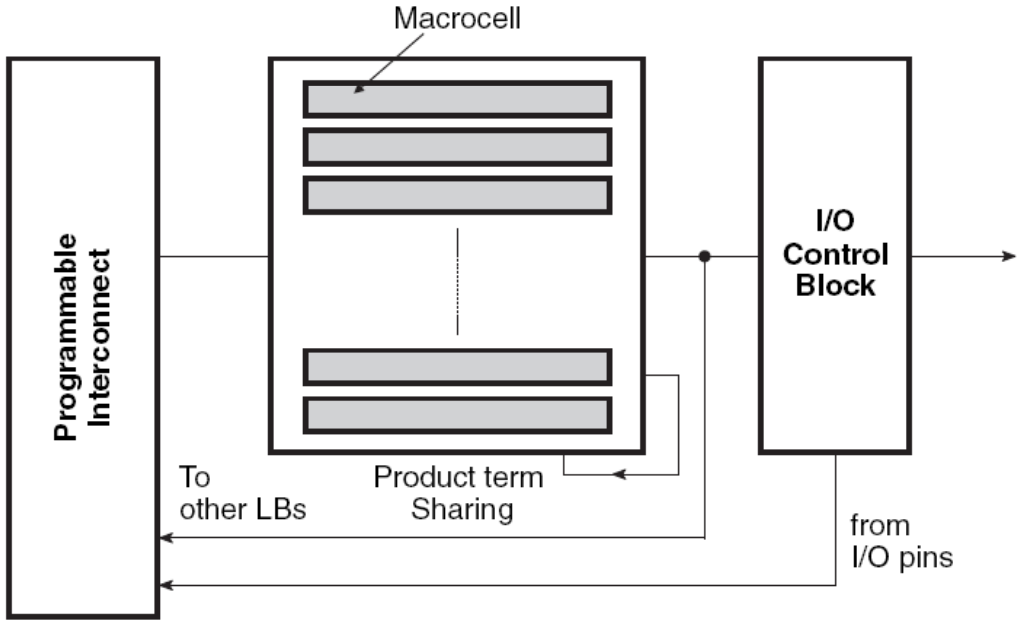
- ادوات PAL , PLA و GAL در دسته ای به نام SPLD (Simple Programmable Logic Devices) قرار داده می شوند تا از گروه CPLD که ساختار بسیار پیچیده تری دارند متمایز شوند.
- در SPLDها میزان پیچیدگی را نمیتوان (با افزایش تعداد ورودیها) از حدی بیشتر افزایش داد و برای این کار باید چندین SPLD را در کنار هم قرار داده و آنها را با اتصالات قابل برنامه ریزی به یکدیگر ربط داد. این تکنیک در CPLDها مورد استفاده قرار می گیرد
- یک CPLD ممکن است مداری معادل چندین PAL داشته باشد که توسط برخی اتصالات قابل برنامه ریزی به یکدیگر متصل شده باشند
- در حالیکه مرتبه پیچیدگی یک PAL نمونه در حدود چند صد گیت منطقی است مرتبه پیچیدگی یک CPLD ممکن است در حدود دهها هزار گیت باشد
- CPLD را میتوان به کمک پروگرامر PAL برنامه ریزی کرد و اگر روی یک برد لحیم شده باشد میتوان آن را از طریق خط داده سریال PC نیز برنامه ریزی نمود

- مثالی از یک نمونه متداول از ساختار داخلی یک CPLD
- اجزاء اصلی یک CPLD
  - چندین SPLD
  - ماتریس اتصالات قابل برنامه ریزی
  - بلوک کنترل I/O

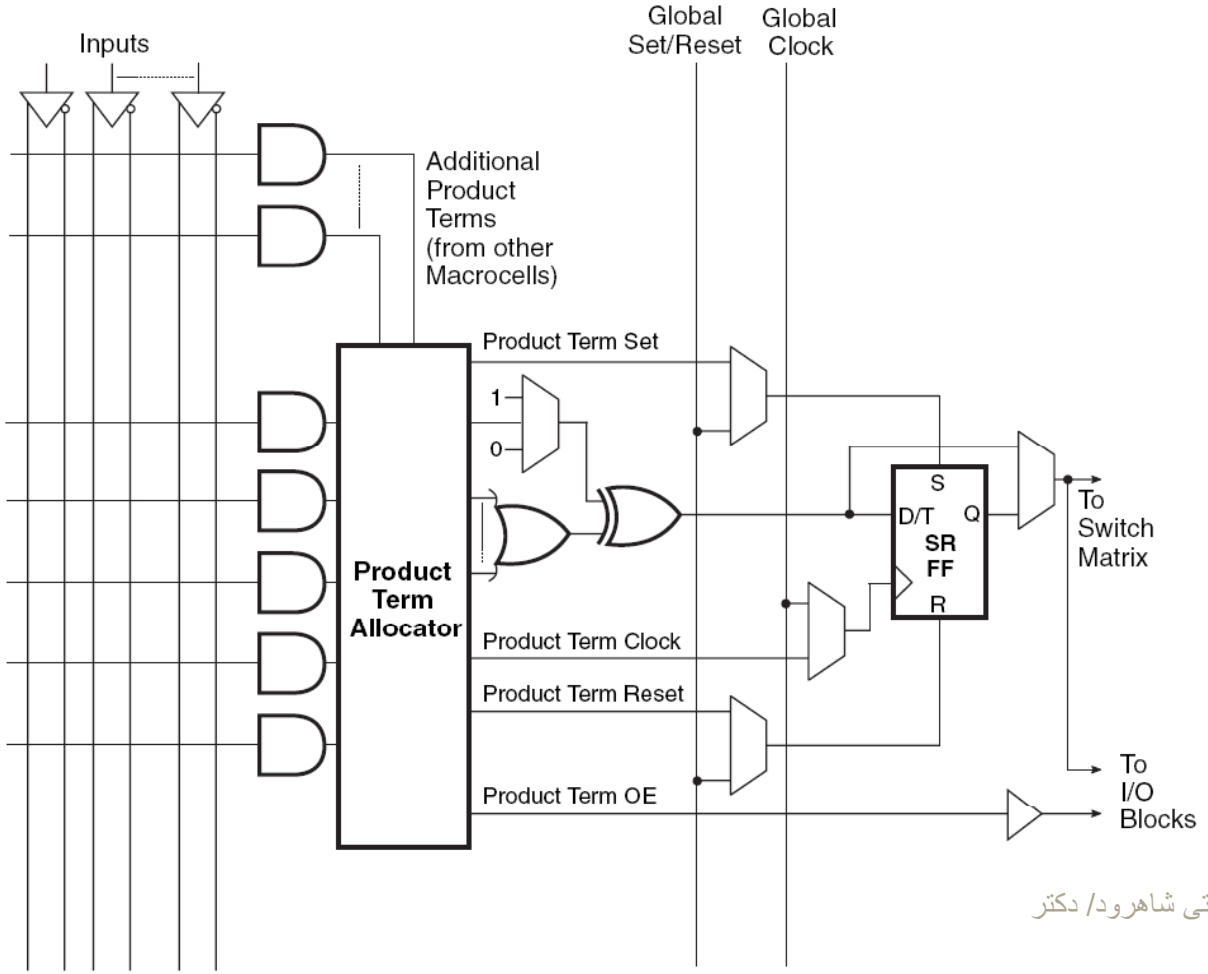




- هر بلوک منطقی (LB یا Logic Block) معادل یک PLD (مثلا یک PAL) بوده و دارای اتصالات قابل برنامه ریزی می باشد
- در این معماری از یک ماتریس سوئیچ قابل برنامه ریزی برای اتصال بین بلوکهای منطقی مختلف استفاده شده است.
- توسط ماتریس اتصالات:
  - ورودی یا خروجی هریک از بلوکهای منطقی قابل متصل شدن به هر بلوک منطقی دیگر است
  - پینهای ورودی و خروجی مستقیما قابل اتصال به ماتریس اتصالات و نیز بلوکهای منطقی هستند
- هر بلوک منطقی خود شامل چندین واحد منطقی کوچکتر به نام ***ماکروسل*** می باشد



- هرماکروسل معمولا شامل مجموعه ای از مینترمها (یا همان جملات حاصلضرب) – که توسط زیرمجموعه ای از آرایه AND قابل برنامه ریزی میشوند- و فیدبک کردن یک منطق خروجی قابل پیکره بندی میباشد.
- این منطق خروجی معمولا شامل یک گیت OR و یک گیت XOR و یک فلیپ فلاپ میباشد
- ورودیهای گیت OR میتوانند شامل تعدادی یا تمام مینترمهای تولید شده داخل ماکروسل و نیز گاها مینترمهای تولید شده توسط دیگر ماکروسل ها باشند

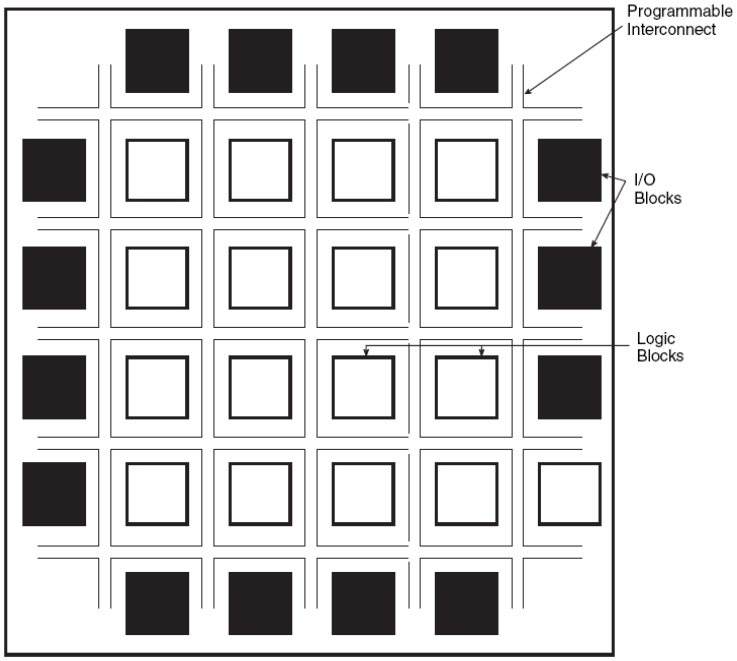


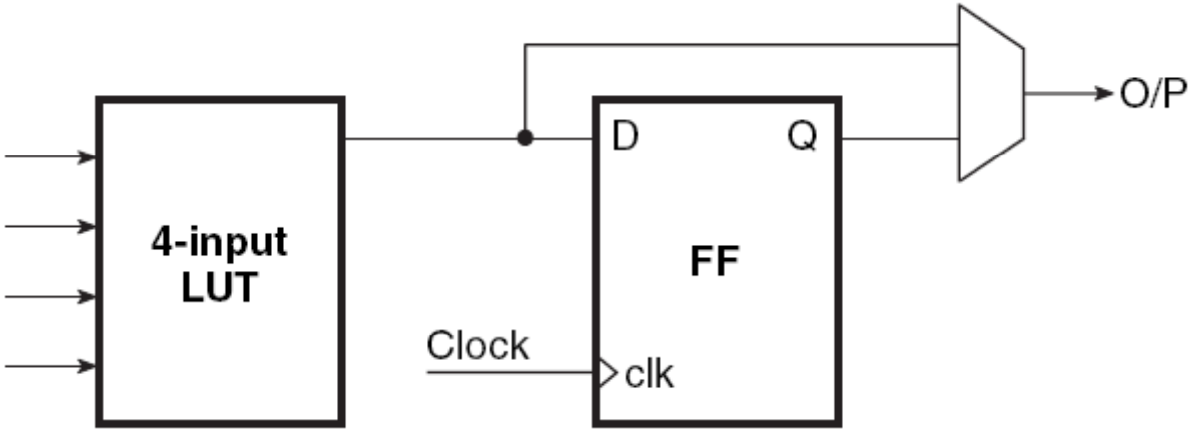
- برخی کاربردهای CPLDها
  - کاربردهای منطق متصل (Glue Logic)
  - پیاده سازی طراحی های کنترلی حساس و مهم مانند کنترلهای گرافیکی, کنترل کش (Cache Control), UARTها (universal asynchronous receiver/transmitter) و کنترلرهای LAN
  - تلفنهای همراه
  - همیارهای دیجیتال (Digital Assistants)
  - کاربردهایی که نیاز مبرم آنها به گیتهای AND و OR است نه فلیپ فلاپها
  - کاربردهایی که نیاز به تغییر سریع طراحی و حتی تغییر پیکره بندی سیستم دارند مثلا در تغییر پروتکل ارتباطی سیستمهای مخابراتی

- **آرایه گیتهای قابل برنامه ریزی در زمان استفاده یا FPGA**
- میزان پیچیدگی CPLDها نیز از حدی بیشتر قابل افزایش نیست. برای افزایش ظرفیت منطقی امروزه از FPGAها یا MPGAها استفاده میشود.
- تفاوت اساسی FPGA و CPLD مربوط به معماری داخلی آنهاست
  - CPLDها دارای تعداد بلوکهای منطقی کمتر اما پیچیده تر از FPGAها هستند
  - دو ویژگی بارز FPGAها ***اتصالات قابل برنامه ریزی*** آنها و ***بلوکهای منطقی نسبتا ساده*** آنهاست (ممکن است در حد یک ماکروسل باشند)
  - FPGAهای امروزی بعضا شامل برخی توابع سطح بالا , حافظه های درونی, میکروپروسسور, و حتی تمام ادوات لازم در یک SOPC (System On a Programmable Chip) میباشند. مانند FPGAهای Virtex-II Pro و Virtex 4 از شرکت Xilinx که شامل یک یا چند میکروپروسسور PowerPC (Performance Optimized With Enhanced RISC Processor Chip) می باشند.

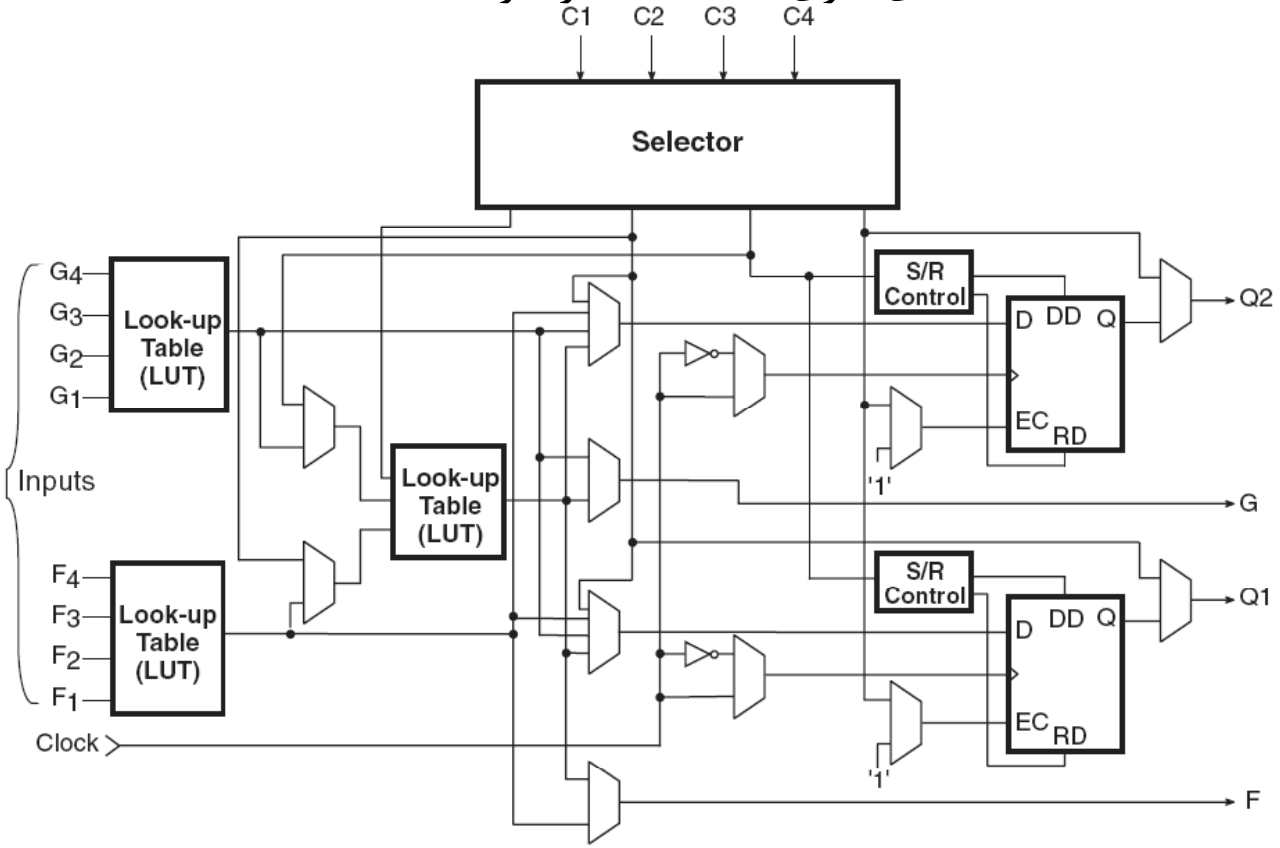
آرایه گیتهای قابل برنامه ریزی در زمان استفاده یا FPGA

- یک FPGA شامل آرایه ای از بلوکهای منطقی است که توسط کاربر قابل سازمان دهی می باشد
- سه بخش اصلی معماری داخلی یک FPGA
  - آرایه ای از بلوکهای منطقی
  - اتصالات داخلی قابل برنامه ریزی
  - بلوکهای I/O
- بلوکهای منطقی معمولا شامل تعدادی گیتهای منطقی و یا یک جدول جستجو (lookup table) هستند که متصل به یک فلیپ فلاپ هستند:





- بلوکهای I/O تعیین می کنند که پایه های همه-منظوره FPGA بصورت ورودی یا خروجی یا دوجهته (Bidirectional) استفاده شوند
- اتصالات داخلی قابل برنامه ریزی اتصال (دوطرفه) بین بلوکهای منطقی و بلوکهای I/O را برقرار می کنند
- به منظور افزایش تنوع توابع منطقی, بلوکهای منطقی ممکن است شامل بیش از یک LUT و یک فلیپ فلاپ باشند مانند FPGAهای سری XC4000 از شرکت Xilinx :



- برخی کاربردهای FPGAها
  - پردازش سیگنالهای دیجیتال
  - پردازش و ذخیره سیگنالها
  - رادیوی تعریف شده با نرم افزار(Software Defined Radio)
  - نمونه سازی برای ASIC (ASIC Prototyping)
  - بازشناسی گفتار (Speech Recognition)
  - بینایی کامپیوتر (Computer vision)
  - رمزنگاری (Cryptography)
  - تصویربرداری پزشکی (Medical Imaging)
  - سیستمهای دفاعی (Defense Systems)
  - ویژگیهای حیاتی (Bioinformatics)
  - نمونه سازی سخت افزاری کامپیوتر (Computer Hardware Emulation)
  - محاسبه با قابلیت پیکره بندی مجدد (Reconfigurable Computing)

- مقایسه بین FPGA ها و CPLD ها
  - در CPLD معماری داخلی دارای انعطاف کمتری است لذا مشخصات زمانی CPLD ها قابل پیش بینی تر می باشد
  - به دلیل ویژگی مذکور CPLD ها مناسب کارهای کنترلی حساس و دیگر کاربردهایی هستند که نیازمند سطح کارایی بالایی می باشند
  - به دلیل مصرف توان بسیار کم و قیمت کمتر CPLD ها برای کاربردهای قابل حمل و مبتنی بر باطری بسیار مناسب می باشند
  - ظرفیت منطقی FPGA ها و در نتیجه قدرت مانور آنها بیشتر از CPLDها است. FPGA های امروزی دارای ظرفیت حدود ده میلیون گیت هستند
  - FPGAهای امروزی بعضا شامل پردازشگرهای داخلی, حافظه های داخلی بزرگ, سیستمهای مدیریت کلاک, و پشتیبانی از بسیاری از تکنولوژیهای سیگنال دهی بین ادوات می باشند
  - FPGA ها در بسیاری از کاربردها عمدتا شامل ذخیره و پردازش داده, پردازش سیگنالهای دیجیتال, اندازه گیری, و مخابرات استفاده دارند
  - FPGA ها مشابه با CPLDها زمانی که روی برد لحیم شده باشند قابل برنامه ریزی هستند. این برنامه ریزی معمولا فرار (volatile) بوده و هربار که برق سیستم وصل میشود یا هر بار که نیاز به اعمال تغییرات و اصلاحات و یا تغییر کاربری باشد باید دوباره برنامه ریزی شوند

دانشگاه صنعتی شاهرود
دانشکده برق

عنوان درس

Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

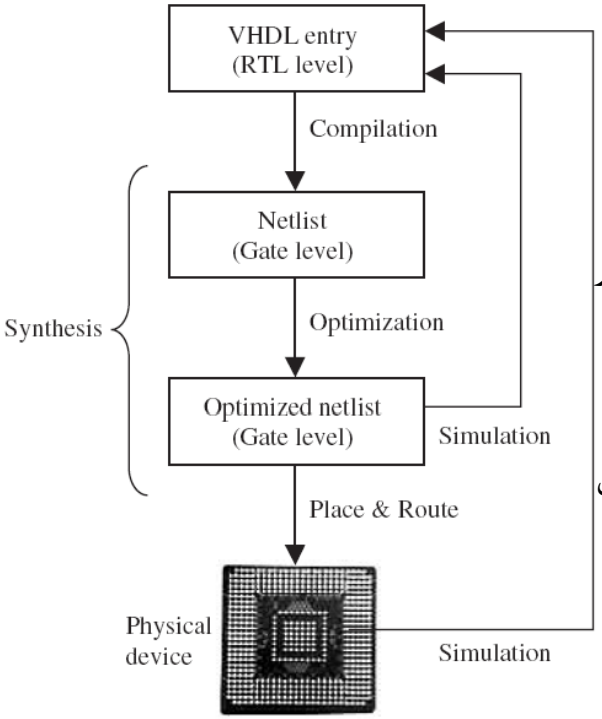
فصل اول : مقدمه (زبان VHDL)

- نام دیگر VHSIC HDL :

Very High Speed Integrated Circuits Hardware Description Language

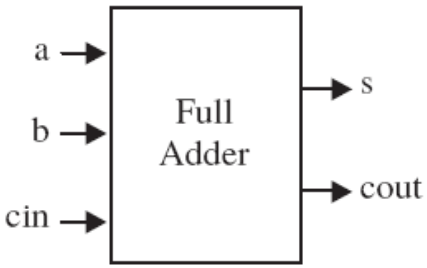
- دو کاربرد اصلی زبان VHDL (و البته رقیب آن Verilog)
  - ادوات منطقی برنامه پذیر (PLD)
  - مدارات مجتمع وابسته به کاربرد (ASIC)
- کد نوشته شده به زبان VHDL را میتوان یا برای پیاده سازی در یک PLD و یا برای ساخت آی سی متناظر آن توسط شرکتهای مربوطه مورد استفاده قرار داد.
- برای برنامه های نوشته شده به زبان VHDL معمولا از کلمه «کد» به جای «برنامه» استفاده می شود تا بر همزمان بودن (concurrent) رخدادها تاکید شود.

مراحل طراحی تا ساخت



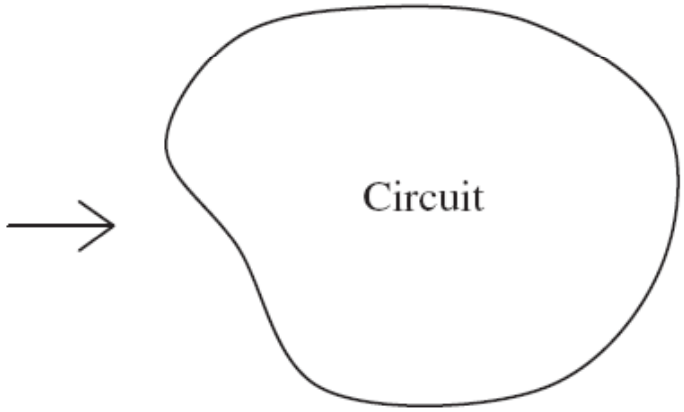
- ابتدا کد VHDL در یک فایل با پسوند vhd نوشته می شود (نام موجودیت مورد استفاده همان نام فایل است) این کد را اصطلاحا در سطح "انتقال ثبات" یا RTL (Register Transfer Level) گویند
- مرحله بعد سنتز کردن است:
  - در اولین مرحله کد را کامپایل می کنیم تا کد اولیه (در سطح انتقال ثبات) به «لیست عناصر یا netlist» تبدیل شود. لیست عناصر همان توصیف در سطح گیت است
  - در مرحله دوم مدار طراحی شده از لحاظ سرعت و مساحت اشغالی بهینه می شود
  - حال میتوان شبیه سازی را انجام داده و در صورت لزوم تغییرات مورد نظر را اعمال و سپس مراحل قبل را تکرار نمود
- در مرحله آخر از یک نرم افزار تطبیق دهنده (fitter) به منظور جایگذاری و تعیین مسیر و در نتیجه تولید چیدمان فیزیکی (physical layout) چیپ PLD/FPGA استفاده می شود.

- مثالی از یک نمونه کد VHDL
  - کد شامل یک موجودیت (Entity) و یک معماری (Architecture) می باشد:
    - موجودیت شامل توصیف پینها (یا پورتها)ی مدار
    - معماری شامل توصیف نحوه عملکرد مدار
  - از کد نشان داده شده میتوان چندین مدار معادل را بدست آورد که عملکرد یکسانی داشته باشند

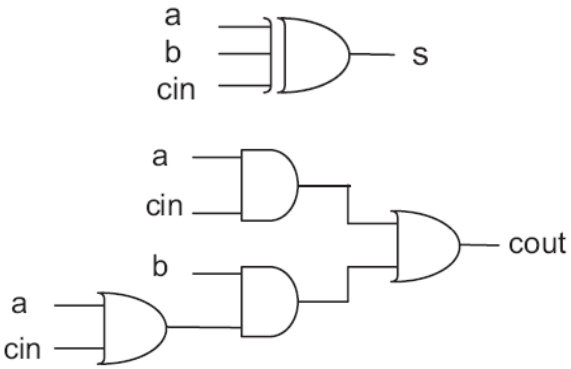


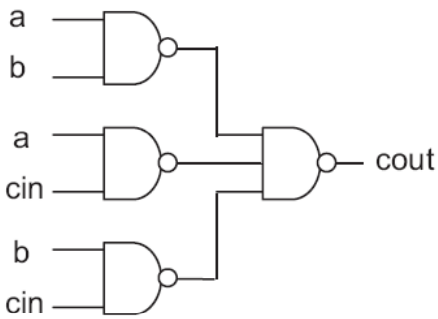
a b	cin	s	cout
0 0	0	0	0
0 1	0	1	0
1 0	0	1	0
1 1	0	0	1
0 0	1	1	0
0 1	1	0	1
1 0	1	0	1
1 1	1	1	1

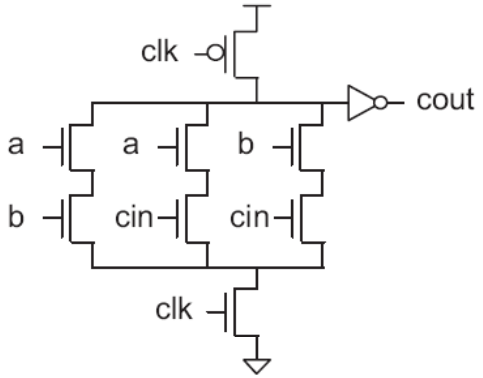
```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
      s, cout: OUT BIT);
END full_adder;
---------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```



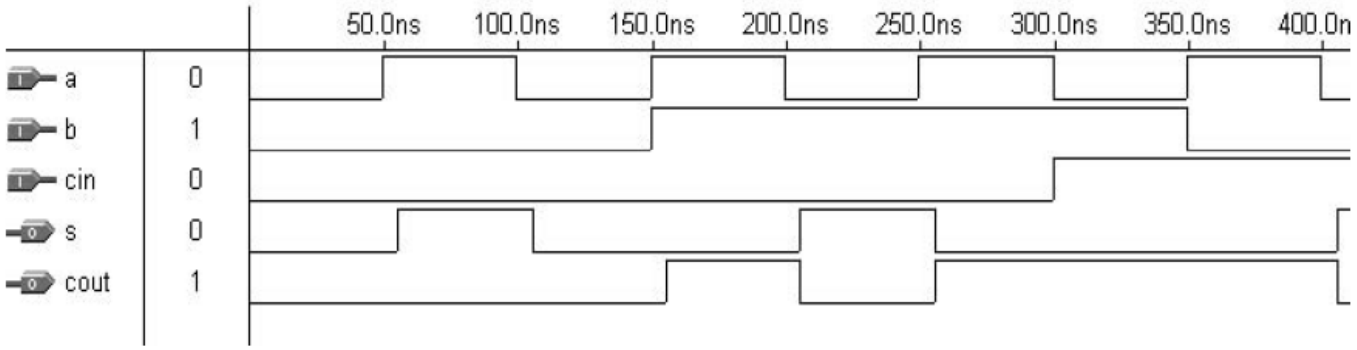
- چند نمونه مدار ممکن





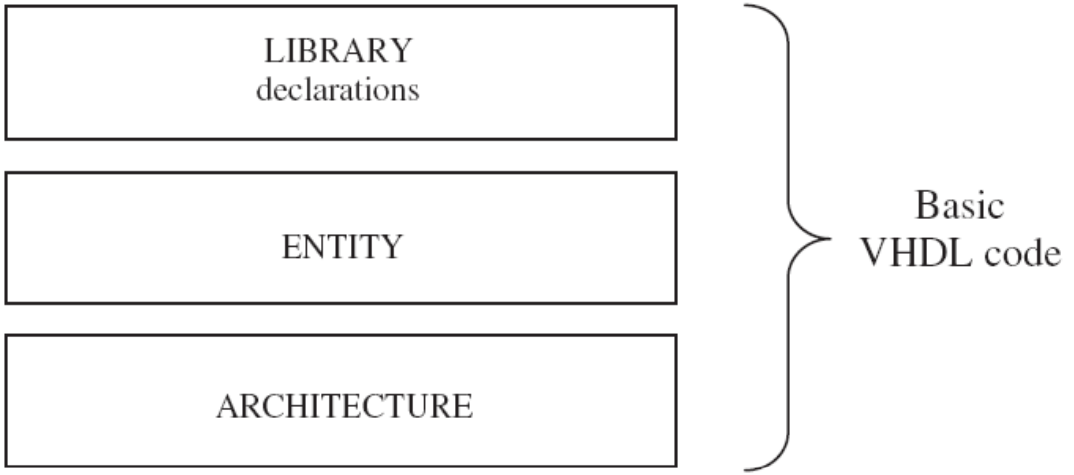


  - نوع مداری که نهایتا انتخاب می شود بستگی به نرم افزار کامپایلر, نرم افزار بهینه ساز, و نوع تکنولوژی مورد استفاده در افزاره مورد نظر ما دارد لذا در اکثر نرم افزارها نوع افزاره در همان مرحله شروع پرسیده می شود
- نتیجه شبیه سازی کد VHDL مذکور



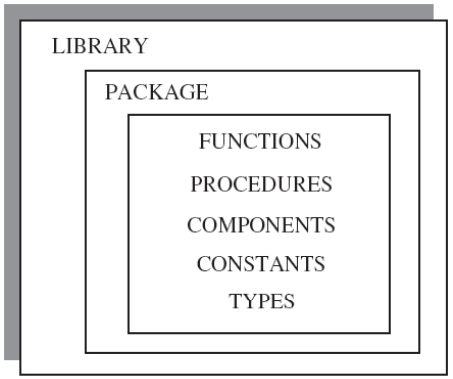
فصل دوم : ساختار کدنویسی

- سه بخش اساسی هر کد VHDL
  - ***اعلان کتابخانه*** (Library Declaration) شامل لیستی از تمام کتابخانه های مورد استفاده در کد – طراحی هم گفته میشود – می باشد
  - ***موجودیت*** (Entity): پینها (یا پورتها)ی I/O مدار را مشخص می کند.
  - ***معماری*** (Architecture): نحوه عملکرد مدار را مشخص می کند.



کتابخانه

- یک کتابخانه مجموعه ای از قطعات کد متداول (پراستفاده) است.
- استفاده از ایده قراردادن چنین کدهایی در کتابخانه باعث برخوردار شدن از قابلیتهای استفاده مجدد (reuse) و به اشتراک گذاری (sharing) می شود.
- ساختار معمول یک کتابخانه



- نحوه ***اعلان*** کتابخانه : به دو خط کد نیاز داریم

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

- معمولا در هر کد VHDL به حداقل سه بسته (Package) متعلق به سه کتابخانه مختلف نیاز داریم:
  - بسته std_logic_1164 متلق به کتابخانه ieee
  - بسته standard متلق به کتابخانه std
  - بسته work متلق به کتابخانه work

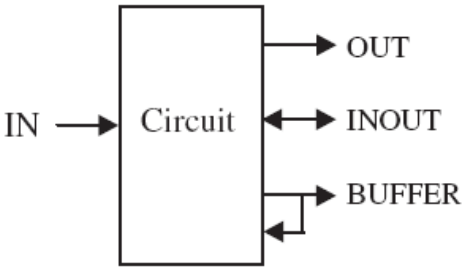
```
LIBRARY ieee;                    -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all;     -- the end of a statement or

LIBRARY std;                     -- declaration, while a double
USE std.standard.all;            -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;
```

موجودیت (Entity)

- نحوه تعریف موجودیت

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

- پارامتر ***mode*** شامل یکی از مقادیر in , out , inout , و یا buffer میباشد
  - نوعهای in و out یکطرفه و نوع inout دوطرفه میباشند.
  - نوع buffer زمانی استفاده میشود که بخواهیم یک سیگنال خروجی را به صورت داخلی نیز استفاده کنیم.



- پارامتر ***type*** شامل یکی از انواع داده مانند bit و std_logic و integer میباشد (انواع داده در فصل سوم بررسی میشوند)
- پارامتر ***name*** میتواند هر اسمی به جز کلمات رزرو شده VHDL باشد (این کلمات در ضمیمه E لیست شده اند)

معماری (Architecture)

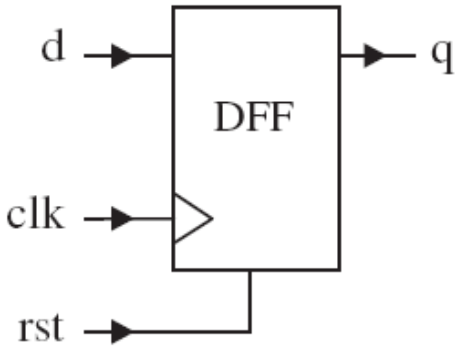
- **تعریف معماری** : دو بخش دارد
  - **بخش اعلان** (اختیاری و در صورت لزوم) : سیگنالها و متغیرها و ثوابت تعریف میشوند
  - **بخش کد** : عملکرد مدار توصیف میشود
- **نام معماری**
  - از قوانینی مشابه با نام موجودیت تبعیت می کند
  - نام معماری میتواند همان نام موجودیت نیز باشد

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

- مثال: یک گیت NAND ساده (یک مدار کاملا ترکیبی)

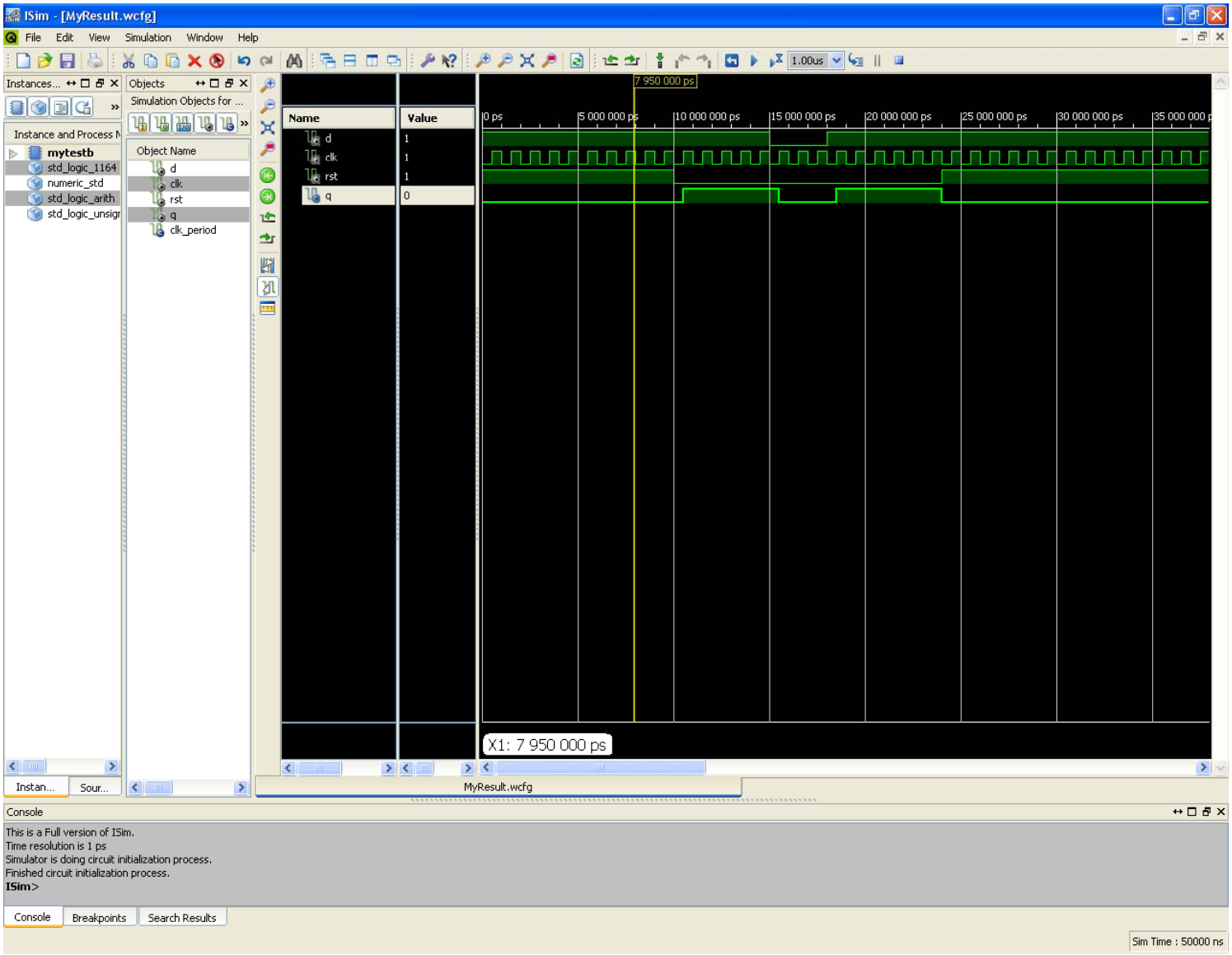
```
ENTITY nand_gate IS
   PORT (a, b : IN BIT;
         x : OUT BIT);
END nand_gate;
ARCHITECTURE myarch OF nand_gate IS
BEGIN
   x <= a NAND b;
END myarch;
```

- **مثال** : یک DFF با ورودی ریست سنکرون (یک مدار کاملا ترتیبی)

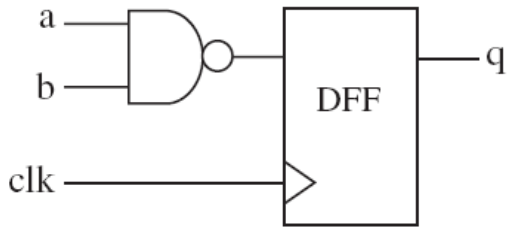



```
1  ---------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk, rst: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END dff;
9  ---------------------------------------
10 ARCHITECTURE behavior 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 ---------------------------------------
```

- نتیجه شبیه سازی با نرم افزار ISE



- مثال : یک DFF به همراه گیت NAND




```
1  ---------------------------------------
2  ENTITY example IS
3     PORT ( a, b, clk: IN BIT;
4            q: OUT BIT);
5  END example;
6  ---------------------------------------
7  ARCHITECTURE example OF example IS
8     SIGNAL temp : BIT;
9  BEGIN
10    temp <= a NAND b;
11    PROCESS (clk)
12    BEGIN
13       IF (clk'EVENT AND clk='1') THEN q<=temp;
14       END IF;
15    END PROCESS;
16 END example;
17 ---------------------------------------
```

فصل سوم : انواع داده

- انواع داده
  1. الف- داده های از قبل تعریف شده
  2. ب- داده های تعریف شده توسط کاربر
- **الف**- داده های از قبل تعریف شده
  1. BIT و BIT_VECTOR
  2. STD_LOGIC و STD_LOGIC_VECTOR
  3. STD_ULOGIC و STD_ULOGIC_VECTOR
  4. BOOLEAN
  5. INTEGER
  6. NATURAL
  7. REAL
  8. SIGNED و UNSIGNED
- **ب**- داده های تعریف شده توسط کاربر
  1. INTEGER
  2. ENUMERATED

- الف – داده های از قبل تعریف شده
  - این داده ها در استانداردهای IEEE 1076 و IEEE 1164 و در کتابخانه ها و بسته های زیر تعریف شده اند:
    - بسته standard از کتابخانه std شامل نوعهای BIT و BOOLEAN و INTEGER و REAL
    - بسته std_logic_1164 از کتابخانه ieee شامل نوعهای STD_LOGIC و STD_ULOGIC
    - بسته std_logic_arith از کتابخانه ieee شامل نوعهای SIGNED و UNSIGNED به همراه تعریف چندین تابع تبدیل نوع مانند conv_integer(p) و conv_uinsigned(p) و ... است
    - بسته های std_logic_signed و std_logic_unsigned از کتابخانه ieee شامل توابعی است که اجازه انجام عملیات روی داده های STD_LOGIC_VECTOR را گویی که آنها از نوع به ترتیب SIGNED و UNSIGNED هستند می دهد

۱. نوع BIT و BIT_VECTOR

- همان سطح منطقی ۰ و ۱ است

```
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.
```

Based on the signals above, the following assignments would be legal (to assign a value to a signal, the "<=" operator must be used):

```
x <= '1';
-- x is a single-bit signal (as specified above), whose value is
-- '1'. Notice that single quotes (' ') are used for a single bit.

y <= "0111";
-- y is a 4-bit signal (as specified above), whose value is "0111"
-- (MSB='0'). Notice that double quotes (" ") are used for
-- vectors.

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

۲. نوع داده STD_LIGIC و STD_LOGIC_VECTOR

- یک سیستم منطقی ۸ مقداره متعلق به استاندارد ieee 1164 است که بیشتر این مقادیر فقط برای مقاصد شبیه سازی استفاده می شوند
- فقط مقادیر '0' و '1' و 'Z' قابل سنتز هستند
- حالتهای ضعیف (weak) و قوی (forcing) به این دلیل استفاده شده اند که هرگاه دو سیگنال از نوع std_logic به یک گره متصل باشند مقادیر نوع قوی تعیین کننده نهایی خواهند بود (جدول زیر)

'X'	Forcing Unknown	(synthesizable unknown)
'0'	Forcing Low	(synthesizable logic '1')
'1'	Forcing High	(synthesizable logic '0')
'Z'	High impedance	(synthesizable tri-state buffer)
'W'	Weak unknown	
'L'	Weak low	
'H'	Weak high	
'–'	Don't care	

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

۳. نوع STD_ULOGIC و STD_ULOGIC_VECTOR

- یک سیستم منطقی ۹ مقداره متعلق به استاندارد ieee 1164 است که بیشتر این مقادیر فقط برای مقاصد شبیه سازی استفاده می شوند
- علاوه بر هشت مقدار قبلی مقدار ‘U’ (Unresolved) نیز اضافه شده است
- در اینجا بر خلاف جدول مذکور در نوع قبلی (جهت حل اختلاف مقادیر متصل به یک گره) نمیتوان سیمهای با نوع مخالف را به یکدیگر متصل نمود و برای آشکارسازی چنین خطایی مقدار جدید U اضافه شده است.

۴. نوع BOOLEAN

- شامل مقادیر TRUE و FALSE می باشد

۵. نوع INTEGER

- شامل اعداد صحیح از ۲۱۴۷۴۸۳۶۴۷- تا ۲۱۴۷۴۸۳۶۴۷ می باشد

۶. نوع NATURAL

- شامل اعداد صحیح نامنفی از ۰ تا ۲۱۴۷۴۸۳۶۴۷ می باشد

۷. نوع REAL

- شامل اعداد حقیقی در بازه 1.0E38- تا 1.0E38 است. این نوع قابل سنتز نیست

۸. نوع SIGNED و UNSIGNED

- این انواع داده در بسته std_arith_logic از کتابخانه ieee تعریف شده اند
- ظاهر آنها مشابه با نوع STD_LOGIC_VECTOR است اما قابلیت انجام اعمال محاسباتی را نیز دارند (بر خلاف نوع STD_LOGIC_VECTOR) همانطور که میدانیم تاکنون فقط نوع INTEGER قابلیت پذیرش اعمال محاسباتی را دارا بود
- این انواع در بخش 3.6 بیشتر توضیح داده میشوند

- مثال ۱

```
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
```

- مثال ۲

Example: Legal and illegal operations between data of different types.

```
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)
```

- ب – داده های تعریف شده توسط کاربر
  - کاربر امکان تعریف دو نوع داده را دارد:
    - نوع INTEGER

```
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 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.
```

    - نوع شمارش شده یا ENUMERATED

```
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.
```

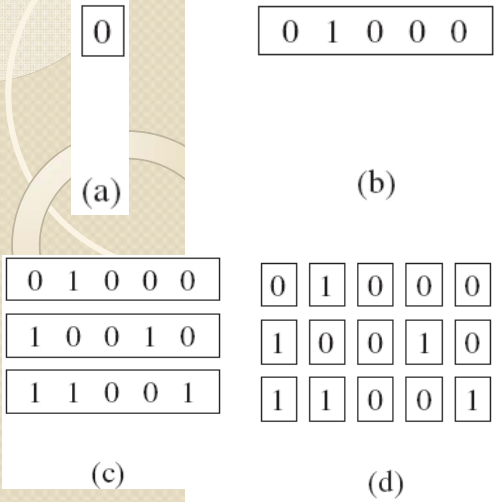
- **زیرنوع** یا SUBTYPE
- یک زیرنوع در واقع همان نوع یا TYPE است که برای آن محدودیتی قائل شده ایم
- انجام اعمال مختلف (مانند اعمال محاسباتی) بین داده های از انواع مختلف **ممنوع** است اما برای دو مقداری که یکی زیرنوع دیگری باشد **مجاز** است
- مثال۱: (با توجه به مثالهای قبلی)

```
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.
```

- مثال ۲

```
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)
```

- آرایه (ARRAY)
  - یک آرایه در حقیقت مجموعه ای از اشیاء هم نوع می باشد
  - انواع یا دسته های مختلف آرایه ها:
    - (الف) تک مقدار یا اسکالر (scalar) – شکل a
    - (ب) بردار یا آرایه یک بعدی (نوع 1D) – شکل b
    - (ج) آرایه ای از بردارها (نوع 1D×1D) – شکل c
    - (د) آرایه ای از اسکالرها (نوع 2D) – شکل d



  - در عمل از بین چهار دسته فوق فقط دسته های الف (اسکالر) و ب (آرایه ای از اسکالرها) در VHDL تعریف شده اند. در این دسته ها آن انواعی که قابل سنتز هستند عبارتند از
    - در اسکالرها انواع BIT, STD_LOGIC, STD_ULOGIC, BOOLEAN
    - در بردارها انواع BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, INTEGER, SIGNED, UNSIGNED
  - دو دسته دیگر (یعنی دسته های **ج** و **د**) را خود کاربر میتواند تعریف کند. برای این کار به کمک دستور TYPE نوع جدیدی تعریف شده و سپس متغیرها یا سیگنالها یا ثابتهای دلخواه از نوع جدید مذکور اعلان می شوند ....

...

- نحوه تعریف نوع جدید

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

- نحوه اعلان (یا استفاده)

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

  - به جای SIGNAL میتوان در صورت لزوم از VARIABLE یا CONSTANT استفاده کرد
  - مقدار اولیه فقط به منظور شبیه سازی بوده و قابل سنتز نیست

- **مثال** : تعریف یک آرایه 1D×1D

```
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
```

- راه دیگر برای حل مثال فوق

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

- **مثال** : تعریف یک آرایه 2D

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

- **مثال** : مقداردهی اولیه یک آرایه
  - مقداردهی اولیه یک آرایه اختیاری بوده و فقط در شبیه سازی کاربرد دارد و نحوه آن چنین است:

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

- **مثال** : برخی مثالهای مربوط به تخصیصهای مجاز و غیرمجاز آرایه ای
  - ابتدا تعاریف را انجام می دهیم:

```
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;
```

- حال تخصیصها را انجام می دهیم:

```
--------- 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)
```

- اعلان پورتهای ورودی/خروجی یک مدار به عنوان آرایه ای از بردارها:
  - داخل موجودیت نمیتوان از TYPE جهت ایجاد نوع جدید و استفاده از آن جهت اعلان پورتها استفاده کنیم به جای آن باید نوع مورد را داخل یک بسته (PACKAGE) اعلان نمود تا برای کل طراحی قابل مشاهده و استفاده باشد
- مثال

```
------- Package: -------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------
PACKAGE my_data_types IS
   TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
      STD_LOGIC_VECTOR(7 DOWNTO 0);
END my_data_types;
-----------------------------------------

------- Main code: ------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_data_types.all;     -- user-defined package
---------------------------
ENTITY mux IS
   PORT (inp: IN VECTOR_ARRAY (0 TO 3);
   ... );
END mux;
   ... ;
-----------------------------------------
```

- در مثال اخیر میتوان بسته را بصورت زیر با استفاده از CONSTANT نوشت

```
------- Package: ------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------
PACKAGE my_data_types IS
   CONSTANT b: INTEGER := 7;
   TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
      STD_LOGIC_VECTOR(b DOWNTO 0);
END my_data_types;
-----------------------------------------------
```

- انواع داده ای علامتدار (SIGNED) و بدون علامت (UNSIGNED)
  - این انواع در بسته std_arith_logic از کتابخانه ieee تعریف شده اند.
  - گرامر استفاده با ذکر مثال چنین است:

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

  - ***توجه*** کنید که گرامر آن بسیار شبیه به نوع STD_LOGIC_VECTOR است نه نوع INTEGER (بر خلاف آنچه ممکن است در ابتدا به نظر آید)
  - مقادیر UNSIGNED همگی مثبت هستند مثلا 0101 معادل ۵ و 1101 معادل ۱۳ است اما در نوع SIGNED مقادیر ب هصورت متمم ۲ هستند یعنی 0101 معادل ۵ و 1101 معادل ۳- است
  - از نوع STD_LOGIC_VECTOR فقط در عملیات ***منطقی*** میتوان استفاده و نه در عملیات ***حسابی***
  - از نوعهای SIGNED و UNSIGNED فقط در عملیات حسابی میتوان استفاده کرد و نه در عملیات ***منطقی***
  - اگر بخواهیم از نوع STD_LOGIC_VECTOR هم در عملیات حسابی و هم در عملیات منطقی استفاده کنیم باید بسته های std_logic_signed و std_logic_unsigned متعلق به کتابخانه ieee استفاده کنیم.
  - **مثالها ...**

Example: Legal and illegal operations with signed/unsigned data types.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;    -- extra package necessary
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
v <= a + b;      -- legal (arithmetic operation OK)
w <= a AND b;    -- illegal (logical operation not OK)
```

Example: Legal and illegal operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;    -- no extra package required
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;      -- illegal (arithmetic operation not OK)
w <= a AND b;    -- legal (logical operation OK)
```

Example: Arithmetic operations with std_logic_vector.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;    -- extra package included
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;       -- legal (arithmetic operation OK), unsigned
w <= a AND b;     -- legal (logical operation OK)
```

- **تبدیل داده:**
  - هیچ عملیاتی را نمیتوان مستقیما بین دو داده با نوع مختلف انجام داد و برای اینکار باید تبدیل نوع انجام داد
  - دو راه تبدیل داده:
    - یا یک قطعه کد مینویسیم که کار تبدیل داده را انجام دهد
    - یا از تابع مناسبی از یکی از بسته های موجود استفاده میکنیم

- **تبدیل داده:**
- اگر پایه دو نوع مختلف، یکی باشد آنگاه بسته std_logic_1164 از کتابخانه ieee روش سرراستی برای تبدیل در اختیار ما قرار می دهد:

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
```

- بسته std_logic_arith از کتابخانه ieee شامل چندین تابع مفید است:

۱. تابع conv_integer(p) برای تبدیل مقدار یک پارامتر p از نوع INTEGER، UNSIGNED، SIGNED، یا STD_ULOGIC به مقداری از نوع INTEGER تبدیل می کند.

۲. conv_unsigned(p,b) همان انواع قبلی را به مقداری از نوع UNSIGNED تبدیل میکند

۳. conv_signed(p,b) همان انواع قبلی را به مقداری از نوع SIGNED تبدیل میکند

۴. Conv_std_logic_vector(p,b) پارامتر p از نوع INTEGER، UNSIGNED، SIGNED، و یا STD_LOGIC را به مقداری از نوع STD_LOGIC_VECTOR به اندازه b بیت تبدیل می کند

- **مثال**

Example: Data conversion.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
y <= CONV_STD_LOGIC_VECTOR ((a+b), 8);
-- Legal operation: a+b is converted from UNSIGNED to an
-- 8-bit STD_LOGIC_VECTOR value, then assigned to y.
```

- **خلاصه انواع داده ها**

Synthesizable data types.

Data types	Synthesizable values
BIT, BIT_VECTOR	'0', '1'
STD_LOGIC, STD_LOGIC_VECTOR	'X', '0', '1', 'Z' (resolved)
STD_ULOGIC, STD_ULOGIC_VECTOR	'X', '0', '1', 'Z' (unresolved)
BOOLEAN	True, False
NATURAL	From 0 to +2, 147, 483, 647
INTEGER	From −2,147,483,647 to +2,147,483,647
SIGNED	From −2,147,483,647 to +2,147,483,647
UNSIGNED	From 0 to +2,147,483,647
User-defined integer type	Subset of INTEGER
User-defined enumerated type	Collection enumerated by user
SUBTYPE	Subset of any type (pre- or user-defined)
ARRAY	Single-type collection of any type above
RECORD	Multiple-type collection of any types above

دانشگاه صنعتی شاهرود
دانشکده برق

عنوان درس

Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

فصل ششم : کد ترتیبی

- تنها قسمتهایی از VHDL که بصورت ترتیبی اجرا میشوند شامل PROCESS، FUNCTION، و PROCEDURE هستند. البته هرکدام از اینها خود بطور موازی با دیگر اجزای بیرونی اجرا میشوند.
- از کدترتیبی میتوان برای پیاده سازی هم مدارات ترتیبی و هم مدارات ترکیبی استفاده کرد.
- به کدترتیبی، کدرفتاری هم گفته میشود.
- دستوراتی که در این فصل بحث میشوند (یعنی IF، WAIT، CASE و LOOP) و همچنین متغیرها (VARIABLE) فقط داخل سه عبارت مذکور (یعنی PROCESS، FUNCTION، و PROCEDURE) قابل استفاده میباشند.
- **یک فرق اساسی بین سیگنال و متغیر**: یک متغیر (برخلاف سیگنال) هرگز نمیتواند عمومی (global) باشد یعنی نمیتوان آن را مستقیما به بیرون از موجودیت انتقال داد.
- در این فصل فقط ***فرآیند*** (دستور PROCESS) توضیح داده میشود (بقیه در فصلهای بعدی).

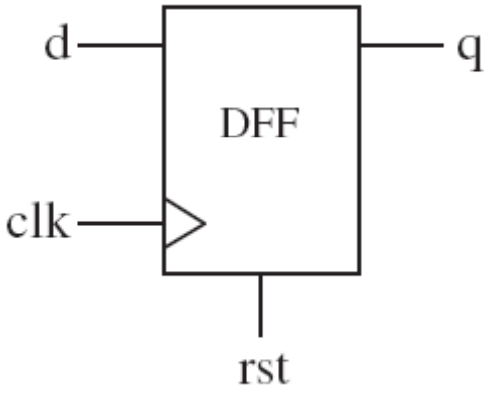
- **فرایند،** یک مجموعه دستورات با قالب مشخصی است که دارای یک لیست حساسیت بوده و دستورات مذکور فقط زمانی اجرا میشوند که یکی از سیگنالهای موجود در لیست حساسیت تغییر کند.
- در صورتی که در فرایند از دستور WAIT استفاده شود نیازی به لیست حساسیت نبوده و در صورت تحقق شرط مشخص شده در این دستور، بدنه فرایند اجرا میشود.
- یک مشخصه بارز فرایندها، وجود دستورات IF، WAIT، CASE و LOOP است.
- گرامر یک فرایند:

```
[label:] PROCESS (sensitivity list)
  [VARIABLE name type [range] [:= initial_value;]]
BEGIN
  (sequential code)
END PROCESS [label];
```

  - استفاده از متغیرها اختیاری بوده و در صورت استفاده از آنها، باید در قسمت اعلان فرایند، اعلان شوند.
  - مقدار اولیه متغیرها قابل سنتز نبوده و فقط در شبیه سازیها استفاده دارند.
  - استفاده از برچسب اختیاری بوده و به منظور افزایش خوانایی کد میباشند. یک برچسب شامل هر نامی به جز کلمات کلیدی میباشد.
- برای پیاده سازی مدارات سنکرون همواره لازم است یک سیگنال (مثلا کلاک) بررسی (و به اصطلاح ***مانیتور***) شود. یک راه متداول برای بررسی و آشکارسازی سیگنالها، استفاده از خاصیت EVENT است (مثال بعدی را ملاحظه کنید).

- مثال: یک DFF حساس به لبه بالارونده با ورودی ریست آسنکرون

```
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 ---------------------------------------
```



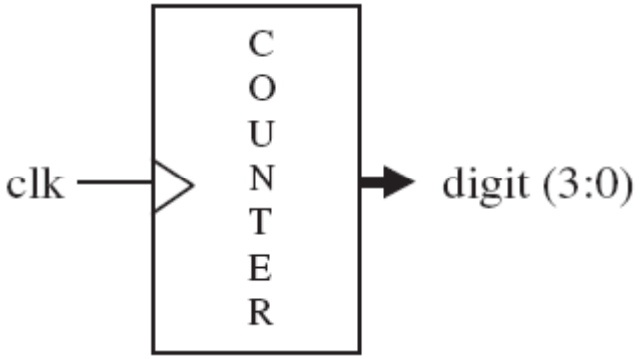
سیگنالها و متغیرها

- جزئیات بیشتر در فصل هفتم آورده خواهد شد.
- در VHDL برای انتقال مقادیر غیرثابت، دو راه وجود دارد: سیگنالها (دستور SIGNAL) و متغیرها (دستور VARIABLE) که در ادامه این دو با هم مقایسه میشوند.
- یک سیگنال را میتوان در یک بسته (PACKAGE)، یک موجودیت (ENTITY)، ویا در قسمت اعلان یک معماری (ARCHITECTURE) معرفی (یا اعلان) کرد؛ اما یک متغیر را میتوان فقط در قسمتی از یک کد ترتیبی (مثلا در یک فرایند) اعلان نمود.
- یک سیگنال میتواند عمومی باشد اما یک متغیر میتواند فقط محلی (local) باشد. یعنی، مقدار یک متغیر را نمیتوان مستقیما به بیرون از یک فرایند منتقل کردو درصورت لزوم باید مقدار آن را به یک سیگنال منتسب کرد.
- به روز شدن یک متغیر بلافاصله انجام میشود اما به روز شدن یک سیگنال پس از اتمام اجرای فعلی فرایند انجام میشود.
- عملگر انتساب برای یک سیگنال بصورت <= و برای یک متغیر بصورت := می باشد.

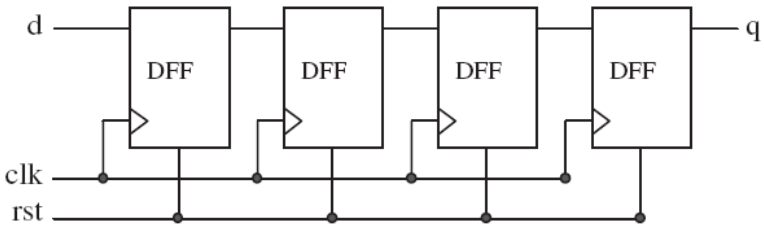
- دستور IF
  - گرامر دستور
  - مثال: یک شمارنده BCD یک رقمی بدون ورودی ریست

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

```
1  ------------------------------------------
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;
23 ------------------------------------------
```



- مثال: تحقق یک شیفت رجیستر چهاربیتی شامل ورودی ریست آسنکرون




```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY shiftreg IS
6     GENERIC (n: INTEGER := 4);     -- # of stages
7     PORT (d, clk, rst: IN STD_LOGIC;
8           q: OUT STD_LOGIC);
9  END shiftreg;
10 ------------------------------------------------
11 ARCHITECTURE behavior OF shiftreg IS
12    SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
13 BEGIN
14    PROCESS (clk, rst)
15    BEGIN
16       IF (rst='1') THEN
17          internal <= (OTHERS => '0');
18       ELSIF (clk'EVENT AND clk='1') THEN
19          internal <= d & internal(internal'LEFT DOWNTO 1);
20       END IF;
21    END PROCESS;
22    q <= internal(0);
23 END behavior;
24 ------------------------------------------------
```

- دستور WAIT
  - دستور WAIT مشابه دستور IF است اما دارای چندین فرم مختلف میباشد.
  - هرگاه داخل یک فرایند از دستور WAIT استفاده شود، آن فرایند نمیتواند شامل لیست حساسیت باشد.
  - این دستور دارای سه نوع گرامر است:

```
WAIT UNTIL signal_condition;

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

WAIT FOR time;
```

  - در **نوع اول**، فقط یک سیگنال میتواند مورد استفاده قرار بگیرد ولذا بیشتر مناسب مدارات سنکرون است تا آسنکرون. در این حالت، دستور WAIT UNTIL باید اولین دستور مورد استفاده در فرایند مربوطه باشد. هر بار که شرط تحقق یابد، فرآیند اجرا میشود.
  - مثال: یک نمونه ثبات ۸بیتی با ریست سنکرون

```
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;
```

- در **نوع دوم**، (یعنی WAIT ON) چندین سیگنال نیز قابل قبول است. در این حالت فرآیند همواره اجرا میشود مگر اینکه تغییری در یکی از سیگنالهای مشخص شده در دستور WAIT ON روی دهد.
- مثال: یک نمونه ثبات ۸بیتی با ریست آسنکرون

```
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;
```

- نوع سوم (یعنی WAIT FOR) فقط برای شبیه سازی (تولید شکل موج) و تولید بستر آزمایش (tetsbench) استفاده میشود.

- مثال: تحقق یک DFF با ورودی ریست آسنکرون

```
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 ------------------------------------
```

- مثال: تحقق یک شمارنده BCD یک رقمی

```
1  ------------------------------------------
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;
22 ------------------------------------------
```

- دستور CASE
  - این دستور هم برای تولید کدهای ترتیبی استفاده میشود.
  - گرامر این دستور:

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

  - مثال: (به کاربرد OTHERS در اینجا توجه کنید)

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

  - در اینگونه مواقع میتوان از UNAFFECTED و معادل آن، NULL، استفاده کرد:

```
 WHEN OTHERS => NULL;
```

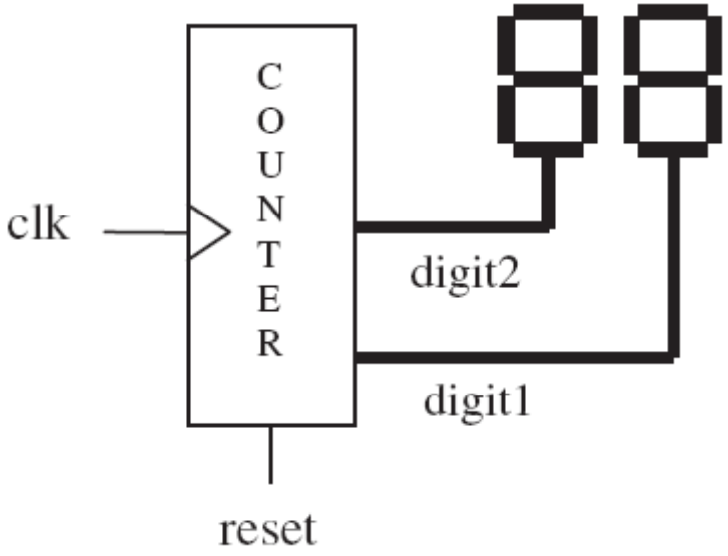
  - مشابه دستور WHEN در اینجا هم نحوه مشخص کردن مقدارها به سه صورت میتواند باشد.

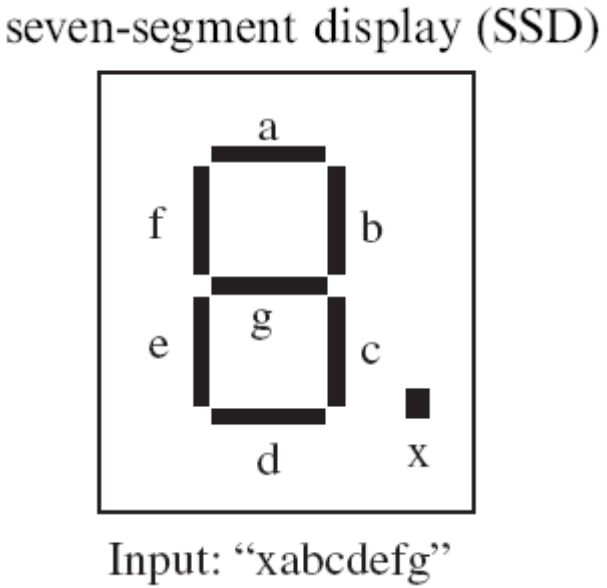
```
WHEN value                      -- single value
WHEN value1 to value2           -- range, for enumerated data types
                                -- only
WHEN value1 | value2 |...       -- value1 or value2 or ...
```

- مثال: تحقق یک DFF با ورودی ریست آسنکرون (توجه کنید که عمدا از تعدادی اعلان یا دستور غیرضروری به منظور نشان دادن نحوه انجام آنها استفاده شده است)

```
1  ---------------------------------------------
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;
27 ---------------------------------------------
```

- **مثال:** تحقق یک شمارنده BCD دو رقمی با ورودی ریست آسنکرون و تبدیل BCD به 7-segment (توجه کنید که در این مثال قسمتی از کد، دوبار تکرار شده است که بعدا خواهیم دید که برای جلوگیری از تکرار یک کد میتوان آن را در کتابخانه های تعریف شده توسط کاربر قرار داده و استفاده کرد)





```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY counter IS
6     PORT (clk, reset : IN STD_LOGIC;
7           digit1, digit2 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
8  END counter;
9  ------------------------------------------------
10 ARCHITECTURE counter OF counter IS
11 BEGIN
12    PROCESS(clk, reset)
13       VARIABLE temp1: INTEGER RANGE 0 TO 10;
14       VARIABLE temp2: INTEGER RANGE 0 TO 10;
15    BEGIN
16    ---- counter: ---------------------
17       IF (reset='1') THEN
18          temp1 := 0;
19          temp2 := 0;
20       ELSIF (clk'EVENT AND clk='1') THEN
21          temp1 := temp1 + 1;
22          IF (temp1=10) THEN
23             temp1 := 0;
24             temp2 := temp2 + 1;
25             IF (temp2=10) THEN
26                temp2 := 0;
```

```
27              END IF;
28           END IF;
29        END IF;
30        ---- BCD to SSD conversion: --------
31        CASE temp1 IS
32           WHEN 0 => digit1 <= "1111110";    --7E
33           WHEN 1 => digit1 <= "0110000";    --30
34           WHEN 2 => digit1 <= "1101101";    --6D
35           WHEN 3 => digit1 <= "1111001";    --79
36           WHEN 4 => digit1 <= "0110011";    --33
37           WHEN 5 => digit1 <= "1011011";    --5B
38           WHEN 6 => digit1 <= "1011111";    --5F
39           WHEN 7 => digit1 <= "1110000";    --70
40           WHEN 8 => digit1 <= "1111111";    --7F
41           WHEN 9 => digit1 <= "1111011";    --7B
42           WHEN OTHERS => NULL;
43        END CASE;
44        CASE temp2 IS
45           WHEN 0 => digit2 <= "1111110";    --7E
46           WHEN 1 => digit2 <= "0110000";    --30
47           WHEN 2 => digit2 <= "1101101";    --6D
48           WHEN 3 => digit2 <= "1111001";    --79
49           WHEN 4 => digit2 <= "0110011";    --33
50           WHEN 5 => digit2 <= "1011011";    --5B
51           WHEN 6 => digit2 <= "1011111";    --5F
52           WHEN 7 => digit2 <= "1110000";    --70
53           WHEN 8 => digit2 <= "1111111";    --7F
```

```
54          WHEN 9 => digit2 <= "1111011";     --7B
55          WHEN OTHERS => NULL;
56        END CASE;
57     END PROCESS;
58 END counter;
59 -------------------------------------------------
```

- دستور LOOP
  - جهت تکرار قسمتی از کد استفاده میشود.
  - این دستور نیز فقط در کدهای ترتیبی (یعنی داخل PROCESS، FUNCTION، و PROCEDURE) استفاده میشود.
  - دو فرم این دستور: FOR/LOOP و WHILE/LOOP
  - در فرم FOR/LOOP، یک حلقه به تعداد ثابت و مشخصی تکرار میشود:

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

  - در فرم WHILE/LOOP، حلقه تا زمانی تکرار میشود که شرط خاصی دیگر برقرار نباشد

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

- از EXIT برای مشخص کردن پایان حلقه استفاده میشود:

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

- از NEXT برای ردکردن گامهای حلقه استفاده میشود:

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

- مثال:

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

- توجه کنید که حدود مورد استفاده در FOR/LOOP باید ثابت باشند، بنابراین استفاده از دستوراتی مانند `FOR i IN 0 TO choice LOOP` که در آن choice یک پارامتر ورودی است، قابل سنتز نیست.
- مثال از WHILE/LOOP:

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

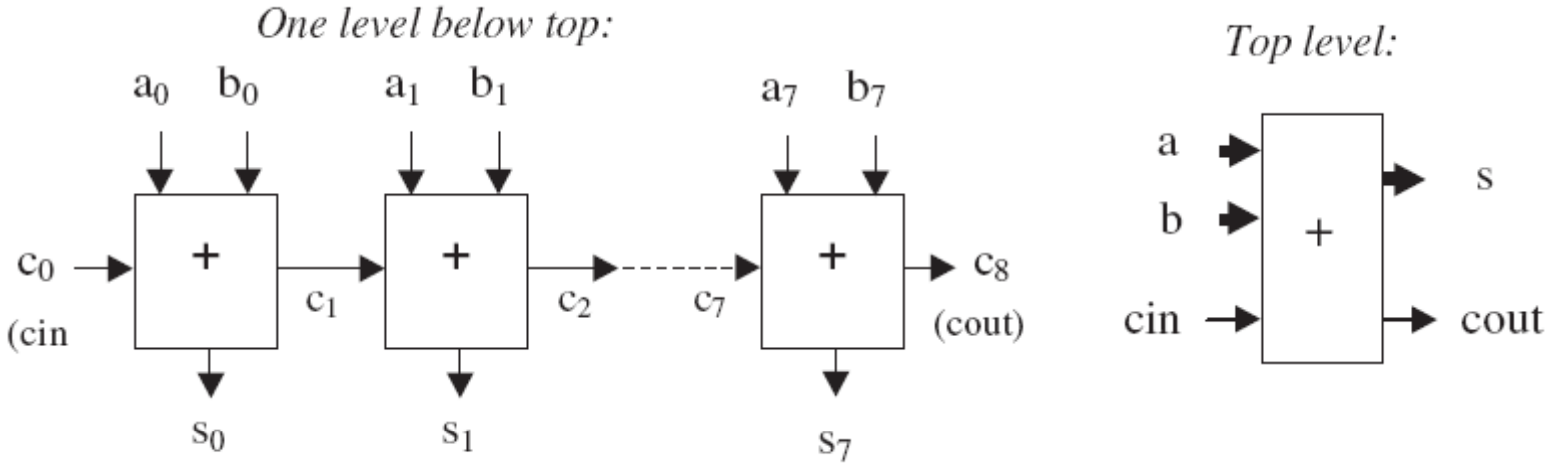
- مثال استفاده از EXIT (جهت توقف کامل حلقه):

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

- مثال از NEXT (صرفنظر از اجرای فعلی و رفتن به دور بعدی):

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

- مثال: تحقق یک جمع کننده بدون علامت ریپل به دو روش
  - روش اول: بصورت generic و استفاده از بردارها و FOR/LOOP
  - روش دوم: بصورت حالت خاص ۸بیتی و استفاده از اعداد صحیح و دستور IF



$$s_j = a_j \text{ XOR } b_j \text{ XOR } c_j$$

$$c_{j+1} = (a_j \text{ AND } b_j) \text{ OR } (a_j \text{ AND } c_j) \text{ OR } (b_j \text{ AND } c_j)$$

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

```
1  ---- Solution 2: non-generic, with INTEGERS ----
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY adder IS
6     PORT ( a, b: IN INTEGER RANGE 0 TO 255;
7            c0: IN STD_LOGIC;
8            s: OUT INTEGER RANGE 0 TO 255;
9            c8: OUT STD_LOGIC);
10 END adder;
11 ------------------------------------------------
12 ARCHITECTURE adder OF adder IS
13 BEGIN
14    PROCESS (a, b, c0)
15       VARIABLE temp : INTEGER RANGE 0 TO 511;
16    BEGIN
17       IF (c0='1') THEN temp:=1;
18       ELSE temp:=0;
19       END IF;
20       temp := a + b + temp;
21       IF (temp > 255) THEN
22          c8 <= '1';
23          temp := temp---256;
24       ELSE c8 <= '0';
25       END IF;
26       s <= temp;
27    END PROCESS;
28 END adder;
29 ------------------------------------------------
```

- مثال: شمارنده تعداد صفرهای متوالی بردار ورودی

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

- مقایسه IF و CASE
  - از نظر عملکرد سخت افزاری این دو معادل هم بوده و تفاوت چندانی ندارند.

```
---- With IF: --------------
IF (sel="00") THEN x<=a;
ELSIF (sel="01") THEN x<=b;
ELSIF (sel="10") THEN x<=c;
ELSE x<=d;
```

```
---- With CASE: ------------
CASE sel IS
   WHEN "00" => x<=a;
   WHEN "01" => x<=b;
   WHEN "10" => x<=c;
   WHEN OTHERS => x<=d;
END CASE;
---------------------------
```

- مقایسه CASE و WHEN :
  - دستور WHEN همزمان و دستور CASE ترتیبی است.
  - شباهتها و تفاوتها در جدول آورده شده است.

	WHEN	CASE
Statement type	Concurrent	Sequential
Usage	Only outside PROCESSES, FUNCTIONS, or PROCEDURES	Only inside PROCESSES, FUNCTIONS, or PROCEDURES
All permutations must be tested	Yes for WITH/SELECT/WHEN	Yes
Max. # of assignments per test	1	Any
No-action keyword	UNAFFECTED	NULL

- **کلاک دادن نامناسب**
  - در کدهایی که هم در لبه بالارونده و هم در لبه پایین رونده کلاک، یک سیگنال مقدار دهی میشود، قابلیت سنتز وجود ندارد زیرا تکنولوژی مربوطه (مثلا CPLD) عملا فقط به یکی از لبه های کلاک حساس است.
  - در مورد خاصیت EVENT باید حتما از یک حالت تست مشخص استفاده شود. بطور مثال دستور `IF(clk'EVENT AND clk='1')` صحیح است اما از دستوراتی مانند `IF(clk'EVENT)` باید خودداری کرد چرا که در اینصورت یا کامپایلر حالت خاصی را بصورت پیش فرض در نظر خواهد گرفت (مثلا `AND clk='1'`) یا اینکه پیغام خطا یا هشدار مناسبی خواهد داد.

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

  - اگر سیگنالی در لیست حساسیت یک فرآیند موجود باشد اما در هیچیک از تخصیصهای موجود در فرآیند استفاده نشود، احتمالا کامپایلر آن سیگنال را نادیده خواهد گرفت و پیغام هشدار مناسبی نمایش خواهد داد.

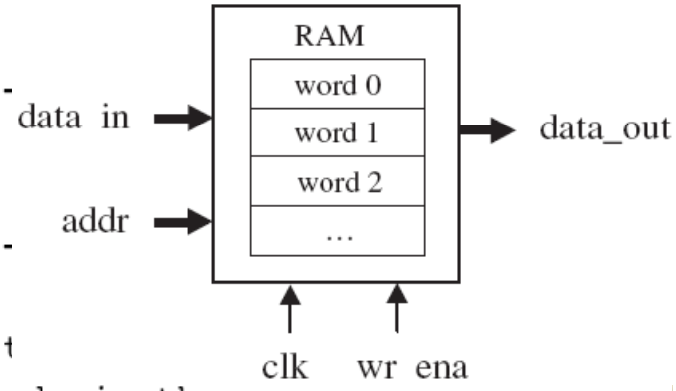
```
PROCESS (clk)
BEGIN
   counter := counter + 1;
   ...
END PROCESS;
```

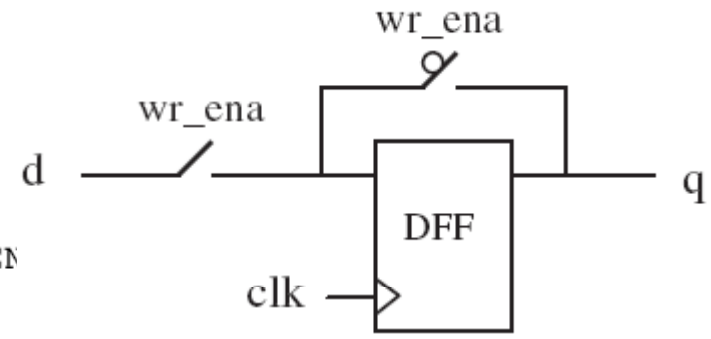
- در کد زیر علیرغم اینکه هم از لبه بالارونده و هم از لبه پایین رونده در دو فرآیند مجزا استفاده شده است، اما بدلیل استفاده از دو سیگنال مختلف (x و y) برای انجام تخصیصها، این کد همواره قابل سنتز است. (بنابراین در هر فرآیند میتوان از (فقط) یک لبه دلخواه کلاک استفاده کرد اما تخصیصهای این فرآیندها نباید روی سیگنال مشترکی اعمال شوند)

```
----------------------
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='1') THEN
      x <= d;
   END IF;
END PROCESS;
----------------------
PROCESS (clk)
BEGIN
   IF(clk'EVENT AND clk='0') THEN
      y <= d;
   END IF;
END PROCESS;
----------------------
```

- **مثال:** تحقق یک حافظه RAM (دیاگرام آن در شکل بالایی نشان داده شده است؛ در اینجا خروجی حافظه همواره عدد ذخیره شده در آدرس addr را منعکس میکند). از دیدگاه رجیستری، حافظه را میتوان معادل مدار شکل پایینی در نظر گرفت.

```
1  ---------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY ram IS
6  GENERIC ( bits: INTEGER := 8;      -- # of bit
7            words: INTEGER := 16);   -- # of words in the memory
8     PORT ( wr_ena, clk: IN STD_LOGIC;
9            addr: IN INTEGER RANGE 0 TO words-1;
10           data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
11           data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
12 END ram;
13 -------------------------------------------------
14 ARCHITECTURE ram OF ram IS
15    TYPE vector_array IS ARRAY (0 TO words-1) OF
16       STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
17    SIGNAL memory: vector_array;
18 BEGIN
19    PROCESS (clk, wr_ena)
20    BEGIN
21       IF (wr_ena='1') THEN
22          IF (clk'EVENT AND clk='1') THEN
23             memory(addr) <= data_in;
24          END IF;
25       END IF;
```





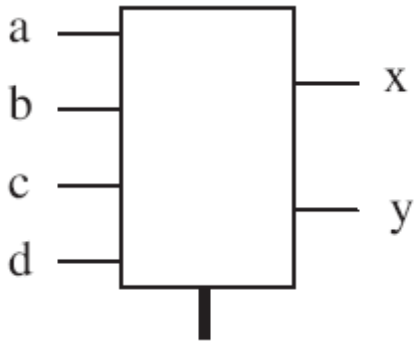
- استفاده از کدترتیبی برای پیاده سازی مدارات ترکیبی
  - در مدارات ترتیبی، استفاده از ثباتها الزامی است و کامپایلر از این نکته پی به ترتیبی بودن مدار میبرد؛ اما در حالت مدارات ترکیبی نباید از ثباتها استفاده کرد؛ همچنین، باید جدول صحت را بطور کامل در کد مشخص کرد.
  - دو قانون زیر را همیشه اجرا کنید:

۱. اطمینان حاصل کنید که تمام سیگنالهای ورودی که در فرآیند استفاده شده اند، حتما در لیست حساسیت ذکر شوند.

۲. اطمینان حاصل کنید که تمام ترکیبات ممکن سیگنالهای ورودی/خروجی در کد مشخص شده باشند.

  - عدم رعایت قانون ۱ مشکل خاصی ایجاد نکرده و معمولا یک پیغام هشدار تولید میشود. اما عدم رعایت قانون ۲ معمولا این تفسیر را در پی دارد که مدار جهت حفظ مقادیر خروجی خود حاوی ثبات میباشد.
  - مثال (طراحی نامناسب یک مدار ترکیبی): میخواهیم مدار ترکیبی a که جدول صحت آن در شکل b نشان داده شده است را پیاده سازی کنیم. اما همانطور که از جدول مذکور مشخص است تمام حالات ممکن y مشخص نشده است. اگر فقط همین خواسته ها و مشخصات داده شده را بخواهیم پیاده سازی کنیم، کد VHDL چنین خواهد شد:

```
1  ------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY example IS
6     PORT (a, b, c, d: IN STD_LOGIC;
7        sel: IN INTEGER RANGE 0 TO 3;
8        x, y: OUT STD_LOGIC);
9  END example;
10 ------------------------------------
11 ARCHITECTURE example OF example IS
12 BEGIN
13    PROCESS (a, b, c, d, sel)
14    BEGIN
15       IF (sel=0) THEN
16          x<=a;
17          y<='0';
18       ELSIF (sel=1) THEN
19          x<=b;
20          y<='1';
21       ELSIF (sel=2) THEN
22          x<=c;
23       ELSE
24          x<=d;
25       END IF;
26    END PROCESS;
27 END example;
28 ------------------------------------
```

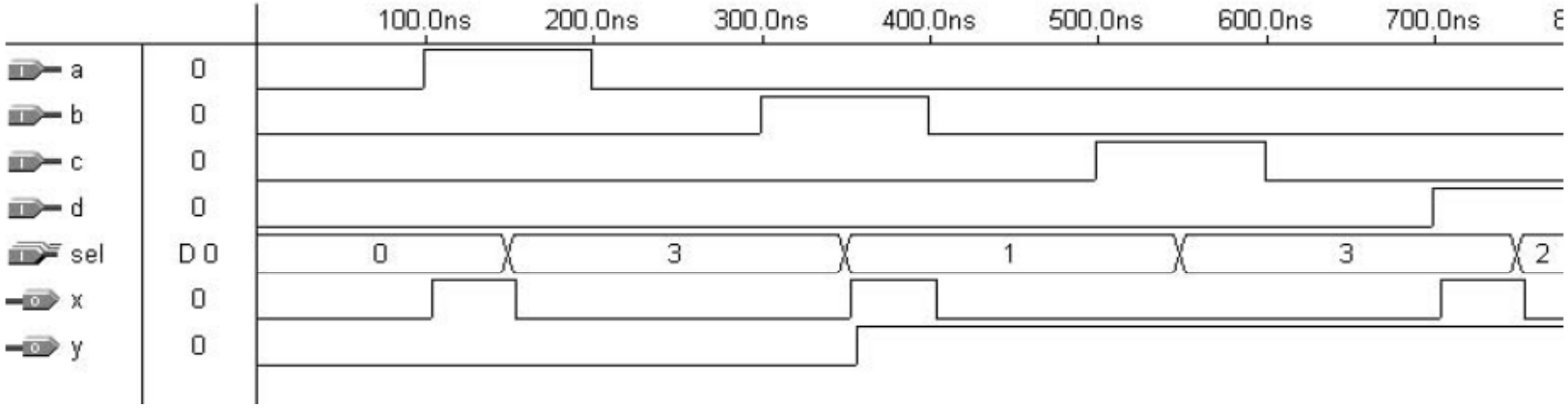


(a)

sel	x	y
00	a	0
01	b	1
10	c	
11	d	

(b)

- اگر به نتایج شبیه سازی دقت کنیم، ملاحظه میکنیم که علیرغم ترکیبی بودن مدار ، دو مقدار مختلف برای خروجی y و به ازاء یک مقدار مشخص ورودی انتخاب sel بدست آمده است:



- در اولین مورد، قبل از sel=3 ، sel=0 و در دومین مورد sel=1 بوده است. یعنی توسط کامپایلر نوعی حافظه در مدار بوجود آمده است. اگر به فرمول خروجی (که در نرم افزار Quartus بدست آمده است) دقت کنیم خواهیم دید:

$$y = (sel(0)\ AND\ sel(1))\ OR\ (sel(0)\ AND\ y)\ OR\ (sel(1)\ AND\ y)$$

یعنی مقدار خروجی به مقادیر قبلی خود نیز بستگی داشته و نیاز به حافظه بوجود آمده است.
در واقع مدار پیاده سازی شده دارای جدول صحت شکل c می باشد.

sel	x	y
00	a	0
01	b	1
10	c	y
11	d	y

(c)

- برای حل این مشکل باید جدول صحت را مطابق شکل d تغییر داد که در آن x به معنای بی اهمیت بودن (don’t care) می باشد. بنابراین در کد اخیر باید دستور `y<='x'` را بعد از خطوط ۲۲ و ۲۴ قرار داد.

sel	x	y
00	a	0
01	b	1
10	c	X
11	d	X

(d)

دانشگاه صنعتی شاهرود
دانشکده برق

عنوان درس

Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

فصل هفتم : سیگنالها و متغیرها

- برای مدیریت مقادیر غیرثابت دو راه وجود دارد: سیگنالها و متغیرها
- برای مدیریت مقادیر ثابت نیز دو راه وجود دارد: CONSTANT و GENERIC
- مقادیر SIGNAL و CONSTANT میتوانند عمومی باشند و همچنین در هر دو نوع کد ترتیبی و همزمان قابل استفاده اند.
- مقدار VARIABLE فقط میتواند محلی باشد و فقط داخل کد ترتیبی (یعنی در PROCESS، PROCEDURE و FUNCTION) استفاده شود. مقدار یک متغیر هرگز نمیتواند بطور مستقیم به بیرون منتقل شود.
- دستور CONSTANT:
  - گرامر این دستور:

```
CONSTANT name : type := value;
```

  - مثال:

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

- نوع CONSTANT را میتوان (مشابه SIGNAL) داخل PACKAGE، ENTITY و یا ARCHITECTURE اعلان کرد:
  - اگر داخل بسته اعلان شود، عمومی بوده و توسط دیگر موجودیتها قابل استفاده است.
  - اگر داخل یک موجودیت (بعد از PORT) اعلان شود برای تمام معماریهایی که بعد از آن موجودیت آمده اند، عمومی است.
  - اگر داخل بخش اعلان یک معماری اعلان شود، فقط برای کد مربوط به آن معماری، عمومی است.
  - متداولترین محل برای اعلان یک CONSTANT داخل یک معماری و یا داخل یک بسته است.
- نوع سیگنال (SIGNAL)
  - نوع سیگنال به انتقال مقادیر **به درون یا بیرون از** یک مدار و نیز **بین اجزای داخلی** آن مدار کمک میکند. ب هعبارت دیگر یک سیگنال معرف اتصالات موجود در یک مدار است (یعنی همان سیم فیزیکی است).
  - تمام پورتهای یک موجودیت بطور پیش فرض از نوع سیگنال میباشند.
  - محلهای ممکن برای اعلان یک یبگنال مشابه با نوع CONSTANT است.
  - گرامر اعلان یک سیگنال:

```
SIGNAL name : type [range] [:= initial_value];
```

  - مثال:

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

- یک نکته مهم در مورد سیگنالهایی که داخل کدهای ترتیبی (بطور مثال داخل یک فرآیند) استفاده میشوند این است که تغییرات آنها بلافاصله انجام نمیشود بلکه پس از اجرای نوبت فعلی فرآیند (یا تابع یا زیرروال) به روز میشود.
- عملگر انتساب سیگنالها => بوده و مقادیر اولیه فقط به منظور شبیه سازی استفاده میشوند و قابلیت سنتز ندارند.
- انتسابهای چندگانه به یک سیگنال غیرمجاز است؛ به مثال توجه کنید:
- **مثال ۱:** هدف طراحی یک مدار برای شمارش تعداد ۱ها در یک بردار ورودی است. برای اینکار از کد زیر استفاده میکنیم که فقط از سیگنالها استفاده کرده است؛ در حقیقت، هدف این است که نشان داده شود متغیرها هم گاهی لازم و ضروری هستند و سیگنالها همیشه نمیتوانند جای آنها را بگیرند. در این کد خطوط ۱۵ و ۱۸ **همزمان** سعی در تغییر سیگنال temp دارند زیرا تغییرات خط ۱۵ تا پایان نوبت اجرای فعلی فرآیند، اعمال نمیشود. در اینگونه مواقع باید از متغیرها استفاده کنیم.
- همچنین در این مثال نیازی به استفاده از سیگنال واسط temp نداریم و میتوانیم مستقیما از ones به جای temp استفاده کنیم اما در این حالت باید توجه داشته باشیم که خروجی ones را باید از نوع BUFFER تعریف کنیم (و نه OUT) زیرا هم به آن مقداری نسبت میدهیم و هم مقدار آن را در داخل معماری استفاده میکنیم.
- حل مشکل مثال ۱ در مثال ۲ انجام میشود.

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY count_ones IS
6     PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8);
8  END count_ones;
9  ----------------------------------------
10 ARCHITECTURE not_ok OF count_ones IS
11    SIGNAL temp: INTEGER RANGE 0 TO 8;
12 BEGIN
13    PROCESS (din)
14    BEGIN
15       temp <= 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i)='1') THEN
18             temp <= temp + 1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END not_ok;
24 ----------------------------------------
```

- نوع متغیر (VARIABLE)
  - فقط داخل کد ترتیبی (یعنی داخل PROCESS، PROCEDURE، و FUNCTION) قابل استفاده است.
  - به روز رسانی آن بلافاصله است و در خطوط بعدی میتوان از آن استفاده کرد.
  - اعلان یک متغیر فقط باید در بخش اعلان PROCESS، PROCEDURE، و FUMCTION باشد.
  - نحوه اعلان یک متغیر
  - مثال:

```
VARIABLE name : type [range] [:= init_value];

VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

  - عملگر انتساب برای متغیرها، := می باشد.
  - دادن مقدار اولیه فقط برای شبیه سازی مفید بوده و قابلیت سنتز ندارد.
  - **مثال ۲:** همان مثال ۱ را مجددا بررسی میکنیم (مدار شمارشگر تعداد ۱ها). تنها تفاوت این مثال با مثال قبلی این است که در آن به جای تعریف سیگنال، از تعریف VARIABLE استفاده شده است. در این حالت، بر خلاف مثال ۱، کد حاصله معتبر و صحیح است.

```
1  ---------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY count_ones IS
6     PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            ones: OUT INTEGER RANGE 0 TO 8);
8  END count_ones;
9  ---------------------------------------
10 ARCHITECTURE ok OF count_ones IS
11 BEGIN
12    PROCESS (din)
13       VARIABLE temp: INTEGER RANGE 0 TO 8;
14    BEGIN
15       temp := 0;
16       FOR i IN 0 TO 7 LOOP
17          IF (din(i)='1') THEN
18             temp := temp + 1;
19          END IF;
20       END LOOP;
21       ones <= temp;
22    END PROCESS;
23 END ok;
24 ---------------------------------------
```

- خلاصه مهمترین تفاوتهای بین سیگنال و متغیر

Comparison between SIGNAL and VARIABLE.

	SIGNAL	VARIABLE
Assignment	<=	:=
Utility	Represents circuit interconnects (wires)	Represents local information
Scope	Can be global (seen by entire code)	Local (visible only inside the corresponding PROCESS, FUNCTION, or PROCEDURE)
Behavior	Update is not immediate in sequential code (new value generally only available at the conclusion of the PROCESS, FUNCTION, or PROCEDURE)	Updated immediately (new value can be used in the next line of code)
Usage	In a PACKAGE, ENTITY, or ARCHITECTURE. In an ENTITY, all PORTS are SIGNALS by default	Only in sequential code, that is, in a PROCESS, FUNCTION, or PROCEDURE

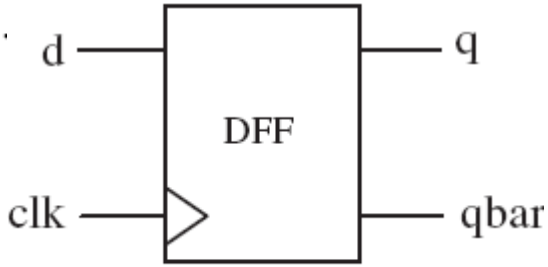
- **مثال:** دو روش برای پیاده سازی یک مالتی پلکسر؛ یکی مبتنی بر سیگنال برای حمل اطلاعات میانی (روش نامناسب) و دیگری استفاده از متغیر برای حمل اطلاعات میانی (روش مناسب)

```
1  -- Solution 1: using a SIGNAL (not ok) --
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 not_ok OF mux IS
11    SIGNAL sel : INTEGER RANGE 0 TO 3;
12 BEGIN
13    PROCESS (a, b, c, d, s0, s1)
14    BEGIN
15       sel <= 0;
16       IF (s0='1') THEN sel <= sel + 1;
17        END IF;
18        IF (s1='1') THEN sel <= sel + 2;
19        END IF;
20        CASE sel IS
21           WHEN 0 => y<=a;
22           WHEN 1 => y<=b;
23           WHEN 2 => y<=c;
24           WHEN 3 => y<=d;
25        END CASE;
26    END PROCESS;
27 END not_ok;
28 ---------------------------------------
```

```
1  -- Solution 2: using a VARIABLE (ok) ----
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 ok OF mux IS
11 BEGIN
12    PROCESS (a, b, c, d, s0, s1)
13       VARIABLE sel : INTEGER RANGE 0 TO 3;
14    BEGIN
15       sel := 0;
16       IF (s0='1') THEN sel := sel + 1;
17       END IF;
18       IF (s1='1') THEN sel := sel + 2;
19       END IF;
20       CASE sel IS
21          WHEN 0 => y<=a;
22          WHEN 1 => y<=b;
23          WHEN 2 => y<=c;
24          WHEN 3 => y<=d;
25       END CASE;
26    END PROCESS;
27 END ok;
28 ---------------------------------------
```

- مثال: پیاده سازی یک DFF

```
1  ---- Solution 1: not OK ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ---------------------------------------
11 ARCHITECTURE not_ok OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          q <= d;
17          qbar <= NOT q;
18       END IF;
19    END PROCESS;
20 END not_ok;
21 ---------------------------------------
```



```
1  ---- Solution 2: OK -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ---------------------------------------
11 ARCHITECTURE ok OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          q <= d;
17       END IF;
18    END PROCESS;
19    qbar <= NOT q;
20 END ok;
21 ---------------------------------------
```

- در راه حل ۱، چون دو دستور مربوط به q و qbar همزمان شروع به اجرا میکنند، بنابراین مقدار qbar برابر متمم مقدار ***قبلی*** q خواهد بود نه مقدار ***جدید***. بنابراین مقدار صحیح qbar همیشه یک دوره کلاک عقب تر از q خواهد بود. این مشکل در راه حل ۲ برطرف شده است.

- **کامپایلر در چه صورت پی به وجود ثبات میبرد؟**
  - هرگاه در زمان تغییر (transition) یک سیگنال، یک تخصیص (یا همان انتساب) به یک سیگنال اعمال شود، وجود (حداقل یک) فلیپ فلاپ نتیجه میشود. چنین تخصیصی (که به طور سنکرون با سیگنال دیگری انجام میشود) فقط میتواند داخل یک PROCESS، FUNCTION، و یا PROCEDURE انجام شود (و معمولا پس از وجود دستوری مانند 'IF signal'EVENT. یا 'WAIT UNTIL ...').
  - یک متغیر تا زمانیکه مقدار آن به خارج از PROCESS، FUNCTION، و یا PROCEDURE منتقل نشده باشد موجب تولید فلیپ فلاپ نمیشود. اما اگر در هنگام تغییرات (transition) ی کسیگنال، مقداری به یک متغیر نسبت داده شود و مقدار این متغیر به سیگنالی نسبت داده شود (که نهایتا از فرآیند خارج شود) آنگاه حداقل یک فلیپ فلاپ تولید خواهد شد.
  - اگر از متغیری قبل از آن که مقداری به آن نسبت داده شود، استفاده کنیم فلیپ فلاپ تولید خواهد شد.

- **مثال:** در کد زیر هر دو خروجی output1 و output2 ذخیره میشوند؛ یعنی، به دو فلیپ فلاپ نیاز داریم. علت این است که در حین تغییر (گذر) یک سیگنال دیگر (در اینجا سیگنال clk) روی این سیگنالها یک تخصیص اعمال میشود.

```
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk='1') THEN
      output1 <= temp;     -- output1 stored
      output2 <= a;        -- output2 stored
   END IF;
END PROCESS;
```

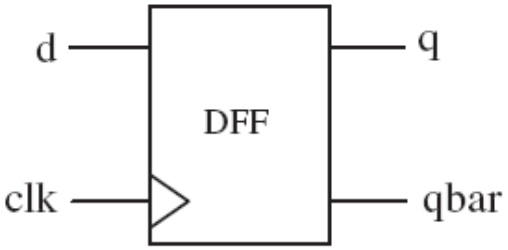
- **مثال:** در کد زیر فقط خروجی output1 ذخیره میشود و برای خروجی output2 فقط از گیتهای منطقی ترکیبی استفاده میشود.

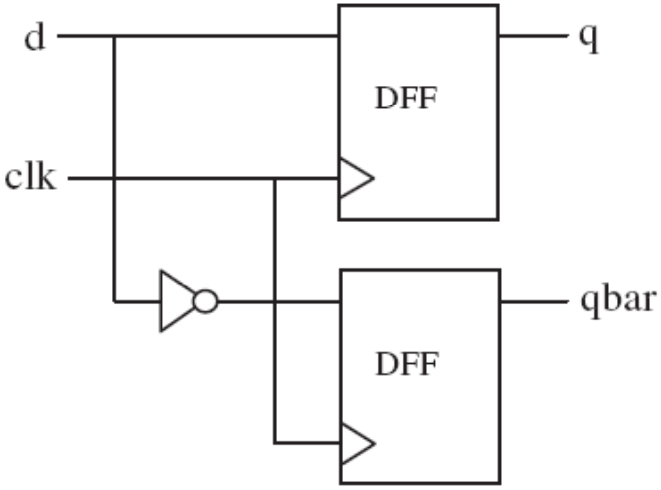
```
PROCESS (clk)
BEGIN
   IF (clk'EVENT AND clk='1') THEN
      output1 <= temp;     -- output1 stored
   END IF;
   output2 <= a;           -- output2 not stored
END PROCESS;
```

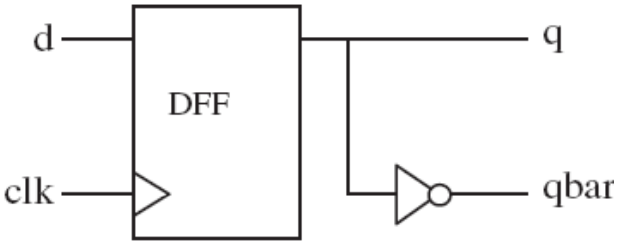
- **مثال:** در کد زیر، یک متغیر (temp) باعث ذخیره یک سیگنال (x) میشود.

```
PROCESS (clk)
   VARIABLE temp: BIT;
BEGIN
   IF (clk'EVENT AND clk='1') THEN
      temp <= a;
   END IF;
   x <= temp;     -- temp causes x to be stored
END PROCESS;
```

- **مثال:** در این مثال دو روش (صحیح) برای پیاده سازی همان DFF قبلی ارائه میشود. تفاوت این دو روش در تعداد فلیپ فلاپهایی است که مصرف میکنند. راه حل اول شامل دو تخصیص همزمان سیگنالها است (خطوط ۱۶ و ۱۷) لذا دو فلیپ فلاپ تولید میشود. اما در راه حل دوم، یکی از تخصیصها دیگر همزمان انجام نمیشود (خط ۱۹).
- مدارات مربوط به هرکدام از این راه حلها در پایین نشان داده شده است.



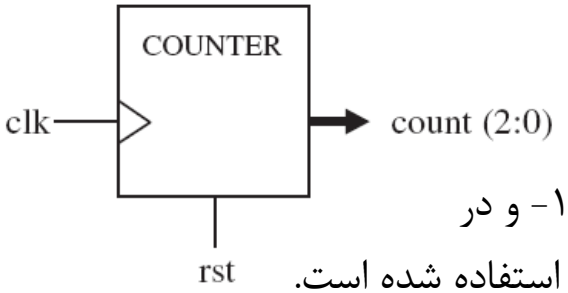




```
1  ---- Solution 1: Two DFFs ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ----------------------------------------
11 ARCHITECTURE two_dff OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          q <= d;           -- generates a register
17          qbar <= NOT d;    -- generates a register
18       END IF;
19    END PROCESS;
20 END two_dff;
21 ----------------------------------------
```

```
1  ---- Solution 2: One DFF ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY dff IS
6     PORT ( d, clk: IN STD_LOGIC;
7            q: BUFFER STD_LOGIC;
8            qbar: OUT STD_LOGIC);
9  END dff;
10 ----------------------------------------
11 ARCHITECTURE one_dff OF dff IS
12 BEGIN
13    PROCESS (clk)
14    BEGIN
15       IF (clk'EVENT AND clk='1') THEN
16          q <= d;      -- generates a register
17       END IF;
18    END PROCESS;
19    qbar <= NOT q;    -- uses logic gate (no register)
20 END one_dff;
21 ----------------------------------------
```

- **مثال:** تحقق یک شمارنده ۰ تا ۷



  - در این مثال دو روش برای پیاده سازی استفاده شده است.

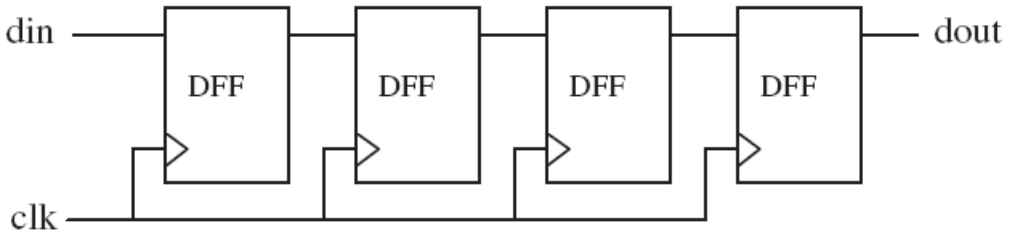
در اولی دو تخصیص همزمان متغیر (VARIABLE)- خطور ۱۴ و ۱۵- و در دومی دو تخصیص همزمان سیگنال (SIGNAL)- خطوط ۱۳و ۱۴- استفاده شده است.

  - هر دو روش منجر به استفاده از سه فلیپ فلاپ میشوند که برای ذخیره هرکدام از ارقام خروجی استفاده میشوند. روش اول در حقیقت نشان میدهد که یک متغیر میتواند منجر به تولید فلیپ فلاپ شود؛ دلیل این مطلب این است که تخصیص آن در لحظه تغییر یک سیگنال دیگر (در اینجا clk در خط ۱۴) انجام شده و نیز مقدار آن به بیرون از فرآیند انتقال داده میشود(خط ۱۷).
  - روش دوم فقط از سیگنالها کمک گرفته است. توجه کنید که چون از هیچ سیگنال واسط (یا کمکی) استفاده نشده است، خروجی count از نوع BUFFER تعریف شده است (خط ۴) زیرا هم به آن مقداری تخصیص داده شده است و هم اینکه از آن به صورت داخلی استفاده شده است.
  - نکته بعدی در مورد روش دوم این است که ***داخل یک کد ترتیبی*** برای سیگنالها هم، مشابه با متغیرها، میتوان از عمل افزایش استفاده کرد.
  - نکته آخر این که در هیچیک از دو روش مذکور از بسته std_ogic_1164 استفاده نشده است زیرا از هیچ نوع داده ای که متعلق به این بسته باشد، در این روشها استفاده نشده است.

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

```
1  ------ Solution 2: With SIGNALS only -------
2  ENTITY counter IS
3     PORT ( clk, rst: IN BIT;
4            count: BUFFER INTEGER RANGE 0 TO 7);
5  END counter;
6  -----------------------------------------
7  ARCHITECTURE counter OF counter IS
8  BEGIN
9     PROCESS (clk, rst)
10    BEGIN
11       IF (rst='1') THEN
12          count <= 0;
13       ELSIF (clk'EVENT AND clk='1') THEN
14          count <= count + 1;
15       END IF;
16    END PROCESS;
17 END counter;
18 -----------------------------------------
```

- **مثال:** تحقق یک شیفت رجیستر



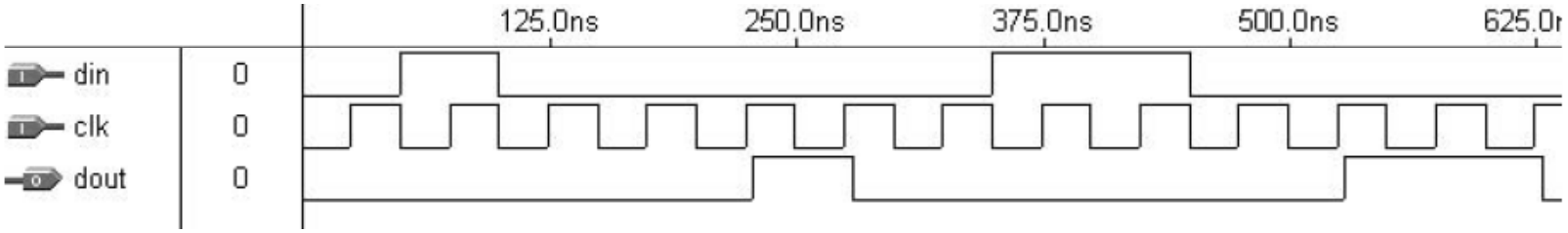
  - در این مثال از دو روش مختلف (سیگنالها و متغیرها) برای تحقق یک شیفت رجیستر استفاده شده است.
  - در روش اول، از سه متغیر (a، b، و c) در خط ۱۰ استفاده شده است. متغیرها قبل از آن که مقداری به آنها نسبت داده شود، مورد استفاده قرار گرفته اند (تخصیصها در خطوط ۱۳ تا ۱۶و به ترتیب معکوس یعنی از dout به din انجام شده اند)؛ در نتیجه، باعث تولید فلیپ فلاپهایی میشوند تا مقادیر حاصل از اجرای قبلی فرآیند را در خود نگه دارند.
  - در روش دوم، از سیگنالها استفاده شده و تخصیصها به ترتیب مستقیم (یعنی از din به dout) انجام شده اند. در اینجا هم به دلیل اینکه تخصیصها در لحظه تغییر یک سیگنال دیگر انجام شده اند، سه فلیپ فلاپ تولید میشود.
  - علاوه بر دو روش مذکور (که هر دو صحیح میباشند)، یک روش سوم هم ارائه شده است که از همان متغیرهای روش اول استفاده میکند اما تخصیصها را به صورت مستقیم انجام میدهد (خطوط ۱۳ تا ۱۶). از آنجا که تغییر متغیرها بلافاصله انجام میشود و در خطوط ۱۳ تا ۱۵ تخصیصها بطور مستقیم انجام شده اند، این سه خط معادل یک خط c:=din میباشند. در خط ۱۶ مقدار متغیر c فرآیند را ترک میکند. تخصیص خط ۱۶ در لحظه تغییر یک سیگنال دیگر انجام گرفته که بنابراین موجب تولید یک فلیپ فلاپ و در نتیجه تولید نتیجه نادرست میگردد.

```
1  -------- Solution 1: -----------------
2  ENTITY shift IS
3     PORT ( din, clk: IN BIT;
4            dout: OUT BIT);
5  END shift;
6  ---------------------------------------
7  ARCHITECTURE shift OF shift IS
8  BEGIN
9     PROCESS (clk)
10       VARIABLE a, b, c: BIT;
11    BEGIN
12       IF (clk'EVENT AND clk='1') THEN
13          dout <= c;
14          c := b;
15          b := a;
16          a := din;
17       END IF;
18    END PROCESS;
19 END shift;
20 ---------------------------------------
```

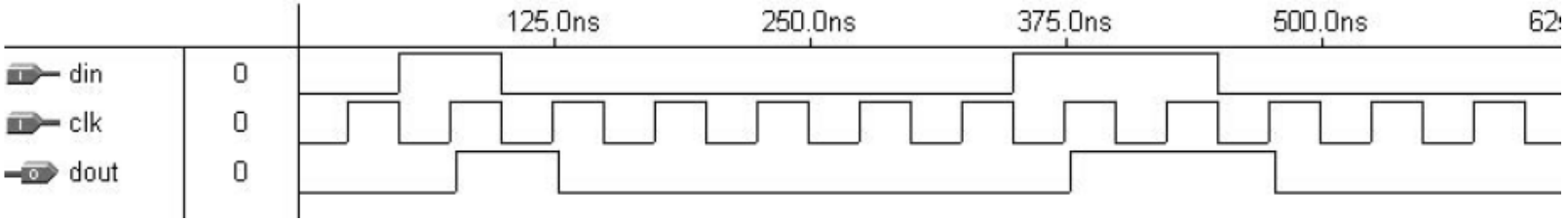
```
1  -------- Solution 2: -----------------
2  ENTITY shift IS
3     PORT ( din, clk: IN BIT;
4            dout: OUT BIT);
5  END shift;
6  ---------------------------------------
7  ARCHITECTURE shift OF shift IS
8     SIGNAL a, b, c: BIT;
9  BEGIN
10    PROCESS (clk)
11    BEGIN
12       IF (clk'EVENT AND clk='1') THEN
13          a <= din;
14          b <= a;
15          c <= b;
16          dout <= c;
17       END IF;
18    END PROCESS;
19 END shift;
20 ---------------------------------------
```

```
1  -------- Solution 3: -----------------
2  ENTITY shift IS
3     PORT ( din, clk: IN BIT;
4            dout: OUT BIT);
5  END shift;
6  ---------------------------------------
7  ARCHITECTURE shift OF shift IS
8  BEGIN
9     PROCESS (clk)
10       VARIABLE a, b, c: BIT;
11    BEGIN
12       IF (clk'EVENT AND clk='1') THEN
13          a := din;
14          b := a;
15          c := b;
16          dout <= c;
17       END IF;
18    END PROCESS;
19 END shift;
20 ---------------------------------------
```

- نتیجه روش اول و دوم

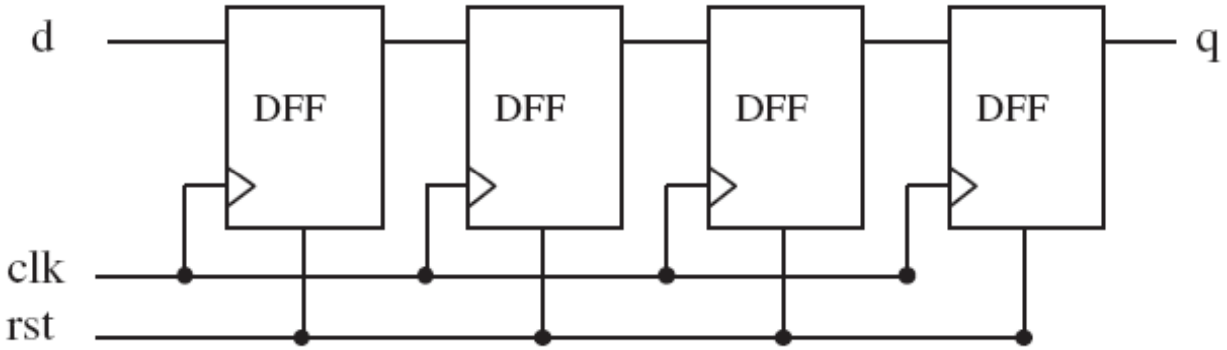


- نتیجه روش سوم



- همانطور که ملاحظه میشود در روشهای اول و دوم، خروجی dout همیشه چهار پالس کلاک عقبتر از ورودی din است اما در روش سومی فقط یک پالس کلاک عقبتر است که نتیجه درستی نیست.

- **مثال:** راه متداول برای پیاده سازی شیفت رجیستر با ورودی ریست آسنکرون



  - در اینجا هم انتظار داریم بیت خروجی چهار پالس کلاک عقبتر از بیت ورودی باشد.
  - دو روش یکی با استفاده از سیگنالها و دیگری با استفاده از متغیرها برای پیاده سازی این شیفت رجیستر ارائه شده است. در هر دو چهار فلیپ فلاپ تولید خواهند شد. در روش اول، بدلیل انجام تخصیصها در لحظه تغییر یک سیگنال دیگر(خطوط ۱۷ و ۱۸)، فلیپ فلاپها تولید میشوند. در روش دوم، تخصیص متغیرها در لحظه تغییر یک سیگنال دیگر انجام میشود (خطوط ۱۷ و ۱۸) اما چون مقادیر متغیرها فرآیند را ترک میکنند (خط ۲۰)، بنابراین موجب تولید فلیپ فلاپ میشوند.

```
1  ---- Solution 1: With an internal SIGNAL ---
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY shiftreg IS
6     PORT ( d, clk, rst: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END shiftreg;
9  ---------------------------------------
10 ARCHITECTURE behavior OF shiftreg IS
11    SIGNAL internal: STD_LOGIC_VECTOR (3 DOWNTO 0);
12 BEGIN
13    PROCESS (clk, rst)
14    BEGIN
15       IF (rst='1') THEN
16          internal <= (OTHERS => '0');
17       ELSIF (clk'EVENT AND clk='1') THEN
18          internal <= d & internal(3 DOWNTO 1);
19       END IF;
20    END PROCESS;
21    q <= internal(0);
22 END behavior;
23 ---------------------------------------
```

```
1  -- Solution 2: With an internal VARIABLE ---
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY shiftreg IS
6     PORT ( d, clk, rst: IN STD_LOGIC;
7            q: OUT STD_LOGIC);
8  END shiftreg;
9  ---------------------------------------------
10 ARCHITECTURE behavior OF shiftreg IS
11 BEGIN
12    PROCESS (clk, rst)
13       VARIABLE internal: STD_LOGIC_VECTOR (3 DOWNTO 0);
14    BEGIN
15       IF (rst='1') THEN
16          internal := (OTHERS => '0');
17       ELSIF (clk'EVENT AND clk='1') THEN
18          internal := d & internal(3 DOWNTO 1);
19       END IF;
20       q <= internal(0);
21    END PROCESS;
22 END behavior;
23 ---------------------------------------------
```

دانشگاه صنعتی شاهرود
دانشکده برق

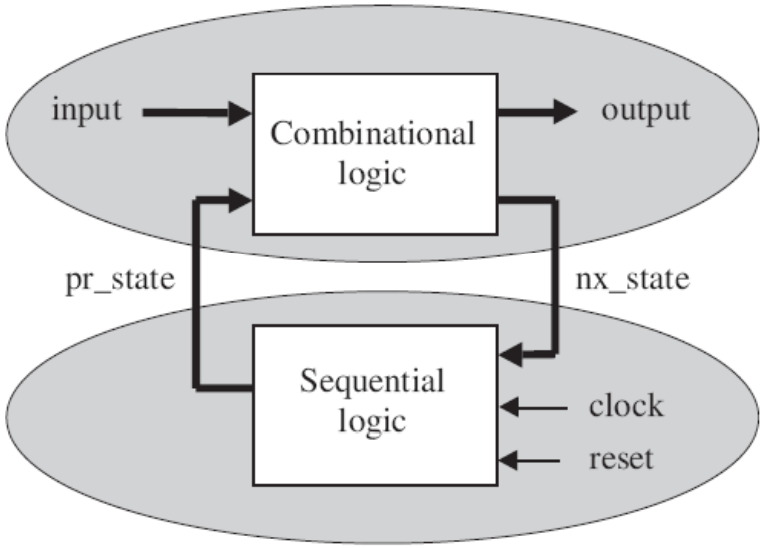
عنوان درس

Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

فصل هشتم : ماشینهای حالت

- ماشین حالت محدود (FSM) مدل خاصی برای توصیف رفتار مدارات ترتیبی است به ویژه آن مداراتی که رفتار آنها در قالب یک دنباله حالت مشخص قابل توصیف باشد.
- ماشین حالت تک فاز (Single Phase)
  - دیاگرام مربوط: همانطور که ملاحظه میشود اینگونه مدارات به دو بخش بالایی و پایینی قابل تجزیه هستند که بخش بالایی یک مدار ترکیبی و بخش پایینی یک مدار ترتیبی است.
  - بخش ترکیبی **دو نوع ورودی** دارد: حالت فعلی (pr_state) و ورودی(های) ناشی از دنیای خارج؛ همچنین، شامل **دو نوع خروجی** حالت بعدی (nx_state) و خروجی(های) مدار میباشد.



- گرچه میتوان برای پیاده سازی قسمت ترکیبی از کدهمزمان و برای پیاده سازی قسمت ترتیبی میتوان از کد ترتیبی استفاده کرد اما از آنجا که کد ترتیبی برای پیاده سازی هر دو نوع مدار قابل استفاده است، میتوان با یک کد ترتیبی (مثلا به کمک یک فرآیند) کل مدار را پیاده سازی کرد.
- در حالت معمول، سیگنالهای ریست و کلاک در لیست حساسیت فرآیند مربوط به بلوک پایینی شکل اخیر قرار داده میشوند (مگر اینکه ریست از نوع سنکرون باشد یا اینکه استفاده نشود یا اینکه به جای IF از WAIT استفاده شود).
- حسن جداسازی مدار ترتیبی به صورت بخش بالایی و بخش پایینی در این است که کدنویسی را هم میتوانیم به دو بخش تقسیم کنیم.
- **تعریف ماشین میلی (Mealy)**: خروجی ماسین نه تنها به حالت فعلی بلکه به ورودی ماشین هم بستگی دارد.
- **تعریف ماشین مور (Moore)**: خروجی ماشین فقط به حالت فعلی بستگی دارد.
- توجه کنید که مدل کردن مدارات ترتیبی به صورت ماشین حالت همواره مناسب و بهینه نیست زیرا باعث پیچیده کردن و حجیم کردن کد میشود؛ به عبارت دیگر، برای برخی مدارات ترتیبی بهتر است به صورت معمول و متداول عمل کنیم تا اینکه آن را به صورت نشان داده شده در شکل اخیر مدل کنیم. یک نمونه از چنین مداراتی شامل شمارنده ها میباشد. یک نمونه از مداراتی که برای آنها بهتر است از مدل ماشین حالت استفاده کنیم، کنترلرهای دیجیتال هستند که در آنها وظیفه سیستم را میتوان به صورت یک لیست مشخص و قابل شمارش از حالتهای مختلف توصیف کرد.

- در این فصل دو شیوه طراحی برای توصیف ماشینهای حالت ارائه میشود.

شیوه طراحی شماره ۱

- در این شیوه، طراحی بخش پایینی شکل اخیر کاملا مجزا از بخش بالایی انجام میشود. تمام حالات ممکن ماشین بطور صریح در قالب داده نوع شمارش شده (enumerated) اعلان میشوند.
- **طراحی بخش پایینی** (بخش ترتیبی): ورودیهای این بخش شامل کلاک (clk)، ریست (reset)، و حالت بعدی (nx_state) و خروجی نیز شامل حالت فعلی (pr_state) میباشد.
- چون میخواهیم یک مدار ترتیبی را پیاده سازی کنیم، باید از یک فرآیند (process) استفاده کنیم. تا داخل آن بتوانیم از IF، WAIT، CASE، و یا LOOP استفاده کنیم.
- قالب طراحی بخش پایینی در شبه کد زیر آمده است.

```
PROCESS (reset, clock)
BEGIN
   IF (reset='1') THEN
      pr_state <= state0;
   ELSIF (clock'EVENT AND clock='1') THEN
      pr_state <= nx_state;
   END IF;
END PROCESS;
```

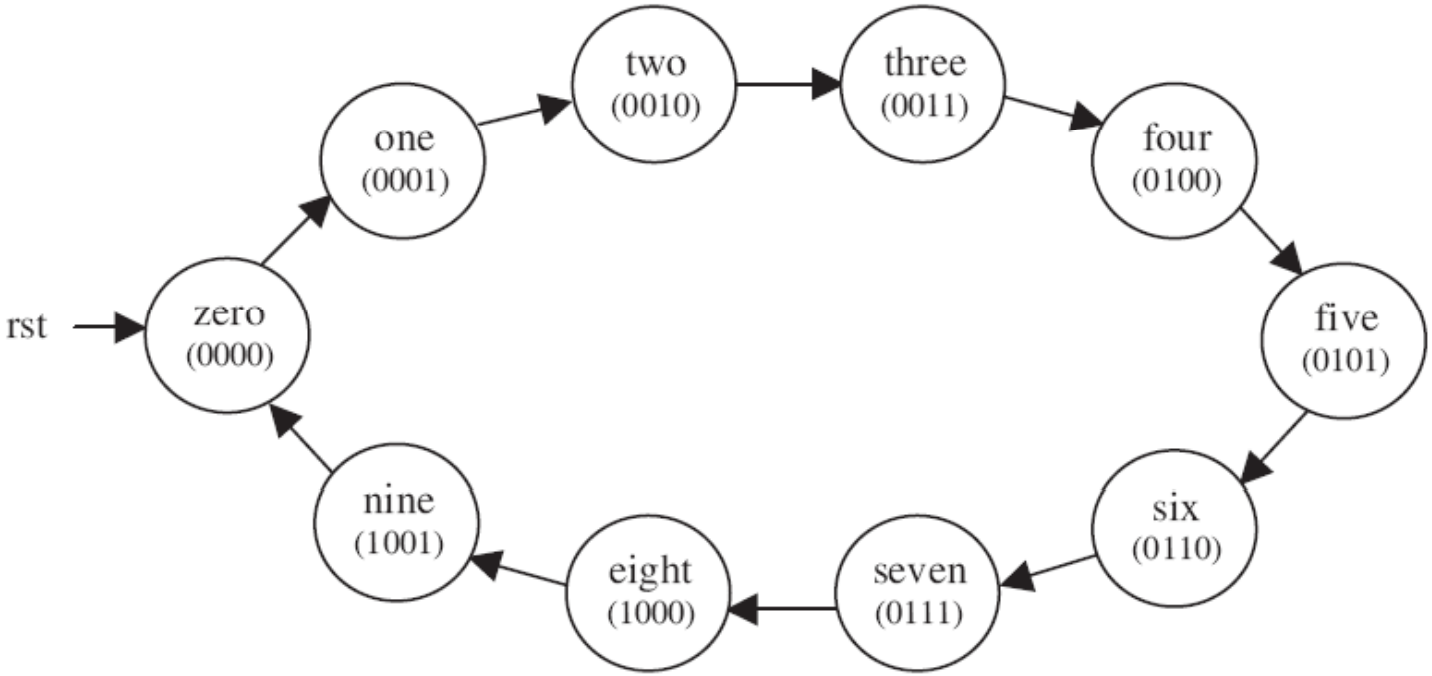
- **طراحی بخش بالایی** (بخش ترکیبی): گرچه برای این بخش میتوان از کد همزمان (ترکیبی) استفاده کرد اما از کدترتیبی هم میتوانیم استفاده کنیم. در قالب کدی که در ادامه آمده از CASE استفاده شده است. توجه کنید که دو قانون ۱ و ۲ که قبلا بیان شدند، اینجا باید رعایت شوند.

```
PROCESS (input, pr_state)
BEGIN
   CASE pr_state IS
      WHEN state0 =>
         IF (input = ...) THEN
            output <= <value>;
            nx_state <= state1;
         ELSE ...
         END IF;
      WHEN state1 =>
         IF (input = ...) THEN
            output <= <value>;
            nx_state <= state2;
         ELSE ...
         END IF;
      WHEN state2 =>
         IF (input = ...) THEN
            output <= <value>;
            nx_state <= state2;
         ELSE ...
         END IF;
      ...
   END CASE;
END PROCESS;
```

- قالب کلی شیوه طراحی شماره ۱

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY <entity_name> IS
   PORT ( input: IN <data_type>;
          reset, clock: IN STD_LOGIC;
          output: OUT <data_type>);
END <entity_name>;
-------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
   TYPE state IS (state0, state1, state2, state3, ...);
   SIGNAL pr_state, nx_state: state;
BEGIN
   ---------- Lower section: -----------------------
   PROCESS (reset, clock)
   BEGIN
      IF (reset='1') THEN
         pr_state <= state0;
      ELSIF (clock'EVENT AND clock='1') THEN
         pr_state <= nx_state;
      END IF;
   END PROCESS;
   ---------- Upper section: -----------------------
   PROCESS (input, pr_state)
   BEGIN
      CASE pr_state IS
         WHEN state0 =>
            IF (input = ...) THEN
               output <= <value>;
               nx_state <= state1;
            ELSE ...
            END IF;
         WHEN state1 =>
            IF (input = ...) THEN
               output <= <value>;
               nx_state <= state2;
            ELSE ...
            END IF;
         WHEN state2 =>
            IF (input = ...) THEN
               output <= <value>;
               nx_state <= state3;
            ELSE ...
            END IF;
         ...
      END CASE;
   END PROCESS;
END <arch_name>;
```

- مثال: طراحی یک شمارنده BCD
  - قبلا این شمارنده را به روش معمول (متداول) دیده بودیم. در این مثال میخواهیم به روش FSM آن را طراحی کنیم.
  - شمارنده یک ماشین مور است.
  - روش FSM در مورد یک شمارنده درصورتیکه تعداد حالات شمارنده زیاد باشد، پیچیده خواهد شد و لذا کارایی چندانی نخواهد داشت؛ در اینگونه موارد، بهتر است از روش معمول (متداول) استفاده کنیم.
  - دیاگرام حالت شمارنده BCD



- کد شمارنده BCD

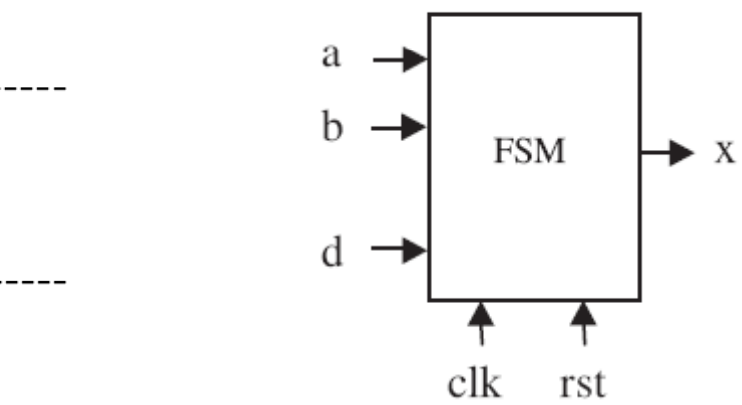
```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  ENTITY counter IS
6     PORT ( clk, rst: IN STD_LOGIC;
7            count: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
8  END counter;
9  ------------------------------------------------
10 ARCHITECTURE state_machine OF counter IS
11    TYPE state IS (zero, one, two, three, four,
12       five, six, seven, eight, nine);
13    SIGNAL pr_state, nx_state: state;
14 BEGIN
15    ------------ Lower section: ----------------
16    PROCESS (rst, clk)
17    BEGIN
18       IF (rst='1') THEN
19          pr_state <= zero;
20       ELSIF (clk'EVENT AND clk='1') THEN
21          pr_state <= nx_state;
22       END IF;
23    END PROCESS;
24    ------------ Upper section: ----------------
25    PROCESS (pr_state)
26    BEGIN
27       CASE pr_state IS
28          WHEN zero =>
29             count <= "0000";
30             nx_state <= one;
31          WHEN one =>
32             count <= "0001";
33             nx_state <= two;
```

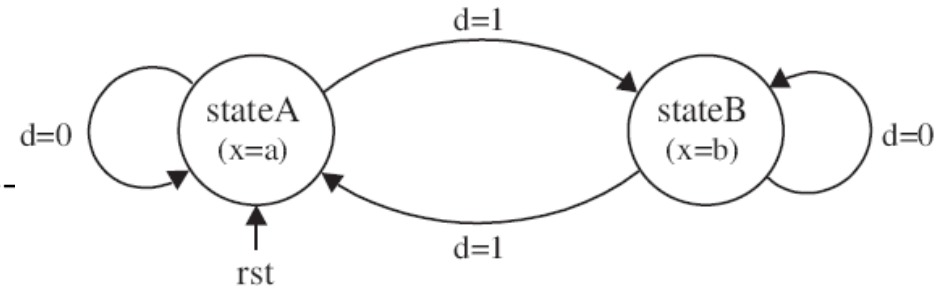
- کد شمارنده BCD ... (ادامه)

```
34          WHEN two =>
35             count <= "0010";
36             nx_state <= three;
37          WHEN three =>
38             count <= "0011";
39             nx_state <= four;
40          WHEN four =>
41             count <= "0100";
42             nx_state <= five;
43          WHEN five =>
44             count <= "0101";
45             nx_state <= six;
46          WHEN six =>
47             count <= "0110";
48             nx_state <= seven;
49          WHEN seven =>
50             count <= "0111";
51             nx_state <= eight;
52          WHEN eight =>
53             count <= "1000";
54             nx_state <= nine;
55          WHEN nine =>
56             count <= "1001";
57             nx_state <= zero;
58       END CASE;
59    END PROCESS;
END state_machine;
-------------------------------------------------
```

- **مثال:** پیاده سازی یک FSM ساده (ماشین میلی)

```
1  ------------------------------------------
2  ENTITY simple_fsm IS
3     PORT ( a, b, d, clk, rst: IN BIT;
4            x: OUT BIT);
5  END simple_fsm;
6  ------------------------------------------
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8     TYPE state IS (stateA, stateB);
9     SIGNAL pr_state, nx_state: state;
10 BEGIN
11    ----- Lower section: ----------------------
12    PROCESS (rst, clk)
13    BEGIN
14       IF (rst='1') THEN
15          pr_state <= stateA;
16       ELSIF (clk'EVENT AND clk='1') THEN
17          pr_state <= nx_state;
18       END IF;
19    END PROCESS;
20    ---------- Upper section: --------------
21    PROCESS (a, b, d, pr_state)
22    BEGIN
23       CASE pr_state IS
24          WHEN stateA =>
25             x <= a;
26             IF (d='1') THEN nx_state <= stateB;
27             ELSE nx_state <= stateA;
28             END IF;
29          WHEN stateB =>
30             x <= b;
31             IF (d='1') THEN nx_state <= stateA;
32             ELSE nx_state <= stateB;
33             END IF;
34       END CASE;
35    END PROCESS;
36 END simple_fsm;
37 ------------------------------------------
```

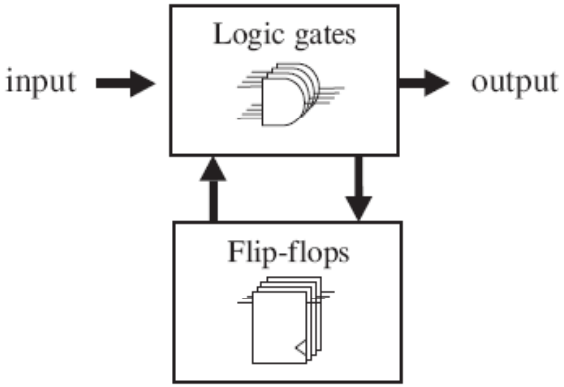




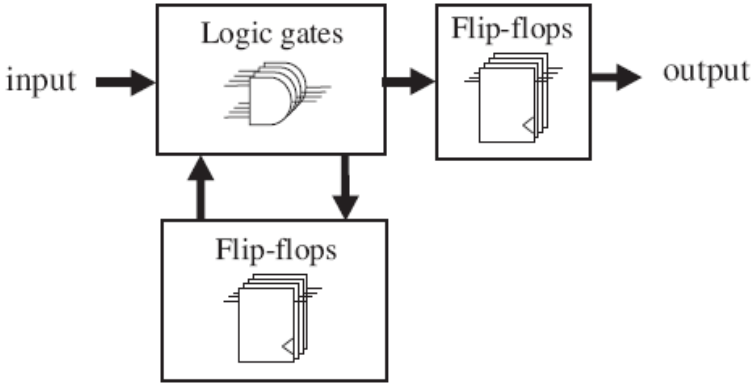
- در شیوه طراحی شماره ۱، خروجی از نوع آسنکرون است به این معنا که مستقل از زمانهای تغییر کلاک، هرگاه ورودی تغییر کند، خروجی میتواند تغییر کند(مانند مثال قبلی). هرگاه بخواهیم خروجی سنکرون باشد، باید از شیوه طراحی شماره ۲ استفاده کنیم.

شیوه طراحی شماره ۲

- در شیوه طراحی شماره ۱ از آنجا که خروجی آسنکرون است، نیازی به ذخیره آن (جهت همگام شدن با کلاک) نداریم. بنابراین، دیاگرام کلی این شیوه طراحی بصورت زیر است:



- در شیوه طراحی شماره ۲، میخواهیم خروجی آسنکرون باشد و لذا باید آن را ذخیره کنیم (تا در زمانهای خاصی که توسط کلاک تعیین میشود، مقدار ذخیره شده را به خروجی منتقل کنیم). لذا دیاگرام کلی شیوه طراحی شماره ۲ بصورت زیر است:



- برای تحقق شیوه طراحی شماره ۲، نیاز به انجام تغییرات بسیار کمی روی شیوه طراحی شماره ۱ داریم. برای اینکار از یک سیگنال واسط (به نام مثلا temp) برای محاسبه مقدار خروجی استفاده میکنیم اما فقط در لحظات تغییرات مورد نظر (که توسط کلاک تعیین میشود)، این مقدار محاسبه شده را به خروجی منتقل میکنیم. این کار در قالب کلی شیوه طراحی شماره ۲ در زیر نشان داده شده است:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY <ent_name> IS
   PORT (input: IN <data_type>;
         reset, clock: IN STD_LOGIC;
         output: OUT <data_type>);
END <ent_name>;
-------------------------------------------------------
ARCHITECTURE <arch_name> OF <ent_name> IS
   TYPE states IS (state0, state1, state2, state3, ...);
   SIGNAL pr_state, nx_state: states;
   SIGNAL temp: <data_type>;
BEGIN
   ---------- Lower section: ------------------------
   PROCESS (reset, clock)
   BEGIN
      IF (reset='1') THEN
         pr_state <= state0;
      ELSIF (clock'EVENT AND clock='1') THEN
         output <= temp;
         pr_state <= nx_state;
      END IF;
   END PROCESS;
```

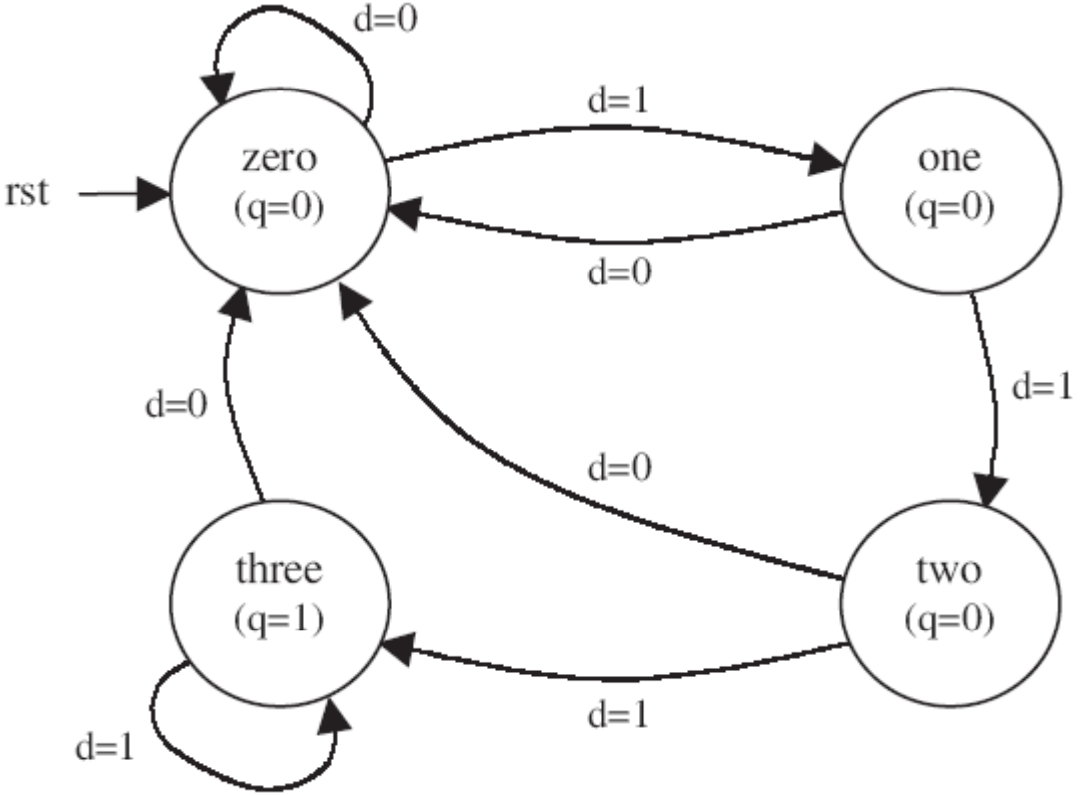
```
---------- Upper section: --------------------------
PROCESS (pr_state)
BEGIN
   CASE pr_state IS
      WHEN state0 =>
         temp <= <value>;
         IF (condition) THEN nx_state <= state1;
         ...
         END IF;
      WHEN state1 =>
         temp <= <value>;
         IF (condition) THEN nx_state <= state2;
         ...
         END IF;
      WHEN state2 =>
         temp <= <value>;
         IF (condition) THEN nx_state <= state3;
         ...
         END IF;
       ...
   END CASE;
END PROCESS;
END <arch_name>;
```

- مثال: بررسی مجدد مثال قبلی (FSM ساده) با این فرض که میخواهیم خروجی سنکرون باشد

```
1  ------------------------------------------
2  ENTITY simple_fsm IS
3     PORT ( a, b, d, clk, rst: IN BIT;
4            x: OUT BIT);
5  END simple_fsm;
6  ------------------------------------------
7  ARCHITECTURE simple_fsm OF simple_fsm IS
8     TYPE state IS (stateA, stateB);
9     SIGNAL pr_state, nx_state: state;
10    SIGNAL temp: BIT;
11 BEGIN
12    ----- Lower section: --------------------
13    PROCESS (rst, clk)
14    BEGIN
15       IF (rst='1') THEN
16          pr_state <= stateA;
17       ELSIF (clk'EVENT AND clk='1') THEN
18          x <= temp;
19          pr_state <= nx_state;
20       END IF;
21    END PROCESS;
```

```
22    ---------- Upper section: ----------------
23    PROCESS (a, b, d, pr_state)
24    BEGIN
25       CASE pr_state IS
26          WHEN stateA =>
27             temp <= a;
28             IF (d='1') THEN nx_state <= stateB;
29             ELSE nx_state <= stateA;
30             END IF;
31          WHEN stateB =>
32             temp <= b;
33             IF (d='1') THEN nx_state <= stateA;
34             ELSE nx_state <= stateB;
35             END IF;
36       END CASE;
37    END PROCESS;
38 END simple_fsm;
39 ------------------------------------------
```

- **مثال:** آشکارساز رشته
- میخواهیم مداری پیاده سازی کنیم که رشته "۱۱۱" را در دنباله ورودی آشکارسازی کرده و خروجی را ۱ کند؛ در غیراینصورت مقدار خروجی برابر ۰ باشد.



```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY string_detector IS
6     PORT ( d, clk, rst: IN BIT;
7            q: OUT BIT);
8  END string_detector;
9  ----------------------------------------
10 ARCHITECTURE my_arch OF string_detector IS
11    TYPE state IS (zero, one, two, three);
12    SIGNAL pr_state, nx_state: state;
13 BEGIN
14    ----- Lower section: --------------------
15    PROCESS (rst, clk)
16    BEGIN
17       IF (rst='1') THEN
18          pr_state <= zero;
19       ELSIF (clk'EVENT AND clk='1') THEN
20          pr_state <= nx_state;
21       END IF;
22    END PROCESS;
```

```
23    ---------- Upper section: ---------------
24    PROCESS (d, pr_state)
25    BEGIN
26       CASE pr_state IS
27          WHEN zero =>
28             q <= '0';
29             IF (d='1') THEN nx_state <= one;
30             ELSE nx_state <= zero;
31             END IF;
32          WHEN one =>
33             q <= '0';
34             IF (d='1') THEN nx_state <= two;
35             ELSE nx_state <= zero;
36             END IF;
37          WHEN two =>
38             q <= '0';
39             IF (d='1') THEN nx_state <= three;
40             ELSE nx_state <= zero;
41             END IF;
42          WHEN three =>
43             q <= '1';
44             IF (d='0') THEN nx_state <= zero;
45             ELSE nx_state <= three;
46             END IF;
47       END CASE;
48    END PROCESS;
49 END my_arch;
50 -----------------------------------------
```

- **راههای انکد کردن حالات**

۱. **روش باینری:** به حالات کدهای باینری نسبت داده میشود. این روش کمترین تعداد فلیپ فلاپها را مصرف میکند اما گیتهای منطقی زیادی را جهت تکمیل طرح لازم دارد؛ لذا، این روش کندترین مدار را تولید میکند.

۲. **روش onehot :** نحوه کدگذاری در جدول زیر آمده است. این روش بیشترین تعداد فلیپ فلاپها (برای n حالت به n فلیپ فلاپ نیاز دارد) و کمترین تعداد گیتهای منطقی (جهت تکمیل طرح) را مصرف میکند. لذا این روش سریعترین مدار را تولید میکند.

۳. **روش twohot :** روشی مابین دو روش قبلی است و نحوه کدگذاری آن در جدول زیر آمده است. دراین روش با n فلیپ فلاپ میتوان n(n-1)/2 حالت را کدگذاری کرد.

	Encoding Style		
STATE	BINARY	TWOHOT	ONEHOT
state0	000	00011	00000001
state1	001	00101	00000010
state2	010	01001	00000100
state3	011	10001	00001000
state4	100	00110	00010000
state5	101	01010	00100000
state6	110	10010	01000000
state7	111	01100	10000000

فصل دهم : بسته ها و قطعه ها

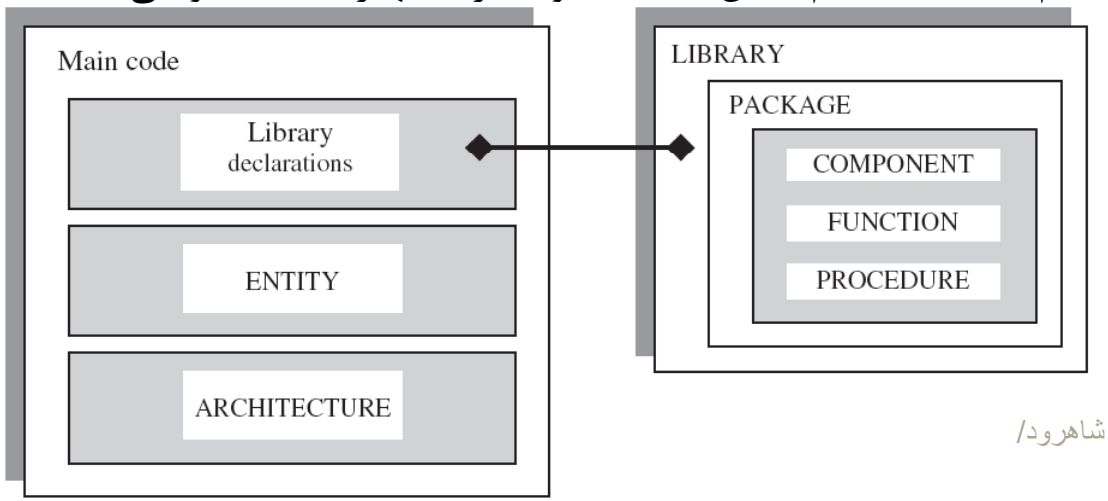
- تاکنون تمام اجزای لازم جهت نوشته یک کد VHDL را (همانطور که در شکل سمت چپی زیر نشان داده شده) دیدیم. در این فصل و فصل بعدی برخی مطالب تکمیلی در مورد کتابخانه ها آورده میشود. این مطالب تکمیلی در مورد بلوکهای سازنده جدیدی است به نامهای بسته ها (فصل دهم)، قطعه ها (فصل دهم)، توابع (فصل یازدهم)، و رویه ها (فصل یازدهم)
- سه حسن اصلی این بلوکهای سازنده عبارتند از :

۱. بخش بندی کد (Code Partitioning)

۲. به اشتراک گذاری کد (Code Sharing)

۳. استفاده مجدد (Reuse)

- علیرغم اینکه از بلوکهای فوق میتوان در کداصلی مستقیما استفاده کرد، اما متداول (وبهتر) است که آنها را داخل کتابخانه استفاده کنیم (کمک به انجام بخش بندی کد و مدیریت بهتر کدهای طولانی).



- قطعه کدهایی را که زیاد استفاده میشوند، به فرم یک ***قطعه*** (Component) یا ***تابع*** (Function) یا ***رویه*** (Procedure) نوشته و داخل یک ***بسته*** (Package) قرار میدهیم. بسته نیز نهایتا داخل یک کتابخانه کامپایل میشود.
- **بسته (Package)**
  - گرامر تعریف یک بسته

```
PACKAGE package_name IS
    (declarations)
END package_name;

[PACKAGE BODY package_name IS
    (FUNCTION and PROCEDURE descriptions)
END package_name;]
```

  - داخل یک بسته میتوان علاوه بر قطعه ها، توابع، و رویه ها از تعاریفی همچون TYPE و CONSTANTS نیز استفاده کرد.
  - گرامر فوق از دو بخش PACKAGE و PACKAGE BODY (بسته و بدنه بسته) تشکیل شده که اولی الزامی و دومی اختیاری (یعنی در صورت لزوم باید استفاده کرد) است. بخش اول شامل تمام اعلانها و بخش دوم زمانی لازم است که در بخش اول یک تابع یا یک رویه اعلان شده باشد که در اینصورت توصیف کامل بدنه این توابع یا رویه ها باید داخل بخش دوم آورده شود.
  - توجه کنید که هر دو بخش مذکور باید یک نام (مشترک) داشته باشند.
  - **مثال:** تعریف یک بسته ساده

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  PACKAGE my_package IS
6     TYPE state IS (st1, st2, st3, st4);
7     TYPE color IS (red, green, blue);
8     CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9  END my_package;
10 ------------------------------------------------
```

از بدنه بسته استفاده نشده زیرا نیازی نیست.

- **مثال:** تعریف یک بسته شامل یک تابع

```
1  ------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  PACKAGE my_package IS
6     TYPE state IS (st1, st2, st3, st4);
7     TYPE color IS (red, green, blue);
8     CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11111111";
9     FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
10 END my_package;
11 ------------------------------------------------
12 PACKAGE BODY my_package IS
13    FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
14    BEGIN
15       RETURN (s'EVENT AND s='1');
16    END positive_edge;
17 END my_package;
18 ------------------------------------------------
```

- هر کدام از بسته های دو مثال اخیر را کامپایل نموده و جزء کتابخانه work (یا هر نام دیگری) قرار داد. برای استفاده از این بسته ها در داخل یک کد VHDL باید از USE مطابق زیر استفاده کرد:

```
-----------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
-----------------------------------
ENTITY...
...
ARCHITECTURE...
...
-----------------------------------
```

- **قطعه (Component)**
  - یک "قطعه" راه دیگری برای بخش بندی کد و استفاده مجدد و به اشتراک گذاری آن است. بطور مثال مدارات پراستفاده مانند فلیپ فلاپها، مالتی پلکسرها، جمع کننده ها و غیره را میتوان داخل یک کتابخانه قرار داده و در هر پروژه ای بدون نیاز به تعریف مجدد، از آنها استفاده کرد.
  - به هر تکه کدی که زیاد مورد استفاده قرار بگیرد، میتوان "قطعه" گفت. اگر چنین کدی را "قطعه" تعریف و اعلان کنیم میتوانیم از آن داخل هر مدار دیگری استفاده کنیم و به این ترتیب طرحهای سلسله مراتبی (Hierarchical Designs) بسازیم.

- برای استفاده از یک قطعه، ابتدا باید آن را ***اعلان*** (Declare) و سپس آن را ***نمونه سازی*** (Instantiate) کنیم. نحوه اعلان و نمونه سازی در زیر نشان داده شده است.

- **اعلان:**

```
COMPONENT component_name IS
    PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END COMPONENT;
```

- ملاحظه میکنید که نحوه اعلان مشابه اعلان یک موجودیت است یعنی باید نام پورتها، **مد(Mode)** یا حالت هرکدام و نیز **نوع(Type)** هر کدام ذکر شود

- **نمونه سازی:**

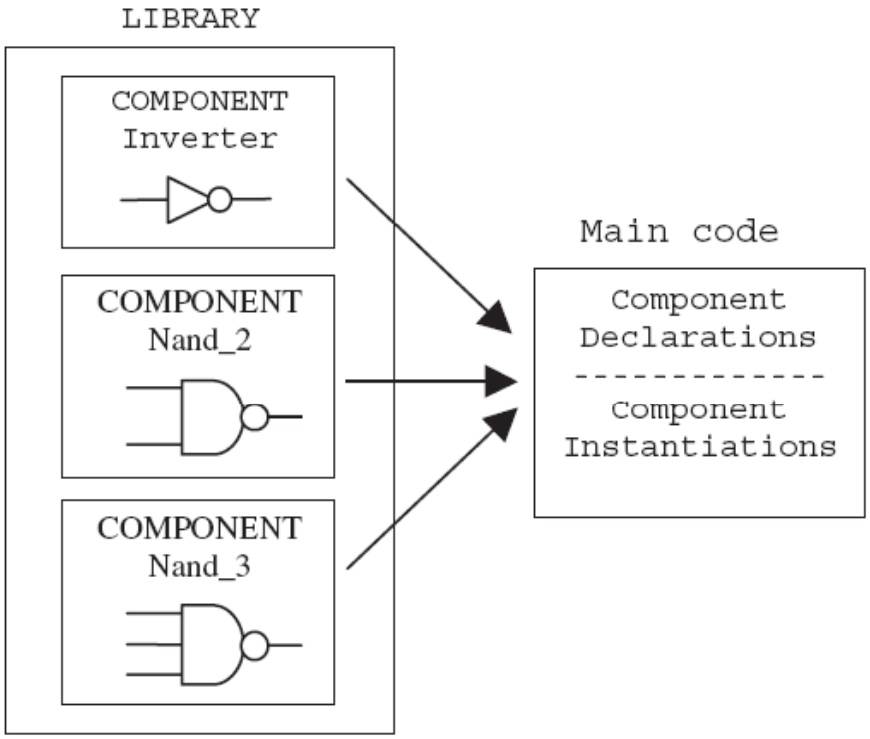
```
label: component_name PORT MAP (port_list);
```

- در نمونه سازی از یک برچسب، نام قطعه، و لیست پورتها استفاده میکنیم. لیست پورتها در حقیقت رابطه بین پورتهای مدار حقیقی و پورتهای قطعه از قبل تعریف شده را مشخص مسکند.

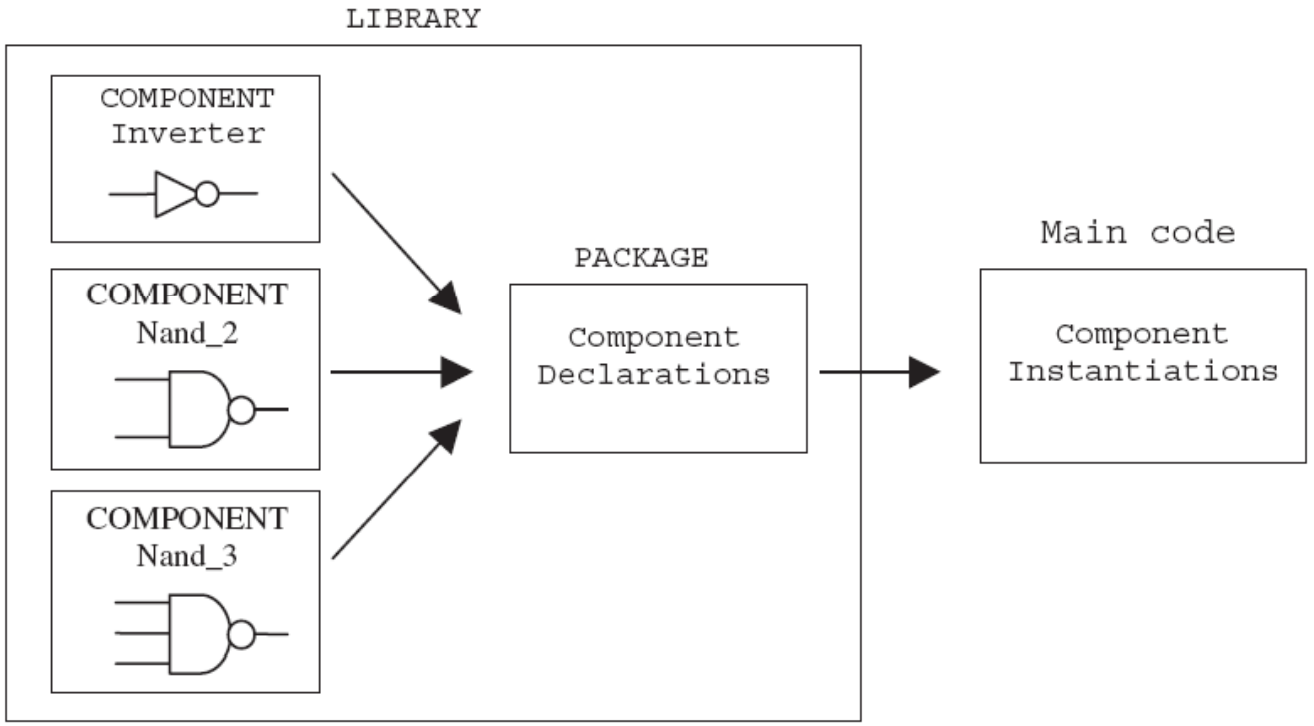
- **مثال:** فرض کنید قبلا یک معکوس کننده را داخل فایل inverter.vhd تعریف و داخل کتابخانه work کامپایل کرده ایم. حال به کمک کد زیر میتوان از این قطعه استفاده کرد. در این کد فرض شده که نام پورتهای مدار واقعی x و y و نام پورتها در زمان تعریف قطعه، a و b بوده است. به این روش نگاشت بین پورتها، ***نگاشت مکانی*** (positional mapping) گفته میشود.

```
----- COMPONENT declaration: -----------
COMPONENT inverter IS
      PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
END COMPONENT;

----- COMPONENT instantiation: -----------
U1: inverter PORT MAP (x, y);
```

- وقتی که قطعه ای را طراحی کرده و داخل کتابخانه مورد نظرمان قرار دادیم، دو راه برای اعلان آن قطعه وجود دارد:
- راه اول این است که مستقیما میتوانیم آن را داخل بدنه اصلی برنامه اعلان کنیم:

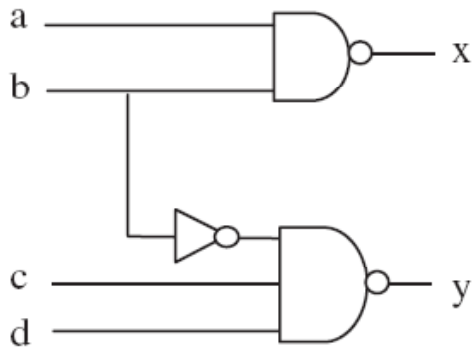


- راه دوم این است که از یک بسته جهت اعلان آن استفاده کنیم.
- روش دوم بهتر است زیرا هر بار که بخواهیم از قطعه استفاده کنیم نیازی به اعلان مجدد آن نداریم.



- در ادامه دو مثال که هر کدام یکی از این دو روش را نشان میدهند، آورده میشود.

- **مثال:** میخواهیم مدار شکل زیر را فقط به کمک **قطعه ها** (شامل inverter، nand_2، و nand_3) و نه استفاده از **بسته ها** پیاده سازی کنیم. برای اینکار چهار تکه کد لازم داریم: یک تکه کد برای هر یک از قطعات و یک تکه کد هم برای برنامه اصلی. تمام این چهار تکه کد در ادامه نشان داده شده اند. توجه کنید که چون از بسته ها استفاده نکرده ایم، تمام قطعات باید در بخش اعلان معماری برنامه اصلی، اعلان شوند.




```
1  ------ File inverter.vhd: -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY inverter IS
6     PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
7  END inverter;
8  ------------------------------------
9  ARCHITECTURE inverter OF inverter IS
10 BEGIN
11    b <= NOT a;
12 END inverter;
13 ----------------------------------------------
```

```
1  ------ File nand_2.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY nand_2 IS
6     PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
7  END nand_2;
8  ------------------------------------
9  ARCHITECTURE nand_2 OF nand_2 IS
10 BEGIN
11    c <= NOT (a AND b);
12 END nand_2;
13 ---------------------------------------------
```

```
1  ----- File nand_3.vhd: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY nand_3 IS
6     PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
7  END nand_3;
8  ------------------------------------
9  ARCHITECTURE nand_3 OF nand_3 IS
10 BEGIN
11    d <= NOT (a AND b AND c);
12 END nand_3;
13 ---------------------------------------------
```

```
1  ----- File project.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY project IS
6     PORT (a, b, c, d: IN STD_LOGIC;
7           x, y: OUT STD_LOGIC);
8  END project;
9  ------------------------------------
10 ARCHITECTURE structural OF project IS
11    -------------
12    COMPONENT inverter IS
13       PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
14    END COMPONENT;
15    -------------
16    COMPONENT nand_2 IS
17       PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
18    END COMPONENT;
19    -------------
20    COMPONENT nand_3 IS
21       PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
22    END COMPONENT;
23    -------------
24    SIGNAL w: STD_LOGIC;
25 BEGIN
26    U1: inverter PORT MAP (b, w);
27    U2: nand_2 PORT MAP (a, b, x);
28    U3: nand_3 PORT MAP (w, c, d, y);
29 END structural;
30 -----------------------------------------
```

- **مثال:** در این مثال همان مدار قبلی را میخواهیم به کمک تعریف قطعه ها و قرار دادن آنها در یک بسته انجام دهیم. در این حالت به پنج تکه کد نیاز داریم: برای تعریف هر قطعه یک تکه کد، یک تکه کد برای تعریف بسته، و یک تکه کد نیز برای برنامه اصلی.
- بر خلاف ظاهر کار که یک تکه کد بیشتر تولید شده، حسن بزرگ این روش در این است که دیگر نیازی به تکرار اعلان قطعه ها در برنامه اصلی به ازاء هر بار نمونه سازی این قطعه ها نیست.
- توجه کنید که برنامه اصلی شامل دستور USE به منظور استفاده از بسته مذکور میباشد.

```
1  ------ File inverter.vhd: -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY inverter IS
6     PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
7  END inverter;
8  ------------------------------------
9  ARCHITECTURE inverter OF inverter IS
10 BEGIN
11    b <= NOT a;
12 END inverter;
13 ----------------------------------------------
```

```
1  ------ File nand_2.vhd: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------
5  ENTITY nand_2 IS
6     PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
7  END nand_2;
8  ---------------------------------
9  ARCHITECTURE nand_2 OF nand_2 IS
10 BEGIN
11    c <= NOT (a AND b);
12 END nand_2;
13 -----------------------------------------
1  ----- File nand_3.vhd: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------
5  ENTITY nand_3 IS
6     PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
7  END nand_3;
8  ---------------------------------
9  ARCHITECTURE nand_3 OF nand_3 IS
10 BEGIN
11    d <= NOT (a AND b AND c);
12 END nand_3;
13 -----------------------------------------
```

```
1  ----- File my_components.vhd: ---------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------
5  PACKAGE my_components IS
6     ------ inverter: -------
7     COMPONENT inverter IS
8        PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
9     END COMPONENT;
10    ------ 2-input nand: ---
11    COMPONENT nand_2 IS
12       PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
13    END COMPONENT;
14    ------ 3-input nand: ---
15    COMPONENT nand_3 IS
16       PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
17    END COMPONENT;
18    ------------------------
19 END my_components;
20 ----------------------------------------
```

```
1  ----- File project.vhd: ---------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_components.all;
5  ---------------------------------
6  ENTITY project IS
7     PORT ( a, b, c, d: IN STD_LOGIC;
8            x, y: OUT STD_LOGIC);
9  END project;
10 ---------------------------------
11 ARCHITECTURE structural OF project IS
12    SIGNAL w: STD_LOGIC;
13 BEGIN
14    U1: inverter PORT MAP (b, w);
15    U2: nand_2 PORT MAP (a, b, x);
16    U3: nand_3 PORT MAP (w, c, d, y);
17 END structural;
18 -----------------------------------------
```

- **نگاشت پورتها (Port Map)**
  - به عمل تعیین نحوه ارتباط بین پورتهای واقعی مدار و پورتهای از قبل تعریف شده یک قطعه (در حین نمونه سازی)، نگاشت پورتها گفته میشود.
  - فرض کنید قطعه ای بصورت زیر تعریف شده باشد:

```
COMPONENT inverter IS
   PORT (a: IN STD_LOGIC; b: OUT STD_LOGIC);
END COMPONENT;
```

  - دو روش نگاشت پورتها:
  - ***نگاشت مکانی (Positional Mapping):*** (که قبلا دیده شد) به صورت زیر میتوان در حین نمونه سازی از نگاشت مکانی استفاده کرد:

```
U1: inverter PORT MAP (x, y);
```

  - در این نگاشت، پس از نمونه سازی، پورتهای واقعی مدار، یعنی x و y به پورتهای قطعه از قبل تعریف شده (در بالا) یعنی به ترتیب به a و b متصل میشوند.
  - ***نگاشت نامی (Nominal Mapping):*** در این روش صریحا ذکر میکنیم که کدام پورتها به یکدیگر متصل شوند. در مثال فعلی مینویسیم:

```
U1: inverter PORT MAP (x=>a, y=>b);
```

  - روش نگاشت مکانی ساده تر و راخت تر است اما روش نگاشت نامی، نسبت به خطاها مقاومتر است. در روش نگاشت نامی میتوان برخی پورتها را بدون اتصال رها کرد. برای اینکار از کلمه کلیدی OPEN استفاده میکنیم:

```
U2: my_circuit PORT MAP (x=>a, y=>b, w=>OPEN, z=>d);
```

- **نگاشت عمومی (GENERIC Map)**
  - برای نمونه سازی قطعاتی که از پارامترهای GENERIC استفاده کرده باشند، باید بصورت زیر عمل کرد:

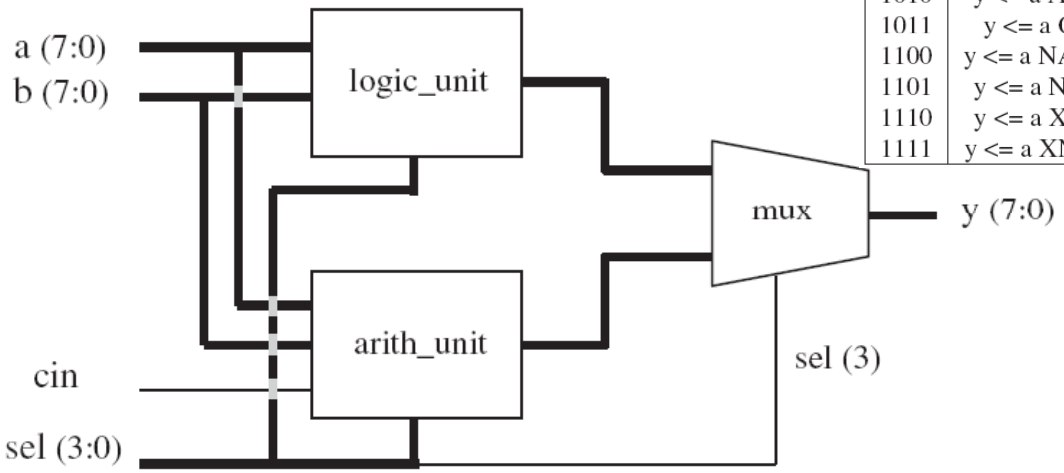
```
label: compon_name GENERIC MAP (param. list) PORT MAP (port list);
```

  - **مثال:** تحقق یک تولید کننده توازن زوج
  - در کد نشان داده شده توجه کنید که مقدار پیش فرض پارامتر n در برنامه مربوط به تعریف قطعه برابر ۷ میباشد اما در برنامه اصلی و در هنگام نمونه سازی قطعه مذکور، مقدار پارامتر به ۲ تغییر می یابد (به کمک دستور GENERIC MAP). این مطلب از ویژگیهای قطعات GENERIC است.
  - کد نشان داده شده دو بخش دارد: یکی مربوط به تعریف قطعه و دیگری مربوط به بدنه اصلی برنامه است. توجه کنید که در هنگام اعلان قطعه در برنامه اصلی، استفاده از GENERIC الزامی است اما لزومی ندارد که همان مقدار اولیه پارامترها دوباره استفاده شود.

```
1  ------ File parity_gen.vhd (component): -------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  ENTITY parity_gen IS
6     GENERIC (n : INTEGER := 7);  -- default is 7
7     PORT ( input: IN BIT_VECTOR (n DOWNTO 0);
8            output: OUT BIT_VECTOR (n+1 DOWNTO 0));
9  END parity_gen;
10 ------------------------------------
11 ARCHITECTURE parity OF parity_gen IS
12 BEGIN
13    PROCESS (input)
14       VARIABLE temp1: BIT;
15       VARIABLE temp2: BIT_VECTOR (output'RANGE);
16    BEGIN
17       temp1 := '0';
18       FOR i IN input'RANGE LOOP
19          temp1 := temp1 XOR input(i);
20          temp2(i) := input(i);
21       END LOOP;
22       temp2(output'HIGH) := temp1;
23       output <= temp2;
24    END PROCESS;
25 END parity;
26 ------------------------------------------------------
```

- **مثال:** پیاده سازی یک ALU به کمک تعریف قطعات
  - این مثال گرچه قبلا حل شده اما آن کد خودشمول (self-contained) بود؛ اما این بار، به کمک تعریف قطعات میخواهیم آن را پیاده سازی کنیم.
  - دیاگرام این ALU مجددا در شکل زیر آورده شده است.
  - فرض میکنیم که کتابخانه مان شامل سه قطعه به نامهای logic_unit، arith_unit و mux است و از روی این قطعات میخواهیم ALU را بسازیم. در این مثال برای پیاده سازی، از بسته استفاده نشده است و اعلان قطعات در بدنه اصلی برنامه دوباره آورده شده است.

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	



```
1  -------- COMPONENT arith_unit: --------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_unsigned.all;
5  ----------------------------------------
6  ENTITY arith_unit IS
7     PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
8            sel: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
9            cin: IN STD_LOGIC;
10           x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
11 END arith_unit;
12 ----------------------------------------
13 ARCHITECTURE arith_unit OF arith_unit IS
14    SIGNAL arith, logic: STD_LOGIC_VECTOR (7 DOWNTO 0);
15 BEGIN
16    WITH sel SELECT
17       x <=  a WHEN "000",
18             a+1 WHEN "001",
19             a-1 WHEN "010",
20             b WHEN "011",
21             b+1 WHEN "100",
22             b-1 WHEN "101",
23             a+b WHEN "110",
24             a+b+cin WHEN OTHERS;
25 END arith_unit;
26 ----------------------------------------------
```

```
1  -------- COMPONENT logic_unit: -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY logic_unit IS
6     PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            sel: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
8            x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END logic_unit;
10 ----------------------------------------
11 ARCHITECTURE logic_unit OF logic_unit IS
12 BEGIN
13    WITH sel SELECT
14       x <=  NOT a WHEN "000",
15             NOT b WHEN "001",
16             a AND b WHEN "010",
17             a OR b WHEN "011",
18             a NAND b WHEN "100",
19             a NOR b WHEN "101",
20             a XOR b WHEN "110",
21             NOT (a XOR b) WHEN OTHERS;
22 END logic_unit;
23 ----------------------------------------------
```

```
1  -------- COMPONENT mux: --------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY mux IS
6     PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7            sel: IN STD_LOGIC;
8            x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END mux;
10 ----------------------------------------
11 ARCHITECTURE mux OF mux IS
12 BEGIN
13    WITH sel SELECT
14       x <=   a WHEN '0',
15              b WHEN OTHERS;
16 END mux;
17 ------------------------------------------------
1  -------- Project ALU (main code): ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY alu IS
6     PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7            cin: IN STD_LOGIC;
8            sel: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9            y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
```

```
10 END alu;
11 ---------------------------------------
12 ARCHITECTURE alu OF alu IS
13 ----------------------
14    COMPONENT arith_unit IS
15    PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
16           cin: IN STD_LOGIC;
17           sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
18           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
19    END COMPONENT;
20    ----------------------
21    COMPONENT logic_unit IS
22    PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
23           sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
24           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
25    END COMPONENT;
26    ----------------------
27    COMPONENT mux IS
28    PORT ( a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
29           sel: IN STD_LOGIC;
30           x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
31    END COMPONENT;
32    ----------------------
33    SIGNAL x1, x2: STD_LOGIC_VECTOR(7 DOWNTO 0);
34 ----------------------
35 BEGIN
36    U1: arith_unit PORT MAP (a, b, cin, sel(2 DOWNTO 0), x1);
37    U2: logic_unit PORT MAP (a, b, sel(2 DOWNTO 0), x2);
38    U3: mux PORT MAP (x1, x2, sel(3), y);
39 END alu;
40 ------------------------------------------------
```

دانشگاه صنعتی شاهرود
دانشکده برق

عنوان درس

Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

فصل یازدهم : توابع و رویه ها

- توابع (FUNCTION) و رویه ها (PROCEDURE) را مجموعا زیربرنامه (sub-program) نیز می گویند.
- از لحاظ ساختاری توابع و رویه ها مشابه فرآیندها می باشند زیرا تنها محل برای نوشتن کدهای ترتیبی هستند و بنابراین داخل توابع و رویه ها هم مشابه فرآیندها میتوان از دستورات ترتیبی – که مخصوص کدهای ترتیبی اند- مانند IF، CASE و LOOP استفاده کرد. البته از WAIT نمیتوان استفاده کرد.
- تفاوت اساسی بین توابع و رویه ها با فرآیندها در این است که فرآیندها بیشتر به منظور استفاده فوری در برنامه اصلی برنامه هستند اما توابع و رویه ها بیشتر به منظور استفاده در کتابخانه و به منظور ذخیره کدهایی که به کرات مورد استفاده اند، کاربرد دارند. به عبارت دیگر میتوان گفت که فرآیندها در هر برنامه ای مخصوص همان برنامه اند، اما توابع و رویه ها را میتوان در برنامه های مختلف استفاده کرد.
- **تابع (FUNCTION)**
  - همه دستوراتی که میتوان در یک فرآیند استفاده کرد، به جز دستور WAIT، در یک تابع نیز میتوان استفاده کرد. البته دو کار دیگر نیز در تابع مجاز نیست: یکی اعلان سیگنال و دیگری نمونه سازی یک قطعه.

- برای ایجاد و استفاده از توابع باید ابتدا بدنه تابع را تعریف کرده و سپس آن را فراخوانی کنیم:
- ***نحوه تعریف بدنه یک تابع:***

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

- لیست پارامترها به صورت زیر میتواند شامل سیگنالها و ثوابت باشد (نوع متغیر مجاز نیست):

⟨parameter list⟩ = [CONSTANT] constant_name: constant_type;

⟨parameter list⟩ = SIGNAL signal_name: signal_type;

- تعداد پارامترها هر مقداری (حتی صفر) میتواند باشد. نوع آنها میتواند هر نوع قابل سنتزی باشد مانند BOOLEAN، STD_LOGIC، و INTEGER ؛ البته توجه کنید که در این انواع قابل سنتز نمیتوانید از RANGE و یا TO/DOWNTO (و یا هر چیز دیگری) برای مشخص کردن محدوده استفاده کنید.
- تابع فقط دارای یک مقدار بازگشتی است که نوع آن به کمک *data_type* مشخص میشود.
- **مثال:** تابعی که در ادامه آورده میشود، f1 نام داشته و سه پارامتر ورودی a و b و c دارد. a و b از نوع ثابت بوده و c از نوع سیگنال هستند. توجه کنید که پارامترها بطور پیش فرض از نوع ثابت میباشند و بنابراین اگر از CONSTANT استفاده نشود، مشکلی پیش نمی آید. علاوه بر مشخصات فوق، a و b از نوع INTEGER و c از نوع STD_LOGIC_VECTOR میباشند. توجه کنید که از کلماتی مانند RANGE و TO/DOWNTO برای تعیین محدوده استفاده نشده است.

```
FUNCTION f1 (a, b: INTEGER; SIGNAL c: STD_LOGIC_VECTOR)
    RETURN BOOLEAN IS
BEGIN
    (sequential statements)
END f1;
```

- ***نحوه فراخوانی یک تابع:***
- فراخوانی یک تابع قسمتی از یک عبارت (ترتیبی یا همزمان) میتواند باشد.

```
x <= conv_integer(a);          -- converts a to an integer
                               -- (expression appears by itself)
y <= maximum(a, b);            -- returns the largest of a and b
                               -- (expression appears by itself)
IF x > maximum(a, b) ...       -- compares x to the largest of a, b
                               -- (expression associated to a
                               -- statement)
```

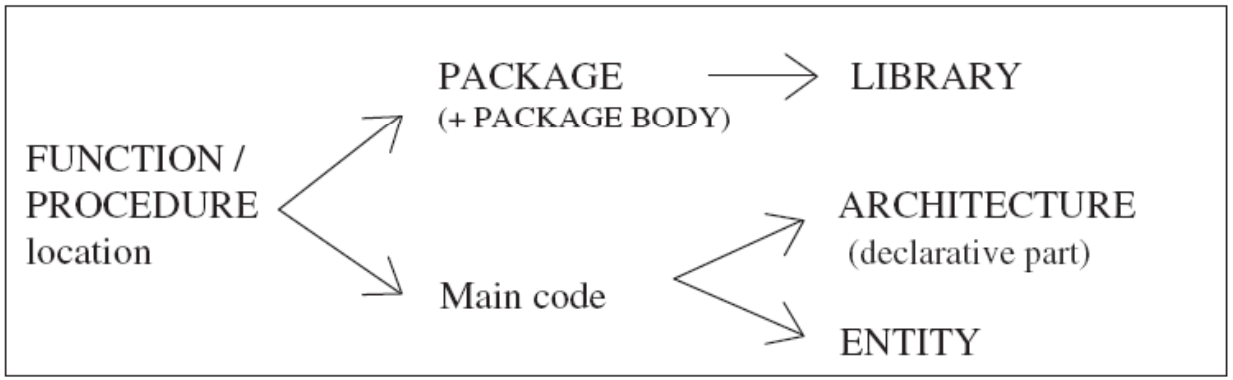
- **مثال:** تابع positive_edge
- این تابع لبه بالارونده (یا لبه مثبت) کلاک را تشخیص داده و معادل دستور IF (clk'EVENT AND clk='1') میباشد.

```
------ Function body: -------------------------------
FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
BEGIN
   RETURN (s'EVENT AND s='1');
END positive_edge;
------ Function call: -------------------------------
...
IF positive_edge(clk) THEN...
...
-----------------------------------------------------
```

- **مثال:** تابعی برای تبدیل نوع STD_LOGIC_VECTOR به نوع INTEGER
- توجه کنید که این کد، عمومی (یا GENERIC) است یعنی برای هر محدوده یا ترتیبی از ورودی کار میکند.

```
------ Function body: -------------------------------
FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
      RETURN INTEGER IS
   VARIABLE result: INTEGER RANGE 0 TO 2**vector'LENGTH-1;
BEGIN
   IF (vector(vector'HIGH)='1') THEN result:=1;
   ELSE result:=0;
   END IF;
   FOR i IN (vector'HIGH-1) DOWNTO (vector'LOW) LOOP
      result:=result*2;
      IF(vector(i)='1') THEN result:=result+1;
      END IF;
   END LOOP;
   RETURN result;
END conv_integer;

------ Function call: -------------------------------
...
y <= conv_integer(a);
...
-----------------------------------------------------
```

- محلهای قرارگیری تابع
  - محلهای متداول قرارگیری یک تابع(و البته یک رویه) در شکل زیر نشان داده شده است.



  - گرچه یک تابع معمولا در یک بسته قرار میگیرد (به منظور بخش بندی کد، یا استفاده مجدد، و یا به اشتراک گذاری کد) اما میتوان آن را در بدنه اصلی برنامه (یا داخل ARCHITECTURE یا داخل ENTITY) نیز قرار داد.
  - هرگاه تابع (یا رویه) داخل یک ***بسته*** قرار داده شود، باید برای آن بسته، ***بدنه بسته*** نیز نوشت و بدنه تابع(یا رویه) را داخل این بدنه تعریف کرد.
  - **مثال:** در این مثال یک تابع را در بدنه برنامه اصلی قرار میدهیم. برای این کار میتوانیم آن را در ENTITY یا بخش اعلان ARCHITECTURE قرار دهیم. در اینجا همان تابع positive_edge() را که قبلا دیدیم، داخل بدنه اصلی و بخش اعلان معماری قرار می دهیم و سپس از آن برای ساخت یک DFF استفاده میکنیم..

```
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 my_arch OF dff IS
11 ------------------------------------------
12    FUNCTION positive_edge(SIGNAL s: STD_LOGIC)
13       RETURN BOOLEAN IS
14    BEGIN
15       RETURN s'EVENT AND s='1';
16    END positive_edge;
17 ------------------------------------------
18 BEGIN
19    PROCESS (clk, rst)
20    BEGIN
21       IF (rst='1') THEN q <= '0';
22       ELSIF positive_edge(clk) THEN q <= d;
23       END IF;
24    END PROCESS;
25 END my_arch;
26 ------------------------------------------
```

- **مثال:** در این مثال همان تابع قبلی را داخل یک بسته مورد استفاده قرار میدهیم و به این ترتیب این قابلیت را بدست می آوریم که تابع مذکور توسط دیگر برنامه ها و پروژه ها نیز قابل استفاده شود.
- در این حالت باید تابع را داخل PACKAGE اعلان و داخل PACKAGE BODY تعریف (یا توصیف) کنیم.
- در ادامه دو کد نشان داده شده است که در اولی تابع تعریف میشود و در دومی نمونه از فراخوانی این تابع نشان داده میشود. این دو کد را یا میتوان جداگانه در دو فایل مجزا کامپایل کرد یا اینکه در یک فایل آنها را کامپایل کنیم (داخل فایلی به نام dff.vhd که در واقع نام موجودیت است).
- به جمله USE work.my_package.all در برنامه اصلی توجه کنید.

```
1  ------- Package: ----------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------------
5  PACKAGE my_package IS
6     FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
7  END my_package;
8  ----------------------------------------------
```

```
9  PACKAGE BODY my_package IS
10    FUNCTION positive_edge(SIGNAL s: STD_LOGIC)
11       RETURN BOOLEAN IS
12    BEGIN
13       RETURN s'EVENT AND s='1';
14    END positive_edge;
15 END my_package;
16 ---------------------------------------------

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

- **مثال:** تابعی که در این مثال بررسی میشود همان تابع conv_integer() است که قبلا بررسی شد و برای تبدیل نوع STD_LOGIC_VECTOR به نوع INTEGER استفاده میشد. در اینجا آن را داخل یک بسته (و البته بدنه بسته) تعریف کرده و در برنامه اصلی آن را فراخوانی میکنیم.

```
1  --------- Package: ----------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------
5  PACKAGE my_package IS
6     FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
7        RETURN INTEGER;
8  END my_package;
9  ------------------------------------------------
10 PACKAGE BODY my_package IS
11    FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
12          RETURN INTEGER IS
13       VARIABLE result: INTEGER RANGE 0 TO 2**vector'LENGTH-1;
14    BEGIN
15       IF (vector(vector'HIGH)='1') THEN result:=1;
16       ELSE result:=0;
17       END IF;
```

```
18        FOR i IN (vector'HIGH-1) DOWNTO (vector'LOW) LOOP
19           result:=result*2;
20           IF(vector(i)='1') THEN result:=result+1;
21           END IF;
22        END LOOP;
23        RETURN result;
24     END conv_integer;
25 END my_package;
26 ---------------------------------------------
1  -------- Main code: -------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_package.all;
5  ---------------------------------------------
6  ENTITY conv_int2 IS
7     PORT ( a: IN STD_LOGIC_VECTOR(0 TO 3);
8            y: OUT INTEGER RANGE 0 TO 15);
9  END conv_int2;
10 ---------------------------------------------
11 ARCHITECTURE my_arch OF conv_int2 IS
12 BEGIN
13    y <= conv_integer(a);
14 END my_arch;
15 ---------------------------------------------
```

- **مثال:** سربازگذاری عملگر "+"
- در این مثال تابعی به نام "+" معرفی میشود که عملکرد عملگر + را سربارگذاری میکند. توجه داریم که عملگر + در حالت معمولی فقط عملوندهای از نوع INTEGER، SIGNED و UNSIGNED می پذیرد اما ما در این مثال علاقه مندیم تابعی داشته باشیم که به ما اجازه دهد که عمل + را روی عملوندهای از نوع STD_LOGIC_VECTOR نیز انجام دهیم.
- در انجا هم دو کد ارائه میشود که یکی برای تعریف (داخل بسته) و دیگری برای فراخوانی تابع مورد نظر استفاده میشود. در این تابع پارامترهای ورودی و خروجی همگی از نوع STD_LOGIC_VECTOR میباشند و ما فرض میکنیم که اینها همگی طول بیت یکسانی دارند.

```
1   -------- Package: ----------------------------
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   ------------------------------------------------
5   PACKAGE my_package IS
6      FUNCTION "+" (a, b: STD_LOGIC_VECTOR)
7         RETURN STD_LOGIC_VECTOR;
8   END my_package;
9   ------------------------------------------------
10  PACKAGE BODY my_package IS
11     FUNCTION "+" (a, b: STD_LOGIC_VECTOR)
12           RETURN STD_LOGIC_VECTOR IS
13        VARIABLE result: STD_LOGIC_VECTOR;
14        VARIABLE carry: STD_LOGIC;
15     BEGIN
```

```
16        carry := '0';
17        FOR i IN a'REVERSE_RANGE LOOP
18           result(i) := a(i) XOR b(i) XOR carry;
19           carry := (a(i) AND b(i)) OR (a(i) AND carry) OR
20                    (b(i) AND carry);
21        END LOOP;
22        RETURN result;
23     END "+";
24 END my_package;
25 ------------------------------------------------
1  --------- Main code: ------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_package.all;
5  ------------------------------------------------
6  ENTITY add_bit IS
7     PORT ( a: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8            y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
9  END add_bit;
10 ------------------------------------------------
11 ARCHITECTURE my_arch OF add_bit IS
12    CONSTANT b: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0011";
13    CONSTANT c: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
14 BEGIN
15    y <= a + b + c;    -- overloaded "+" operator
16 END my_arch;
17 ------------------------------------------------
```

- **رویه (PROCEDURE)**
  - یک رویه تا حد بسیار زیادی شبیه و یک تابع بوده و همان اهداف را دارد با این تفاوت که میتواند بیش از یک مقدار خروجی برگرداند.
  - مشابه یک تابع، برای ساخت یک رویه، دو بخش لازم است: بدنه رویه، و فراخوانی رویه
  - ***بدنه رویه (Procedure Body):***
  - نحوه تعریف:

```
PROCEDURE procedure_name [<parameter list>] IS
    [declarations]
BEGIN
    (sequential statements)
END procedure_name;
```

  - پارامترهای ورودی یک رویه، بر خلاف یک تابع، میتوانند شامل متغیر نیز باشند:

⟨parameter list⟩ = [CONSTANT] constant_name: mode type;

⟨parameter list⟩ = SIGNAL signal_name: mode type; or

⟨parameter list⟩ = VARIABLE variable_name: mode type;

  - یک رویه میتواند هر تعداد پارامتر ورودی و خروجی از نوعهای IN، OUT، و INOUT داشته باشد که هر کدام میتوانند یک سیگنال، متغیر، یا ثابت باشند. برای سیگنالهای ورودی (حالت IN) نوع پیش فرض، ثابت (CONSTANT) میباشد حال آنکه برای سیگنالهای خروجی (حالت OUT و INOUT) نوع پیش فرض، متغیر (VARIABLE) میباشد.

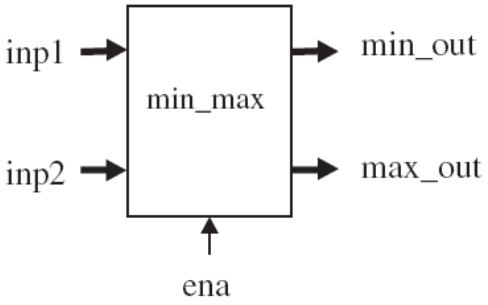
- همانطور که قبلا هم گفته شد، در مورد توابع و رویه ها، در صورت استفاده از WAIT، اعلان سیگنال، و استفاده از قطعه ها کد حاصله دیگر قابل سنتز نخواهد بود. البته در مورد رویه این استثنا وجود دارد که میتوان از اعلان سیگنال استفاده کرد اما در اینصورت رویه مربوطه باید داخل یک فرایند قرار داشته باشد.
- نکته دیگر در مورد ***فقط*** رویه ها این است علاوه بر WAIT هر نوع آشکارسازی لبه نیز در رویه ها باعث غیرقابل سنتز شدن کد میشود. به عبارت دیگر، برخلاف توابع، یک رویه قابل سنتز نباید موجب تولید رجیستر شود.
- **مثال:** رویه معرفی شده در زیر دارای سه ورودی a و b و c (و حالت IN) میباشد. a یک ثابت (CONSTANT) از نوع BIT، و b و c سیگنالهایی از نوع BIT میباشند. توجه کنید که کلمه CONSTANT برای پارامترهای ورودی قابل حذف است. این رویه دارای دو سیگنال خروجی به نامهای x (با حالت OUT و نوع BIT_VECTOR) و y (با حالت INOUT و نوع INTEGER) میباشد.

```
PROCEDURE my_procedure ( a: IN BIT; SIGNAL b, c: IN BIT;
                         SIGNAL x: OUT BIT_VECTOR(7 DOWNTO 0);
                         SIGNAL y: INOUT INTEGER RANGE 0 TO 99) IS
BEGIN
    ...
END my_procedure;
```

- ***فراخوانی رویه (Procedure Call):***
  - بر خلاف یک تابع که فراخوانی آن قسمتی از یک عبارت یا دستور میباشد، فراخوانی یک رویه خود یک عبارت یا دستور است و میتواند به تنهایی یا در ترکیب با یک دستور (ترتیبی یا همزمان) دیگر استفاده شود.
  - **مثال:** نمونه هایی از فراخوانی یک رویه

```
compute_min_max(in1, in2, 1n3, out1, out2);
            -- statement by itself

divide(dividend, divisor, quotient, remainder);
            -- statement by itself

IF (a>b) THEN compute_min_max(in1, in2, 1n3, out1, out2);
            -- procedure call associated to another statement
```

- ***محل قرارگرفتن یک رویه:*** کاملا مشابه با یک تابع است.
  - ***رویه داخل بدنه اصلی برنامه:***
  - در کدی که در ادامه می آید از یک رویه به نام sort استفاده شده است. این رویه دو عدد صحیح ۸ بیتی بدون علامت گرفته و ماکزیمم آنها را به عنوان max_out و مینیمم آنها را به عنوان min_out بر میگرداند.



```
1  ------------------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------------------------
5  ENTITY min_max IS
6     GENERIC (limit : INTEGER := 255);
7     PORT ( ena: IN BIT;
8            inp1, inp2: IN INTEGER RANGE 0 TO limit;
9            min_out, max_out: OUT INTEGER RANGE 0 TO limit);
10 END min_max;
11 ------------------------------------------------------
12 ARCHITECTURE my_architecture OF min_max IS
13    -------------------------
14    PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 TO limit;
15       SIGNAL min, max: OUT INTEGER RANGE 0 TO limit) IS
16    BEGIN
17       IF (in1 > in2) THEN
18          max <= in1;
19          min <= in2;
20       ELSE
21          max <= in2;
22          min <= in1;
23       END IF;
24    END sort;
25    -------------------------
```

```
26 BEGIN
27    PROCESS (ena)
28    BEGIN
29       IF (ena='1') THEN sort (inp1, inp2, min_out, max_out);
30       END IF;
31    END PROCESS;
32 END my_architecture;
33 ------------------------------------------------------
```

- ***رویه داخل بسته:***
- مثال: این مثال همان مثال قبلی است اما داخل یک بسته تعریف شده است؛ بنابراین، میتوان آن را در برنامه ها و پروژه های مختلف استفاده کرد یا به اشتراک گذاشت. کد نشان داده شده را یا میتوان در دو فایل جداگانه کامپایل کرد یا اینکه داخل یک فایل (در اینجا به نام min_max.vhd که همان نام ENTITY است) کامپایل کرد.

```
1  ------------ Package: -------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ------------------------------------
5  PACKAGE my_package IS
6     CONSTANT limit: INTEGER := 255;
7     PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 TO limit;
8        SIGNAL min, max: OUT INTEGER RANGE 0 TO limit);
9  END my_package;
10 ------------------------------------
```

```
11 PACKAGE BODY my_package IS
12    PROCEDURE sort (SIGNAL in1, in2: IN INTEGER RANGE 0 TO limit;
13       SIGNAL min, max: OUT INTEGER RANGE 0 TO limit) IS
14    BEGIN
15       IF (in1 > in2) THEN
16          max <= in1;
17          min <= in2;
18       ELSE
19          max <= in2;
20          min <= in1;
21       END IF;
22    END sort;
23 END my_package;
24 ------------------------------------------------

1  --------- Main code: ----------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE work.my_package.all;
5  ---------------------------------------
6  ENTITY min_max IS
7     GENERIC (limit: INTEGER := 255);
8     PORT ( ena: IN BIT;
9            inp1, inp2: IN INTEGER RANGE 0 TO limit;
10           min_out, max_out: OUT INTEGER RANGE 0 TO limit);
11 END min_max;
12 ---------------------------------------
```

```
13 ARCHITECTURE my_architecture OF min_max IS
14 BEGIN
15    PROCESS (ena)
16    BEGIN
17       IF (ena='1') THEN sort (inp1, inp2, min_out, max_out);
18       END IF;
19    END PROCESS;
20 END my_architecture;
21 ------------------------------------------------
```

- **مقایسه توابع و رویه ها بطور خلاصه**
  - یک تابع دارای تعداد صفر یا بیشتر ورودی و یک خروجی است و ورودیها میتوانند فقط از نوع CONSTANT یا SIGNAL باشند.
  - یک رویه میتواند شامل هر تعداد دلخواه پارامتر ورودی از نوع IN، OUT، و یا INOUT و هر کدام نیز میتواند از نوع SIGNAL، CONSTANT، و یا VARIABLE باشد. پارامترهای ورودی (با حالت IN) بطور پیش فرض از نوع CONSTANT و پارامترهای خروجی (با حالت OUT یا INOUT) نیز بطور پیش فرش از نوع VARIABLE هستند.
  - یک تابع فقط میتواند به عنوان جزئی از یک دستور یا عبارت فراخوانی شود اما یک رویه میتواند بطور مستقل و جداگانه نیز فراخوانی شود.
  - در هر دو (تابع و رویه) استفاده از WAIT و COMPONENT باعث غیرقابل سنتز شدن کد میشود.
  - محلهای قرارگرفتن هردو (تابع و رویه) یکی است.

- **دستور ASSERT**
  - این دستور یک دستور غیرقابل سنتز است و وظیفه آن نمایش یک پیام (روی صفحه نمایش) در مواقعی است که خطایی در حین شبیه سازی رخ دهد. بسته به شدت و اهمیت خطا میتوان شبیه ساز را متوقف ساخت.
  - گرامر این دستور:

```
ASSERT condition
[REPORT "message"]
[SEVERITY severity_level];
```

  - منظور از severity_level شدت و اهمیت خطا است و مقدار آن میتواند Note، Warning، Error (مقدار پیش فرض)، و یا Failure باشد. هر گاه شرط مشخص شده توسط condition نادرست (FALSE) باشد، پیغام مشخص شده در "message" نمایش داده میشود.
  - **مثال:** فرض کنید تابعی نوشته ایم که دو عدد ورودی که دارای تعداد بیتهای یکسانی باشند را با هم جمع کند. برای چک کردن برقراری این شرط، میتوان دستور زیر را در بدنه تابع قرار داد.

```
ASSERT a'LENGTH = b'LENGTH
REPORT "Error: vectors do not have same length!"
SEVERITY failure;
```

  - توجه کنید که استفاده از دستور ASSERT موجب تولید سخت افزار نمیشود و فقط به منظور شبیه سازی استفاده میشود. کامپایلر با دیدن این دستور، آن را نادیده میگیرد (برای سنتز کردن).

دانشگاه صنعتی شاهرود
دانشکده برق

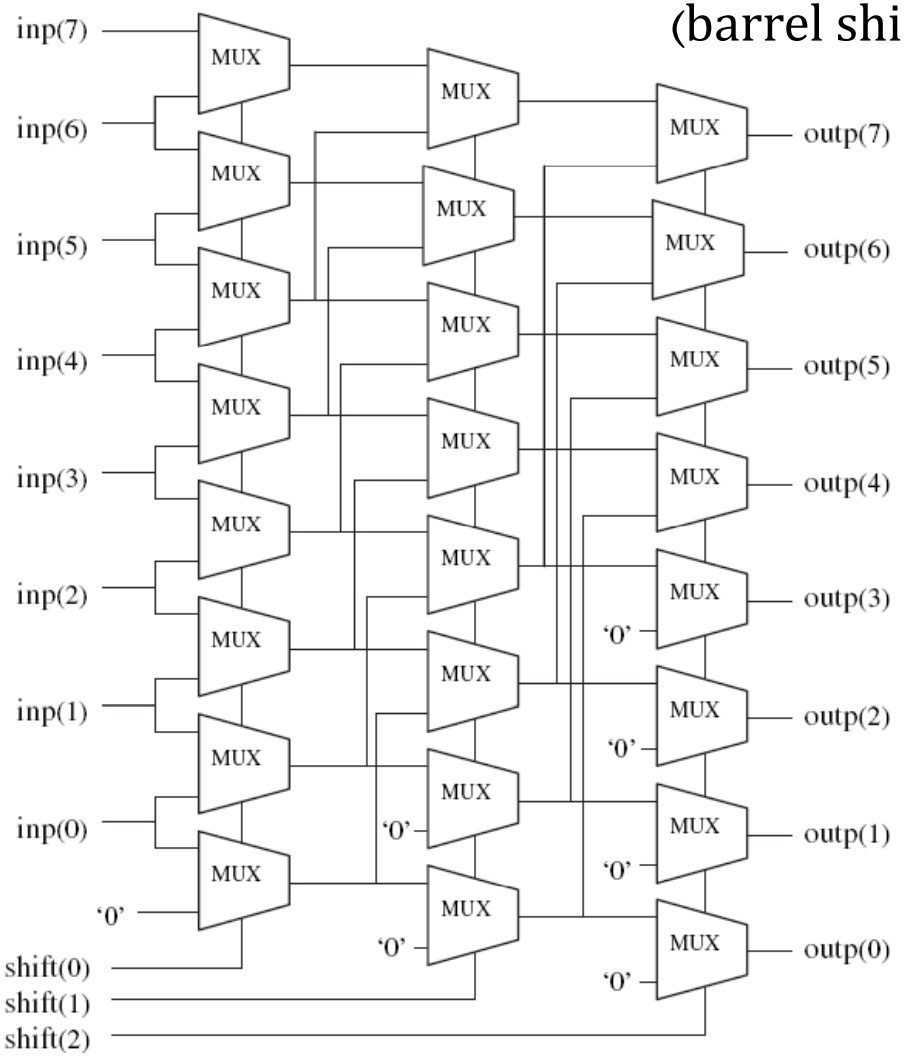
عنوان درس

Programmable Logic Devices (PLD)
مدارات منطقی برنامه پذیر

استاد درس : دکتر گرایلو
ترم بهار ۸۹-۱۳۸۸

فصل دوازدهم : برخی مثالها

- در این فصل کاربردهای بیشتری در قالب برخی مثالها ارائه می شود.
- **مثال ۱:** شیفت دهنده زنجیری (barrel shifter)
- **دیاگرام :**



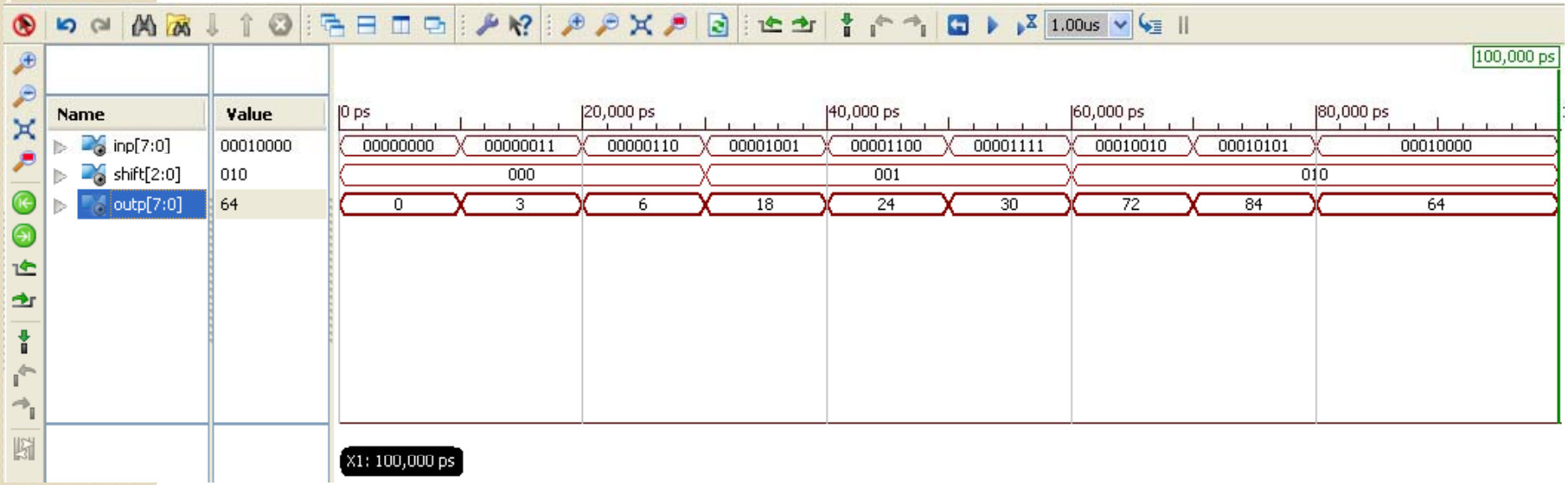
- ورودی ۸ بیتی و خروجی نیز ۸ بیتی و در عین حال معادل شیفت یافته ی عدد ورودی است. مقدار شیفت توسط عدد سه بیتی shift تعیین می شود. این شیفت دهنده شامل سه طبقه شیفت دهنده است که به هرکدام یک شیفت دهنده ساده گفته می شود. از هر طبقه به طبقه بعدی که برویم تعداد صفرهای متصل به آن دو برابر می شود. اگر shift=001 باشد آنگاه فقط اولین طبقه شیفت می دهد و اگر shift=111 باشد آنگاه هر سه طبقه شیفت می دهند.
- کد VHDL

```
1  entity Ex91 is
2      PORT (     inp: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
3                 shift: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
4                   outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
5  end Ex91;
6
7  architecture Behavioral of Ex91 is
8
9  begin
10 PROCESS (inp, shift)
11     VARIABLE temp1: STD_LOGIC_VECTOR (7 DOWNTO 0);
12     VARIABLE temp2: STD_LOGIC_VECTOR (7 DOWNTO 0);
13 BEGIN
```

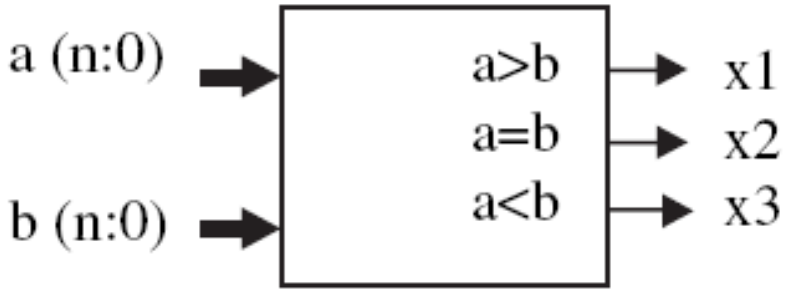
```
14    ---- 1st shifter -----
15        IF (shift(0)='0') THEN
16            temp1 := inp;
17        ELSE
18        temp1(0) := '0';
19        FOR i IN 1 TO inp'HIGH LOOP
20            temp1(i) := inp(i-1);
21        END LOOP;
22        END IF;
23    ---- 2nd shifter -----
24        IF (shift(1)='0') THEN
25            temp2 := temp1;
26        ELSE
27        FOR i IN 0 TO 1 LOOP
28            temp2(i) := '0';
29        END LOOP;
30        FOR i IN 2 TO inp'HIGH LOOP
31            temp2(i) := temp1(i-2);
32        END LOOP;
```

```
33      END IF;
34  ---- 3rd shifter -----
35      IF (shift(2)='0') THEN
36          outp <= temp2;
37      ELSE
38      FOR i IN 0 TO 3 LOOP
39          outp(i) <= '0';
40      END LOOP;
41      FOR i IN 4 TO inp'HIGH LOOP
42          outp(i) <= temp2(i-4);
43      END LOOP;
44      END IF;
45  END PROCESS;
46
47
48  end Behavioral;
```

- نتیجه شبیه سازی



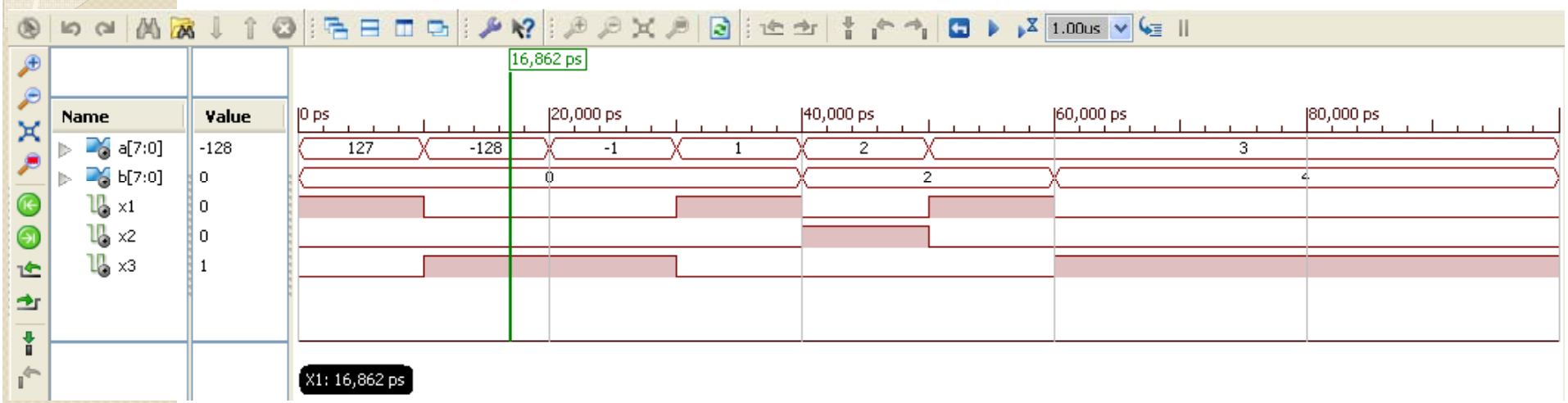
- **مثال ۲ :** مقایسه گر اعداد علامتدار و بدون علامت



- اندازه (یا طول) اعداد ورودی، دلخواه و به صورت کلی برابر n+1 است.
- **حالت اول:** مقایسه با فرض علامتدار بودن
  - در این حالت به استفاده از بسته std_logic_arith در کد مربوطه توجه کنید. این بسته برای کار با انواع signed و unsigned ضروری است.

```
1  ---- Signed Comparator: ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all; -- necessary!
5  ENTITY comparator IS
6  GENERIC (n: INTEGER := 7);
7  PORT (a, b: IN SIGNED (n DOWNTO 0);
8  x1, x2, x3: OUT STD_LOGIC);
9  END comparator;
10 ----------------------------------------
11 ARCHITECTURE signed OF comparator IS
12 BEGIN
13     x1 <= '1' WHEN a > b ELSE '0';
14     x2 <= '1' WHEN a = b ELSE '0';
15     x3 <= '1' WHEN a < b ELSE '0';
16 END signed;
```

- نتیجه شبیه سازی

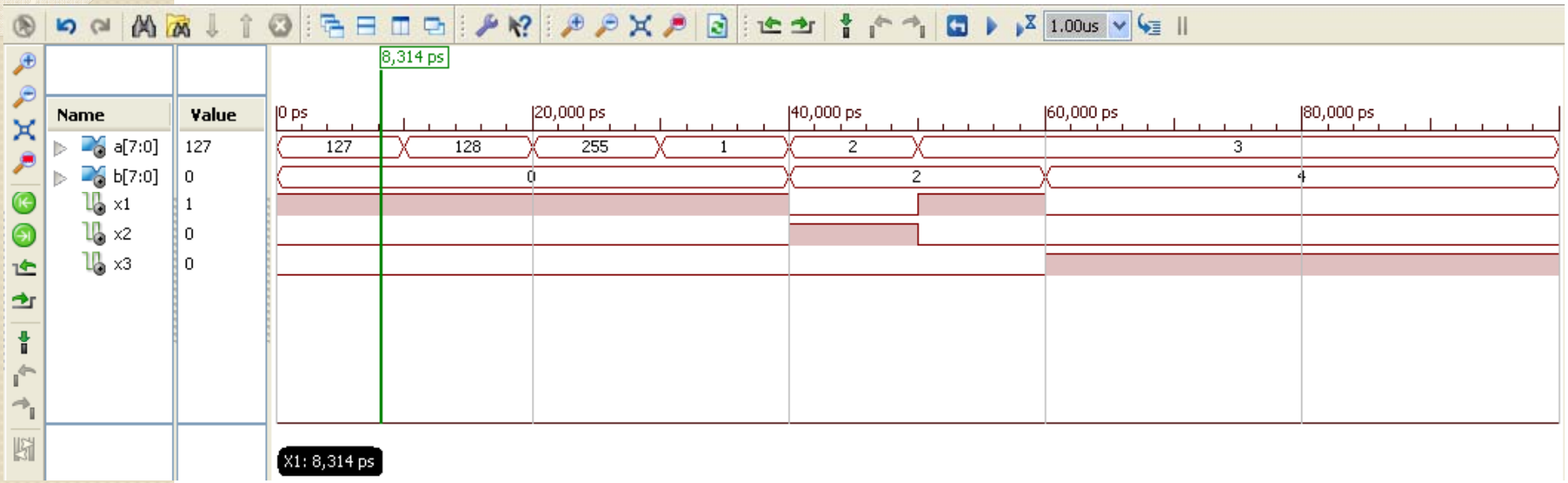


- **حالت دوم :** مقایسه با فرض بدون علامت بودن
  - به نکته قبلی همچنان توجه کنید.

- کد VHDL

```
1  ---- Unsigned Comparator: ----------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  USE ieee.std_logic_arith.all; -- necessary!
5  ----------------------------------------
6  ENTITY comparator IS
7  GENERIC (n: INTEGER := 7);
8  PORT (a, b: IN UNSIGNED (n DOWNTO 0);
9  x1, x2, x3: OUT STD_LOGIC);
10 END comparator;
11 ----------------------------------------
12 ARCHITECTURE signed OF comparator IS
13 BEGIN
14     x1 <= '1' WHEN a > b ELSE '0';
15     x2 <= '1' WHEN a = b ELSE '0';
16     x3 <= '1' WHEN a < b ELSE '0';
17 END signed;
18 ----------------------------------------
```

- نتیجه شبیه سازی

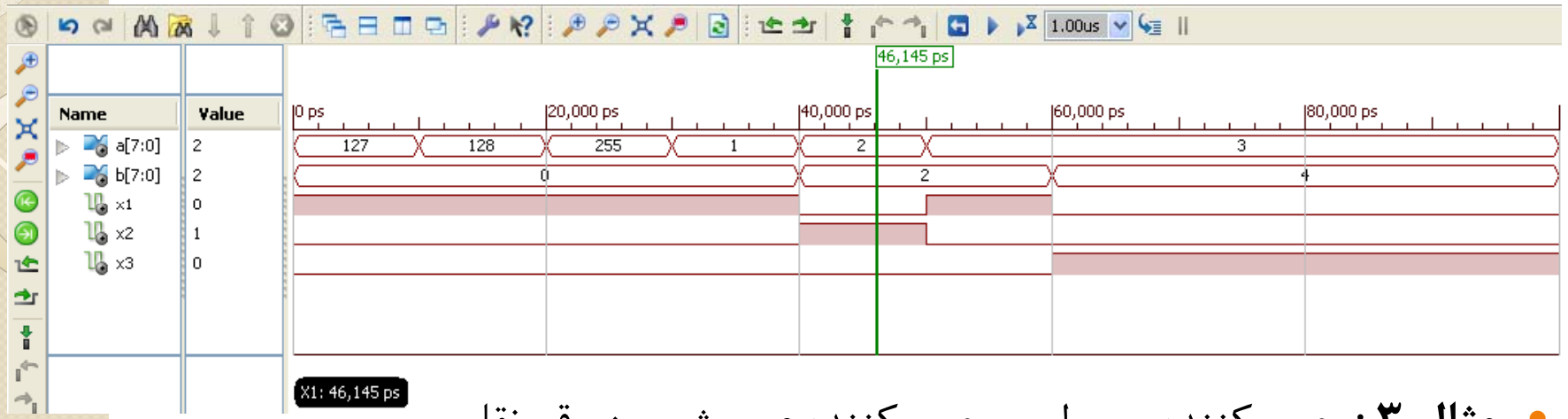


- می توانیم از نوع STD_LOGIC_VECTOR نیز برای مقایسه در حالت بدون علامت استفاده کنیم. در این حالت نیازی به استفاده از بسته std_logic_arith نداریم. کد مربوطه و نتیجه شبیه سازی در ادامه آمده است.

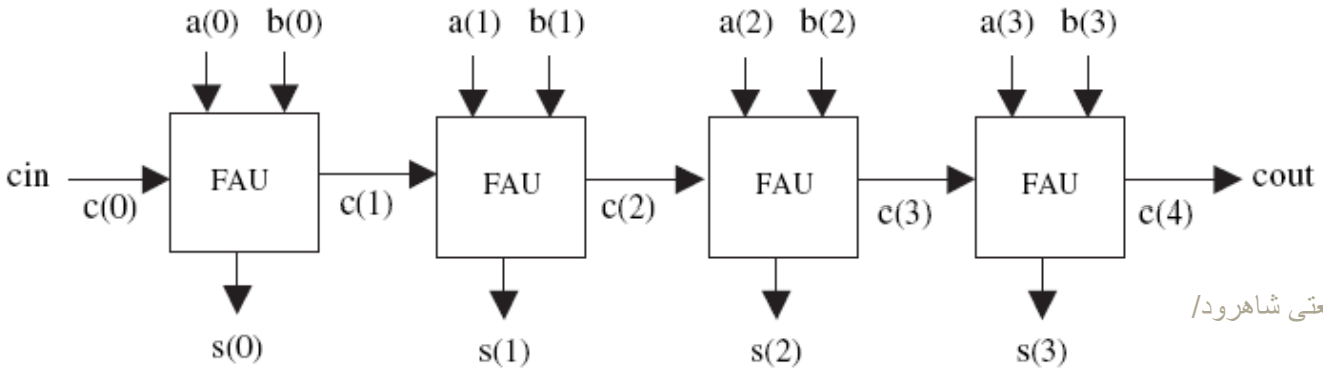
- کد VHDL

```
1  ---- Unsigned Comparator #2: ----------------
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------
5  entity Ex92p3 is
6      GENERIC (n: INTEGER := 7);
7      PORT (a, b: IN STD_LOGIC_VECTOR (n DOWNTO 0);
8            x1, x2, x3: OUT STD_LOGIC);
9  end Ex92p3;
10 ----------------------------------------
11 architecture Behavioral of Ex92p3 is
12
13 begin
14     x1 <= '1' WHEN a > b ELSE '0';
15     x2 <= '1' WHEN a = b ELSE '0';
16     x3 <= '1' WHEN a < b ELSE '0';
17 end Behavioral;
18 ----------------------------------------
```

- نتیجه شبیه سازی



- **مثال ۳ :** جمع کننده معمولی و جمع کننده ی پیش بین رقم نقلی
  - اولی سخت افزار کمتر دارد اما دومی سریعتر است.
- **جمع کننده معمولی** (Carry Ripple Adder) :
  - شامل تعدادی تمام جمع کننده (FAU) است. خروجی s همان تابع توازن و خروجی cout تابع اکثریت (majority function) است.



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

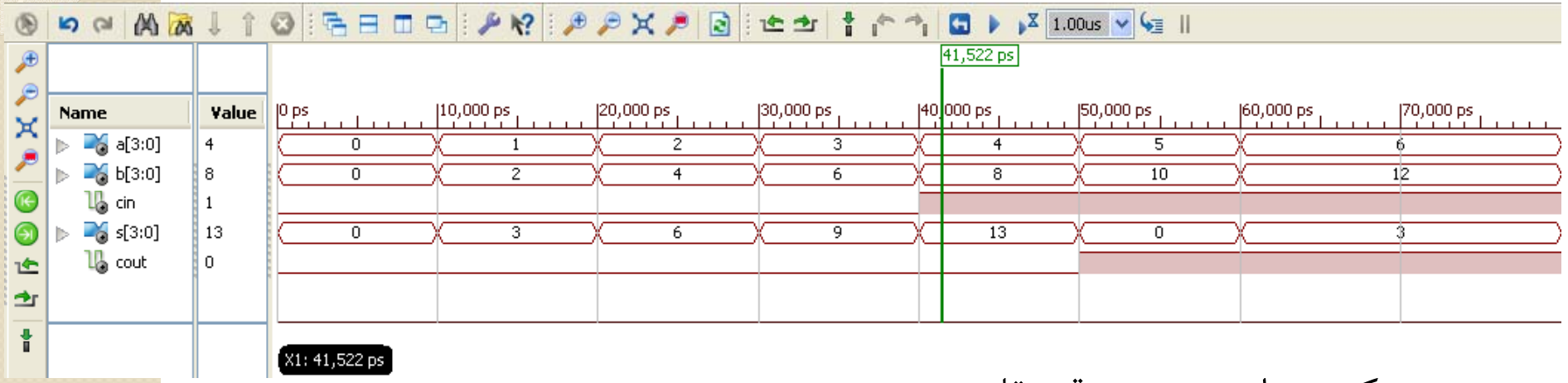
- کد VHDL

s = a XOR b XOR cin

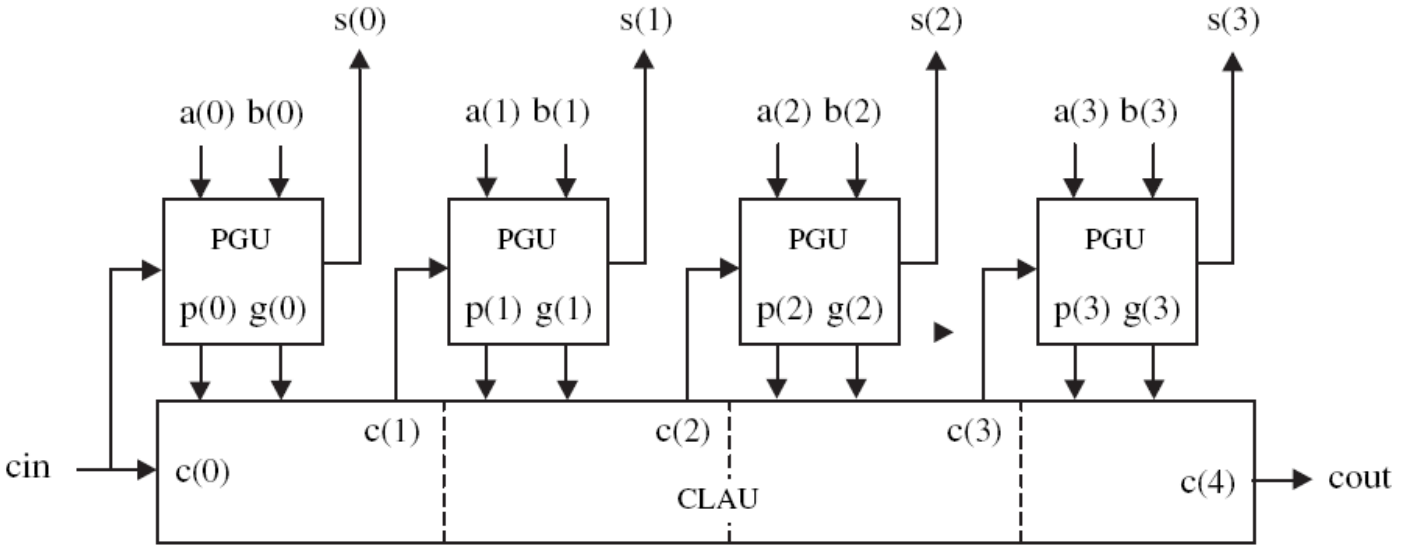
cout = (a AND b) OR (a AND cin) OR (b AND cin)

```
1  ----------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ----------------------------------------
5  ENTITY adder_cripple IS
6      GENERIC (n: INTEGER := 4);
7      PORT ( a, b: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
8             cin: IN STD_LOGIC;
9               s: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0);
10           cout: OUT STD_LOGIC);
11 END adder_cripple;
12 ----------------------------------------
13 architecture Behavioral of adder_cripple is
14     SIGNAL c: STD_LOGIC_VECTOR (n DOWNTO 0);
15 BEGIN
16     c(0) <= cin;
17     G1: FOR i IN 0 to n-1 GENERATE
18         s(i) <= a(i) XOR b(i) XOR c(i);
19         c(i+1) <= (a(i) AND b(i)) OR
20                   (a(i) AND c(i)) OR
21                   (b(i) AND c(i));
22     END GENERATE;
23     cout <= c(n);
24 end Behavioral;
25 ----------------------------------------
```

- نتیجه شبیه سازی



- جمع کننده با پیش بین رقم نقلی



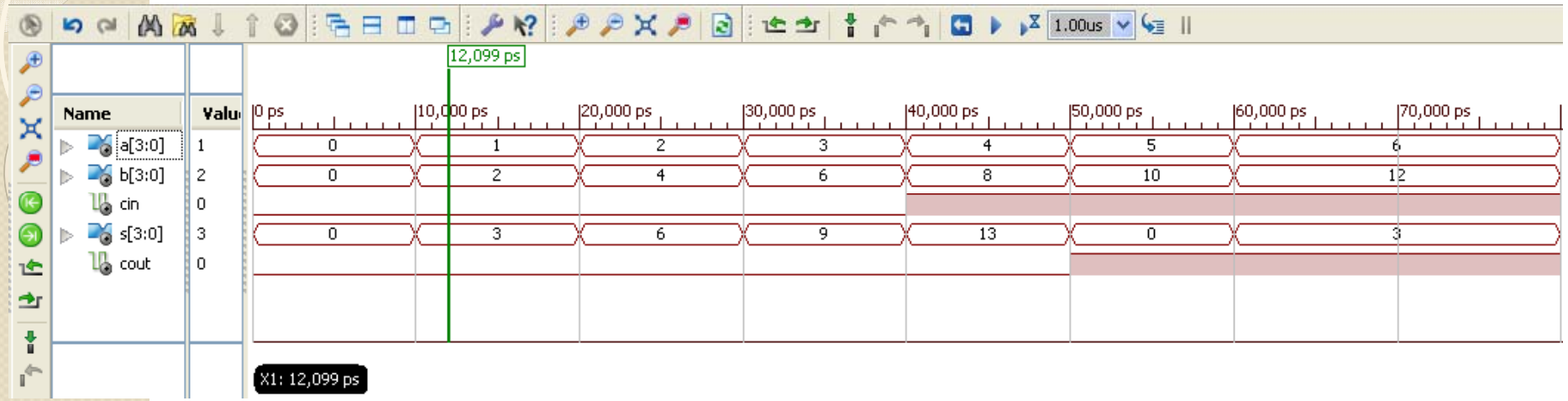
$$c(0) \equiv cin$$
$$c(1) = c(0)p(0) + g(0)$$
$$c(2) = c(0)p(0)p(1) + g(0)p(1) + g(1)$$
$$c(3) = c(0)p(0)p(1)p(2) + g(0)p(1)p(2) + g(1)p(2) + g(2)$$

- در این جمع کننده حجم سخت افزار به سرعت رشد میکند
- برای ساخت جمع کننده های با بیتهای بیشتر بهتر است از کسکود جمع کننده های ۴بیتی که در این مثال آمده استفاده شود.
- کد VHDL

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  ---------------------------------------------
4  ENTITY CLA_Adder IS
5      PORT ( a, b: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
6               cin: IN STD_LOGIC;
7                 s: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
8              cout: OUT STD_LOGIC);
9  END CLA_Adder;
10 ---------------------------------------------
11 ARCHITECTURE CLA_Adder OF CLA_Adder IS
12     SIGNAL c: STD_LOGIC_VECTOR (4 DOWNTO 0);
13     SIGNAL p: STD_LOGIC_VECTOR (3 DOWNTO 0);
14     SIGNAL g: STD_LOGIC_VECTOR (3 DOWNTO 0);
15 begin
```

```
16  -------PGU: ------------------------------------
17      G1: FOR i IN 0 TO 3 GENERATE
18          p(i) <= a(i) XOR b(i);
19          g(i) <= a(i) AND b(i);
20          s(i) <= p(i) XOR c(i);
21      END GENERATE;
22  ------- CLAU: ------------------------------------
23      c(0) <= cin;
24      c(1) <= (cin AND p(0)) OR
25          g(0);
26      c(2) <= (cin AND p(0) AND p(1)) OR
27          (g(0) AND p(1)) OR
28              g(1);
29      c(3) <= (cin AND p(0) AND p(1) AND p(2)) OR
30          (g(0) AND p(1) AND p(2)) OR
31              (g(1) AND p(2)) OR
32                  g(2);
33      c(4)<=(cin AND p(0) AND p(1) AND p(2) AND p(3)) OR
34          (g(0) AND p(1) AND p(2) AND p(3)) OR
35              (g(1) AND p(2) AND p(3)) OR
36                  (g(2) AND p(3)) OR
37                      g(3);
38      cout <= c(4);
39  end Behavioral;
```

- نتیجه شبیه سازی (مشابه قبلی است)



- **مثال ۴ :** تقسیم صحیح a/b
- **روش اول :** گام به گام

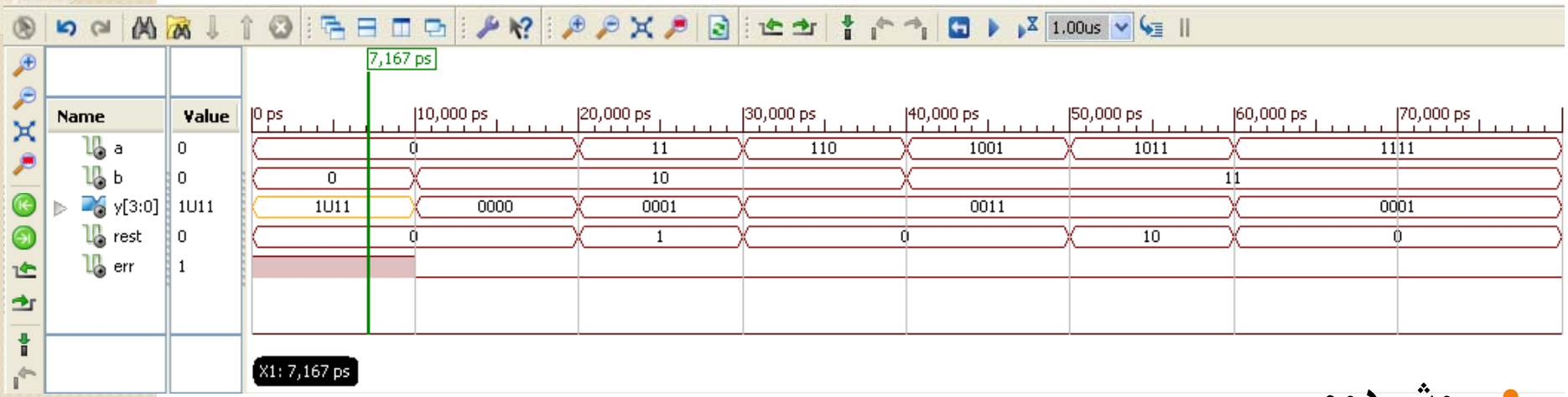
Index (i)	a-related input (a_inp)	Comparison	b-related input (b_inp)	y (quotient)	Operation on 1st column
3	1011	<	0011000	0	none
2	1011	<	0001100	0	none
1	1011	>	0000110	1	a_inp(i)-b_inp(i)
0	0101	>	0000011	1	a_inp(i)-b_inp(i)
	0010 (rem)				

- کد VHDL

```
1  ----- Solution 1: Step-by-Step -------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------
5  ENTITY divider IS
6      PORT ( a,b: IN INTEGER RANGE 0 to 15;
7                  y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
8             rest: OUT INTEGER RANGE 0 to 15;
9             err : OUT STD_LOGIC);
10 END divider;
11 ---------------------------------------
12 ARCHITECTURE rtl OF divider IS
13 BEGIN
14     process (a,b)
15         VARIABLE temp1: INTEGER RANGE 0 to 15;
16         VARIABLE temp2: INTEGER RANGE 0 to 15;
17     begin
18 ----- Error and initialization: -------
19         temp1 := a;
20         temp2 := b;
21         IF (b=0) THEN err <= '1';
22         ELSE err <= '0';
23         END IF;
24 ----- y(3): ----------------------------
```

```
25          IF (temp1 >= temp2 * 8) THEN
26              y(3) <= '1';
27              temp1 := temp1 - temp2*8;
28          ELSE y(3) <= '0';
29          END IF;
30      ----- y(2): ----------------------------
31          IF (temp1 >= temp2 * 4) THEN
32              y(0) <= '1';
33              temp1 := temp1 - temp2 * 4;
34          ELSE y(2) <= '0';
35          END IF;
36      ----- y(1): ----------------------------
37          IF (temp1 >= temp2 * 2) THEN
38              y(1) <= '1';
39              temp1 := temp1 - temp2 * 2;
40          ELSE y(1) <= '0';
41          END IF;
42      ----- y(0): ----------------------------
43          IF (temp1 >= temp2) THEN
44              y(0) <= '1';
45              temp1 := temp1 - temp2;
46          ELSE y(0) <= '0';
47          END IF;
48      ----- Remainder: ---------------------
49          rest <= temp1;
50
51      end process;
52  END rtl;
```

- نتیجه شبیه سازی



- **روش دوم**
- اگر b دارای n+1 بیت باشد، b را به چپ به اندازه n بیت شیفت داده و اگر a از آن بزرگتر باشد y(n)=1 قرار داده و مقدار b را از a کم می کنیم تا مقدار جدید b به دست آید. در غیر این صورت y(n)=0 قرار میدهیم. حال در مرحله بعد b را یک بیت به راست شیفت می دهیم و همان کارها را تکرار میکنیم.
- کد VHDL روش دوم، بر خلاف روش اول، عمومی (یا GENERIC) بوده و قابل تعمیم به هر مقدار دلخواه از n است.

```vhdl
------ Solution 2: compact and generic -----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY divider IS
    GENERIC(n: INTEGER := 3);
    PORT ( a, b: IN INTEGER RANGE 0 TO 15;
                 y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
              rest: OUT INTEGER RANGE 0 TO 15;
              err : OUT STD_LOGIC);
END divider;
-----------------------------------------
ARCHITECTURE rtl OF divider IS
BEGIN
    PROCESS (a,b)
        VARIABLE temp1: INTEGER RANGE 0 TO 15;
        VARIABLE temp2: INTEGER RANGE 0 TO 15;
    BEGIN
----- Error and initialization: -------
        temp1 := a;
        temp2 := b;
        IF (b = 0) THEN err <= '1';
        ELSE err <= '0';
        END IF;
```

```
25          FOR i IN n DOWNTO 0 LOOP
26              IF(temp1 >= temp2 * 2**i) THEN
27                  y(i) <= '1';
28                  temp1 := temp1 - temp2 * 2**i;
29              ELSE y(i) <= '0';
30              END IF;
31          END LOOP;
32  ----- Remainder: ----------------------
33          rest <= temp1;
34      END PROCESS;
35  END rtl;
36  ----------------------------------------
```

- نتیجه شبیه سازی