

# جزوه آموزشی VHDL

تهیه کنندگان:  
حسین برهانی فر  
غلامرضا جهانی پور  
رضا شعبانعلی نژاد

بهمن ماه 1390

## فهرست

4	مقدمه
---	-------

## فصل اول: ساختار کلی VHDL

7	1-1- مقدمه
7	2-1- یک مثال طراحی
9	3-1- طراحی Entities
10	1-3-1 تعریف Entity
11	2-3-1 پورتهای
12	3-3-1 Architecture
17	4-1- مشخصه ها ، Data type ، Data object و نسبتها
17	1-4-1 مشخصه ها
18	2-4-1 Data Object
19	3-4-1 نوع داده ها
23	5-1- خطاها

## فصل دوم: ایجاد مدارهای ترکیبی و ترتیبی

28	1-2- مقدمه
28	2-2- مدارهای ترکیبی
38	1-2-2 دستورات همزمان
39	2-2-2 دستورات ترتیبی
46	3-2- مدارات منطقی سنکرون
51	1-3-2 Reset در مدارات سنکرون
53	2-3-2 Reset و preset آسنکرون
56	3-3-2 سیگنالهای دو طرفه و بافرهای سه حالته
60	4-3-2 ساختارهای دوطرفه ها و سه حالته ها
62	4-2- طراحی یک FIFO
63	5-2- حلقه ها

## فصل سوم: طراحی ماشین حالت

- 69 3-1- مقدمه
- 69 3-2- یک مثال ساده طراحی
- 69 3-2-1- روش طراحی رایج
- 71 3-2-2- ماشین حالت در VHDL
- 75 3-3- طراحی یک کنترلر حافظه
- 76 3-3-1- تبدیل دیاگرام حالت به VHDL
- 83 3-3-2- یک ساختار کد دیگر

## فصل چهارم: توابع و رویه ها

- 88 4-1- توابع
- 89 4-2- نوع توابع تبدیل
- 92 4-3- استفاده از توابع در ساده سازی Component ها
- 94 4-4- بیان توابع نامعلوم
- 95 4-5- استفاده از توابع
- 98 4-6- اپراتورهای فراخوانی
- 101 4-7- توابع فراخوانی
- 105 4-8- توابع استاندارد

## مقدمه

آی‌سی‌های FPGA که در زمینه الکترونیک دیجیتال استفاده می‌شوند، یکی از تکنولوژی‌هایی است که در سالهای اخیر کاربرد وسیعی در پروژه‌های صنعتی خصوصا در مدارات فرکانس بالا پیدا کرده است و در بسیاری از پروژه‌ها قابل رقابت با روشهای میکروپروسسوری و میکروکنترلی و در بعضی جاها نیز بسیار بهتر عمل می‌کند. آی‌سی‌های FPGA که توسعه یافته آی‌سی‌های قابل برنامه پذیر قبلی از جمله PAL, PLA, GAL, ... می‌باشد نیاز داشت تا روشهای بهتر و ساده‌تری غیر از شماتیک جهت طراحی ارائه شود. چرا که در طرحهای پیچیده طراحی با شماتیک بسیار مشکل می‌شود. بدین منظور اساتید محترمی در این زمینه گام برداشته و زبان برنامه نویسی سخت افزاری را بنا نهادند تا طراحان بتوانند با این روش در کنار روش شماتیک، طراحی خود را راحت تر انجام دهند. یکی از زبانهای برنامه نویسی که بصورت استاندارد نیز می‌باشد زبان VHDL می‌باشد و امروزه یکی از روشهای مهم و بسیار قدرتمند در زمینه طراحی می‌باشد. با توجه به کاربردی بودن این مبحث، بر آن شدیم تا یکی از منابع را بطور خلاصه و مفید و بصورت جزوه‌ای در اختیار علاقمندان به این مبحث قرار بگیرد.

در این جزوه مطالب در چهار فصل ارائه شده‌اند و جهت آشنایی بیشتر خوانندگان محترم با مطالب موجود توضیح مختصری در مورد هر فصل داده می‌شود.

در فصل اول ساختار کلی برنامه نویسی به زبان VHDL مطرح شده و قسمتهای مختلف آن یعنی Entity و Architecture با جزییات مربوطه و یک سری اصول اولیه برنامه نویسی با ذکر مثالهای ساده توضیح داده شده است.

روش طراحی مدارات ترکیبی و ترتیبی (سنکرون و آسنکرون) در فصل دوم ارائه گردیده و در این میان انواع دستورات همزمان و ترتیبی و حلقه‌ها با ذکر مثال معرفی شده‌اند. در پایان این فصل با ذکر معرفی سیگنالهای دو طرفه و سه حالت، به عنوان یک مثال پیچیده‌تر، طراحی یک حافظه FIFO و نحوه دسترسی به آن، مطرح شده است.

در فصل سوم روش دیاگرام حالت بیان شده است. از آنجاییکه زبان برنامه نویسی سخت افزاری بدلیل ساختاری با سایر زبانهای برنامه نویسی تفاوت اساسی دارد، روش فلوچارت چندان مناسب نیست و نمی‌تواند رفتار مدار را بدرستی بیان کند. ولی روش دیاگرام حالت برای این

منظور مناسب می‌باشد که در این فصل این روش بطور کامل با ذکر مثال معرفی شده است.

در فصل چهارم به بیان توابع، رویه‌ها و نحوه تعریف و استفاده از آنها در زبان VHDL پرداخته شده است. از توابع و رویه‌ها می‌توان جهت ساده‌سازی مدارات و دسته بندی برنامه‌ها استفاده کرد. در پایان با امید به اینکه این جزوه مورد استفاده علاقمندان به این مبحث قرار بگیرد، از خوانندگان محترم خواهشمندیم که در صورت برخورد هر گونه ایراد و اشکال، نظرات خود را در سایت [www.fpgagroup.com](http://www.fpgagroup.com) ارائه نمایند و قطعاً انتقادات و پیشنهادات شما عزیزان باعث ارائه بهتر این جزوه و کارهای بعدی خواهد شد.

حسین برهانی فر

غلامرضا جهانی پور

رضا شعبانعلی نژاد

## بخش اول

# Entity و Architecture

## 1-1- مقدمه

در این قسمت بلوکهای اصلی زبان VHDL که شامل Architecture , entity می باشد. با جزئیات کامل همراه با مثالهایی توضیح داده می شود.

## 1-2- یک مثال طراحی :

### مقایسه کننده چهار بیتی :

در مثال 1-1 کدهای VHDL مربوط به یک مقایسه کننده چهار بیتی آورده شده است که قسمتهای مختلف آن را توضیح می دهیم .

```

1- library ieee;
2- use ieee.std_logic_1164.all;
3- use ieee.std_logic_unsigned.all;
-----
4- --eqcomp4 is a four bit equality comparator
5- entity eqcomp4 is
6-     port(a,b :in bit_vector(3 downto 0);
7- equals :out bit); --equals is active high
8- end eqcomp4 ;
9- architecture dataflow of eqcomp4 is
10- begin
11-     equals <='1' when ( a=b ) else'0' ;
12- end dataflow;
```

### لیست 1-1

همانطوریکه دیده می شود در این برنامه سه بخش اساسی وجود دارد. اول Library دوم توضیح entity و سوم بدنه Architecture وجود دارد. در اول برنامه کتابخانه یا Library مورد استفاده شده توسط برنامه را باید معرفی کنیم البته در فصول بعدی کتابخانه را معرفی می کنیم.

در زبان VHDL یک سری کلمات کلیدی وجود دارند که هر کدام معنی خاص خود را دارد .

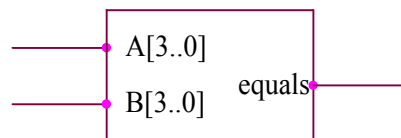
دو خط ( -- ) که در خط چهارم برنامه آمده برای ارائه توضیحاتی در مورد برنامه می باشد و تاثیری در Compile برنامه و طراحی ندارد . با

این کار می توان روند طراحی مدار را گزارش بدهیم . بنابراین هر کجا خواستیم گزارش بنویسیم دو خط گذاشته و تا پایان آن خط به عنوان گزارش محسوب می شود . این دو خط می تواند در طول یک خط و بعد از یک کد VHDL همانند گزارشی که در خط چهارم برنامه وجود دارد قرار بگیرد . در ضمن اگر گزارش بیش از یک خط باشد در ابتدای هر خطی باید دو خط قرار بگیرد .

خط 5 تا 8 ورودیها و خروجیهای مقایسه کننده چهار بیتی که با برنامه eqcomp4 است را مشخص می کند . خط 5 و 8 شروع و پایان entity برای eqcomp4 را مشخص می کنند .

خط 6 با یک Port شروع شده است و با پرانتزی که در جلو آن مشخص شده است می توان ورودیها و خروجیها را معرفی کنیم و بعد از پرانتز ؛ قرار بگیرد . در حقیقت Port ارتباطات مربوط به entity را مشخص می کند .

در خط 6 این مثال، دو پورت a و b معرفی شده اند که بصورت باس یا bit-vector چهار بیتی می باشد هر شماره ای از این bit\_vector بطور مثال (0) a به عنوان یک بیت بوده و می تواند مقادیر '0' یا '1' را بپذیرد . ( در طرفین اعداد یک بیتی باید یک quote قرار بگیرد و اگر عدد بیش از یک عدد باشد باید دو quote قرار بگیرد بطور مثال ("0101") . در خط 7 equals معرفی شده که به عنوان خروجی مدار می باشد . پس به طور کلی entity پایه های ورودی و خروجی مدار شما را در نظر می گیرد و به چیز دیگری کار ندارد .  
نمای شماتیکی که entity مشخص می کند بصورت شکل 1-1 می باشد .



شکل 1-1 : نمای شماتیکی از طرح entity

خطوط 9 تا 12 مشخص می کنند که entity مربوطه باید چه کاری انجام دهد . که این کار در بدنه Architecture تعریف می شود . این عملیات با یک کلمه کلیدی Architecture در خط 9 شروع و با END در خط 12 پایان یافته است .



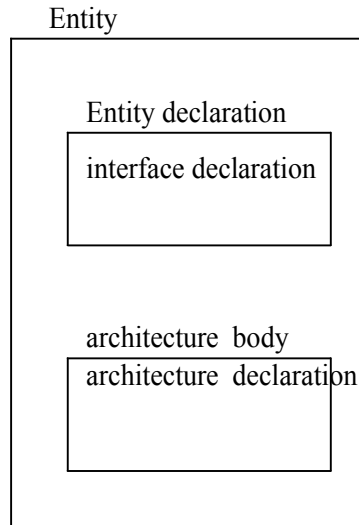
در خط 9 يك نام (dataflow) براي Architecture از entity مربوط به eqcomp4 معرفي شده است. نامي كه انتخاب مي شود اختياري مي باشد در اين مثال اين نام data flow معرفي شده براي اينكه روش معرفي Architecture به روش data flow مي باشد .

در فصلهاي بعدي روش هاي مختلف Architecture را توضيح مي دهيم .  
بدنه Architecture با يك كلمه كليدي بنام begin كه در خط 10 مشخص شده شروع مي گردد . در خط 11 عملياتي كه بايد صورت بگيرد تعريف شده است . اين Architecture ساده تساوي مقايسه كننده را معرفي مي كند و بيان مي كند در صورتي كه مقدار دو باس ورودي a,b با هم برابر بود آنگاه سيگنال خروجي equals برابر '1' و در غير اينصورت برابر '0' مي شود . نماد  $=$  برابر مقدار دهی به يك سيگنال استفاده مي شود . عمل مقايسه از سمت چپ به سمت راست صورت مي گيرد . يعني  $a(3)$  با  $b(3)$  و  $a(2)$  با  $b(2)$  و الي آخر عمل مقايسه صورت مي گيرد .

در اين مثال براي معرفي كردن باس يا bit\_vector از كلمه كليدي down to استفاده شده است (( bit\_vector (3 Downto 0) در اين روش ما مي خواسته ايم كه بيت با ايندكس بالاتر داراي با ارزش ترين بيت (MSB) باشد. در اين مثال  $a(3)$  و  $b(3)$  داراي با ارزش ترين بيت هستند . اگر اين باسها بصورت bit\_vector ( 0 to 3) معرفي مي شوند در اين حال  $a(3)$  و  $b(3)$  به عنوان كم ارزش ترين بيت (LSB) مشخص مي شوند .

### 3-1- طراحی Entities

طراحي كه به زبان VHDL صورت مي گيرد، شامل يك entity و يك Architecture است كه در شكل زير نشان داده شده است .



شکل 1-2: رابطه بین تعریف entity و بدنه Architecture از یک طراحی

### 1-3-1- تعریف Entity

در قسمت entity ورودی ها و خروجی های مدار مشخص می شوند . برای مثال یک کد VHDL برای قسمت Entity از یک Adder چهار بیتی در زیر نوشته شده و نمای شماتیکی آن در شکل 1-3 نشان داده شده است . a,b به عنوان دو عدد چهار بیتی در ورودی باید عمل جمع روی آنها صورت بگیرد . ci به عنوان عدد carry ورودی می باشد . Sum نتیجه جمع را نشان می دهد . و Co هم به عنوان Carry خروجی می باشد .

entity add4 is

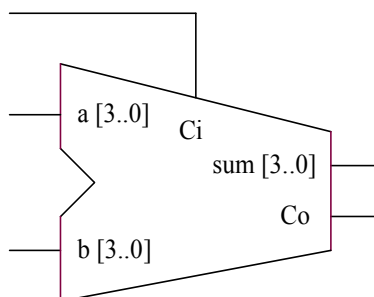
port(a,b : in std\_logic\_vector(3 downto 0) ;

ci : in std\_logic ;

sum:out std\_logic\_vector(3 downto 0) ;

co :out std\_logic );

end add4;



شکل 1-3: نمای معادل از Adder چهار بیتی

### 1-3-2- پورت ها

هر سیگنالی که در قسمت **entity** تعریف می شود در قسمت پورت قرار می گیرد که در حقیقت به عنوان یک پین در نمای شماتیکی است. هر پورتی که در این قسمت تعریف می شود ابتدا باید مد آن سیگنال و سپس نوع **data** را مشخص کنیم .

#### مد (MODE) :

مد، جهت **data** که روی آن خط انتقال قرار می گیرد را مشخص می سازد. مد می تواند چهار نوع مختلف باشد : **in, out, inout, buffer** .

اگر برای یک پورتی مد آن را مشخص نسازیم ، بصورت پیش فرض مد آن **in** قرار می گیرد. حال به توضیح این مدها می پردازیم :

**In** : در این مد جهت **Data** بسمت داخل **entity** است. مقدار دهی یا راه اندازی این پورت در خارج از **entity** یا خارج از مدار مورد طراحی انجام می شود. معمولاً از این مد برای ورودی کلاک، ورودیهای کنترلی ( نظیر **load, reset, enable** ) و **data** هایی که فقط به عنوان ورودی هستند استفاده می گردد .

**OUT** : در این مد، جهت **data** به سمت خارج از **entity** است و راه اندازی این پورت توسط خود **entity** صورت می گیرد . باید توجه داشت که در این پورت اجازه نداریم به عنوان فیدبک استفاده کنیم . از این مد برای کلیه سیگنالها خروجی نظیر خروجی یک کانتر یا یک مقایسه کننده استفاده می گردد .

**Buffer** : این مد شبیه مد **out** ، خروجی بوده و مانند آن راه اندازی می شود فقط با این تفاوت که در این مد می توان از این پورت به عنوان فیدبک نیز در مدار استفاده کرد .

**Inout** : از این مد برای پورتهای دو جهته نظیر **data bus** استفاده می گردد . بنابراین راه اندازی این پورت هم می تواند از داخل **entity** و هم از خارج آن صورت گیرد . مد **inout** اجازه گرفتن **Feedback** در مدار را می دهد . این مد می تواند جایگزین دیگر مدهای **buffer** و **out** بشود چون تمامی خواص آنها را دارد .

**نوع ( type ) data :**

بعد از اینکه مد پورت مشخص گردید، باید نوع داده آن را مشخص کنیم. نوع های استاندارد که توسط استاندارد IEEE 1076/93 فراهم شده و قابل سنتز می باشند شامل Boolean , bit , bit\_vector , integer می باشد. در ضمن نوع دیگر که بصورت پکیج های IEEE Std\_Logic\_1164 می باشند نظیر Std\_Logic و Std\_Ulogic و بسیار قابل استفاده هم هستند قابل سنتز می باشند.

برای اینکه Compiler مورد نظر بتواند اسامی Standard Logic را سنتز کند باید نوع Library آن را مشخص کنیم. بنابراین مثال entity که برای add4 گفته شده قابل سنتز نبوده مگر اینکه به صورت زیر اصلاح یابد:

```
library ieee ;
use ieee.std_logic_1164.all ;
entity add4 is port(
    a,b : in std_logic_vector(3 downto 0);
    ci : in std_logic ;
    sum:out std_logic_vector(3 downto 0) ;
    co : out std_logic);
end add4 ;
```

**Architecture -3-3-1**

در قسمت Architecture توابع یا عملکرد مدار بیان می شود. VHDL اجازه می دهد که طراحی به شیوه های مختلفی صورت بگیرد. این شیوه ها به نامهای بیان رفتاری، جریان داده و ساختاری می باشند.

**بیان رفتاری ( behavioral ) :**

لیست 1-2 یک مثال از طراحی به بیان رفتاری را نشان می دهد که شبیه لیست 1-1 است. بیان رفتاری یک بیان سطح بالا نسبت به دیگر روش های طراحی می باشد.

وقتی طراحی به زبان سطح بالا باشد دیگر لزومی به تمرکز روی نحوه پیاده سازی در سطح گیت نمی باشد. بنابراین در این روش شما بیشتر به چگونگی

پیاده سازی ایده های خود توجه دارید نه به نحوه پیاده سازی مدار مورد طراحی .

```
library ieee;
use ieee.std_logic_1164.all ;
entity eqcomp4 is port(
    a,b : in std_logic_vector(3 downto 0);
    equals: out std_logic) ;
end eqcomp4 ;
architecture behavioral of eqcomp4 is
begin
    Comp : process (a,b)
begin
    if a=b then
        equals <='1' ;
    else
        equals <='0' ;
    end if ;
end process comp ;
end behavioral ;
```

## لیست 1-2

خطوط 1 و 2 برای شناسایی نوع های ( types ) که در قسمت entity ملاحظه می گردد لازم هستند . خطوط 3 تا 6 ، قسمت entity برای یک مقایسه کننده 4 بیتی را معرفی می کند . خطوط 7 تا 17 بدنه Architecture را شامل می شود که بر اساس یک الگوریتمی، رفتار مدار را بیان می کند . لیست Process یک ساختار طراحی VHDL است که برای شکل دهی به الگوریتم می باشد . لیست Process می تواند شامل یک نام باشد که این نام اختیاری است . ( در این مثال با نام Comp است . ) این نام با علامت : مجزا می گردد . لیست حساسیت که در جلو نام Process آمده است نشان دهنده این است که Process بر اساس این سیگنالها اجرا می گردد (یا حساس می باشد). که در این مثال سیگنالهای a,b لیست حساسیت Process را تشکیل می دهند . خطوط 12 تا 16 شامل لیست های پشت سر هم می باشند که عملیات مقایسه کننده چهار بیتی را مدل سازی کرده است و

بیان می کند زمانی خروجی فعال ( یک ) می شود که شرط  $a=b$  برقرار باشد . در `Process` می توان از انواع دستوراتی که در VHDL وجود دارند استفاده کرد. شما به کمک این دستورات می توانید نحوه ارتباط خروجیها با ورودیها را برقرار کنید. `Architecture` می تواند شامل چندین `Process` باشد که همگی موازی هم کار می کنند. قسمت `Architecture` مربوط به لیست 1-2 می تواند بصورت لیست زیر نوشته شود :

architecture behavioral of eqcomp4 is

```
begin
  comp :process (a,b)
  begin
    equals <='0'
    if a=b then
      equals <='1' ;
    end if ;
  end process comp ;
end behavioral ;
```

لیست 1-3

این `Process` بیان می کند که خروجی `equals` بصورت پیش فرضی دارای مقدار 0 است و اگر شرط  $a=b$  برقرار شود در اینصورت این خروجی 1 می شود .

### بیان جریان داده ( Data Flow ) :

لیست 1-1 بیان جریان داده است ؛ لیست 1-4 نیز بیانی از جریان داده با انتخاب نوع پورتها می باشد .

--eqcomp4 is a four bit equality comparator

```
library ieee ;
use ieee.std_logic_1164.all ;
entity eqcomp4 is port(
  a,b :in std_logic_vector(3 downto 0);
  equals :out std_logic );
end eqcomp4 ;
architecture dataflow of eqcomp4 is
begin
```

```

    equals <='1' when ( a=b ) else'0';--equals is active high
end dataflow ;

```

لیست 4-1 بیان ساختار جریان داده برای eqcomp4 .

این ساختار یک بیان جریان داده است برای اینکه بیان می کند که چگونه داده از یک سیگنال دیگر و از ورودی به خروجی بدون استفاده از دستورات سری ( Sequential Statements ) انتقال پیدا می کند . اولین اختلافی که دیده می شد این است که در این ساختار Process تعریف نشده است . در این مثال data flow ما از جملات شرطی ( When – else ) استفاده کرده ایم می توان برای پیاده سازی الگوریتم ، از جملات انتخابی ( with – Select – when ) نیز استفاده کرد .

لیست 5-1 نشان می دهد که چگونه با استفاده از لاجیکها می توان مدار طراحی کرد .

```

library ieee ;
use ieee.std_logic_1164.all ;

entity eqcomp4 is
    port(a,b :in std_logic_vector(3 downto 0);
    equals :out std_logic ) ;
end eqcomp4 ;

architecture bool of eqcomp4 is
begin
    equals <=  not(a(0) xor b(0))
    and not(a(1) xor b(1))
    and not(a(2) xor b(2))
    and not(a(3) xor b(3));
end bool ;

```

لیست 5-1: ساختار data Flow برای eqcomp با استفاده از معادلات بولین

همانطور که ملاحظه می کنید تفاوت بین لیست 2-1 و 4-1 با 5-1 این است که در لیست 2-1 عملیات بصورت ترتیبی ( Sequential ) صورت می گیرد ولی در لیست 4-1 عملیات بصورت همزمان ( Canurency ) انجام می گیرد .

**بیان ساختاری ( Structural ) :**

لیست 6-1 را بخوانید و مشاهده کنید که چگونه یک بیان ساختاری تشکیل می شود .

```
library ieee
use ieee.std_logic_1164.all
entity eqcomp4 is
    port(a,b :in std_logic_vector(3 downto 0);
    equals :out std_logic);
end eqcomp4;
use work.gatespkg.all;
architecture struct of eqcomp4 is
    signal x : std_logic_vector(0 to 3);
begin
    u0 : xnor2 port map (a(0),b(0),x(0));
    u1 : xnor2 port map (a(1),b(1),x(1));
    u2 : xnor2 port map (a(2),b(2),x(2));
    u3 : xnor2 port map (a(3),b(3),x(3));
    u4 : and4 port map (x(0),x(1),x(2),x(3),equals);
end struct;
```

#### لیست 6-1 : بیان ساختاری eqcomp4

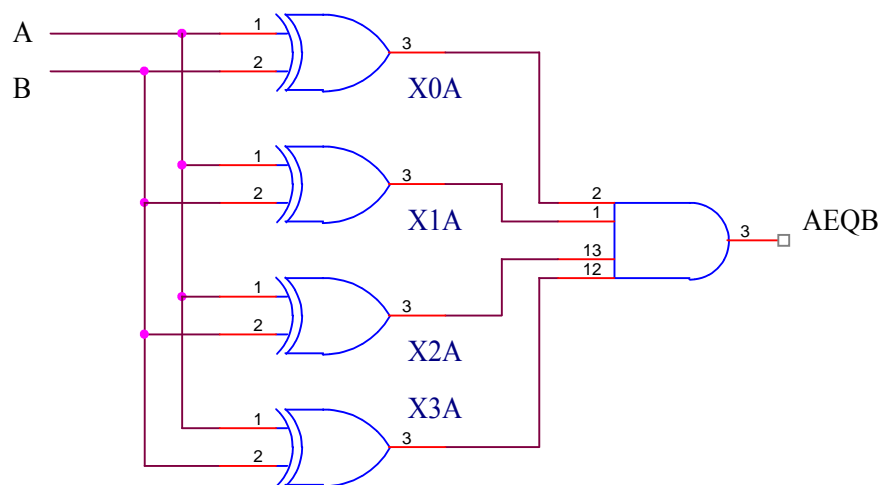
این طراحی نیازمند این است که Component های xnor2 و and4 بصورت بسته هایی (Package) تعریف شده باشند و در Library موجود و Compile شده باشند. که در این مثال این Component از پکیج gatespkg از Work library فراخوانی شده است. (مطالب مربوط به Package و Library در فصلهای بعدی گفته خواهد شد .)

بیان ساختاری شامل netlist از VHDL می شود. این netlist خیلی متناظر با netlist شماتیک هستند. component های تعریف شده با سیگنالها به همدیگر اتصال پیدا کرده اند .

طراحی ساختاری بصورت سلسله مراتبی است. در این مثال طراحی eqcomp4 با استفاده از component های xnor2 , and 4 صورت گرفته است.

شکل 4-1 حالت سلسله مراتبی را نشان می دهد .





شکل 4-1

يك مدار مقایسه كننده چهار بیتی معمولاً به روش ساختاری بیان نمی گردد و چنین مداری با این روش ابتدایی است. در طراحیهای بزرگ و پیچیده روش ساختاری يك روش مناسب می باشد. بدین صورت كه مدار بزرگ و پیچیده به بلوكهای كوچكتر تبدیل شده و هر بلوك به روش behavioral یا data flow طراحی می گردد و هر بلوك به عنوان يك Component در نظر گرفته می شود و سپس به روش ساختاری این بلوكها را به هم ارتباط داده و در نهایت تمامی مدار طراحی می گردد .

#### 4-1- مشخصه ها ، Data Object , Data type و نسبتها :

در این قسمت ما جزئیات مربوط به مشخصه ها ، Data type ، Data object و نسبتها را با استفاده از مثالهایی جهت درك بهتر مفاهیم اساسی، بیان می کنیم .

##### 1-4-1- مشخصه ها :

مشخصه های اساسی از حروف الفبایی ، عددی و پارامترهای زیر خط دار ساخته می شوند قوانین زیر در مورد مشخصه ها وجود دارد :

- \_ اولین کاراکتر باید يك حرف باشد .
- \_ آخرین کاراکتر نباید يك زیر خط باشد .
- \_ دو زیر خط نمی تواند کنار هم قرار بگیرد .

زبان VHDL دارای یک سری کلمات رزرو می باشد که نمی توان از آنها در مشخصه ها استفاده کرد. در ضمن حروف کوچک و بزرگ در مشخصه ها معادل هم هستند.

#### 1-4-2 Data Object

**Data Object** مقادیری از نوع مشخص شده خودشان، در خود نگه می دارند. آنها شامل چهار دسته مختلف هستند :

ثابتها ، سیگنالها ، متغیرها و فایلها که قبل از استفاده باید آنها را معرفی کرد .

##### ثابتها :

ثابتها مقداری را در خود ذخیره می کنند که در طول برنامه دیگر نمی توان آنها را تغییر داد. این مقادیر معمولاً در ابتدای توضیح برنامه مشخص می شوند. ثابتها می توانند در توضیحات , Architecture,entity Package یا Process معرفی شوند . مثال زیر نمونه ای از مقدار دهی ثابت ها می باشد .

Constant width : integer : = 8 ;

##### سیگنالها :

توسط سیگنالها می توان خطوط ارتباطی یا اتصالات بین Component ها را مشخص کرد . سیگنالهای نیز می توانند ورودی یا خروجی گیتهای لاجیکی باشند. مثال زیر نمونه ای از معرفی یک سیگنال می باشد :

Signal count : bit – vector ( 3 downto 0 ) ;

سیگنالها می توانند دارای مقدار اولیه نیز باشند :

Signal Conut : bit- vector ( 3 downto 0 ) : = "0101"

در طول برنامه زمانیکه بخواهیم به سیگنال مقدار بدهیم از نماد <= استفاده می شود در ضمن سیگنال خاصیت خواندنی و نوشتنی دارد .

#### متغیرها : Variables

از متغیرها در Process و زیر برنامه ها می توان استفاده کرد . و در محدوده Process یا زیر برنامه باید معرفی شوند از یک متغیر در دو

Process نمی توان استفاده کرد. متغیرها برخلاف سیگنالها دارای سیم و اجزاء حافظه ای نیستند و بنابراین تاخیر ندارند و معمولاً برای اهداف محاسباتی استفاده می گردند. در هنگام معرفی متغیر، می توان به آن مقدار اولیه نیز داده شود بطور مثال :

```
variable result : std_logic: '0';
```

متغیرها فقط مقداری را در یک زمان نگه می دارند و نمی توان شکل موج خروجی آنرا مشاهده کرد. در طول برنامه برای مقدار دهی به متغیر از نماد := استفاده می شود.

### فایلها ( Files ) :

فایلها مقادیری از یک نوع مشخص را معین می کنند. از فایلها معمولاً در برنامه های تست استفاده می گردد.

### Aliases :

از یک Alias برای شناسایی یک Object خارجی استفاده می گردد و آن بعنوان یک Object جدید نمی باشد. یک Alias معادل Object اصلی آن می باشد. و معمولاً بعنوان یک روش مناسب برای مشخص کردن رنگی از یک Array type استفاده می شود. برای مثال ، در مشخص کردن Field هایی در یک آدرس :

```
signal address : std_logic_vector(31 downto 0);
alias tp_ad : std_logic_vector(3 downto 0) is address (31 downto 28);
alias bank : std_logic_vector(3 downto 0) is address (27 downto 24);
alias row_ad : std_logic_vector(11 downto 0) is address (23 downto 12);
```

### 1-4-3- نوع داده ها Data Types :

#### Scalar type :

این نوع اجازه می دهد که اپراتورهای نسبی بتوانند با هم کار بکنند که کلاً دارای سه دسته هستند. enumeration و , flouting , Ploysical integer .

#### Enumeration type :

یک Enumeration type یک لیستی از مقادیر است که Object ی از همان نوع را می تواند در خود نگه دارد . Enumeration type اغلب برای State Machine تعریف می گردد :

type states is ( idle, preamble, data, nosfd, error);

يك سيگنال مي تواند از نوع enumeration type مشخص شده ،تعريف گردد .

Signal Current\_State : States :

Enumeration type که توسط استاندارد IEEE 1076 براي استفاده در سنتز معرفي شده است شامل Boolean و Bit است که بصورت زیر تعريف مي شوند :

Type boolean is (FALSE , TRUE);

Type bit is ( '0' , '1' ) ;

استاندارد IEEE 1164 يك type اضافي بنام Std – ulogic معرفي کرده است که داراي Sub type هاي مختلف است و براي هر دو مورد سنتز و شبیه سازي بکار مي رود . نوع Std\_ulogic 9 حالت مختلف دارد که بصورت زیراند :

type std\_ulogic is ( 'u' ,--uninitialized

'x' ,--forcing unknown

'0' ,--forcing 0

'1' ,-- forcing 1

'Z' ,--high impedance

'W' ,--weak unknown

'L' ,-- " 0'

'H' ,-- " '1'

'\_ ' ,-- Do not care );

نوع Std\_Logic نیز داراي مقادير یکسان با Std – ulogic مي باشد . استاندارد IEEE 1164 داراي مجموعه Std\_Logic و Std\_ulogic و std – ulogic\_vector مي باشد .

: integer type

اپراتورهای نسبی و integer توسط VHDL تعریف شده است. ابزارهای نرم افزاری که VHDL را پردازش می کنند باید Integer را از رنج  $2,147,483,687 - (2^{31}-1)$  تا  $2,147,483,687 (2^{31}-1)$  پشتیبانی کنند .  
یک سیگنال یا متغیر که از نوع integer است باید با یک رنجی مشخص گردد . برای مثال :

Variable a :integer range - 255 to 255;

### : Floating type

مقادیری که از نوع Floating\_Point بکار می رود برای اعداد تقریبی استفاده می شود. شبیه integer انواع Flouting می توانند محدود شوند .  
نوع Floating اغلب توسط ابزارهای سنتز پشتیبانی نمی شوند برای اینکه برای پیاده سازی عملکردهای محاسباتی با آنها نیاز دارد که به مقادیر عددی دسترسی پیدا کند .

### : Physical types

مقادیر از نوع Physical بصورت واحدهای اندازه گیری استفاده می شود. تنها نوع فیزیکی که تعریف شده زمان است :

Type time is rang - 2147483647 to 2147483647

Unit

Fs;

Ps = 1000fs ;

ns = 1000ps ;

us = 1000ns ;

ms = 1000us ;

Sec = 1000ms ;

min = 60sec ;

Hr = 60min ;

End units ;

از نوع فیزیکی برای ایجاد برنامه های تست یا test benche استفاده می کنیم. برای دیگر واحدهای اندازه گیری پایه نظیر متر ، گرم ، فوت و ... باید خودتان تعریف کنید .

### : Composite type

Data object از نوع Scalar در طول زمان شبیه سازی فقط يك مقدار مي توانند پيدا کنند ولي Composite Object كلا دو دسته هستند: record type , array type .

**Array type** : يك Object از نوع array شامل چندین جزء با نوع یکسان است. اغلب array type که استفاده می شوند توسط استاندارد , 1164 1079 استفاده شده است :

Type bit\_vector is array (natural range <>) of bit;

Type std\_ulogic\_vector is array (natural range <>) of std\_ulogic;

Type std\_logic\_vector is array (natural range <>) of std\_logic;

این نوع ها بصورت array بدون محدوده تعریف شده اند: تعداد بیت Std\_ulogic یا Std\_logic در آنها مشخص شده اند. array ها فقط توسط natural محدود شده اند. از این نوع اغلب برای باسها استفاده می گردد. برای مثال :

Signal a: std\_logic\_vector (3 downto 0);

يك باس همچنین مي تواند با Type مشخص شده تعريف شود .

Type word is array (16 downto 0) of bit ;

Signal b:word;

از array هاي دو بعدي براي ايجاد کردن يك جدول دو بعدي استفاده مي گردد .

Type table 8\*4 is array (0 to 7 , 0 to 3 ) of bit ;

Constant exclusive\_or : table 8\*4 : = (

"000\_0"

"001\_1"

"010\_1"

"011\_0"

"100\_1"

"101\_0"

"110\_0"

"111\_1"

**record type** : يك Object از نوع record شامل چندین جزء با نوعهاي مختلف مي باشد .

field های اختصاصی از یک record می توانند توسط نام اجزاء، مرجع قرار بگیرد. برنامه زیر یک record\_type را نشان می دهد که برای iocell تعریف شده است، object بصورت آن نوع، تعریف و مقادیری به آنها تعلق گرفته است:

Type iocell is record .

Buffer\_inp : bit\_vector ( 7 downto 0 )

Enable : bit ;

Buffer\_out : bit\_vector ( 7 downto 0 )

End record ;

Signal bus a , bus b , bus c : iocell ;

Signal rec : bit\_vector ( 7 downto 0 );

Bus a , buffer\_inp <= vec ;

Bus b , buffer\_inp <= bus a , buffer\_inp ;

Bus b , enable <= '1' ;

Bus c <= bus b ;

#### error های عمومی -5-1

کدی که در لیست 1-7 آمده شامل چندین error است. ببیند آیا می توانید آنها را تشخیص دهید :

```

Library ieee ;                                --line 1
Use ieee.Std_logic_1164.all ;                  --line 2
Use work.std_arith.all                          --line 3
Entity terminal_count is port(                  --line 4
    clock , reset, enable    : in bit ;        --line 5
    data                      : in std_logic_vector ( 7 downto 0 ); --line 6
    equals , term_cnt        : out std_logic);   --line 7
end terminal_count ;                            --line 8
architecture terminal_count of terminal_count is --line 9
    signal count : std_logic_vector ( 7 downto 0 ); --line 10
begin                                           --line 11
    compare = process                          --line 12
    begin                                       --line 13
        if data = count then                  --line 14

```

```

        equals = '1';                                --line 15
    End if ;                                          --line 16
End process ;                                       --line 17
                                                    --line 18
counter = process (clk)                             --line 19
begin                                              --line 20
    if reset = '1' then                             --line 21
        count <= "11111111"                         --line 22
    elsif rising _ edge ( clock ) then              --line 23
        count <= count + 1 ;                         --line 24
    End if ;                                         --line 25
End process ;                                       --line 26
Term_cnt <= 'z' when enable = '0' else              --line 27
        '1' when count = "1-----" Else            --line 28
        '0';                                         --line 29
end terminal_count ;                               --line 30

```

#### لیست 1-7: برنامه ای با داشتن چندین غلط

اولین error در خط 11 است. Process به یک لیست حساسیت نیاز دارد که باید شامل data و count باشد .

error بعدی در خط 14 است، اپراتور باید بصورت <= نوشته شود.

error بعدی یک error مفهومی است فقدان یک else برای مقدار دهی equals دیده می شود که بایستی بصورت زیر نوشته می شد :

```

if data = count then
    equals = '1';
else
    equals = '0';
end if;

```

این قسمت از برنامه با استفاده از دستور when – else بصورت زیر می توان نوشت :

```

equals <= '1' when data = count else '0';

```



error بعدی در خط 18 و در لیست حساسیت پروس counter می باشد  
اولاً باید clk به clock تغییر یابد و ثانیاً علاوه بر clock سیگنال reset  
نیز در لیست حساسیت قرار بگیرد. خط 21 نیز شامل error است : در array  
یک عدد 1 اضافی است .

error بعدی در خط 22 است. در استفاده تابع rising\_edge سیگنال  
باید از نوع std\_logic باشد. بنابراین یا باید نوع سیگنال clock را  
به std\_logic تغییر داد یا اینکه از تابع استفاده شود  
در خط 23 نیز یک error وجود دارد. اپراتور + برای نوع std\_logic\_vector  
یا integer تعریف نشده است بنابراین ما باید پکیج std\_arith را به  
برنامه اضافه کنیم. خط 27 شامل error بعدی است. حرف 2 تنها بایستی  
بصورت بزرگ نوشته شود نه کوچک .

z به object ی تعلق می گیرد که از نوع std\_logic باشد. آخرین error در خط  
28 قرار گرفته است. که تساوی count با "11111111" غلط است برای  
نوشتن صورت باید تابع std\_match استفاده شده باشد. در این حالت  
برنامه بصورت زیر نوشته شود :

```
term_cnt <= 'z' when enable = '0' else
    '1' when count > "10000000" else -- or count > 127
    '0';
```

کد صحیح در لیست 1-8 قرار گرفته است .

```
Library ieee ;
Use ieee.Std_logic_1164.all ;
Use work.std_arith.all
Entity terminal_count is port (
    clock , reset, enable    : in std_logic ;
    data                      : in std_logic_vector ( 7 downto 0 ) ;
    equals , term_cnt        : out std_logic);
end terminal_count ;
architecture terminal_count of terminal_count is
    signal count : std_logic_vector ( 7 downto 0 );
begin
    compare = process ( clock,reset )
    begin
        if data = count then
```

```
        equals = '1' ;
    else
        equals = '0' ;
    end if ;
end process ;

counter = process (clk)
begin
    if reset = '1' then
        count <= "11111111"
    elsif rising _ edge ( clolk ) then
        count <= count + 1 ;
    End if ;
End process ;
    term_cnt <= 'z' when enable = '0' else
        '1' when std_match ( count, "1-----" ) else
        '0'
;
end terminal_count ;

لیست 8-1 : کد VHDL تصحیح یافته لیست 7-1
```

# فصل دوم

## ایجاد مدارات ترکیبی

### و سنکرون

#### 2-1- مقدمه

در قسمت قبل توضیحات مربوط به entity و بدنه Architectar داده شد که entity شامل لیستی از پورتهای مشخص می باشد. پورها هر کدام دارای

يك type و mode مشخص بودند. و همچنین دیدیم که بدنه Architectar ممکن است بصورت يك یا ترکیبی از ساختارهای structural, dataflow, behavioral باشد. در این قسمت، چگونگی تعریف مدارات ترکیبی مدارات ترکیبی و سنکرون که در ساختارهای مختلف استفاده می شود بیان می شود و با مثالهای ساده برنامه ها به چندین روش نوشته شده اند.

## 2-2- مدارات ترکیبی

مدارات ترکیبی به روشهای مختلفی می توانند طراحی شوند. دستورهای همزمان در بیان dataflow و structural استفاده می شوند و دستورهای پشت سرهم در بیان behavioral استفاده می شود.

### 2-2-1- استفاده از دستورهای همزمان

دستورهای همزمان خارج از يك Process قرار گرفته و بصورت همزمان اجرا می گردند.

در بیان data flow سه نوع دستور همزمان وجود دارد: دستورهای همزمان با جبر بولین، دستورهای انتخابی با دستور With – seled – when و دستور شرطی when – else.

### معادلات بولین:

معادلات بولین در هر نوع دستور همزمان یا پشت سرهم می تواند استفاده شود. در این قسمت استفاده معادلات بولین را در دستور همزمان جهت استفاده در مدارات ترکیبی بیان می کنیم. لیست زیر تعریف يك مالتی پلکسر 4 به 1 که باسهایی 4 بیتی را مالتی پلکس می کند می باشد.

Library ieee;

Use ieee std\_logic\_1164.all;

Entity mux is port (

A,b,c,d : in std\_logic\_vector ( 3 downto 0 );

S : in std\_logic\_vector ( 1 downto 0 );

X : in std\_logic\_vector ( 3 downto 0 );

End mux;

Architecture archmux of mux is

Begin

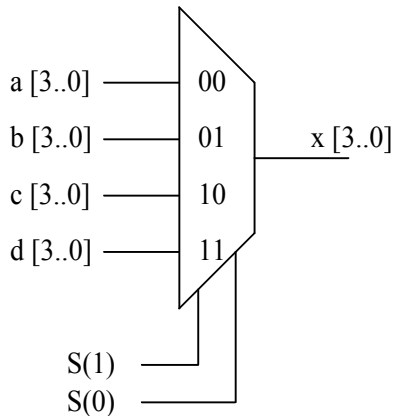
X(3) <= (a(3) and not (s(1) and not (s(0))))

```

Or (b(3) and not (s(1) and (s(0)))
Or ( c (3) and s(1) and not (s(0)))
Or ( d (3) and s(1) and s(0));
X(2) <= (a(2) and not (s(1) and not (s(0)))
Or (b(2) and not (s(1) and (s(0)))
Or ( c (2) and s(1) and not (s(0)))
Or ( d (2) and s(1) and s(0));
X(1) <= (a(1) and not (s(1) and not (s(0)))
Or (b(1) and not (s(1) and (s(0)))
Or ( c (1) and s(1) and not (s(0)))
Or ( d (1) and s(1) and s(0));
X(0) <= (a(0) and not (s(1) and not (s(0)))
Or (b(0) and not (s(1) and (s(0)))
Or ( c (0) and s(1) and not (s(0)))
Or ( d (0) and s(1) and s(0));
End archmux ;

```

لیست 2-1 : یک مالتی پلکسر 4 بیتی .



شکل 2-1 : بلوک دیاگرام مالتی پلکسر

لیست 2-1 یک تعریف طاقت فرسا ( سخت ) و زبان سطح پایین است که عموماً در گذشته برای PAL استفاده می شد بکار می رفت. برای اینچنین توابعی با معادلات جبر بولین بسیار ساده تر تعریف می شوند و VHDL از این توانایی برخوردار است. برای مثال، در لیست

2-2 چندین سیگنال با استفاده از جبر بولین مقداردهی شده اند. در حالتی شبیه به این استفاده از دستور `if_then_else` طاقت فرسا می باشد. ما مثالهای زیادی که بر معادلات بولین مناسب هستند در ادامه جزوه خواهیم آورد .

```
Entity my_dedign is port (
    Men_op , io_op : in bit ;
    Reud , write    : in bit ;
    Memr , memw     : out bit ;
    Io_rd , io_wr    : out bit ) ;
End my_design ;
Architecture control of my_design is
Begin
    Memw <= mem-op    and write ;
    Memr <= mem-op    and read  ;
    Io-wr <= io -op    and write ;
    Io-rd <= io -op    and read  ;
End control ;
```

لیست 2-2: تعریف سیگنالها با معادلات بولین

### عملکرد ( اپراتورهای ) لاجیکی :

اپراتورهای لاجیکی اساس معادلات بولین هستند. اپراتورهای لاجیکی `array` یا `boolean` یا `bit` های برای `not`, `xnor`, `xor`, `nand`, `or`, `and` های تک بعدی از `bit` و بولین از پیش تعریف شده اند. در استفاده از این اپراتورها ( بجز `not` ) با `array` هایی از بیت یا بولین ، دو عملوند باید پهنای بیت مساوی داشته باشند. استاندارد IEEE 1164 این اپراتورها را برای `type` های `std_logic`, `std_ulogic`, `array` های یک بعدی از آنها فراهم کرده است .

اپراتورهای لاجیکی دارای اولویت هستند . برای مثال با جبر بولین، شما جمله  $A+B.C$  را انتظار دارید که بصورت  $A+(B.C)$  ارزیابی شود. بهر حال در VHDL ، اپراتورهای لاجیکی وجود ندارد که دارای اولویت بیشتری باشد. برای مثال در مجزا سازی عملیات پرانتزها

موردنیاز هستند . جمله  $(A+B) \cdot C$  در حقیقت بصورت زیر است :  $x \leq A \text{ or } B \text{ and } C$  ;

**with – Select - when :**

دستور with – select – when سیگنال انتخاب شده برای مقدار دهی را مشخص می کند .  
در ترکیب :

With selection\_signal select

```
Signal_name <= value _ a when value_1_of_selection_signal,
               value _ b when value_2_of_selection_signal,
               value _ c when value_3_of_selection_signal,...
               value _ x when last_value_of_selection_signal;
```

signal\_name با توجه به مقدار selection\_signal مقدار دهی می شود. تمامی مقادیر Selection\_signal باید در شرط های when وجود داشته و متقابلاً انحصاری باشند . در لیست 2-3 با استفاده از این دستور مالتی پلکسر مربوط به شکل 2-1 را تعریف می کنیم .

library ieee ;

use ieee . std\_logic\_1164 . all ;

entity mux is port (

A,b,c,d : in std\_logic\_vector ( 3 downto 0 ) ;

S : in std\_logic\_vector ( 1 downto 0 ) ;

X : in std\_logic\_vector ( 3 downto 0 ) ;

End mux ;

Architecture archmux of mux is

Begin

With s select

X <= a when "00"

b when "01"

c when "10"

d when others ;

end archmux ;

لیست 2-3- مقدار دهی سیگنال بصورت انتخابی

با توجه به مقدار  $s$ ، سیگنال  $x$  یکی از چهار مقدار  $a, b, c$  یا  $d$  را پیدا می کند. `others` بجای 11 برای موارد زیر بکار می رود :

$s$  بصورت `type` های از `std_logic_vector` باشد و 9 مقدار ممکن برای یک `data object` از نوع `Std_Logic` وجود دارد .

برای سخت افزار و ابزارهای سنتز، 11 فقط مقدار با معنی دارد اما `code` باید برای VHDL امکان پذیر باشد. شما می توانید مقدار 11 را صریحاً بصورت یکی از مقادیر  $s$  بکار ببرید ولی بهر حال حالت `Others` برای مشخص کردن تمامی حالات  $s$  لازم است :

Architecture archmux of mux is

Begin

With  $s$  select

$X \leq a$  when "00"

$b$  when "01"

$c$  when "10"

$d$  when others ;

end archmux ;

مقدار `metallogical` همچنین می تواند جهت مشخص کردن مقدار  $x$

`don't_care` استفاده شود بصورت :

Architecture archmux of mux is

Begin

With  $s$  select

$X \leq a$  when "00"

$b$  when "01"

$c$  when "10"

$d$  when "11",

"\_" when others ;

end archmux ;

سنتز، نتایج برای هر کدام از سه ورژن Architecture یکسان است،

برای اینکه سخت افزار مشخص میکند که اگر برای  $S$  مقدار لاجیکی

شناخته نشود  $x$  نیز تعریف نمی شود.

: when – else



ساختار when – else دستوری است که برای مقدار دهی سیگنالها بصورت شرطی بکار می رود به این معنی که سیگنال به یک مقدار تعلق می گیرد، در صورتیکه شرط آن برقرار باشد. ساختار کلی بصورت زیر است :

```
signal_name <= value_a when condition 1 else
                value_b when condition 2 else
                value_c when condition 3 else ...
                value_x ;
```

signal\_name به مقداری که در معادلات شرطها وجود دارد تعلق می یابد. اولین شرطی که برقرار باشد مقدار آن به signal\_name تعلق می گیرد. در لیست 2-4 همان مالتی پلکسر چهار به یک با دستور when\_else نوشته شده است .

```
library ieee ;
use ieee . std_logic_1164 . all ;
entity mux is port (
    A,b,c,d : in std_logic_vector ( 3 downto 0 ) ;
    S      : in std_logic_vector ( 1 downto 0 ) ;
    X      : out std_logic_vector ( 3 downto 0 ) ;
End mux ;
Architecture archmux of mux is
Begin
    With s select
        X <= a when ( S= "00" ) else
            b when ( S= "01" ) else
            c when ( S= "10" ) else
            d ;
end archmux ;
```

لیست 2-4 : مقدار دهی سیگنال بصورت شرطی

در برنامه فوق همه شرطها در Statement و when – else بصورت انحصاری لیست شده اند.

اگر در when\_else شرطها بصورت انحصاری نباشد بالاترین اولویت با اولین شرط صحیح است. اگر مقادیر سیگنالها بصورت انحصاری استفاده شود ، ساختار when\_else اندکی از ساختار with\_select\_when طولانی تر می شود .

( شبیه مالتی پلکسر ) اما این ساختار میتواند به شما در مختصر کردن بیان الگوهای اولویت کمک کند شبیه مثال که در لیست 2-5 آمده است :

```
library ieee ;
use ieee . std_logic_1164 . all ;
entity priority is port (
    A,b,c,d,w,x,y,z : in std_logic ;
    J : out std_logic);
```

End priority ;

Architecture priority of priority is

Begin

```
J <=  w when a = '1' else
      x when b = '1' else
      y when c = '1' else
      z when d = '1' else
      '0' ;
```

end priority ;

لیست 2-5 : انکودر اولویت دار با مقداردهی سیگنال بصورت شرطی

اگر a, b, c و d متناظراً منحصر بفرد هستند اگر در یک زمان فقط

توسط یکی شناخته شوند. کدهایی که در لیست 2-6 آمده اختصاصی است :

```
library ieee ;
use ieee . std_logic_1164 . all ;
entity no_priority is port (
    a,b,c,d,w,x,y,z : in std-logic ;
    J : out std_logic );
```

End no\_priority ;

Architecture no\_priority of no\_priority is

Begin

```
J <=  ( a and w ) or ( b and x ) or ( c and y ) or ( d and z ) ;
```

end no\_priority ;

لیست 2-6 : انتخاب بدون اولویت با استفاده از معادلات بولین .

اگر سیگنالهای  $a, b, c$  و  $d$  انحصاری شناخته شوند حاصل لیست 2-6  
معادل توابعی با ورودیهای کمتر است. حاصل کدهایی که در لیست 2-7  
آمده است با لیست قبل تفاوت دارد.

```
library ieee ;
use ieee . std_logic_1164 .all ;
entity compares is port (
    a,b,c,d,w,x,y,z : in std_logic ;
    J : out std_logic ) ;
End compares ;
Architecture compares of compares is
    Signal tmp : std_logic_vector ( 3 downto 0 ) ;
Begin
    Tmp <= ( a,b,c,d ) ;
With tmp select
    J <= w when "1000" ,
        x when "0100" ,
        y when "0010" ,
        z when "0001" ,
        '0' when others ;
end compares ;
```

لیست 2-7 : انتخاب با استفاده از ترکیبی مقادیر  $a, b, c, d$

### شروط به زبان ساده :

شرط ها در جملات `when_else` می تواند بصورت زبان ساده باشند اگر  
بصورت کدی که در زیر آمده است نوشته شوند. این نوع شرط با جملات  
`with_select_when` معمولاً امکان پذیر نیست برای اینکه شرطها باید  
مقادیری از سیگنال انتخاب باشند.

```
Signal stream , instrm , oldstrm : std_logic_vector ( 3 downto 0 ) ;
Signal state : states ;
Signal we : std_logic ;
Signal id : std_logic_vector ( 15 downto 0 ) ;
-----
stream <= "0000" when ( state = idle and sturt = '0' ) else
    "0001" when ( state = idle and sturt = '1' ) else
```

```

instrm when ( state = incoming ) else
oldstrm ;
we <='1' when ( state = wirte and id < x "1FFF" ) else '0' ;

```

### اپراتورهای نسبی :

اپراتورهای نسبی برای تست کردن برابری ، نابرابری و آرایشی از آنها استفاده می شوند . اپراتورهای برابری و نابرابری ( = و /= ) برای تمامی type ها معرفی می شوند . اپراتورهای دامنه دار ( <, <=, >, >= ) برای type های scalar یا array با یک رنج پیوسته تعریف می شوند . array هایی که هم ارز و دارای طول برابر هستند می توان استفاده شوند . نتیجه اپراتورهای نسبی، بولین است ( جواب آن یا درست یا نادرست ) .

operand type ها در یک اپراتور نسبی باید مثل هم باشند . جمله زیر غلط است برای اینکه یک Std\_logic\_vector است و 123 یک integer است :

```

signal a: std_logic_vector ( 7 downto 0 )
if a = 123 then ...

```

### اپراتورهای بارگذاری :

اپراتورهای نظیر اپراتورهای نسبی ممکن است بارگذاری باشند . اپراتورهای بارگذاری اجازه استفاده از اپراتورهای با چندین type را می دهند . برای هر اپراتور توسط استاندارد IEEE 1076 تعریف نشده است . اپراتور ممکن است با استفاده از تعریف function بارگذاری باشند ، اما تعدادی اپراتور بارگذاری در استاندارد IEEE 1164 و 1076.3 تعریف شده است . برای مثال استاندارد IEEE 1076.3 تابعی برای اپراتور

```

Library ieee ;
Use ieee.Std_logic_1164.all ;
Use work.numeric_std.all ;
Entity compare is port (
    A = in unsigned ( 3 downto 0 ) ;
    X = out std_logic ;
End add_vec ;
Architecture compare of compare is

```

Begin

X <= '1' when a = 123 else '0' ;

End ;

لیست 2-8 : اپراتور بارگذاری = تعریف شده در numeric\_std

توابع اپراتور بارگذاری در numeric\_std package از استاندارد 1076.3 تعریف شده اند و این Package باید در entity طراحی با یک ماده ای ( قضیه ای ) از آن فعال شود . package هایی نیز برای اپراتورهای بارگذاری برای std\_logic اضافه شده است . Std\_arith package ساخته شده، برای چندین تابع جدید و اپراتورهای ریاضی بدون علامت در راه اندازی std\_logic\_vector تعریف شده است. مثال زیر نمونه ای از این مورد است .

Library ieee ;

Use ieee. Std\_logic\_1164.all ;

Use work . std\_arith.all ; \_\_std\_arith , rather than numeric\_std

Entity compare is port (

A = in std\_logic\_vector ( 3 downto 0 ) ; \_\_ type std\_logic\_vector

X = out std\_logic ;

End add\_vec ;

Architecture compare of compare is

Begin

X <= '1' when a = 123 else '0' ;

End ;

لیست 2-9 : اپراتور بارگذاری = تعریف شده در std\_arith

برای مثالهایی از این نوع ، ما numeric\_std package در بیان بکارگیری type ، توابع و اپراتورها استفاده می کنیم ما همچنین std\_arith package را در مثالهای دیگر در بیان چگونگی استفاده از یک array type , std\_logic\_rector که می تواند overhead را کاهش دهد استفاده می کنیم .

Package های numeric\_std و std\_arith همچنین شامل تابع مهم دیگری بنام std\_match می باشند.

این تابع در دو مقدار از نوع std\_logic بصورت یک don't care یا  
 card - with استفاده می شود. مقدار آن می تواند بصورت '0' یا '1'  
 ارزیابی شود. این تابع استاندارد میتواند مفید باشد برای اینکه  
 مقایسه ارزیابی زیر برای همه مقادیر بحر "1-1" است و در سخت افزار  
 هرگز درست (true) نیست.

If a = "1\_\_1" then \_\_\_ \_\_always evaluates false in synthesis

ابزارهای سنتز و شبیه سازی باید بصورت یک مقدار ترجمه کنند.  
 بنابراین مقایسه فقط زمانی درست است که یک باشد.  
 گرچه ارزیابی don't care، اپراتور = نمی تواند در شناسایی  
 حالتی don't care استفاده شود.

با استفاده از تابع std\_match مقایسه "1-1" از 3 std\_logic\_vector  
 (0 downto 3) زمان درست است که بیتیهای اول و آخر یک باشند.

If std\_mutch ( a, "1\_\_1" )then ... \_\_ true if a (3) = a (0) = '1'

### شناسایی component ها :

شناسایی component ها در جملات، همزمان می باشد که اتصالات  
 سیگنالها در یک طراحی را مشخص می کند. نوشتن یک برنامه مقایسه کننده  
 4 بیتی لیست 2-10 غیر معقولانه است ولی برای بیان شناسایی Statement که  
 می تواند در پیاده سازی لاجیکها ترکیبی بکار رود، کدها به این صورت  
 نوشته شده اند.

```
library ieee ;
use work . std_logic_1164. all ;
entity compare is port (
    a ,b : in std_logic_vector ( 3 downto 0 ) ;
    aegb : out std_logic ) ;
end compare ;
architecture archcompare of compare is
    signal c : std_logic_vector ( 3 downto 0 ) ;
begin
    X0 : xor2 port map ( a (0) ,b (0) ,c (0) ) ;
    X1 : xor2 port map ( a (1) ,b (1) ,c (1) ) ;
    X2 : xor2 port map ( a (2) ,b (2) ,c (2) ) ;
```

```
X3 : xor2 port map ( a (3) ,b (3) ,c (3) ) ;
N1 : nor4 port map ( c (0) ,c (1) ,c (2) ,c (3) , a eqb ) ;
End ;
```

لیست 2-10 : بیان یک مقایسه کننده 4 بیتی به روش structural

data Component ها توسط استاندارد VHDL تعریف شده اند این طراحی نیاز دارد که گیت‌های nor2, xor2 در دیگر Package تعریف شود .

## 2-2-2- با استفاده از جملات ترتیبی:

جملات ترتیبی در process ها ، تابع ها و procedur بکار می روند . توابع و Procedure در فصل بعدی معرفی می شوند. در این قسمت جملات ترتیبی در process که چگونه در طراحی مدارات ترکیبی استفاده می شوند را بیان می کنیم .

### If\_then\_else

ساختار if\_then\_else برای انتخاب یا مقدار دهی جملات می باشد، و اجرای آن بر اساس ارزیابی بولین ( درست یا غلط ) از یک شرط است. در ساختار زیر :

```
if ( condition ) then
```

```
do something ;
```

```
else
```

```
do something different ;
```

```
end if ;
```

اگر شرط بصورت true ارزیابی شود بعد از اجرا جمله do something انتخاب می گردد و اگر بصورت false ارزیابی گردد بعد از else جمله something different انتخاب می گردد ساختار با end if بسته می شود . Process هایی که در زیر آمده یک کار را انجام می دهند .

```
Signal step : std_logic ;
```

```
Signal addr : std_logic_vector ( 7 downto 0 ) ;
```

—

—

—

```
similar 1 = process ( addr )
begin
    step <= '0' ;
    if addr > x "0F" then
        step <= '1' ;
    end if ;
end process ;
```

```
similar 2 = process ( addr )
begin
    if addr > x "0F" then
        step <= '1' ;
    else
        step <= '0' ;
    end if ;
end process similar 2 ;
```

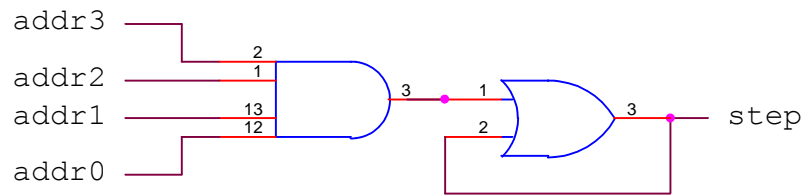
در هر دو process اگر که addr بزرگتر از hex of باشد step مقدار '1' و اگر کمتر یا مساوی باشد مقدار '0' را پیدا میکند. پروسس زیر لاجیک مساوی قبلی را بیان نمی کند برای اینکه غیر از این شرط هیچ مقدار پیش فرضی به step تعلق نمی یابد .

```
not_similar = process ( addr )
begin
    if addr > x "0F" then
        step <= '1' ;
    end if ;
end process ;
```

پروسس not\_similar دلالت بر این دارد که step باید زمانی که addr کمتر یا مساوی hex of است مقداری در خود نگه دارد. این بصورت یک خود نگه دارنده یا یک حافظه عمل می کند . بنابراین یک بار فعال شده و step به آنچه که فعال شده باقی می ماند بصورتی که در شکل 2-2 نشان داده شده و توسط معادله زیر تعریف شده است .

$$step = addr(3) * addr(2) * addr(1) * addr(0) + step$$





شکل 2-2

اگر شما نخواهید که به **step** مقدار بدهید مطمئناً دارای یک مقدار پیش فرض است یا می توانید که دستور **if-then** را با یک **else** تکمیل کنید . دستور **if-then-else** می تواند توسعه یابد و تعداد **elsif** های بیشتری جهت مقدار دهی و اولویت پذیری داشته باشد فرم کلی بصورت زیر است :

```

if (condition 1 ) then
    do something ;
elsif (condition 2 ) then
    do something different ;
else
    do something completely different ;
end if ;

```

برای هر سیگنال **x** به یک مقدار تعلق یافته بر اساس شرط بوده است ، حاصل سنتز یک معادله است برای مثال کد زیر :

```

if (condition 1 ) then
    x <= value 1 ;
elsif (condition 2 ) then
    x <= value 2 ;
else
    x <= value 3 ;
end if ;

```

نتیجه در این معادله است :

```

X = condition 1 * value 1
  + / condition 1 * condition 2 * value 2
  + / condition 1 * condition 2 * condition 3 * value 3
  + _ _ _

```

يك ساختار When – else را مي توان بصورت بيان if – then – else نوشت

مالتی پلکسر 4 بیتی، چهار به يك مي تواند با ساختار if – then – elsif – else بيان شود بصورتی كه در لیست 2-11 آمده است .

Architecture arch mux of mux is

Begin

Mux 4-1 : process (a, b, c, d, s )

Begin

If s = "00" then

X <= a ;

elsif s = "01" then

X <= b ;

elsif s = "10" then

X <= c ;

Else

X <= d ;

End if ;

End process mux 4-1 ;

End arch mux ;

لیست 2-11 : تعریف يك مالتی پلکسر با يك بيان if \_ then

معادله نتیجه شده از سنتز برای این طراحی بصورت زیر است :

$$x = \overline{s1} \overline{s0} a + \overline{s1} s0 b + s1 \overline{s0} c + s1 s0 d$$

لیست 2-12 برای يك دیکودر آدرس است. طراحی كه برای يك آدرس،

1 بیتی داده شده، مشخص می كند كه برای قسمتی از فضای آدرس باید فعال شوند، و این فعال سازی به بخش فعال اختصاص یافته است. در این طراحی برای تعریف فضای حافظه و فعال سازی سیگنالهای درست آن از ساختار if – then – else – و اپراتورهای نسبی استفاده شده است . برای اینکه رنج های آدرس ( شرط ها ) متقابلاً اختصاصی هستند فقط يك نتیجه به ازای

سیگنال خروجی نیاز است، کدهایی که با معادلات بهینه شده اند بصورت زیر است .

```
Library ieee ;
```

```
Use ieee .std_logic_1164.all ;
```

```
Entity decode is port (
```

```
    Address = in std_logic_vector ( 15 downto 0 ) ;
```

```
    Valid ,boot_up = in std_logic ;
```

```
    Sram , prom , eeprom , shadow , periph 1 , periph 2 , :out std_logic ) ;
```

```
End decode ;
```

```
Architecture mem_decode of decode is
```

```
begin
```

```
mapper = process (address ,valid , boot_up )
```

```
begin
```

```
    shadow <= '0' ;
```

```
    prom    <= '0' ;
```

```
    periph 1 <= '0' ;
```

```
    periph 2 <= '0' ;
```

```
    eeprom <= '0' ;
```

```
    Sram    <= '0' ;
```

```
    If valid = '1' then
```

```
        If address >= X "0000" and address < X "4000" then
```

```
            If boot_up = '1' then
```

```
                Shadow <= '1' ;
```

```
            Else
```

```
                prom <= '1' ;
```

```
            end if ;
```

```
        elsif address >= X "4000" and address < X "4008" then
```

```
            periph 1 <= '1' ;
```

```
        elsif address >= X "4008" and address < X "4010" then
```

```
            periph 2 <= '1' ;
```

```
        elsif address >= X "8000" and address < X "C000" then
```

```
            sram 2 <= '1' ;
```

```
        elsif address >= X "C000" then
```

```
            eeprom <= '1' ;
```

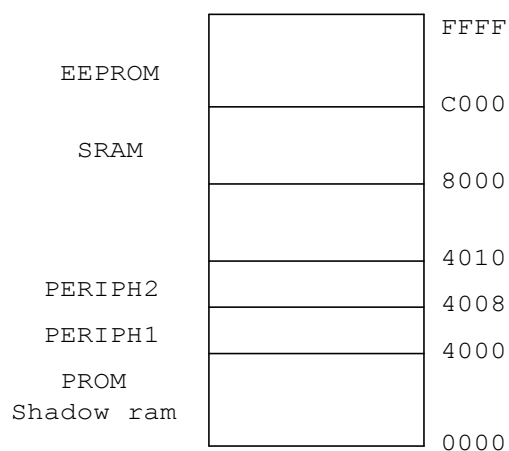
```

    end if ;
end if ;
end process ;
end mem_decode ;

```

لیست 2-12 : دیکودر آدرسی..

شکل زیر قسمتهای مختلف حافظه برای برنامه ای که گفته شده را نشان می دهد.



شکل 2-3 : نقشه حافظه

### : Case – when

بیان case خود شامل چندین بیان دیگر است و با اجرای Case یکی از آن بیانها فعال یا مقداردهی می شود و اجرای آن بر اساس مقدار سیگنالی است که در عبارت case آمده است . ساختار زیر شکل کلی بیان case را نشان می دهد که بر اساس مقدار selection\_signal اجرا میگردد .

Case selection\_signal is

When value\_1\_of\_selection\_signal =>

( do something ) \_\_ set of statement 1

When value\_2\_of\_selection\_signal =>

( do something ) \_\_ set of statement 2

When value\_3\_of\_selection\_signal =>

( do something ) \_\_ set of statement 3

---

When last\_value\_of\_selection\_signal =>  
 ( do something ) \_\_set of statements X

برای مثال لیست زیر که با عبارات case-when نوشته شده است  
 برای بیان یک دیکودر آدرس بکار رفته است .

```
Library ieee ;
Use ieee.Std_logic_1164.all
Entity test_case is port (
    Address = in std_logic_vector ( 2 downto 0 ) ;
    Decode = out std_logic_vector ( 7 downto 0 ) ;
End test_case ;
Architecture design of test_case is
    Begin
    Process ( address )
        Begin
        Case address is
            When "001" => decode <= X "11" ;
            When "111" => decode <= X "42" ;
            When "010" => decode <= X "44" ;
            When "101" => decode <= X "88" ;
            When others => decode <= X "00" ;
        End case ;
    End process ;
End design ;
```

لیست 2-13 : یک برنامه آدرسی دیکودر با استفاده از دستور case-when

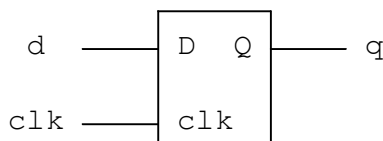
این برنامه مشخص میکند که با توجه به مقدار address چه مقدار  
 به سیگنال decode تعلق می گیرد. عبارت when others شامل همه حالاتی است  
 که در when های قبلی ذکر نشده است .  
 شما می توانید در هر شرط when به چندین سیگنال مختلف مقدار بدهید .

### 3-2- مدارات منطقی سنکرون

ابزارهای قابل برنامه ریزی در کاربردهای سنکرون مناسب طراحی شده اند. ساختار این ابزارها طراحی بلوکهایی هستند که از ترکیب لاجیکها که به ورودی یک فلیپ فلاپ وصل شده اند ، بصورت یک بلوک پایه در یک ماکروسل CPLD یک یک لاجیک سل FPGA ساخته شده اند. در این قسمت ما نشان می دهیم که چگونه می توان با زبان VHDL و با ساختارهای behavioral و Structural مدارات لاجیک سنکرون را ساخت. کد زیر یک فلیپ فلاپ ساده از نوع D را نشان می دهد .

```
Library ieee ;
Use ieee. Std_logic_1164.all ;
Entity dff_logic is port (
  d ,clk = in std_logic ;
  q  = out std_logic ) ;
End dff_logic ;
Architecture example of dff_logic is
  Begin
  Process (clk ) begin
    If ( clk' event and clk = '1' ) then
      q <= d ;
    End if ;
  End process ;
End example ;
```

لیست 2-14 : بیان یک فلیپ فلاپ از نوع D که با لبه بالارونده فعال می شود .



شکل 2-4 : بلوک دیاگرام DFF

پروسس فقط به تغییر کلاک حساس است. بنابراین یک شبیه ساز VHDL فقط زمانی که CLK حالت گذرا دارد این پروسس را اجرا می کند. شرط فقط زمانی که یک تغییری در مقدار آن است درست است. Clk حتماً باید در لیست حساسیت پروسس قرار بگیرد کلاک را باید با لبه بالارونده یا پایین رونده تعریف کرد. اگر  $clk=1$  باشد حساس به لبه بالارونده و اگر  $Clk=0$  حساس به لبه پایین رونده است و به صورت زیر نوشته می شوند :

```
If ( clk' event and clk = '1' )
یا If ( clk' event and clk = '0' )
```

شما می توانید یک latch حساس به سطح بصورت زیر تعریف کنید :

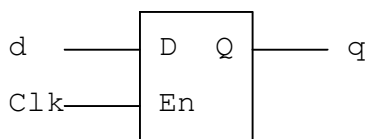
```
Process (clk , d )
```

```
Begin
```

```
Q <= d ;
```

```
End if ;
```

```
End process ;
```



شکل 2-5 : بلوک دیاگرام یک D\_latch

2 نوشته زیر برای تعریف فلیپ فلاپ یک نتیجه را دارند :

```
If (clk' event and clk = '1' ) then
```

```
q <= d ;
```

```
end if ;
```

```
If (clk' event and clk = '1' ) then
```

```
q <= d ;
```

```
else
```

```

q <= q ;
end if ;

```

در زیر دو مثال کامل یکی برای تعریف فلیپ فلاپ از نوع t و دیگری یک رجیستر 8 بیتی آمده که هر دو حساس به لبه بالا رونده کلاک عمل می کنند .

```

Library ieee ;
Use ieee. Std_logic_1164.all ;
Entity tff_logic is port (
    t,clk = in std_logic ;
    q = buffer std_logic ) ;
End tff_logic ;
Architecture t-example of dff_logic is
Begin
Process (clk ) begin
    If ( clk' event and clk = '1' ) then
        If ( t = '1' ) then
            q <= not (q) ;
        else
            q <= q ;
        end if ;
    end if ;
end process ;
end t_example ;

```

لیست 2-15 : تعریف یک فلیپ فلاپ از نوع T

```

Library ieee ;
Use ieee. Std_logic_1164.all ;
Entity reg_logic is port (
    d = in std_logic_vector ( 0 to 7 ) ;
    clk = in std_logic ;
    q = out std_logic_vector ( 0 to 7 ) ) ;
End reg_logic ;
Architecture r-example of reg_logic is

```



```

Begin
Process (clk ) begin
    If ( clk' event and clk = '1' ) then
        q <= d ;
    end if ;
end process ;
end r_example ;

```

لیست 2-16 : تعریف یک رجیستر 8 بیتی

### بیان wait untit :

شما همچنین می توانید رفتار مدار رجیستر را با استفاده از بیان wait until بجای بیان If ( clk' event and clk = '1' ) then تعریف کنید

Architecture example of dff\_logic is

```

Begin
Process begin
    Wait until ( clk = '1' )
        q <= d ;
    end process ;
end t_example ;

```

در این پروسس از یک لیست حساسیت استفاده نشده است ولی با بیان wait آغاز شده است بنابراین در یک پروسس که از این بیان استفاده شده نیازی به لیست حساسیت وجود ندارد .

### توابع مربوط به لبه بالارونده و پایین رونده :

توابع لبه بالارونده و پایین رونده در آشکار سازی لبه بالا رونده سیگنالها را تعریف کرده است . یکی از این توابع اگر سیگنال از نوع std\_logic باشد می تواند بصورت clk' event and clk = '1' تعریف شود ما می توانیم بجای clk' event and clk = '1' از rising\_edge ( clk ) استفاده کنیم . لیست 2-17 باز یک بیان از فلیپ فلاپ نوع D رانشان می دهد .

```

Library ieee ;
Use ieee. Std_logic_1164.all ;
Entity dff_logic is port (

```

```

    d ,clk = in std_logic ;
    q  = out std_logic );
End dff_logic ;
Architecture example of dff_logic is
Begin
Process (clk ) begin
    If rising_edge ( clk ) then
        q <= d ;
    End if ;
End process ;
End example ;

```

لیست 2-17 : تعریف یک فلیپ فلاپ نوع D با استفاده از تابع  
rising\_edge

### 2-3-2 reset در مدارات سنکرون :

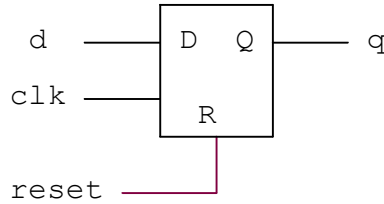
هیچکدام از مثالهایی که تا بحال گفته شده از شرط فعال سازی یا reset استفاده نشده بود . شما می توانید reset و preset را با تغییر ساده بصورتی که در لیست زیر آمده است تعریف کرد :

```

Architecture rexample of dff_logic is
Begin
Process (clk , reset ) begin
    If reset = '1' then
        q <= '0'
    elsif rising_edge ( clk ) then
        q <= d ;
    End if ;
End process ;
End example ;

```

لیست 2-18 : تعریف یک reset آسنکرون



شکل 2-6: بلوک دیاگرام DFF با reset آسنکرون

در لیست حساسیت هر دو سیگنال `reset, clk` معرفی شده اند. هر تغییر حالتی از این سیگنالها باعث اجرای ترتیبی پروسس خواهد شد. در این مدار اگر `reset` فعال شود بدون توجه به `clk` مقدار یک صفر را پیدا می کند. و اگر `reset` غیر فعال بود در اینصورت با تغییر کلاک از 0 به 1 مقدار سیگنال `d` به `q` تعلق می گیرد.

برای تعریف `preset` بجای `reset` شما می توانید تغییر زیر را انجام دهید :

```

If ( preset = '1' ) then
    q <= '1' ;
elsif rising _ edge (clk) then ---
  
```

در ضمن شما می توانید `reset` یا `preset` فلیپ فلاپها را سنکرون تعریف کنید برای اینکار شرط `reset` یا `preset` درون شرط کلاک قرار می گیرد مثال زیر نمونه ای از DFF با `reset` سنکرون را نشان می دهد .

Architecture `sync _rexample` of `dff _logic` is

```

Begin
Process (clk , ) begin
  if rising _ edge ( clk ) then
    If ( reset = '1' ) then
      q <= '0'
    else
      q <= d ;
    End if ;
  End if ;
End process ;
  
```

End example ;

لیست 2-19 تعریف یک reset سنکرون

لیست پروسس را تعریف کرده است که فقط به تغییر حساس باشد ، حاصل سنتز یک فلیپ فلاپ نوع D است که سنکرن با reset است و زمانیکه reset فعال باشد و لبه بالا رونده کلاک اتفاق بیافتد آنگاه reset عمل می نماید .

شما همچنین می توانید یک مدار سنکرون / آسنکرون با reset یا preset در VHDL تعریف کنید . مثال زیر یک رجیستر 8 بیتی را نشان می دهد که آسنکرون با reset و سنکرون با init می باشد . اگر reset فعال شد تمامی بیتها صفر می شود و اگر init فعال بوده و لبه بالا رونده کلاک هم زده شود تمامی بیتها یک می گردد :

Library ieee ;

Use ieee. Std\_logic\_1164.all ;

Entity reg\_logic is port (

d = in std\_logic\_vector (0 to 7) ;

reset , init , clk = in std\_logic

q = out std\_logic\_vector (0 to 7) ) ;

End reg\_logic ;

Architecture fancy\_example of reg\_logic is

Begin

Process (clk,reset) begin

If ( reset = '1' ) then

q<= b "00000000" ;

If ( clk' event and clk = '1' ) then

If (init = '1') then

q <= b "11111111" ;

else

q<= d ;

End if ;

End if ;

End process ;

End fancy\_example ;

لیست 20-2 یک رجیستر 8 بیتی آسنکرون با reset و سنکرون با preset کردن

### اپراتوری محاسباتی :

اپراتورهای محاسباتی شامل : جمع ، تفریق ، concatenation علامت گذاری ، ضرب ، تقسیم ، modulus باقیمانده و قدر مطلق مقدار می باشد . اغلب از اپراتورهای محاسباتی جمع و تفریق استفاده میشود که در تعریف جمع کننده ها ، تفریق کننده ها ، افزایش دهنده ها و کاهش دهنده ها استفاده می گردد . تمامی اپراتورهای محاسباتی برای انواع integer ، floatation تعریف شده است . مثال ساده زیر یک جمع کننده چهار بیتی از نوع integer را نشان می دهد .

Entity myadd is port (

a , b = in integer range 0 to 3 ;

sum = out integer range 0 to 6 ) ;

End myadd ;

Architecture archmyadd of myadd is

Begin

Sum <= a + b ;

End archmyadd ;

جمله زیر در صورتی که overflow آن لازم نباشد در کانترها مورد استفاده قرار می گیرد :

Count <= count + 1 ;

### 2-3-2 - reset و preset آسنکرون

لیست زیر یک کانتر 8 بیتی را تعریف می کند که با یک سیگنال آسنکرون خروجی مقدار "00111010" را پیدا می کند . کانتر همچنین یک enable و یک load سنکرون دارد . در این طراحی اپراتور + دو پکیج numeric\_std موجود است . پورتهای out , data از نوع unsigned تعریف شده اند .

Library ieee ;

Use ieee. Std\_logic\_1164.all ;

Use work.numeric\_std . all ;

Entity cnt8 is port (

Txclk , grst = in std\_logic ;

```

    Enable , load = in std_logic ;
    Data          = in unsigned ( 7 downto 0 ) ;
    Cnt           = buffer unsigned ( 7 downto 0 ) ;
End cnt8 ;
Architecture archcnt8 of cnt8 is
begin
    Count = process ( grst , txclk )
begin
    If ( grst = '1' ) then
        cnt <= "00111010" ;
    If ( txclk' event and txclk = '1' ) then
        If load = '1' then
            Cnt <= data ;
        Elsif enable = '1' then
            Cnt < cnt + 1 ;
        End if ;
    End process count ;
End archcnt8 ;

```

لیست 21-2 یک کانتر 8بیتی با یک سیگنال آسنکرون برای صفر و یک کردن فلیپ فلاپهای این کانتر

لیست حساسیت پروسس شامل grst و txclk می باشد . زمانی که grst فعال شده باشد cnt به یک مقدار پیش فرض بصورت آسنکرون قرار می گیرد. در لبه بالا رونده txclk اگر load فعال باشد در اینصورت مقدار data ورودی در cnt قرار می گیرد و اگر load فعال نباشد و enable فعال باشد در این حالت cnt یک مقدار به آن اضافه می شود .

### ترکیب reset با preset :

گاهی اوقات در یک طراحی نیاز است که دو سیگنال reset و preset آسنکرون باشد. لیست زیر که برای همان کانتر 8 بیتی منظور ما را برآورده می کند، لیست حساسیت شامل grst و gpst و txclk می باشد .

```
Library ieee ;
```

```

Use ieee. Std_logic_1164.all ;
Use work.numeric_std . all ;
Entity cnt8 is port (
    gpst , txclk , grst  = in std_logic ;
    Enable , load = in std_logic ;
    Data          = in unsigned ( 7 downto 0 ) ;
    Cnt           = buffer unsigned ( 7 downto 0 ) ;
End cnt8 ;
Architecture archcnt8 of cnt8 is
begin
    Count = process ( grst , gpst , txclk )
begin
    If ( grst = '1' ) then
        cnt <= ( others => '0' ) ;
    elsif gpst = '1' then
        cnt <= ( others => '1' ) then
    elsif ( txclk' event and txclk = '1' ) then
        If load = '1' then
            Cnt <= data ;
        Elsif enable = '1' then
            Cnt < cnt + 1 ;
        End if ;
    End if ;
    End process count ;
End archcnt8 ;

```

لیست 2-22 کانتر با reset و preset آسنکرون

عبارات ( others => '0' ) یا ( others => '1' ) باعث می شود که تمامی بیت‌های سیگنال مربوطه 0 یا 1 شوند. با این روش می توان فقط تعدادی را صفر یا یک کرد .  
مثال :

```
Signal a : std_logic_vector ( 7 downto 0 ) ;
```

----

```
a <= ( '1' , '0' , others => '1' ) ;
```

در اینصورت مقدار a برابر "10111111" می شود .

### 2-3-3- سیگنالهای دو طرفه و بافرهای سه حالتی :

مقادیری که سیگنال سه حالتی پیدا میکند شامل '0', '1', 'z' می باشد که همگی توسط نوع `std_logic` پشتیبانی می شوند. در کد زیر که باز همان مثال کانتر 8 بیتی است خروجی به عنوان یک سیگنال سه حالتی در نظر گرفته شده است. در این دو مثال همچنین از پکیج `std_qrith` و `std_logic_vector` برای `data` استفاده کرده ایم .

```
Library ieee ;
```

```
Use ieee. Std_logic_1164.all ;
```

```
Use work . std_arith . all ;
```

```
Entity cnt8 is port (
```

```
    txclk , grst  = in std_logic ;
```

```
    Enable , load = in std_logic ;
```

```
    Oe          = in std_logic ;          -- out put enable
```

```
    Data        = in unsigned ( 7 downto 0 ) ;
```

```
    Cnt_out      = buffer unsigned ( 7 downto 0 ) ;  -- cnt out put
```

```
End cnt8 ;
```

```
Architecture archcnt8 of cnt8 is
```

```
    Signal cnt : std_logic_vector ( 7 downto 0 ) ; -- cnt signal for counting
```

```
begin
```

```
    Count = process ( grst , txclk )
```

```
begin
```

```
    If grst = '1' then
```

```
        cnt <= "00111010" ;
```

```
    elsif rising _ edge ( txclk ) then
```

```
        if load = '1' then
```

```
            Cnt <= data ;
```

```
        Elsif enable = '1' then
```

```
            Cnt < cnt + 1 ;
```

```
        End if ;
```

```
    End if ;
```

```
    End process count ;
```

```
Oes : process ( oe , cnt )
```

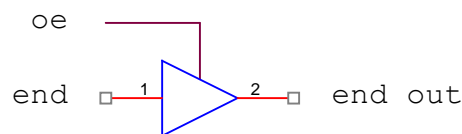


```

begin
  if oe = '0' then
    cnt_out <= (others => 'z') ;
  else
    cnt_out <= cnt ;
  end if ;
end process oes ;
End archcnt8 ;

```

سیگنال **oe** که در این شکل استفاده شده است برای این است که خروجی را به حالت امپدانس بالا قرار دهد. بنابراین به عنوان کنترل **three\_state** می باشد. در این برنامه از دو پروسس استفاده شده است و از آنجایی که یک خروجی را در دو پروسس نمی توان تغییر داد به این دلیل یک سیگنال بنام **cnt** تعریف شده است. در پروسس اول مقدار **cnt** محاسبه می شود و در پروسس دوم مقدار سیگنال **cnt** به خروجی **cnt\_out** تعلق می گیرد. شکل زیر یک بافر سه حالتی را نشان می دهد .



شکل 2-7: بافر سه حالتی

بافرهای سه حالتی همچنین می توانند با دستور **when - else** نوشته شود. در مثال پایین نمونه ای از این دستور استفاده شده است. در این مثال یک سیگنال **collision** استفاده شده است و زمانی که سیگنال **oe** = 1 باشد برابر نتیجه **and** سیگنالهای **enabe**, **load** می شود .

```

Library ieee ;
Use ieee. Std_logic_1164.all ;
Use work . std_arith . all ;
Entity cnt8 is port (
  txclk , grst : in std_logic ;
  Enable , load : in std_logic ;
  Oe           : in std_logic ;
  Data         : in std_logic_vector ( 7 downto 0 ) ;

```

```

Collision      : out std_logic ;                -- 3 state out put
Cnt_out        : buffer std_logic_vector ( 7 downto 0 ) ;
End cnt8 ;
Architecture archcnt8 of cnt8 is
Signal cnt : std_logic_vector ( 7 downto 0 ) ;
begin
Count = process ( grst , txclk )
begin
If grst = '1' then
    cnt <= "00111010" ;
elsif rising _ edge ( txclk ) then
    if load = '1' then
        Cnt <= data ;
    Elsif enable = '1' then
        Cnt < cnt + 1 ;
    End if ;
End if ;
End process count ;
-- three - state out puts described here :
cnt_out <= (other => 'z' ) when oe = '0' else cnt ;
collision <= ( enable and load ) when oe = '1' else 'z' ;
End archcnt8 ;

```

لیست 31-2 خروجی های سه حالتی که با ساختار when\_else تعریف شده اند.

### دو طرفه ها :

باتغییر اندکی در کدهای 2-30 و 2-31 براحتی می توان سیگنال دوطرفه تعریف کرد. در لیست 2-32 یک سیگنال دوطرفه معرفی شده است زمانی که لبه بالا رونده کلاک زده و Load فعال باشد این سیگنال به عنوان ورودی بوده و مقادیر آن در سیگنال قرار می گیرد . و در پروسس دوم وقتی که مخالف صفر باشد سیگنال ( که در اینجا به عنوان خروجی است ) مقدار داده می شود.

```
Library ieee ;
Use ieee. Std_logic_1164.all ;
Use work . std_arith . all ;
Entity cnt8 is port (
    txclk , grst    : in std_logic ;
    Enable , load   : in std_logic ;
    Oe              : in std_logic ;
    Cnt_out         : inout std_logic_vector ( 7 downto 0 )) ;    -- inout reg,d
End cnt8 ;
Architecture archcnt8 of cnt8 is
    Signal cnt : std_logic_vector ( 7 downto 0 ) ;
begin
    Count = process ( grst , txclk )
begin
    If grst = '1' then
        cnt <= "00111010" ;
    elsif ( txclk' event and txclk = '1' ) then
        if load = '1' then
            Cnt <= cnt_out ;    -- cnt now loaded from the cnt_out port
        elsif enable = '1' then
            Cnt < cnt + 1 ;
        End if ;
    End if ;
    End process count ;
Oes : process ( oe , cnt )
begin
    if oe = '0' then
        cnt_out <= (others => 'z') ;
    else
        cnt_out <= cnt ;
    end if ;
end process oes ;
End archcnt8 ;
```

لیست 32 \_ 2 : I/O های استفاده شده بصورت 2 طرفه

**فعال کننده های بدون شرط خروجی :**

در لیست 33-2 خروجی از یک بافر سه حالتی که بصورت بی قید و شرط تعریف شده فعال گردیده است. اگر مقدار Present\_state برابر col\_address , res\_assent , row\_address یا Cas\_assert باشد آنگاه بافرهای سه حالتی برای سیگنال dram فعال می گردد. بافرهای سه حالتی به ازای سایر مقادیر present\_state فعال نمی گردد .

```

multiplexer : process ( row_addr , col_addr , present_state )
begin
    if ( present_state = row_address or present_state = ras_assert ) then
        dram <= row_addr ;
    elsif ( present_state = col_address or present_state = cas_assert ) then
        dram <= col_addr ;
    else
        dram <= ( others => 'z' ) ;
    end if ;
end process ;

```

لیست 33 - 2 : کنترل فعال کننده خروجی Implicit

**2-3-4- ساختار دو طرفه ها و سه حالتی ها :**

با دیگر ساختار ها شما می توانید یک componet برای یک بافر سه حالتی ایجاد کنید. کدی که در لیست 33-2 قرار گرفته برای مثال، بافرهای سه حالتی ای برای لاجیک output\_enable ایجاد کرده است. اگر شما خواسته باشید Component فعال کننده خروجی را صریحاً ( واضح ) تعریف کنید ، شما ب راحتی می توانید بصورت زیر Component را تعریف کنید . ( اینجا ، نام Component ، three state است ) :

U0 : threestate port map ( cnt(0) , oe , cnt\_out(0) ) ;

**: For – Genarate**

اگر شما Component و three state راجهت پیاده سازی یک بافر سه حالتی برای یک باس 32 بیتی تعریف کنید، شما باید 32 Component جداگانه تعریف کنید که این کار پرزحمتی است. دستور For\_generate در این حالت کمک می کند.

gen\_label :

for I in 0 to 31 generate

inst\_label : threestate port map ( vaalue(i) , read , value\_out(i) ) ;

end generate ;

این دستور دو قسمت همزمان ( concurrent ) , یک architecture بدون یک پروسس پیاده سازی می شود. این دستور به نام نیاز دارد که در این جا، gen\_label است. شما همچنین می توانید در این دستور از عبارتهای شرطی نیز استفاده کنید .

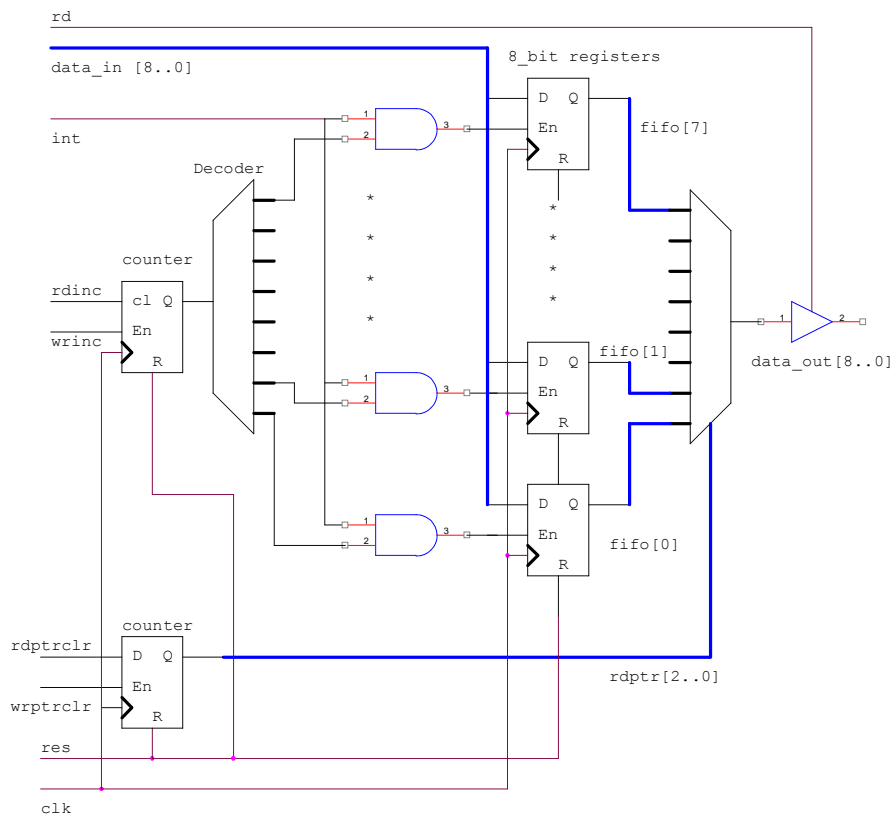
## 2-4- طراحی یک FIFO :

در این قسمت می خواهیم چگونگی ایجاد مدارات سنکرون و ترکیبی را با چندین مثل ساده نشان دهیم. ما می خواهیم که یک FIFO 9 بیتی 8 تایی را طراحی کنیم. زمانی که سیگنال خواندن (rd) فعال شد می خواهیم که خروجی data out از FIOF در زمانی که فعال نیست، خروجی به حالت امپدانس بالا قرار گیرد. و زمانی که سیگنال نوشتن (wr) فعال شد می خواهیم که یک رجیستر با پهنای 9 بیتی نوشته شود. سیگنالهای wrinc , rdinc جهت افزایش آدرس رجیسترهای خواندن و نوشتن استفاده می شود.

Rdptrelnr و Wrptrelnr برای reset کردن آدرس روی اولین رجیستر FIFO قرار می گیرد . از Data\_in برای laod کردن data در رجیسترها استفاده می شود . شکل 2-8- بلوک دیاگرام FIFO را نشان می دهد.

لیست 1- 2-33 کد VHDL برای این طراحی را نشان می دهد. این نحوه بیان ساده بوده و از مفهوم های جدید استفاده نشده است. شامل 8 رجیستر با طول 9 بیت و از نوع Std\_logic\_vector است. سیگنال FIFO برای این نوع بیان شده است. ( یک array تک بعدی از std\_logic\_vector ) ما می توانیم std\_logic\_vector ها را با مشخص کردن index نظیر , fifo(2) , fifo(1) , fifo(3) دسترسی داشته باشیم .

مفهوم جدید دیگری که می توانیم استفاده کنیم loop است که در ادامه این فصل دیده می شود



شکل 2-8: بلوک دیاگرام FIFO

## 2-5- loop ها

در جاهایی که تکرار نیاز باشد از دستورات Loop استفاده می شود . مثال for Loop ,

While\_Loop ها می باشند .

دستور For برای تعداد مشخص از حلقه ها اجرا می گردد. دستور while اجرای یک عملکرد را ادامه می دهد تا زمانی که شرط صحیح باشد. یک مرحله برای مقداردهی اولیه به متغیرهای یک دستور while نیاز است. برای نمونه ،reset آسنکرون array FIFO بااستفاده از Loop بصورت زیر است :

```
for i in 7 downto 0 loop
  fifo(i) <= (others => '0' );
end loop ;
```

دستور `while_loop` بجای `For Loop` می تواند استفاده شود، ولی باید اولاً به متغیر مقدار اولیه تعلق گیرد و در `Loop` باید متغیر را اضافه کرد .

```
reg_array : process (rst , clk )
    variable i : integer = '0' ;
begin
    if rst = '1' then
        while i < 7 loop
            fifo(i) <= ( others => '0' ) ;
        end loop ;
```

### : Conditional Iterations

دستور `next` جهت نگهداری یک عمل روی شرط خاصی استفاده می شود . برای مثال فرض کنید زمانی که `rst` فعال شد باید همه رجیسترهای `Fifo` بجز رجیستر (4) `Fifo` ریست می شوند :

```
reg_array : process (rst , clk )
    variable i : integer = '0' ;
begin
    if rst = '1' then
        for i in 7 downto 0 loop
            if i = 4 then
                next ;
            else
                fifo(i) <= ( others => '0' ) ;
            end loop ;
```

-----

یا با یک دستور `while Loop` بصورت زیر نوشته می شود :

```
reg_array : process (rst , clk )
    variable i : integer ;
begin
    i := 0 ;
    if rst = '1' then
```

```

while i < 8 loop
  if i = 4 then
    next ;
  else
    fifo(i) <= ( others => '0' ) ;
  end loop ;
-----

```

### خارج شدن از loop ها :

در loop هایی که گفته شد ما متغیر ها را درون process قرار می دادیم. دستور exit برای خارج شدن از loop استفاده می شود و می توان جهت چک کردن یک شرط غیر مجاز استفاده شود. شرط باید در compile\_time تعیین شود. برای مثال فرض کنید که یک fifo یک component ی است که در یک طراحی hierarchical معرفی شده است. درضمن فرض کنید که عمق fifo توسط یک generic یا پارامتر مشخص شده است شما ممکن است بخواهید زمانی که عمق fifo بزرگتر از مقدار از پیش تعیین شده باشد از loop خارج شود . برای مثال:

```

reg_array : process (rst , clk )
begin
  if rst = '1' then
    loop1 for i in deep downto 0 loop
      if i > 20 then
        exit loop1 ;
      else
        fifo(i) <= ( others => '0' ) ;
      end loop ;
    end loop ;
  end loop ;

```

این کد همچنین بصورت زیر می تواند نوشته شود :

```

reg_array : process (rst , clk )
begin
  if rst = '1' then
    loop1 for i in deep downto 0 loop
      exit loop1 when i > 20 ;
    else
      fifo(i) <= ( others => '0' ) ;
    end loop ;
  end loop ;

```



end loop ;

اکنون با استفاده از مفاهیم جدید برنامه Fifo را دوباره  
بازنویسی می کنیم :

Library ieee ;

Use ieee. Std\_logic\_1164.all ;

Use work . std\_arith . all ;

Entity fifoxbyy is generic ( wide : integer := 32 ); -- width is 31+1

Port (

clk , rst , oe : in std\_logic ;

rd,wr, rdinc ,wrinc : in std\_logic ;

rdptrclr , wrptrclr : in std\_logic ;

data\_in : in std\_logic\_vector ( wide downto 0 ) ;

data\_out : out std\_logic\_vector ( wide downto 0 ) ;

End fifoxbyy ;

Architecture archfifoxbyy of fifoxbyy is

Constant deep : integer := 20 ; --depth is 20 + 1

Type fifo\_array is array( deep downto 0 ) of std\_logic\_vector (wide downto 0 ) ;

Signal fifo : fifo\_array ;

Signal rdptr , wrptr : integer range 0 to deep ;

Signal en : std\_logic\_vector (deep downto 0 ) ;

Signal dmuxout : std\_logic\_vector ( wide downto 0 ) ;

begin

--fifo register array :

reg\_array : process (rst , clk )

begin

If rst = '1' then

for i in fifo, range loop

fifo(i) <= ( others => '0' ) ;

```
        end loop ;
    elsif rising_edge ( clk ) then
        if wr = '1' then
            fifo( wrptr ) <= data_in ;
        End if ;
    End if ;
End process ;

-- read pointer
read_count : process ( rst , clk )
begin
    if rst = '1' then
        rdptr <= 0 ;
    elsif rising_edge ( clk ) then
        if rdptrclr = '1' then
            rdptr <= 0 ;
        elsif rdinc = '1' then
            rdptr <= rdptr + 1 ;
        end if ;
    end if ;
end process ;

-- write pointer
write_count : process ( rst , clk )
begin
    if rst = '1' then
        wrptr <= 0 ;
    elsif rising_edge ( clk ) then
        if wrptrclr = '1' then
            wrptr <= 0 ;
        elsif rdinc = '1' then
            wrptr <= wrptr + 1 ;
        end if ;
    end if ;
end process ;
```

```
-- data output multiplexer
dmuxout <= fifo ( wrptr );

-- three_state control of outputs
three_state : process ( oe, dmuxout )
begin
    if oe = '1' then
        data _ out <= dmuxout ;
    else
        data _ out <= (others => 'z') ;
    end if ;
end process ;
end archfifoxbyy ;
```

# فصل سوم :

## طراحی State machine

### 3-1- مقدمه :

در فصول گذشته بلوکهای اساسی و ساختارهای زبان در VHDL را بیان کردیم. در این فصل، ما همان مفاهیم را بصورت طراحی State machine را بررسی میکنیم. روش State machine را انتخاب کردیم برای اینکه این روش معمولا در پیاده سازی آیسهای قابل برنامه ریزی استفاده میشود. ما چگونگی تعریف و سنتز State machine را با هدف طراحی بصورت بهینه شدن سرعت و سطح بیان میکنیم. در این فصل مثالهای ساده ای بیان

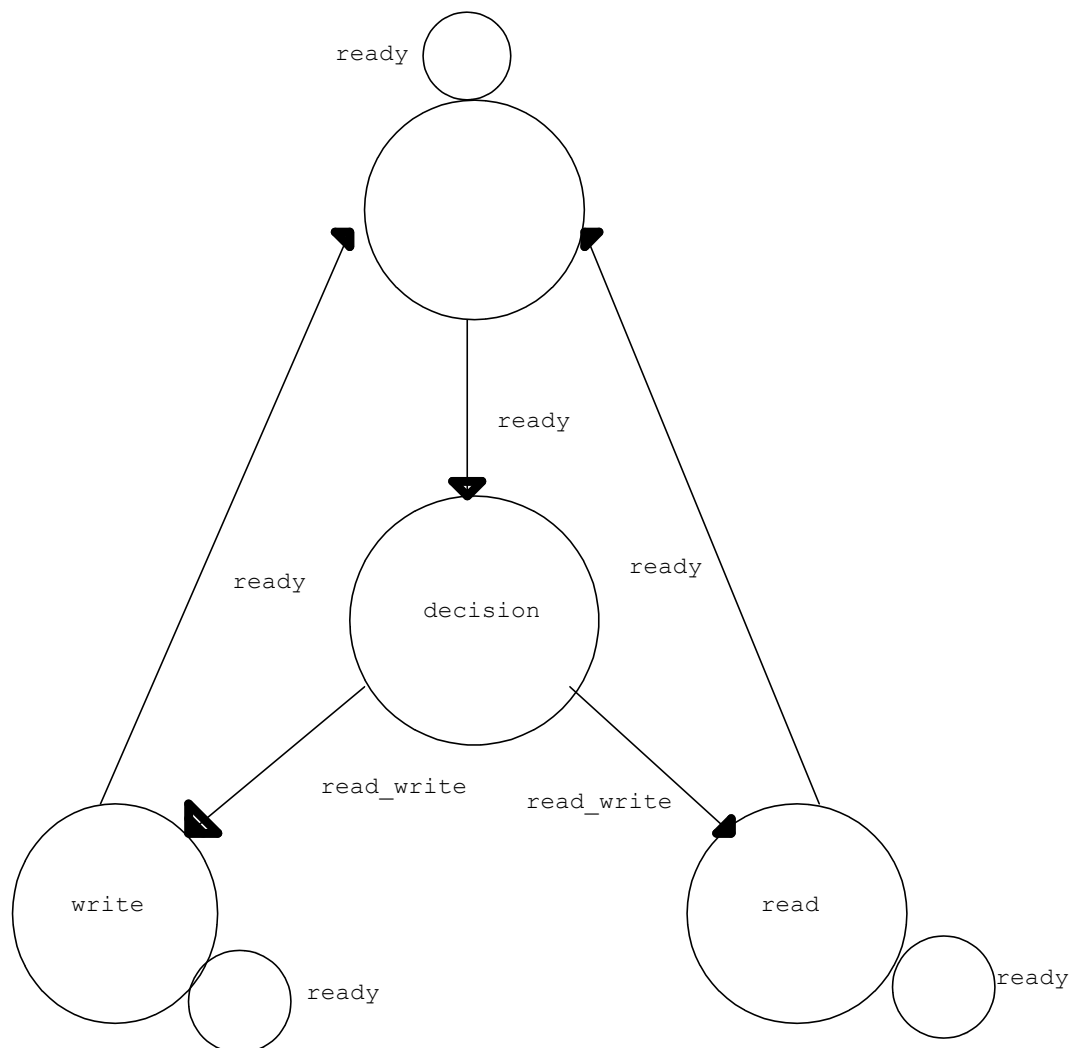
شده تا بتوان رفتار State machine در VHDL با دستورات if-when و case-when را نشان دهیم .

### 2-3 يك مثال ساده طراحی :

يك كنترلر با استفاده از سيگنالهاي كنترلي (wr) write enable و output enable (oe) انتقالات خواندن و نوشتن در يك حافظه را فعال و غير فعال مي‌کند . سيگنالهاي ready و read-write خروجي از يك ميكروپروسسور و ورودي به كنترلر هستند. سيگنالهاي we و oe خروجيهاي كنترلر هستند. مقدار read-write نشان دهنده انتقال خواندن و نوشتن است. زماني که ready فعال شود آنگاه يك انتقال مي‌تواند صورت گيرد. خروجي we در طول انتقال نوشتن و oe در طول انتقال خواندن فعال مي‌گردند.

### 3-2-1- روش طراحی مرسوم :

اولين مرحله براي تشكيل يك دياگرام حالت بايد جدول حالت را مشخص كنيم. ما با داشتن حالتهاي فعال شدن و ايجاد حالتهاي گذرا قادر هستيم که حالتهاي بعدي و خروجي را مشخص كنيم. از تعريف مسئله و مشخص کردن حالتهاي مختلف شکل 3-1 حاصل مي‌گردد.



شك 3-1: State machine ساده

این دیاگرام نشان می‌دهد که یک انتقال خولندن یا نوشتن با فعال شدن **ready** آغاز می‌گردد که حالت **State machine** از **idle** به **decision** تغییر پیدا می‌کند. با توجه به مقدار **read-write** در کلاک بعدی، انتقال خواندن یا نوشتن فعال شده و **State machine** به حالت مربوطه هدایت می‌شود. یک انتقال زمانی پایان می‌یابد که **ready** فعال شده باشد و کنترلر در حالت **idle** قرار بگیرد. زمانی که **ready** فعال نمی‌شود کنترلر در حالت خود باقی می‌ماند. حالتهای برابر در این دستگاه وجود ندارد.

دیگرام حالتی که نشان داده شده براحتی می‌تواند با یک بیان سطح بالای VHDL بدون داشتن شکل فعال شدن حالتها، جدول تولید حالتها، گذرا یا مشخص بودن حالتها بعدی بر مبنای نوع فلیپ‌فلاپهای موجود، تعریف شود. در VHDL هر حالتی می‌تواند به کمک یک ساختار case-when تعریف شود. تغییر حالتها می‌تواند با دستورات if-else-then مشخص شود. برای مثال، ما یک type تعریف کرده‌ایم که شامل اسامی حالتهاست و دو سیگنال از این type معرفی کرده‌ایم :

```
type statetype is (idle , decision , read , write);
```

```
signal present_state , next_state : statetype;
```

سپس ما یک پروسس ایجاد می‌کنیم . next\_state توسط یک تابعی از present\_state و ورودیهای ready و read-write تعریف شده است. بنابراین لیست حساسیت شامل این سیگنالها است:

```
state_comb : process (present_state , read_write , ready)
```

```
begin
```

```
.....
```

```
end process state_comb;
```

در پروسس حالتها مختلف را بیان کرده‌ایم . ما یک ساختار case-when ایجاد کرده‌ایم و اولین حالت را idle\_state قرار داده‌ایم. برای این حالت خروجیهایی که با idle\_state فعال می‌شوند را مشخص می‌کنیم :

```
state_comb : process (present_state , read_write , ready)
```

```
begin
```

```
case present_state is
```

```
when idle =>
```

```
oe <= '0' ; we<='0';
```

```
if ready='1' then
```

```
next_state <= decision;
```

```
else
```

```
-- else not necessary
```

```
next_state <= idle;
```

```
-- include for readability
```

```
end if;
```

در این حالت دو راه وجود دارد ( البته زمانی که present\_state ، idle است ) :

1- اگر ready اتفاق افتد به decision تغییر حالت می‌دهد .

2- باقی ماندن در حالت idle .

حالت‌های گذرا در این برنامه باید کامل قرار گیرند : شاخه‌ای از بیان case برای هر حالتی (when state\_name =>) ، تعیین کردن خروجی‌های آن حالت و تعیین حالت‌های گذرا با بیان if-then-else . در زیر، برنامه کامل از حالت‌های گذرا و خروجی‌ها آمده است :

State\_comb : process(present\_state , read\_write , ready) begin

Case present\_state is

When idle => oe<='0' ; we<='0';

If ready='1' then

Next\_state <= decision;

Else

Next\_state <= idle;

End if;

When decision => oe<='0' ; we<='0';

If (read\_write='1') then

Next\_state <= read;

Else

Next\_state <= write;

End if;

When read => oe<='1' ; we<='0';

If ready='1' then

Next\_state <= idle;

Else

Next\_state <= read;

End if;

When write => oe<='0' ; we<='1';

If ready='1' then

Next\_state <= idle;

Else

Next\_state <= write;

End if;

End case;

End process state\_comb;



**FSM دو پروسس :** پروسس فوق به این اشاره دارد که چگونه حالت بعدی (next\_state) با توجه به حالت جدید و ورودیها تعیین میشود. این رخداد سنکرون با لبه بالا رونده کلاک است که باید بصورت یک پروسس دیگر تعریف شود. شبیه آنچه که در زیر آمده است برای اینکه این FSM با دو پروسس تعریف شده است ما آنرا FSM دو پروسس نامیده ایم. بیان دو پروسس بصورت زیر است:

```
State_clocked : process(clk) begin
```

```
  If (clk' event and clk='1') then
```

```
    Present_state <= next_state;
```

```
  End if;
```

```
End process state_clocked;
```

The complete code for this two-process FSM follows.

Entity example is port(

```
  Read_write , ready , clk : in bit;
```

```
  Oe , we      : out bit );
```

End example;

Architecture state\_machine of example is

```
  Type statetype is (idle , decision , read , write);
```

```
  Signal present_state , next_state : state_type;
```

Begin

```
State_comb : process(present_state , read_write , ready) begin
```

```
  Case present_state is
```

```
    When idle =>    oe<='0' ; we<='0';
```

```
      If ready='1' then
```

```
        Next_state <= decision;
```

```
      Else
```

```
        Next_state <= idle;
```

```
      End if;
```

```
    When decision =>    oe<='0' ; we<='0';
```

```
      If (read_write='1') then
```

```
        Next_state <= read;
```

```

    Else
        Next_state <= write;
    End if;
When read =>    oe<='1' ; we<='0';
    If ready='1' then
        Next_state <= idle;
    Else
        Next_state <= read;
    End if;
When write =>    oe<='0' ; we<='1';
    If ready='1' then
        Next_state <= idle;
    Else
        Next_state <= write;
    End if;
End case;
End process state_comb;

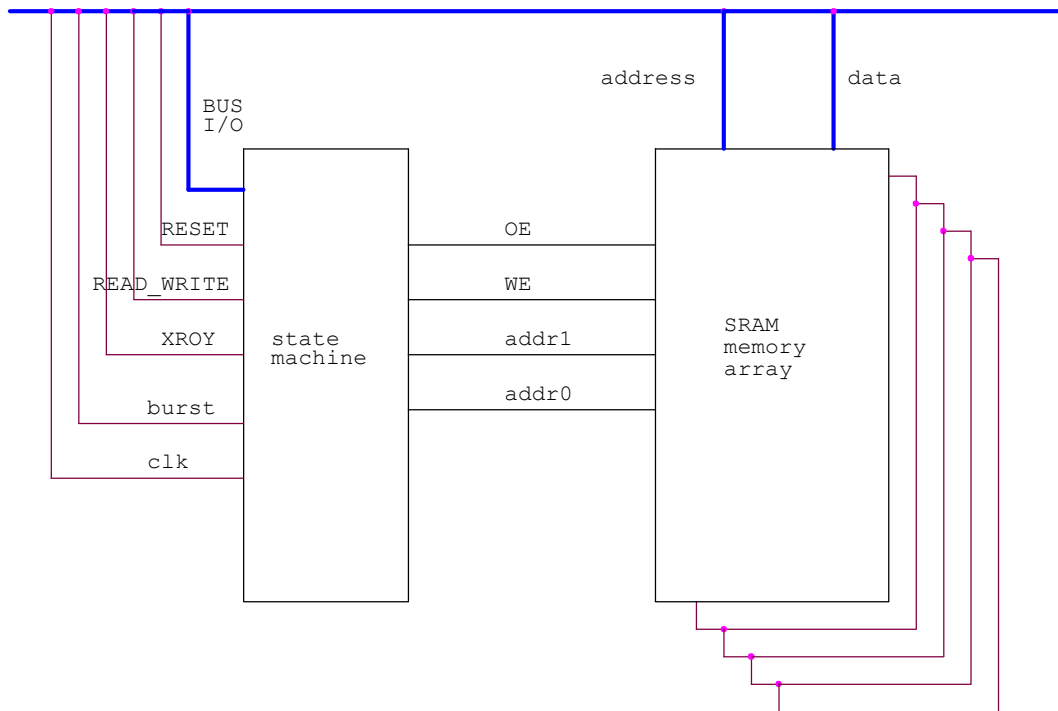
State_clocked : process(clk) begin
    If (clk'event and clk='1') then
        Present_state <= next_state;
    End if;
End process state_clocked;
End architecture state_machine;
“architecture” is optional ; for clarity

```

لیست 3-1 : طراحی یک کنترلر حافظه ساده

### 3-3- یک کنترلر حافظه

مثال طراحی بعدی باز یک کنترلر حافظه است، اما آن عملی‌تر و جزئیات بیشتری نسبت به مثال قبلی دارد. شکل 2-3 یک بلوک دیاگرام از سیستمی که یک State machine برای یک کنترلر حافظه استفاده کرده را نشان می‌دهد.



شکل 2-3 : بلوک دیاگرام کنترلر حافظه

سیستم به این صورت کار می‌کند: ابزارهای دیگری روی باس قرار گرفته‌اند که با شناسایی مقدار F3(hex) دریافتی به منظور بافر کردن حافظه استفاده می‌شوند. یک سیکل بعد، سیگنال read\_write که نشان‌دهنده این است که یک انتقال نوشتن روی بافر حافظه قرار گرفته است. اگر حافظه انتقال خواندن دریافت کند، خواندن ممکن است یا بصورت خواندن از نوع single باشد یا از نوع burst (انفجاری). یک خواندن درست با فعال شدن burst در طول اولین کلاک مشخص می‌شود، که باعث می‌شود کنترلر به چهار قسمت از بافر دسترسی داشته باشد. محلهای متوالی با فعال شدن متوالی ready دسترسی می‌شوند. کنترلر فعال کننده خروجی oe را بعد از یک خواندن غیر فعال کرده و دو بیت آدرس را یکی کاهش می‌دهد.

انتقال نوشتن در بافر همیشه بصورت single صورت می‌گیرد نه بصورت burst. در طول انتقال نوشتن با مشخص کردن آدرس data، در حافظه نوشته می‌شود. عمل خواندن و نوشتن با فعال شدن ready تمام می‌شود.

### 3-3-1- تبدیل دیاگرام حالت به VHDL

دیاگرام حالت بسادگی می‌تواند با یک سری از حالت‌هایی در ساختار case - when بصورت زیر تعریف شود ( در اینجا reset سنکرون نادیده گرفته شده است ) :

```

case present_state is
  when idle =>      oe<='0'; we<='0' ; addr<="00";
    if (bus_id = '11110011') then
      next_state <= decision;
    else
      next_state <= idle;
    end if;
  when decision =>  oe<='0'; we<='0' ; addr<="00";
    if (read_write='1') then
      next_state <= read1;
    else
      next_state <= write;
    end if;
  when read1 =>      oe<='1'; we<='0' ; addr<="00";
    if (ready='0') then
      next_state <= read1;
    elsif (burst='0') then
      next_state <= idle;
    else
      next_state <= read2;
    end if;
  when read2 =>      oe<='1'; we<='0' ; addr<="01";
    if (ready='1') then
      next_state <= read3;
    else
      next_state <= read2;
    end if;
  when read3 =>      oe<='1'; we<='0' ; addr<="10";
    if (ready='1') then

```

```

        next_state <= read4;
    else
        next_state <= read3;
    end if;
when read4 =>      oe<='1'; we<='0' ; addr<="11";
    if (ready='1') then
        next_state <= idle;
    else
        next_state <= read4;
    end if;
when write =>      oe<='0'; we<='1' ; addr<="00";
    if (ready='1') then
        next_state <= idle;
    else
        next_state <= write;
    end if;
end case;

```

شبيه آنچه كه مي‌بينيد هر حالت بسادگي با يك بيان case بدست مي‌آيد. براي هر حالي، خروجيهاي machine state با دستورات ترتيبي تعريف مي‌شوند، و همه حاليهاي گذرا در دستورات if-then-else تعريف شده است.

### reset سنكرون در يك FSM دو پروسس :

اين machine state به يك reset سنكرون احتياج دارد . با اعمال resert بايد هر حالي كه فعال است به حالت idle تغيير حالت دهد. ما مي‌توانيم با بيان يك دستور if-then-else در شروع پروسس بدین منظور كه اگر reset فعال گرديد به حالت idle هدايت شود، تعريف كنيم. و اگر reset فعال نباشد برنامه كار عادي خود را انجام دهد. همچنين در reset بايد خروجيهاي oe و wr و addr را در حالت پيش فرض قرار دهيم. نحوه نوشتن reset بصورت زير است:

```

state_comb : process (reset , present_state , burst , read_write , ready)
begin

```

```

        if (reset='1') then
            oe<='-' ; we='-' ; addr="—";
            next_state <= idle;
        else
            case present_state is
                .....
            end case;
        end if;
    end process state_comb;

```

کد کامل از State\_machine کنترلر حافظه در لیست 2-3 نشان داده شده است.

```

library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
    reset , read_write, ready ,
    burst , clk      : in std_logic;
    bus_id           : in std_logic_vector(7 downto 0);
    oe , we          : out std_logic;
    addr             : out std_logic_vector(1 downto 0));
end memory_controller;

architecture state_machine of memory_controller is
    type state_type is (idle , decision , read1 , read2 , read3 , read4 , write);
    signal present_state , next_state : statetype;
begin
    state_comb : process(reset , bus_id , present_state , burst , read_write , ready)
    begin
        .
        if (reset='1') then
            oe <= '-' ; we <= '-' ; addr <= "—";
            next_state <= idle;
        else

```

```

case present_state is
    when idle =>      oe<='0'; we<='0' ; addr<="00";
    if (bus_id = '11110011') then
        next_state <= decision;
    else
        next_state <= idle;
    end if;
    when decision =>   oe<='0'; we<='0' ; addr<="00";
    if (read_write='1') then
next_state <= read1;
    else
        next_state <= write;
    end if;
    when read1 =>      oe<='1'; we<='0' ; addr<="00";
    if (ready='0') then
        next_state <= read1;
    elsif (burst='0') then
        next_state <= idle;
    else
        next_state <= read2;
    end if;
    when read2 =>      oe<='1'; we<='0' ; addr<="01";
    if (ready='1') then
        next_state <= read3;
    else
        next_state <= read2;
    end if;
    when read3 =>      oe<='1'; we<='0' ; addr<="10";
    if (ready='1') then
        next_state <= read4;
    else
        next_state <= read3;
    end if;
    when read4 =>      oe<='1'; we<='0' ; addr<="11";

```

```

    if (ready='1') then
        next_state <= idle;
    else
        next_state <= read4;
    end if;
    when write =>      oe<='0'; we<='1' ; addr<="00";
        if (ready='1') then
            .
            next_state <= idle;
        else
            next_state <= write;
        end if;
    end case;
end if;
end process state_comb;

state_clocked : process(clk) begin
    if rising_edge(clk) then
        present_state <= next_state;
    end if;
end process state_clocked;

end state_machine;

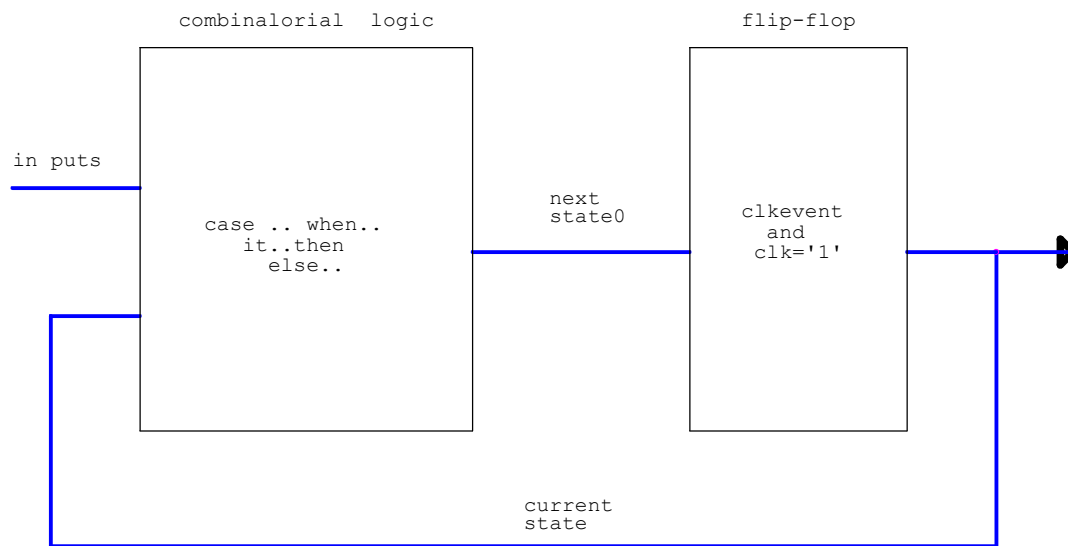
```

لیست 3-2 : تعریف کنترلر حافظه بصورت FSM دو پروسس

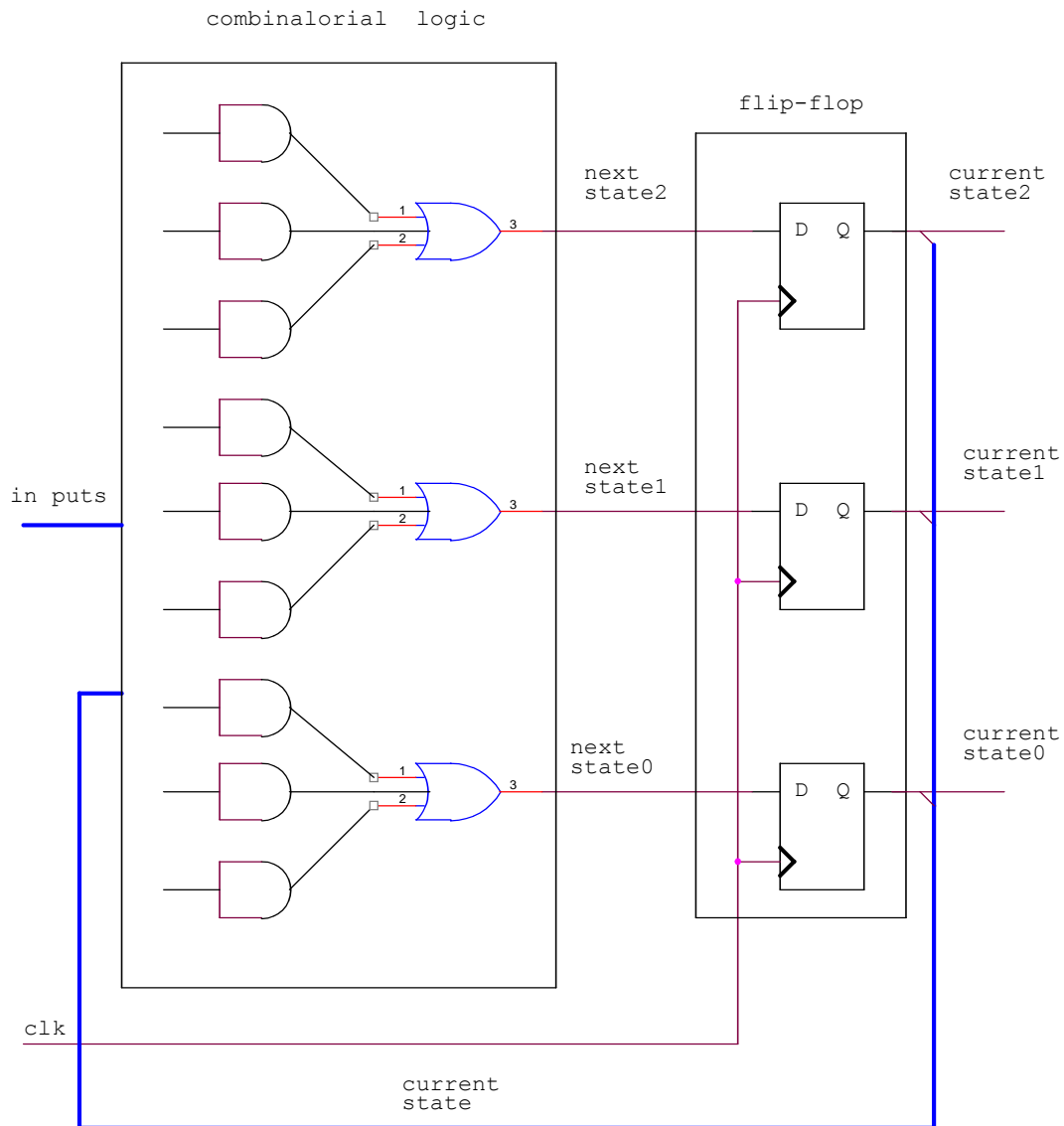
لیست 3-2 يك بيان FSM دو پروسس است. يك پروسس لاجيكهاي تركيبي و تغيير حالتها با كلاك را نشان ميدهد. اين ساختار كد قابل قياس با شكل 3-3 ميباشد كه در يك ساختار CPLD پياده شده است.

كدهاي present\_state و وروديتها در قسمت تركيبي لاجيك بلوكها بصورت آنچه در شكلها تعريف شده اجرا ميشود. حالت next\_stat سنكرون در پروسس state\_clocked تعريف شده كه بانكي از رجیسترها نظیر ماکروسلها در يك بلوك لاجيك را بيان ميكند.





شکل 3-3-الف : ساختار کد لیست 2



شکل 3-3-ب : پیاده سازی State machine در یک PLD

**reset کردن آسنکرون در یک FSM دو پروسس :**

اگر یک reset آسنکرون بجای reset سنکرون بکار رود در کد پروسس

state\_clocked باید بصورت لیست زیر نوشته شود:

```
state_clocked : process(clk , reset) begin
    if reset='1' then
        present_state <= idle;
    elsif rising_edge(clk) then
        present_state <= next_state;
```

```
end if;
```

```
end process state_clocked;
```

اگر سیگنال reset فقط برای فعال سازی سیستم یا زمانیکه مدار hang می‌کند استفاده شود، در اینصورت استفاده کردن reset آسنکرون ممکن است بهتر از یک reset سنکرون باشد .

### 2-3-3 یک ساختار کد دیگر

**FSM یک پروسس:** کدی که در لیست 3 آمده است از لحاظ معادلات انجام شده و نتایج خروجی معادل کدی است که در لیست 2 آمده است البته اگر reset آن بصورت آسنکرون استفاده شود. لیست 3-3 برای تعریف حالت‌های گذرا و سنکرون سازی این حالت‌ها با کلاک فقط از یک پروسس استفاده کرده است بنابراین ما آنرا FSM یک پروسس نامیده ایم .

architecture state\_machine of memory\_controller is

```
type state_type is (idle , decision , read1 , read2 , read3 , read4 , write);
```

```
signal present_state , next_state : statetype;
```

```
begin
```

```
state_tr : process(reset , clk) begin -- one process fsm
```

```
if (reset='1') then -- asynchronous reset
```

```
state <= idle;
```

```
elsif rising_edge(clk) then -- synchronization to clk
```

```
case state is -- state transitions defined
```

```
when idle =>
```

```
if (bus_id = '11110011') then
```

```
state <= decision;
```

```
else -- not req'd; for clarity
```

```
state <= idle;
```

```
end if;
```

```
when decision =>
```

```
if (read_write='1') then
```

```
state <= read1;
```

```
else -- read_write='0'
```

```
state <= write;
```

```
end if;
```

```
when read1 =>
  if (ready='0') then
    state <= read1;
  elsif (burst='0') then
    state <= idle;
  else
    state <= read2;
  end if;
when read2 =>
  if (ready='1') then
    state <= read3;
  else
    state <= read2;
  end if;
when read3 =>
  if (ready='1') then
    state <= read4;
  else
    state <= read3;
  end if;
when read4 =>
  if (ready='1') then
    state <= idle;
  else
    state <= read4;
  end if;
when write =>
  if (ready='1') then

    state <= idle;
  else
    state <= write;
  end if;
end case;
```

```

    end if;
end process state_tr;
--combinatorially decoded outputs
With state select
    Oe<='1' when ead1 | read2 | read3 | read4,
        '0' when others;
We <= '1' when state=write else '0';

```

```

With state select
    Addr <= "01" when read2,
        "10" when read3,
        "11" when read4,
        "00" when others;

```

```
End state_machine;
```

لیست 3-3 : تعریف یک کنترلر حافظه بصورت یک FSM تک پروسس در این بیان طراحی، حالت‌های گذرای پروسس state\_tr برای لاجیک next\_state و برای کلاک زدن state register ها استفاده شده است. فقط یک سیگنال state از نوع state type مورد نیاز است. بیانی که در زیر آمده، مقدار دهی همه سیگنال‌های ترتیبی در پروسس روی لبه بالا رونده کلاک اتفاق می‌افتد:

```

    elsif rising_edge (clk) then
        در مجموعه زیر مقدار دهی state بصورت سنکرون صورت گرفته است:
        state_tr : process(reset , clk) begin
            .....
            Elself rising_edge(clk) then
                Case state is
                    When idle =>
                        If (bus_id="11110011" then
                            State<=decision;
                        Else
                            -- not reg'd ; for clarity
                            State <= idle;
                        End if;

```

با استفاده از این ساختار، اگر state فعلی مقدار idle و باس مقدار F3(hex) را داشته باشد، آنگاه در لبه کلاک بعدی decision یک مقدار جدید پیدا می‌کند. قطعه کد بالا می‌تواند بصورت زیر تغییر کند؛ زمانی که حالت فعلی idle است، اگر bus\_id مقدار 11110011 را داشته باشد، آنگاه حالت جدید decision است. در سایر حالات که با else مشخص شده  $state \leq idle$  خواهد شد. سیگنالهای خروجی که در لیست 3-3 نشان داده شده با دستورات همزمان که مدارات آن قابل سنتز در مدارات ترکیبی است مقدار دهی شده‌اند.

## فصل 4 :

# توابع و رویه‌ها

b12bit (بولین به بیت) . لیست 4-1 يك نوع تابع تبدیل را انجام می‌دهد .

```

1      function b12bit (a : boolean) return bit is
2          begin
3              if a then
4                  return '1';
5              else
6                  return '0';
7              end if;
8      end b12bit;
```

لیست 4-1 : يك نوع تابع تبدیل بولین به بیت

لیست 4-1 يك تابع کد برای تبدیل يك سیگنال از نوع boolean به نوع bit استفاده شود را تعریف می‌کند. boolean و bit هر دو در استاندارد IEEE 1076 تعریف شده اند. خط 1 تابع b12bit را که پارامتر ورودی است و بصورت نوع boolean تعریف شده و باید به نوع bit برگردانده شود را معرفی می‌کند. خط 2 و 8 ابتدا و انتهای تابع را مشخص می‌کند.

تمامی دستوراتی که برای تعریف تابع بکار می‌روند باید از نوع دستورات ترتیبی باشند. خطوط 3 تا 7 دستورات ترتیبی هستند و سیگنال a را که مقدار آن يك نوع boolean است، برمی‌گرداند. اگر a درست باشد مقدار 1 و اگر غلط باشد مقدار 0 را خواهد داشت. دیگر توابع تبدیلی که اغلب استفاده می‌شود، bit به boolean ، bit به std\_logic و bit\_vector به std\_logic\_vector هستند.

از يك تابع تبدیل نوع bit به boolean یا boolean به bit می‌توان در نوشتن معادلات بولین یا ارزیابی شروط مورد استفاده قرار داد. برای مثال اگر سیگنال clk از نوع boolean باشد شما می‌توانید اینگونه بنویسید:

```

wait until clk ;
rather than
```



wait until clk='1'

یا

if (clk'event and clk) then ....

Rather than

If (clk' event and clk='1') then .....

همینطور

if ((A and B) xor (C and D)) then .....

می‌تواند جایگزین شود بجای

if (((A and B) xor ((C and D)='1')) then .....

یک راه لزوماً بهتر از دیگری نیست. برای طراحان نوشتن مدهای VHDL برای سنتز، نوع std\_logic ترجیح داده می‌شود.

### پارامترهای تابع :

پارامترهای تابع فقط می‌تواند ورودی باشند. بنابراین پارامترها نمی‌توانند بهینه شوند. پارامتر a در لیست 1 فقط یک ورودی است. با فرض اینکه تمامی پارامترها از مد in هستند، بنابراین نیازی نیست که مد آنها صریحاً نوشته شوند. توابع فقط می‌توانند به یک آرگومان برگردند. ( درحالی‌که procedure ها می‌توانند به چندین آرگومان برگردند ). در مجموع سیگنال جدیدی در تابع function نمی‌تواند تعریف شود، ولی متغیر variable ممکن است در تابع تعریف شود و مقادیری به آن تعلق گیرد.

### 4-2- نوع توابع تبدیل :

bv2I ( bit\_vector به integer ) . لیست 2-4 را بخوانید و چگونگی کار توابع را دریابید.

```

1      -- bv2I
2      -- bit_vector to integer.
3      -- in : bit_vector.
4      -- return ; integer.
5      --
6      function bv2I (bv : bit_vector) return integer is
7          variable result , abit : integer := 0;
8          variable count      : integer := 0;
```

```

9      begin -- bv2I
10         bits : for I in bv' low to bv' high loop
11             abit := 0;
12             if ((bv(I) = '1')) then
13                 abit := 2** (I - bv' low);
14             end if;
15             result := result + abit;      -- add in bit if '1' .
16             count := count + 1;
17             exit bits when count=32;
18         end loop bits;
19         return (result);
20     end bv2I;

```

#### لیست 2-4 : یک نوع تابع تبدیل

خطوط 1 تا 5 از لیست 2-4 توضیحاتی هستند که نشاندهنده نام تابع، تعریف نوع تابع تبدیل، و مشخص کننده پارامتر ورودی و نوعی که برمیگرداند، میباشد. به این تابع یک ورودی `bit_vector` داده میشود و یک تبدیل باینری به دسیمال را اعمال کرده و یک مقدار `integer` برمیگرداند. خط 6 یک پارامتر ورودی دارد که پهنای `bit_vector` آن محدود نشده است. اما پهنای بردار در فراخوانی تابع در زمان کامپایل باید مشخص شود. خطوط 7 و 8 تعریف `variable`هایی هستند که شبیه به پروسس در حاشیه آن ساخته میشوند. در این تابع `bv2I`، سه `variable` بصورت `integer` تعریف شده که با صفر مقدار دهی شده اند.

تابع نیز با یک `begin` شروع و با یک `end` پایان می یابد که در این برنامه خطوط 9 و 20 می باشند. خط 10 ابتدای یک `loop` را مشخص میکند که با کمترین مقدار `bit_vector` `bv` شروع میشود. صفت های `low` و `high` در VHDL معرفی شده که در اینجا برای برگرداندن کمترین و بزرگترین `bit_vector`ی که به تابع بصورت یک پارامتر داده می شود، استفاده شده است. بنابراین بدون توجه به عامل `bit_vector_bv` ( `x downto y` ) یا `y to` ( `x` ) کم ارزشترین بیت شناخته شده و مقدار `integer` که برای `bit_vector` ساخته می شود، `MSB x`، `LSB y` می باشد.

برای مثال، دو `bit_vector` ، `a` و `b` ممکن است بصورت زیر تعریف گردد:

```
signal a : bit_vector(13 downto 6);
signal b : bit_vector(6 to 13);
```

در هر کدام از این `bit_vector` ها ، `a(6)` و `b(6)` به عنوان LSB ملاحظه می گردد. تابع باید بصورتی نوشته شود که همیشه مقدار سمت چپ به عنوان MSB مشخص شود. `loop` `bit_vector` از LSB به MSB با تغییری که به عنوان `bit_vector index` استفاده شده، افزایش یافته است. `abits` با توجه به وضعیت `bit_vector` توانی به آن اختصاص داده می شود.

ملاحظه می کنید `bit_vector(13 downto 6) a` : اگر `a(8)` '1' باشد ، آنگاه `abits` مقدار `integer` 4 را پیدا می کند. برای اینکه `I` 8 است ، `low bv' 6` است ، `low bv' 2` و `2**2` 4 می باشد.

این مقدار باینری 100 را ارائه می کند، جمع `result` برای هر تکراری از `loop` مقدار `abits` است. `count` که استفاده شده مشخص کننده پهنای `bit_vector` است که به یک `integer` تبدیل شده است. رنج `integer` که در VHDL استفاده می شود به  $2^{32}$  محدود شده است. زمانی که `loop` پایان می یابد یا خارج می گردد، `result` برگردانده می شود (خط 19) ، و تبدیل `bit_vector` به `integer` پایان می یابد.

### **12bv (integer به bit\_vector) :**

تابع `i2bv` تبدیل `integer` به `bit_vector` را انجام می دهد. لیست 3-4 را بخوانید و چگونگی انجام آنرا درک کنید:

-- i2bv

--integer to Bit-vector.

--In: Integer,value and width.

--Return: bit- vector, with right Bit the most significant.

--

function i12bv (val , width : integer) return bit\_vector is

variable result : bit\_vector(0 to width-1) := (others=>'0');

variable bits : integer := width;

begin

if ( bits > 32 ) then -- avoid overflow errors.

```

    Bits := 32;
Else
    Assert 2**bits > VAL report
        "Value too big for bit_vector width"
        severity warning;
end if;

for I in 0 to bits-1 loop
    if ((val/(2**I)) mod 2 = 1) then
        result(I) := '1';
    end if;
end loop;

return (result);
end i2bv;

```

لیست 3-4 : تابع تبدیل نوع integer به bit\_vector

این تابع دو ورودی یکی مقدار integer و دیگری اندازه یا پهنای bit\_vector که باید انجام شود را می گیرد. تابع یک مبدل دسیمال به باینری است و مقدار integer را به bit\_vector برمی گرداند. در تابع تعریف شده، result بصورت یک متغیر از نوع bit\_vector تعریف شده که اندازه آن با مقدار width مشخص شده است. عرض width به عنوان یک متغیر در این تابع می باشد و آن باید از مد in بوده و نمی تواند بهینه شود. اندازه bit\_vector به 32 بیت محدود شده است.

#### 3-4- توابعی که در ساده سازی component ها استفاده می شوند :

توابع گاهی اوقات در قسمتهایی از component های جاری استفاده می شود. برای اینکه آنها یک روش برای اختصار نویسی فراهم می کنند. توابع برای جایگذاری component با یک خروجی محدود شده است. آنها دستورات wait ندارند و می توانند فقط از دستورات ترتیبی استفاده کنند.

**inv\_bv ( افزایش bit\_vector ) :**

تمرین لیست 4-4 چگونگی انجام یک تابع افزایشی را نشان می دهد .

```
-- inc_bv
--increment Bit-vector.
--In :    bit_vector.
--Return: bit- vector.
--
function inc_bv (a : bit_vector ) return bit_vector is
    variable s : bit_vector(a'range);
    variable carry : bit;
begin
    carry := '1';
    for I in a'low to a'high loop
        s(I) := a(I) xor carry;
        carry := a(I) and carry;
    end loop;
    return (s);
end inc_bv;
```

لیست 4-4 : یک تابع برای افزایش bit\_vector

تابع inv\_bv یک ورودی a از نوع bit\_vector را گرفته و مقدار آنرا افزایش می دهد، و یک bit\_vector به همان اندازه ورودی برمی گرداند. صفت rang از قبل در VHDL تعریف شده و رنج یک array را برمی گرداند. متغیر S بصورت یک bit\_vector با رنج مساوی (x downto y) یا (y to x) - شبیه به بردار ورودی a فعال شده است. carry بصورت یک بیت مشخص شده است.

**اکثریت :**

تابع بعدی یک بیتی را برمی گرداند که اکثریت سیگنالهای ورودی تابع را مشخص می کند .

function majority (a,b , c : bit) return bit is.

Begin

Return ((a and b) or (a and c) or (b and c));

End majority;

لیست 4-5 : تابع اکثریت برای سه ورودی

سپس چگونگی بکارگیری این توابع را بیان می‌کند .

#### 4-4- بیان توابع نامعلوم :

باید توجه داشت که برخی از توابع نامعلوم ممکن است در برخی از سنتزکننده‌ها نتوانند راه اندازی شوند. لیست 4-6 یک تابع اکثریت برای یک `bit_vector` با یک پهنای مشخص تعریف شده که پهنای `bit_vector` در فراخوانی تابع مشخص می‌گردد. این تابع از یک `loop` استفاده کرده که به تعداد یک‌ها در `bit_vector` تکرار می‌گردد. یک کانتر متغیر جهت نگهداری ردیابی تعداد یک‌ها استفاده شده است. سپس مقدار این متغیر با مقدار نصف `bit_vector` مقایسه می‌گردد. اگر مقدار متغیر کانتر بزرگتر از این مقدار باشد، آنگاه تابع عدد '1' و در غیر اینصورت عدد '0' را برمی‌گرداند.

function maj(vec : bit\_vector) return bit is

variable tmp : integer;

begin

temp := 0;

for I in vec'range loop

if vec(I) = '1' then

tmp := tmp + 1;

end if;

end loop;

if tmp > (vec'high)/2 then return('1');

else return('0');

end if;

end maj;

لیست 4-6 : تابع اکثریت برای یک bit\_vector با پهنای n\_bits

#### 4-5- استفاده کردن توابع :

یک تابع ممکن است در حاشیه یک architecture تعریف شود، در این حالت تابع بصورت یک تابع تعریف شده معنی پیدا می کند. شما ممکن است بخواهید یک مجموعه ای از انواع و توابع جانشینی component ها و مبدلها ایجاد کنید، و آنها را در یک package قرار دهید و آن package را در یک library کامپایل کنید و بنابراین شما می توانید براحتی آنها را در یک طراحی بکار گیرید.

یک تابعی که در یک architecture تعریف گردیده فقط توسط همان architecture می تواند استفاده گردد. یک تابع تعریف شده در یک package می تواند با یک شرطی در دیگر طراحی ها استفاده گردد .

در اینجا، اولین تابع اکثریت که در لیست 5 آمده بود را می خواهیم برای طراحی یک full adder که در لیست 4-7 آمده استفاده کنیم. ما این تابع را در محاسبه carry\_out بکار می گیریم. تعریف تابع در ابتدای architecture صورت گرفته و همچنین بصورت یک تابع تعریف شده بکار می رود.

```
entity full_add is port (
    a , b , carry_in : in bit;
    sum , carry_out : out bit);
end full_add;

architecture full_add of full_add is
    function majority (a , b , c : bit ) return bit is
    begin
        return ((a and b) or (a and c) or (b and c));
    end majority;
begin
    sum <= a xor b xor carry_in;
    carry_out <= majority(a , b , carry_in);
end ;
```

لیست 4-7 : طراحی يك full adder با استفاده از تابع اکثریت

بصورتی دیگر، تابع می‌تواند در يك تعریف package ، بیان شود و در بدنه package اختصاص یابد. در این حالت ما تابع majority را بخوبی توابع inc\_bv و i2bv در package بیان می‌کنیم.

```
package my_package is
    function majority (a , b , c : bit) return bit;
    function inc_bv (a : bit_vector) return bit_vector;
    function i2bv (val , width : integer) return bit_vector;
end my_package;
```

```
package body my_package is
    function majority (a , b , c : bit ) return bit is
    begin
        return ((a and b) or (a and c) or (b and c));
    end majority;
    -- inc_bv
    -- increment bit_vector.
    -- in : bit_vector.
    -- return : bit_vector.
    --
    function inc_bv (a : bit_vector) return bit_vector is
        variable s      : bit_vector (a'range);
        variable carry   : bit;
    begin
        carry := '1';
        for i in a'low to a'high loop
            s(i) := a(i) xor carry;
            carry := a(i) and carry;
        end loop;

        return (s);
    end loop;
```



```

-- i2bv
-- integer to bit_vector.
-- in : integer , value and width.
-- return : bit_vector , with right bit the most significant.
--
function i2bv (val , width , : integer) return bit_vector is
    variable result : bit_vector(0 to width-1) := (others => '0');
    variable bits   : integer := width;
begin
    if (bits > 32) then
        bits := 32;
    else
        assert 2**bits > VAL report
            "value too big for bit_vector width"
            severity warning;
    end if;
    for i in 0 to bits-1 loop
        if ((val/(2**i)) mod 2 = 1) then
            result(i) := '1';
        end if;
    end loop;
    return (result);
end i2bv;

end my_package;

```

لیست 4-8 : package که شامل چهار نوع تابع تبدیل است

تعریف package فقط با تعریف تابع می‌باشد. این تعریف تابع یک واسطه template برای طراحی‌هایی است که این تابع را صدا می‌کنند.

تعریف تابع در بدنه package صورت گرفته است. لیست 9 این package موجود را ساخته است. فرض براین است که package کامپایل شده و در کتابخانه work قرار گرفته است.

```

entity full_add is port(
    a , b , carry_in : in bit;
    sum , carry_out : out bit);
end full_add;

use work.my_package.majority -- could specify .all , but not needed
architecture full_add of full_add is
begin
    sum <= a xor b xor carry_in;
    carry_out <= majority(a , b , carry_in);
end;

```

لیست 4-9 : بکار بردن توابع تعریف شده در یک package

#### 4-6- اپراتورهای فراخوانی

یکی از توانایی‌های استفاده تابع در اپراتورهای فراخوانی می‌باشد. برخی از توابع فراخوانی همچنین در استانداردهای IEEE1164 و 1076.3 معرفی شده‌اند. ما می‌خواهیم چگونگی تعریف اپراتورهای فراخوانی را بیان کنیم.

اپراتور + توسط استاندارد IEEE1076 تعریف شده که روی انواع عددی ( انواع integer ، floating point ، physical ) عمل می‌کند، اما با انواع غیر عددی نظیر std\_logic یا bit\_vector عمل نمی‌کند. در عمل جمع یک integer ثابت با یک سیگنال از نوع std\_logic ، یک اپراتور فراخوانی نیاز است. لیست 4-10 برنامه جمع کردن یک integer با یک bit\_vector را نشان می‌دهد. موارد استفاده دیگر این اپراتورها نظیر جمع کردن یک bit\_vector با یک integer ، یک bit\_vector با bit ، یک std\_logic\_vector با std\_logic ، یک std\_logic\_vector با یک std\_logic می‌باشد.

```

entity counter is port(
    clk , rst , pst , load , counten : in bit;
    data : in bit_vector(3 downto 0);
    count: buffer bit_vector(3 downto 0));
end counter;

```

```

use work.myops.all;
architecture archcounter of counter is
begin
upcount : process(clk , rst , pst)
begin
    if rst='1' then
        count <= "0000";
    elsif pst='1' then
        count <= "1111";
    elsif (clk' event and clk='1') then
        if load='1' then
            count <= data;
        elsif counten='1' then
            count <= count +1;
        end if;
    end if;
end process upcount;
end archcounter;

```

لیست 4-10 : يك شمارنده با عملگر + که عملوندهایی از نوع bit\_vector و integer دارد .

VHDL اصلي این جمع را انجام نمی دهد برای اینکه عملوندها از نوع bit\_vector و integer است. اپراتورهای فراخوانی باید همراه package خودشان معرفی گردند.

شما می‌توانید چندید تابع که عملکردی مساوی برای انواع type ها داشته باشند ایجاد کنید. ابزارهای سنتز و شبیه ساز VHDL برای هماهنگی کامل احتیاج به تغییر فرم دارند.

لیست 4-11 شامل يك package تعریف شده و بدنه package است که دو اپراتور فراخوانی برای عمل + را تعریف و مشخص کرده است.

```

package myops is
function "+" (a , b : bit_vector) return bit_vector

```

```

    function "+" (a : bit_vector ; b : integer) return bit_vector
end myops;

use work.my_package .all;

package body myops is
    -- "+"
    -- add overload for;
    -- in : two bit_vector.
    -- return : bit_vector.
    --
    function "+" (a , b : bit_vector) return bit_vector is
        variable s    : bit_vector (a'range);
        variable carry : bit;
        variable bi    : integer;    -- indexes b;
    begin
        carry := '0';
        for i in a'low to a'high loop
            bi := b'low + (i - a'low);
            s(i) := (a(i) xor b(bi)) xor carry;
            carry := ((a(i) or b(bi)) and carry or (a(i) and b(bi)));
        end loop;

        return (s);
    end "+";    -- two bit_vector.

    -- "+"
    -- overload "+" for bit_vector plus integer.
    -- in : bit_vector and integer.
    -- return : bit_vector;
    --
    function "+" (a : bit_vector ; b : integer) return bit_vector is
    begin
        return (a + i2bv(b , a'length));
    end "+";
end myops;

```

لیست 4-11 : تعریف اپراتورهای توابع فراخوانی شده

این پکیج همچنین استفاده `clause` در ساختن `my_package` موجود در این واحد طراحی را نیز شامل است. تابع `i2bv` برای دومین تابع + لازم است. که این تا مقادیر `integer` را به `bit_vector` تبدیل می‌نماید. سپس در مقدار `bit_vector` با هم جمع شده‌اند. اولین اپراتور فراخوانی + ممکن است توسط این طراحی استفاده شود برای اینکه همه توضیحات یک `package` ضمناً در بدنه `package` موجود هستند. خط زیر کدی از اولین تابع فراخوانی می‌باشد که جهت بدست آوردن بیت با کمترین ارزش در یک بردار، استفاده شده است.

`bi : b' low + (I-a'low);`

زمانیکه آن بصورت مثال کانتر لیست 4-10 استفاده شده باشد، کامپایلر باید برای تابع جمع با هماهنگ بودن نوع `operand` های برای جمله `count <= count+1;` جستجو نماید.

لیست 4-10 می‌تواند بصورتی که کانتر تابع `inc_bv` را استفاده کند نوشته شود. خط `count <= count+1;` بصورت زیر جایگزین شود: `count <= inc_bv(count);`

#### 4-7- توابع فراخوانی :

اپراتورها محدود به فراخوانی توابع مربوطه نیستند بلکه شما می‌توانید هر تابعی را فراخوانی کنید. برای مثال، توابعی است که در `package majority` در لیست 4-12 بیان شده است.

`package majorities is`

-- majority for 3 single bit/std\_logic inputs

`function majority (a , b , c : bit) return bit;`

`function majority (a , b , c : std_logic) return std_logic;`

-- majority for 4 single bit/std\_logic inputs

`function majority (a , b , c , d : bit) return bit;`

`function majority (a , b , c , d : std_logic) return std_logic;`

-- majority for 2 , 3 or 4 inputs bit\_vector/std\_logic\_vector

`function majority (vec : bit) return bit;`

```
function majority (vec : std_logic) return std_logic;  
end majorities;
```

package body majorities is

```
-- majority for 3 single bit/std_logic inputs
```

```
-- function #1
```

```
function majority for 3 single bit/std_logic inputs
```

```
begin
```

```
    return ((a and b) or (a and c) or (b and c));
```

```
end majority;
```

```
-- function #2
```

```
function majority (a , b , c : std_logic) return std_logic;
```

```
begin
```

```
    return ((a and b) or (a and c) or (b and c));
```

```
end majority;
```

```
-- majority for 4 single bit/std_logic inputs
```

```
-- function #3
```

```
function majority (a , b , c , d : bit) return bit is
```

```
begin
```

```
    return ((a and b and c) or (a and b and d) or  
            (a and c and d) or (b and c and d));
```

```
end majority;
```

```
-- function #4
```

```
function majority (a , b , c , d : std_logic) return std_logic is
```

```
begin
```

```
    return ((a and b and c) or (a and b and d) or  
            (a and c and d) or (b and c and d));
```

```
end majority;
```

```
-- majority for 2 , 3 or 4 inputs bit_vector/std_logic_vector
```

```
-- function #5
```

```
function majority (vec : std_logic) return bit is
```

```
    variable a : bit_vector (vec'length -1 downto 0);
```

```
begin
    a := vec;
    if a'length = 2 then
        return (a(0) or a(1));
    elsif a'length = 3 then          -- a'length mut exc1 ; no priority
        return ((a(0) and a(1) and a(2)) or (a(0) and a(1) and a(3)) or
            (a(0) and a(2) and a(3)) or (a(1) and a(2) and a(3)));
    else
        assert (false)
            report "majority function only support 2 , 3 or 4 inputs."
            severity warning;
        return('0');
    end if;
end majority;

-- function #6
function majority (a : std_logic_vector) return std_logic;
    variable a : bit_vector (vec'length -1 downto 0);
begin
    a := vec;
    if a'length = 2 then
        return (a(0) or a(1));
    elsif a'length = 3 then          -- a'length mut exc1 ; no priority
        return ((a(0) and a(1) and a(2)) or (a(0) and a(1) and a(3)) or
            (a(0) and a(2) and a(3)) or (a(1) and a(2) and a(3)));
    else
        assert (false)
            report "majority function only support 2 , 3 or 4 inputs."
            severity warning;
        return('0');
    end if;
end majority;

end mygates;
```

لیست 4-12 : فراخوانی تابع majority برای اعداد متغیر و انواع ورودیها

شش تابع هم نام تعریف شده است. هر تابع بهر حال عملکرد majority برای یک تعداد مختلفی از ورودیها یا typeها را تعریف میکند. زمانیکه توابع majority در لیست 4-13 استفاده شده اند، کامپایلر باید توابع مختص برای صدا کردن توابع انتخاب گردد.

```
library ieee;
use ieee.std_logic_1164.all;
entity find_majority is port(
    a , b , c , d : in bit;
    e : in bit_vector(1 downto 0);
    f : in bit_vector(2 downto 0);
    g : in bit_vector(3 downto 0);
    h , i , j , k : in std_logic;
    l : in std_logic_vector(1 downto 0);
    m : in std_logic_vector(2 downto 0);
    n : in std_logic_vector(3 downto 0);
    o : in bit_vector(4 downto 0);
    p : in std_logic_vector(7 downto 0);
    x1 , x3 , x5 , x6 , x7 , x11 : bit;
    x2 , x4 , x8 , x9 , x10 , x12 : std_logic;
end find_majority;
architecture find_majority of find_majority is
begin
    -- requires function #1;
    x1 <= majority(a , b , c);
    -- requires function #2;
    x2 <= majority(h , i , j);
    -- requires function #3;
    x3 <= majority(a , b , c , d);
    -- requires function #4;
    x4 <= majority(h , i , j , k);
```



```

-- requires function #5;
x5 <= majority(e);
x6 <= majority(f);
x7 <= majority(g);
-- requires function #6;
x8 <= majority(l);
x9 <= majority(m);
x10 <= majority(n);
-- require function #5 or function #6 , but result in compile
-- time warning and function always returning '0';
x11 <= majority(o);
x12 <= majority(p);
end;

```

لیست 4-13 : استفاده توابع فراخوانی شده majority

لیست 4-13 اثبات می‌کند که تابع می‌تواند بصورت یک alternative در type های مشخص component جاری استفاده شود. این لیست همچنین بیان می‌کند که تابع majority می‌تواند در پذیرفتن مقادیر متغیر و انواع پارامترهای ورودی و انواع متغیر مقادیر برگشتی فراخوانی شود.

#### 4-8- توابع استاندارد :

خوشبختانه، توابع استاندارد ساخته شده اند بنابراین اولین ورژن مرجع زبان VHDL ( استاندارد IEEE1076 ) در سال 1987 ارائه شد. بسته های استاندارد که شامل اپراتور فراخوانی است برای چندین نوع تعریف شده اند. کد VHDL با استفاده از پکیج های استاندارد ساخته شده است بسیار ساده تر و کم حجم تر از یک ابزار دیگر است. برای مثال بسته std\_logic\_1164 یک سیستم نوع استاندارد data تهیه می‌کند چون که توسط چندین فروشنده ابزار پشتیبانی شده است. توانایی های شما با استفاده از انواع data در این بسته توسط clause و library following مشخص می‌شود :

```

library ieee;
use ieee.std_logic_1164.all;

```

برای هر ابزار سنتز یا شبیه ساز که از این package پشتیبانی میکند شما قادر خواهید بود که کدهای برنامه را بدون هیچ بهینه سازی استفاده کنید. اگر ابزاری که جهت دستیابی به اپراتورها و انواع داده ها نیاز به یک پکیج اختصاصی داشته باشد در اینصورت شما به آسانی نمیتوانید که کدهای خود را از یک سیستم به یک سیستم دیگر انتقال دهید.

استاندارد IEEE\_1164 ، فقط برای تعریف انواع داده های یک سیستم نیست بلکه اپراتورهای فراخوانی و توابع تبدیل را نیز شامل است. این استاندارد همچنین شامل زیر نوع X01 و X01Z نیز میباشد. در ضمن اپراتورهای لاجیکی ( and, or و مانند اینها ) فراخوانی کرد و بعضی از توابع تبدیل معمول را نیز فراهم کرده است:

```
function to_bit (s : std_logic; xmap : bit := '0') return bit;
```

```
function to_bitvector (s : std_logic_vector ; xmap : bit := '0')
return bit_vector;
```

```
function to_bitvector (s : std_ulogic_vector ; xmap : bit := '0')
return bit_vector;
```

```
function to_stdulogic (b : bit)
return std_ulogic;
```

```
function to_stdlogicvector (b : bit_vector)
return std_logic_vector;
```

```
function to_stdlogicvector (s : std_ulogic_vector)
return std_logic_vector;
```

```
function to_stdulogicvector (b : std_logic_vector)
return std_ulogic_vector;
```

to-bitvector , to-stdlogicvector , to-stdulogicvector همگی را میتوان برای انواع پارامترهای ورودی فراخوانی کرد. استاندارد مهم دیگر برای سنتز کردن IEEE1076.3 است. این استاندارد مشخص میکند " دو package که type های بردار را تعریف کرده است برای نشان دادن مقادیر محاسباتی علامت دار و بدون علامت، shift و انواع تبدیل عملکردها روی این انواع. پکیج های numeric\_std , numeric\_bit هستند که ما آنها را شرح داده ایم. یک

مثال از اپراتورهای محاسباتی که این استاندارد تعریف کرده در لیست 4-14 داده شده است.

بیانات زیر از numeric\_bit package می‌باشند:

type unsigned is array (natural range  $\diamond$ ) of bit;

type signed is array (natural range  $\diamond$ ) of bit;

-- id : A.3;

function "+" (L , R : unsigned) return unsigned ;

-- result subtype : unsigned(max(l'length , r'length) - 1 downto 0).

-- result : adds two unsigned vectors that may be of different lengths.

-- id : A.4;

function "+" (L , R : signed) return signed ;

-- result subtype : signed(max(l'length , r'length) - 1 downto 0).

-- result : adds two signed vectors that may be of different lengths.

-- id : A.5;

function "+" (L : unsigned; R : natural) return unsigned ;

-- result subtype : unsigned(l'length - 1 downto 0).

-- result : adds a signed vectors , l , with a non\_negative integer , r.

-- id : A.6;

function "+" (L : natural; R : unsigned) return unsigned ;

-- result subtype : unsigned(r'length - 1 downto 0).

-- result : adds a non\_negative integer , l , with an unsigned vector , r.

-- id : A.7;

function "+" (L : integer ; R : signed) return signed ;

-- result subtype : signed(r'length - 1 downto 0).

-- result : adds an integer , l(may be positive or negative) , to a signed vector , r.

-- id : A.8;

function "+" (L : signed ; R : integer) return signed ;

-- result subtype : unsigned(l'length - 1 downto 0).

-- result : adds a signed vector , l , to an integer , r.

لیست 4-15 : برخی اپراتورهای فراخوانی در numeric\_std

تابع مهم دیگری که در numeric\_std تعریف شده ، std\_match می باشد .  
تابع برای چندین type فراخوانی شده است :

-- support constant for std\_match;

type boolean\_table is array(std\_ulogic, std\_ulogic) of boolean;

constant match\_table : boolean\_table := (

-----  
-- u    x    0    1    z    w    l    h    -

-----  
(false, false, false, false, false, false, false, false, true), -- u  
(false, false, false, false, false, false, false, false, true), -- x  
(false, false, true, false, false, false, true, false, true), -- 0  
(false, false, false, true, false, false, true, false, true), -- 1  
(false, false, false, false, false, false, false, false, true), -- Z  
(false, false, false, false, false, false, false, false, true), -- w  
(false, false, true, false, false, false, true, false, true), -- L  
(false, false, false, true, false, false, false, true, true), -- H  
(true, true, true, true, true, true, true, true, true), -- -  
);

-- id : M.1

function std\_match (L , R : std\_ulogic) return boolean is

variable value : std\_ulogic;

begin

return match\_table(L , R);

end std\_match;

-- id : M.2

function std\_match (L , R : unsigned) return boolean is

```
alias LV : unsigned (1 to L'Length) is L;
alias RV : unsigned (1 to R'Length) is R;
begin
  if ((l'length < 1) or ( r'length < 1) then
    assert no_warning
      report "numeric_std.std_match : null detected, returning false"
      severity warning;
    return false;
  end if;
  if LV'length /= RV' length then
    assert no_warning
      report "numeric_std.std_match : L'length /= R'length , returning false"
      severity warning;
  else
    for i in LV'high loop
      if not (match_table(LV(i) , RV(i))) then
        return false;
      end if;
    end loop;
    return true;
  end if;
end std_match;
```

لیست 4-16 : فراخوانی کردن دو تابع std\_match