جزوه آموزشی

VHDL

فهرست

فصل سوم: طراحی ماشین حالت

فصل چهارم: توابع و رویه ها

مقدمه

آی‌سی‌های FPGA که در زمینه الکترونیک دیجیتال استفاده می‌شوند، یکی از تکنولوژیهایی است که در سالهای اخیر کاربرد وسیعی در پروژه-های صنعتی خصوصا در مدارات فرکانس بالا پیدا کرده است و در بسیاری از پروژه ها قابل رقابت با روشهای میکروپروسسوری و میکروکنترلری و در بعضی جاها نیز بسیار بهتر عمل می‌کند. آی‌سی‌های FPGA که توسعه یافته آی‌سی‌های قابل برنامه پذیر قبلی از جمله PAL, PLA, GAL, ... می-باشد 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 نهادندتا 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 حالته، 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 ارائه نمایند و قطعا 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 اين قسمت بلوكهاي اصلي زبان VHDL كه شامل Architecture , entiny مي باشد. با جزئيات كامل همراه با مثالهايي توضيح داده مي شود.

1-2- يك مثال طراحي :

مقايسه كننده چهار بيتي :

در مثال 1-1 كدهاي VHDL مربوط به يك مقايسه كنندة چهار بيتي آورده شده است كه قسمتهاي مختلف آن را توضيح مي دهيم .

```
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
       equals <='1' when ( a=b ) else'0' ;
11-
12-  end dataflow;
```

ليست1-1

همانطوريكه ديده مي شود در اين برنامه سه بخش اساسي وجود دارد. اول Library دوم توضيح entity و سوم بدنه Architecture وجود دارد.

در اول برنامه كتابخانه يا Library مورد استفاده شده توسط برنامه را بايد معرفي كنيم البته در فصول بعدی كتابخانه را معرفي می كنيم.

در زبان VHDL يك سري كلمات كليدي وجود دارند كه هر كدام معني خاص خود را دارد .

دو خط (--) كه در خط چهارم برنامه آمده براي ارائه توضيحاتي در مورد برنامه مي باشد و تاثيري در Compile برنامه و طراحي ندارد . با

اين كار مي توان روند طراحي مدار را گزارش بدهيم . بنابراين هر كجا خواستيم گزارش بنويسيم دو خط گذاشته و تا پايان آن خط به عنوان گزارش محسوب مي شود . اين دو خط مي تواند در طول يك خط و بعد از يك كد VHDL همانند گزارشي كه در خط چهارم برنامه وجود دار قرار بگيرد . در ضمن اگر گزارش بيش از يك خط باشد در ابتداي هر خطي بايد دو خط قرار بگيرد .

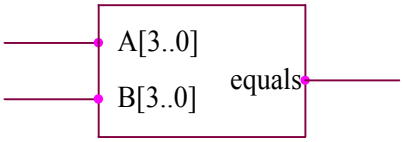
خط 5 تا 8 ورودیها و خروجيهاي مقايسه كنندة چهار بيتي كه با برنامه eqcomp4 است را مشخص مي كند . خط 5 و 8 شروع و پايان entity براي eqcomp4 را مشخص مي كنند .

خط 6 با يك Port شروع شده است و با پرانتزي كه در جلو آن مشخص شده است مي توان ورودیها و خروجيها را معرفي كنيم و بعد از پرانتز ; قرار بگيرد . در حقيقت Port ارتباطات مربوط به entity را مشخص مي كند .

در خط 6 اين مثال، دو پورت a و b معرفي شده اند كه بصورت باس يا bit – vector چهار بيتي مي باشد هر شماره اي از اين bit_vector بطور مثال a (0) به عنوان يك بيت بوده و مي تواند مقادير '0' يا '1' را بپذيرد. (در طرفين اعداد يك بيتي بايد يك quote قرار بگيرد و اگر عدد بيش از يك عدد باشد بايد دو quote قرار بگيرد بطور مثال ("0101").

در خط 7 equals معرفي شده كه به عنوان خروجي مدار مي باشد. پس به طور كلي entity پايه هاي ورودي و خروجي مدار شما را در نظر مي گيرد و به چيز ديگري كار ندارد .

نماي شماتيكي كه entity مشخص مي كند بصورت شكل 1-1 مي باشد.



شكل 1-1 : نماي شماتيكي از طرح entity

خطوط 9 تا 12 مشخص مي كنند كه entity مربوطه بايد چه كاري انجام دهد. كه اين كار در بدنه Architecture تعريف مي شود . اين عمليات با يك كلمة كليدي Architecture در خط 9 شروع و با END در خط 12 پايان يافته است.

در خط 9 يك نام (dataflow) براي Architecture از entity مربوط به eqcomp4معرفي 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 .

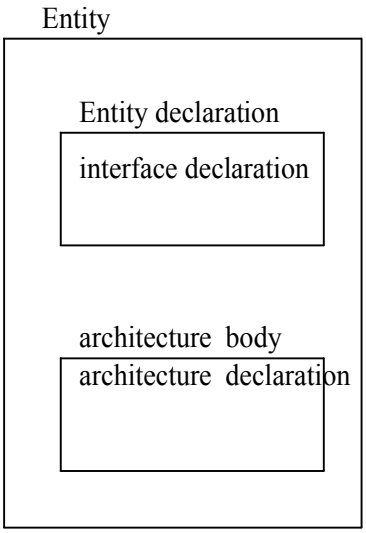
در فصلهاي بعدي روش هاي مختلف Architecture را توضيح مي دهيم.

بدنة Architecture با يك كلمة 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 دهي 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 كلمة 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 .

1-3- طراحي Entities

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

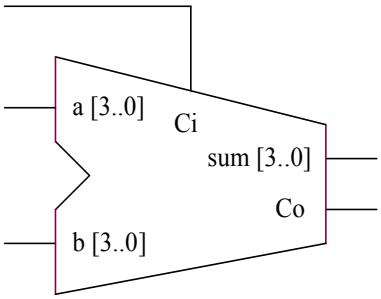


شکل 1-2 :رابطه بین تعریف entity و بدنه Architecture از یک طراحی

1-3-1- تعریف Entity

در قسمت entity ورودي ها و خروجي هاي مدار مشخص مي شوند . براي مثال يک کد VHDL براي قسمت Entity از يک Adder چهار بيتي در زير نوشته شده و نماي شماتيکي آن در شکل 1-3 نشان داده شده است . a,b به عنوان دو عدد چهار بيتي در ورودي بايد عمل جمع روي آنها صورت بگيرد . ci به عنوان عدد carry ورودي مي باشد . Sum نتيجة جمع را نشان مي دهد . و Co هم به عنوان Carry خروجي مي 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 : نماي معادل از Adder چهار بيتي

1-3-2- پورت ها

هر سیگنالي که در قسمت entity تعریف مي شود در قسمت پورت قرار مي گیرد که در حقیقت به عنوان یك پین در نماي شماتیکي است. هر پورتي که در این قسمت تعریف مي شود ابتدا باید مد آن سیگنال و سپس نوع data را مشخص کنیم .

مد (MODE) :

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

اگر براي یك پورتي مد آن را مشخص نسازیم ، بصورت پیش فرض مد آن in قرار مي گیرد. حال به توضیح این مدها مي پردازیم :

In : در این مد جهت Data بسمت داخل entity است. مقدار دهي یا راه اندازي این پورت در خارج از entity یا خارج از مدار مورد طراحي انجام مي شود. معمولاً از این مد براي ورودي کلاك، ورودیهاي کنترلي (نظیر enable , reset , load) و data هایي که فقط به عنوان ورودي هستند استفاده میگردد .

OUT: در این مد، جهت data به سمت خارج از entity است و راه اندازي این پورت توسط خود entity صورت مي گیرد . باید توجه داشت که در این پورت اجازه نداریم به عنوان فیدبك استفاده کنیم . از این مد براي کلیه سیگنالها خروجي نظیر خروجي یك کانتر یا یك مقایسه کننده استفاده مي گردد .

Buffer : این مد شبیه مد out ، خروجي بوده و مانند آن راه اندازي مي شود فقط با این تفاوت که در این مد مي توان از این پورت به عنوان فیدبك نیز در مدار استفاده کرد .

Inout : از این مد براي پورتهاي دو جهته نظیر data bus استفاده مي گردد . بنابراین راه اندازي این پورت هم مي تواند از داخل entity و هم از خارج آن صورت گیرد . مد inout اجازه گرفتن Feedback در مدار را مي دهد . این مد مي تواند جایگزین دیگر مدهاي buffer و out بشود چون تمامي خواص آنها را دارد .

نوع data (type) :

بعد از اينكه مد پورت مشخص گرديد، بايد نوع داده آن را مشخص كنيم. نوع هاي استانداردي كه توسط استاندارد IEEE 1076/93 فراهم شده و قابل سنتز مي باشند شامل integer , bit_vector , bit , Boolean مي باشد .
در ضمن نوع ديگر كه بصورت پكيج هاي IEEE Std_Logic_1164 مي باشند نظير Std_Ulogic و Std_Logic و بسيار قابل استفاده هم هستند قابل سنتز مي باشند .
براي اينكه Compiler مورد نظر بتواند اسامي Standard Logic را ستنز كند بايد نوع Library آن را مشخص كنيم. بنابراين مثال entity كه براي add4 گفته شده قابل سنتز نبوده مگر اينكه به صورت زير اصلاح يابد:

```
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 يا عملكرد مدار بيان مي شود .
VHDL اجازه مي دهد كه طراحي به شيوه هاي مختلفي صورت بگيرد . اين شيوه ها به نامهاي بيان رفتاري ، جريان داده و ساختاري مي باشند .

بيان رفتاري (behavioral) :

ليست 1-2 يك مثال از طراحي به بيان رفتاري را نشان مي دهد كه شبيه ليست 1-1 است . بيان رفتاري يك بيان سطح بالا نسبت به ديگر روش هاي طراحي مي باشد .
وقتي طراحي به زبان سطح بالا باشد ديگر لزومي به تمركز روي نحوة پياده سازي در سطح گيت نمي باشد. بنابراين در اين روش شما بيشتر به چگونگي

پياده سازي ايده هاي خود توجه داريد نه به نحوة پياده سازي مدار مورد طراحي .

```
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 ;
```

ليست 1-2

خطوط 1 و 2 براي شناسايي نوع هاي (types) كه در قسمت entityملاحظه مي گردد لازم هستند . خطوط 3 تا 6 ، قسمت entity براي يك مقايسه كنندة 4 بيتي را معرفي مي كند . خطوط 7 تا 17 بدنة Architecture را شامل مي شود كه بر اساس يك الگوريتمي، رفتار مدار را بيان مي كند . ليست Process يك ساختار طراحي VHDL است كه براي شكل دهي به الگوريتم مي باشد . ليست Process مي تواند شامل يك نام باشد كه اين نام اختياري است . (در اين مثال با نام Comp است .) اين نام با علامت : مجزا مي گردد . ليست حساسيت كه در جلو نام Process آمده است نشان دهندة اين است كه Process بر اساس اين سيگنالها اجرا مي گردد(يا حساس مي باشد). كه در اين مثال سيگنالهاي a,b ليست حساسيت Process را تشكيل مي دهند . خطوط 12 تا 16 شامل ليست هاي پشت سر هم مي باشند كه عمليات مقايسه كنندة چهار بيتي را مدل سازي كرده 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 نحوة 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 داراي 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 1-4 بیان ساختار جریان داده براي eqcomp4 .

این ساختار یك بیان جریان داده است براي اینکه بیان مي کند که چگونه داده از یك سیگنال دیگر و از ورودي به خروجي بدون استفاده از دستورات سري (Sequential Statements) انتقال پیدا مي کند . اولین اختلافي که دیده مي شد این است که در این ساختار Process تعریف نشده است . در این مثال data flow ما از جملات شرطي (When – else) استفاده کرده ایم مي توان براي پیاده سازي الگوریتم، از جملات انتخابي (with – Select – when) نیز استفاده کرد .
لیست 1-5 نشان مي دهد که چگونه با استفاده از لاجیکها مي توان مدار طراحي 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 1-5: ساختار data Flow براي eqcomp با استفاده از معادلات بولین

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

بیان ساختاري (Structural) :

لیست 1-6 را بخوانید و مشاهده کنید که چگونه یك بیان ساختاري تشکیل مي شود .

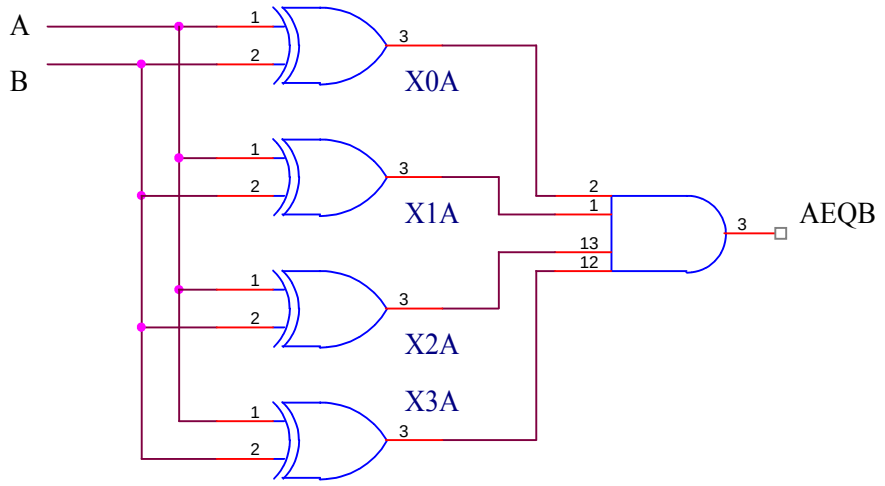
```
library ieee
use ieee.std_logic_1164.all
   entity eqcomp4 is
       port(a,b :in std_logic_vector(3 downto 0);
      equals :out std_logic);
end eqcomp4;
use work.gatespkg.all;
architecture struct of eqcomp4 is
       signal x : std_logic_vector(0 to 3);
  begin
      u0 : xnor2  port map (a(0),b(0),x(0));
      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 1-6 : بیان ساختاري eqcomp4

این طراحي نیازمند این است که Component هاي xnor2 و and4 بصورت بسته هایي (Package) تعریف شده باشند و در Library موجود و Compile شده باشند. که در این مثال این Component از پکیج gatespkg از Work library فراخواني شده است. (مطالب مربوط به Package و Library در فصلهاي بعدي گفته خواهد شد .)

بیان ساختاري شامل netlist از VHDL مي شود. این netlist خیلي متناظر با netlist شماتیك هستند. component هاي تعریف شده با سیگنالها به همدیگر اتصال پیدا کرده اند .

طراحي ساختاري بصورت سلسله مراتبي است. در این مثال طراحي eqcomp4 با استفاده از component هاي xnor2 , and 4 صورت گرفته است. شکل 1-4 حالت سلسله مراتبي را نشان مي دهد .



شکل 1-4

يك مدار مقايسه كنندة چهار بيتي معمولاً به روش ساختاري بيان نمي گردد و چنين مداري با اين روش ابتدايي است. در طراحيهاي بزرگ و پيچيده روش ساختاري يك روش مناسب مي باشد. بدين صورت كه مدار بزرگ و پيچيده به بلوكهاي كوچكتر تبديل شده و هر بلوك به روش behavioral يا data flow طراحي مي گردد و هر بلوك به عنوان يك Component در نظر گرفته مي شود و سپس به روش ساختاري اين بلوكها را به هم ارتباط داده و در نهايت تمامي مدار طراحي مي گردد .

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

در اين قسمت ما جزئيات مربوط به مشخصه ها ، Data type ، Data object و نسبتها را با استفاده از مثالهايي جهت درك بهتر مفاهيم اساسي، بيان مي كنيم .

1-4-1- مشخصه ها :

مشخصه هاي اساسي از حروف الفبايي ، عددي و پارامترهاي زير خط دار ساخته مي شوند قوانين زير در مورد مشخصه ها وجود دارد :

_ اولين كاراكتر بايد يك حرف باشد .

_ آخرين كاراكتر نبايد يك زير خط باشد .

_ دو زير خط نمي توانند كنار هم قرار بگيرد .

زبان VHDL داراي يك سري كلمات رزرو مي باشد كه نمي توان از آنها در مشخصه ها استفاده كرد. در ضمن حروف كوچك و بزرگ در مشخصه ها معادل هم هستند.

Data Object -2-4-1

Data Object مقاديري از نوع مشخص شدة خودشان، در خود نگه مي دارند. آنها شامل چهار دستة مختلف هستند :
ثابتها ، سيگنالها ، متغيرها و فايلها كه قبل از استفاده بايد آنها را معرفي كرد .

ثابتها :

ثابتها مقداري را در خود ذخيره مي كنند كه در طول برنامه ديگر نمي توان آنها را تغيير داد. اين مقادير معمولاً در ابتداي توضيح برنامه مشخص مي شوند. ثابتها مي توانند در توضيحات Architecture,entity , Package يا Process معرفي شوند . مثال زير نمونه اي از مقدار دهي ثابت ها مي باشد .

Constant width : integer : = 8 ;

سيگنالها :

توسط سيگنالها مي توان خطوط ارتباطي يا اتصالات بين Component ها را مشخص كرد . سيگنالهاي نيز مي توانند ورودي يا خروجي گيتهاي لاجيكي باشند. مثال زير نمونه اي از معرفي يك سيگنال مي باشد :

Signal count : bit – vector (3 downto) ;

سيگنالها مي توانند داراي مقدار اوليه نيز باشند :

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

در طول برنامه زمانيكه بخواهيم به سيگنال مقدار بدهيم از نماد <= استفاده مي شود در ضمن سيگنال خاصيت خواندني و نوشتني دارد .

متغيرها : Varibles

از متغيرها در Process و زير برنامه ها مي توان استفاده كرد . و در محدودة Process يا زير برنامه بايد معرفي شوند از يك متغير در دو

Process نمي توان استفاده كرد. متغيرها برخلاف سيگنالها داراي سيم و اجزاء حافظه اي نيستند و بنابراين تاخير ندارند و معمولاً براي اهداف محاسباتي استفاده مي گردند. در هنگام معرفي متغير، مي توان به آن مقدار اوليه نيز داده شود بطور مثال :

variable result : std_logic: '0';

متغييرها فقط مقداري را در يك زمان نگه مي دارند و نمي توان شكل موج خروجي آنرا مشاهده كرد. در طول برنامه براي مقدار دهي به متغير از نماد =: استفاده مي شود.

فايلها (Files) :

فايلها مقاديري از يك نوع مشخص را معين مي كنند. از فايلها معمولا در برنامه هاي تست استفاده مي گردد.

Aliases :

از يك Aliase براي شناسايي يك Object خارجي استفاده مي گردد و آن بعنوان يك Object جديد نمي باشد. يك Aliase معادل Object اصلي آن مي باشد.و معمولاً بعنوان يك روش مناسب براي مشخص كردن رنجي از يك Array type استفاده مي شود . براي مثال ، در مشخص كردن Field هايي در يك 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);
```

3-4-1- نوع داده ها Data Types :

Scalar type :

اين نوع اجازه مي دهد كه اپراتورهاي نسبي بتوانند با هم كار بكنند كه كلاً داراي سه دسته هستند. enumeration و , Ploysical , flouting integer .

Enumeration type :

يك Enumeration type يك ليستي از مقادير است كه Object ي از همان نوع را مي تواند در خود نگه دارد . Enumeration type اغلب براي State Machine تعريف مي 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 كه توسط استاندارد IEEE 1076 براي استفاده در سنتز معرفي شده است شامل Boolean و Bit است كه بصورت زير تعريف مي شوند :

Type boolean is (FALSE , TRUE);

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

استاندارد IEEE 1164 يك type اضافي بنام Std – ulogic معرفي كرده است كه داراي Sub type هاي مختلف است و براي هر دو مورد سنتز و شبيه سازي بكار مي رود . نوعStd_ulogic 9حالت مختلف داراد كه بصورت زيراند :

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 نيز داراي مقادير يكسان با Std – ulogic مي باشد .

استاندارد IEEE 1164 داراي مجموعة Std_Logic و Std _ulogic و std– ulogic_vector مي باشد .

: integer type

اپراتورهاي نسبي و integer توسط VHDL تعريف شده است. ابزارهاي نرم افزاري كه VHDL را پردازش مي كنند بايد Integer را از رنج (2^31-1)-2,147,483,687 - تا (2^31-1) 2,147,483,687 پشتيباني كنند .

يك سيگنال يا متغير كه از نوع integer است بايد با يك رنجي مشخص گردد . براي مثال :

Variable a :integer range - 255 to 255;

Flooating type :

مقاديري كه از نوع Floating_Point بكار مي رود براي اعداد تقريبي استفاده مي شود. شبيه integer انواع Flouting مي توانند محدود شوند . نوع Floating اغلب توسط ابزارهاي سنتز پشتيباني نمي شوند براي اينكه براي پياده سازي عملكردهاي محاسباتي با آنها نياز دارد كه به مقادير عددي دسترسي پيدا كند .

Physical types :

مقادير از نوع Physical بصورت واحدهاي اندازه گيري استفاده مي شود. تنها نوع فيزيكي كه تعريف 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 فيزيكي براي ايجاد برنامه هاي تست يا test benche استفاده مي كنيم. براي ديگر واحدهاي اندازه گيري پايه نظير متر ، گرم ، فوت و ... بايد خودتان تعريف كنيد .

Composite type :

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

Array type : يك Object از نوع array شامل چندين جزء با نوع يكسان است. اغلب array type كه استفاده مي 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;
```

اين نوع ها بصورت array بدون محدوده تعريف شده اند: تعداد بيت Std_logic يا Std_ulogic در آنها مشخص شده اند. array ها فقط توسط natural محدود شده اند. از اين نوع اغلب براي باسها استفاده مي گردد. براي مثال :

```
Signal   a: std_logic_vector (3 downto 0);
```

يك باس همچنين 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 كردن 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 چندين جزء با نوعهاي مختلف مي باشد .

field هاي اختصاصي از يك record مي توانند توسط نام اجزا، مرجع قرار بگيرد . برنامه زير يك record_type را نشان مي دهد كه كه براي iocell تعريف شده است ، object بصورت آن نوع، تعريف و مقاديري 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 ;
```

5-1- error هاي عمومي

كدي كه در ليست 1-7 آمده شامل چندين error است. ببينند آيا مي توانيد آنها را تشخيص دهيد :

```
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
          clolk , 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 <=  "111111111"                              --line 22
  elsif  rising _ edge ( clolk )  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
```

لیست1-7: برنامه اي با داشتن چندین غلط

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

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

error بعدي یك error مفهومي است فقدان یك else براي مقدار دهي equalse دیده مي شود که بایستي بصورت زیر نوشته مي شد :

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

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

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

error بعدي در خط 18 و در ليست حساسيت پروسس counter مي باشد اولاً بايد clk به clock تغيير يابد و ثانياً علاوه بر clock سيگنال reset نيز در ليست حساسيت قرار بگيرد. خط 21 نيز شامل error است : در array يك عدد 1 اضافي است .

error بعدي در خط 22 است. در استفاده تابع rising_edge سيگنال بايد از نوع std_logic باشد. بنابراين يا بايد نوع سيگنال clock را به std_logic تغيير داد يا اينكه از تابع استفاده شود

درخط 23 نيز يك error وجود دارد. اپراتور + براي نوع std_logic_vector يا integer تعريف نشده است بنابراين ما بايد پكيج std_arith را به برنامه اضافه كنيم. خط 27 شامل error بعدي است. حرف 2 تنها بايستي بصورت بزرگ نوشته شود نه كوچك .

z به ejbect ي تعلق مي گيرد كه از نوع std_logic باشد. آخرين error در خط 28 قرار گرفته است. كه تساوي count با "11111111" غلط است براي نوشتن صورت بايد تابع std_match استفاده شده باشد. در اين حالت برنامه بصورت زير نوشته شود :

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

كد صحيح در ليست 1-8 قرار گرفته است .

```
Library ieee ;
Use ieee. Std_logic_1164.all ;
Use work.std_arith.all
Entity terminal_count is port (
        clolk , 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 1-8 : کد VHDL تصحیح یافته لیست 1-7

فصل دوم
ايجاد مدارات تركيبي و سنكرون

2-1- مقدمه

در قسمت قبل توضيحات مربوط به entity و بدنه Architectar داده شد كه entity شامل ليستي از پورتهاي مشخص مي باشد. پورتها هر كدام داراي

يك type و mode مشخص بودند. و همچنين ديديم كه بدنه Architectar ممكن است بصورت يك يا تركيبي از ساختارهاي behavioral , dataflow , structural باشد .

دراين قسمت ، چگونگي تعريف مدارات تركيبي مدارات تركيبي و سنكرون كه در ساختارهاي مختلف استفاده مي شود بيان مي شود و با مثالهاي ساده برنامه ها به چندين روش نوشته شده اند .

2-2- مدارات تركيبي

مدارات تركيبي به روشهاي مختلفي مي توانند طراحي شوند . دستورهاي همزمان در بيان dataflow و structural استفاده مي شوند و دستورهاي پشت سرهم در بيان behavioral استفاده مي شود .

2-2-1- استفاده از دستورهاي همزمان

دستورهاي همزمان خارج از يك Process قرار گرفته و بصورت همزمان اجرا مي گردند .

در بيان data flow سه نوع دستور همزمان وجود دارد : دستورهاي همزمان با جبر بولين، دستورهاي انتخابي با دستور With – seled – when و دستور شرطي when – else .

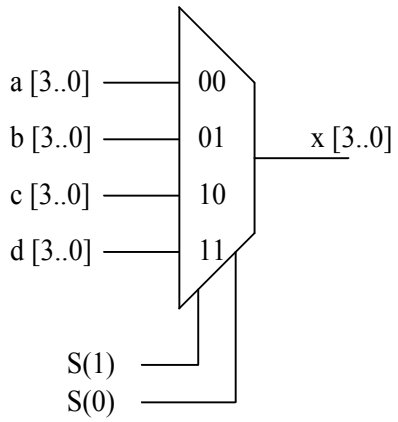
معادلات بولين :

معادلات بولين در هر نوع دستور همزمان يا پشت سرهم مي تواند استفاده شود. در اين قسمت استفاده معادلات بولين را در دستور همزمان جهت استفاده در مدارات تركيبي بيان مي كنيم. ليست زير تعريف يك مالتي پلكسر 4 به 1 كه باسهاي 4 بيتي را مالتي پلكس مي كند مي 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 مالتي پلکسر 4 بیتي .



شکل 2-1 : بلوک دیاگرام مالتي پلگسر

لیست 2-1 یک تعریف طاقت فرسا (سخت) و زبان سطح پایین است که عموماً در گذشته براي PAL استفاده مي شد بکار مي رفت.

براي اینچنین توابعي با معادلات جبر بولین بسیار ساده تر تعریف مي شوند و VHDL از این توانایي برخوردار است. براي مثال، در لیست

2-2 چندين سيگنال با استفاده از جبر بولين مقداردهي شده اند. در حالتي شبيه به اين استفاده از دستور if_then_else طاقت فرسا مي باشد. ما مثالهاي زيادي كه بر معادلات بولين مناسب هستند در ادامه جزوه خواهيم 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 ;
```

ليست 2-2 :تعريف سيگنالها با معادلات بولين

عملكرد (اپراتورهاي) لاجيكي :

اپراتورهاي لاجيكي اساس معادلات بولين هستند. اپراتورهاي لاجيكي not , xnor , xor , nand , or , and براي type هاي bit يا boolean يا براي array هاي تك بعدي از bit و بولين از پيش تعريف شده اند. در استفاده از اين اپراتورها (بجز not) با array هايي از بيت يا بولين ، دو عملوند بايد پهناي بيت مساوي داشته باشند. استاندارد IEEE 1164 اين اپراتورها را براي type هاي array , std_logic , std_ulogic هاي يك بعدي از آنها فراهم كرده است .

اپراتورهاي لاجيكي داراي اولويت هستند . براي مثال با جبر بولين، شما جملة A+B . C را انتظار داريد كه بصورت (A + (B.C ارزيابي شود. بهر حال در VHDL ، اپراتورهاي لاجيكي وجودندارد كه داراي اولويت بيشتري باشد. براي مثال در جزا سازي عمليات پرانتزها

موردنیاز هستند . جمله (A+B) . C در حقیقت بصورت زیر است : x <= A or B and C ;

with – Select - when :

دستور with – select – when سیگنال انتخاب شده براي مقدار دهي را مشخص مي كند .

در تركيب :

```
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 دهي مي شود. تمامي مقادير Selection_signal بايد در شرط هاي when وجود داشته و متقابلاً انحصاري باشند . درليست 3-2 با استفاده از اين دستور مالتی پلكسر مربوط به شكل 1-2 را تعريف مي كنيم .

```
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 ;
```

ليست 3-2- مقدار دهی سيگنال بصورت انتخابی

با توجه به مقدار s ، سیگنال x یکي از چهار مقدار a,b,c یا d را پیدا مي کند . others بجاي 11 براي موارد زیر بکار مي رود :

s بصورت type هاي از std_logic_vector باشد و 9 مقدار ممکن براي یك data object از نوع Std_Logic وجود دارد .

براي سخت افزار و ابزارهاي سنتز ، 11 فقط کقدار با معني دارد اما code باید براي VHDL امکان پذیر باشد. شما مي توانید مقدار 11 را صریحاً بصورت یکي از مقادیر s بکار ببرید ولي بهر حال حالت Others براي مشخص کردن تمامي حالات s لازم است :

```
Architecture archmux of mux is
 Begin
    With s select
     X <= a when "00"
          b when "01"
          c when "10"
          d when others ;
 end archmux ;
```

مقدار metalogical همچنین مي تواند جهت مشخص کردن مقدار x don't_care استفاده شود بصورت :

```
Architecture archmux of mux is
 Begin
    With s select
     X <= a when "00"
          b when "01"
          c when "10"
          d when "11",
         "_ _" when others ;
 end archmux ;
```

سنتز ، نتایج براي هر کدام از سه ورژن Architecture یکسان است، براي اینکه سخت افزار مشخص میکند که اگر براي S مقدار لاجیکي شناخته نشود x نیز تعریف نمي شود.

: when – else

ساختار when – else دستوري است كه براي مقدار دهي سيگنالها بصورت شرطي بكار مي رود به اين معني كه سيگنال به يك مقدار تعلق مي گيرد، در صورتيكه شرط آن برقرار باشد. ساختار كلي بصورت زير 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 مقداري كه درمعادلات شرطها وجود دارد تعلق مي يابد. اولين شرطي كه برقرار باشد مقدار آن به signal_name تعلق مي گيرد. در ليست 4-2 همان مالتي پلكسر چهار به يك 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 ;
```

ليست 4-2 : مفدار دهی سيگنال بصورت شرطی

در برنامه فوق همة شرطها در Statement و when – else بصورت انحصاري ليست شده اند.

اگر در when_else شرطها بصورت انحصاري نباشد بالاترين اولويت با اولين شرط صحيح است. اگر مقادير سيگنالها بصورت انحصاري استفاده شود ، ساختار when_else اندكي از ساختار with_select_when طولاني تر مي شود .

(شبیه مالتي پلکسر) اما این ساختار میتواند به شما درمختصر کردن بیان الگوهاي اولویت کمك کند شبیه مثال که در لیست 5-2 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 5-2 : انکودر اولویت دار با مقداردهی سیگنال بصورت شرطی

اگر a,b,c و d متناظراً منحصر بفرد هستند اگر در یك زمان فقط توسط یکي شناخته شوند. کدهایي که در لیست 6-2 آمده اختصاصي 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 6-2 : انتخاب بدون اولویت با استفاده ازمعادلات بولین .

اگر سیگنالهاي c , b , a و d انحصاري شناخته شوند حاصل ليست 2-6 معادل توابعي با ورودیهاي کمتر است . حاصل کدهايي که در ليست 2-7 آمده است با ليست 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  ;
```

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

شروط به زبان ساده :

شرط ها در جملات when_else مي تواند بصورت زبان ساده باشند اگر بصورت کدي که در زير آمده است نوشته شوند. اين نوع شرط با جملات with_select_when معمولاً امکان پذير نيست براي اينکه شرطها بايد مقاديري از سيگنال 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' ;
```

اپراتورهاي نسبي :

اپراتورهاي نسبي براي تست كردن برابري ، نابرابري و آرايشي از آنها استفاده مي شوند . اپراتورهاي برابري و نابرابري (= و =/) براي تمامي type ها معرفي مي شوند. اپراتورهاي دامنه دار (>,<=,>=,<) براي type هاي scalar يا array با يك رنج پيوسته تعريف مي شوند. array هايي كه هم ارز و داراي طول برابر هستند مي توان استفاده شوند . نتيجه اپراتورهاي نسبي، بولين است (جواب آن يا درست يا نادرست) .

oprand type ها در يك اپراتور نسبي بايد مثل هم باشند. جمله زير غلط است براي اينكه يك Std_logic_vector است و 123 يك integer است :

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

اپراتورهاي بارگذاري :

اپراتورهايي نظير اپراتورهاي نسبي ممكن است بارگذاري باشند. اپراتورهاي بارگذاري اجازه استفاده از اپراتورهاي با چندين type را مي دهند. براي هر اپراتور توسط استاندارد IEEE 1076 تعريف نشده است. اپراتور ممكن است با استفاده از تعريف function بارگذاري باشند ، اما تعدادي اپراتور بارگذاري دراستاندارد IEEE 1164 و 1076.3 تعريف شده است . براي مثال استاندارد IEEE 1076.3 تابعي براي 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 8-2 : اپراتور بارگذاری = تعریف شده در numeric _ std

توابع اپراتور بارگذاري در numeric_std package از استاندارد 1076.3 تعريف شده اند و اين Package بايد در entity طراحي با يك ماده اي (قضيه اي) از آن فعال شود . package هايي نيز براي اپراتورهاي بارگذاري براي std_logic اضافه شده است .

Std_arith package ساخته شده، براي چندين تابع جديد و اپراتورهاي رياضي بدون علامت در راه اندازي std_logic_vector تعريف شده است. مثال زير نمونه اي از اين 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 9-2 : اپراتور بارگذاری = تعریف شده در std _ arith

براي مثالهايي از اين نوع ، ما numeric_std package در بيان بكارگيري type ،توابع و اپراتورها استفاده مي كنيم ما همچنين std_arith package را در مثالهاي ديگر در بيان چگونگي استفاده از يك std_logic_rector , array type كه مي تواند overhead را كاهش دهد استفاده مي كنيم .

Package هاي numeric_std و std_arith همچنين شامل تابع مهم ديگري بنام std_match مي باشند.

این تابع در دو مقدار از نوع std_logic بصورت یك don't care یا witd – card استفاده مي شود. مقدار آن مي تواند بصورت '0' یا '1' یا '1' ارزیابي شود . این تابع استاندارد میتواند مفید باشد براي اینکه مقایسه ارزیابي زیر براي همه مقادیر بجر "1—1" است و در سخت افزار هرگز درست (true) نیست.

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

ابزارهاي سنتز و شبیه سازي باید بصورت یك مقدار ترجمه کنند. بنابراین مقایسه فقط زماني درست است که یك باشد .

گرچه ارزیابي don't care ، اپراتور = نمي تواند در شناسایي حالتهاي don't care استفاده شود.

با استفاده از تابع std_match مقایسه "1–1" از std_logic_vector (3 donwto 0) زمان درست است که بیتهاي اول و آخر یك باشند .

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

شناسایي component ها :

شناسایي component ها در جملات، همزمان مي باشد که اتصالات سیگنالها در یك طراحي را مشخص مي کند. نوشتن یك برنامه مقایسه کننده 4 بیتي لیست 10-2 غیر معقولانه است ولي براي بیان شناسایي Statement که مي تواند در پیاده سازي لاجیکها ترکیبي 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 donwto 0 ) ;
      aegb : out std_logic  ) ;
end compare ;
architecture archcompare of compare is
  signal  c :  std_logic_vector  ( 3 donwto 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 10-2 : بیان یك مقایسه کننده 4 بیتی به روش structural

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

2-2-2- با استفاده از جملات ترتیبي:

جملات ترتیبي در process ها ، تابع ها و procedur بکار مي روند . توابع و Procedure در فصل بعدي معرفي مي شوند. در این قسمت جملات ترتیبي در process که چگونه در طراحي مدارات ترکیبي استفاده مي شوند را بیان مي کنیم .

If_then_else

ساختار if_then_else براي انتخاب یا مقدار دهي جملات مي باشد، و اجراي آن بر اساس ارزیابي بولین (درست یا غلط) از یك شرط است. در ساختار زیر :

```
if ( condition ) then
  do something ;
else
  do something  different ;
end if ;
```

اگر شرط بصورت true ارزیابي شود بعد از اجرا جملة do something انتخاب مي گردد و اگر بصورت false ارزیابي گردد بعد از else جمله something different انتخاب مي گردد ساختار با end if بسته مي شود .

Process هایي که در زیر آمده یك کار را انجام مي دهند .

```
Signal step : std_logic ;
Signal addr : std_logic_vector ( 7 downto 0 ) ;
 –
 –
 –
```

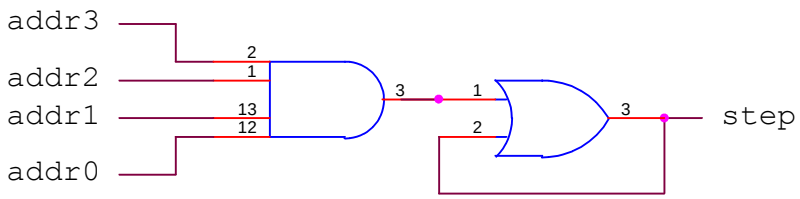
```
similar 1  = process  ( addr )
 begin
    step  <= '0'  ;
    if addr > x  "0F"  then
      step <= '1'  ;
end if  ;
end  process ;

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 كه addr بزرگتر از of hex باشد step مقدار '1' و اگر كمتر يامساوي باشد مقدار '0' را پيدا ميكند. پروسس زير لاجيك مساوي قبلي را بيان نمي كند براي اينكه غير از اين شرط هيچ مقدار پيش فرضي به step تعلق نمي يابد .

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

پروسس not_similar دلالت بر اين دارد كه step بايد زمانيكه addr كمتر يا مساوي of hex است مقداري در خود نگه دارد. اين بصورت يك خود نگه دارنده يا يك حافظه عمل مي كند . بنابراين يك بار فعال شده و step به آنچه كه فعال شده باقي مي ماند بصورتي كه در شكل 2-2 نشان داده شده و توسط معادلة زير تعريف شده است .

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



شکل 2-2

اگر شما نخواهید که به step مقدار بدهید مطمئناً داراي يك مقدار پيش فرض است يا مي توانيد كه دستور if – then را با يك else تكميل كنيد . دستور if – then – else مي تواند توسعه يابد و تعداد elsif هاي بيشتري جهت مقدار دهي و اولويت پذيري داشته باشد فرم كلي بصورت زير است :

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

براي هر سيگنال x به يك مقدار تعلق يافته بر اساس شرط بوده است ، حاصل سنتز يك معادله است براي مثال كد زير :

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

نتيجه در اين معادله است :

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

یک ساختار When – else را می توان بصورت بیان if – then – else نوشت .

مالتي پلکسر 4 بيتي، چهار به يک مي تواند با ساختار if – then – elsif – else بيان شود بصورتي که در ليست 11-2 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 ;
```

ليست 11-2 : تعريف يک مالتی پلکسر با يک بيان if_ then

معادله نتيجه شده از سنتز براي اين طراحي بصورت زير است :

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

ليست 12-2 براي يک ديکودر آدرس است. طراحي که براي يک آدرس، 1 بيتي داده شده، مشخص مي کند که براي قسمتي از فضاي آدرس بايد فعال شوند، و اين فعال سازي به بخش فعال اختصاص يافته است. در اين طراحي براي تعريف فضاي حافظه و فعال سازي سيگنالهاي درست آن از ساختار if – then – else و اپراتورهاي نسبي استفاده شده است . براي اينکه رنج هاي آدرس (شرط ها) متقابلاً اختصاصي هستند فقط يک نتيجه به ازاي

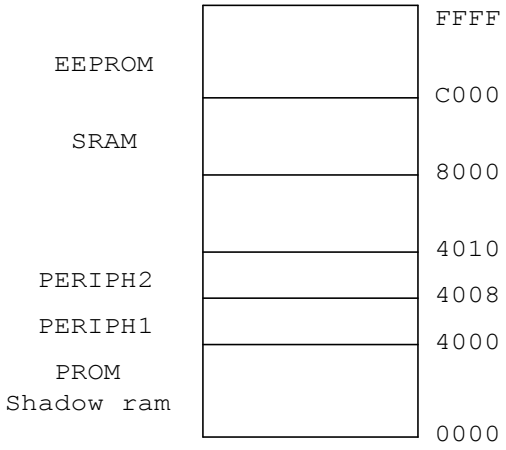
سیگنال خروجي نیاز است ،کدهایي که با معادلات بهینه شده اند بصورت زیر است .

```
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 12-2 : 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 اجراي Case یکي از آن بیانها فعال یا مقداردهي مي شود و اجراي آن بر اساس مقدار سیگنالي است که در عبارت case آمده است . ساختار زیر شکل کلي بیان case را نشان مي 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

براي مثال ليست زير كه با عبارات case – when نوشته شده است براي بيان يك ديكودر آدرس بكار 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
  Bagin
 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 ;
```

ليست 13-2 : يك برنامه آدرسي ديكودر با استفاده از دستور case – when

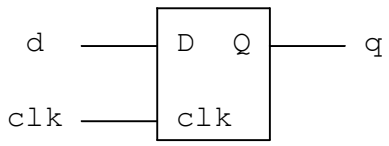
اين برنامه مشخص ميكند كه با توجه به مقدار address چه مقدار به سيگنال decode تعلق مي گيرد. عبارت when others شامل همة حالاتي است كه در when هاي قبلي ذكر نشده است .
شما مي توانيد در هر شرط when به چندين سيگنال مختلف مقدار بدهيد.

2-3- مدارات منطقي سنكرون

ابزارهاي قابل برنامه ريزي در كاربردهاي سنكرون مناسب طراحي شده اند. ساختار اين ابزارها طراحي بلوكهايي هستند كه از تركيب لاجيكها كه به ورودي يك فيلپ فلاپ وصل شده اند ، بصورت يك بلوك پايه در يك ماكروسل CPLD يك يك لاجيك سل FPGA ساخته شده اند. در اين قسمت ما نشان مي دهيم كه چگونه مي توان با زبان VHDL و با ساختارهاي behaviral و Structural مدارات لاجيك سنكرون را ساخت. كد زير يك فليپ فلاپ ساده از نوع D را نشان مي 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 ;
```

ليست 2-14 : بيان يك فليپ فلاپ از نوع D كه با لبه بالارونده فعال مي شود.



شكل 2-4 : بلوك دياگرام DFF

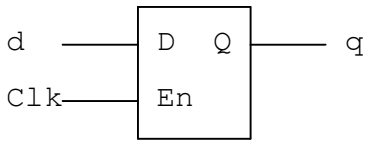
پروسس فقط به تغيير كلاك حساس است. بنابراين يك شبيه ساز VHDL فقط زماني كه CLK حالت گذرا دارد اين پروسس را اجرا مي كند. شرط فقط زماني كه يك تغييري در مقدار آن است درست است . Clk حتماً بايد در ليست حساسيت پروسس قرار بگيرد كلاك را بايد با لبه بالارونده يا پايين رونده تعريف كرد . اگر clk=1 باشد حساس به لبه بالارونده و اگر Clk=0 حساس به لبه پايين رونده است و به صورت زير نوشته مي شوند :

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

يا `If ( clk' event and clk = '0' )`

شما مي توانيد يك latch حساس به سطح بصورت زير تعريف كنيد :

```
Process (clk , d )
 Begin
 Q <= d ;
 End if ;
End process ;
```



شكل 2-5 : بلوك دياگرام يك D_latch

2 نوشتة زير براي تعريف فليپ فلاپ يك نتيجه 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 یکي براي تعریف فلیپ فلاپ از نوع t ودیگري یك رجیستر 8 بیتي آمده که هر دو حساس به لبة بالا روندة کلاك عمل مي 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 15-2 : تعریف یك 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 16-2 : تعریف یك رجیستر 8 بیتي

بیان wait untit :

شما همچنین مي توانید رفتار مدار رجیستر را با استفاده از بیان wait until بجاي 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 یك لیست حساسیت استفاده نشده است ولي با بیان wait آغاز شده است بنابراین در یك پروسس که از این بیان استفاده شده نیازي به لیست حساسیت وجود ندارد .

توابع مربوط به لبة بالارونده و پایین رونده :

توابع لبة بالارونده و پایین رونده در آشکار سازي لبة بالا رونده سیگنالها را تعریف کرده است . یکي از این توابع اگر سیگنال clk' event and clk = '1' تعریف شود از نوع std_logic باشد مي تواند بصورت rising_edge (clk) از clk' event and clk = '1' بجاي ما مي توانیم استفاده کنیم . لیست 17-2 باز یك بیان از فلیپ فلاپ نوع D رانشان مي 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  ;
```

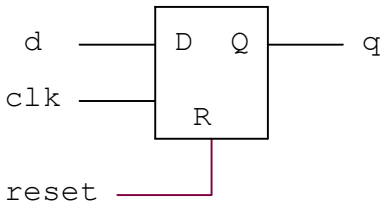
لیست 2-17 : تعریف یك فلیپ فلاپ نوع D با استفاده از تابع rising_edge

2-3-1 reset در مدارات سنکرون :

هیچکدام از مثالهایي که تا بحال گفته شده از شرط فعال سازي یا reset استفاده نشده بود . شما مي توانید reset و preset را با تغییر ساده بصورتي 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 یك reset آسنکرون



شکل 2-6: بلوك دياگرام DFF با reset آسنكرون

درليست حساسيت هر دو سيگنال reset , clk معرفي شده اند. هر تغيير حالتي از اين سيگنالها باعث اجراي ترتيبي پروسس خواهد شد. در اين مدار اگر reset فعال شود بدون توجه به clk مقدار يك صفر را پيدا مي كند. و اگر reset غير فعال بود در اينصورت با تغيير كلاك از 0 به 1 مقدار سيگنال d به q تعلق مي گيرد .

براي تعريف preset بجاي reset شما مي توانيد تغيير زير را انجام دهيد :

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

در ضمن شما مي توانيد reset يا preset فليپ فلاپها را سنكرون تعريف كنيد براي اينكار شرط reset يا preset درون شرط كلاك قرار مي گيرد مثال زير نمونه اي از DFF با reset سنكرون را نشان مي 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 19-2 تعریف یك reset سنكرون

لیست پروسس را تعریف كرده است كه فقط به تغییر حساس باشد ،حاصل سنتز یك فلیپ فلاپ نوع D است كه سنكرن با reset است و زمانیكه reset فعال باشد و لبة بالا رونده كلاك اتفاق بیافتد آنگاه reset عمل مي نماید.
شما همچنین مي توانید یك مدار سنكرون / آسنكرون با reset یا preset در VHDLتعریف كنید. مثال زیر یك رجیستر 8 بیتي را نشان مي دهد كه آسنكرون با reset و سنكرون با init مي باشد. اگر reset فعال شد تمامي بیتهاصفر مي شود و اگر init فعال بوده و لبه بالا رونده كلاك هم زده شود تمامي بیتها یك مي 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 وسنکرون با presetکردن

اپراتوري محاسباتي :

اپراتولهاي محاسباتي شامل : جمع ،تفریق، concatention علامت گذاري،ضرب ،تقسیم، modulus باقیمانده و قدر مطلق مقدار مي باشد. اغلب از اپراتورهاي محاسباتي جمع و تفریق استفاده میشود که در تعریف جمع کننده ها ،تفریق کننده ها ، افزایش دهنده ها و کاهش دهنده ها استفاده مي گردد. تمامي اپراتورهاي محاسباتي براي انواع integer , floationتعریف شده است. مثال ساده زیر یک جمع کننده چهار بیتي از نوع integer را نشان مي 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 صورتي کهoverflow آن لازم نباشد در کانترها مورد استفاده قرار مي گیرد:

```
Count <= count + 1 ;
```

2-3-2- reset و preset آسنکرون

لیست زیر یک کانتر 8 بیتي را تعریف مي کند که با یک سیگنال آسنکرون خروجي مقدار"00111010" را پیدا مي کند. کانتر همچنین یک enable و یک load سنکرون دارد . در این طراحي اپراتور + دو پکیج numeric_std موجود است. پورتهاي data , out 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 unsignded  ( 7 downto 0 ) ;
        Cnt            = buffer unsignded  ( 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  ;
```

لیست21-2 یک کانتر 8بیتي با یک سیگنال آسنکرون براي صفر و یک کردن فلیپ فلاپهاي این کانتر

لیست حساسیت پروسس شامل grst و txclk مي باشد . زمانیکه grst فعال شده باشد cnt به یک مقدار پیش فرض بصورت اسنکرون قرار مي گیرد. در لبه بالا رونده txclk اگر load فعال باشد در اینصورت مقدار data ورودي در cnt قرار مي گیرد و اگر load فعال نباشد و enable فعال باشد در این حالت cnt یک مقدار به آن اضافه مي شود .

ترکیب reset با preset :

گاهي اوقات در یک طراحي نیاز است که دو سیگنال reset و preset آسنکرون باشد. لیست زیر که براي همان کانتر 8 بیتي منظور ما را برآورده مي کند، لیست حساسیت شامل grst و gpst و txclk میباشد .

```
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 unsignded  ( 7 downto 0 ) ;
          Cnt              = buffer unsignded  ( 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 22-2 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 .

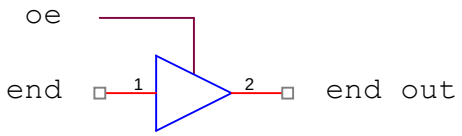
2-3-3- سيگنالهاي دو طرفه و بافرهاي سه حالتي :

مقاديري كه سيگنال سه حالتي پيدا ميكند شامل '0' , '1' ,'z' مي باشد كه همگي توسط نوع std_logic پشتيباني مي شوند. در كد زير كه باز همان مثال كانتر 8 بيتي است خروجي به عنوان يك سيگنال سه حالتي در نظر گرفته شده است. در اين دو مثال همچنين از پكيج std_qrith و std_logic_vector براي data استفاده كرده ايم .

```
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 unsignded ( 7 downto 0 ) ;
          Cnt_out          = buffer unsignded ( 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  ;
```

سيگنال oe كه در اين شكل استفاده شده است براي اين است كه خروجي را به حالت امپدانس بالا قرار دهد. بنابراين به عنوان كنترل three_state مي باشد. در اين برنامه از دو پروسس استفاده شده است و از آنجايي كه يك خروجي را در دو پروسس نمي توان تغيير داد به اين دليل يك سيگنال بنام cnt تعريف شده است. در پروسس اول مقدار cnt محاسبه مي شود و در پروسس دوم مقدار سيگنال cnt به خروجي cnt_out تعلق مي گيرد. شكل زير يك بافر سه حالتي رانشان مي دهد .



شكل 2-7: بافر سه حالته

بافرهاي سه حالتي همچنين مي توانند با دستور when – else نوشته شود. در مثال پايين نمونه اي از اين دستور استفاده شده است. در اين مثال يك سيگنال collision استفاده شده است و زماني كه سيگنال oe = 1 باشد برابر نتيجه and سيگنالهاي enabe , load مي 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 خروجي هاي سه حالته که با ساختار when_else تعریف شده اند.

دو طرفه ها :

باتغییر اندکي در کدهاي 30-2 و 31-2 براحتي مي توان سیگنال دوطرفه تعریف کرد. در لیست 32-2 یك سیگنال دوطرفه معرفي شده است زماني که لبه بالا روندة کلاك زده و Load فعال باشد این سیگنال به عنوان ورودي بوده و مقادیر آن در سیگنال قرار مي گیرد . و در پروسس دوم وقتي که مخالف صفر باشد سیگنال (که در اینجا به عنوان خروجي است) مقدار داده مي 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 هاي استفاده شده بصورت 2 طرفه

فعال کننده هاي بدون شرط خروجي :

در لیست 33-2 خروجي از یك بافر سه حالتي كه بصورت بي قید و شرط تعریف شده فعال گردیده است. اگر مقدار Present_state برابر col_address , res_assent , row_address یا Cas_assert باشد آنگاه بافرهاي سه حالته براي سیگنال dram فعال مي گردد. بافرهاي سه حالتي به ازاي سایر مقادیر present_state فعال نمي 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 ) thene
        dram <=  col_addr ;
    else
        dram <= ( others  => 'z' ) ;
  end if ;
end process ;
```

لیست 33 - 2 : کنترل فعال کننده خروجي Implicit

2-3-4- ساختار دو طرفه ها و سه حالته ها :

با دیگر ساختار ها شما مي توانید یك componet براي یك بافر سه حالته ایجادکنید. كدي كه در لیست 33-2 قرار گرفته براي مثال، بافرهاي سه حالته اي براي لاجیك output_enable ایجاد كرده است. اگر شما خواسته باشید Component فعال كنندة خروجي را صریحاً (واضح) تعریف كنید ، شما براحتي مي توانید بصورت زیر Component را تعریف كنید . (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 پياده سازي يك بافر سه حالتي براي يك باس 32 بيتي تعريف كنيد، شما بايد 32 Component جداگانه تعريف كنيد كه اين كار پرزحمتي است. دستور For_generate در اين حالت كمك مي كند.

```
gen_label :
 for I  in 0 to 31 generate
   inst_label : threestate port map ( vaalue(i) , read , value_out(i) ) ;
 end generate ;
```

اين دستور دو قسمت همزمان (concurrent) , يك architecture بدون يك پروسس پياده سازي مي شود. اين دستور به نام نياز دارد كه در اين جا، gen_label است. شما همچنين مي توانيد در اين دستور از عبارتهاي شرطي نيز استفاده كنيد .

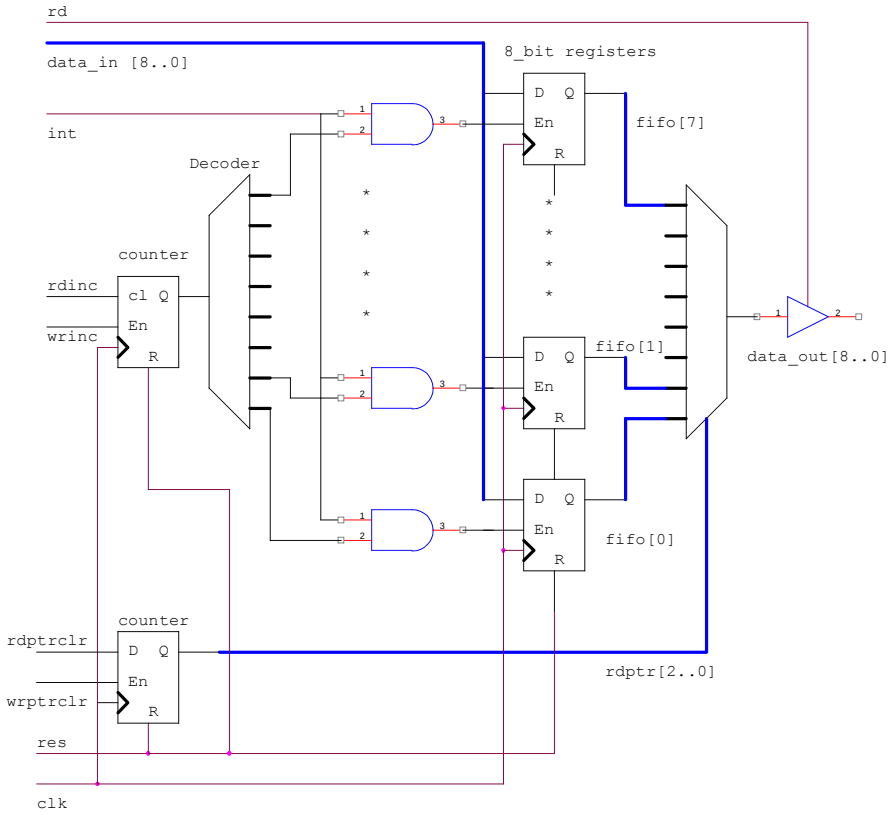
2-4- طراحي يك FIFO :

در اين قسمت مي خواهيم چگونگي ايجاد مدارات سنكرون و تركيبي را با چندين مثل ساده نشان دهيم. ما مي خواهيم كه يك FIFO 9 بيتي 8 تايي را طراحي كنيم. زمانيكه سيگنال خواندن (rd) فعال شد مي خواهيم كه خروجي data out از FIOF در زمانيكه فعال نيست، خروجي به حالت امپدانس بالا قرار گيرد. و زمانيكه سيگنال نوشتن (wr) فعال شد مي خواهيم كه يك رجيستر با پهناي 9 بيتي نوشته شود. سيگنالهاي wrinc , rdinc جهت افزايش آدرس رجيسترهاي خواندن و نوشتن استفاده مي شود.

Rdptrelr و Wrptrclr براي reset كردن آدرس روي اولين رجيستر FIFO قرار مي گيرد . از Data_in براي laod كردن data در رجيسترها استفاده مي شود . شكل 2-8- بلوك دياگرام FIFO را نشان مي دهد.

ليست 1- 33-2 كد VHDL براي اين طراحي را نشان مي دهد. اين نحوه بيان ساده بوده و از مفهوم هاي جديد استفاده نشده است. شامل 8 رجيستر با طول 9 بيت و از نوع Std_logic_vector است. سيگنال FIFO براي اين نوع بيان شده است. (يك array تك بعدي از std_logic_vector) ما مي توانيم std_logic_vector ها را با مشخص كردن index نظير fifo(1) , fifo(2) , fifo(3) دسترسي داشته باشيم .

مفهوم جديد ديگري كه مي توانيم استفاده كنيم loop است كه در ادامه اين فصل ديده مي شود



شكل 2-8: بلوك دياگرام FIFO

looP -5-2 ها

در جاهايي كه تكرار نياز باشد از دستورات Loop استفاده مي شود . مثال for Loop ,

While_Loop ها مي باشند .

دستور For براي تعداد مشخص از حلقه ها اجرا مي گردد. دستور white اجراي يك عملكرد را ادامه مي دهد تا زماني كه شرط صحيح باشد. يك مرحله براي مقداردهي اوليه به متغيرهاي يك دستور while نياز است. براي نمونه، reset آسنكرون array FIFO با استفاده از Loop بصورت زير است :

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

دستور while_loop بجاي For Loop مي تواند استفاده شود، ولي بايد اولاً به متغير مقدار اوليه تعلق گيرد و در Loop بايد متغير را اضافه كرد .

```
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 Iteratiors

دستور next جهت نگهداري يك عمل روي شرط خاصي استفاده مي شود . براي مثال فرض كنيد زمانيكه rst فعال شد بايد همة رجيسترهاي Fifo بجز رجيستر Fifo (4) ريست مي 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 ;
------
```

يا با يك دستور while Loop بصورت زير نوشته مي 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 هايي كه گفته شد ما متغير ها را درون process قرار مي داديم. دستور exit براي خارج شدن از loop استفاده مي شود و مي توان جهت چك كردن يك شرط غير مجاز استفاده شود. شرط بايد در compile_time تعين شود. براي مثال فرض كنيد كه fifo يك component ي است كه در يك طراحي hierarchical معرفي شده است. درضمن فرض كنيد كه عمق fifo توسط يك generic يا پارامتر مشخص شده است شما ممكن است بخواهيد زماني كه عمق fifo بزرگتر از مقدار از پيش تعين شده باشد از loop خارج شود . براي 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 ;
```

اين كد همچنين بصورت زير مي 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 ;
```

اکنون با استفاده از مفاهم جدید برنامة Fifo را دوباره بازنویسي مي 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 :
طراحي State machine

3-1- مقدمه :

در فصول گذشته بلوكهاي اساسي و ساختارهاي زبان در VHDL را بيان كرديم. در اين فصل، ما همان مفهوم‌ها را بصورت طراحي State machine را بررسي مي‌كنيم. روش State machine را انتخاب كرديم براي اينكه اين روش معمولا در پياده سازي آي‌سي‌هاي قابل برنامه ريزي استفاده مي‌شود. ما چگونگي تعريف و سنتز State machine را با هدف طراحي بصورت بهينه شدن سرعت و سطح بيان مي‌كنيم. در اين فصل مثال‌هاي ساده‌اي بيان

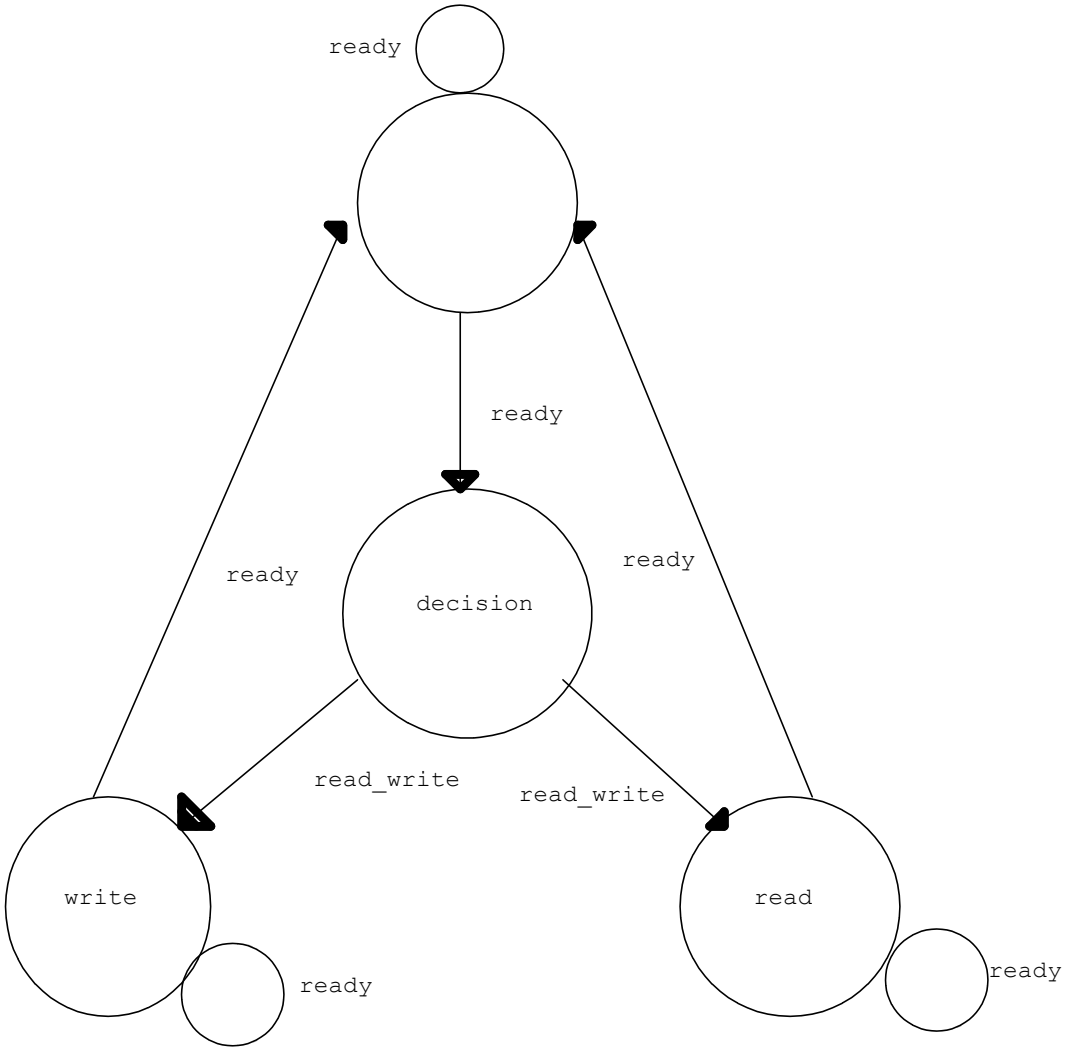
شده تا بتوان رفتار State machine در VHDL با دستورات case-when و if-then-else را نشان دهیم .

2-3 يك مثال ساده طراحي :

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

3-2-1- روش طراحي 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 كه حالتهاي بعدي و خروجي را مشخص كنيم. از تعريف مسئله و مشخص كردن حالتهاي مختلف شكل 3-1 حاصل مي گردد.



شكل 3-1: State machine ساده

اين دياگرام نشان مي‌دهد كه يك انتقال خولندن يا نوشتن با فعال شدن ready آغاز مي‌گردد كه حالت State machine از idle به decision تغيير پيدا مي‌كند. با توجه به مقدار read-write در كلاك بعدي ، انتقال خواندن يا نوشتن فعال شده و State machine به حالت مربوطه هدايت مي‌شود. يك انتقال زماني پايان مي‌يابد كه ready فعال شده باشد و كنترلر در حالت idle قرار بگيرد. زمانيكه ready فعال نمي‌شود كنترلر در حالت خود باقي مي‌ماند. حالتهاي برابر در اين دستگاه وجود ندارد.

3-2-2- State machine در VHDL

دیاگرام حالتي كه نشان داده شده براحتي مي‌تواند با يك بيان سطح بالاي VHDL بدون داشتن شكل فعال شدن حالتها، جدول توليد حالتهاي گذرا يا مشخص بودن حالتهاي بعدي بر مبناي نوع فليپفلاپهاي موجود، تعريف شود. در VHDL هر حالتي مي‌تواند به كمك يك ساختار case-when تعريف شود. تغيير حالتها مي‌تواند با دستورات if-else-then مشخص شود. براي مثال، ما يك typeي تعريف كرده‌ايم كه شامل اسامي حالتهاست و دو سيگنال از اين type معرفي كرده ايم :

```
type statetype is (idle , decision , read , write);
signal present_state , next_state : statetype;
```

سپس ما يك پروسس ايجاد مي‌كنيم . next_state توسط يك تابعي از present_state و ورودیهاي ready و read-write تعريف شده است. بنابراين ليست حساسيت شامل اين سيگنالها است:

```
state_comb : process (present_state , read_write , ready)
        begin
                ……
        end process state_comb;
```

در پروسس حالتهاي مختلف را بيان كرده‌ايم . ما يك ساختار case-when ايجاد كرده‌ايم و اولين حالت را idle_state قرار داده‌ايم. براي اين حالت خروجيهايي كه با idle_state فعال مي‌شوند را مشخص مي‌كنيم :

```
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 اين حالت دو راه وجود دارد (البته زمانيكه present_state ، idle است) :

1- اگر ready اتفاق افتد به decision تغيير حالت مي‌دهد .

2- باقي ماندن در حالت idle .

حالتهاي گذرا در اين برنامه بايد كامل قرار گيرند : شاخه‌اي از بيان case براي هر حالتي (when state_name =>) ، تعيين كردن خروجيهاي آن حالت و تعيين حالتهاي گذرا با بيان if-then-else .

در زير، برنامه كامل از حالتهاي گذرا و خروجيها 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 بعدي (next_state) با توجه به حالت جدید و ورودیها تعیین مي‌شود. این رخداد سنکرون با لبه بالا رونده کلاك است که باید بصورت یك پروسس دیگر تعریف شود. شبیه آنچه که در زیر آمده است براي 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.

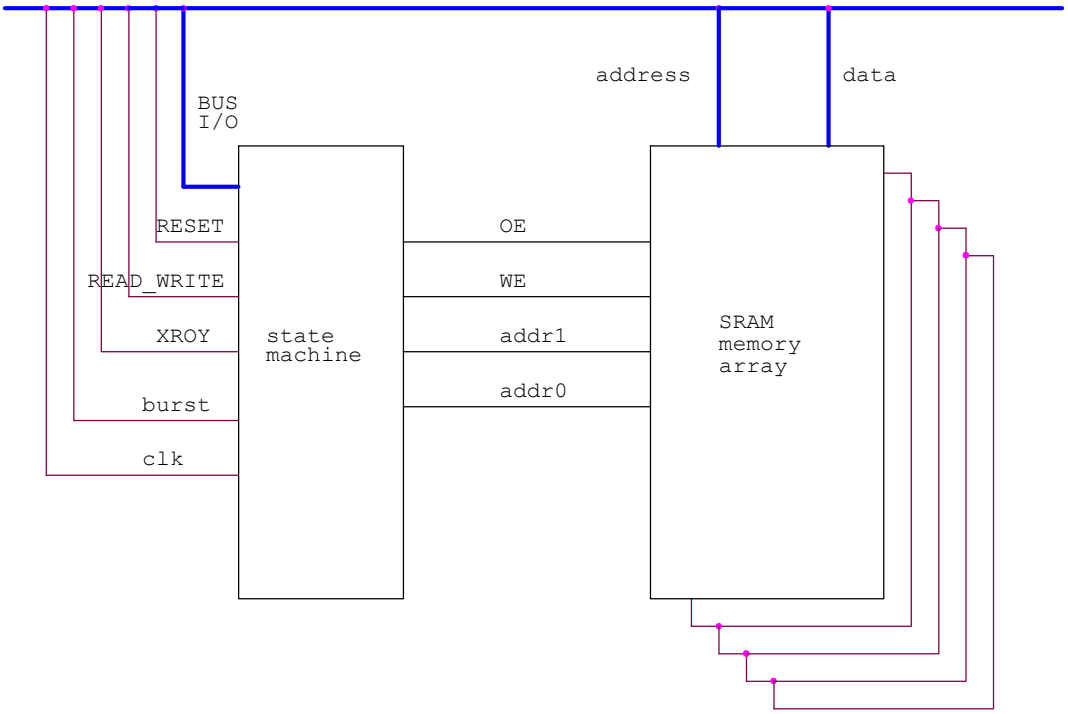
```
Entity example is port(
  Read_write , ready , clk : in bit;
  Oe , we                  : out bit );
End example;
Architecture state_machine of example is
   Type statetype is (idle , decision , read , write);
    Signal present_state , next_state : state_type;
Begin
State_comb : process(present_state , 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 : طراحي يك كنترلر حافظه ساده

3-3- يك كنترلر حافظه

مثال طراحي بعدي باز يك كنترلر حافظه است، اما آن عملي‌تر و جزئيات بيشتري نسبت به مثال قبلي دارد. شكل 3-2 يك بلوك دياگرام از سيستمي كه يك State machine براي يك كنترلر حافظه استفاده كرده را نشان مي‌دهد.



شکل3-2 : بلوك دياگرام كنترلر حافظه

سيستم به اين صورت كار مي‌كند: ابزارهاي ديگري روي باس قرار گرفته‌اند كه با شناسايي مقدار F3(hex) دريافتي به منظور بافر كردن حافظه استفاده مي‌شوند. يك سيكل بعد، سيگنال read_write كه نشاندهندة اين است كه يك انتقال نوشتن روي بافر حافظه قرار گرفته است.

اگر حافظه انتقال خواندن دريافت كند، خواندن ممكن است يا بصورت خواندن از نوع single باشد يا از نوع burst (انفجاري). يك خواندن برست با فعال شدن burst در طول اولين كلاك مشخص مي‌شود، كه باعث مي‌شود كنترلر به چهار قسمت از بافر دسترسي داشته باشد. محلهاي متوالي با فعال شدن متوالي ready دسترسي مي‌شوند. كنترلر فعال كنندة خروجي oe را بعد از يك خواندن غير فعال كرده و دو بيت آدرس را يكي كاهش مي‌دهد.

انتقال نوشتن در بافر هميشه بصورت single صورت مي‌گيرد نه بصورت burst. در طول انتقال نوشتن با مشخص كردن آدرس data، در حافظه نوشته مي‌شود. عمل خواندن و نوشتن با فعال شدن ready تمام مي‌شود.

3-3-1- تبدیل دیاگرام حالت به VHDL

دیاگرام حالت بسادگي مي‌تواند با يك سري از حالتهايي در ساختار case – when بصورت زير تعريف شود (در اينجا reset سنكرون ناديده 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 حالتي، خروجيهاي machine state با دستورات ترتيبي تعريف مي شوند، و همة حالتهاي 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 حالتي كه فعال است به حالت idle تغيير حالت دهد. ما مي‌توانيم با بيان يك دستور if-then-else در شروع پروسس بدين 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. نحوة 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
```

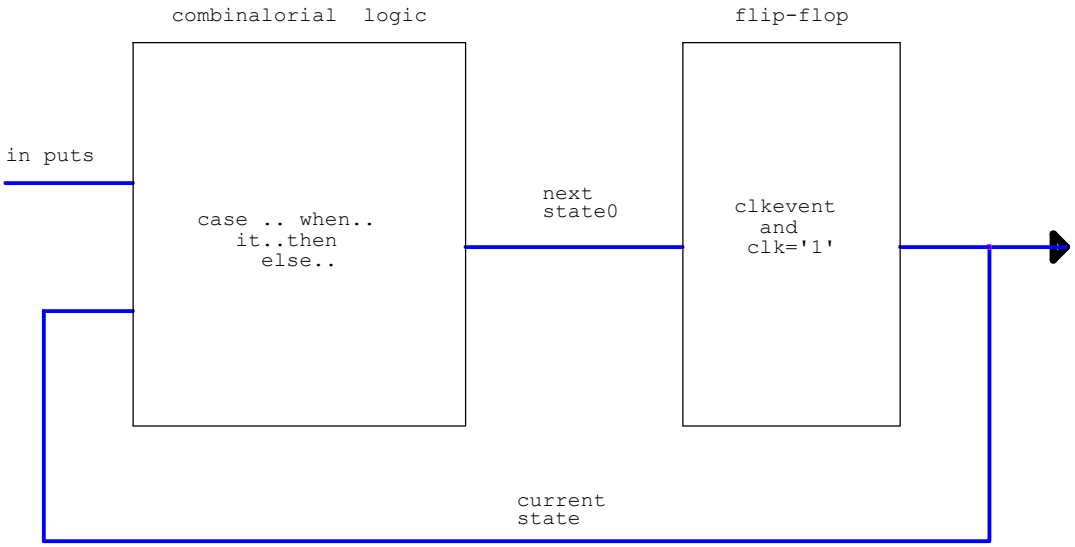
```
        case present_state is
           when idle =>          oe<='0'; we<='0' ; addr<="00";
        if (bus_id = '11110011") then
            next_state <= decision;
        else
            next_state <= idle;
        end if;
        when decision =>       oe<='0'; we<='0' ; addr<="00";
        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;
```

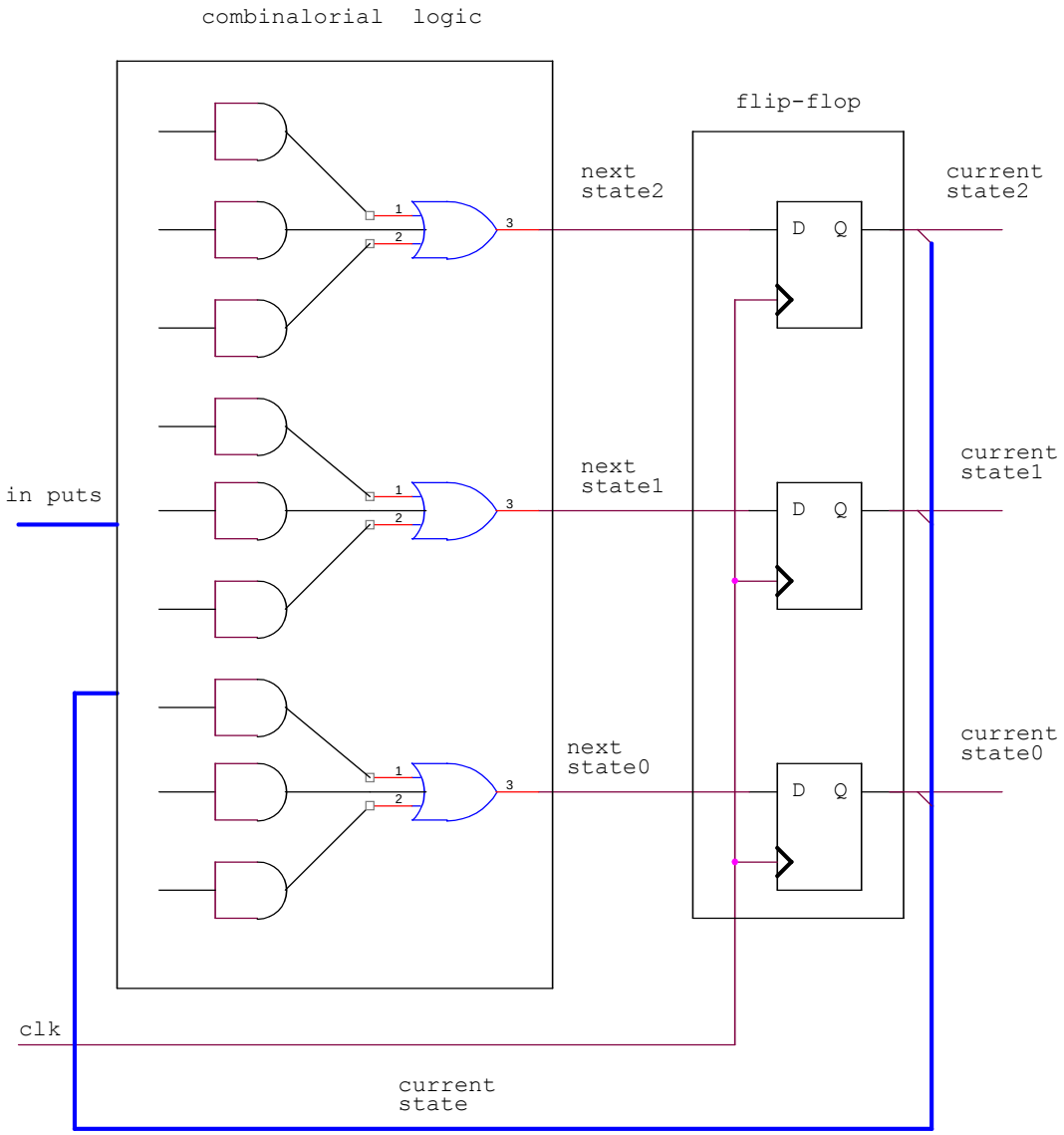
لیست 2-3 : تعریف کنترلر حافظه بصورت FSM دو پروسس

لیست 2-3 یك بیان FSM دو پروسس است. یك پروسس لاجیکهاي ترکیبي و تغییر حالتها با کلاك را نشان میدهد. این ساختار کد قابل قیاس با شکل 3-3 میباشد که در یك ساختار CPLD پیاده شده است.

کدهاي present_state و ورودیها در قسمت ترکیبي لاجیك بلوکها بصورت آنچه در شکلها تعریف شده اجرا میشود. حالت next_stat سنکرون در پروسس state_clocked تعریف شده که بانکي از رجیسترها نظیر ماکروسلها در یك بلوك لاجیك را بیان میکند.



شکل3-3-الف : ساختار کد لیست 2



شكل 3-3-ب : پيـاده سازي State machine در يـك PLD

reset كـردن آسنكرون در يـك FSM دو پـروسس :

اگـر يـك reset آسـنكرون بجاي reset سنكرون بـكـار رود در كـد پـروسس state_clocked بـايـد بـصورت لـيست زيـر نـوشته 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 براي فعال سازي سیستم یا زمانیکه مدار hang مي‌کند استفاده شود، در اینصورت استفاده کردن reset آسنکرون ممکن است بهتر از یك reset سنکرون باشد .

3-3-2 یك ساختار کد دیگر

FSM یك پروسس: کدي که در لیست 3 آمده است از لحاظ معادلات انجام شده و نتایج خروجي معادل کدي است که در لیست 2 آمده است البته اگر reset آن بصورت آسنکرون استفاده شود. لیست 3-3 براي تعریف حالتهاي گذرا و سنکرون سازي این حالتها با کلاك فقط از یك پروسس استفاده کرده است بنابراین ما آنرا FSM یك 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 یك كنترلر حافظه بصورت یك FSM تك پروسس

در این بیان طراحي، حالتهاي گذراي پروسس state_tr براي لاجیك next_state و براي كلاك زدن state register ها استفاده شده است . فقط یك سیگنال state از نوع state type مورد نیاز است. بیاني كه در زیر آمده، مقدار دهي همة سیگنالهاي ترتیبي در پروسس روي لبة بالا روندة كلاك اتفاق مي‌افتد:

```
        elsif rising_edge (clk) then
```

در مجموعه زیر مقدار دهي state بصورت سنكرون صورت گرفته است:

```
        state_tr : process(reset , clk) begin
          …….
          Elsif  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 فعلي مقدار idle و باس مقدار F3(hex) را داشته باشد، آنگاه در لبة كلاك بعدي decision يك مقدار جديد پيدا مي‌كند. قطعة كد بالا مي‌تواند بصورت زير تغيير كند؛ زمانيكه حالت فعلي idle است، اگر bus_id مقدار 11110011 را داشته باشد، آنگاه حالت جديد decision است. در ساير حالات كه با else مشخص شده state <= idle خواهد شد. سيگنالهاي خروجي كه در ليست3-3 نشان داده شده با دستورات همزمان كه مدارات آن قابل سنتز در مدارات تركيبي است مقدار دهي شده اند.

فصل 4 :

توابع و رویه‌ها

4-1- توابع (function) :

b12bit (بولين به بيت) . ليست 1-4 يك نوع تابع تبديل را انجام مي‌دهد .

```
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;
```

ليست 1-4 : يك نوع تابع تبديل بولين به بيت

ليست 1-4 يك تابع كد براي تبديل يك سيگنال از نوع boolean به نوع bit استفاده شود را تعريف مي‌كند. boolean و bit هر دو در استاندارد IEEE 1076 تعريف شده اند. خط 1 تابع b12bit را كه پارامتر ورودي است و بصورت نوع boolean تعريف شده و بايد به نوع bit برگردانده شود را معرفي مي‌كند. خط 2 و 8 ابتدا و انتهاي تابع را مشخص مي‌كند.

تمامي دستوراتي كه براي تعريف تابع بكار مي‌روند بايد از نوع دستورات ترتيبي باشند. خطوط 3 تا 7 دستورات ترتيبي هستند و سيگنال a را كه مقدار آن يك نوع boolean است، برمي‌گرداند. اگر a درست باشد مقدار 1 و اگر غلط باشد مقدار 0 را خواهد داشت. ديگر توابع تبديلي كه اغلب استفاده مي‌شود، bit به boolean ، bit به std_logic و bit_vector به std_logic_vector هستند.

از يك تابع تبديل نوع bit به boolean يا boolean به bit مي‌توان در نوشتن معادلات بولين يا ارزيابي شروط مورد استفاده قرار داد. براي مثال اگر سيگنال clk از نوع boolean باشد شما مي‌توانيد اينگونه بنويسيد:

```
wait until clk ;
rather than
```

wait until clk='1'

يا

if (clk'evevnt and clk) then ….

Rather than

If (clk' event and clk='1') then …..

همينطور

if ((A and B) xor (C and D)) then ……

مي‌تواند جايگذين شود بجاي

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

يك راه لزوما بهتر از ديگري نيست. براي طراحان نوشتن مدهاي VHDL براي سنتز، نوع std_logic ترجيح داده مي‌شود.

پارامترهاي تابع :

پارامترهاي تابع فقط مي‌تواند ورودي باشند. بنابراين پارامترها نمي‌توانند بهينه شوند. پارامتر a در ليست 1 فقط يك ورودي است. با فرض اينكه تمامي پارامترها از مد inهستند، بنابراين نيازي نيست كه مد آنها صريحا نوشته شوند. توابع فقط مي‌توانند به يك آرگومان برگردند. (درحاليكه procedure ها مي‌توانند به چندين آرگومان برگردند).

در مجموع سيگنال جديدي در تابع function نمي‌تواند تعريف شود، ولي متغير variable ممكن است در تابع تعريف شود و مقاديري به آن تعلق گيرد.

4-2- نوع توابع تبديل :

bv2I (bit_vector به integer) . ليست 2-4 را بخوانيد و چگونگي كار توابع را دريابيد.

```
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 توضیحاتي هستند که نشاندهندة نام تابع، تعریف نوع تابع تبدیل، و مشخص کنندة پارامتر ورودي و نوعي که برمي‌گرداند، مي‌باشد. به این تابع یك ورودي bit_vector داده مي‌شود و یك تبدیل بایناري به دسیمال را اعمال کرده و یك مقدار integer بر مي گرداند. خط 6 یك پارامتر ورودي دارد که پهناي bit_vector آن محدود نشده است. اما پهناي بردار در فراخواني تابع در زمان کامپایل باید مشخص شود. خطوط 7 و 8 تعریف variableهایي هستند که شبیه به پروسس در حاشیه آن ساخته مي‌شوند. در این تابع bv2I ، سه variableبصورت integer تعریف شده که با صفر مقدار دهي شده‌اند.

تابع نیز با یك begin شروع و با یك end پایان مي یابد که در این برنامه خطوط 9 و 20 مي باشند. خط 10 ابتداي یك loop را مشخص مي‌کند که با کمترین مقدار bit_vector bv شروع مي‌شود. صفتهاي low و high در VHDL معرفي شده که در اینجا براي برگرداندن کمترین و بزرگترین bit_vectorي که به تابع بصورت یك پارامتر داده مي شود ، استفاده شده است. بنابراین بدون توجه به عامل bit_vector_bv – (x downto y) یا y – (y to x) کم ارزشترین بیت شناخته شده و مقدار integer که براي bit_vector ساخته مي شود، MSB x , LSB y مي باشد.

براي مثال، دو bit_vector ، a و b ممكن است بصورت زير تعريف گردد:

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

در هر كدام از اين bit_vector ها ، a(6) و b(6) به عنوان LSB ملاحظه مي گردد. تابع بايد بصورتي نوشته شود كه هميشه مقدار سمت چپ به عنوان MSB مشخص شود. bit_vector loop از LSB به MSB با تغييري كه به عنوان bit_vector index استفاده شده، افزايش يافته است. abit با توجه به وضعيت bit_vector تواني به آن اختصاص داده مي‌شود.
ملاحظه مي كنيد a(13 downto 6) bit_vector : اگر a(8) ’1’ باشد ، آنگاه abit مقدار integer 4 را پيدا مي كند. براي اينكه I 8 است ، low’ 6 bv است ، low’ I-bv 2 و 2**2 4 مي باشد.
اين مقدار بايناري 100 را ارائه مي كند، جمع result براي هر تكراري از loop مقدار abit است. count كه استفاده شده مشخص كنندة پهناي bit_vector است كه به يك integer تبديل شده است. رنج integer كه در VHDL استفاده مي‌شود به 2^32 محدود شده است. زمانيكه loop پايان مي‌يابد يا خارج مي‌گردد، result برگردانده مي‌شود (خط 19) ، و تبديل bit_vector به integer پايان مي يابد.

12bv (integer به bit_vector) :

تابع i2bv تبديل integer به bit_vector را انجام مي‌دهد. ليست 3-4 را بخوانيد و چگونگي انجام آنرا درك كنيد:

```
-- i2bv
--integer to Bit-vector.
--In:      Integer,value and width.
--Return: bit- vector, with right Bit the most significant.
--
functon i12bv   (val , widht : 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 ورودي يكي مقدار integer و ديگري اندازه يا پهناي bit_vector كه بايد انجام شود را مي گيرد. تابع يك مبدل دسيمال به بايناري است و مقدار integer را به bit_vector برمي‌گرداند.

در تابع تعريف شده، result بصورت يك متغير از نوع bit_vector تعريف شده كه اندازة آن با مقدار width مشخص شده است. عرض width به عنوان يك متغير در اين تابع مي‌باشد و آن بايد از مد in بوده و نمي‌تواند بهينه شود. اندازه bit_vector به 32 بيت محدود شده است.

3-4- توابعي كه در ساده سازي component ها استفاده مي‌شوند :

توابع گاهي اوقات در قسمتهايي از component هاي جاري استفاده مي‌شود. براي اينكه آنها يك روش براي اختصار نويسي فراهم مي‌كنند. توابع براي جايگذاري component با يك خروجي محدود شده است. آنها دستورات wait ندارند و مي توانند فقط از دستورات ترتيبي استفاده كنند.

inv_bv (افزايش bit_vector) :

تمرين ليست 4-4 چگونگي انجام يك تابع افزايشي را نشان مي دهد .

```
-- inc_bv
--increment Bit-vector.
--In :      bit_vector.
--Return: bit- vector.
--
functon 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;
```

ليست 4-4 : يك تابع براي افزايش bit_vector

تابع inv_bv يك ورودي a از نوع bit_vector را گرفته و مقدار آنرا افزايش مي‌دهد، و يك bit_vector به همان اندازة ورودي برمي‌گرداند. صفت rang از قبل در VHDL تعريف شده و رنج يك array را برمي‌گرداند. متغير S بصورت يك bit_vector با رنج مساوي – (x downto y) يا (y to x) – شبيه به بردار ورودي a فعال شده است. carry بصورت يك بيت مشخص شده است.

اكثريت :

تابع بعدي يك بيتي را برمي‌گرداند كه اكثريت سيگنالهاي ورودي تابع را مشخص مي‌كند .

```
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 5-4 : تابع اکثریت براي سه ورودي

سپس چگونگي بکارگيري اين توابع را بيان مي‌کند .

4-4- بيان توابع نامعلوم :

بايد توجه داشت که برخي از توابع نامعلوم ممکن است در برخي از سنتزکننده ها نتوانند راه اندازي شوند. ليست 6-4 يک تابع اکثريت براي يک bit_vector با يک پهناي مشخص تعريف شده که پهناي bit_vector در فراخواني تابع مشخص مي گردد. اين تابع از يک loop استفاده کرده که به تعداد يک ها در bit_vector تکرار مي‌گردد.
يک کانتر متغير جهت نگهداري رديابي تعداد يکها استفاده شده است. سپس مقدار اين متغير با مقدار نصف bit_vector مقايسه مي‌گردد. اگر مقدار متغير کانتر بزرگتر از اين مقدار باشد، آنگاه تابع عدد ’1‘ و در غير اينصورت عدد ’0‘ را برمي‌گرداند.

```
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 6-4 : تابع اکثریت براي یك bit_vector با پهناي n_bits

4-5- استفاده کردن توابع :

یك تابع ممکن است در حاشیه یك architecture تعریف شود، در این حالت تابع بصورت یك تابع تعریف شده معني پیدا مي کند. شما ممکن است بخواهید یك مجموعه اي از انواع و توابع جانشیني componentها و مبدلها ایجاد کنید، و آنها رادر یك package قرار دهید و آن package را در یك library کامپایل کنید و بنابراین شما مي‌توانید براحتي آنها را در یك طراحي بکار گیرید.

یك تابعي که در یك architecture تعریف گردیده فقط توسط همان architecture مي‌تواند استفاده گردد. یك تابع تعریف شده در یك package مي‌تواند با یك شرطي در دیگر طراحي‌ها استفاده گردد .

در اینجا، اولین تابع اکثریت که در لیست 5 آمده بود را مي خواهیم براي طراحي یك full adder که در لیست 7-4 آمده استفاده کنیم. ما این تابع را در محاسبة carry_out بکار مي‌گیریم. تعریف تابع در ابتداي architecture صورت گرفته و همچنین بصورت یك تابع تعریف شده بکار مي‌رود.

```
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 7-4 : طراحي يك full adder با استفاده از تابع اكثريت

بصورتي ديگر، تابع مي‌تواند در يك تعريف package ، بيان شود و در بدنة package اختصاص يابد. در اين حالت ما تابع majority را بخوبي توابع inc_bv و 12vb در package بيان مي‌كنيم.

```
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 8-4 : packageي که شامل چهار نوع تابع تبدیل است

تعریف package فقط با تعریف تابع مي‌باشد. این تعریف تابع یك واسط template براي طراحي‌هایي است که این تابع را صدا مي‌کنند . تعریف تابع در بدنة 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 یك package

4-6- اپراتورهاي فراخواني

يكي از توانايي‌هاي استفاده تابع در اپراتورهاي فراخواني مي‌باشد. برخي از توابع فراخواني همچنين در استانداردهاي IEEE1164 و 1076.3 معرفي شده‌اند. ما مي‌خواهيم چگونگي تعريف اپراتورهاي فراخواني را بيان كنيم.

اپراتور + توسط استاندارد IEEE1076 تعريف شده كه روي انواع عددي (انواع integer ، floating point ، physical) عمل مي‌كند، اما با انواع غير عددي نظير std_logic يا bit_vector عمل نمي‌كند. در عمل جمع يك integer ثابت با يك سيگنال از نوع std_logic ، يك اپراتور فراخواني نياز است. ليست 4-10 برنامه جمع كردن يك integer با يك bit_vector را نشان مي‌دهد. موارد استفاده ديگر اين اپراتورها نظير جمع كردن يك bit_vector با يك integer ، يك bit_vector با bit ، يك std_logic_vector با يك integer ، يك std_logic_vector با يك std_logic مي‌باشد.

```
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 : یك شمارنده با عملگر + كه عملوندهايي از نوع bit_vector و integer دارد .

VHDL اصلي اين جمع را انجام نمي دهد براي اينكه عملوندها از نوع bit_vector و integer است. اپراتورهاي فراخواني بايد همراه package خودشان معرفي گردند.
شما مي‌توانيد چنديد تابع كه عملكردي مساوي براي انواع typeها داشته باشند ايجاد كنيد. ابزارهاي سنتز و شبيه ساز VHDL براي هماهنگي كامل احتياج به تغيير فرم دارند.
ليست 4-11 شامل يك package تعريف شده و بدنة package است كه دو اپراتور فراخواني براي عمل + را تعريف و مشخص كرده 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 11-4 : تعریف اپراتورهاي توابع فراخواني شده

این پکیج همچنین استفاده clause در ساختن my_package موجود در این واحد طراحي را نیز شامل است. تابع i2bv براي دومین تابع + لازم است . که این تا مقادیر integer را به bit_vector تبدیل مي‌نماید. سپس در مقدار bit_vector با هم جمع شده‌اند . اولین اپراتور فراخواني + ممکن است توسط این طراحي استفاده شود براي اینکه همة توضیحات یك package ضمنا در بدنه package موجود هستند.
خط زیر کدي از اولین تابع فراخواني مي‌باشد که جهت بدست آوردن بیت با کمترین ارزش در یك بردار، استفاده شده است.

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

زمانیکه آن بصورت مثال کانتر لیست 10-4 استفاده شده باشد، کامپایلر باید براي تابع جمع با هماهنگ بودن نوع operand هاي براي جمله ;count <= count+1 جستجو نماید.
لیست 10-4 مي‌تواند بصورتي 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 فراخواني :

اپراتورها محدود به فراخواني توابع مربوطه نیستند بلکه شما مي توانید هر تابعي را فراخواني کنید. براي مثال، توابعي است که در package majority در لیست 12-4 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 12-4 : فراخواني تابع majority براي اعداد متغير و انواع ورودیها

شش تابع هم نام تعريف شده است. هر تابع بهر حال عملكرد majority براي يك تعداد مختلفي از ورودیها يا typeها را تعريف ميكند. زمانيكه توابع majority در ليست 13-4 استفاده شده اند، كامپايلر بايد توابع مختص براي صدا كردن 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 13-4 : استفادة توابع فراخواني شدة majority

لیست 13-4 اثبات میکند که تابع میتواند بصورت یك alternative در typeهاي مشخص component جاري استفاده شود. این لیست همچنین بیان میکند که تابع majoirity میتواند در پذیرفتن مقادیر متغیر و انواع پارامترهاي ورودي و انواع متغیر مقادیر برگشتي فراخواني شود.

4-8- توابع استاندارد :

خوشبختانه، توابع استاندارد ساخته شده اند بنابراین اولین ورژن مرجع زبان VHDL (استاندارد IEEE1076) در سال 1987 ارائه شد. بستههاي استاندارد که شامل اپراتور فراخواني است براي چندین نوع تعریف شدهاند. کد VHDL با استفاده از پکیجهاي استاندارد ساخته شده است بسیار سادهتر و کم حجمتر از یك ابزار دیگر است. براي مثال بسته std_logic_1164 یك سیستم نوع استاندارد data تهیه میکند چون که توسط چندین فروشندة ابزار پشتیباني شده است. تواناییهاي شما با استفاده از انواع data در این بسته توسط clause و library following مشخص میشود :

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

براي هر ابزار سنتز يا شبيه ساز كه از اين package پشتيباني مي‌كند شما قادر خواهيد بود كه كدهاي برنامه را بدون هيچ بهينه سازي استفاده كنيد. اگر ابزاري كه جهت دستيابي به اپراتورها و انواع داده‌ها نياز به يك پكيج اختصاصي داشته باشد در اينصورت شما به آساني نمي‌توانيد كه كدهاي خود را از يك سيستم به يك سيستم ديگر انتقال دهيد.

استاندارد IEEE_1164 ، فقط براي تعريف انواع داده‌هاي يك سيستم نيست بلكه اپراتورهاي فراخواني و توابع تبديل را نيز شامل است. اين استاندارد همچنين شامل زير نوع X01 و X01Z نيز مي‌باشد. در ضمن اپراتورهاي لاجيكي (and, or و مانند اينها) فراخواني كرد و بعضي از توابع تبديل معمول را نيز فراهم كرده 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-stdulogicvector , to-stdlogicvector , to-bitvector همگي را مي‌توان براي انواع پارامترهاي ورودي فراخواني كرد. استاندارد مهم ديگر براي سنتز كردن IEEE1076.3 است. اين استاندارد مشخص مي‌كند " دو package كه typeهاي بردار را تعريف كرده است براي نشان دادن مقادير محاسباتي علامت دار و بدون علامت، shift و انواع تبديل عملكردها روي اين انواع. پكيج‌هاي numeric_std , numeric_bit هستند كه ما آنها را شرح داده‌ايم. يك

مثال از اپراتورهاي محاسباتي كه اين استاندارد تعريف كرده در ليست 4-14 داده شده است.

بيانات زير از numeric_bit package مي‌باشند:

```
type unsigned is array (natural range <>) of bit;
type signed is array (natural range <>) 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 : nutural) 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 :nutural; 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 15-4 : برخي اپراتورهاي فراخواني در numeric_std

تابع مهم ديگري كه در numeric_std تعريف شده ، std_match مي‌باشد . تابع براي چندين type فراخواني 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,   t rue,  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 16-4 : فراخوانی کردن دو تابع std_match