# باسمه تعالی

دانشکده فنی و حرفه ای دختران تهران
دکتر شریعتی

## جزوه درس طراحی خودکار سیستم های دیجیتال

مدرس: رضا سعیدی نیا

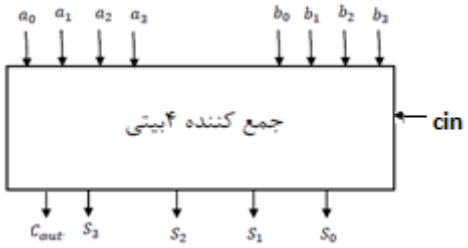
## فصل اول مقدمه

در این درس هدف ارائه روش های طراحی خودکار سیستم های دیجیتال می‌باشد. برای طراحی سخت افزار محیط های نرم افزاری مثل Modelsim و MaxPlus II ایجاد شده است. طراح قبل از ارائه طرح نهایی ابتدا طرح را در این محیط ها آماده می کند و مرحله به مرحله طرح خود را تست و خطایابی می‌کند در صورت عدم وجود خطا به مراحل پیشرفته‌تر گام می-گذارد. در انتها طرح نهایی را به کارخانه سازنده آی سی ارائه می کند. در این درس ما از زبان برنامه نویسی توصیف سخت افزار[1] (HDL) برای انجام این مراحل استفاده می کنیم. مهمترین زبان های توصیف سخت افزار VHDL و Verilog می باشد که ما زبان Verilog را ارائه می دهیم. برای طراحی، شبیه سازی و مدل سازی از نرم افزار modelsim استفاده می‌کنیم.

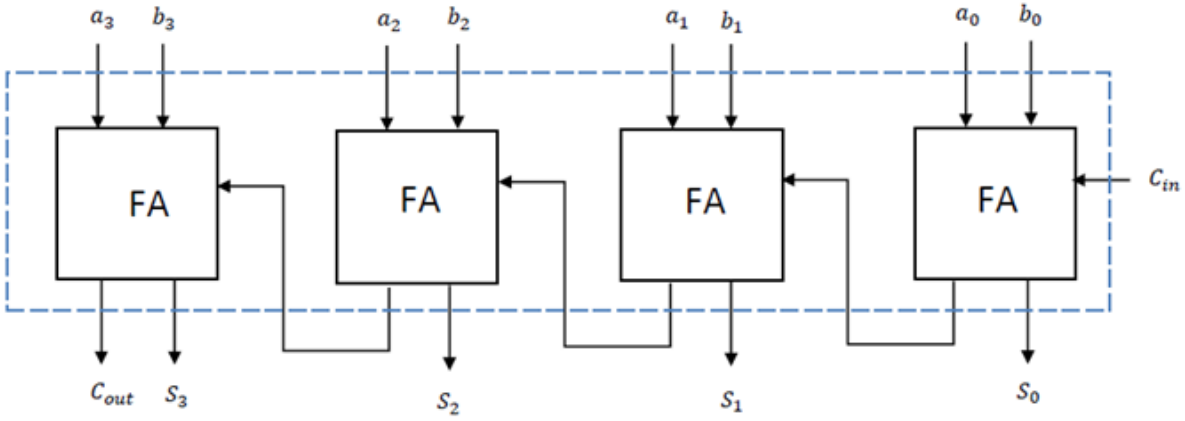
برای طراحی یک سخت افزار سطوح مختلفی وجود دارد که آن را با یک مثال ساده ارائه می‌کنیم.

**سطوح سخت افزار**: مثال) سخت افزاری طراحی کنید که ۲ عدد ۴ بیتی را با هم جمع کند؟

۱- سطح رفتاری[2]: در این مرحله عملکرد مدار مشخص می‌شود. یعنی S=a+b.



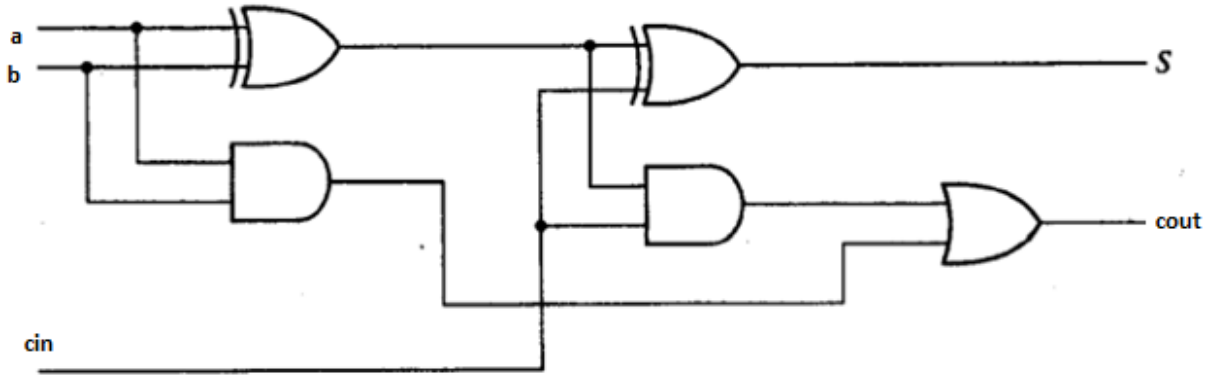

۲- سطح مسیر داده[3]: در این مرحله مسیر داده بین قطعات مشخص می‌شود. مسیر داده بیشتر بین ثبات ها، گذرگاه ها و حافظه مطرح است. اما به خاطر تبیین اولیه مساله، مسیر تبادل بین بخش های مختلف را مد نظر قرار می دهیم.



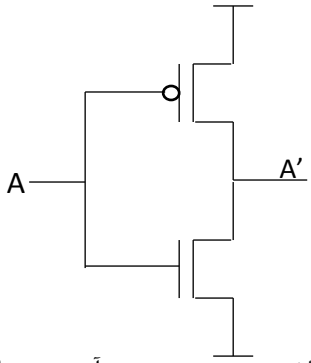

۳- سطح ساختاری[4]: در این سطح قطعات مختلف مدار طراحی می شوند. در این سطح یا از قطعات موجود استفاده می‌شود یا هر قطعه به مرحله گیت های منطقی می رسد. بنابراین شکل قبلی خود در سطح ساختاری می باشد با فرض وجود قطعات FA در کتابخانه قطعات. اما می توانیم هر FA را از گیت های تشکیل دهنده آن بسازیم.

---

[1] Hardware Description Language
[2] Behavioral Level
[3] Datapath level
[4] structural Level




۴- سطح ترانزیستور[۱]: در این سطح ترانزیستورهای سازنده هر گیت مشخص می شوند مثلا یک گیت Notدر تکنولوژی CMOS بصورت زیر ساخته می‌شود:




۵- سطح لایه بندی[۲]: در این سطح طرح نهایی به کارخانه ساخت عرضه می‌شود و آی سی ساخته می شود.

بطورکلی طراحی یک سیستم دیجیتال در مراحل زیر انجام می‌شود


**چه موقع یک طرح کامل است؟**

نیازی نیست که برای طراحی سخت افزار همه مراحل تا سطح گیت و ترانزیستور از ابتدا انجام شود. این مراحل توسط نرم-افزار انجام می شود. یک طرح زمانی کامل است که یک ایده به توصیف معماری مسیر داده یا ساختاری برسد.

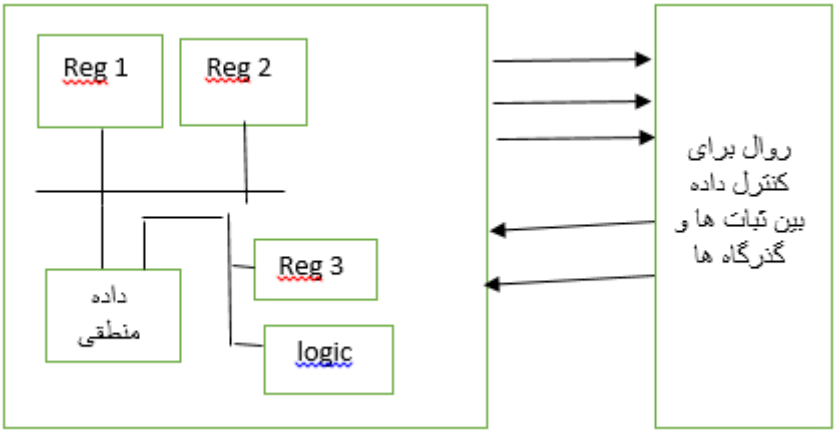
**DA[۳] چیست؟**

فعالیت هایی مثل تبدیل یک سطح طراحی به سطح دیگر،بررسی خروجی یک مرحله طراحی یا تولید داده های تست توسط کامپیوتر راخودکار سازی طراحی می گویند.

---

[۱] Transistor level
[۲] Layout Level
[۳] Design Automation

**مثالی از طراحی مسیر داده:** در طراحی یک CPU ثبات ها و ALU مشخص می شود و ارتباط بین آنها و گذرگاه های آن مشخص می‌شود به این مرحله مسیر داده می‌گویند.



## فصل دوم مدل سازی سخت افزاری

چه سطح از مدل سازی مورد نیاز است؟

۱- اگر مدل مورد استفاده برای مستند سازی عملکرد مدار مورد استفاده شود مدل سطح رفتاری مورد نیاز است.
۲- اگر مدل برای تست جزییات زمانی مدار مورد استفاده باشد جزییات بیشتری در طراحی مورد استفاده است (سطح مسیر داده و گیت).

### ابزار های مدل سازی در دسترس برای یک مهندس

۱- قلم و کاغذ
۲- برنامه های ایجاد شماتیک ORCAD
۳- قابلیت های بردبورد bread board
۴- زبان های توصیف سخت افزار HDL(hardware description language)

### وظیفه HDL چیست؟

برای توصیف سخت افزار به منظور شبیه سازی (simulation) مدل سازی، تست، طراحی و مستند سازی سیستم های دیجیتال استفاده می شود.

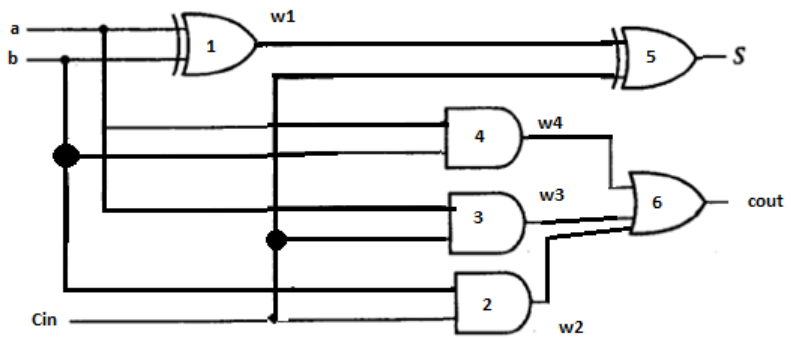
### انواع نرم افزار های HDL

۱- برنامه شبیه سازی: می تواند برای تست طراحی استفاده شود (بررسی verification)
۲- برنامه سنتز( synthesizer): برای تولید سخت افزار به صورت خودکار

نمونه های زبان های توصیف سخت افزار Verilog,VHDL می باشند.

**مثال)** با استفاده از Verilog در سطح گیت یک fulladder طراحی کنید؟

رابطه بین ورودی ها و خروجی های cout و S بصورت $S=a\oplus b\oplus cin$ و Cout=a.b+a.Cin+b.Cin می‌باشد. برای طراحی آن با زبان VHDL با فرض وجود گیت ها بصورت یک کامپوننت می توان بصورت زیر نوشت:




```
entity fulladder Is
        port (a,b,Cin: in bit ; s, Cout : out bit)
end fulladder
```

```
architecture gate_level  of fulladder is
        component and2 port (I1,I2 : IN BIT ,O1: out bit)
        end component;
        component  xor2 port (I1,I2 : IN BIT ,O1: out bit)
        end component;
        component  or3 port (I1,I2,I3 : IN BIT ,O1: out bit)
        end component;
        signal   W1,W2,W3,W4 : bit;
 begin
        G1: xor2  portmap   (a,b,W1);
        G2: and2 portmap (b,Cin,W2);
        G3: and2 portmap (a,Cin,W3);
        G4: and2 portmap (a,b,W4);
        G5: xor2 portmap (W1,Cin,s);
        G6: or3  portmap (W4,W2,W3, Cout);
 end
end gate_level
```

**عناصر وریلاگ:** دو عنصر اصلی برای نوشتن سخت افزار در وریلاگ module, primitive می باشند. برای تعریف ماژول از ساختاری بصورت زیر استفاده می‌کنیم:

**عنصر ماژول:**

```
module module-name
        List of ports
        Declarations
        Specification of the functionality of module
endmodule
```

یک ماژول را می توان در سطوح مختلف طراحی کرد:

۱- سطح رفتاری

```
module module-i
        تعریف پورت ها
        توصیف
        عملکرد رفتاری ماژول
        ماژول بدون توصیف زمانی
endmodule;
```

۲- سطح ساختاری

```
module module-i
        تعریف پورت ها
        توصیف
        توصیف ساختاری
```

ماژول با اطلاعات زمانی
endmodule;

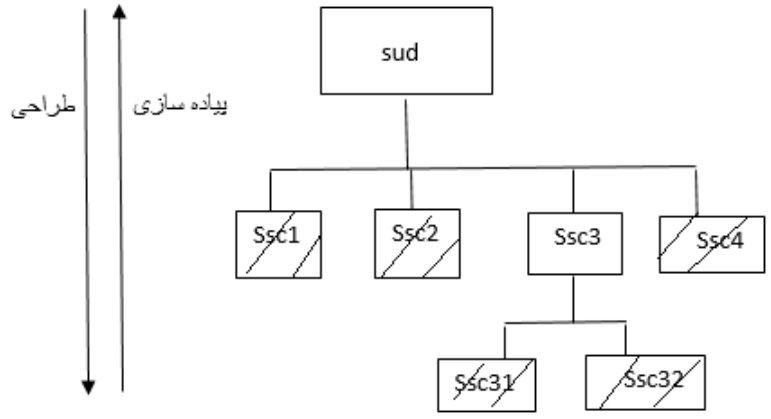
**عنصر primitive** :

primitive name
پورت های سطح بیت
توصیف
عملکرد گیت به صورت جدولی
end primitive;

**طراحی بالا به پایین**

برای طرح های بزرگ به جای حل کردن کل سیستم به یک باره از روش تقسیم و غلبه استفاده می کنیم. آن قدر تقسیم می کنیم تا به یک جز قابل مدیریت برسیم. جزیی قابل مدیریت است که در کتابخانه های نرم افزار موجود باشد یا به سطح گیت برسد یا از قطعاتی که از قبل طراحی کرده ایم برسیم و قابل سنتز باشد.

درخت طراحی به روش بالا به پایین : در طراحی بالا به پایین سیستم اولیه را سیستم تحت طراحی[1] می گویند. با تقسیم آن به زیر قطعات[2] مرحله دوم درخت طراحی بالا به پایین ایجاد می‌شود. اگر به قطعه ای به سطح قابل مدیریت رسید طراحی تمام می شود در غیر اینصورت تقسیم ادامه می‌یابد. مثلا در شکل زیر فرض می‌کنیم Ssc1, Ssc2, Ssc4 به سطح قابل مدیریت رسیده اند و تقسیم ادامه نمی یابد. ولی قطعه Ssc3 هنوز قابل تقسیم است آن را به زیر قطعات دیگر مثل Ssc31, Ssc32 تقسیم می کنیم. هنگامی که مراحل تقسیم تمام شد طراحی بالا به پایین تمام می شود و سپس مرحله پیاده سازی انجام می‌شود و قطعه از پایین به بالا پیاده سازی می شود. در شکل قطعاتی که قابل تقسیم نیستند را هاشور زده ایم.




**روال تقسیم بازگشتی**

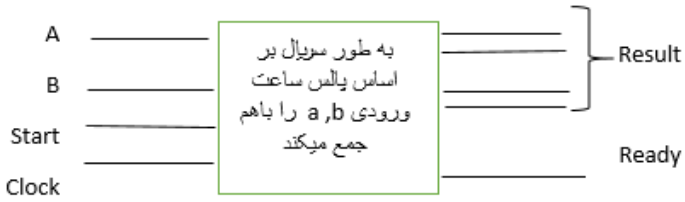
```
Partition (system)
    If hardware mapping of (system) is done then
        Save hardware of system
    Else for every functionally-design part-I  of system
            Partition(part-I)
        End for
```

[1] System Under Design (SUD)
[2] System Sub Component (SSC)

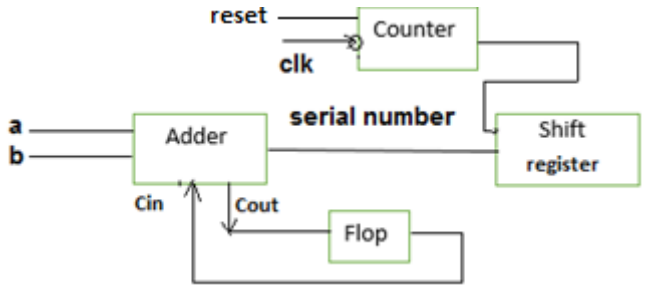
End if

End Partition

**مثال طراحی بالا به پایین:** میخواهیم یک جمع کننده سریال طراحی کنیم خروجی جمع کننده ۸ بیتی خواهد بود. فرض می کنیم کتابخانه یک مالتی پلکسر و یک دی فلیپ فلاپ دارد. کلیت مدار بصورت زیر است:




ورودی A,B بصورت سریال تک بیت، تک بیت وارد می شود هر بیت ورودی در یک پالس ساعت به مدار وارد می‌شود. شروع عمل با یک شدن سیگنال start اتفاق می‌افتد. هنگامی که نتیجه ۸ بیت آماده شد خروجی آن در result قرار می‌گیرد. و سیگنال ready یک می شود تا به مدار مصرف کننده نتیجه اعلام خاتمه عمل نماید. طرح اولیه جمع کننده را بصورت زیر در نظر می گیریم که باید به روش طراحی بالا به پایین به مراحل قابل مدیریت یعنی گیت ها و مالتی پلکسر و فلیپ فلاپ برسند.




فرض می‌کنیم ماژول مالتی پلکسر بصورت زیر از قبل موجود است:

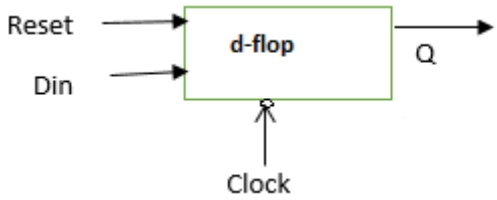



در این ماژول sel, data1, data0 ورودی مالتی پلکسر و z خروجی آن می‌باشد.

دستور assign #6 z=(sel==1'b1)? data1:data0; برای انتساب به خروجی z استفاده می‌شود. عملگر ?برای مقدار دهی شرطی استفاده می شود. در این مثال اگر sel برابر ۱ باشد z=data1 خواهد شد و اگر z=0 شود z=data0 می شود.

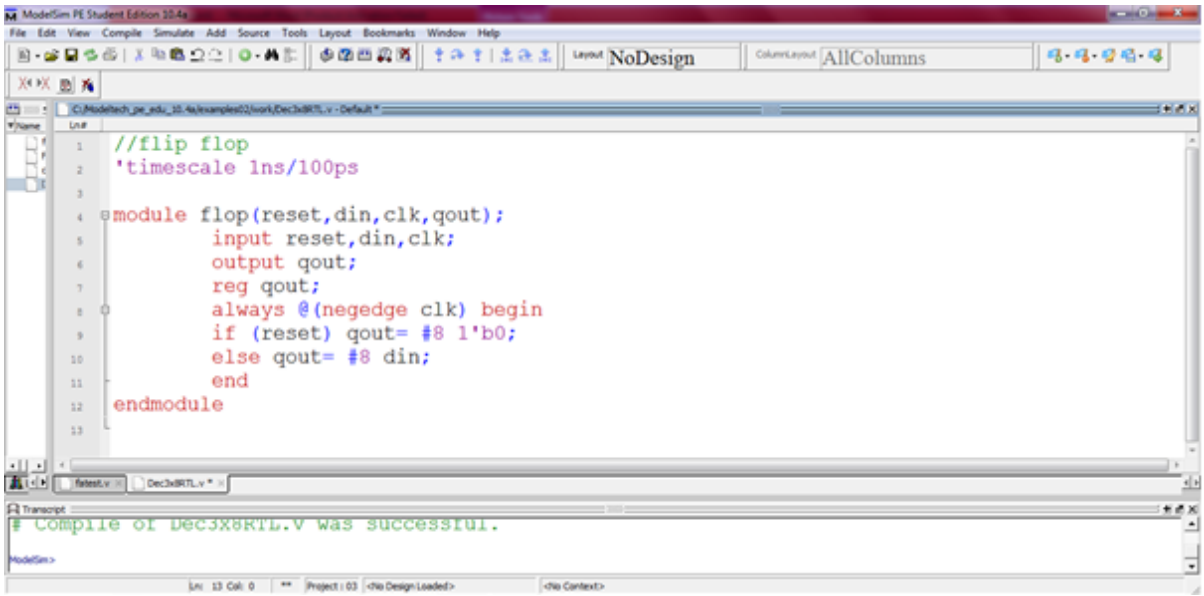
`timescale 1ns/100ps مقیاس زمانی را مشخص می کند که یک نانو ثانیه است با دقت صد پیکو ثانیه. بعد از دستور assign مقدار #6 مشخص کننده تاخیر عمل بر نانو ثانیه است. یعنی z بعد از ۶ نانوثانیه مقدار می گیرد.

فلیپ فلاپ: فرض می کنیم ماژول فلیپ فلاپ هم بصورت زیر آماده است:




در ماژول ارائه شده reset,din,clk ورودی هستند و qout خروجی می‌باشد. در این ماژول reg qout برای تعریف qout بصورت ثبات می باشد. این نوع داده برای داده هایی استفاده می شوند که نقش ورودی و خروجی دارند یا عنصر حافظه هستند. قطعه کد زیر یک بلوک ایجاد می کند که حلقه ای دائمی (always) را ایجاد می کند. بدنه این حلقه به شرطی اجرا می شود که لبه پایین رونده clk اتفاق بیافتد. @(negedge clk). اگر بلوک شامل چند دستور باشد نیازمند begin-end می باشد.

```
always @(negedge clk)
begin
 بدنه
end
```



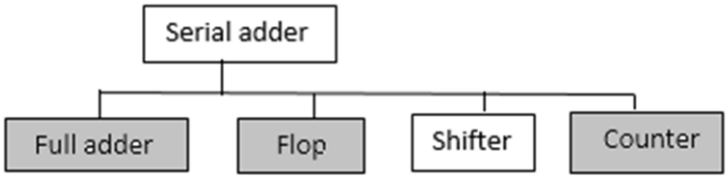

برای این مدار نیازمند یک شمارنده هستیم که تعداد بیت های ورودی را (۰-۷) بشمارد. این مدار ترتیبی است با ورودی reset برای صفر کردن آن و خروجی counting که در صورت مجاز بودن شمارش یک خواهد بود. یعنی هنوز ۸ بیت وارد نشده اند.


```
//counter
`timescale 1ns/100ps

module counter(reset,clk,counting);
        input reset,clk;
        output counting;
        reg counting;
        integer count;
        integer limit; initial limit = 8;
        always @(negedge clk) begin
        if(reset) count=0;
        else begin
             if(count<limit)count=count+1;
             end
             if(count==limit)#8 counting=0;
             else #8 counting=1;
             end
endmodule
```

با توجه به مطالب مطرح شده جمع کننده سریال در سطح رفتاری بصورت ماژول شکل زیر خواهد بود. در این ماژول عملکرد بخش های مدار با هم در سطح رفتاری مطرح شده است. بخش هایی از ماژول امکان تقسیم شدن دارند. مثلا برای انجام شیفت به راست;{{resault[7:1],sum}=resault استفاده شده است. عملگر {} برای ادغام استفاده می شود.

```
//serial adder
`timescale 1ns/100ps

module serial_adder(a,b,start,clk,ready,resault);
        input a,b,start,clk;
        output ready;
        output[7:0] resault;
        reg[7:0] resault;
        reg sum,carry,ready;
        integer count;
        always@(negedge clk)begin
        if(start) begin
            count=0; carry=0;resault=0;
        end else begin
        if(count<8) begin
            count = count+1;
            sum=a^b^carry;
            carry=(a&b)|(a&carry)|(b&carry);
            resault={sum,resault[7:1]};
        end end
        if(count==8)ready=1;
        else ready=0;
        end
endmodule
```

با توجه به طراحی بالا به پایین قطعات مدار را بصورت زیر رسم می کنیم در این شکل shifter بدون هاشور است یعنی هنوز قابل تقسیم به قطعات کوچکتر است.




ماژول تمام جمع کننده بصورت زیر قابل نوشتن است:

```
//Full Adder
'timescale 1ns/100ps

module fulladder(a,b,cin,sum,cout);
        input a,b,cin;
        output sum,cout;
        assign sum=a^b^cin;
        assign cout=(a&b)|(a&cin)|(b&cin);
endmodule
```

ماژول شیفت دهنده را می‌توان بصورت شکل زیر در نظر گرفت. در این ماژول دستور استفاده شده برای مقدار دهی به خروجی (parout) بصورت parout=(reset)?8'b0:((enable)?{sin,parout[7:1]}:parout); می باشد. این یک انتساب شرطی است و معادل عبارت زیر است:

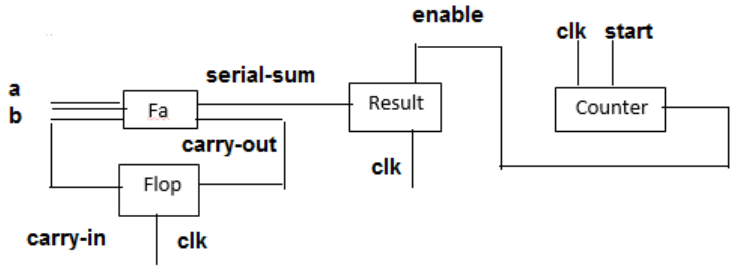
```
If (reset==1) begin
            parout=8'b0;// parout= 8 bit 0 or 00000000;
 else if (enable==0) parout={sin,parout[7:0]}
 else parout=parout;
end
```

```
//shifter
'timescale 1ns/100ps

module shifter(sin,reset,enable,clk,parout);
        input sin,reset,enable,clk;
        output[7:0] parout;
        reg[7:0] parout;
        always@(negedge clk)
        parout=(reset)?8'b0:((enable)?{sin,parout[7:1]}:parout);
endmodule
```

با توجه به اینکه جمع کننده سریال از ۴ قطعه تشکیل شده است می توان آن را بر اساس زیرقطعاتش بصورت زیر نوشت:

```
//serialadder
'timescale 1ns/100ps

module serial_adder(a,b,start,clk,ready,resault);
        input a,b,start,clk;
        output ready;
        output[7:0] resault;
        wire serial_sum,carry_in,carry_out,counting;
        fulladder u1(a,b,carry_in,serial_sum,carry_out);
        flop u2(start,carry_out,clk,carry_in);
        counter u3(start,clk,counting);
        shifter u4(serial_sum,start,counting,clk,resault);
        assign ready =~ counting;
endmodule
```

شکل مدار تاکنون می‌تواند بصورت زیر طراحی شود:



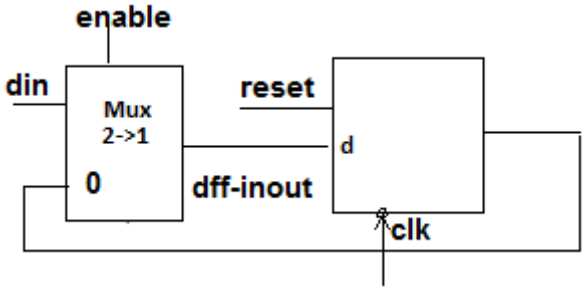

خود فلیپ فلاپ استفاده شده در مدار عملکردی بصورت زیر دارد:

```
//flipflop D
'timescale 1ns/100ps

module der_flop(din,reset,enable,clk,qout);
        input din,reset,enable,clk;
        output qout;
        reg qout;
        always@(negedge clk)begin
        if(reset) qout= #8 1'b0;
        else
        if(enable) qout= #8 din;
        end
endmodule
```

با توجه مطالب گفته شده می‌توان مدار شیفت دهنده را بصورت زیر طراحی کرد:

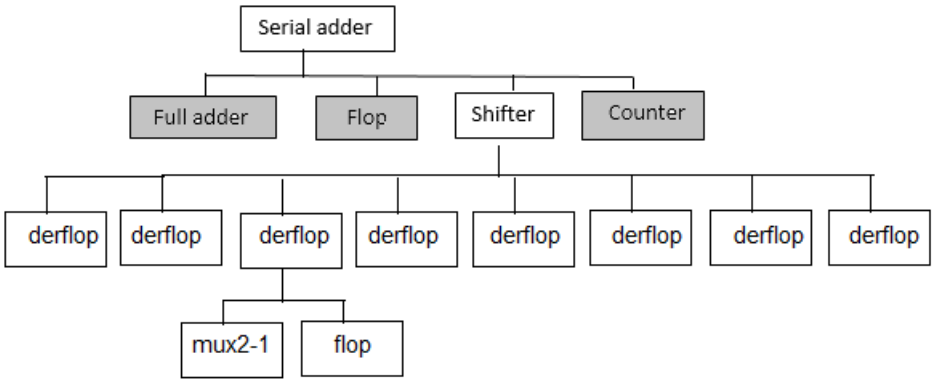
```
//SHIFTER
'timescale 1ns/100ps

module  shifter(sin,reset,enable,clk,parout);
        input sin,reset,enable,clk;
        output[7:0] parout;
        reg[7:0] parout;
        der_flop b7(sin,reset,enable,clk,parout[7]);
        der_flop b6(parout[7],reset,enable,clk,parout[6]);
        der_flop b5(parout[6],reset,enable,clk,parout[5]);
        der_flop b4(parout[5],reset,enable,clk,parout[4]);
        der_flop b3(parout[4],reset,enable,clk,parout[3]);
        der_flop b2(parout[3],reset,enable,clk,parout[2]);
        der_flop b1(parout[2],reset,enable,clk,parout[1]);
        der_flop b0(parout[1],reset,enable,clk,parout[0]);
endmodule
```

برای طراحی der_flop می‌توان از یک فلیپ فلاپ D و یک مالتی پلکسر بصورت زیر استفاده کرد:



```
//flipflop D
`timescale 1ns/100ps

module der_flop(din,reset,enable,clk,qout);
        input din,reset,enable,clk;
        output qout;
        reg qout;
        wire dff_in;
        mux2_1 mx(enable,din,qout,dff_in);
        flop ff(reset,dff_in,clk,qout);
endmodule
```

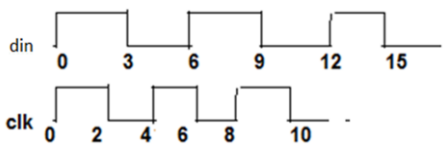
با توجه به مراحل طراحی بالا به پایین شکل نهایی مدار بصورت زیر است:




**نوشتن ماژول تست:** برای اینکه بتوانیم عملکرد هر مداری را بسنجیم باید برای آن تست بنویسیم. در ماژول تست باید ورودی ها را تغییر دهیم و نتایج خروجی ها را مشاهده کنیم. به عنوان مثال ماژول تست فلیپ فلاپ بصورت زیر است:

```
module flop(reset ,din , clk , qout );
        input reset , din , clk;
        output qout;
        reg qout;
        always @(negedge clk)
        begin
                if(reset) qout=# 8 1` b 0;
                else qout=# 8 din;
        end
endmodule;
```

```
//test flipflop D
`timescale 1ns/100ps

module test_flop
        reg din,reset,clk;
        wire qout;
        flop u1(reset,din,clk,qout);
        initial begin
        clk=1'b1; din=1'b1;
        #5 reset=1'b1; #11 reset=1'b0;
        #23 $finish;
        end
        always #3 din=~din;
        always #2 clk=~clk;
endmodule
```



## فصل سوم  عملگر های وریلاگ

عملیات منطقی رایجترین نوع عملیات برای توصیف عملکرد اجزای سخت افزاری در سطح گیت می باشد. علاوه بر این عملیات، عملیات دیگری نیز برای توصیف عملیات سخت افزار در سطح رفتاری وجود دارند. اکثر عملیات پوشش داده شده در نرم افزارهای برنامه نویسی در وریلاگ نیز پشتیبانی می‌شوند. عملگرها را براساس عملکردشان دسته بندی می کنیم.

ابتدا عملگرهای پایه و رابطه ای را لیست می کنیم.

| نتیجه | توصیف | عمل | عملیات پایه |
|---|---|---|---|
| چند بیت | چهار عمل اصلی ریاضی | / * - + | ریاضی |
| یک بیت | مقایسه | >,>=, <,<= | رابطه ای |

عملگرهای ریاضی روی همه انواع داده ها کار می کنند و معنی رایجشان را دارند. عملگرهای رابطه ای و تساوی مقادیر ۰ و ۱ را به عنوان نتیجه به معنی درست و غلط بر میگردانند.

| نتیجه | توصیف | عمل | برابری |
|---|---|---|---|
| یک بیت | شامل z, x نیستند(برابری و نابرابری) | !=, == | منطقی |
| یک بیت | شامل z, x(برابری و نابرابری) | !==,=== | حالت |

سه گروه عملگرهای منطقی شامل عملیات منطقی برای بردارها، اعداد اسکالر و کاهش بردار به عدد اسکالر می باشند.

| نتیجه | توصیف | عمل | منطقی |
|---|---|---|---|
| یک بیت | منطقی ساده | && \|\| ! | اتصال دهنده |
| چند بیت | عملیات منطقی برداری | ~ & \| ^ ^~ | بیتی |
| یک بیت | انجام عملیات روی همه بیت ها | &, ~&, \|,~\|, ^, ~^ | کاهش |

در عملگرهای شیفت عدد سمت راست مشخص کننده تعداد شیفت‌ها می باشد.

| نتیجه | توصیف | عمل | شیفت |
|---|---|---|---|
| چند بیت | از سمت راست صفر وارد می شود nبیت به راست شیفت می کند | <<n | چپ |
| چند بیت | از سمت چپ صفر وارد می شود n بیت به چپ شیفت می کند | n>> | راست |

سایر عملیات در جدول زیر لیست شده است.

| نتیجه | توصیف | عمل | سایر |
|---|---|---|---|
| چند بیت | محاسبه باقی مانده تقسیم | ٪ | ماژول |
| چند بیت | اتصال بیت | {} | الحاق |
| چند بیت | اتصال و تکرار | {{}} | تکرار |
| چند بیت | If then else | شرط | ؟: |

همانطور که قبلا گفتیم وریلاگ حساس به حروف نیست و عبارات طولانی می‌توانند در چند سطر نوشته شوند. برای نوشتن توضیحات از // استفاده می کنیم و برای ایجاد چندین سطر توضیح آنها را بین */ و /* قرار می‌دهیم.

**مفاهیم پایه در وریلاگ**

تفاوت زبان های برنامه نویسی نرم افزار و زبان های برنامه نویسی سخت افزار چیست؟

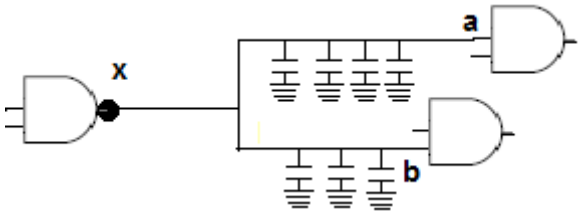
- در زبان های نرم افزاری به صورت ترتیبی (sequential) فکر کنیم. یعنی یک دستور به شرطی اجرا می شود که دستورات قبل از آن تمام شده باشد.اما در سخت افزار (به ویژه درکد ساختاری) به صورت همروند[1] فکر می کنیم. یعنی دستورات با هم اجرا می شوند. (اجرای همزمان اجزای سخت افزاری مختلف).
- تفاوت دوم این است که در نرم افزار برای اجرای دستورات زمان[2] قرار نمی دهیم ولی در سخت افزار از زمان استفاده می کنیم. زیرا مقادیر بین اجزای سخت افزاری از طریق سیم ها و گذرگاه‌ها عبور می‌کنند و سیم ها به دلیل وجود اثرات خازنی و مقاومتی تاخیرهای متفاوتی دارند. البته همیشه در مقدار دهی وریلاگ از توصیف زمانی استفاده نمی‌شود.

A:=x

B:=x در نرم افزار

سخت افزار همروند

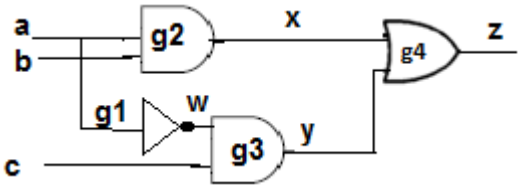
```
assign #(4* unit-dalay) a=x;
assign #(۳* unit-dalay) b=x;
```




همانطور که در شکل مشاهده می‌شود x بطور همروند به دو ورودی a,b منتقل می‌شود ولی به توجه به اینکه فاصله سیمی آنها برابر نیست اثرات خازنی ورودی a بیشتر است. بنابراین x یک واحد زمانی تاخیر دیرتر به a می‌رسد.

هرگاه در کد وریلاگ در سطح ساختاری کد بنویسیم به صورت همروند کار می کند اما در سطح رفتاری به صورت ترتیبی کارمی‌کند.

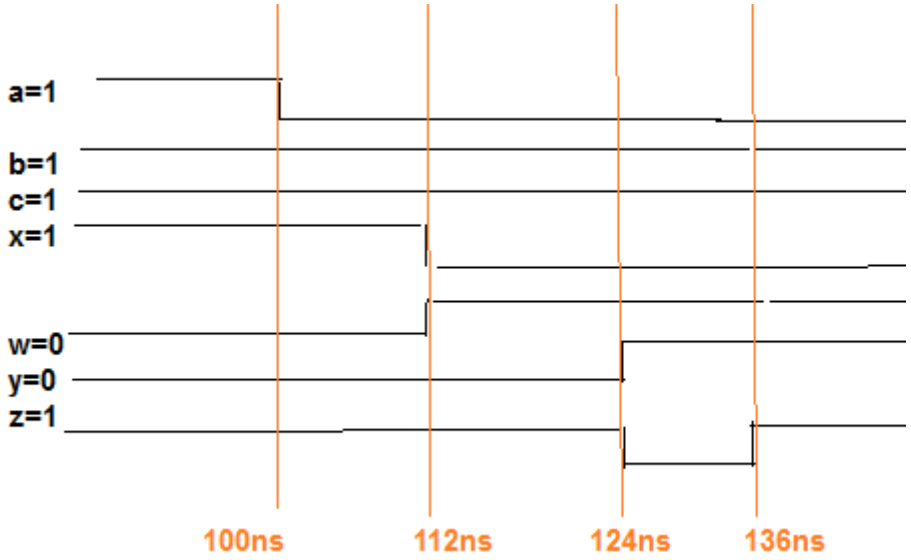
مثال ) مدار زیر را در نظر بگیرید و همروندی سیگنال ها را مشخص کنید.فرض کنید هر گیت تاخیر 12ns دارد . سپس کد وریلاگ بنویسید که عملکرد مدار را نشان می دهد.




فرض کنید در ابتدا a,b,c یک باشند و در زمان 100ns مقدار a=0 شود. نمودار مربوطه را رسم کنید.

---

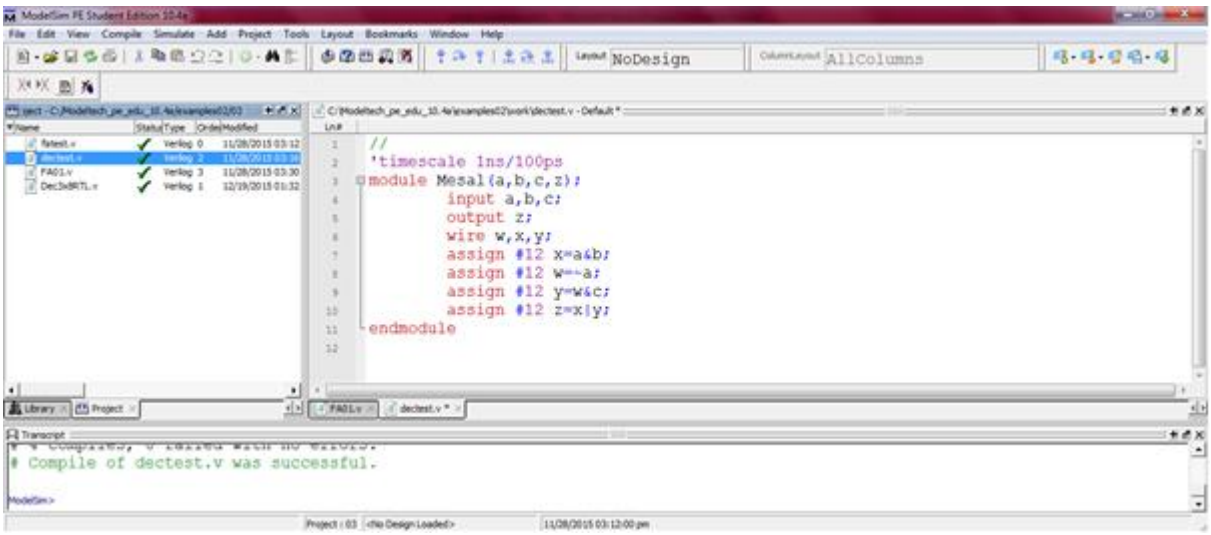
[1] concurrent
[2] Timing




با تغییرa در زمان ns۱۰۰ به صفر ورودی های g1 و g2 تغییر میکند و بعد از ۱۲ ns خروجی این گیت ها یعنی xوy تغییر میکند(زمان ۱۲ns).یعنی g1 وg2 همروند هستند ودر زمان ۱۱۲ns x وy تغییر کرده اند. بنابراین ورودی گیت‌های g3 وg4 تغییر کرده اندو در زمان ۱۲۴ns خروجی y وz تغییر کرده اند و g3 وg4 همروند می باشند. چون در زمان ۱۲۴ ns تغییر کرده اند. ورودی g4 تغییر داشته است. خروجی z در زمان ۱۳۶ns تغییر می‌کند. بنابراین یک پالس هزارد (ایستای سطح صفر)در خروجی z داریم.

بدون زمان z=ab+dc
- z=1 ,abc =111
- Z=1 , abc=011

برنامه وریلاگ معادل مدار بصورت زیر است:




**تعریف متغیر ها درVerilog:** قوانین تعریف متغیرها در وریلاگ مشابه زبان های دیگر است به اضافه موارد زیر:

۱- اولین کاراکتر حرف می باشد.
۲- علامت $ قبل از یک شناسه به عنوان فراخوانی کار های سیستمی و توابع می باشد.
۳- علامت ` برای راهنماهای کامپایلری است.

` timescale 1ns/100 ps

۴- comment به صورت زیر است: یک خط توضیح //

/* توضیحات */

اعداد : مقادیر منطقی در وریلاگ دارای چهار مقدار هستند. x,z,0,1

مقدار دهی اعداد با حروف b باینری-d مبنای ۱۰ -o مبنای ۸- h مبنای ۱۶ می‌باشد و ساختار کلی مقداردهی بشکل زیر است.

مقدار یکی از حروف فوق ' تعداد بیت ها

X= 4'd 5 یعنی متغیر ۴ بیتی است با مقدار ۵=۰۱۰۱

X=8 'b 101 => 00000101

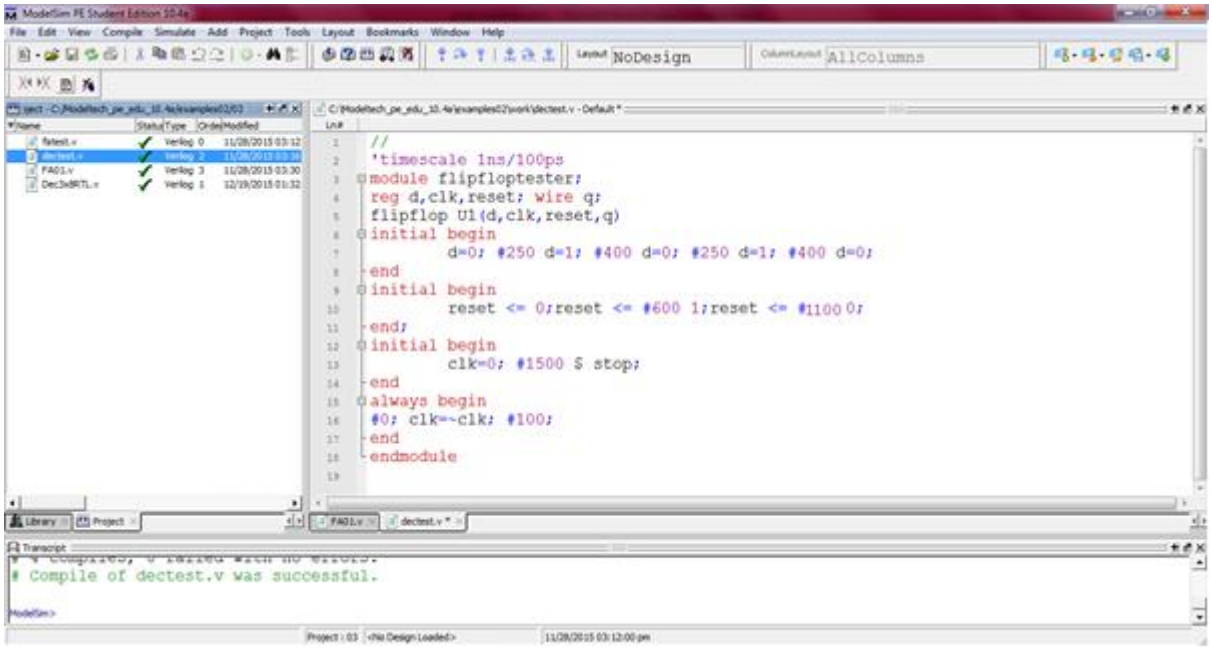
X= -8 'b 101 => 11111011

X=10 'o 752 => 0 111 101 010

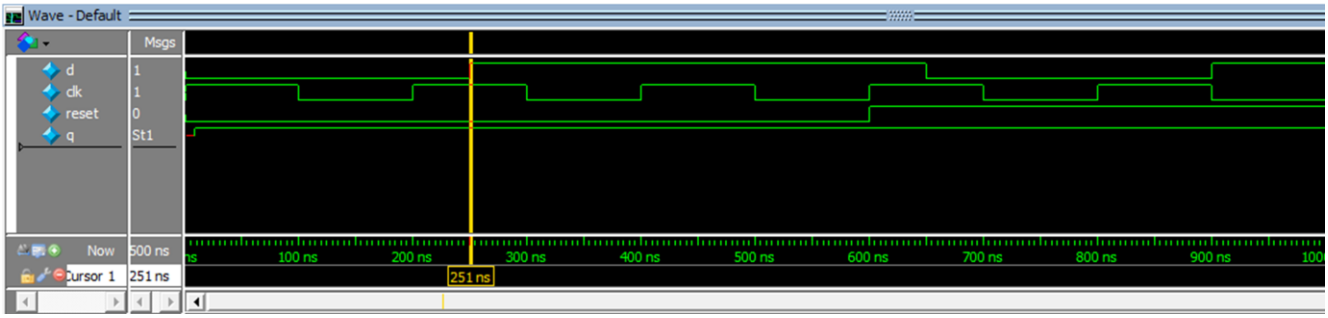
X= 8' h f => 00001111

X= 12 ' h xf => xxxxxxxx1111

مثال: همانطور که گفتیم برای چک کردن عملکرد یک مدار باید برای آن تست نوشت با توجه به مطالب گفته شده یک نمونه تست دیگر برای برای فلیپ فلاپ بصورت زیر است:




وقتی در یک ماژول تست چند بلوک initial و always نوشته می شوند بصورت همروند اجرا می‌شوند و داخل آنها وابسته به نوع نوشتار است. اگر در مقدار دهی از عملگر = استفاده شود مقدار دهی ترتیبی است و اگر از عملگر <= استفاده شود مقدار دهی همروند است. بنابراین موج ایجاد شده روی d, clk, reset بصورت زیر خواهد بود:

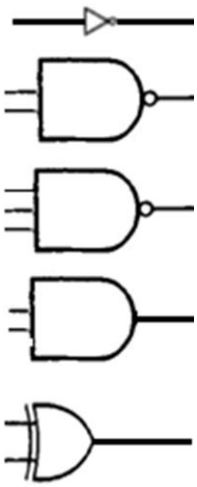



خروجی که مشاهده می کنید نتیجه شبیه سازی تست در نرم افزار modelsim می باشد. همانطور که مشاهده می شود کلاک در زمان ۰ یک می شود و هر ۱۰۰ نانو ثانیه متمم می شود.

## فصل چهارم  توصیف ساختاری سخت افزار

نکات سطح ساختاری:

۱. دستورات Verilog به صورت همروند اجرا می‌شوند.
۲. سیگنال‌های زمانی در طراحی مدار استفاده می‌شود.
۳. ساختار های آماده ی Verilog در این سطح شامل گیت های(and,nand,or,xor,xnor,not,buf) میباشد.



not  # 4 (z,a)

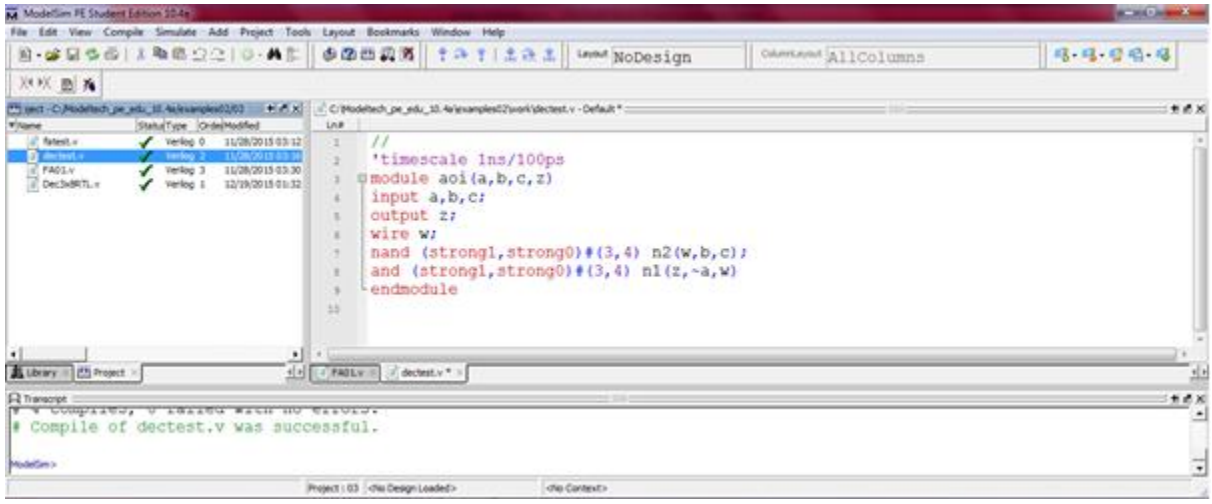
nand #5(z,a,b)

nand #6(z,a,b,c)

and #5(z,a,b)

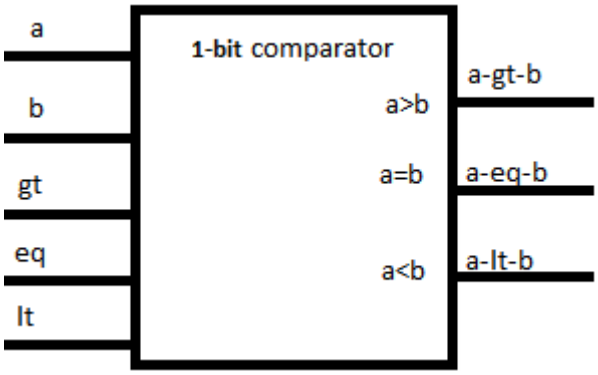
xor #5(z,a,b)

مثال: توصیف ساختاری برای محاسبه ی عبارت z=a'.(bc)' را بنویسید؟


نکته : به جای strong میتوان highz0,weak0,pull0,supply0 را نیز نوشت.

مثال: یک مقایسه گر تک بیتی طراحی کنید که قابل توسعه باشد؟




a-gt-b=ab’+ab.gt+a’b’.gt=a.gt+ab’+b’.gt

a-eq-b=a’b’.eq+ab.eq

a-lt-b=a’b+a’b’.lt+ab.lt=a’b+b.lt+a’.lt

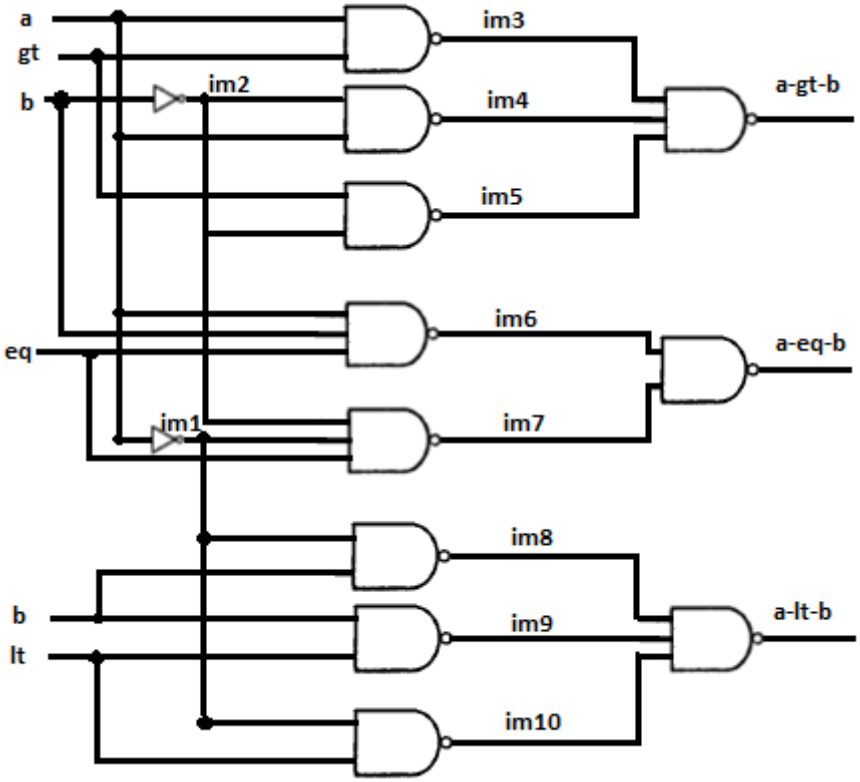
تبدیل عبارات به NAND:

a_gt_b=a gt+a b’+b’ gt=((a gt)’.(a b’)’.(b’ gt)’)’

e_eq_b=a.b.eq+a’b’eq=(( a.b.eq)’.( a’b’eq)’)’

a_lt_b=a’b+b.lt+a’.lt=((a’b)’.(b.lt)’.(a’.lt)’)’

شکل مدار بصورت زیر است:




با توجه به شکل مدار کد وریلاگ ساختاری آن بصورت زیر است:

ModelSim ALTERA STARTER EDITION 10.1b

File Edit View Compile Simulate Add Source Tools Layout Bookmarks Window Help

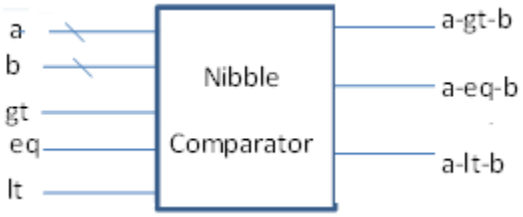
C:/altera/12.1/modelsim_ase/bit_comprator.v (/test_nibble_comprator/u1/c3) - Default

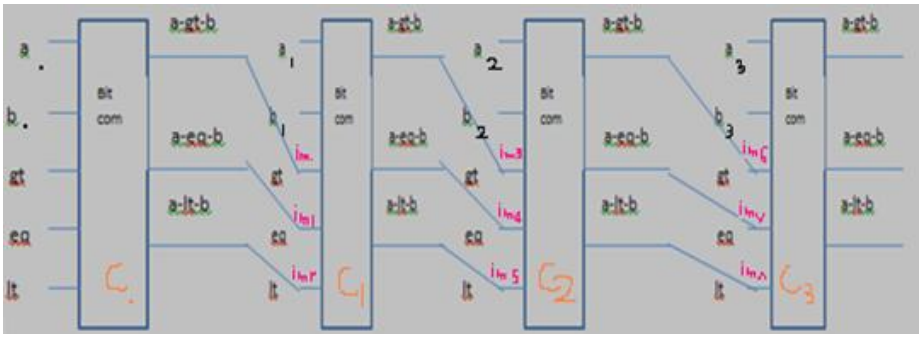
```
1   `timescale 1ns/1ns
2   module bit_comprator(a,b,gt,eq,lt,a_gt_b,a_eq_b,a_lt_b);
3   input a,b,gt,eq,lt;
4   output a_gt_b,a_eq_b,a_lt_b;
5   wire im[1:10];
6   not #(4)   g0(im[1],a);
7   not #(4)   g1(im[2],b);
8   nand #(5)  g2(im[3],a,gt);
9   nand #(5)  g3(im[4],a,im[2]);
10  nand #(5)  g4(im[5],im[2],gt);
11  nand #(6)  g5(a_gt_b,im[3],im[4],im[5]);
12  nand #(6)  g6(im[6],a,b,eq);
13  nand #(6)  g7(im[7],eq,im[2],im[1]);
14  nand #(5)  g8(a_eq_b,im[6],im[7]);
15  nand #(5)  g9(im[8],im[1],b);
16  nand #(5)  g10(im[9],b,lt);
17  nand #(5)  g11(im[10],im[1],lt);
18  nand #(6)  g12(a_lt_b,im[8],im[9],im[10]);
19  endmodule
```

Transcript | test_nibble_comprator.v | Library | Project | sim | Objects | nibble_comprator.v | bit_comprator.v

Ln: 1 Col: 0 | Project : bit_comprator | Now: 550 ns Delta: 0 | sim:/test_nibble_comprator/#INITIAL#9

استفاده از bit comparator و طراحی یک مقایسه گر ۴بیتی:




ModelSim ALTERA STARTER EDITION 10.1b

File Edit View Compile Simulate Add Source Tools Layout Bookmarks Window Help

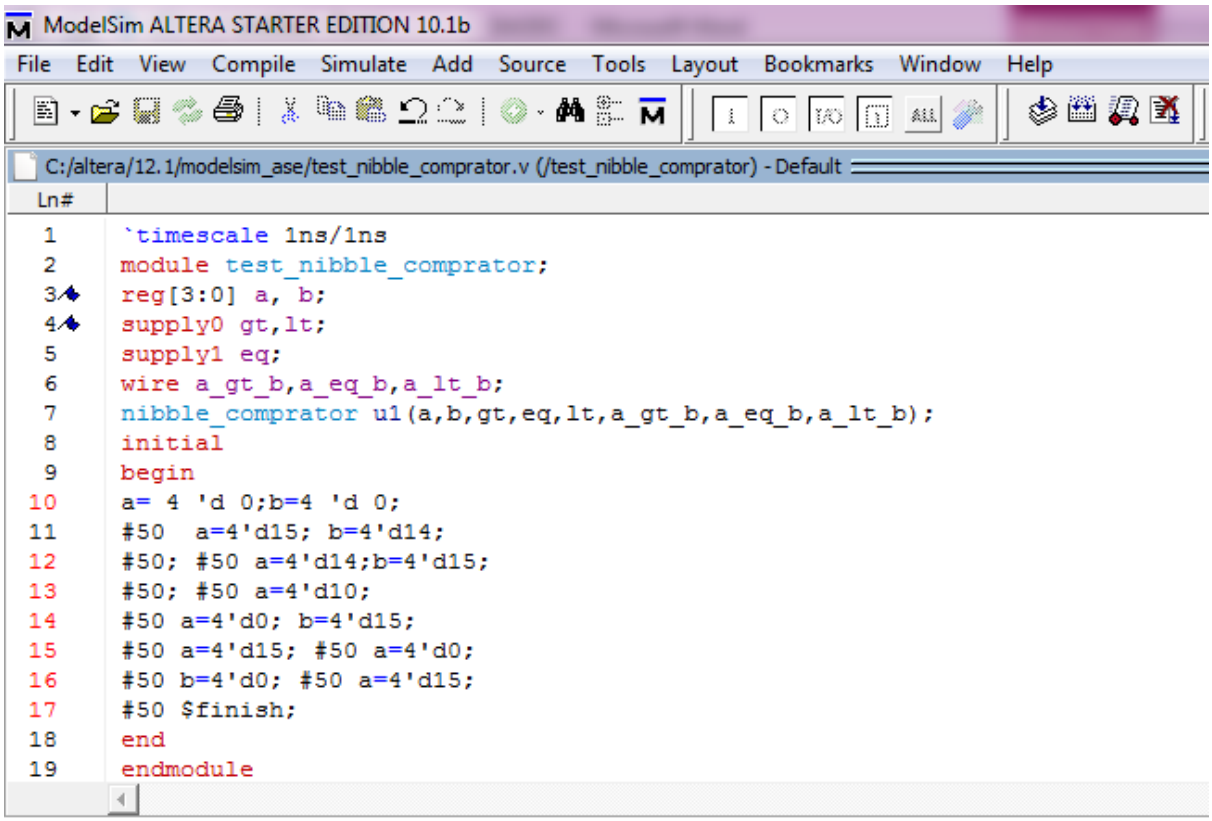
C:/altera/12.1/modelsim_ase/nibble_comprator.v (/test_nibble_comprator/u1) - Default

```
1   module nibble_comprator(a,b,gt,eq,lt,a_gt_b,a_eq_b,a_lt_b);
2   input[3:0] a,b;
3   input gt,eq,lt;
4   output a_gt_b,a_eq_b,a_lt_b;
5   wire im[0:8];
6   bit_comprator c0(a[0],b[0],gt,eq,lt,im[0],im[1],im[2]);
7   bit_comprator c1(a[1],b[1],im[0],im[1],im[2],im[3],im[4],im[5]);
8   bit_comprator c2(a[2],b[2],im[3],im[4],im[5],im[6],im[7],im[8]);
9   bit_comprator c3(a[3],b[3],im[6],im[7],im[8],a_gt_b,a_eq_b,a_lt_b);
10  endmodule
```

Transcript | test_nibble_comprator.v | Library | Project | sim | Objects | nibble_comprator.v

- برای ساخت یک مقایسه گر ۴ بیتی می‌بایست چهار بار یک مقایسه گر تک بیتی را فراخوانی نمود.
- برای ساخت مقایسه گر ۸ بیتی نیز می‌توان یا ۸ بار مقایسه گر تک بیتی را فراخوانی کرد و یا دو بار مقایسه گر ۴ بیتی را فراخوانی کرد.
- Wire را می‌توان در قالب آرایه نیز تعریف کرد.
- نحوه ساخت و کار این مدار موازیست.

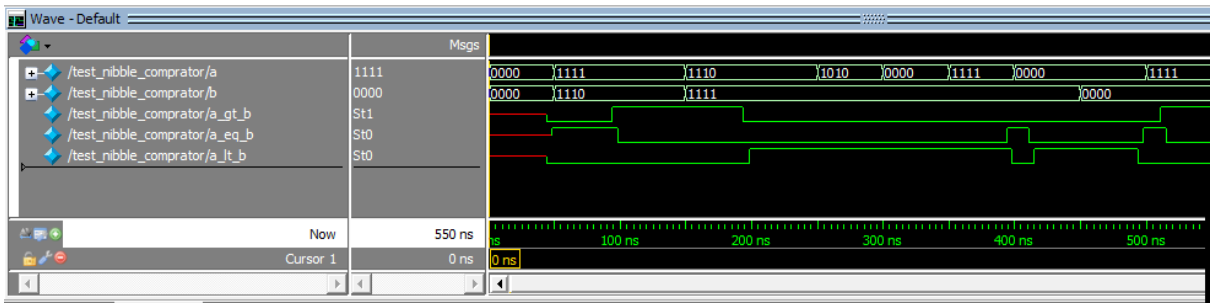
نوشتن تست برای مقایسه گر ۴ بیتی:




مقادیر اختصاص داده شده به a,b بصورت جدول زیر می باشد.

| Time –ns | a | B |
|---|---|---|
| 0 | 0 | 0 |
| 50 | 15 | 14 |
| 100 | 15 | 14 |
| 150 | 14 | 15 |
| 200 | 14 | 15 |
| 250 | 10 | 15 |
| 300 | 0 | 15 |
| 350 | 15 | 1 |
| 400 | 0 | 15 |
| 450 | 0 | 0 |
| 500 | 15 | 0 |
| 550 | end | End |

خروجی سیگنال ها تا زمان ۵۵۰ نانو ثانیه بصورت نمودار زیر است:




## تمرین ۱:

۱. توصیف ساختاری یک فول اددر را با استفاده از گیتهای ارایه شده بنویسید.

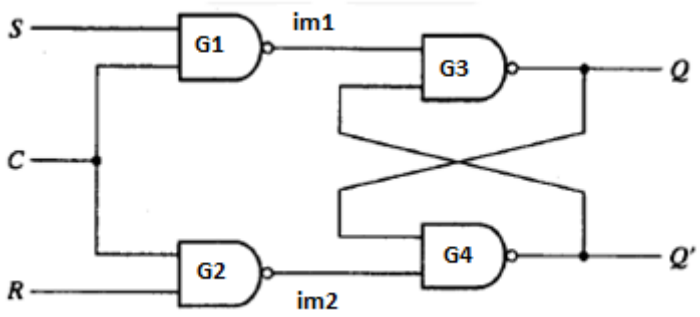
۲. توصیف ساختاری یک جمع کننده ۸بیتی را با استفاده از فول اددر طراحی شده بنویسید.

۳. یک جمع کننده ۸ بیتی با استفاده از گیت‌های xor و جمع کننده ۸ بیتی طراحی کنید.

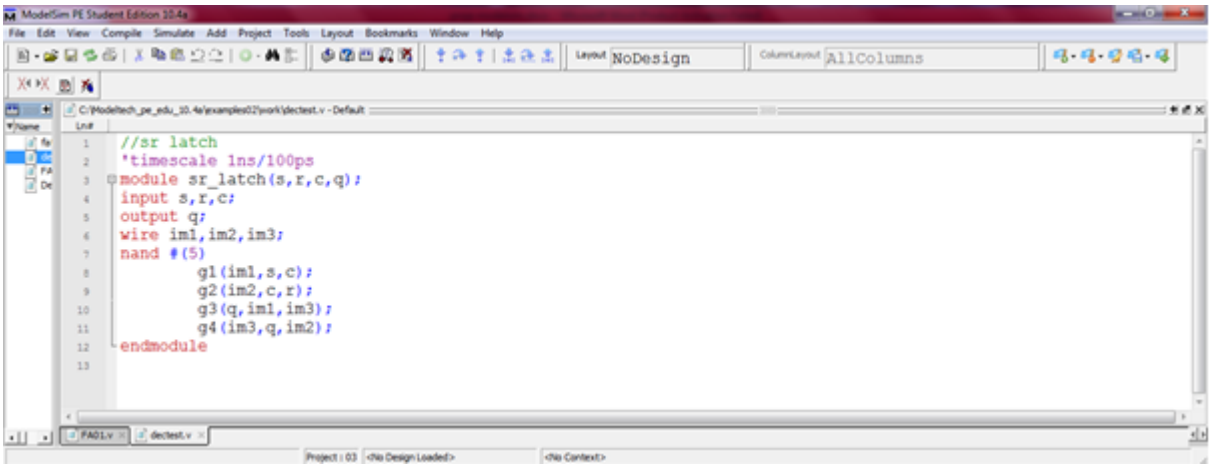
---

طراحی یک فلیپ فلاپ در سطح ساختاری، طراحی یک مدار ترتیبی در سطح ساختاری:

***طراحی فلیپ فلاپ :s_r




ماژول:



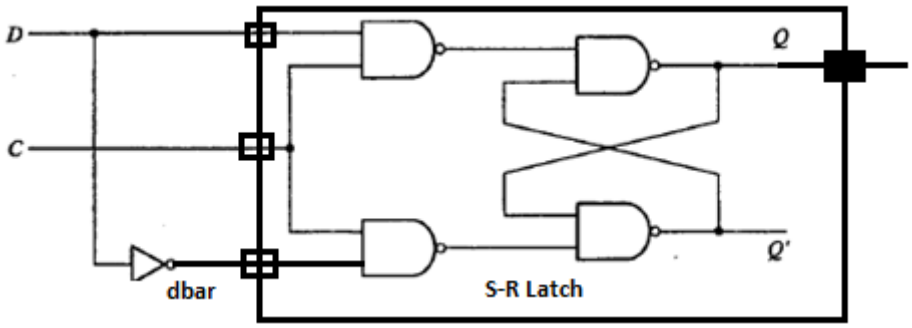
```
//sr latch
`timescale 1ns/100ps
module sr_latch(s,r,c,q);
input s,r,c;
output q;
wire im1,im2,im3;
nand #(5)
        g1(im1,s,c);
        g2(im2,c,r);
        g3(q,im1,im3);
        g4(im3,q,im2);
endmodule
```

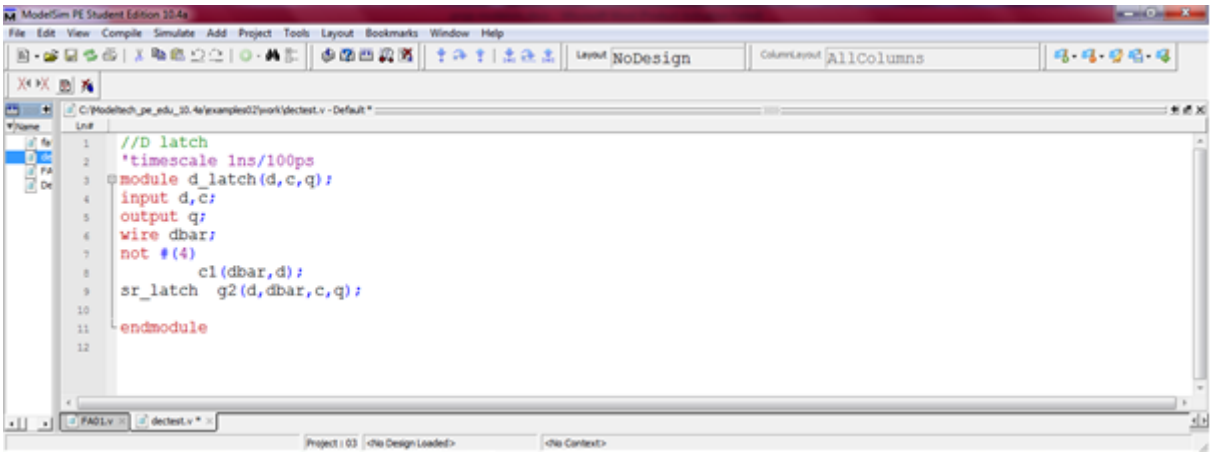
کلاک سطح بالا

***طراحی فلیپ فلاپ d با استفاده از:sr

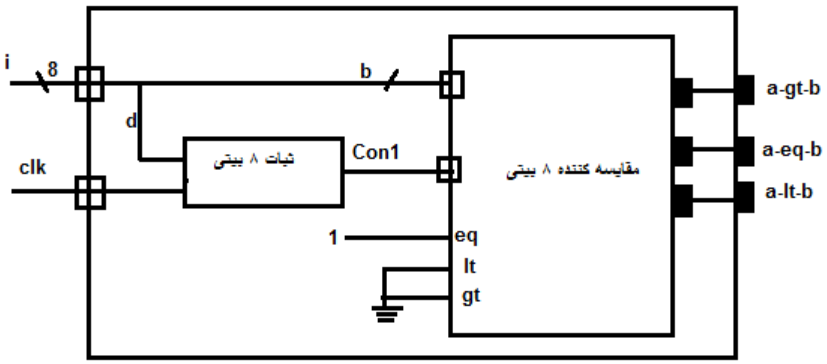
برای رسم فلیپ فلاپ dکافیست که ورودی d را به s و dbarرا به r در فلیپ فلاپ srبدهیم.

نکته:در کتاب وریلاگ ورودی را با □ و خروجی را با▪ نمایش میدهند.



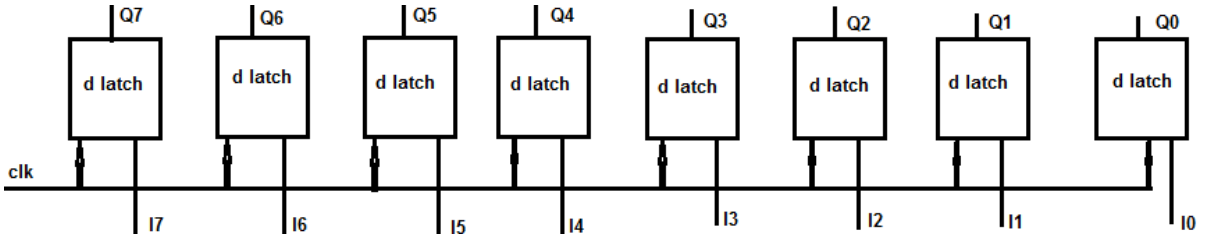




**مدار مقایسه کننده سریال با استفاده از مقایسه کننده هشت بیتی با بار موازی:**



```
//
`timescale 1ns/100ps
module old_new_comprator(i,clk,gt_compare,eq_compare,lt_compare);
input [7:0] i;
input clk;
output gt_compare,eq_compare,lt_compare;
wire [7:0] con1;
supply1 vdd;
supply0 gnd;
byte_latch(i,clk,con1);
byte_latch(con1,i,gnd,vdd,gnd,gt_compare,eq_compare,lt_compare);

endmodule
```

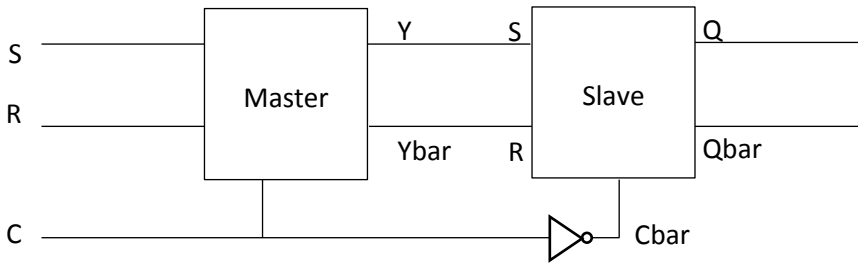
*****طراحی بایت لچ:۸ بار فراخوانی d-latch که تک بیتی است:



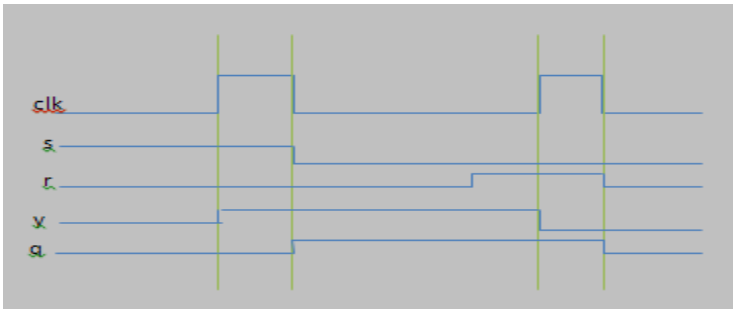

```
//
`timescale 1ns/100ps
module byte_latch(di,c,q);
input [7:0] di;
input c;
output[7:0] q;
d_latch c7(di[7],c,q[7]);
d_latch c6(di[6],c,q[6]);
d_latch c5(di[5],c,q[5]);
d_latch c4(di[4],c,q[4]);
d_latch c3(di[3],c,q[3]);
d_latch c2(di[2],c,q[2]);
d_latch c1(di[1],c,q[1]);
d_latch c0(di[0],c,q[0]);
endmodule
```

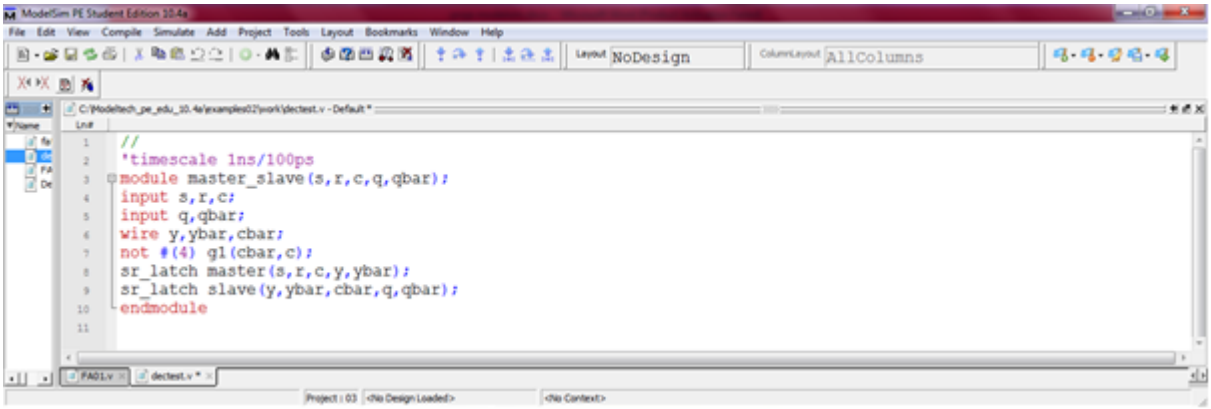
**استفاده از فلیپ فلاپ master-slave برای طراحی کلاک لبه پایین رونده و بالا رونده:**

اگر دو تا فلیپ فلاپ sr را کنار هم بگذاریم و به یکی clk را وصل کنیم و به دیگری clk’ در اینصورت کلاک با لبه ایجاد می‌شود اگر به فلیپ فلاپ master کلاک و به slave، clk’ وصل شود لبه پایین رونده ایجاد می شود و اگر به slave، clk و به master، clk’ متصل شود لبه بالارونده ایجاد می شود.



نمودار زمانی:



## فصل ۵: سازماندهی طراحی

تعریف تابع[۱]

```
Function    اسم تابع;
        تعریف ورودی‌ها;
 Begin
        بدنه تابع
        خروجی=اسم تابع
 End
End function
```

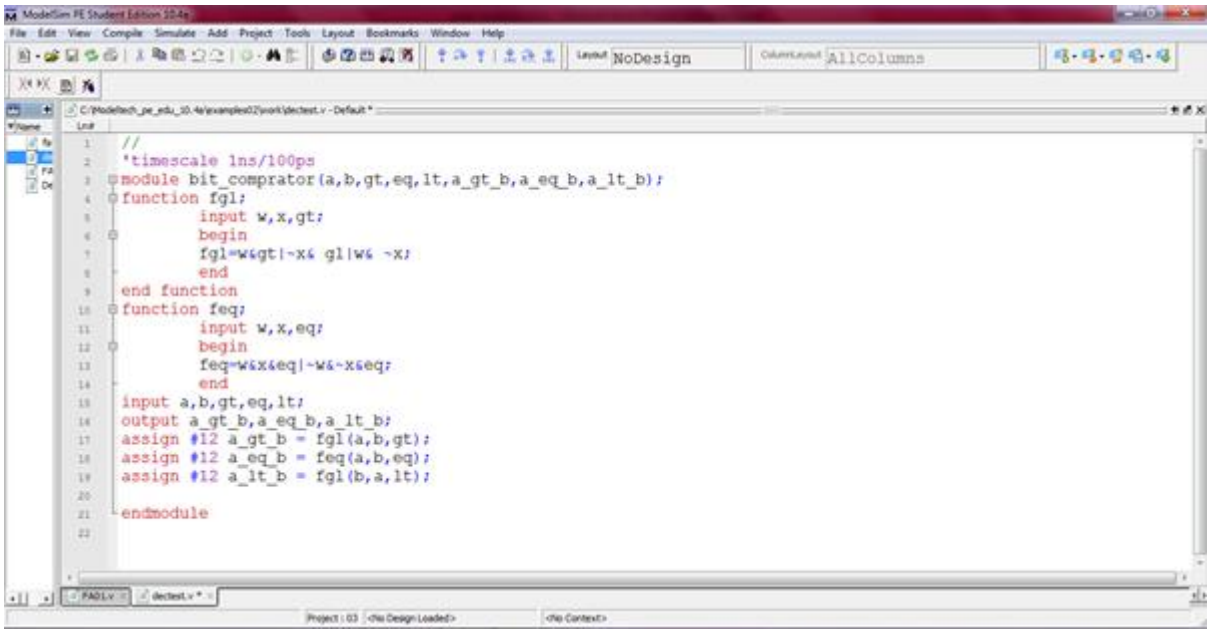
مثال :طراحی مقایسه گر تک بیتی با تابع:

a_gt_b=a gt+a b'+b' gt

e_eq_b=a.b.eq+a'b'eq

a_lt_b=a'b+b.lt+a'.lt

برای طراحی مقایسه گر دو تابع با نام های fgl(greater-less) وfeq (equal) تعریف می کنیم و سپس آنها را برای محاسبه سه خروجی صدا میزنیم.



```
//
`timescale 1ns/100ps
module bit_comprator(a,b,gt,eq,lt,a_gt_b,a_eq_b,a_lt_b);
function fgl;
        input w,x,gt;
        begin
        fgl=w&gt|~x& gl|w& ~x;
        end
end function
function feq;
        input w,x,eq;
        begin
        feq=w&x&eq|~w&~x&eq;
        end
input a,b,gt,eq,lt;
output a_gt_b,a_eq_b,a_lt_b;
assign #12 a_gt_b = fgl(a,b,gt);
assign #12 a_eq_b = feq(a,b,eq);
assign #12 a_lt_b = fgl(b,a,lt);

endmodule
```

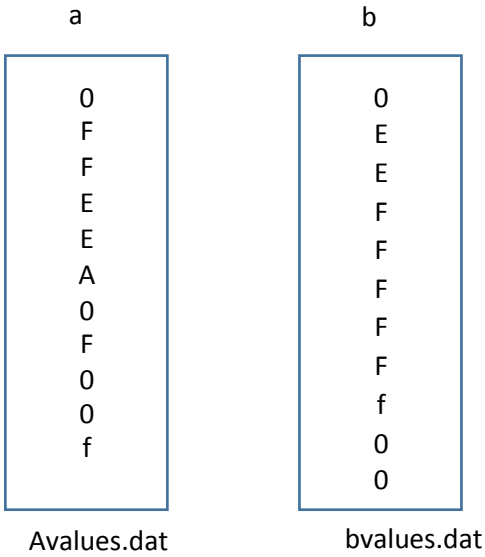
- a_gt_b=fgl(a,b,gt) برای محاسبه بزرگتر بودن a از b می باشد و با جابجایی a,b و صدا زدن آن a کوچکتر از b بدست می آید.
- تاخیر تابع صفر می‌باشد. به همین خاطر تابع را با تاخیر زمانی ۱۲ واحد فراخوانی کرده ایم.

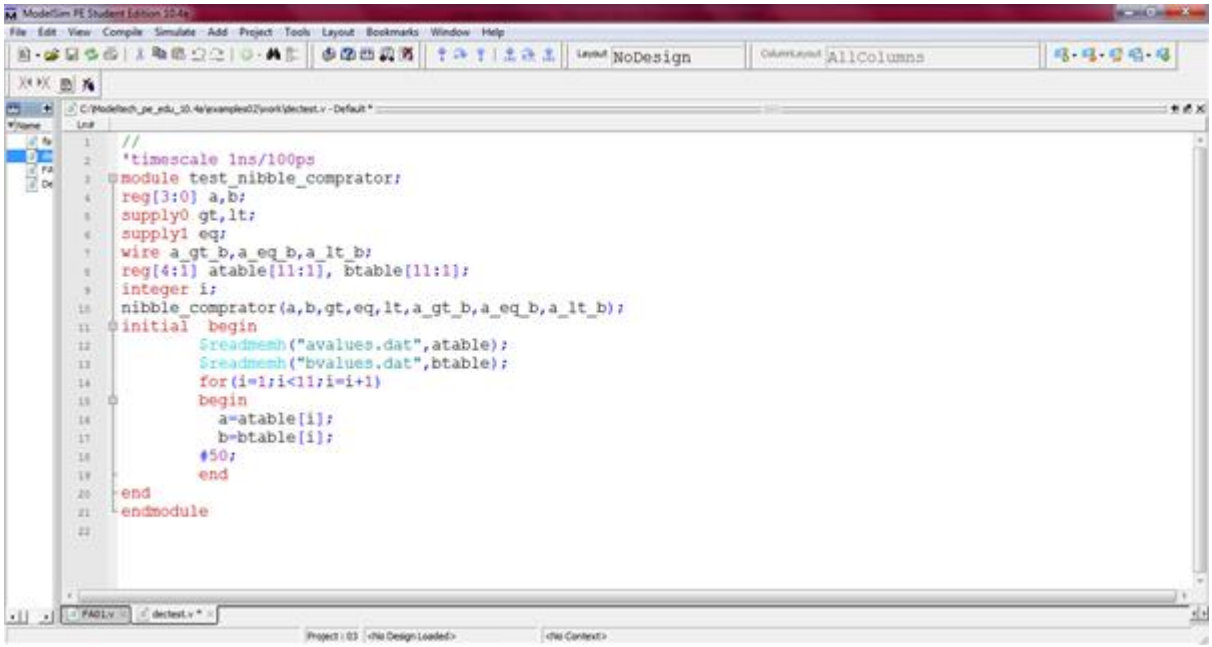
[۱] function

**Task**: دو نوع task داریم:

۱- سیستمی که task های موجود سیستمی را فراخوانی می‌کنیم.
۲- کاربر: که توسط برنامه نویس تولید و فراخوانی می شود.

**روش دیگر نوشتن تست nibble-comparator**: یک فایل(notepad) برای مقادیر a ایجاد و نام avalues.dat را می‌گذاریم. همچنین یک فایل برای مقادیر b ایجاد و نام bvalues.dat را می‌گذاریم. حالا یک تست می‌نویسیم که این دو فایل را صدا بزند.




برای ذخیره فایل‌ها می‌توان از پسوندtxt. نیز استفاده کرد.

- لازم نیست که آرایه‌های table را داخل کد دقیقا هم اندازه فایل‌های اصلی تعریف کنیم، می‌توان یک فایل values هزارتایی داشت ولی فقط ۱۰ تای آنرا در قالب table استفاده کنیم.

- Readmemh(file name,array name,start index,end index)
- Index ها در readmemh اختیاری اند، اگر گذاشته نشوند کل فایل values خوانده می‌شود.

**ساختار task**:

Task اسم کار;

Task تعریف ورودی و خروجی;

Begin:name

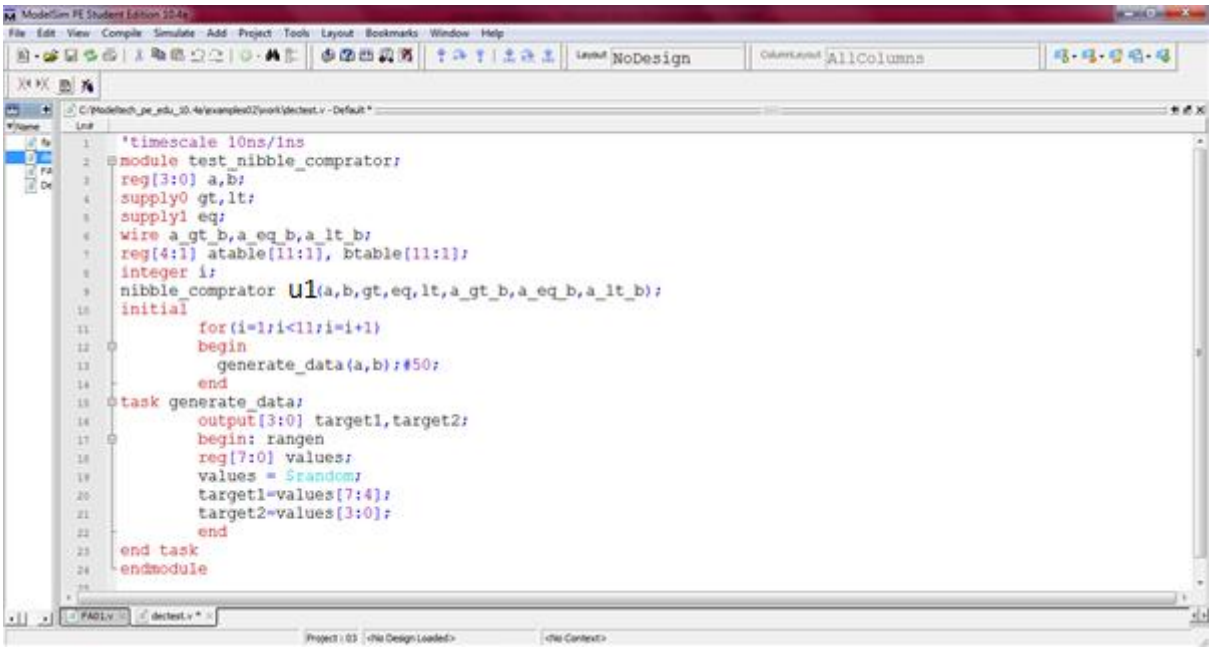
taskبدنه

End

End task

**مثال:ایجاد یک تست برای nibble-comparator به طوری که اعداد تصادفی را خود سیستم برای ما تولید کند.**

تولید ۱۱ عدد تصادفی با task تعریف شده توسط کاربر:



```
'timescale 10ns/1ns
module test_nibble_comprator;
reg[3:0] a,b;
supply0 gt,lt;
supply1 eq;
wire a_gt_b,a_eq_b,a_lt_b;
reg[4:1] atable[11:1], btable[11:1];
integer i;
nibble_comprator u1(a,b,gt,eq,lt,a_gt_b,a_eq_b,a_lt_b);
initial
        for(i=1;i<11;i=i+1)
        begin
          generate_data(a,b);#50;
        end
task generate_data;
        output[3:0] target1,target2;
        begin: rangen
        reg[7:0] values;
        values = $random;
        target1=values[7:4];
        target2=values[3:0];
        end
end task
endmodule
```

- می‌توان task را در فایلی به اسم f1.v بنویسیم و سپس در بالایinitial با کمک module f1.v‘ فراخوانی اش کنیم.
- می‌توانستیم به جای تعریف متغیر values یکجا به target ها با کمک random مقدار دهیم.

## تمرین ۲:

۱. یک تابع برای ایجاد نقلی خروجیِ یک تمام جمع کننده (FA) بنویسید.

۲. یک تابع برای ایجاد sumخروجیِ یک تمام جمع کننده بنویسید.

۳. با استفاده از عبارات بولی ، یک تابع به نام inc-bit را بنویسید که ورودی آن ۴ بیتی و به مقدار ورودی یک واحد افزایش دهد.

## فصل ۶ تعریف انواع

انواع پورتها می‌تواند input, output, inout باشد .برای تعریف پورت‌های ورودی و خروجی از reg نیز می‌توان استفاده کرد. مقدار نوع reg را می‌توان به inout انتساب داد.

```
input a1;
Input  a2;
A1=a2;
```

تعریف پارامتر:

```
Parameter delay 5;
```

برای تعریف زمان، توان و اندازه می‌توان استفاده کرد.

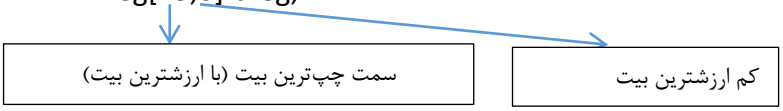
تبدیل نوع: اگر یک متغیر integer از real مقدار بگیرد به نزدیکترین عدد صحیح گرد می‌شود. نوع reg را نیز می‌توان در real قرار داد ولی در real مقادیر x,z نداریم، به جای آنها صفر می‌گذاریم.

```
Real a1;
Rega2='x';
A1=a2;
A1='0';
```

نوع دیگر :signal , wor,wire , :برای تعریف سیم و متغیرهای میانی

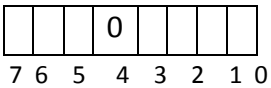
تعریف بردار(آرایه):

```
Wor[7:0] abus,bbus;
Reg[15,0] breg;
```

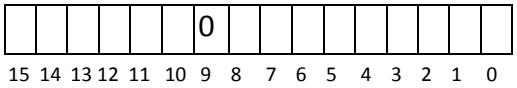



مقدار دهی: طبق ادامه مثال بردار:

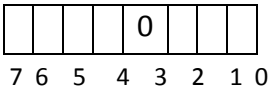
```
Abus[4]=1'b0
```




```
Bbus[3]=breg[9];
```




breg



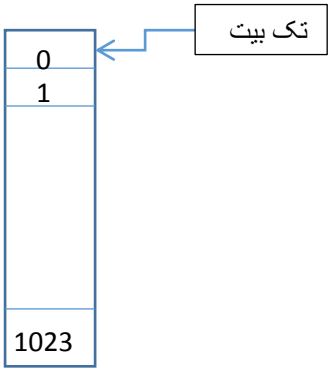

bbus

اگر از محدوده خارج شویم x برمیگردد.

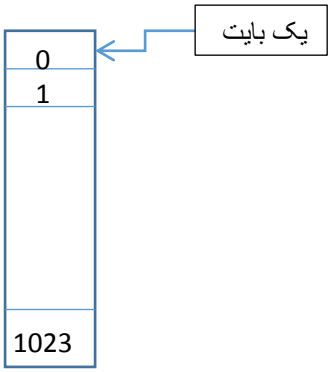
حافظه:

Reg[word reg]  mem[memory reg]

Ex)reg mem1k[0:1023]




Ex)reg[7:0]  mema[0:1023]




میتوان با اندیس به خانه های حافظه اشاره کرد، اما نمیتوان چند کامه حافظه را در یک زمان استفاده کرد: چون حافظه فقط یک گذرگاه داده دارد:

| | |
|---|---|
| Mem1k[10] | نوشتار درست هست |
| Mem1k[10:20] | نوشتار غلط دسترسی به چند خانه حافظه بطور همزمان |
| Mema[100] | نوشتار درست هست |
| Mema[100:101] | نوشتار غلط دسترسی به چند خانه حافظه بطور همزمان |
| Bbus=mema[20]; | محتوای خانه ۲۰ حافظه mema در bbus قرار می‌گیرد |

- در مقدار دهی، اگر متغیر کوچکتر در بزرگتر،قرارگیرد، بخش پر ارزش، صفر می‌شود.
- اگر متغیر بزرگتر در کوچکتر قرار گیرد، بخش پرارزشتر برش می‌خورد.

```
Wire[3:0] a;
Wire [5:0] aa;
```

اگر مقدار اولیه aa=001110 در اینصورت:

Assign a=aa;>>>>>>>>>a=1110

اگر مقدار a=0110 آنگاه

Assign aa=a;>>>>>>>>>>aa=000110

## فصل ۷: طراحی مدارات در سطح رفتاری

در مواردی مثل توابع –کارها-برنامه های تست –ثبات ها وماشین های حالت می توان از سطح رفتاری استفاده کرد.

در سطح رفتاری بدنه به صورت زیر روال (procedure)می باشد و به صورت زیر است:

| | |
|---|---|
| Always<br>Begin<br>دستورات<br>end | Always<br>یک دستور |
| initial<br>Begin<br>دستورات<br>end | initial<br>یک دستور |

رفتاری یا باید داخل always, initial باشد تمام تستها رفتاری بودند.

دستورات initial یکبار تکرار و دستورات always بینهایت تکرار می شود(چند بار). دستورات داخل بدنه initial و always خط به خط اجرا می شوند (ترتیبی) ودر زمان صفر شروع به اجرا می کنند اگر تاخیر نداشته باشند. اما چند بلوک initialو always به صورت همروند اجرا می‌شوند.

مثال:

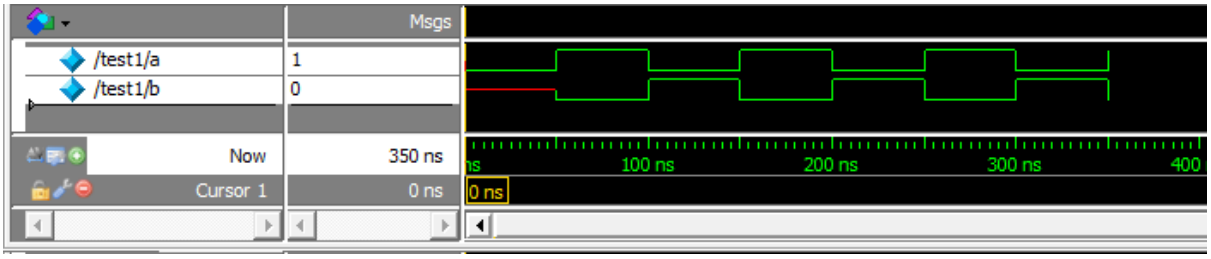
```
Always                                          A=0,b=1
    Begin
    #5;
    a=~a;
    b=~b;
    end
```

---

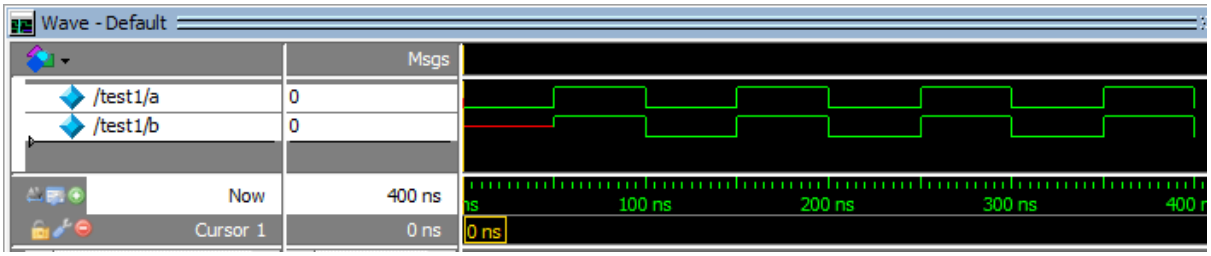
```
    initial
    begin
    #5
    a=~a;
    b=~b;
    end
```

---

```
1   `timescale 10ns/1ns
2   module test1;
3     reg a,b;
4   initial a=0;
5   always
6   begin
7   #5
8   a=~a;
9   b=~a;
10  end
11  endmodule
```



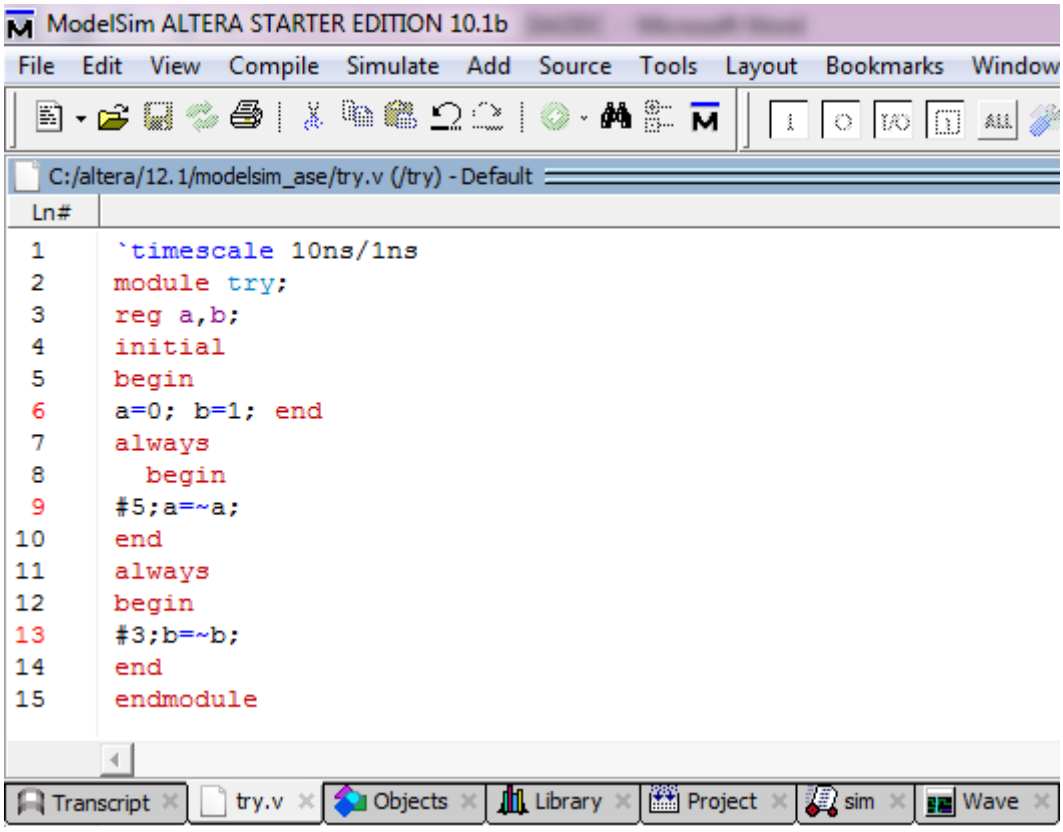

```
1   `timescale 10ns/1ns
2   module test1;
3     reg a,b;
4   initial a=0;
5   always
6   begin
7   #5
8   a<=~a;
9   b<=~a;
10  end
11  endmodule
```

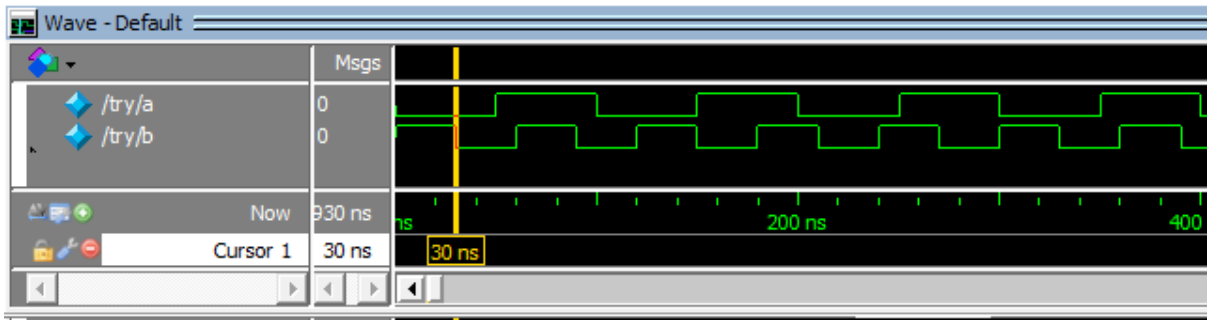



---

‘timescale 1ns/100ps
initial
begin
#1
a<= #4 0;
b<= #2 1;
end

---




دستورات کنترل زمانی:

#Time : به اندازه time صبر می کند.

#(time-experssion): به اندازه عبارت t-e صبر می کند.

@Variable: منتظر تغییر متغیر می ماند.

@(Variable or expression): منتظر تغییر متغیردر عبارت یا متغیر می ماند.

@(posedge Variable or expression): منتظر تغییرصفر به یک عبارت یا متغیر می ماند.

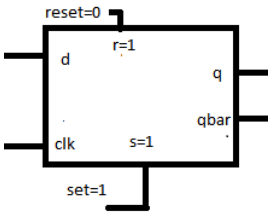
@(negedge Variable or expression): منتظر تغییر یک به صفر عبارت یا متغیر می ماند.

@(event1 or event2 or ...):منتظر تغییر یکی از رخدادها می ماند.

Wait(expression):تا زمانی که عبارت غلط است منتظر می ماند.

Wait(a): تا زمانی که a=0است منتظر است به محض اینکه a=1 شود از انتظار خارج میشود.

**مثال سطح رفتاری ff-sr:**




```
1   `timescale 1ns/100ps
2   module d_sr_ff(d,set,rst,clk,q,qb);
3   input d,set,rst,clk;
4   output q,qb;
5   reg q,qb,state;
6   parameter sq_delay=6,rq_delay=6 ,cq_delay=6;
7   always@ (posedge clk or posedge rst or posedge set)
8   begin
9   if(set)begin
10    state=#sq_delay 1;
11  end else if(rst)
12  begin
13    state=#rq_delay 0;
14  end else if(clk)
15  begin
16    state=#cq_delay d;
17  end
18  end
19  always@( state)
20  begin
21    q=state   ;
22    qb=~state ;
23  end
24  endmodule
```

طریقه نوشتن case:

درتوابع و تسک ها وalwaysوinitialقابل استفاده است.

```
case(expression)
1:statement;
2:statement ;
…
N  :statement ;
defult  :statement ;
endcase
```

در caseعبارت می‌تواند مقادیر صفر ویک داشته باشد.

اگر ازx case استفاده کنیم مقادیرz, x نیز اضافه می‌شود.

اگر از z case استفاده کنیم مقدار z نیز اضافه می‌شود.

نوشتن حلقه:۴روش برای نوشتن حلقه وجود دارد:

۱: استفاده از forever:

```
Initial forever          always
Begin            =       begin
بدنه یا چند دستور        بدنه یا چند دستور
End                      End
```

۲: استفاده از(repeat(expression

```
always
begin
 repeat(5)@(posedgeclock);//5 iterations delay
areg=abus;
end
```

عبارت (posedge clock)@، ۵ بار تکرار می‌شود.

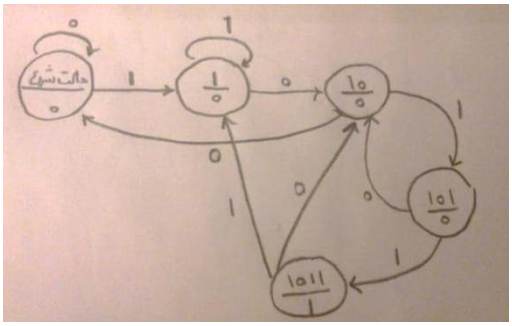
۳:حلقه while:

```
While(experssion)
Begin
…
end
```

۴: حلقه for:

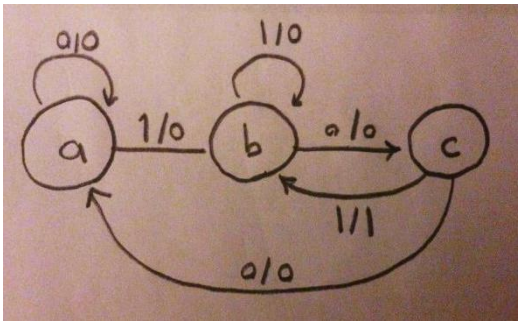
```
for( initial_assignement; expression; step_assignment   )
Begin
…
end
```

**مثال: توصیف رفتاری یک تشخیص دهنده رشته ۱۰۱۱ رابنویسید؟**با استفاده از ماشین حالت دیاگرام تشخیص این رشته را رسم می‌کنیم. سپس کد مربوط به آن را می‌نویسیم. (مدل مور تغییر روی خروجی به حالت وابسته است بنابراین با تغییر کلاک خروجی تغییر می یابد).



```
`timescale 1ns/100ps
module moore_detector(x,clk,z);
input x,clk;
output z;
reg[2:0] current; reg z;
parameter[2:0]reset=0,got1=1,got10=2,got101=3,got1011=4;
initial current = reset;
always@(posedge clk)
                        case(current)
                          reset: begin
                                 if(x==1)       current = got1;
                                 else           current = reset;
                               z=0;
                        end
   got1: begin
                        if(x==0)   current = got10;
                            else      current = got1;
                         z=0;
                        end
   got10:begin
                          if(x==1)   current = got101;
                          else       current = reset;
                          z=0;
                        end
   got101:begin
                           if(x==1)
                           begin
                                  current = got1011;
                                  z=1;
                            end else
                               begin
                                 current = got10;
                                   z=0;
                                end
        end
   got1011:begin
        if(x==1)  current=got1;
        else current=got10;
          z=0;
       end
  endcase
endmodule
```

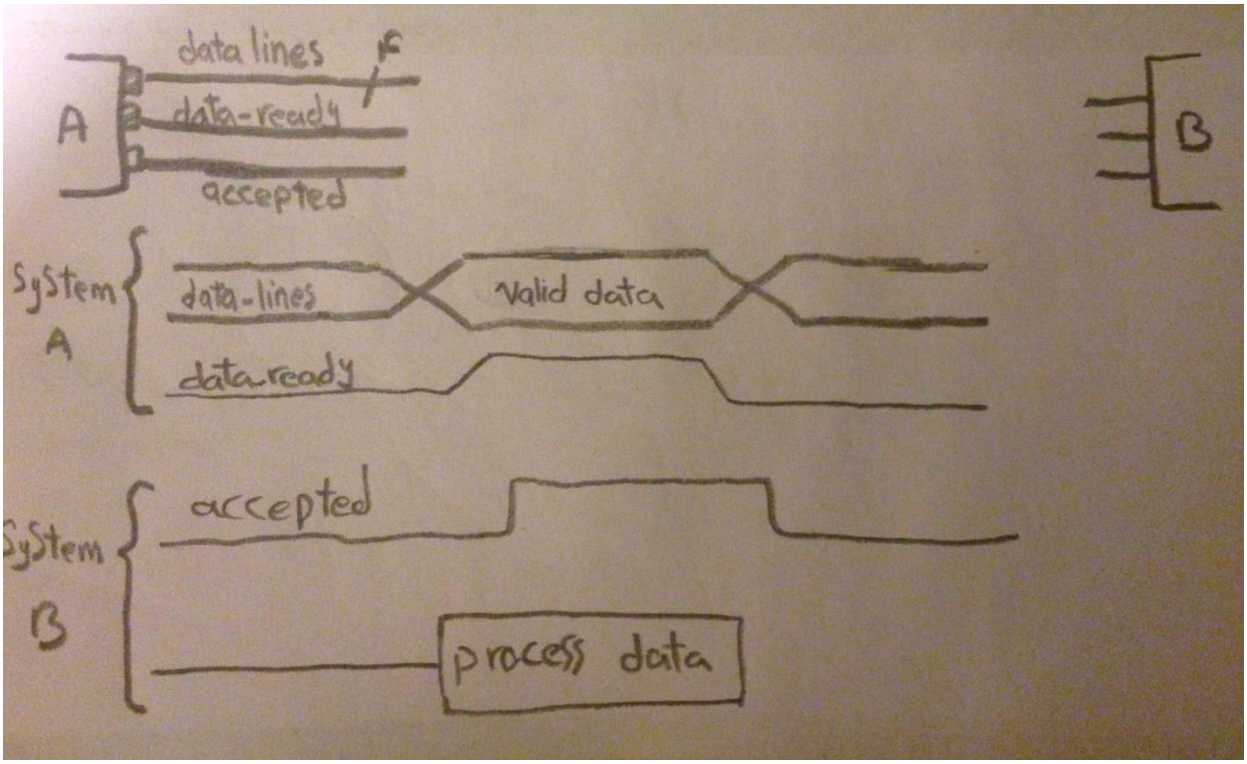
**مثال: توصیف رفتاری تشخیص دهنده ۱۰۱ به مدل میلی:** در مدل میلی خروجی وابسته به ورودی است. در این مدل تنها با تغییر حالت خروجی عوض نمی‌شود بلکه با تغییر در ورودی بدون وابستگی به کلاک تغییر اعمال می‌شود.




```
Ln#
 1  `timescale 1ns/100ps
 2  module asynch_reset_detector(x,r,clk,z);
 3  input x,r,clk;
 4  output z;
 5  reg[1:0] nxt,present;
 6  reg z;
 7  parameter a=0, b=1, c=2;
 8  initial nxt = a;
 9  always @(posedge clk or posedge r)
10   begin:registring
11    if(r==1) present = a;
12    else present = nxt;
13    end
14  always @(present or x)
15   begin:combinational
16    z=0;
17    case(present)
18    a:
19      if(x==0) nxt = a;
20      else nxt = b;
21    b:
22      if(x==0) nxt = c;
23      else nxt = b;
24    c:
25      if(x==0) nxt = a;
26      else nxt = b;
27    default:
28     nxt = a;
29    endcase
30  if(present == c && x==1) z=1;
31  end
32  endmodule
```

**مثال: پیاده سازی دست تکان دهی**[1] : ارتباط غیر همزمان[2] بین سیستم‌ها توسط دست تکان دهی انجام می‌شود. دست تکان دهی یعنی سیگنالهایی که بین دو سیستم انجام می‌شود. وقتیکه یک سیستم داده هایی را برای انتقال به سیستم دیگر آماده می‌کند، فرستنده گیرنده را آگاه می‌کند که داده آماده است. وقتیکه گیرنده داده ها را می‌پذیرد به فرستنده دریافت آنها را اطلاع می‌دهد.

[1] handshaking
[2] asynchronous



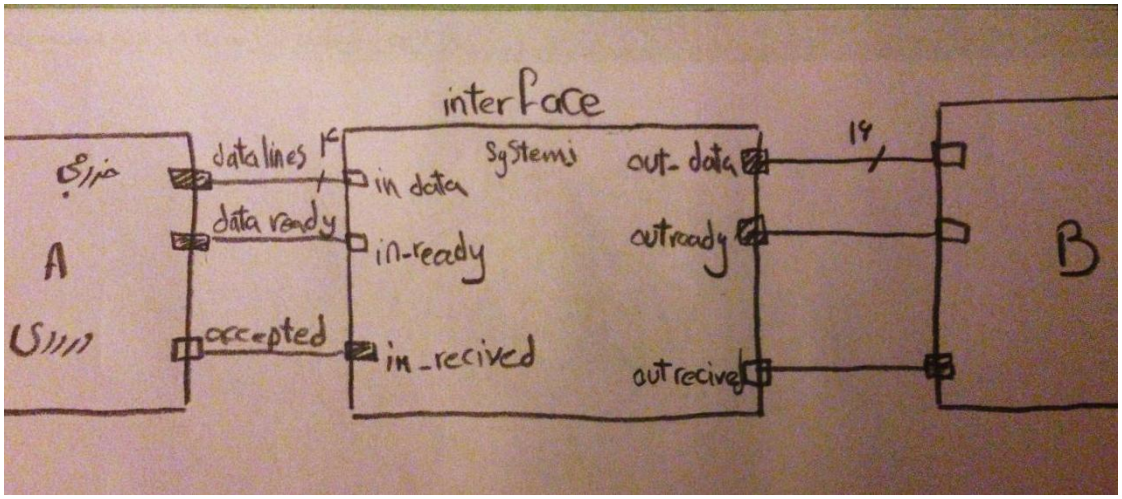

سیستمA بعد از آماده شدن عملیات زیر را انجام می دهد:

- Data-lines= داده های آماده شده روی آن قرار داده می‌شود.
- Data-ready=1
- منتظر لبه مثبت acceptedمی شود.
- Data-ready=0

سیستم B:

- منظر لبه مثبت data-ready می شود .
- Accepted=1
- داده ها را پردازش می کند.
- منتظر لبه منفی data-ready می شود .
- Accepted=0

**مثال: فرض کنید سیستم Aدارای خط داده ۴ بیتی وسیستم B دارای خط داده ۱۶ بیتی است یک مدار رابط طراحی کنید که به روش دست تکان دهی بین این دو ارتباط برقرار کند؟**




```
`timescale ۱ns/۱۰۰ps
module system_i(in_data,out_data,in_ready,out_ready,in_recieved,out_recieved);
input[۳:۰] in_data;         input in_ready,out_recieved;
output[۱۵:۰] out_data;      output out_ready, in_recieved;
reg[۱۵:۰] out_data;         reg out_ready,in_recieved;
reg[۳:۰] word_buffer[۳:۰]; reg buffer_full,buffer_picked;
reg[۱:۰] i;
initial begin i=۰; end
always begin: atalk
  @(posedge in_ready);
  word_buffer[i] = in_data;
  if(i==۳) begin
     buffer_full = ۱;
  @(posedge buffer_picked);
  buffer_full = ۰;
  i = ۰;
  end else i=i+۱;
  in_recieved = ۱;
  @(negedge in_ready);
  in_recieved = ۰;
  end
always begin: b_talk
  wait(buffer_full);
  out_data={word_buffer[۳],
      word_buffer[۲],
      word_buffer[۱],
      word_buffer[۰]};
  buffer_picked = ۱;
  @(negedge buffer_full);
    buffer_picked = ۰;
    out_ready = ۱;
  @(posedge out_recieved);
    out_ready = ۰;
  end
endmodule
```

برنامه تست برای مساله دست تکان دهی:

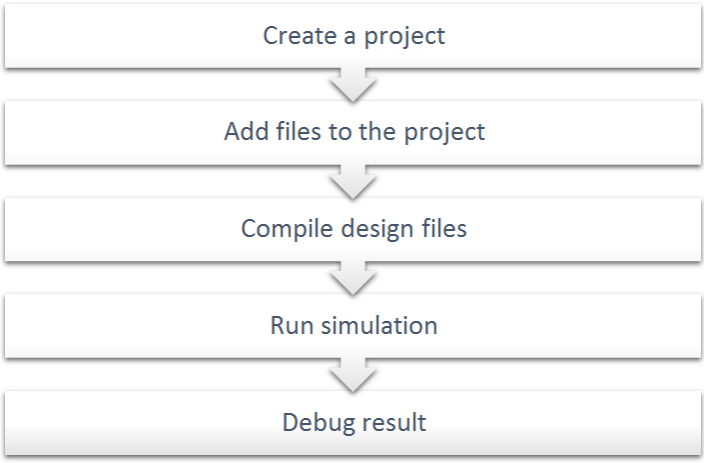
```
`timescale ۱ns/۱ns
module test_system_interface;
  reg [۳:۰] a_data;
  reg a_ready;
  wire a_received;
  wire [۱۵:۰] b_data;
  wire b_ready;
  reg b_received;
  integer i,seed;
  system_i u۱(a_data, b_data, a_ready, b_received, a_received, b_ready);
  initial begin
    for (i=۱;i<=۱۵; i=i+۱) seed<=#(i*۳) i;
    a_ready=۰;b_received=۰;
    #۴۷ $stop;
  end
  //this models system_a
  always begin
    a_data=$random (seed);
    repeat (۴) begin
      #۲ a_data={~a_data[۰],a_data[۳:۱]};
      a_ready=۱;
      @(posedge a_received);
      #۱ a_ready=۰;
    end
  end
  //this models system_b
  always begin
    @(posedge b_ready);
    #۱ b_received=۱;
    @(negedge b_ready);
    #۲ b_received=۰;
    end
  endmodule
```

ضمیمه:

قصد داریم دو مثال را توسط نرم افزار modelsim پیاده سازی کنیم تا شما نحوه ی کار با این نرم افزار را نیز بیاموزید . پس در ابتدا به سراغ آموزش نرم افزار modelsim می رویم .

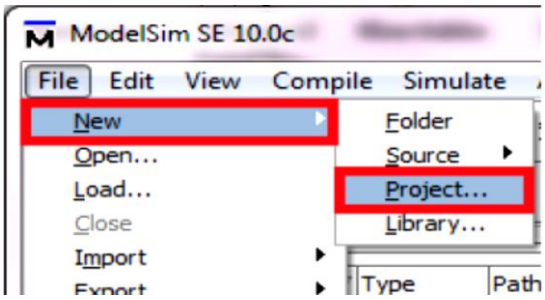
## ✓ آموزش modelsim :

مراحلی که برای پیاده سازی و اجرا گرفتن از یک کد باید طی کنیم به شکل زیر است :

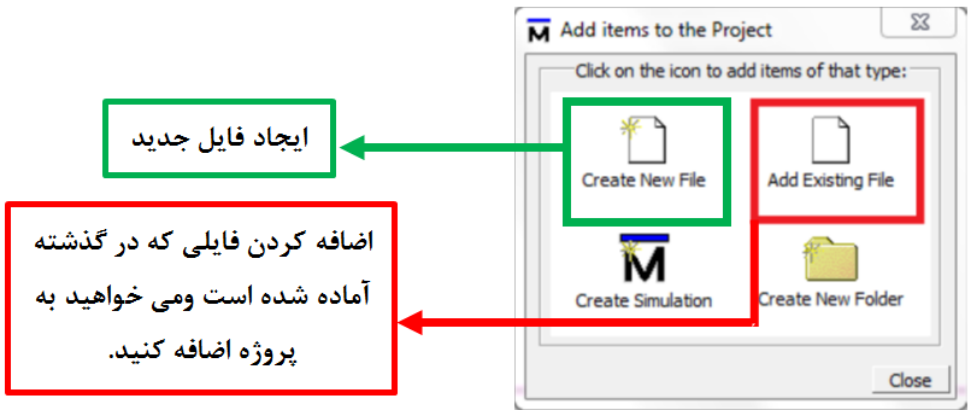



برای ایجاد پروژه جدید به صورت زیر عمل کنید:

File ⇨ New ⇨ Project

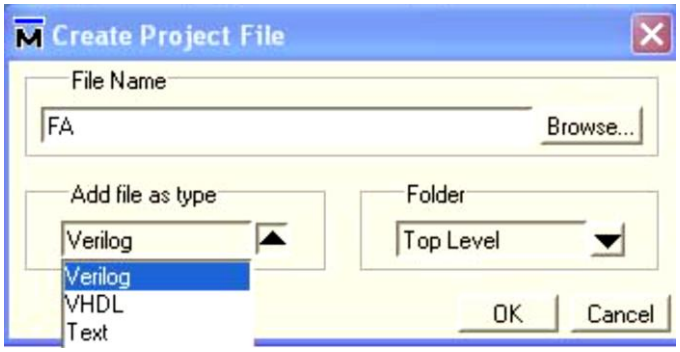



سپس پنجره ای باز خواهد شد که در قسمت Project Name ، نام پروژه و در قسمت Location محل ذخیره پروژه را مشخص می کنید و بعد دکمه OK را می زنید و پنجره زیر باز می شود :

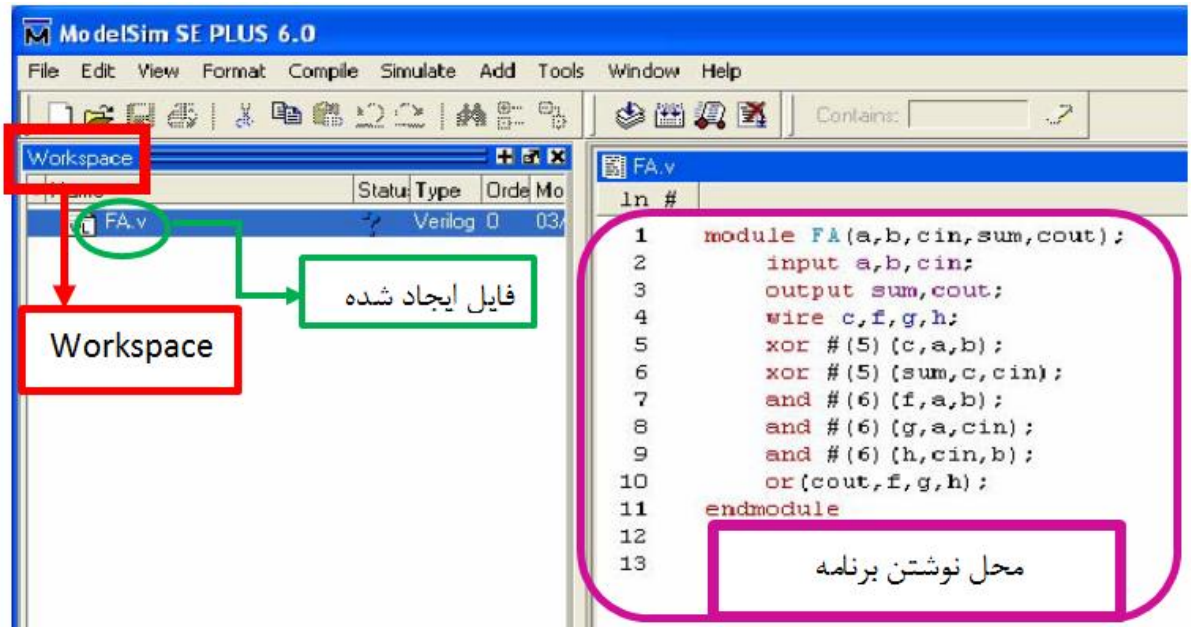



### ۱. انتخاب Create New File :

بعد از کلیک بر روی Create New File صفحه زیر باز می شود که در آن زبان برنامه نویسی و نام فایل خود را انتخاب می کنید . اگر زبان برنامه نویسی خود را VHDL انتخاب کنید ، فایلی با پسوند .vhd ایجاد می کند و اگر زبان Verilog را انتخاب کنید ، فایلی با پسوند .v ایجاد می کند که ما با زبان Verilog کار می کنیم .

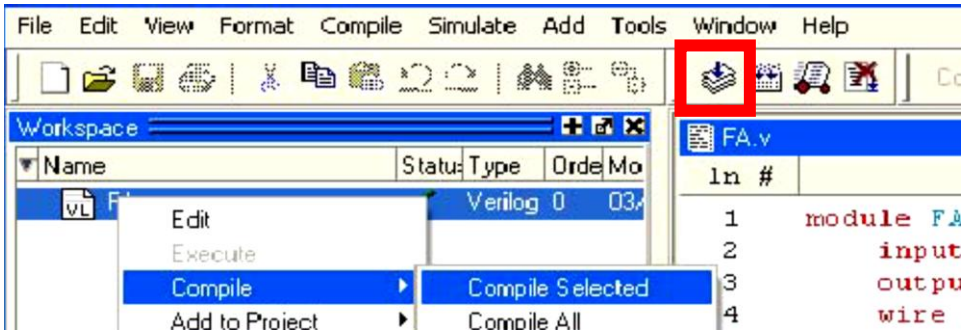



بعد از OK کردن ، فایل شما در Project ⇨ WorkSpace اضافه می شود. حال با دوبار کلیک بر روی نام فایل ( مثلا FA.v ) ، صفحه ای باز می شود که کد را داخل آن می نویسیم و در انتها Save می کنیم .

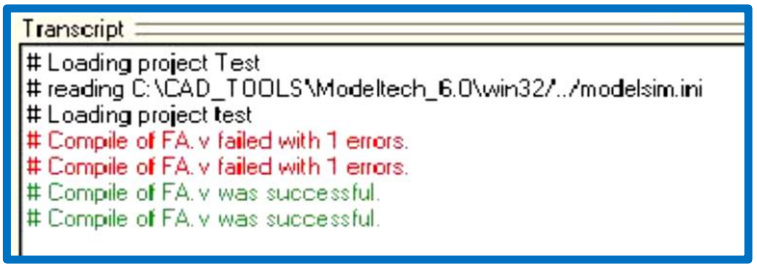



در مرحله بعد باید کد خود را Compile کنیم تا از نظر اصول برنامه نویسی Verilog یا VHDL چک شود. در صورتی که کد شما اشکالی نداشته باشد ، شما را با پیغامی که در Transcript می گذارد ، آگاه میکند.

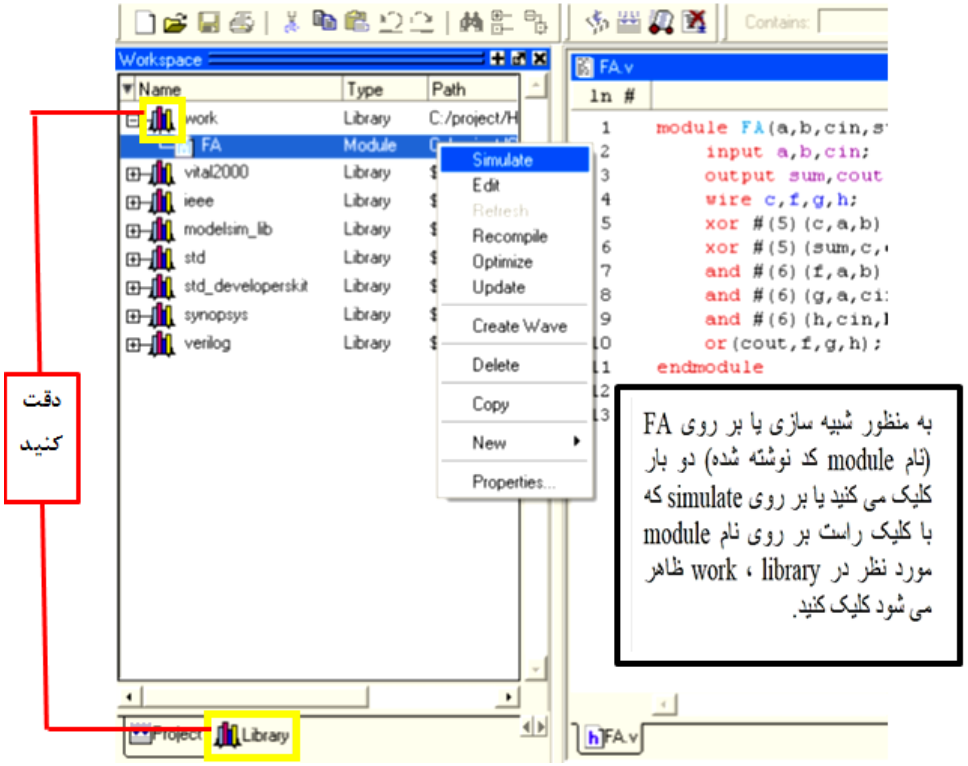
برای این کار دو راه دارید ، می توانید روی فایل راست کلیک کرده و گزینه ی Compile را بزنید و یا گزینه Compile را از نوار ابزار انتخاب کنید :



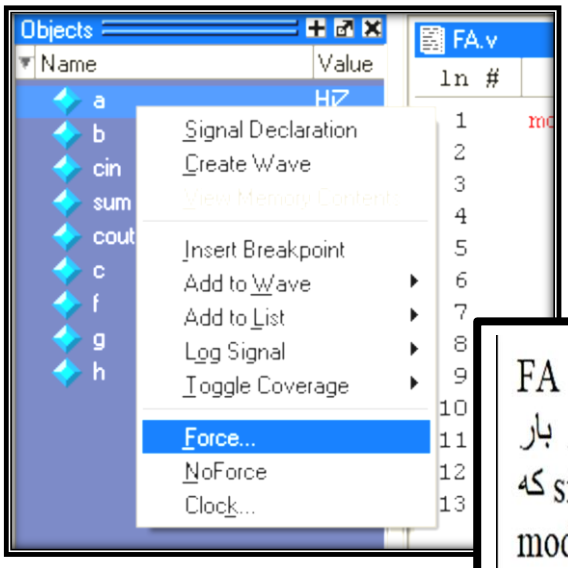

اگر Compile شما موفقیت آمیز باشد در قسمت Transcript به رنگ سبز نشان داده می شود و اگر خطا داشته باشد به رنگ قرمز نشان داده می شود که اگر روی آن دو بار کلیک کنید می توانید از نوع و محل خطا آگاه شوید .



بعد از عمل **Compile** نوبت **Simulation** است .



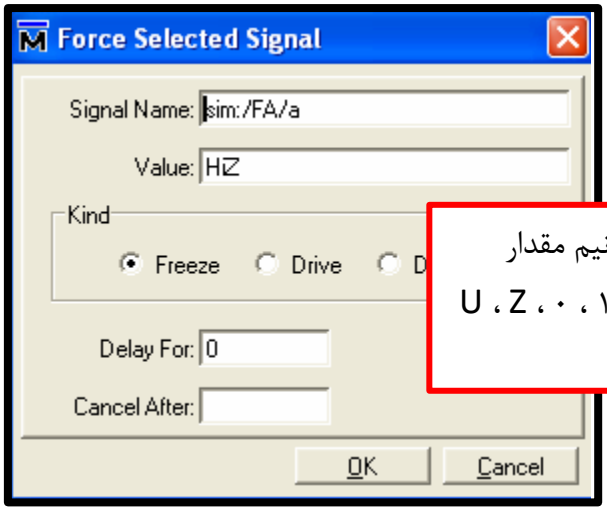

حال نوبت آن است که Port های ورودی را مقداردهی کنیم . با کلیک راست بر روی هر کدام از object های موجود و با انتخاب Force و یا Clock می توانیم آن ها را مقداردهی کنیم .(در بعضی versionها باید روی object کلیک راست کنید و بعد wave را بزنید و بعد در پنجره باز شده دوباره روی object کلیک راست کرده و سپس Force و یا Clock را انتخاب می کنیم .)




دقت کنید

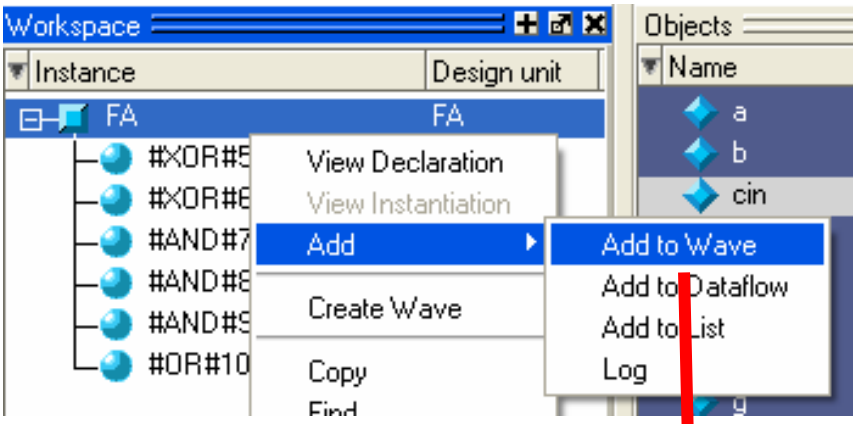
به منظور شبیه سازی یا بر روی FA (نام module که نوشته شده) دو بار کلیک می کنید یا بر روی simulate که با کلیک راست بر روی نام module مورد نظر در work ، library ظاهر می شود کلیک کنید.

یر باز می شود :



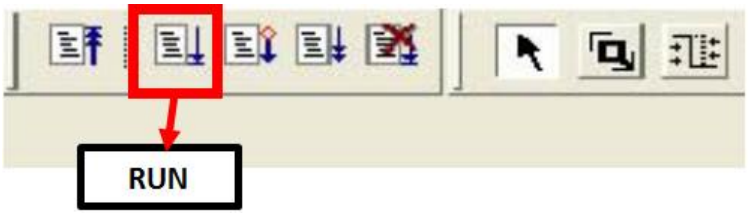

بعد از اینکه object های ورودی را مقداردهی کردید ، نوبت آن است که موج خروجی را ببینید . برای این کار به این روش عمل کنید : راست کلیک روی نام فایل --» Add --» Add to Wave

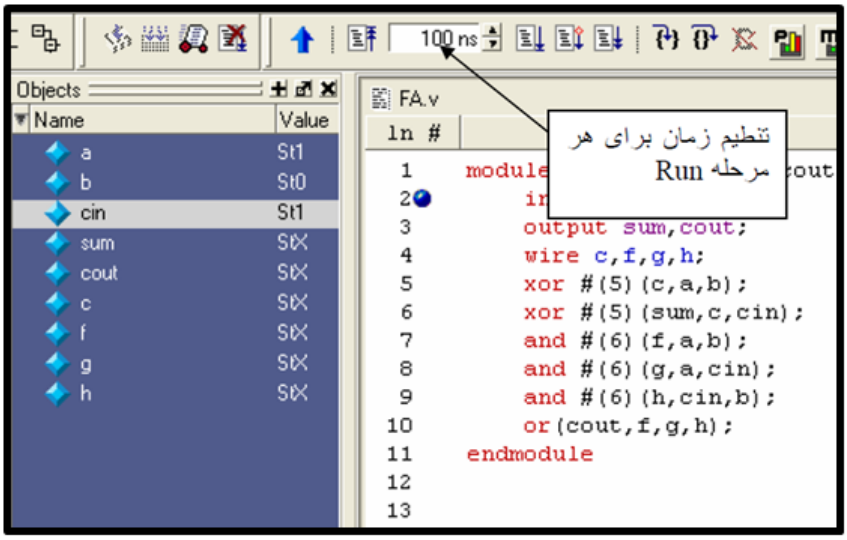
که با کلیک بر روی Add to Wave ، پنجره Wave باز می شود .



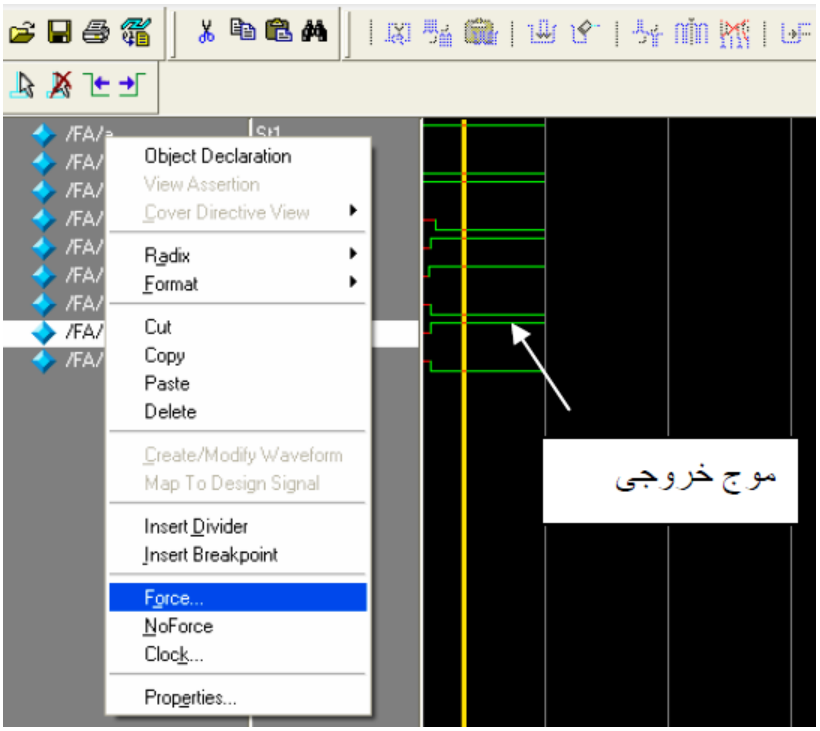

**پنجره ی Wave**

حال با کلیک بر روی Run ، موج خروجی را به اندازه ی زمانی که تعریف شده است ( به طور پیش فرض 100ns ) رسم می کند و با فشار مجدد ، به اندازه آن زمان دوباره رسم می کند .



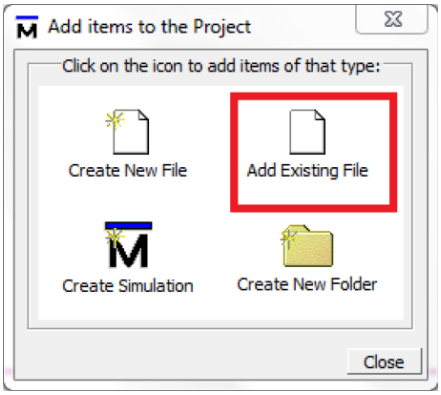




اگر در زمان Run خواستید مقدار یک سیگنال را تغییر دهید می توانید بر روی آن در صفحه Wave کلیک راست کنید و Force را انتخاب کنید .




## ۲. انتخاب Add Existing File :

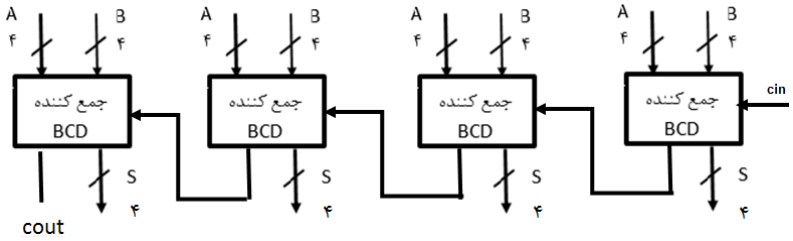
همان طور که قبلا گفتیم زمانی که یک پروژه ایجاد میکنیم ، پنجره زیر باز میشود :




که در قسمت قبل انتخاب Create New File را به طور کامل توضیح دادیم. زمانی از گزینه ی Add Existing File استفاده میکنیم که بخواهیم برنامه هایی که قبلا نوشته ایم را به پروژه اضافه کنیم و از آنها استفاده کنیم تا مجبور نباشیم یک بار دیگر آن ها را بنویسیم. با کلیک بر روی Add Existing File ، پنجره ی Add File to Project باز میشود ، Browse را می زنیم و برنامه هایی را که قبلا نوشتیم که پسوند v. دارند را انتخاب کرده و OK میکنیم .

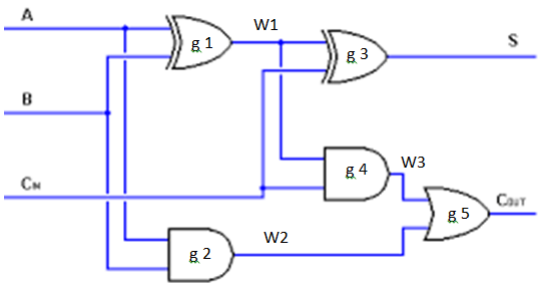
حال در پنجره Project فایل ها را می بینید که در قسمت statues آن ها علامت سوال وجود دارد که یعنی هنوز Compile نشده اند . همان طور که قبلا هم توضیح داده بودیم ، آنها را Compile کرده و باقی مراحل هم(... , simulation) مانند قبل می باشد .

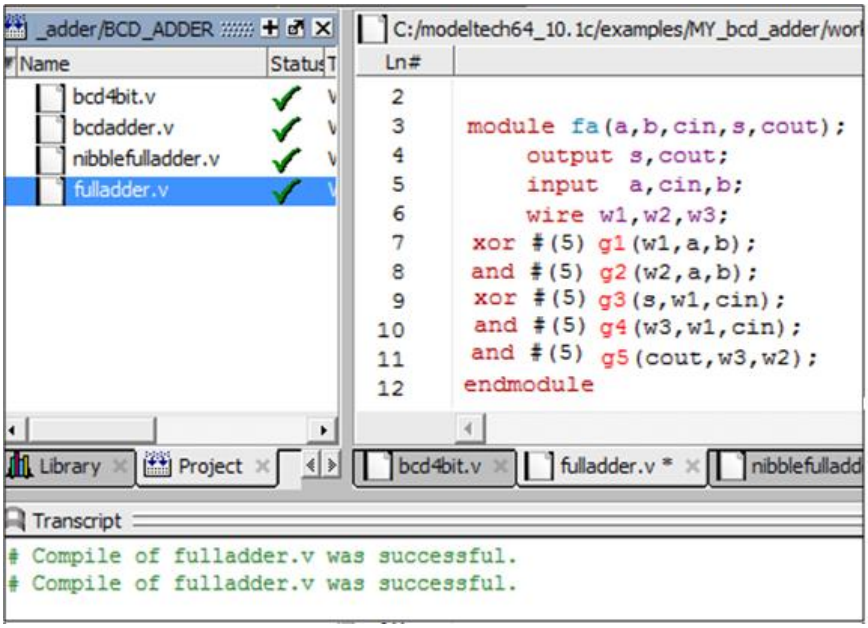
## مثال ۱ ) یک مدار طراحی کنید که دو عدد ۴ رقمی BCD را با هم جمع کند .



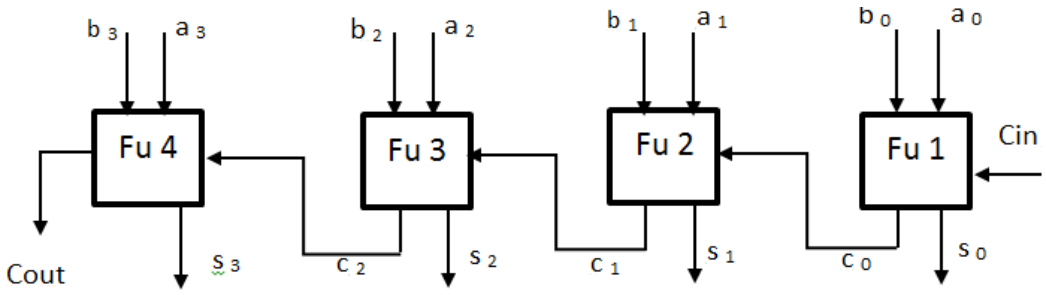

حل ) برای حل این سوال ، باید ۴ فایل ایجاد کنیم :

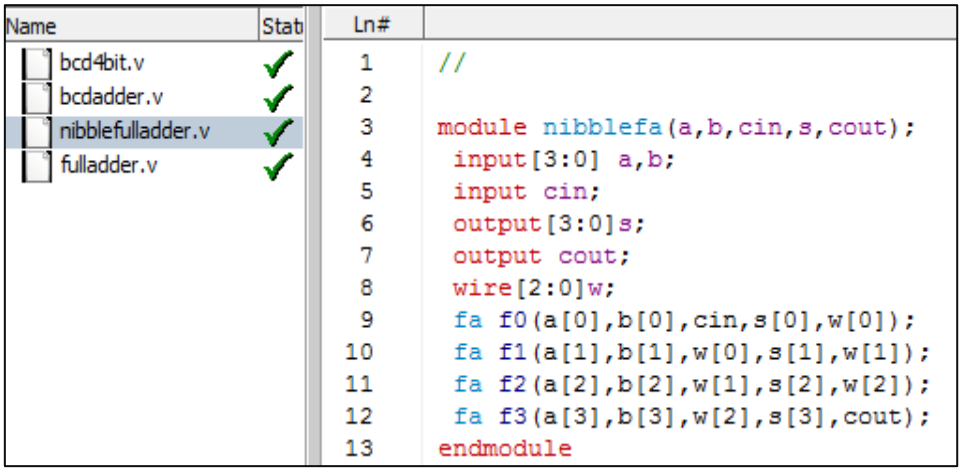
۱- Full Adder:



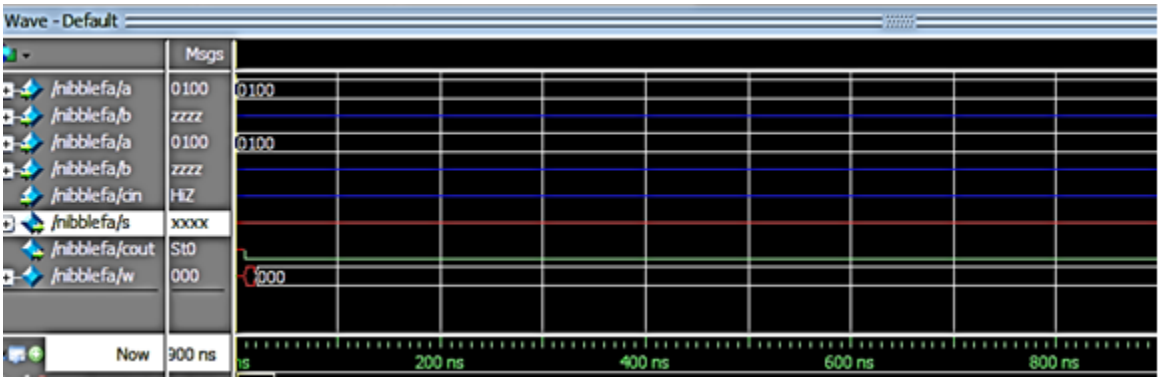



۳- جمع کننده ۴ بیتی ( با استفاده از ۴ عدد Full Adder )

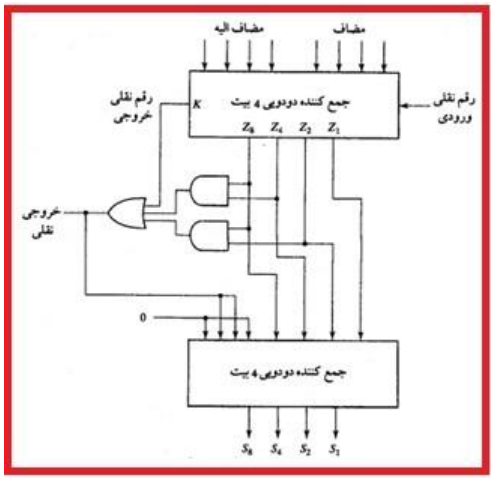


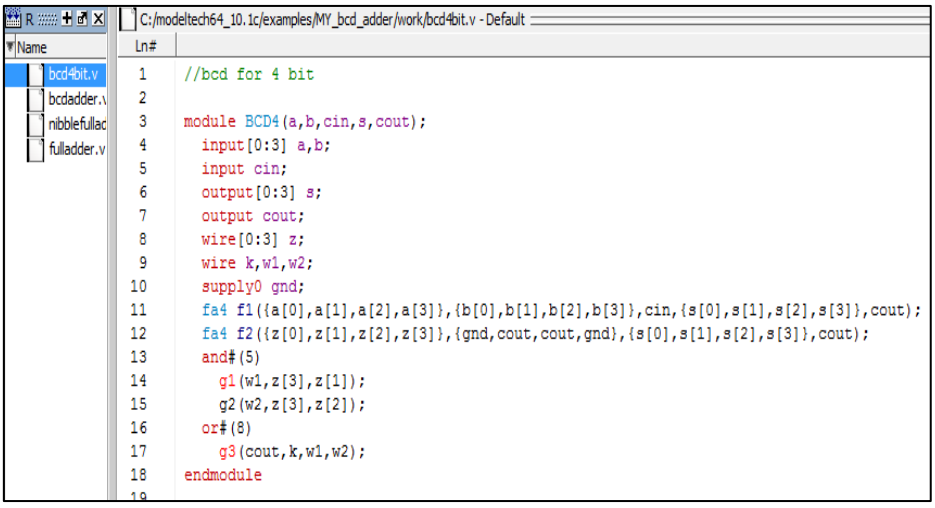




Simulate جمع کننده ۴ بیتی

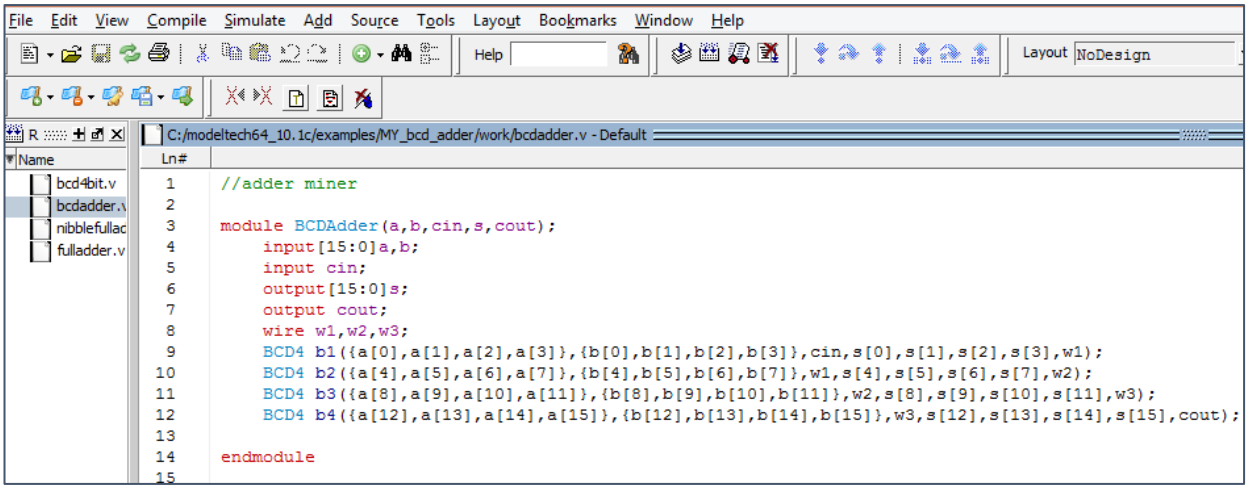



۳- جمع کننده BCD ۴ بیتی



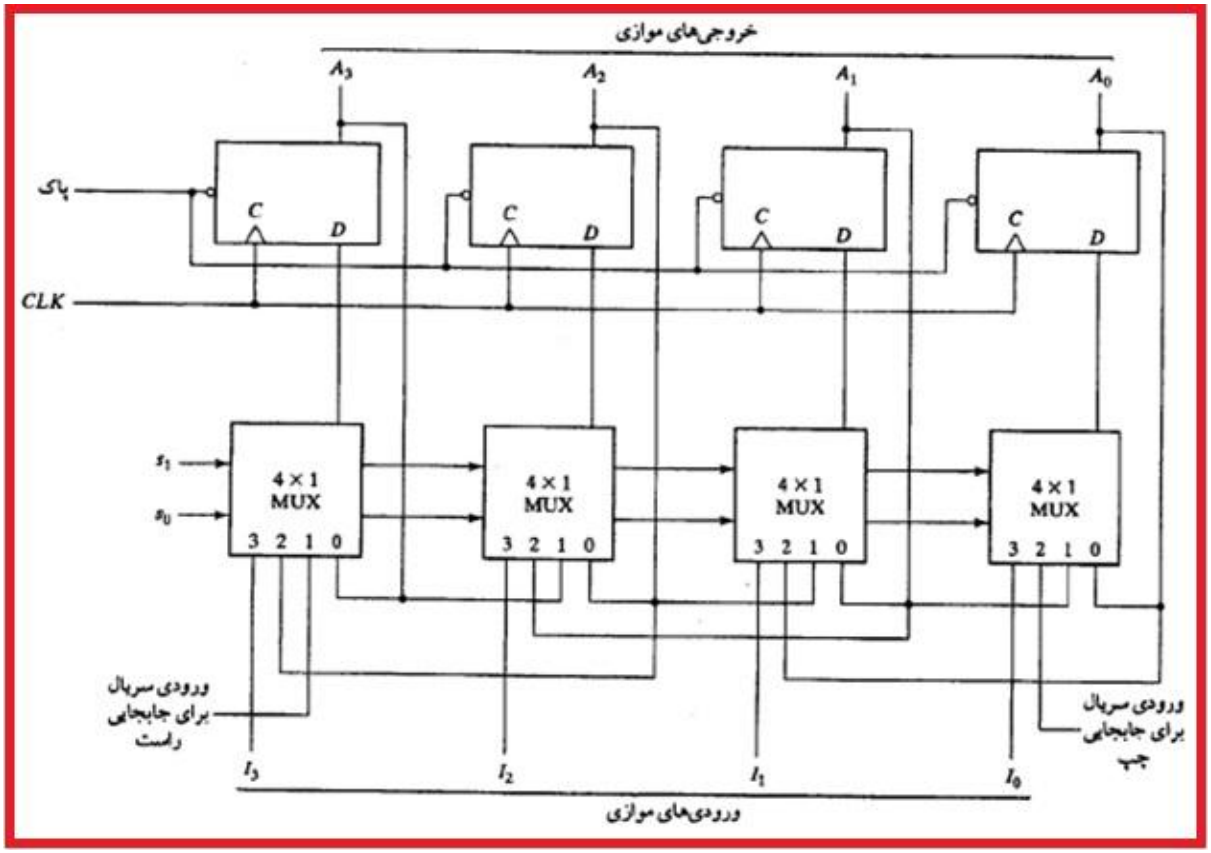




۴- جمع کننده BCD ۱۶ بیتی ( با استفاده از ۴ عدد جمع کننده BCD ۴ بیتی ) ( شکل صورت مثال )




**مثال ۲ ) یک شیفت رجیستر ۸ بیتی با قابلیت بار موازی طراحی کنید که قابلیت های شیفت را داشته باشد .**

| S1 | S0 | عمل |
|---|---|---|
| 0 | 0 | بلا تغییر |
| 0 | 1 | شیفت راست |
| 1 | 0 | شیفت چپ |
| 1 | 1 | بار موازی از ورودی |

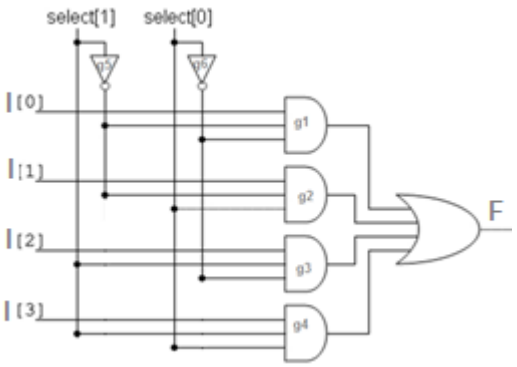



**برای حل این مثال باید سه فایل ایجاد کنیم :**

۱. مالتی پلکسر ۱*۴
۲. D فلیپ – فلاپ
۳. Shift – register

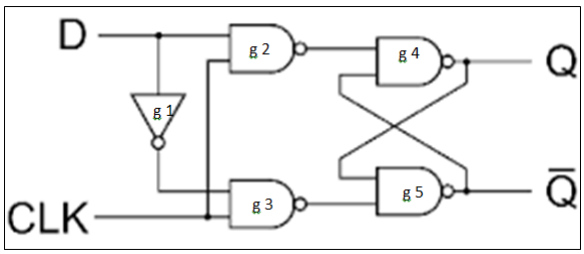
در Shift – register باید هر کدام از مالتی پلکسر ۱*۴ و D فلیپ – فلاپ را چندین بار فراخوانی کنیم . البته با توجه به این که خروجی مالتی پلکسر ، ورودی D فلیپ – فلاپ میباشد میتوان یک ماژول دیگر ساخت که در آن ، این دو را فراخوانی کنیم و در Shift – register از آن ماژول استفاده کنیم .

## ❖ مالتی پلکسر ۱*۴



```
 1   module mux41(I0,I1,I2,I3,sel0,sel1,f);
 2     input I0,I1,I2,I3,sel0,sel1;
 3     output f;
 4     wire[1:6] w;
 5     and #(6) g1(w[1],I0,w5,w6);
 6     and #(6) g2(w[2],I1,sel0,w5);
 7     and #(6) g3(w[3],I2,sel1,w6);
 8     and #(6) g4(w[4],I3,sel1,sel0);
 9     or #(8)  g5(f,w[1],w[2],w[3],w[4]);
10     not #(4) g6(w[5],sel1);
11     not #(4) g7(w[6],sel0);
12   endmodule
```

❖ D – flip flop :




```
 1   module dflop(d,c,q);
 2     input d,c;
 3     output q;
 4     wire r,w1,w2,w3;
 5     not #(4)  g1(r,d);
 6     nand #(5) g2(w1,d,c);
 7     nand #(5) g3(w2,r,c);
 8     nand #(5) g4(q,w1,w3);
 9     nand #(5) g5(w3,w2,q);
10   endmodule
```

❖ Shift – register : ( شکل صورت سوال )

```
 3    module shiftreg(I,serialR,serialL,sel0,sel1,clk,A);
 4    input[0:7] I;
 5    input serialR,serialL,sel0,sel1,clk;
 6    output[0:7] A;
 7    reg[0:7] A;
 8    wire[0:7] w;
 9    mux41 m0(A[0],A[1],serialL,I[0],sel0,sel1,w[0]);
10    mux41 m1(A[1],A[2],A[0],I[1],sel0,sel1,w[1]);
11    mux41 m2(A[2],A[3],A[1],I[2],sel0,sel1,w[2]);
12    mux41 m3(A[3],A[4],A[2],I[3],sel0,sel1,w[3]);
13    mux41 m4(A[4],A[5],A[3],I[4],sel0,sel1,w[4]);
14    mux41 m5(A[5],A[6],A[4],I[5],sel0,sel1,w[5]);
15    mux41 m6(A[6],A[7],A[5],I[6],sel0,sel1,w[6]);
16    mux41 m7(A[7],serialL,A[6],I[7],sel0,sel1,w[7]);
17    dflop d0(w[0],clk,A[0]);
18    dflop d1(w[1],clk,A[1]);
19    dflop d2(w[2],clk,A[2]);
20    dflop d3(w[3],clk,A[3]);
21    dflop d4(w[4],clk,A[4]);
22    dflop d5(w[5],clk,A[5]);
23    dflop d6(w[6],clk,A[6]);
24    dflop d7(w[7],clk,A[7]);
25    endmodule
```

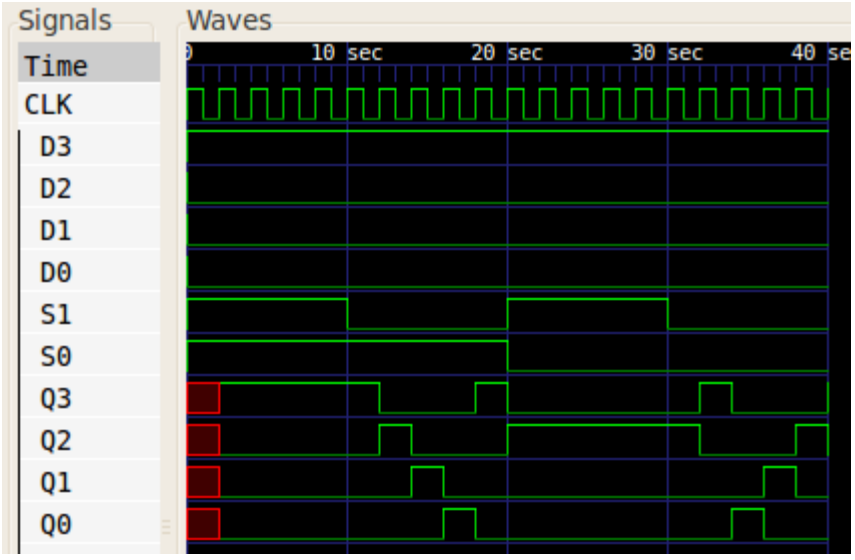



**Figure ۱این شکل برای یک شیفت رجیستر ۴ بیتی است**

تست برای شیفت رجیستر :

C:/modeltech64_10.1c/examples/MY_shift_register/ttt.v (/shifter) - Default

```
Ln#
1   module shifter(
2   input [31:0] shamt,
3   output reg [31:0] result );
4   reg[31:0] temp;
5   always@(*)begin
6      temp=32'hfff00fff;
7      result[31:0]=temp[31:0]>>>shamt[4:0];
8   end
9   endmodule
```

همان طور که قبلا گفتیم با توجه به این که خروجی مالتی پلکسر ، ورودی D فلیپ – فلاپ میباشد میتوان یک ماژول دیگر ساخت که در آن ، این دو را فراخوانی کنیم و در Shift – register از آن ماژول استفاده کنیم .

```
Ln#
1   module mux_flop(i0,i1,i2,i3,s0,s1,clk,q);
2    input i0,i1,i2,i3,s0,s1,clk;
3    output q;
4    wire d;
5    mux41 m(i0,i1,i2,i3,s0,s1,d);
6    dflop f(d,clk,q);
7
8   endmodule
```

که در این صورت کد شیفت رجیستر به صورت زیر خواهد بود :

```
1   //shiftregister for 8 bit
2   module shiftreg(I,serialR,serialL,sel0,sel1,clk,A);
3   input[0:7] I;
4   input serialR,serialL,sel0,sel1,clk;
5   output[0:7] A;
6   reg[0:7] A;
7   mux_flop m1(A[0],A[1],serialL,I[0],sel0,sel1,clk,A[0]);
8   mux_flop m2(A[1],A[2],A[0],I[1],sel0,sel1,clk,A[1]);
9   mux_flop m3(A[2],A[3],A[1],I[2],sel0,sel1,clk,A[2]);
10  mux_flop m4(A[3],A[4],A[2],I[3],sel0,sel1,clk,A[3]);
11  mux_flop m5(A[4],A[5],A[3],I[4],sel0,sel1,clk,A[4]);
12  mux_flop m6(A[5],A[6],A[4],I[5],sel0,sel1,clk,A[5]);
13  mux_flop m7(A[6],A[7],A[5],I[6],sel0,sel1,clk,A[6]);
14  mux_flop m8(A[7],serialR,A[6],I[7],sel0,sel1,clk,A[7]);
15  endmodule
```