جلسه ۱ ۹۵/۷/۴

* زبان وریلاگ : توصیف سخت افزار و کار با سخت افزار FPGA

* اولین ایراد FPGA ها نسبت به IC ها اینه که ابعاد خیلی بزرگ و سرعت کم است. سیم کشی عامل اصلی محدودیت سرعت در CPU های پیشرفته است.

PLD → Programmable Logic Device

FPGA → Field Programmable Gate Array

ASIC Design → Application Spec Integrated Circuit

* کوچک شدن ابعاد در الکترونیک به معنی افزایش سرعت است.

جلسه ۲ ۹۵/۷/۵

فشر : ساده سازی بستری برای تست ایده آل : Field

* کاربرد ها : ۱- کاربرد های نظامی (زیاسه، مکانیزم شنود تلفنی)

۲- پروژه های شامل چند نوع پردازش مختلف

* مثال) پردازش تصویر، ارتباط مخابرات و ماهواره‌ای، ارتباط سنسورها، تنظیم سوخت، سرعت، فشار مخزن و ... محاسبه‌هایی که نیاز به یک پردازنده مرکزی دارند.

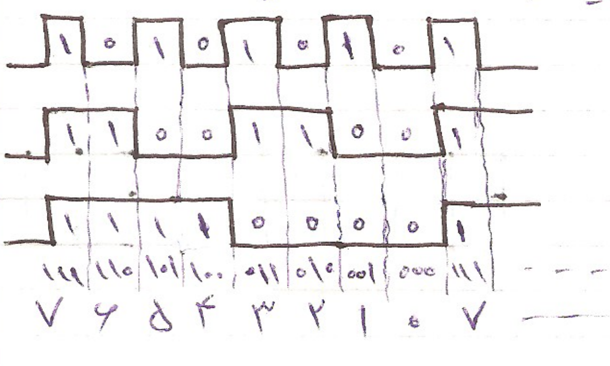
در مثال پرتاب موشک قراره مسیری رو طی کند و در جایی به هدف بخوره.

پردازش تصویر ← پردازنده پرقدرت

ارتباط مخابرات و ماهواره ← ماژول های GPS و ...

۳- شمارنده

تولید سیگنال های دیجیتال برای تست در آزمایشگاه



★ اصل ① : هر تابع منطقی ترکیبی را می‌توان بصورت مجموع حاصلضرب ها نوشت. (مجموع مین ترم ها) Sum of products (sop)

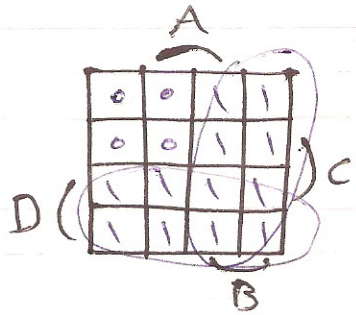
★ اصل ② : هر تابع منطقی را میتوان بصورت حاصلضرب مجموع ها نوشت.

products of Sum (pos)

★ مثال)

$f(A,B,C,D) = B + B'D$

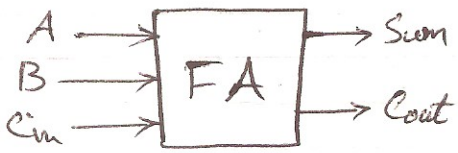
D	C	B	A	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



$f = B + D \Rightarrow$

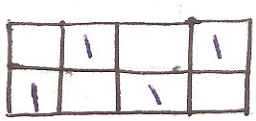
$x + x'y = x + y$

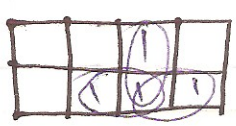
★ جمع کننده (FullAdder) :



Cin	B	A	Sum
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Cin	B	A	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1





$Sum = A \oplus B \oplus C_{in}$

$Cout = AB + BC_{in} + AC_{in}$

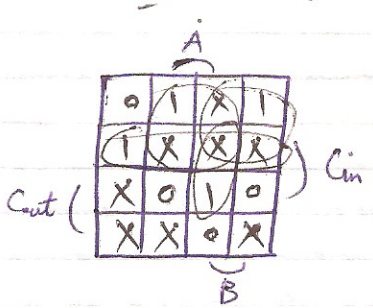
3

توابع Full Adder را به گونه‌ای تغییر دهید که :

۱) فرض کنید Cout در ورودی قرار دارد و Sum را برحسب چهار ورودی A , B , Cin , Cout بسازید.

Cout	Cin	B	A	Sum
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	X
0	1	0	0	1
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	X
1	0	0	1	X
1	0	1	0	X
1	0	1	1	0
1	1	0	0	X
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

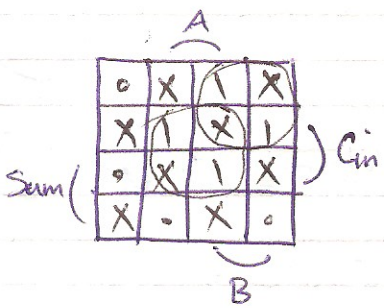
Cout, Cin تشکیل دهید.



$$Sum = C_{in}\,\acute{C}_{out} + B\,\acute{C}_{out} + A\,\acute{C}_{out} + AB\,C_{in}$$

۲) Cout را برحسب ۴ ورودی A , B , Cin , Sum بسازید.

Sum	Cin	B	A	Cout
0	0	0	0	0
0	0	0	1	X
0	0	1	0	X
0	0	1	1	1
0	1	0	0	X
0	1	0	1	1
0	1	1	0	1
0	1	1	1	X
1	0	0	0	X
1	0	0	1	0
1	0	1	0	0
1	0	1	1	X
1	1	0	0	0
1	1	0	1	X
1	1	1	0	X
1	1	1	1	1



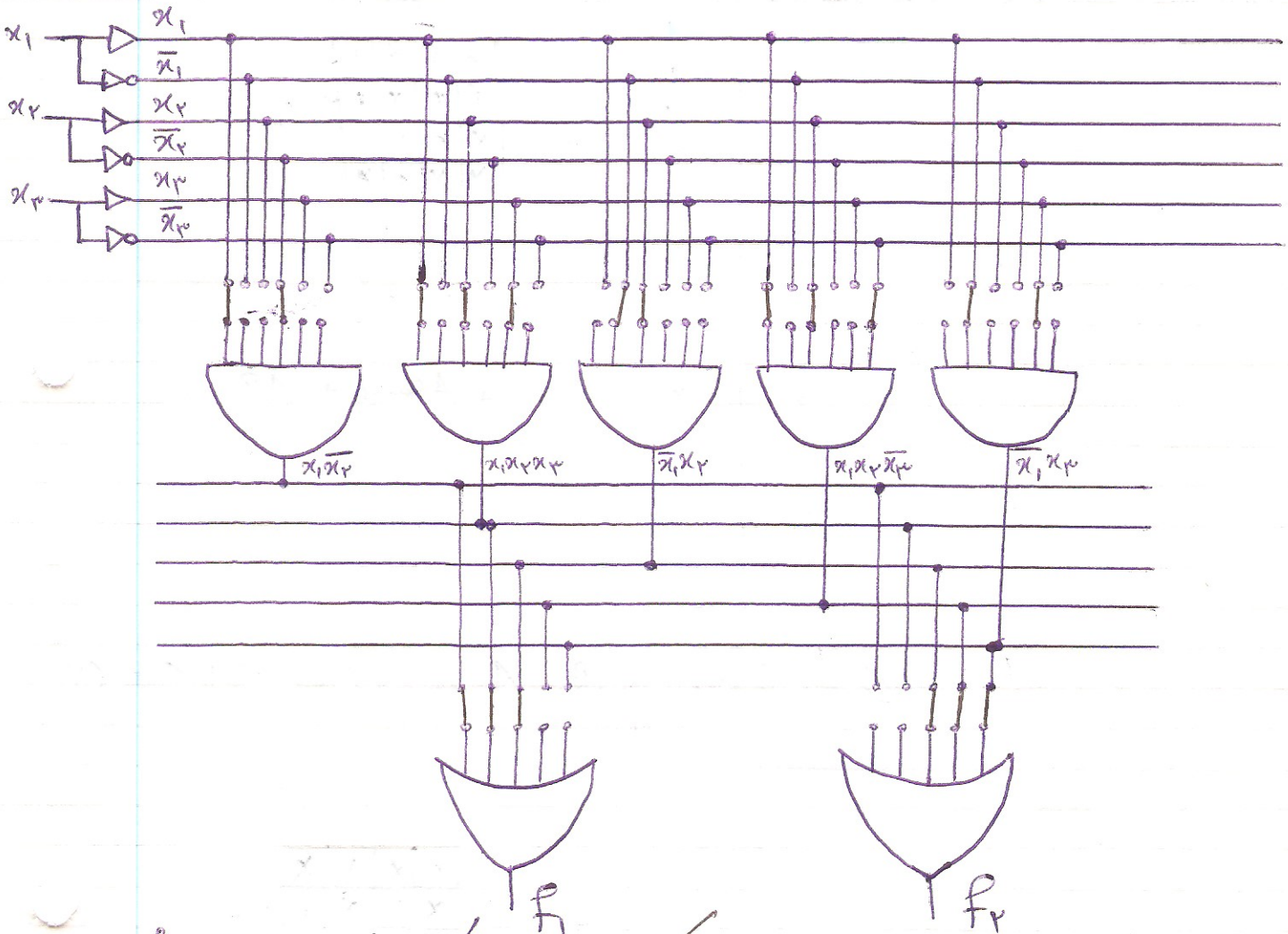
$$Cout = A\,C_{in} + B\,\acute{Sum}$$

جلسه ۳ — ۹۵،۷،۱۱

* مثال) دو تابع زیر را با PLA ساده سازی کنید.

$$f_1 = x_1\bar{x}_2 + x_1x_2x_3 + \bar{x}_1x_2$$

$$f_2 = x_1x_2\bar{x}_3 + \bar{x}_1x_3 + \bar{x}_1x_2$$



* مثال) تابع $A\oplus B\oplus C$ را با PLA ساده سازی کنید. $\Rightarrow$ $f = A\oplus B = A'B + AB'$

$$A\oplus B\oplus C = (A\oplus B)\oplus C = (A'B + AB')'C + (A'B+AB')C'$$

$$= \left[(A+B')(A'+B)\right]C + \left[A'BC' + AB'C'\right] =$$

$$\left[\underbrace{AA'}_{0} + A'B' + AB + \underbrace{BB'}_{0}\right]C + A'BC' + AB'C' =$$

$$A'B'C + ABC + A'BC' + AB'C'$$

شکل مدار در صفحه‌ی ۶ رسم شده است.

1. SSI Tech (Small Scale Integration Tech)

تعداد گیت ها < 100

2. MSI Tech (Medium Scale Int. Tech)

تعداد گیت ها $< 10^4$

3. LSI Tech (Large Scale Int. Tech)

تعداد گیت ها $> 10^4$ احساس نیاز به زبان توصیف سخت افزار

4. VLSI Tech (Very Large Scale Int. Tech)

HDL (Hardware Description Language)

5. PLD (Programmable Logic Device)

مدارهای منطقی قابل برنامه ریزی

نسل های مختلف:
- PLA
- PAL
- CPLD
- FPGA ← بعضی از منابع خود این نسل را جزء نسل های اصلی می دانند.

فصل اول (مدارهای منطقی قابل برنامه ریزی از نوع PLA)

PLA: Programmable Logic Array

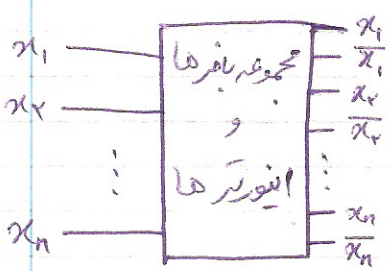
★ براساس اصل SOP:

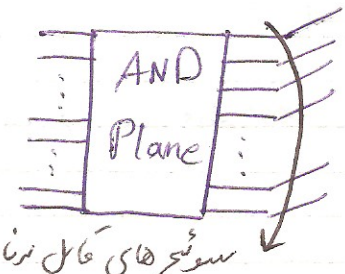
۱- ساخت ورودی ها، invert شده آنها

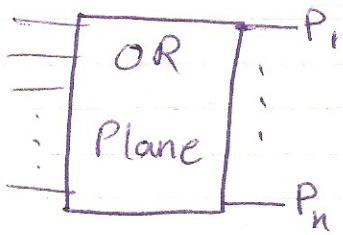
۲- ساخت product ها با آرایه AND

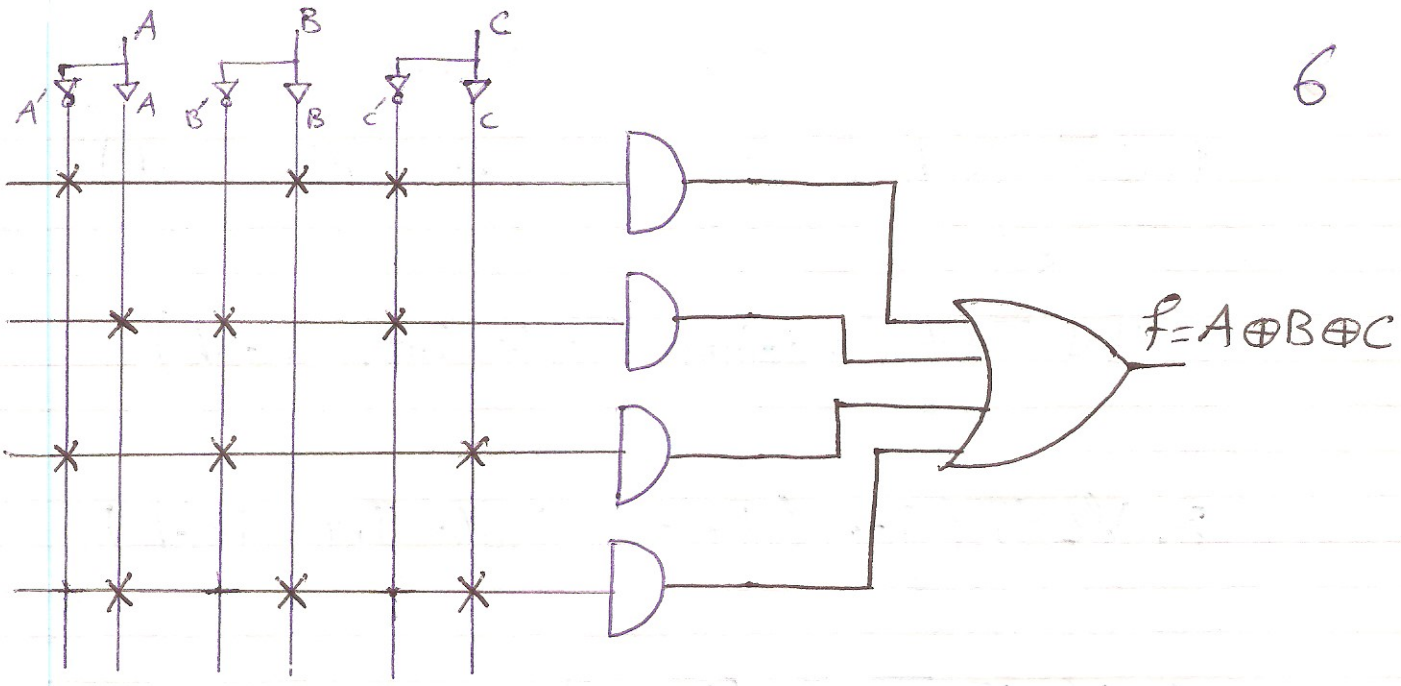
۳- ساخت sum ها با آرایه OR

۴- سرانجام های قابل برنامه ریزی









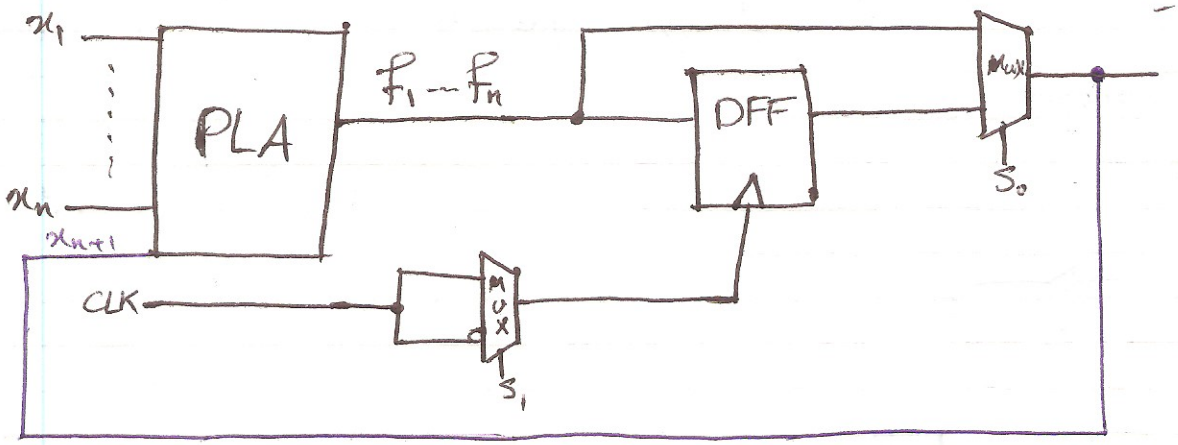
★ایراد های عمده PLA : ۱- دو طبقه سوئیچ قابل برنامه ریزی (توان مصرفی بالا)

۲- تنها مدارهای منطقی ترکیبی را می‌توان پیاده سازی کرد.

۳- سوئیچ ها از جنس فیوز بودند و تنها یکبار program می‌شوند.

٤- خروجی PLA ها قابل استفاده مجدد نبود.

★ رفع ایراد ها :



شماتیک یک "ماکروسل پایه" (PLA های اصلاح شده)

★ در PAL ها و PLA ها پایه های بدون استفاده AND ، باید 1 باشند و پایه های OR ، صفر باشند.

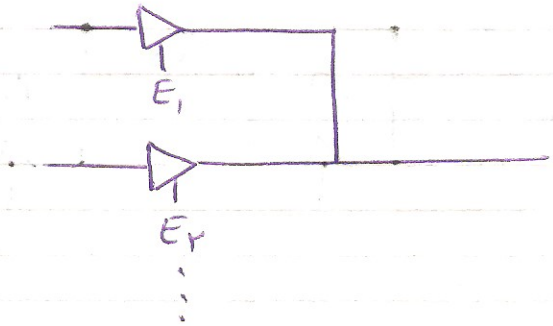
★ PLA ها دارای نقطه ضعف سوئیچ دوگانه است که توان مصرفی را زیاد می کند.

۹۵/۷/۱۲ جلسه ۴

★ PAL (Programmable And Logic):

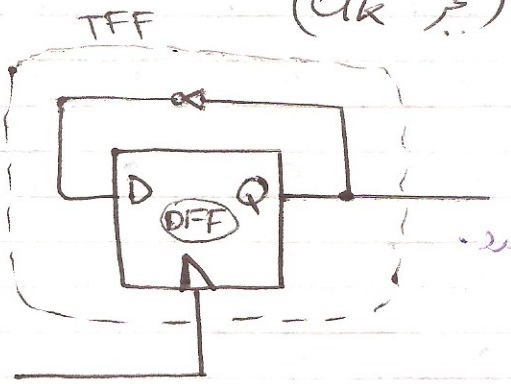
یک طبقه سوئیچ وجود دارد که در طبقه AND است.

(تنها طبقه AND قابل برنامه ریزی است.)

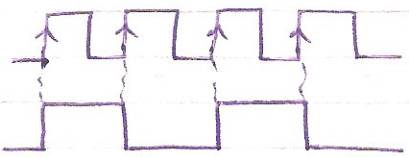


★ نکته) TFF در این درس ورودی ندارد. (بجز Clk)

★ خصوصیات TFF

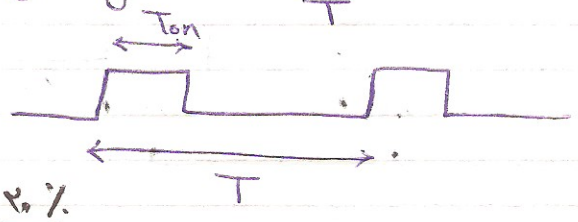


① خاصیت نصف کنندگی فرکانس دارد.

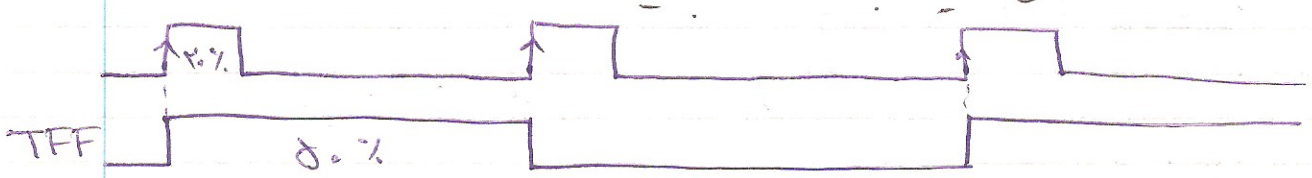


② خاصیت اینورتر دارد. یعنی ورودی ندارد و فقط خروجی را با Duty Cycle نات می‌کند.

$$\text{Duty Cycle} = \frac{T_{on}}{T}$$



★ مثال) فرکانس زیر را به TFF بدهید.

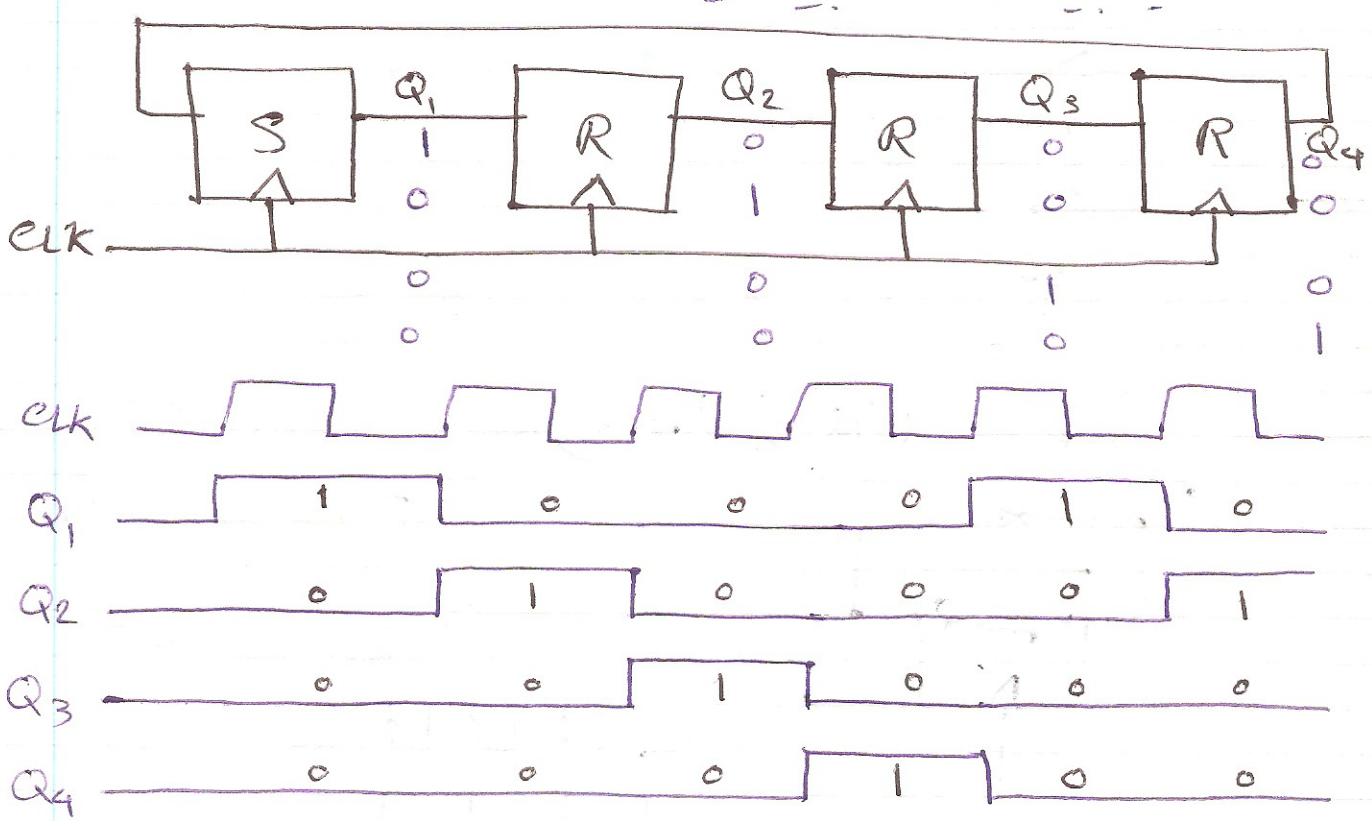


③ از این مثال متوجه می‌شویم هر فرکانسی به TFF داده شود فرکانس خروجی

Duty Cycle = 50 % خواهد داشت.

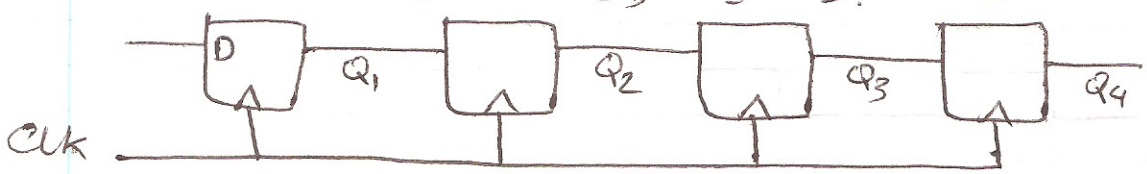
★ شمارنده حلقوی (Ring Counter) :

یک بیت 1 و سه بیت 0



★ تمرین) ① D یا $\bar{D}$ را به ورودی I_1 دیکدر طوری که D در خروجی Q_V ظاهر شود.

② شیفت رجیستر زیر را پیاده سازی کنید.



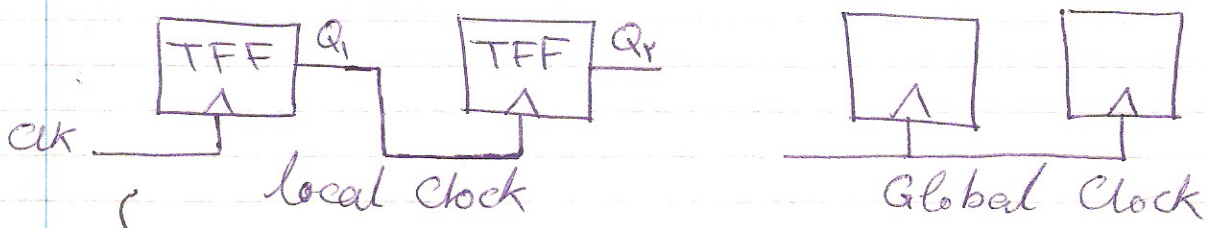
③ تابع Cout را پیاده سازی کنید. (با سل ۱ و ۸)

$$C_{out} = AB + B C_{in} + A C_{in}$$

④ تابع Sum را پیاده سازی کنید.

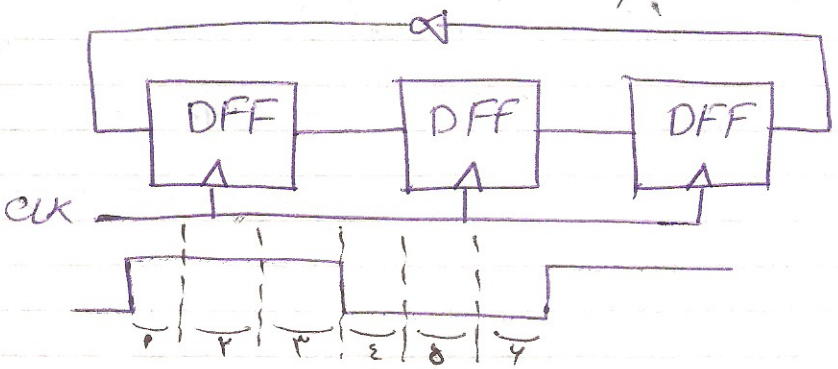
$$Sum = A \oplus B \oplus C_{in}$$

۹۵/۷/۱۸ جلسه ۵



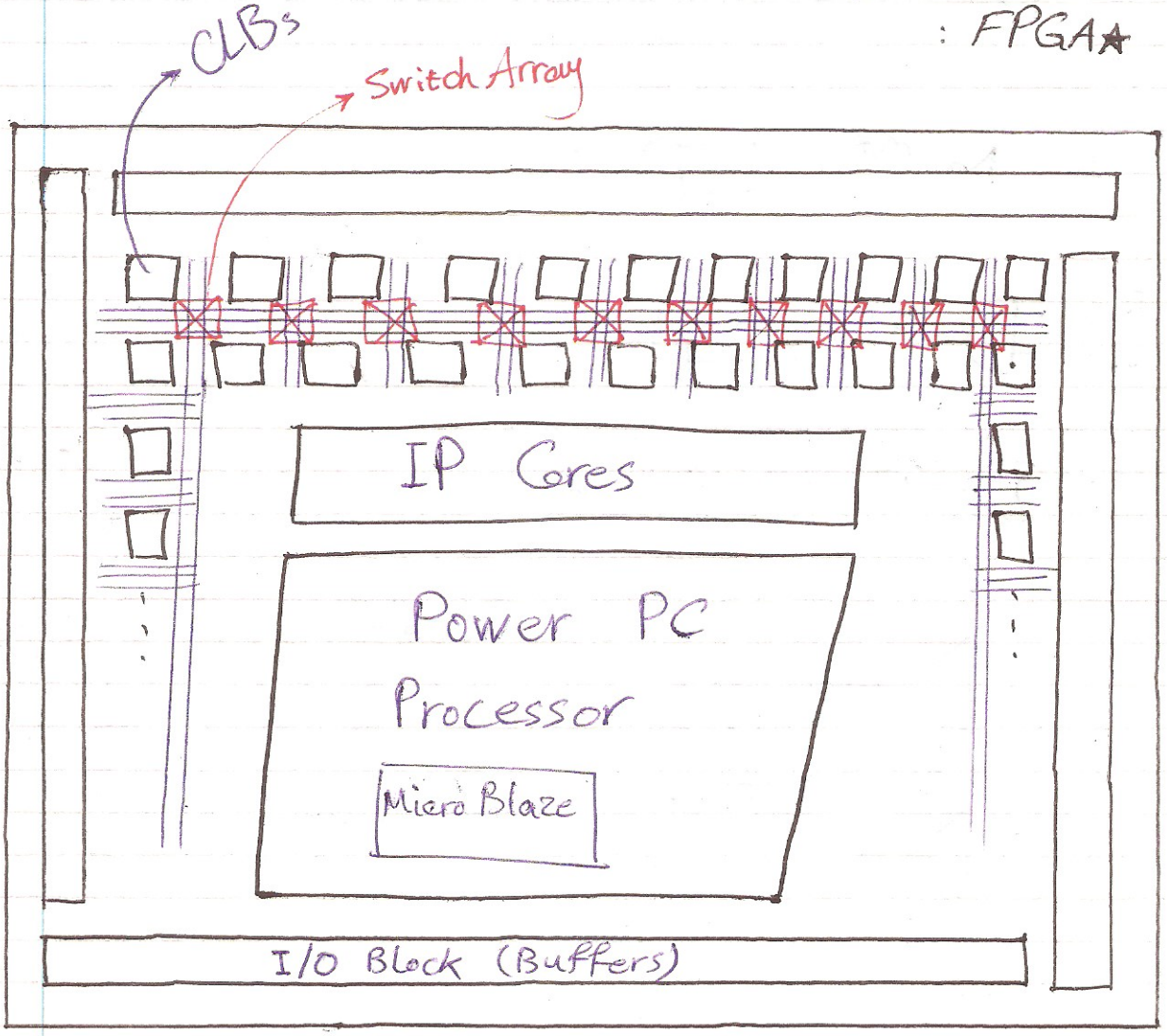
Counter ($\frac{1}{2^n}$ بافرکانس)

★ شمارنده‌ای که فرکانس تقسیم بر ۶ شود نه تقسیم بر توانی از ۲ :



★ CPLD : تشکیل شده از چندین هزار PAL

★ FPGA :



۱- سلول های منطقی قابل برنامه ریزی (Configurable Logic Block)

۲- اتصالات میانی (Interconnection)

۳- بافرهای ورودی / خروجی (I/O Buffers)

۴- آرایه سوئیچ های قابل برنامه ریزی (Switch Array)

۵- هسته های شناخته شده (IP Cores)

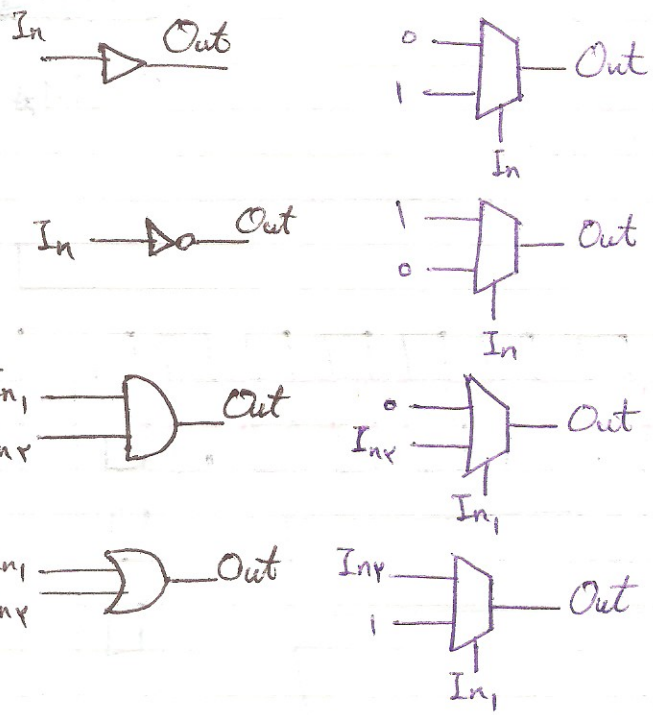
۶- پردازنده پرقدرت مرکزی (Power PC Processor)

★ ساختار داخلی CLB ها در FPGA :

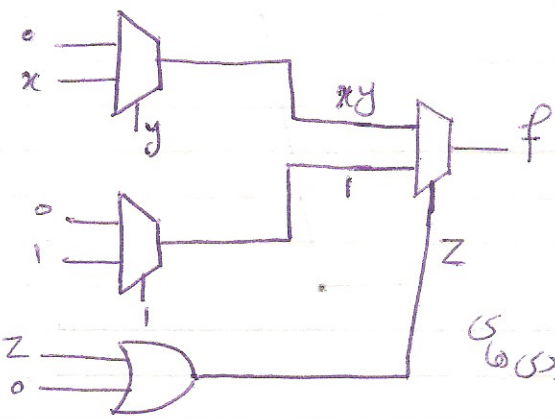
۱- FPGA های مبتنی بر Mux (شرکت Actel)

۲- FPGA های مبتنی بر LUT (شرکت های Altera , Xilinx)

★ پیاده سازی گیت های منطقی با Mux :



★ پیاده سازی تابع $f=z+xy$:

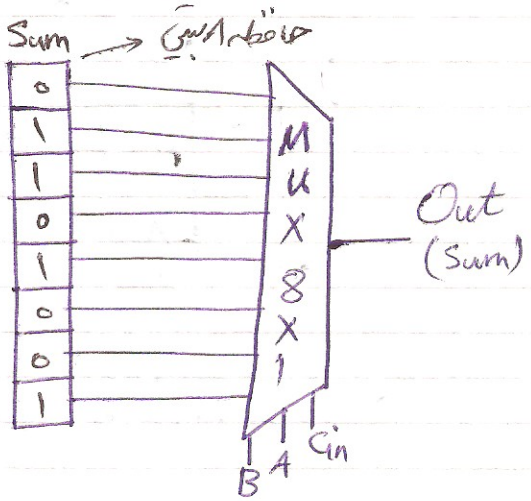


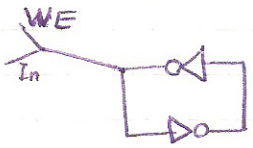
این مدار باید طراحی شده و بهینه برای حالت کلی توسط شرکت Actel می باشد.

★ ایراد کمدر : عدم تقارن خروجی نسبت به ورودی های متفاوت

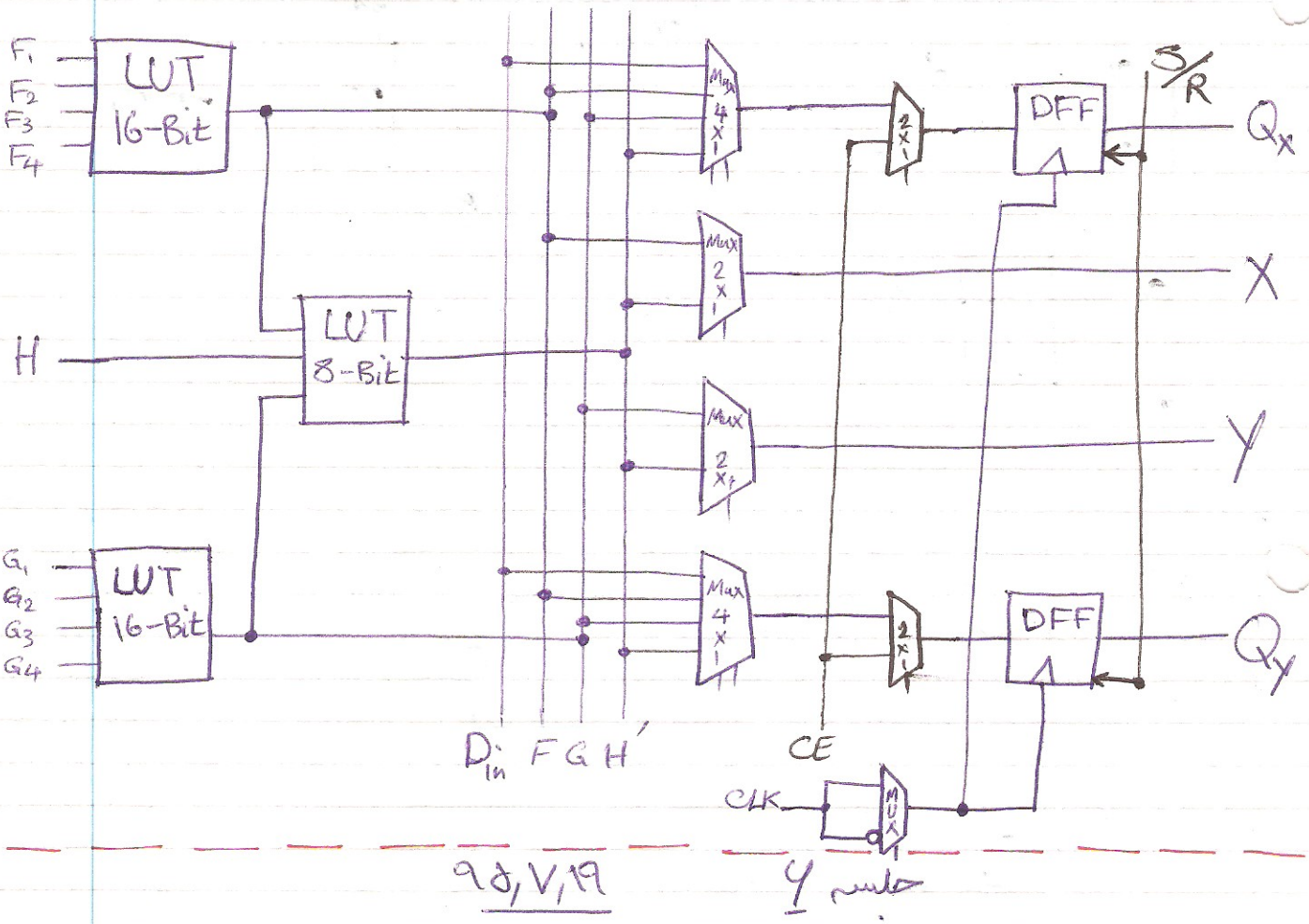
ساخت FPGA، مبتنی بر LUT (Look Up Table):

B	A	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



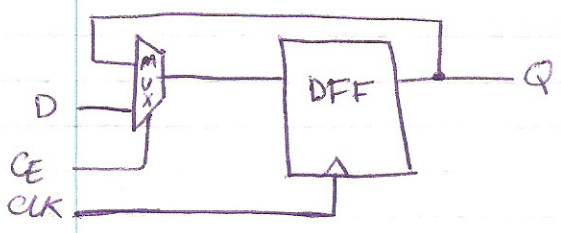


یک سلول حافظه ⇐

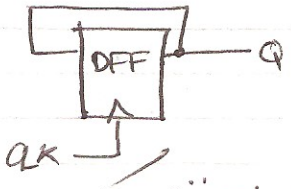


۹۵/۷/۱۹ — جلسه ۷

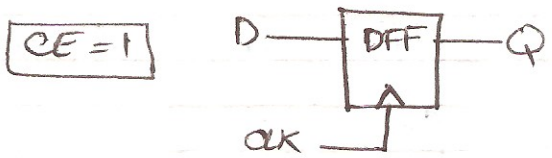
Xilinx Panent ★



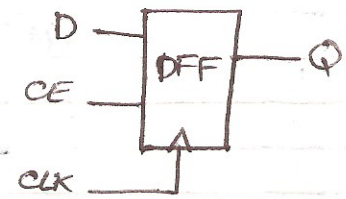
CE = 0



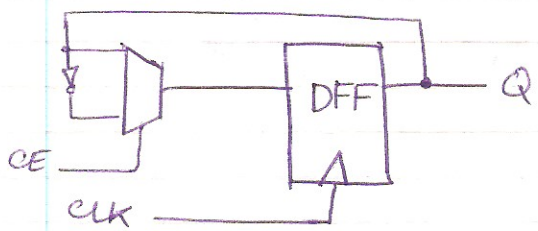
داده‌ی قبلی را حفظ می‌کند.



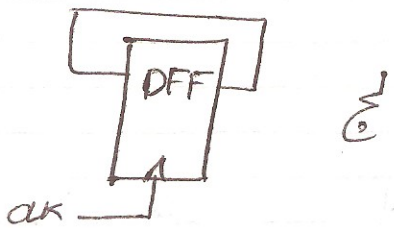
$\begin{cases} CE = 0 & \text{تبدیل به } \not\subset \\ CE = 1 & \text{تبدیل به DFF} \end{cases}$



$\Rightarrow$ CE دار DFF



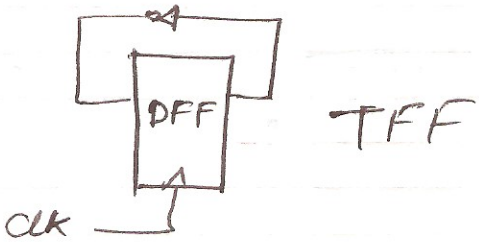
CE = 0



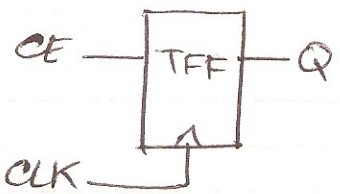
$\not\subset$

$\begin{cases} CE = 0 & \not\subset \\ CE = 1 & \text{عمل‌کرد TFF} \end{cases}$

CE = 1

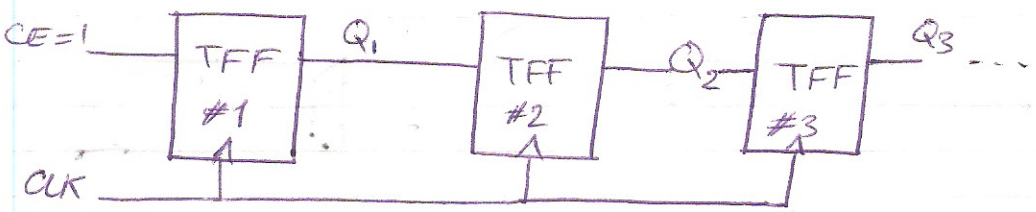


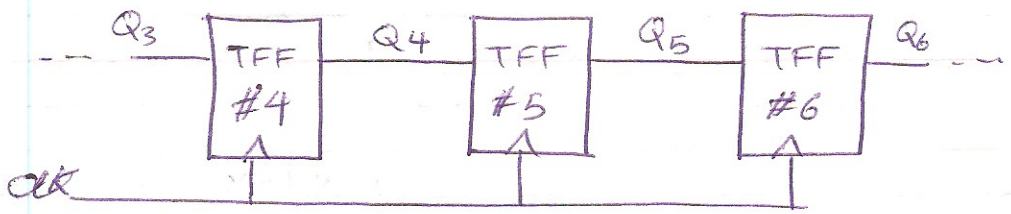
TFF

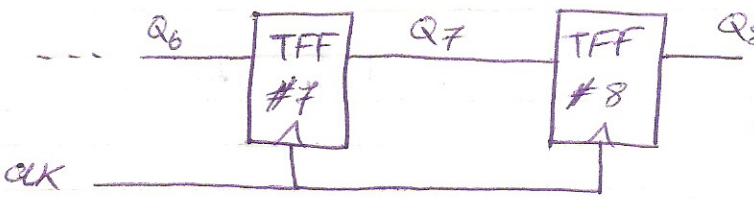


$\Rightarrow$ CE دار TFF

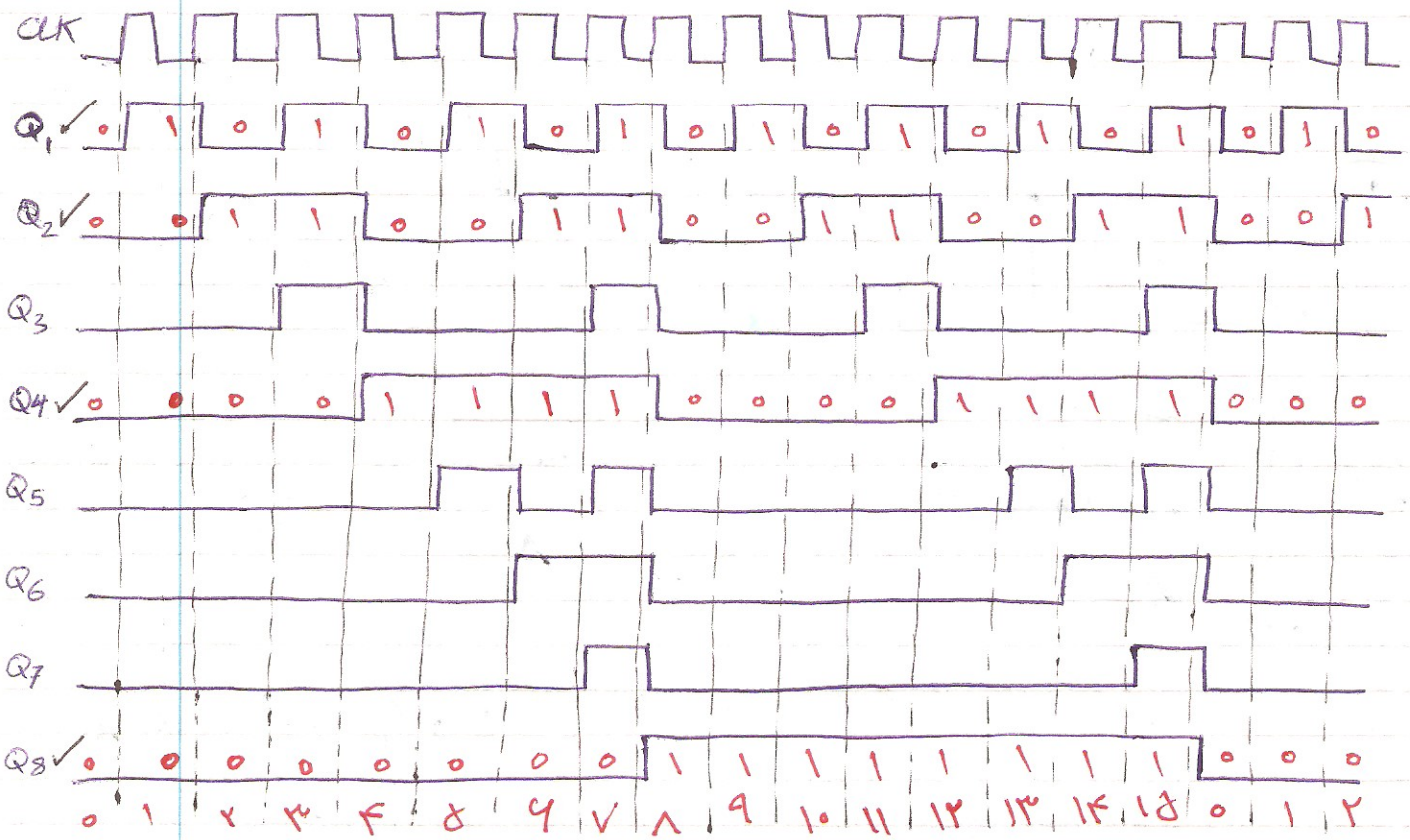
☆ مثال)

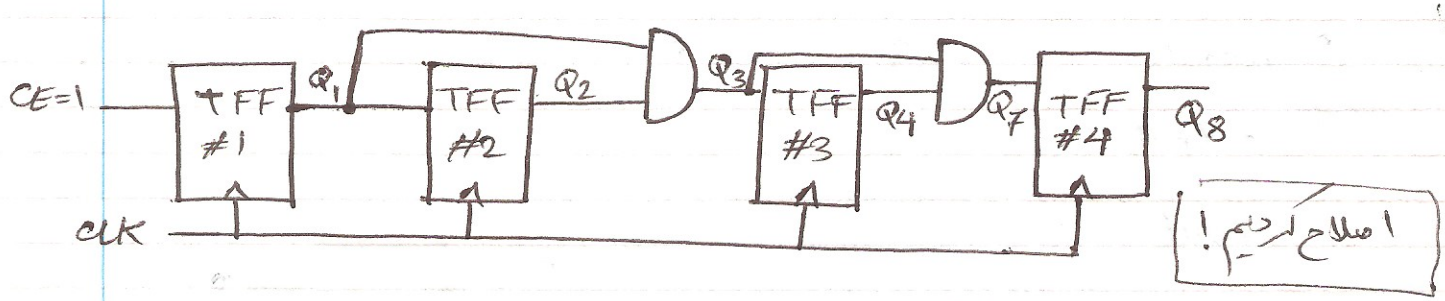






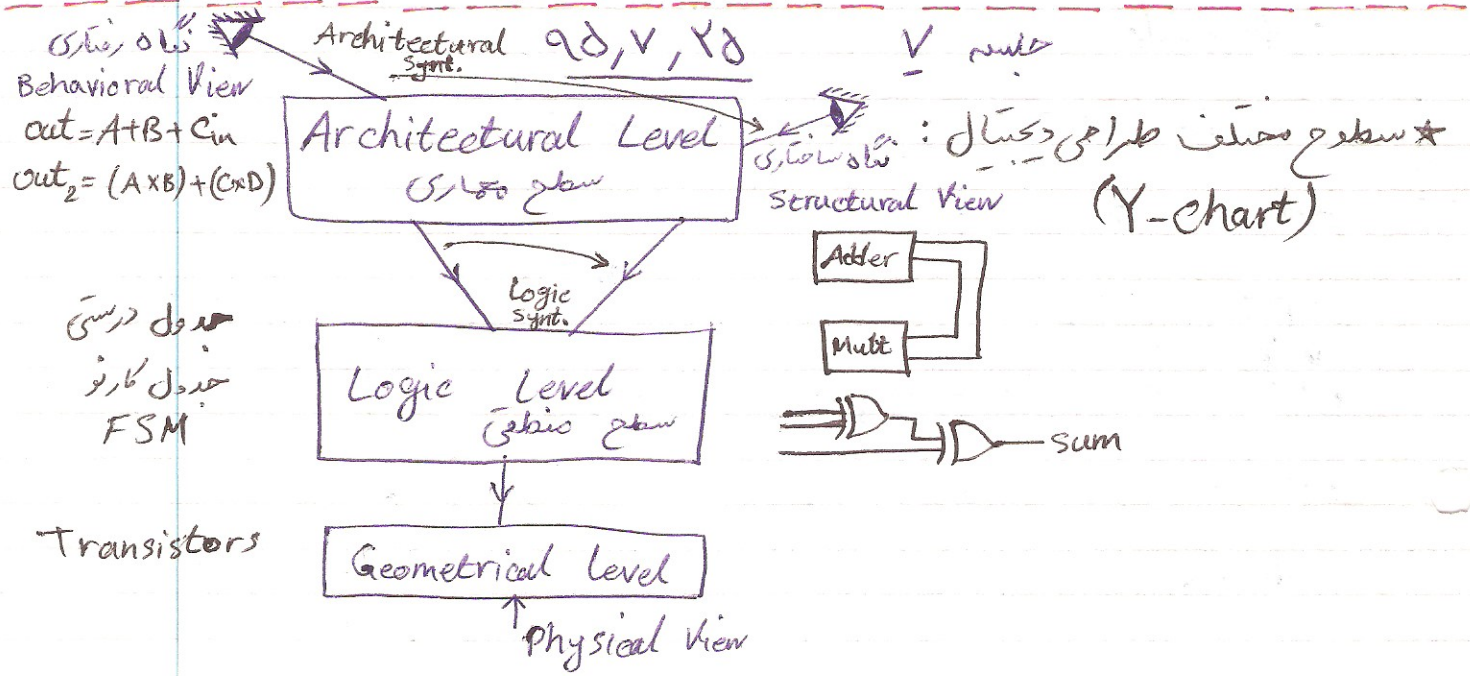
ایراد: برای ساده سازی یک شمارنده ۴ بیتی به ۸ تا TFF نیاز داریم!





★ زبان برنامه نویسی وریلاگ :

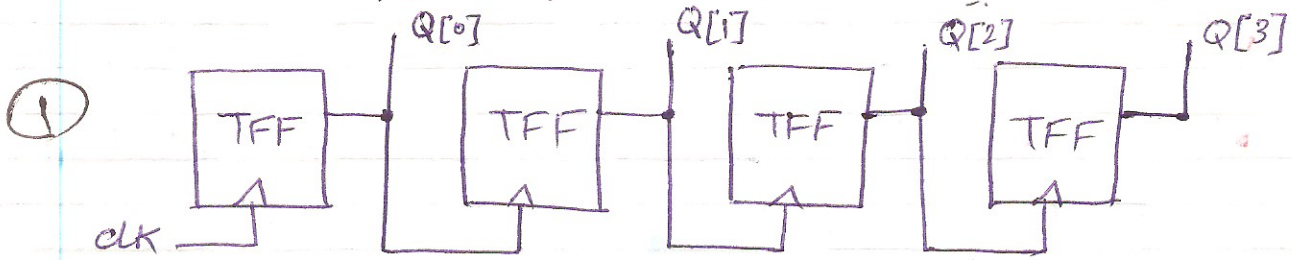
HDL → VHDL
HDL → Verilog

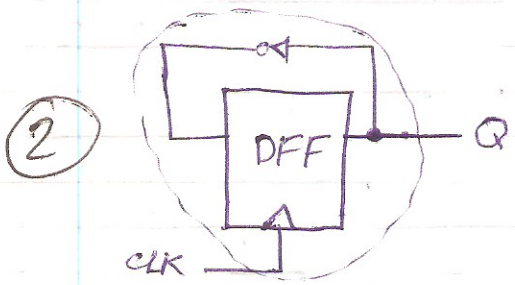


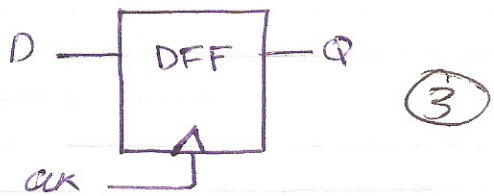
جلسه ۷ ۹۵,۷,۲۸

★ سطوح مختلف طراحی دیجیتال : (Y-chart)

★ مثال) کد شمارنده ۴ بیتی را با استفاده از DFF ها و گیت های منطقی بسازید.



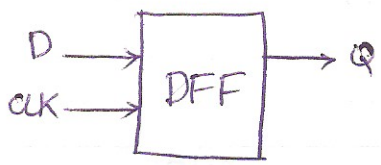




★ module (ماژول) : واحد انجام پروژه در زبان وریلاگ که با استفاده از پورت های ورودی و خروجی با دنیای بیرون در ارتباط است.

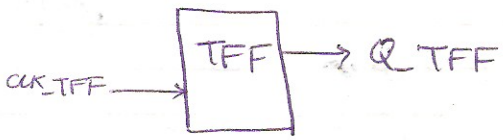
ابتدا پورت های خروجی

```
③ module DFF (Q , clk, D);
DFF output Q;
    input clk, D;
    reg Q;
    always @ (posedge clk)
        Q<=D;
    endmodule
```



سطح معماری - نگاه رفتاری

```
② module TFF (Q_TFF , clk_TFF);
TFF output Q_TFF;
    input clk_TFF;
    wire N1;
    DFF DFF1 (Q_TFF , clk_TFF, N1);
    not NOT1 (N1, Q_TFF);
    endmodule
```



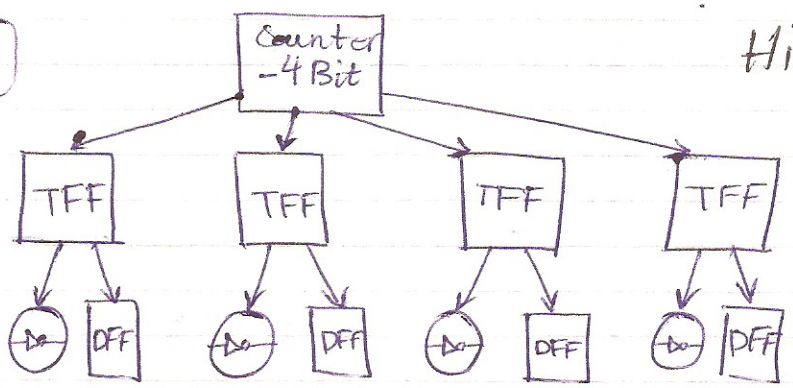
سطح معماری - نگاه ساختاری
سطح Logic - نگاه ساختاری

① 4Bit Counter

```
module Counter_4Bit (Q , CLK);
output [3:0] Q;
input CLK;
TFF TFF1 (Q[0] , CLK);
TFF TFF2 ( Q[1] , Q[0]);
TFF TFF3 (Q[2] , Q[1]);
TFF TFF4 ( Q[3] , Q[2]);
endmodule
```

★ ساختار سلسله مراتبی : Top_module & Sub_module

www.opencores.com
پروژه های وریلاگ



Hierarchy

★ فراخوانی (Instantiation) : ← روش اتصال پورت ها با ترتیب

← روش اتصال پورت ها با نام

```
TFF TFF1 (.CLK_TFF(CLK) , .Q_TFF(Q[0]));
```

جلسه 1 ۹۵/۷/۲۶

★ مفاهیم اولیه در وریلاگ :

۱- حروف کوچک و بزرگ دارای معانی متفاوت در وریلاگ هستند.

۲- اسم گذاری باید کامل باشد و در تعداد حروف محدودیتی ندارد.

۳- فاصله ی خالی در زبان وریلاگ بین In ، It ، Ib تفاوتی وجود ندارد.

۴- کامنت های تک خطی (//) و چند خطی (/* -- */) برای توضیح دادن کد داریم.

۵- الگوی عدد نویسی در وریلاگ

<عدد> <مبنا> <تعداد بیت ها>

- با تعداد بیت مشخص
- بدون تعداد بیت مشخص (32 بیت فرض می‌کند)

مبنا:
- b → 2
- h → 16
- o → 8
- d → 10

4'b 1011 — یک عدد ۴ بیتی در مبنای ۲

4'd 10 — یک عدد ۴ بیتی در مبنای ۱۰

-4'b 10 → معادل → 4'b -10

26'b 1 — اگر بیت MSB صفر یا تک باشد. → بقیه بیت ها با صفر پر می‌شوند.

همه بیت ها با X پر می‌شوند ← اگر X باشد.

همه بیت ها با Z پر می‌شوند ← اگر Z باشد.

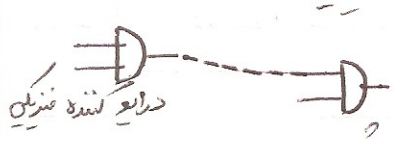
۶- سطوح سیگنال در وریلاگ

- 0 ← سطح Low در منطق دیجیتال / تعبیر غلط بودن یک عبارت منطقی
- 1 ← سطح High در منطق دیجیتال / تعبیر درست بودن یک عبارت منطقی
- X ← متغیر دارای مقدار است اما مقدار آن برای ما مبهم است. / به معنای don't care هم هست.
- Z ← به معنای high impedence است. / اگر به متغیرهای خانواده wire net مقدار داده نشود Z فرض می‌شود.

★ اگر به متغیرهای خانواده‌ی reg مقدار داده نشود، X فرض می‌شود.

۷- انواع متغیرها در وریلاگ

★ net : فقط برای اتصالات فیزیکی بین المان های فیزیکی استفاده می‌شود.



باید دارای درایو کننده فیزیکی باشد.

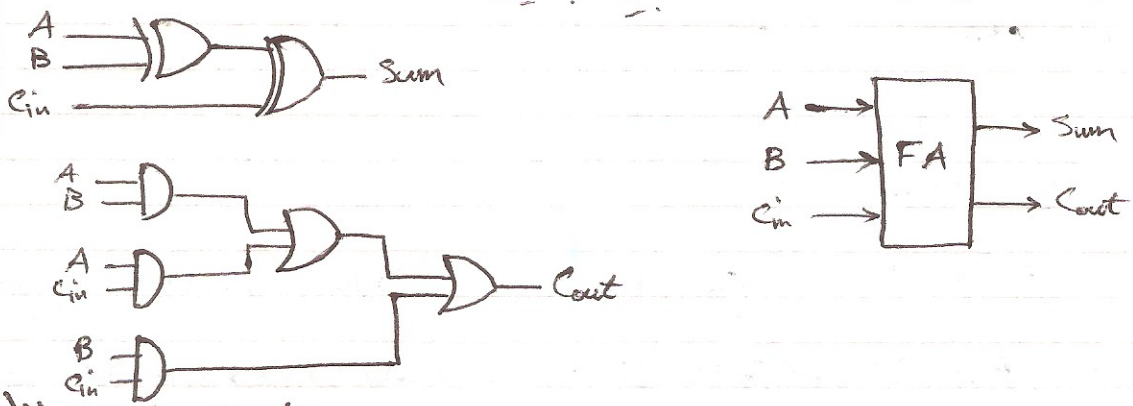
خروجی گیت منطقی باید جنس wire تعریف شود.

wire - wand - wor - tri - triand - trior

trireg - supply0 - supply1 - tri0 - tri1

قابل سنتز نیستند.

★ مثال) مدارول یک FA تک بیتی را بنویسید.

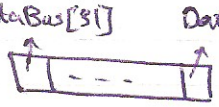



```
`timescale 1ns/1ps
module FA (Cout, Sum, A, B, Cin);
output Cout, Sum;
input A, B, Cin;
wire w1, w2, w3, w4, w5;
xor x1 (w1, A, B);
xor #10 x2 (Sum, w1, Cin);
and A1 (w2, A, B);
and A2 (w3, A, Cin);
and A3 (w4, B, Cin);
or o1 (w5, w2, w3);
or o2 (Cout, w4, w5);
endmodule
```

جلسه ۹ — ۹۵/۸/۲

★ خواص متغیرهای net :

۱- تعریف بصورت برداری

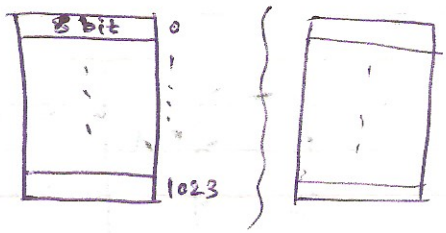



```
wire [31:0] DataBus;
```

۲- قابلیت حفظ اطلاعات را ندارد. در هر لحظه مقدار خود را از یک درایو کننده فیزیکی مثل از خود دریافت می‌کند.

۳- تعریف بصورت آرایه

```
wire [7:0] Bus [0:1023];
```

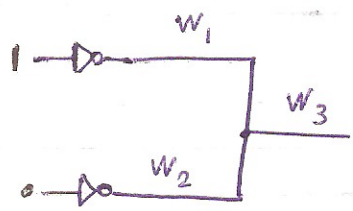


آرایه سه بعدی: `wire [7:0] Bus_3D1 [0:1] [0:1023];` دو باس جداگانه با 1024 سطر که هر سطر ۸ بیت است.

`wire Bus_3D2 [0:1][0:1023][7:0];` دو باس جداگانه با 1024 سطر که در هر سطر ۸ تا جدا هستند.

```
... = Bus_3D1 [1] [510]
```

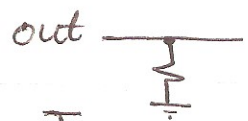
Wor	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z



wor

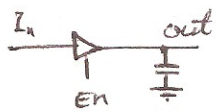
① `wire out;`

② `tri0 out;` (pull down)



③ `tri1 out;` (pull up)

④ `trireg out;`



tri0 { if En=1 → out=In ; else if En=0 → out=1'b0

tri1 { if En=1 → out=In ; if En=0 → out=1'b1

tri reg { if En=1 → out=In ; if En=0 → out=مقدار قبلی

★ اگر به متغیری از خانواده net مقداری داده نشود، مقدارش Z در نظر گرفته می‌شود.

★ اگر به متغیری از جنس tri0 مقداری داده نشود، بطور پیش‌فرض مقدار 0 در نظر می‌گیرد.

★ " " " " " tri1 " " " " " 1 " " " " "

★ " " " " " trireg " " " " " X " " " " "

★ reg : integer - real - time

time و real قابل پیاده سازی روی FPGA (سنتز) نیستند.

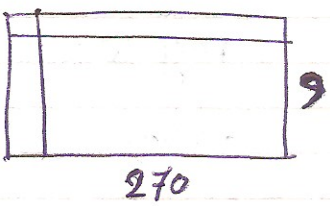
★ خواص reg :

۱- دارای حافظه است و آخرین مقدار را تا تغییرات بعدی در خود نگه می دارد.

۲- به صورت بردار و یا آرایه ای می توان تعریف کرد.

۳- مقدار دهی حتماً به صورت مستقیم (با علامت مساوی) باید انجام شود.

```
reg [7:0]  Memory_3D [0:8][0:269];
```




```
A = Memory_3D [3][4];
```
استاندارد 95

```
reg [7:0] A;
reg B;
B = A[0];
```

```
B = Memory_3D [3] [4][0];
```
استاندارد 2000

★ نوع reg : ابعاد اختیاری - می تواند بردار یا آرایه باشد - بیت علامت ندارد. (مقدار دهی تعریف کرد)

★ نوع integer : طول 32 بیت - نگاه علامت دار (بیت آخر علامت است.) -

به صورت بردار قابل تعریف نیست، تنها به صورت آرایه تعریف می شود.

★ کاربرد های integer :

۱- برای شمارش در حلقه های for

۲- کار با نابل ها

۳- برای ذخیره اعداد علامت دار

★ نوع real : طول 64 بیت (32 بیت برای صحیح و 32 بیت برای اعشار)

جلسه ۱۰ ۹۵,۸,۳

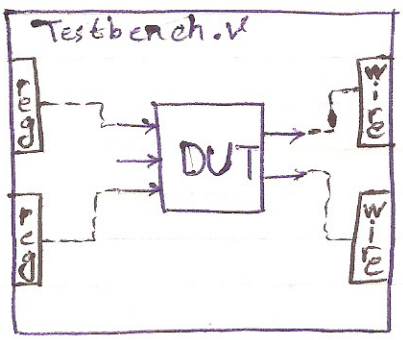
reg	net
مقدار دهی عددی به هر متغیر دلخواه می تواند انجام شود.	حتماً نیاز به درایو فیزیکی دارند. (مثل گیت منطقی)
مقدار اولیه داده نشود، X فرض می شود.	مقدار اولیه داده شود، حتی فرض می شود.
می تواند بردار و آرایه تعریف شود.	می تواند بردار و آرایه تعریف شود.

reg	net
در فصل سطح انتقال داده و سطح گیت های منطقی کاربرد دارد.	در فصل توصیف رفتاری بسیار پرکاربرد هستند.
انواع time , real غیر سنتزشدنی هستند.	انواع trireg , tri1 , tri0 غیر سنتزشدنی هستند.

★ پورت ها در وریلاگ : [inout – output – input]

قانون : پورت ورودی باید از جنس wire تعریف شود.

```
reg Q;
always @ (posedge CLK)
  Q = D;
```



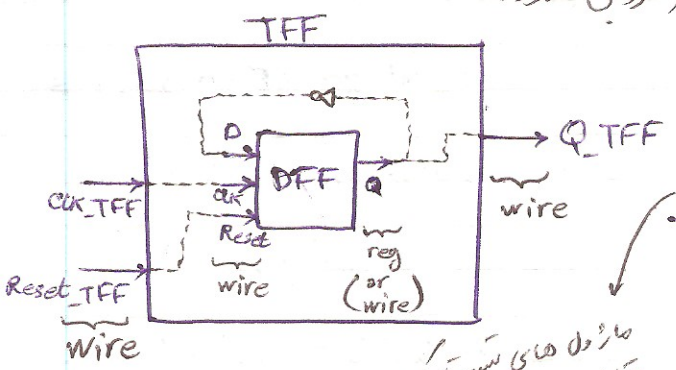
★ در کلیه ماژول های تست بجای پورت های ورودی سیگنال های تست را در متغیرهایی از جنس reg می سازیم. برای مشاهده‌ی سیگنال های خروجی در ماژول تست، متغیرهایی از جنس wire ساخته و به خروجی ماژول اصلی وصل می کنیم.

```
#10 CLK=1'b0
#10 CLK=1'b1
⋮
```

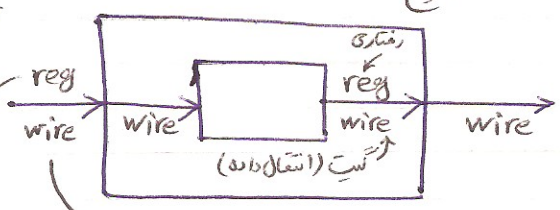
DUT: Device Under Test

★ خود ماژول تست نیاز به پورت ورودی / خروجی ندارد.

★ تست بنچ نیاز به سنتز ندارد.



ماژول های تست که پورت ندارد، مرا است سیگنال تولید شود.



در ماژول های دارای پورت ورودی

★ پارامترها :

```
parameter Data_Width = 8;
reg [Data_Width-1 : 0] Mem_3D [0:1023];
```

defparam
`define
→ قابل سنتز نیستند.

```
module DFF;
parameter Delay=1, Delay2=4;
always @(posedge CLK)
   #Delay  Q=D;
endmodule

module ---;
DFF #(2) DFF1(---);        #(2,3) ⟵ بترتیب
DFF #(4) DFF2(---);        #(.Delay(3), .Delay2(2)) ⟵ با نام
```

★ بلوک های سازنده ماژول :

۱- نام ماژول + کلمه کلیدی ماژول

۲- پورت های ورودی و خروجی

۳- انواع متغیرها و پارامترها

۴- توصیف بدنه اصلی ماژول

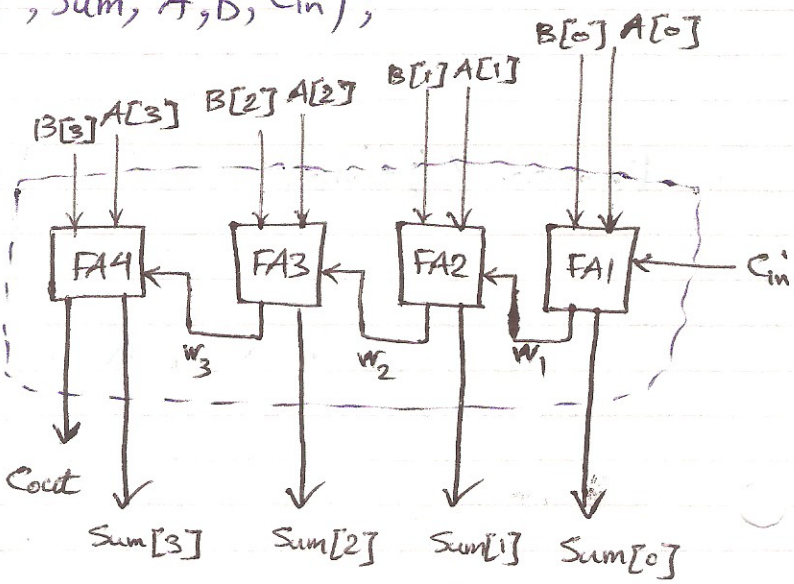
فراخوانی ماژول های دیگر

۵- توابع

۶- کلمه کلیدی endmodule

★ ماژول ای که دو عدد ۴ بیتی را با هم جمع کند. (از FA تک بیتی استفاده کند.)

```
module FA_4Bit (Cout, Sum, A, B, Cin);
input [3:0] A;
input [3:0] B;
input Cin;
output [3:0] Sum;
output Cout;
wire w1, w2, w3;
FA FA1 (W1, S[0], A[0], B[0], Cin);
```



```
FA FA2(w2, S[1], A[1], B[1], w1);
FA FA3(w3, S[2], A[2], B[2], w2);
FA FA4(Cout, S[3], A[3], B[3], w3);
endmodule
```

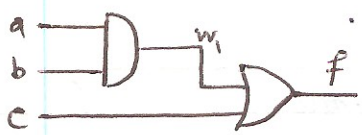
جلسه ۱۱ — ۹۵/۸/۹

★ دستورات سیستمی:

$display	$finish	شروع دستورات سیستمی با علامت $
$monitor	$time	★ دستور display تنها یکبار در زمانی که مشخص کرده‌ایم
$fdisplay	$strobe	اجرا می‌شود و اجرای آن پایان می‌پذیرد. اما دستور
$fmonitor	$fopen	monitor یکبار فعال‌سازی شده و تا پایان اجرای کد،
$stop	$fclose	مرتباً لیست حساسیت خود را بررسی می‌کند. در

صورت هرتغییری در لیست حساسیت، چاپ در خروجی انجام می‌شود.

- زمان انجام و اجرای تمام دستورات سیستمی به عهده‌ی کاربر است.




```
$monitor("a = %b ,b = %b \t, c=%b ,\t w1=%b ,f=%b", a,b,c,w1,f);
```

معمولاً داخل ماژول تست استفاده می‌شود.

- هیچ‌یک از دستورات سیستمی قابل سنتز نیستند.

```
$monitor($time, "---", ---);
```

- نمایش رشته‌ها و کاراکترهای خاص (% . \ . ")

کاراکترهای خاص را بصورت %% یا \\ یا \" نمایش می‌دهیم.

%b	باینری	\b	فاصله
%o	اکتال	\t	تب
%d	دسیمال	\n	خط جدید
%h	هگزادسیمال		
%m	چاپ سلسله مراتبی		

- دستورات fdisplay , fmonitor برای نمایش در فایل هستند.
- دستور stop موقتاً از مد اجرا خارج شده و وارد مد محاوره می‌شود تا با کاربر مبادله دستور نماید.
- دستور finish به طور کلی اجرا را متوقف کرده و کاملاً از simulation خارج می‌شود.

★ دستور stop در بیانهای اجرا هست ولی در finish امکان ندارد.

- دستورات fopen , fclose برای باز و بسته کردن فایل به کار می‌روند.

★ شناسه‌ها:

- ساده: حروف + اعداد + _ + $ شروع با عدد و $ غیرممکن
- مرکب: ***Hello*** شناسه‌ای که ساده نباشد، مرکب است.

فصل دوم (سطح انتقال داده)

به علت معادل بودن با سطح گیت‌های منطقی، از نظر فیزیکی درایورهایی برای تغذیه خروجی‌ها وجود دارند. بنابراین خروجی‌ها در این سطح همواره از جنس wire هستند.

★ کلمه کلیدی assign

```
wire out;
assign out = a & b;
```

حاوی لیست حساسیت

Continuous Assignment (انتساب پیوسته)

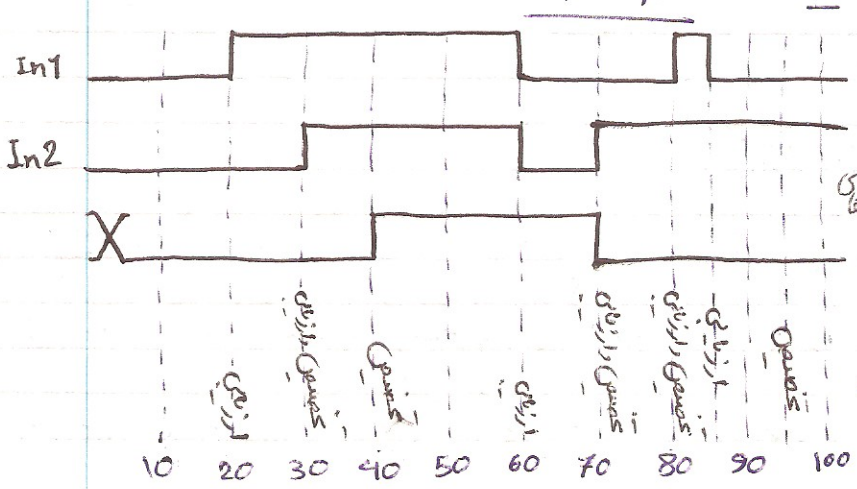
① بلافاصله پس از آپدیت شدن در لیست حساسیت یکبار خروجی ارزیابی شده و نتیجه مشخص می‌شود.

② امکان تعریف تأخیر گیت‌های منطقی در دستور assign وجود دارد.

```
assign #1 out = a & b;
```

- ارزیابی: a&b بلافاصله پس از هر نوع تغییر در لیست حساسیت، بدون تأخیر انجام می‌شود.
- تخصیص: out = نتیجه ارزیابی، پس از تأخیر تخصیص داده می‌شود.

جلسه ۱۳ — ۹۵، ۸، ۱۶




```
wire out;
assign #10 out = In1 & In2;
```

(۳۰)

با در نظر گرفتن این خاصیت از گیت‌های منطقی که تغییرات ورودی باعرض کمتر از تأخیر گیت، در خروجی دیده نمی‌شود

☆جلسه ۱۳ (۹۵،۸،۱۰) کارعملی با نرم‌افزار نوشتن بود!

عملگرد assign نیز کاملاً مشابه گیت های منطقی است.

☆ انواع انتساب :

Explicit Continuous Assignment → { wire out;
assign #1 out = in1 & in2;

Implicit Continuous Assignment → wire #1 out = in1 & in2;

Wire Definition → { wire #1 out;
assign out = in1 & in2;

☆ انواع عملگرها در وریلاگ :

از لحاظ تعداد عملوند
۱- تک عملوندی
۲- دو عملوندی
۳- سه عملوندی

از لحاظ نوع عملگر
- ریاضی (** توان ، % ، / ، * ، - ، +) ازنظر سنتز روی FPGA ممکن است بسیار پرهزینه باشد و فضای زیادی اشغال کند. توصیه می شود برای پیاده سازی این دستورات از IPCores سی ساخته استفاده شود.
- منطقی (&& ، || ، !) نیاز به دو عملوند تک بیتی در دو طرف خود دارند؛ چنانکه عملوندها بردار باشند، به نمایندگی از هر بردار یک بیت انتخاب خواهد کرد.
- مقایسه ای
- تساوی و تساوی حالتی
- بیتی
- کاهش
- ادغام و تکرار
- شرطی
- شیفت

آگر بیت X در بردار وجود داشته باشد. 1'bX
آگر بیت Z در بردار وجود داشته باشد. 1'bZ
آگر تمام بیت ها صفر و یک باشند. ← هم بیت ها صفر باشند. 1'b0
هم بیت ها صفر نباشند. 1'b1

A = 4'b 1101
B = 4'b 1X1X
C = 4'b 0000

1'bX 1'b1
B||A = 1'b1
1'b0 1'bX
C&&B = 1'b0

!B = 1'bX

☆ سعی کنید ابعاد عملوند ها در تمام عملگرهای ریاضی با آگاهی و دقت انتخاب شود.

حداکثر (m+n) بیت ← m بیتی * n بیتی

حداکثر (n+1) بیت ← n بیتی + n بیتی

☆ استفاده از عملگرهای شیفت به چپ و راست می تواند برای ضرب و تقسیم بر 2^n بسیار مفید باشد.
زیرا بجای استفاده از یک تقسیم کننده یا ضرب کننده پیچیده، از عملیات ساده شیفت استفاده می کند.

☆ نتیجه ی عملگرهای منطقی تنها یک بیت خواهد شد.

جلسه ۱۴ — ۹۵, ۸, ۱۷

★ عملگر های مقایسه‌ای: `>=` `>` `<=` `<`

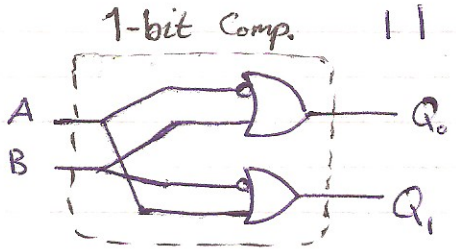
خروجی آنها تنها یک بیت خواهد شد.

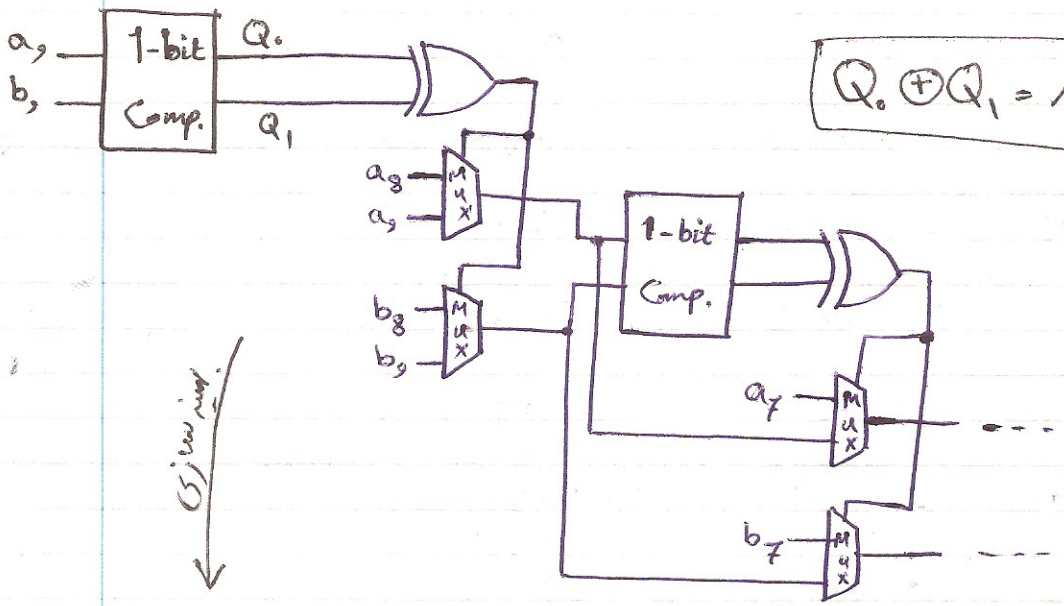
★ مقایسه کننده ی تک بیتی: $Q_1\ Q_0$

10	A>B
01	A<B
11	A=B

B	A	Q_1	Q_0
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

$$\begin{cases} Q_0 = \bar{A} + B \\ Q_1 = A + \bar{B} \end{cases}$$



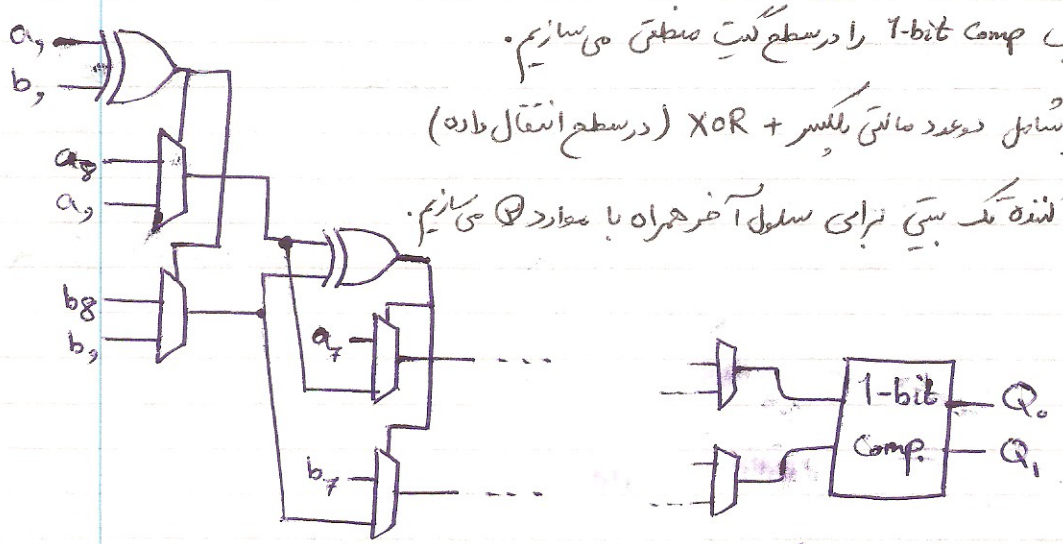


$$Q_0 \oplus Q_1 = A \oplus B$$

① ماژول وریلاگ 1-bit Comp را در سطح گیت منطقی می سازیم.

② یک ماژول با یه شامل دو عدد ماژول بالایی + XOR (در سطح انتقال داده)

③ یک مقایسه کننده تک بیتی برای سلول آخر همراه با موارد ② می سازیم.



★ مثال)

A = 4'b 1120

B = 4'b 1000

=> A>B = 1'b1

A = 1X01
B = 1101
$\Rightarrow (A<B) = 1'bX \quad , \quad (A<=B) = 1'b1$

★ عملگر های تساوی و تساوی حالتی :

!== === تساوی حالتی ← ۰,۱
!= == تساوی → X, ۱, ۰

A = 4'b 1X11
B = 4'b 1X11
$\Rightarrow (A==B) = 1'bX \quad , \quad (A===B) = 1'b1$

★ عملگر های بیتی : AND & , OR | , XOR ^ , NOT ~ , NAND ~& , NOR ~| , XNOR ~^

4'b 1011 & 4'b 0X10 → 4'b 0010

★ عملگر های کاهشی : & , | , ^ , ~& , ~| , ~^ ; &A , |A , ^A

★ عملگر های شیفت : تعداد << عدد ، تعداد >> عدد ، <<< ، >>>

A = 4'b 1101 ⇝ A >> 2 ⇒ 4'b 0011
↘ A << 2 ⇒ 4'b 0100

B = 4'b 0010 (۲) ⇝ B << 2 ⇒ 4'b 1000 (۸)

X = 4'b 1110 (-۲) ⇝ X >>> 1 ⇒ 4'b 1111 (-۱)
↘ X <<< 1 ⇒ 4'b 1100 (-۴)

★ با کمک عملگر ادغام می توانیم شیفت گردشی را نیز بسازیم .

★ عملگر ادغام : { }

```
{Cout , Sum} = A+B+ Cin ;
```

```
reg [3:0] A,B;
assign out = {A,B};
wire [7:0] out;
```

شیفت گردشی

```
reg [7:0] A;
wire [7:0] out;
assign out = {A[3:0], A[7:4]}
```

جلسه ۱۵ — ۹۵,۱,۲۳

★ عملگر تکرار:

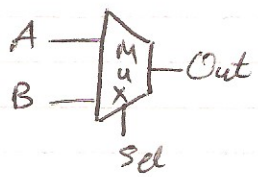
assign out = {4{A}} ⟶ out = {A,A,A,A}

این عملگر غیرقابل سنتز است.

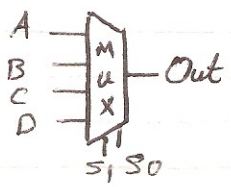
★ عملگرهای شرطی:

<condition> ? <True> : <False>

assign Out = En? In : 1'bZ;

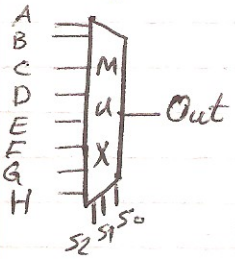


assign Out = Sel ? B : A;



S1	S0	Out
0	0	A
0	1	B
1	0	C
1	1	D

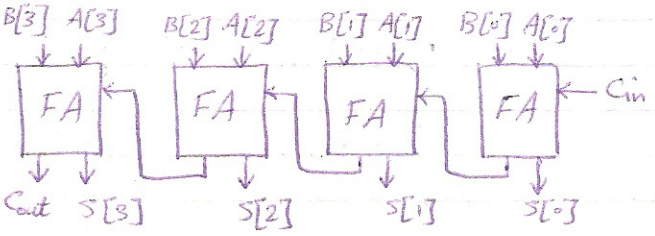
assign Out = S1 ? (S0 ? D : C) : (S0 ? B : A);



S2	S1	S0	Out
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

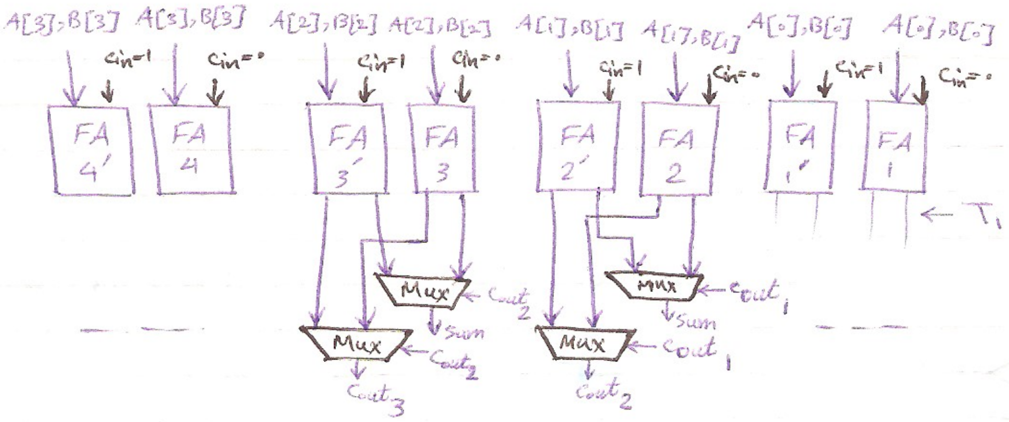
assign Out = S2 ? (S1 ? (S0 ? H : G) : (S0 ? F : E)) :
(S1 ? (S0 ? D : C) : (S0 ? B : A));

★ تغییر روند طراحی دیجیتال برای بهینه‌سازی سرعت، توان، مساحت و ... :



جمع کننده‌ی ۴ بیتی:

اگر هر FA به مقدار n نانوثانیه تأخیر داشته باشد، در کل ⟸ نانوثانیه تأخیر خواهیم داشت.



$T_p' = T_1 + 4T_{max}$

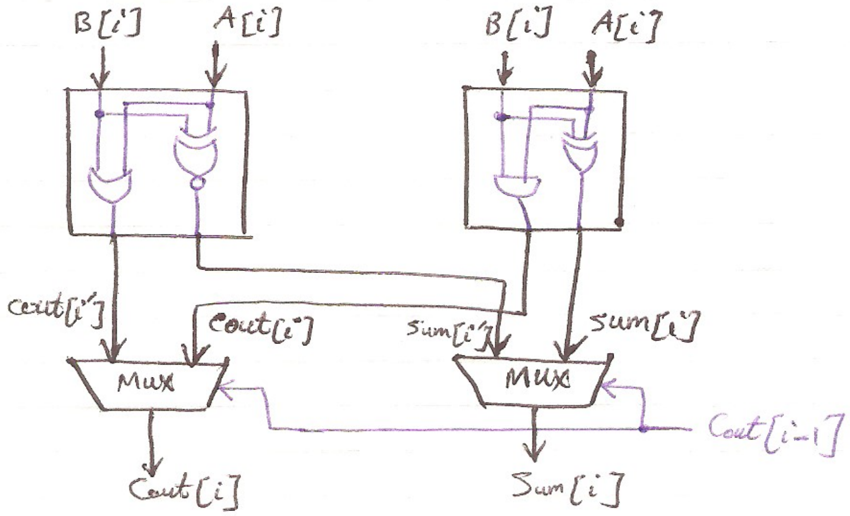
T_1 ← برای FA ها ، $4T_{max}$ ← برای هر Mux

FA, $C_i = 0$:

$$\begin{cases} Sum[i] = A_i \oplus B_i \oplus C_i \longrightarrow A_i \oplus B_i \\ Cout[i] = A_iB_i + A_iC_i + B_iC_i \longrightarrow A_iB_i \end{cases}$$

FA', $C_i = 1$:

$$\begin{cases} sum[i] = A_i \oplus B_i \oplus C_i \longrightarrow \overline{A_i \oplus B_i} \\ Cout[i] = A_iB_i + A_iC_i + B_iC_i \longrightarrow A_i + B_i \end{cases}$$




```
module FA_Mod (Cout, Sum, A, B, Cin);
    output wire Cout, Sum;
    input wire A, B, Cin;
  assign sum = Cin ? (A ~^ B) : (A ^ B);
  assign cout = Cin ? (A | B) : (A & B);
endmodule
```

میتوان اینجا خلاصه کرد.

```
assign {Cout, Sum} = Cin ? {(A|B), (A ~^B)} :
                           {(A&B), (A ^ B)};
```

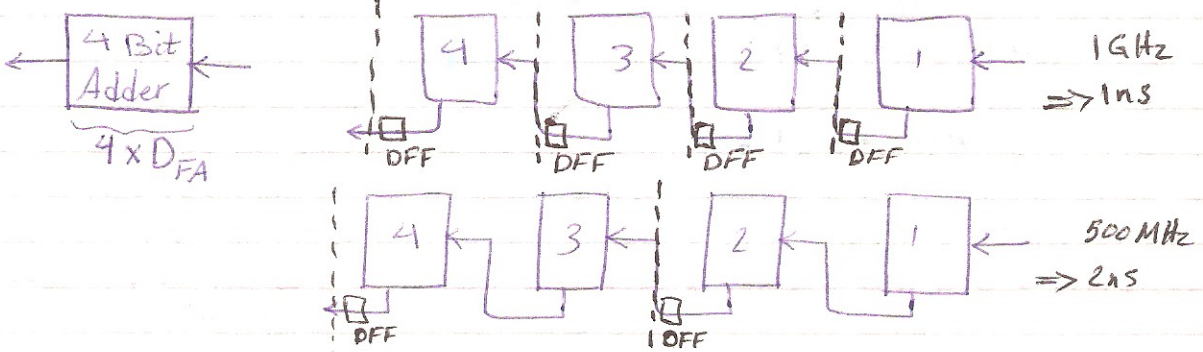
★ میانترم تا پایان این صفحه گرفته خواهد شد! (۹۵,۹,۲۱)

جلسه ۱۴ ۹۵،۱،۲۴

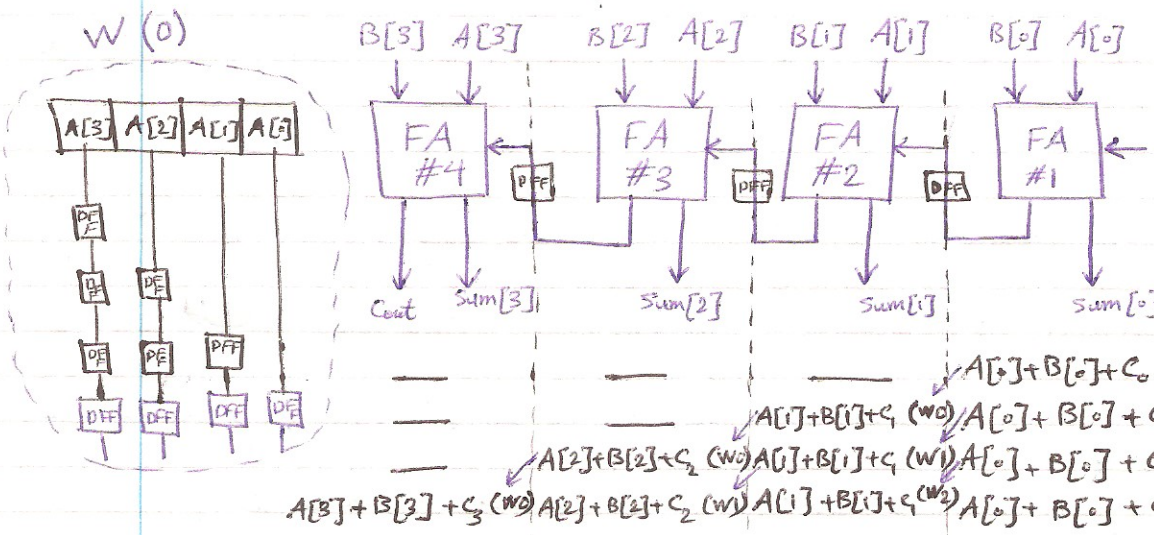
★ طراحی سیستم Pipeline :

Latency : مدت زمانی که طول می‌کشد تا اولین داده دیجیتال بعد از پردازش در خروجی ظاهر گردد.

Update rate ← دفعات به کندترین بلوک دیجیتال است.



★ تحلیل کلی :



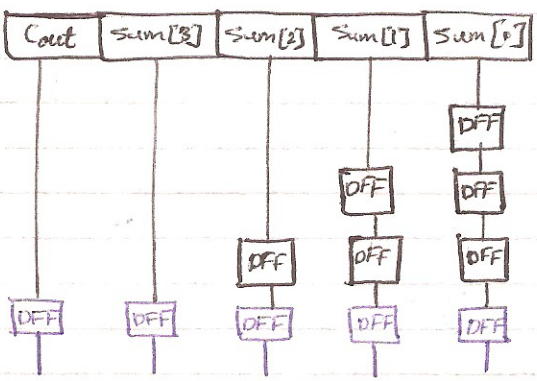
$A[0]+B[0]+C_0$ (W0) : T1

$A[1]+B[1]+C_1$ (W0) $A[0]+B[0]+C_0$ (W1) : T2

$A[2]+B[2]+C_2$ (W0) $A[1]+B[1]+C_1$ (W1) $A[0]+B[0]+C_0$ (W2) : T3

$A[3]+B[3]+C_3$ (W0) $A[2]+B[2]+C_2$ (W1) $A[1]+B[1]+C_1$ (W2) $A[0]+B[0]+C_0$ (W3) : T4

★ برای طراحی این Adder ۴ بیتی، به ۲۴ عدد DFF نیاز داریم!



مدارات ترکیبی

Combinational

Combinatorial

★ توصیف رفتاری :

- زمان نیز به عهده‌ی برنامه نویس است.
- نزدیک‌ترین سطح به درک انسان.
- always ✓ , initial ✗
- اجرای تمام بلوک‌های always, initial همزمان است.

اجرای برنامه

- متغیرها در این سطح توصیف معمولاً از جنس reg هستند.
- در این سطح نیازی به مدلسازی فیزیکی مدارها نیست و می‌توانیم به راحتی به متغیرها مقدار عددی داد.

```
initial begin
    CLK=1'b0;        // در زمان 0
    #10 CLK=1'b1;    // 10 -> 10
    #5  CLK=1'b0;    // 10 -> 15
end
```

جلسه ۱۷ — ۹۵/۹/۱

★ توصیف رفتاری:

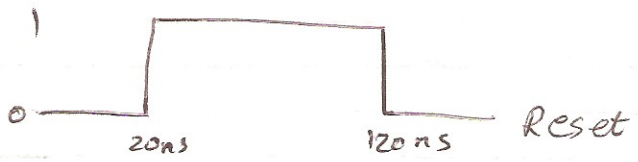
۱- initial اجرای آن در زمان صفر آغاز شده و پس از زمان‌بندی پایان می‌یابد. ✗ ق سنتز

۲- always اجرای آن در زمان صفر آغاز شده و تا پایان شبیه‌سازی ادامه دارد. ✓ ق سنتز

- سمت چپ مساوی در این سطح از جنس reg تعریف می‌شود.
- امکان اجرای موازی بلوک‌ها در این سطح‌ها وجود دارد.
- در این سطح توصیف، زمان‌بندی باید جداگانه تعریف شود.

{ begin / end } ✓ { fork / join } ✗

```
reg Reset;
initial begin
    Reset = 1'b0;
    #20  Reset = 1'b1;
    #100 Reset = 1'b0;
end
```



★ مثال) ایجاد فرکانس 50MHz در CLK:

```
initial CLK=1'b0;
initial begin
    forever #10 CLK=~CLK;
end
```

```
reg CLK=1'b0;
always
    #10 CLK=~CLK;
```

* زمانبندی در سطح رفتاری :

X ۱- براساس تأخیرها (#)

① - تأخیر معمولی

② - تأخیر درون انتسابی

③ - تأخیر صفر

اولویت اول

```
initial                 | initial
    a=1'b0;             |     #0 a=1'b0;
```

✓ ۲- براساس رخدادها (اسم رخداد) @

① - رخداد فلزکی ← مدارهای ترکیبی / مدارهای ترتیبی

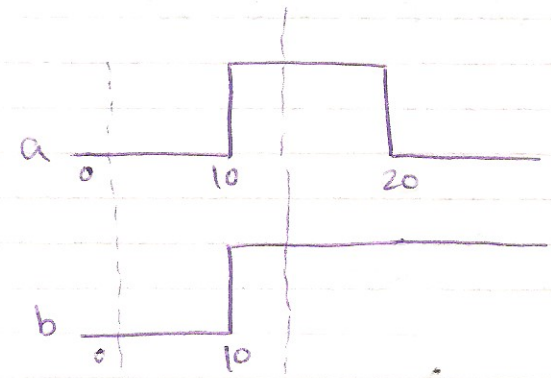
② - رخداد مجازی

③ - رخداد ترکیبی

X ۳- براساس سطح سیگنال (دستور wait)

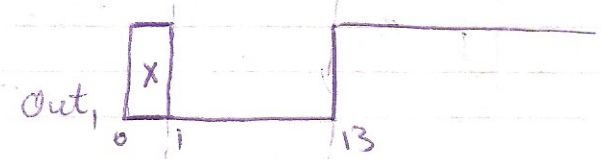
a, b

```
initial begin
    a=1'b0; b=1'b0;
#10 a=1'b1; b=1'b1;
#10 a=1'b0;
end
```



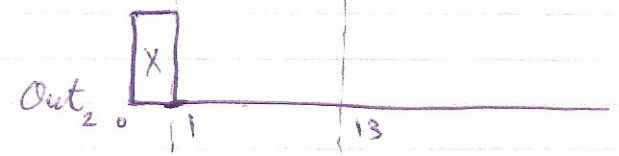
Out1

```
initial begin
    #1  out = a | b;
    #12 out = a & b;
end
```



Out2

```
initial begin
    #1  Out = a | b;
    Out = #12 a&b;
end
```



★ مثال) تاخیر با رجیستر :

```
initial  CLK = 1'b0;
initial  forever #10 CLK = ~CLK;
initial  begin
   @(CLK)   Reset = 1'b1;
   @(CLK);
end
```

★ رجیستر فلیپ فلاپ :

۱- مدار ترکیبی : لیست حساسیت نباید شامل کلمات { posedge / negedge } باشد و باید شامل تمام ورودی ها باشد.

```
always @(a,b)
   out = a&b;
```

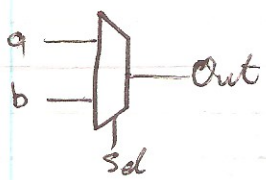
۲- مدار ترتیبی : باید از کلمات { posedge / negedge } استفاده شود.

@(posedge CLK)

0 → 1
X → 1
Z → 1
0 → X
0 → Z

@(negedge CLK)

1 → X
1 → 0
1 → Z
X → 0
Z → 0




```
always @(a,b,Sel)
   Out = Sel? b:a;
```

★ مثال) یک جمع کننده ی تک بیتی با دستور always :

```
module FA
(Sum, Cout, a, b, Cin);
output Sum, Cout;
input a, b, Cin;
reg Sum, Cout;
always @(a,b,Cin)
   {Cout, Sum} = a+b+Cin;
endmodule
```

جلسه ۱۸ ۹۵/۹/۷

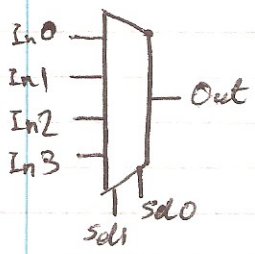
★ گام های برنامه نویسی ماژول های رفتاری :

① بلوک رفتاری را تعیین وضعیت می کنیم ، اگر ترکیبی بود ؛ قانون حضور تمام ورودی ها در لیست حساسیت آن را وصل می کنیم . اگر ترتیبی بود ، از کلمات { posedge / negedge } استفاده می شود .

② اگر ماژول و رابط تلفیقی از بلوک های ترکیبی و ترتیبی بود ، ابتدا به بخش های کوچکتر ترکیبی خالص و ترتیبی خالص ، تقسیم بندی می کنیم . برای هر بلوک ترکیبی خالص و ترتیبی ، یک بلوک always تشکیل می دهیم .

③ تست و اجرای همزمان بلوک ها و اطمینان از عملکرد صحیح برنامه .

★ مثال) طراحی مالتی پلکسر ۴ به ۱



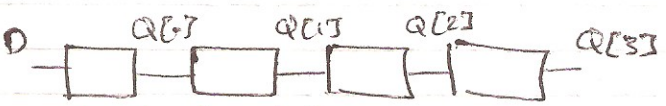
ترکیبی خالص

```
output out;
input [3:0] In;
input [1:0] Sel;
reg out;
always @(In, Sel)
    out = Sel[1] ? Sel[0] ? In[3] : In[2]
        : Sel[0] ? In[1] : In[0];
```

★ مثال) TFF

```
reg Q;
always @(posedge clk)
    Q = !Q;
```

★ مثال) شیفت رجیستر ۴ بیتی




```
output [3:0] Q;
input D;
reg [3:0] Q;
always @(posedge clk)
begin
    Q[3] = Q[2];
    Q[2] = Q[1];
    Q[1] = Q[0];
    Q[0] = D;
end
```

Blocking Assignment

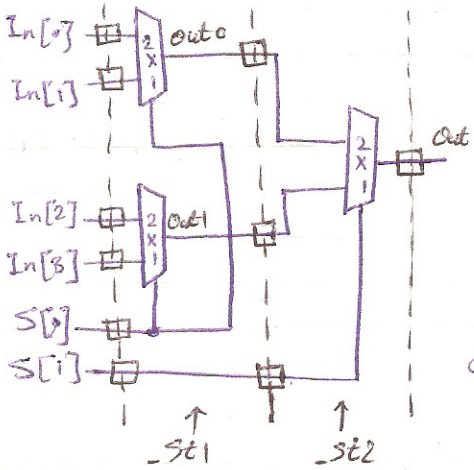
```
begin
    Q[0] <= D;
    Q[1] <= Q[0];
    Q[2] <= Q[1];
    Q[3] <= Q[2];
end
```

Non-Blocking Assignment

★ مثال) جابجایی دو متغیر A و B با استفاده از انتساب Non-Blocking :

```
always @ (posedge Clk)
    A<=B;
always @ (posedge Clk)
    B<=A;
```

★ مثال) ساختن یک Mux 4x1 با سرعت Mux 2x1 (pipeline)




```
reg [3:0] In_st1;
reg [1:0] S_st1;
reg Out0_st1, Out1_st1, Out_st2, Out0_st2,
    Out1_st2, S1_st2, Out;
always @ (In_st1, S[0]_st1)
    begin
        Out0_st1 = S_st1[0] ? In_st1[1] : In_st1[0];
        Out1_st1 = S_st1[0] ? In_st1[3] : In_st1[2];
    end
always @ (Out0_st2, Out1_st2, S1_st2)
    out_st2 = S1_st2 ? Out1_st2 : Out0_st2;
always @ (negedge clk)
    begin
        In_st1 = In;
        S_st1 = S;
    end
always @ (negedge clk)
    begin
        Out0_st2 = Out0_st1;
        Out1_st2 = Out1_st1;
        S1_st2 = S_st1[1];
    end
```

```
always @ (negedge clk)
    Out = Out_st2;
```

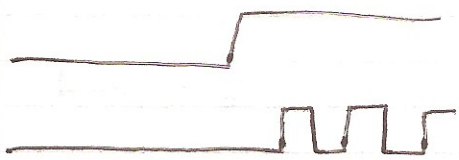
★ رخداد محاری :

```
event parity;
always @(posedge clk)
    if(Last_bit_recived)
```

تعریف event
حساسیتی → @(---)
تحریک

```
wait (Enable)
    #10 clk =! clk;
```

★ سطح سیگنال : دستور wait



★ نکته) شروط قابل سنتز :

```
if ( <Cond> )
    <true_st>;
```

X

```
if ( <Cond> )
    <true_st>;
else
    <false_st>;
```

```
if (<Cond1>)
    <st1>;
else
if (<Cond2>)
    <st2>;
else
    :
if (<Cond n>)
    <st n>;
else
    <st>;
```

جلسه ۱۹ ۹۵,۹,۱۴

```
case (<exp>)
    alt1: <st1>;
    alt2: <st2>;
      :
    alt n: <st n>;
    default: <def_st>;
endcase
```

حق نداریم X, Z استفاده کنیم.

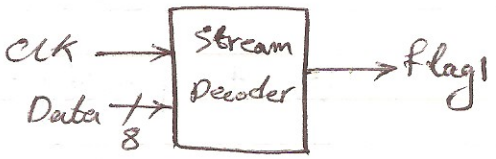
caseX (<exp>)
:
در alt ها اگر X یا Z باید بصورت Dont Care در نظر می گیرد.

caseZ (<exp>)
:
در alt ها فقط Z می تواند باید نه X.

★ همانند Switch , case های زبان C ، در اینجا هر alt که اول برقرار شود ، دقیقا بررسی می شوند.

```
module Stream_Dec (Flag1, Clk, Data);
output reg Flag1;
input Clk;
input [7:0] Data;
parameter A1=8'hFA, A2=8'hBD;
reg [1:0] Count=2'b00;
always @(posedge clk)
    if ((Data==A1) && (Count==2'b00))
    begin Count = Count + 2'b01; Flag1=1'b0; end
    else
    if ((Data == A1) && (Count == 2'b01))
    begin Count = Count + 2'b01; Flag1=1'b0; end
    else
    if ((Data == A2) && (Count == 2'b10))
    begin Count = Count + 2'b01; Flag1=1'b0; end
    else
    if ((Data==A1) && (Count == 2'b01))
    begin Count = Count + 2'b00; Flag1=1'b0; end
    else
    if ((Data==A2) && (Count == 2'b11))
    begin Flag1=1'b1; Count = 2'b00; end
    else
    if ((Data == A1) && (Count == 2'b11))
    begin Count = 2'b01; Flag1=1'b0; end
    else
    begin Count = 2'b00; Flag1=1'b0; end
endmodule
```

Stream Decoder (آشکارساز رشته) *



$A_1A_1A_2A_2$

$A_1 = 8'hFA$

$A_2 = 8'hBD$

$A_1A_1A_2A_1A_1A_2A_2$

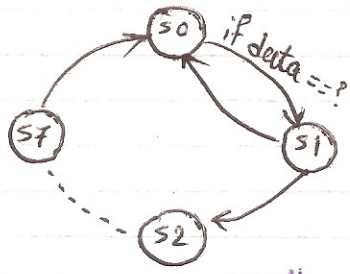
```
always @ (flag1)
    if (flag1 == 1'b1)
        Count2 = Count2 + 3'b1;
```

★ ماشین حالت (FSM) : Finite-State Machine

ترکیبی ۱- پیش بینی حالت بعدی (ورودی ها + حالت قبلی)

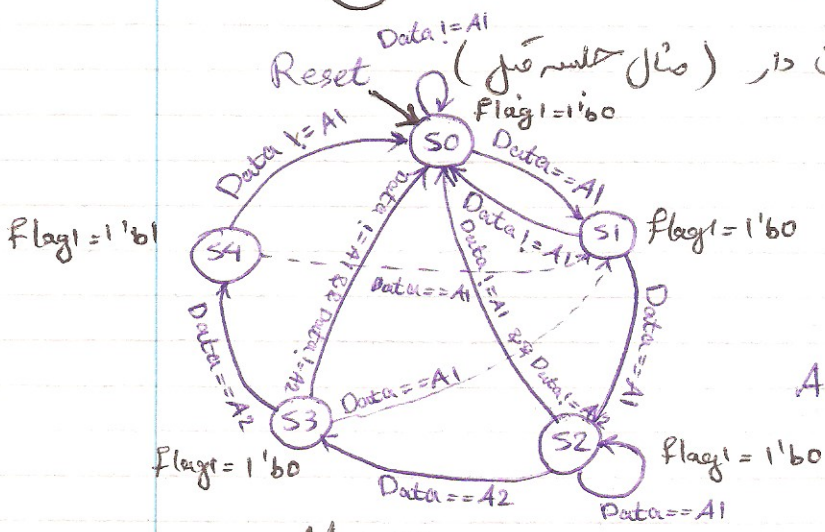
ترکیبی ۲- پیش بینی مقدار خروجی
- Moore: فقط وابسته به حالت فعلی
- Mealy: وابسته به حالت فعلی + ورودی ها

ترکیبی ۳- جابجایی بین حالت ها (انتقال بین رجیسترها)

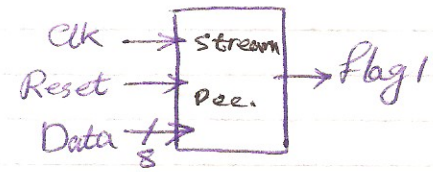


★ اگر یک FSM بزنیم هر دوره‌ای را می‌شود حتماً با ۳ دستور always پیاده سازی کرد.

★ مثال) طراحی StreamDec ریست دار (مثال جلسه قبل)



Moore



★ تمرین کدنویسی : $A_1A_1A_1A_2A_2A_1$

```
parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100;

reg [2:0] State, Next_State;

always @ (State, Reset, Data)
    if (Reset == 1'b1)
        Next_State = S0;
    else
        case (state)
```

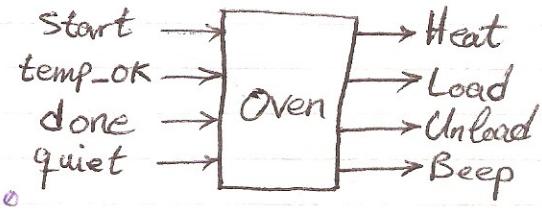
پیش بینی حالت بعدی

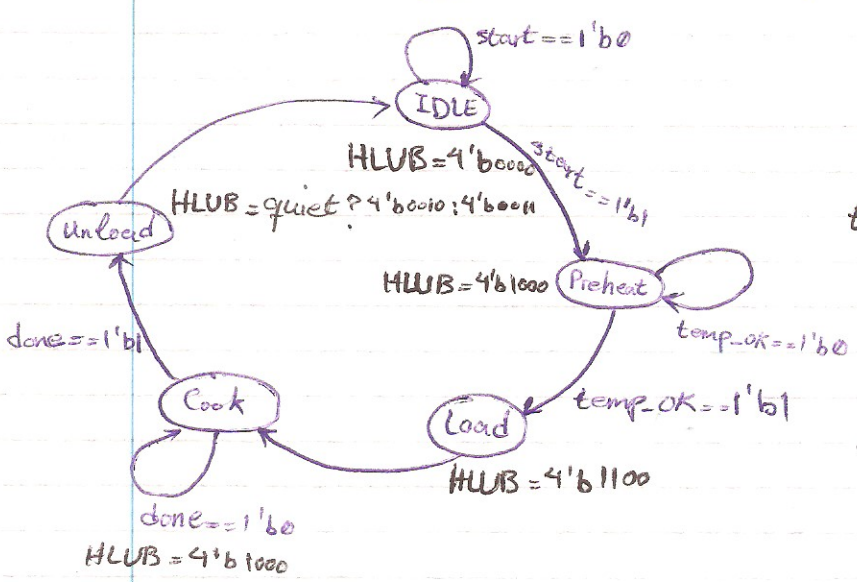
```
        S0: Next_state = (Data==A1)? S1: S0;
        S1: Next_state = (Data==A1)? S2: S0;
        S2: Next_state = (Data==A1)? S2: ((Data==A2)? S3: S0);
        S3: Next_state = (Data==A1)? S1: ((Data=A2)? S4: S0);
        S4: Next_state = (Data==A1)? S1: S0;
      default: Next_state = S0;
    endcase
always @ (state)
    case (state)
        S0: flag1 = 1'b0;
        S1: flag1 = 1'b0;
        S2: flag1 = 1'b0;
        S3: flag1 = 1'b0;
        S4: flag1 = 1'b1;
      default: flag1 = 1'b0;
    endcase
always @ (posedge clk)
    state = Next_state;
parameter A1=8'hFA , A2 = 8'hBD;

initial clk = 1'b0;
always @ #10 clk =! clk;
  :
```

جلسه ۲۲ ۹۵,۹,۲۲

جلسه ۲۱ در تاریخ ۹۵,۹,۲۱
مبانی ترم بود!

★ مثال) فر برقی!





Mealy → ! چون quiet بررسی شده

```
parameter IDLE = 3'b000, Preheat = 3'b001, Load = 3'b010
        , Cook = 3'b011, Unload = 3'b100;

reg [2:0] State, Next_State;

always @ (State, Start, temp_OK, done)
    case (State)
        IDLE: Next_State = Start ? Preheat : IDLE;
        Preheat: Next_State = temp_OK ? Load : Preheat;
        Load: Next_State = Cook;
        Cook: Next_State = done? Unload : Cook;
        Unload: Next_state = IDLE;
        default: Next_state = IDLE;
    end case

always @ (state, quiet)
    case (state)
        IDLE: {Heat, Load, Unload, Beep} = 4'h0;
        Preheat:  "  = 4'h8;
        Load:     "  = 4'hC;
```

```
Cook:   {Heat, Load, Unload, Beep} = 4'h8;
Unload:          "          = quiet ? 4'h2 : 4'h3;
default:         "          = 4'h0;
endcase
always @ (posedge clk)
    state = Next_state;
```

★ تمرین ① - سیستم گیرنده خودکار تلفن

② - دستگاه فروش نوشیدنی

1	500
2	1000
3	1500

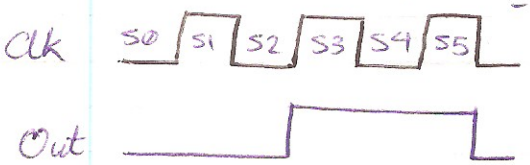
جلسه ۲۳ ۹۵/۹/۲۸

★ تقسیم کننده فرکانس به عدد فرد :

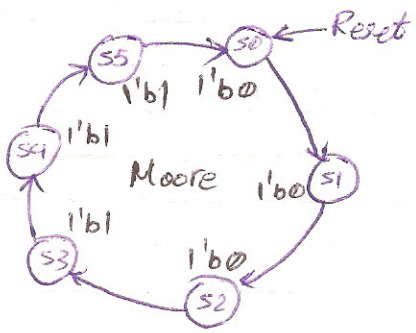
local clock ← زنجیره TFF مقوی ← $\frac{f}{2^n}$

Global clock ← CE های مجهز به TFF

Cell آخرین Q ← زنجیره از DFF و برگرداندن ← $\frac{f}{2n}$




```
reg [2:0] state, Next_state;
parameter s0 = 3'b000, s1 = 3'b001, ...;
always @ (state, Reset)
    if (Reset)
        {out, Next_state} = {1'b0, s0};
```



```
else
    case (state)
        s0 : {out , Next_state} = {1'b0, s1 }
        s1 : {out , Next_state} = {1'b0 , s2 }
        s2 : {out , Next_state} = {1'b0 , s3 }
        s3 : {out , Next_state} = {1'b1 , s4 }
        s4 : {out , Next_state} = {1'b1 , s5 }
        s5 : {out , Next_state} = {1'b1 , s0 }
      default : {out , Next_state} = {1'b0, s0 }
    endcase
always @ (clk)
    state = Next_state ;
```

★ تمرین) Duty Cycle=10% ؟ $\frac{f}{5}$

★ ساختار حلقه ها در وریلاگ :

✓ for ① تعداد دفعات اجرای حلقه باید مشخص باشد.

✓ while ② حلقه دفعات اجرا می تواند شامل ترکیبی feedback باشد. (درون بلوک به یک متغیر چند مقدار داده نشود.)

for (index = low_range ; index < high_range ; index = index+stp) غیر سنتز forever

for (index = high_range; index >= low_range ; index = index+stp) ✓ repeat

```
always @ (posedge clk)
    for (i=1 ; i<128 ; i=i+1)
        Data = Memory[i] ; → #1   Data = Memory[i]
        Memory[i] = Data ;
```

ترکیبی feedback ← فیدبک ترکیبی ندارد.

Data برای feedback ترکیبی

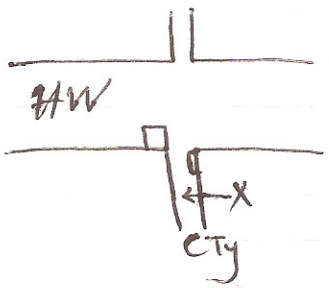
از این رفت. اما قابل سنتز نیست.

```
parameter Mem_Width = 1024;
always @ (posedge clk)
    if (Reset)
        for (i=0; i< Mem_Width ; i=i+1)
            mem[Mem_Width-1:0] = 8'h00;
    else ---
```

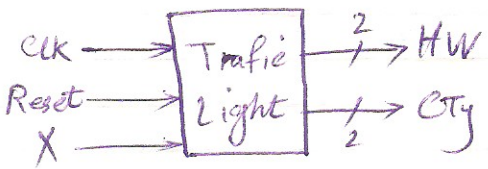
با اینکه روی X فیدبک ترکیبی داریم ولی این ساختار قابل سنتز است.

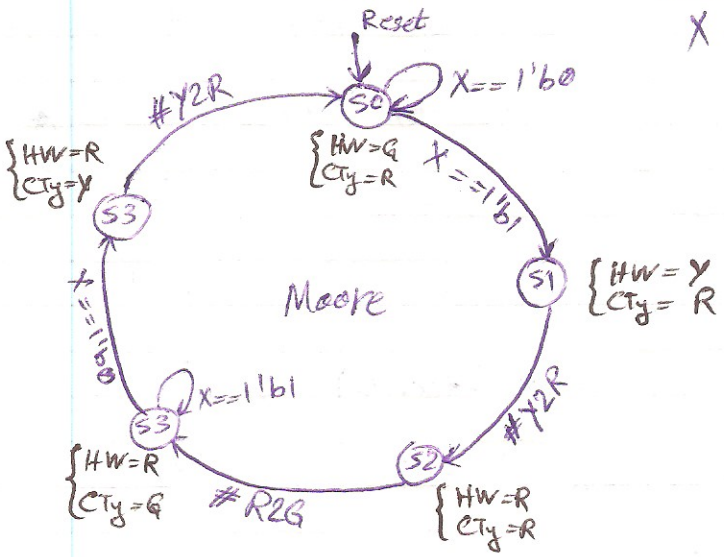
```
always @ (posedge clk)
    while (X>Y)
        Y=Y+1; ⟶
```

روی Y فیدبک ترکیبی داریم.



* مثال) اتومبیل متقاطع با مرکز





```
parameter   G = 2'b11 , Y = 2'b 10 , R = 2'b 00;
parameter   S0 = 3'b000 , S1 = 3'b001 , ...;
parameter   Y2R = 10000 , R2G = 5000;
always @ (state, X , Reset)
    if (Reset)
        Next_state = s0;
    else
        case (state)
            S0: Next_State = X ? S1:S0;
            S1: begin
                    X repeat (Y2R) @ (posedge clk);
                end Next_state = S2;
            S2: begin
                    X repeat (R2G) @ (posedge clk);
                end Next_State = S3;
            S3: Next_State = X ? S3:S4;
            S4: begin
                    X repeat (Y2R) @ (posedge Clk);
                end Next_State = S0;
            default: Next_State = S0;
        endcase
```

```
always @ (state)
    case (state)
        S0: {HW, CTy} = {G, R};
        S1:  "   "  =  {Y, R};
        S2:  "   "  =  {R, R};
        S3:  "   "  =  {R, G};
        S4:  "   "  =  {R, Y};
        default:  "   "  =  {G, R};
    endcase

always @ (posedge clk)
    state = Next_state;

always @ (posedge clk)
    if (state == S1)
        count = count + 14'b1;
        if (count == 10000)
            flag1 = 1'b1;
    else if (state == S2)
        count = count + 14'b1;
        if (count == 5000)
            flag2 = 1'b1;
    else if (state == S4)
        count = count + 14'b1;
        if (count == Y2R)
            flag3 = 1'b1;
```

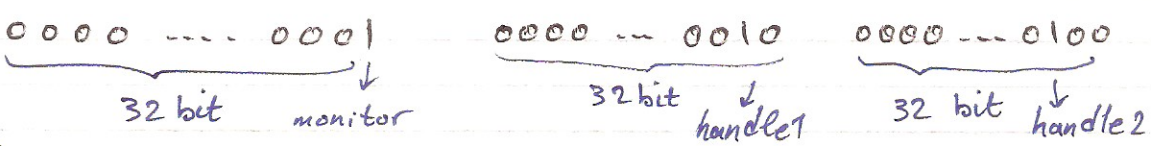
★ کار با فایل ها:

۱- باز کردن فایل $fopen

۲- نوشتن در فایل $fdisplay, $fmonitor

۳- بستن فایل $fclose

```
integer handle1;
handle1 = $fopen ("output.txt");
handle2 = $fopen ("out2.txt");
$fmonitor (handle2, $time, "----", لیست حساسیت);
desc1 = handle1 | handle2 | 32'b1;
$fclose (handle1);
```




```
initial
    handle1 = $fopen ("...");
initial
    $fmonitor (---);
initial
    #1000000 $fclose (handle1);
```

★ کار با توابع:

۱- task ← ورودی نداشته باشد.
← بیش از ۱ خروجی داشته باشد.
← دارای زمان بندی باشد.

۲- function ← باید دارای حداقل یک ورودی باشد.
← تنها یک خروجی دارد.
← نمی تواند شامل زمان بندی باشد.

★ تعریف توابع : باید در داخل بدنه‌ی اصلی ماژول انجام شود.

- احضارش در داخل بلوک های زمانی (always/ initial) انجام می‌شود.