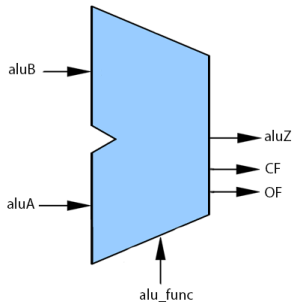
Single Cycle computer

ماژول های کلی مدار به صورت یک ALU برای محاسبات منطقی مانند جمع و تفریق و از اعمالی از این قبیل ، یک بانک رجیستر که شامل 8 رجیستر هر کدام به طول 8 بیت به نام RegBank ، یک حافظه برای ذخیره دستورات به نام IM ، یک حافظه جداگانه برای ذخیره داده های تولید شده یا ورودی در مدار به نام DM و در نهایت یک ماژول کلی به نام core که با سیم بندی و استفاده از رجیستر های مجازی درون کد این 4 ماژول گفته شده بالا را بهم متصل و با تولید سیگنال های کنترلی به آنها دستورات لازم برای کار را میدهد.

ماژول ALU:



این ماژول با دریافت سه ورودی به شرح دو عدد 8 بیتی و یک تابع انجام عملیات مورد نظر سه خروجی که یکی از آنها خروجی 8 بیتی aluZ و دوتای دیگری دو پرچم OF و CF که به ترتیب برای تشخیص سرریز و بیت رقم نقلی هستند، را تولید میکند.

برای مثال در عملیات جمع اینگونه عمل میشود:

```
5'b00001://ADD
begin
    {CF,aluZ} <= aluA + aluB;
    if((!aluA[7] & !aluB[7] & aluZ[7]) | (aluA[7] & aluB[7] & !aluZ[7]))
        OF <= 1;
    else
        OF <= 0;
end
```

تشخیص سرریز با مداری منطقی که در غالب یک شرط آمده انجام میشود.

دیگر دستورات نیز مانند همین دستور برای خود یک تابه 5 بیتی مخصوص دارند که با فراخوانی آن در بخش کنترل خروجی ALU نسبت به ورودی ها مشخص شده و به ماژول core بازمیگردد

ماژول DM :

یک حافظه که دارای 256 سطر با محتوای 8 بیتی است.

نوشتن در آن سنکرون و خواندن آسنکرون است.

با داشتن آدرس میتوان از سطری خواند یا در آن سطر نوشت.

```
module DM (clk, we, addr, din, dout);
    input        clk;
    input        we;
    input  [7:0] addr;
    input  [7:0] din;
    output [7:0] dout;

    reg [7:0] dm [0:255];

    always @(posedge clk)
        if (we)
            dm[addr] = din;

    assign dout = dm[addr];
endmodule
```

ماژول IM :

یک حافظه که دارای 256 سطر با محتوای 16 بیتی است.

برای شبیه سازی سطر های 0 تا 6 مقدار گرفته اند که جلوتر این کد های باینری مورد برسی قرار خواهد گرفت.

در این حافظه با دادن آدرس که همان PC ما است میتوان محتوای سطری را از آن خواند.

```
module IM (addr, dout);
    input  [7:0] addr;
    output [15:0] dout;
    reg [15:0] im [0:255];
    initial begin
        im[0] <= 16'b1100001100001010;
        im[1] <= 16'b0000001111010000;
        im[2] <= 16'b0000000001001010;
        im[3] <= 16'b0000010100011010;
        im[4] <= 16'b1001000000000000;
        im[5] <= 16'b1101000101100100;
        im[6] <= 16'b1100110001100100;
    end
    assign dout = im[addr];
endmodule
```

ماژول RegBank :

این ماژول با گرفتن دو آدرس به نام های R1 و R2 محتوای آن دو رجیستر مربوطه را در خروجی میگذارد و نوشتن در این ماژول همیشه در R1 انجام شده و به صورت سنکرون است و خواندن از آن آسنکرون.

```
module RegBank(clk, R1, R2, WB, RegWrite, regA, regB);
    input        clk;
    input  [2:0] R1;
    input  [2:0] R2;
    input        RegWrite;
    input  [7:0] WB;
    output [7:0] regA;
    output [7:0] regB;

    integer i;

    reg [0:7] regBank [7:0];

    initial
        for (i=0;i<8;i=i+1)
            regBank[i] = 8'd0;

    always @(posedge clk)
        if(RegWrite)
            regBank[R1] <= WB;

    assign regA = regBank[R1];
    assign regB = regBank[R2];
endmodule
```

ماژول core :

این ماژول تنها ورودی که دارد یک ریست و یک کلاک است. یک خروجی نمادین هم فقط برای سنتز شدن مدار قرار داده شده.

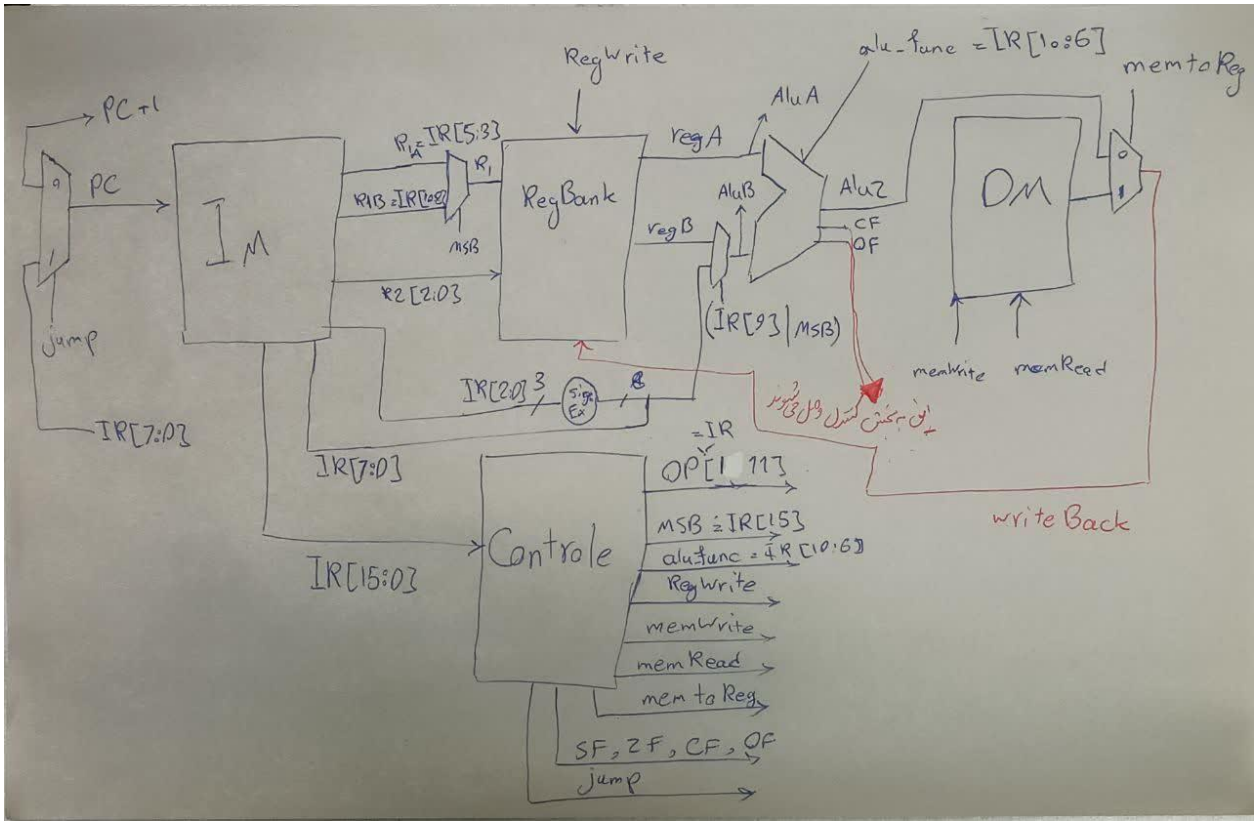
با دقت به MSB = IR[15] و Op = IR[14:11] که وابسته به دستور است سیگنال های کنترلی تولید و به مقصد خود وصل میشوند.

برای مثال: تمامی دستورات منطقی سیگنال های کنترلی یکسانی دارند پس ابتدا آنها را ساخته و سپس فقط با توجه به func = IR[10:6] سیگنال alu_func را تولید کرده و پرچم های ZF , CF , OF , SF را طبق دستورات بروزرسانی میکنیم. با تعیین تابع ALU این واحد شروع به کار میکند و خروجی مورد نظر مارا میسازد.

در این ماژول یک زیرماژول در غالب یک بلاک always امده که نقش مدیریت PC را دارد.

در صورتی که دستور جامپ باشد و طبق شرایط پرشی باید صورت گیرد آدرسی که از دستور پرش آمده در PC قرار گرفته و به سطر متناظر آن دستور در IM میپرد و از آنجا اجرا خود را ادامه میدهد. اگر پرش صورت نمیگرفت PC یک واحد بیشتر میشود و به خط بعدی در IM اشاره خواهد کرد. حالا برای اینکه ما خط حاضر را داشته باشیم باید یک واحد از PC از غقب باشیم تا دستوری جا نیافتد، پس برای اینکار یک رجیستر مجازی میگیریم (به نام iaddr) تا ابتدا مقدار PC در آن ذخیره شده و سپس PC یک واحد بیشترشود، منتها اگر پرشی صورت گیرد باید این رجیستر با ادرس پرش ست شود.

در نهایت در core از ماژول های دیگر instance میسازیم تا اتصالات را بین ماژول ها برقرار کنیم.


PC+1
PC
jump
IM
RegWrite
alu_func =IR[10:6]
memtoReg
AluA
RegA
RegBank
AluB
RegB
Aluz
CF
OF
DM
memWrite
memRead
write Back
IR[7:0]
IR[15:0]
Controle
OP[1 11]
MSB =IR[15]
aluFunc = IR[10:6]
RegWrite
memWrite
memRead
memtoReg
SF, ZF, CF, OF
jump

برای تست عدد 1 تا 10 را باهم جمع میکنیم.

اسمبلی دستورات به صورت روبرو است:

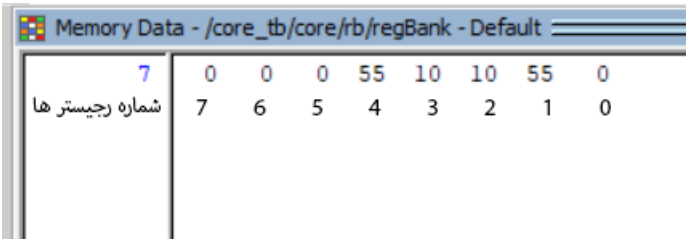
```
0: li   r3,10
1: inc  r2
2: add  r1,r2
3: cmp  r3,r2
4: ja   0
5: sm   r1,100
6: lm   r4,100
```

کد باینری همین دستورات این صورت خواهد بود:

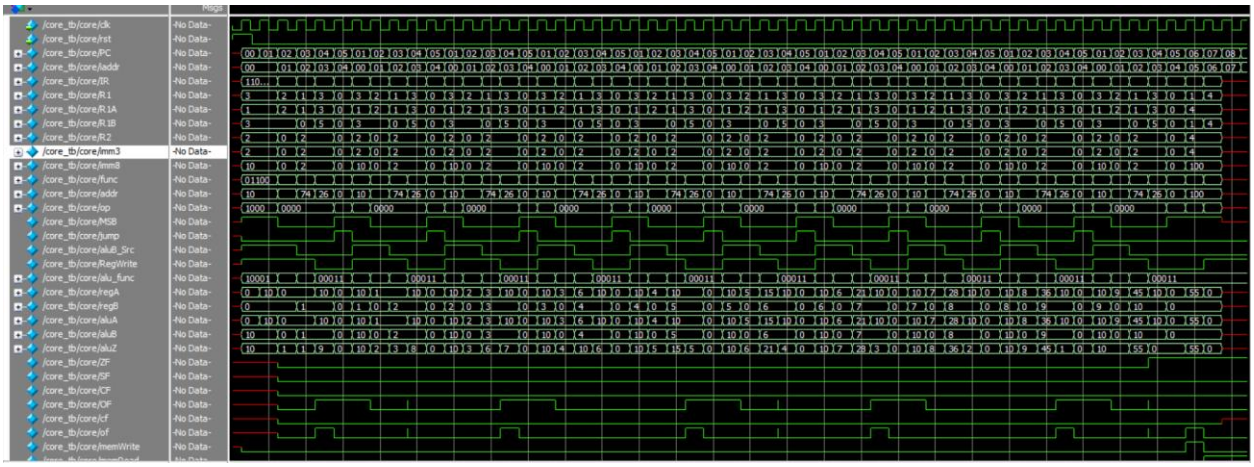
```
0: 1100001100001010
1: 0000001111010000
2: 0000000001001010
3: 0000010100011010
4: 1001000000000000
5: 1101000101100100
6: 1100110001100100
```

در خط 0 در رجیتسر 3 مقدار 10 را میریزد و خط یک مقدار رجیستر 2 را یکی بیشتر میکند. در خط دو رجیتستر 1 و 2 را جمع کرده و حاصل را توی 1 میریزد. در خط بعد رجیستر 3 تا از 2 کم میکند تا بفهمد ایا رجیستر 2 به 10 رسیده که از جامپ رد شود یا نه. در خط چهار در صورت برقرار بودن شرط جامپ طبق لیست دستورات به خط 0 جامپ میکند. تا زمانی از حلقه ادامه میابد که رجیستر 2 به 10 برسید. آنگاه از حلقه بیرون امده و مقدار رجیستر 1 که جمع اعداد 1 تا 10 است که برابر با 55 هست را در سطر DM 100 ذخیره کرده و در خط آخر از همان سطر میخواند و در رجیستر 4 میریزد.

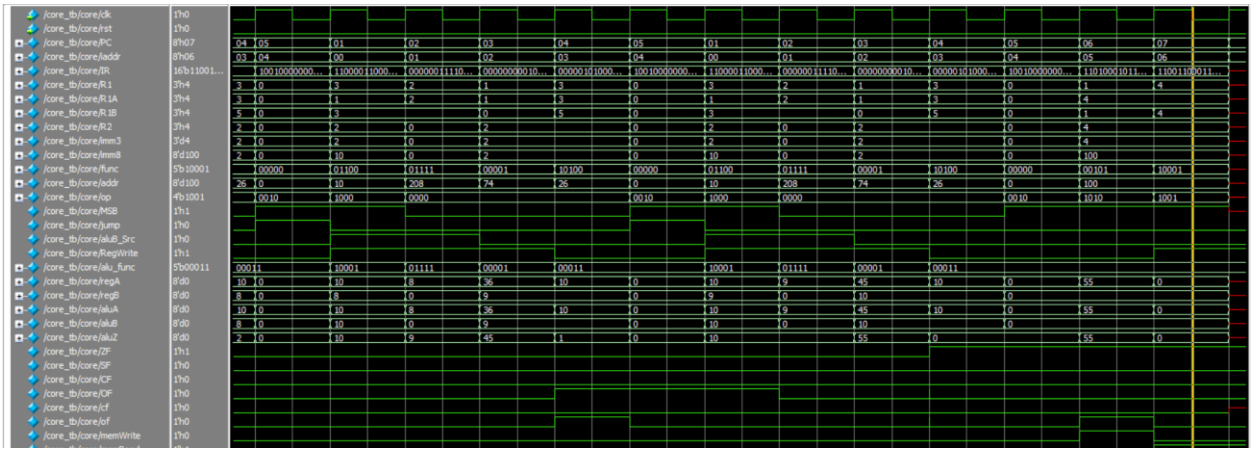
RegBank در پایان اجرای همه دستورات به صورت زیر است:



خروجی مدار در صورت کلی اینگونه است:



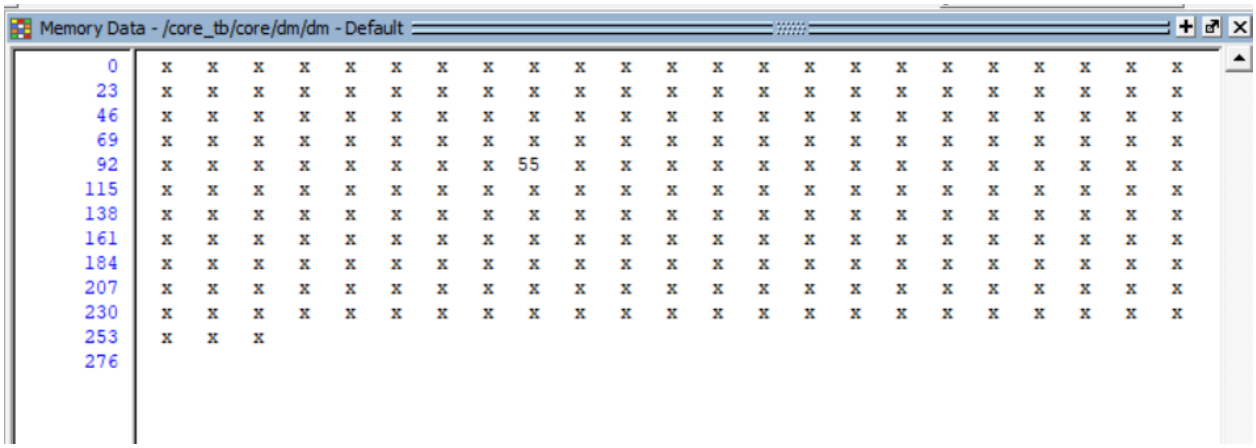
بخش آخر شکل موج بزرگ نمایی شده:



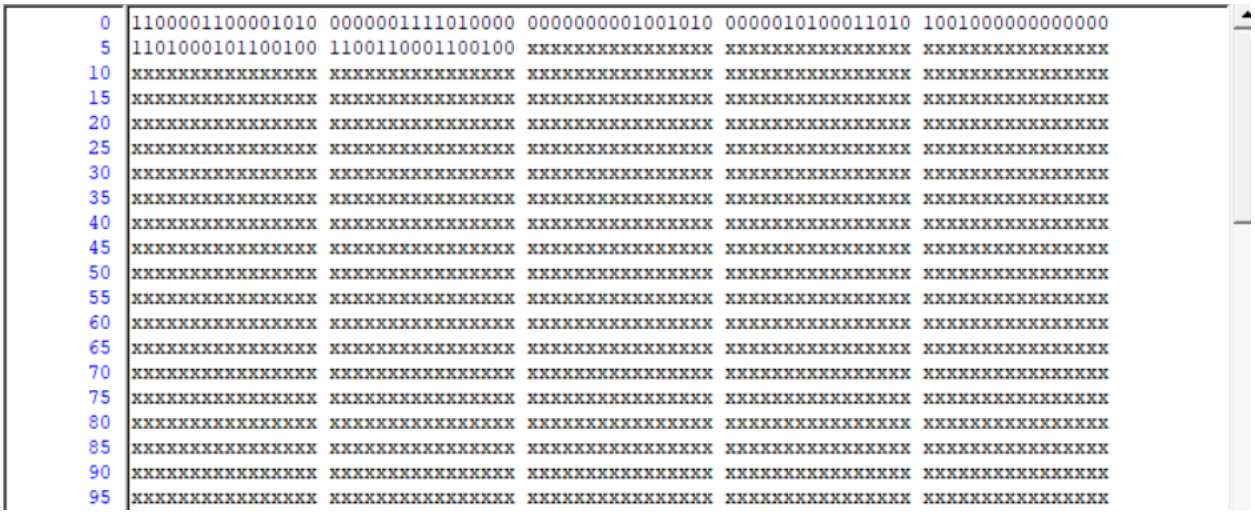
همانطور که مشاده میشود مدار مدام تا قبل ازرسیدن رجیستر 2 به مقدار 10 پس از رسیدن به خط 4 به خط 0 پرش میکند. با دقت به iaddr میتوان به این موضوع پی برد.

پس از آنکه رجیستر 2 به 10 رسید شرط حلقه نفی شده و از آن خارج میشود و دو خط نهایی یعنی ذخیره و خواندن از مموری را اجرا میکند که خروجی آن در بالا آمده که رجیستر 4 با محتوای رجیستر 1 برابر است.

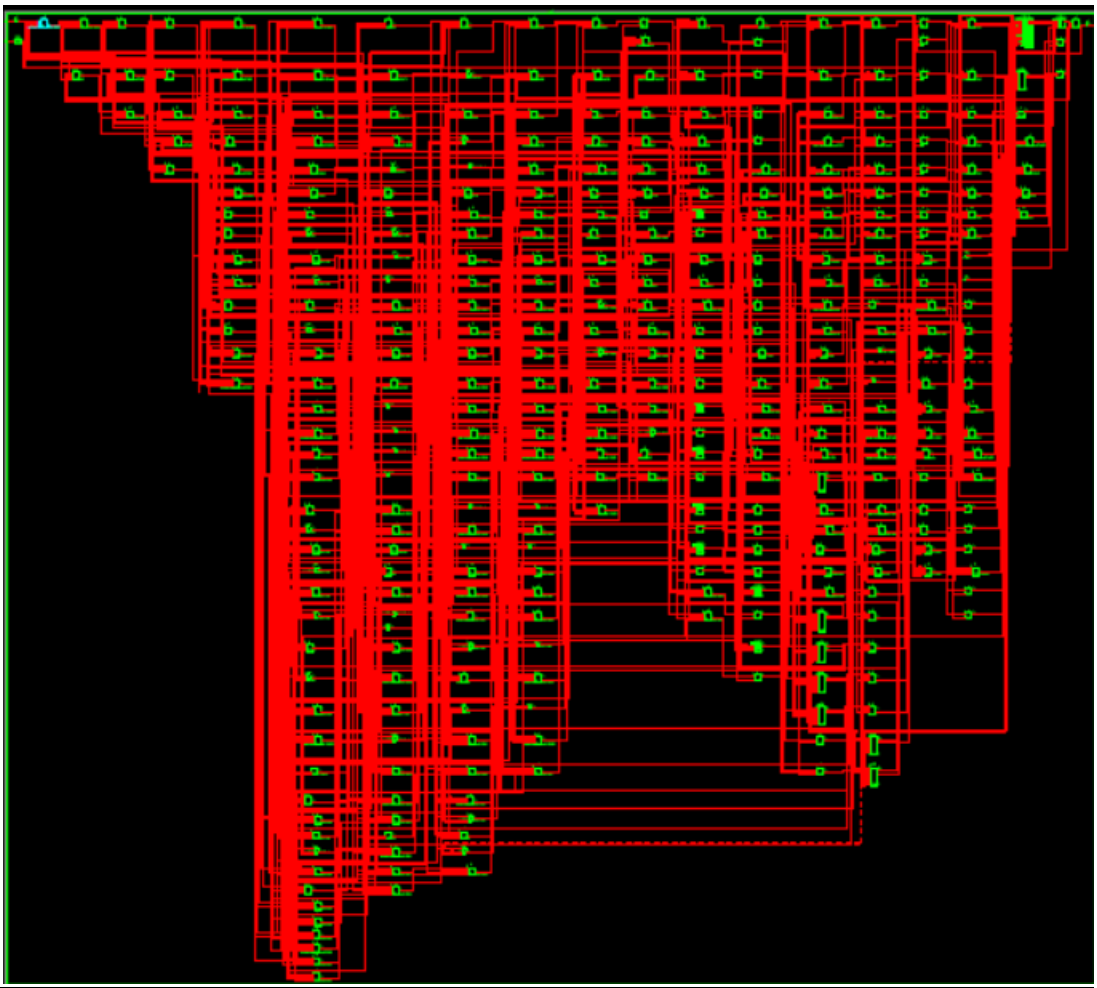
DM به صورت زیر خواهد بود. سطر 100 برابر با 55 است.

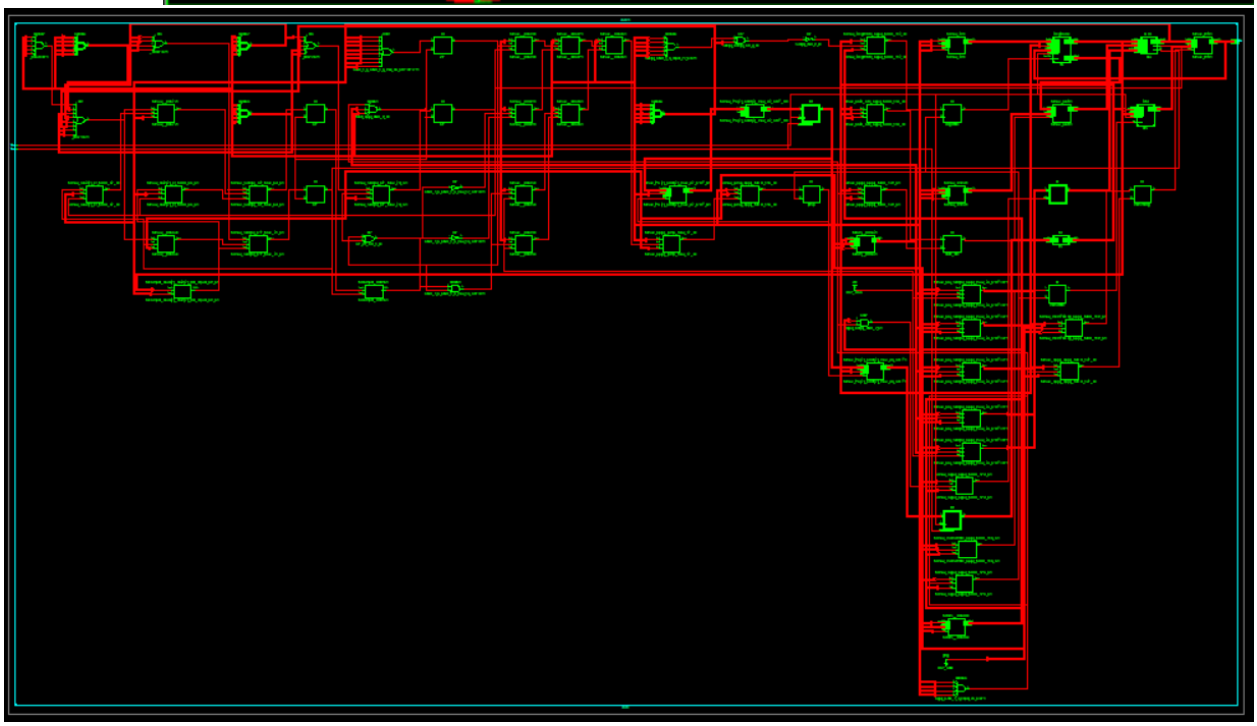


IM به صورت زیر خواهد بود. دستوراتی که به صورت پیش فرض درون آن ریختم موجودند.



Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	33	11,440	1%	
Number used as Flip Flops	8			
Number used as Latches	24			
Number used as Latch-thrus	0			
Number used as AND/OR logics	1			
Number of Slice LUTs	283	5,720	4%	
Number used as logic	241	5,720	4%	
Number using O6 output only	201			
Number using O5 output only	0			
Number using O5 and O6	40			
Number used as ROM	0			
Number used as Memory	42	1,440	2%	
Number used as Dual Port RAM	10			
Number using O6 output only	2			
Number using O5 output only	2			
Number using O5 and O6	6			
Number used as Single Port RAM	32			
Number using O6 output only	32			
Number using O5 output only	0			
Number using O5 and O6	0			
Number used as Shift Register	0			
Number of occupied Slices	96	1,430	6%	
Number of MUXCYs used	20	2,860	1%	
Number of LUT Flip Flop pairs used	283			
Number with an unused Flip Flop	251	283	88%	
Number with an unused LUT	0	283	0%	
Number of fully used LUT-FF pairs	32	283	11%	
Number of unique control sets	10			
Number of slice register sites lost to control set restrictions	32	11,440	1%	
Number of bonded IOBs	3	102	2%	
Number of RAMB16BWERs	0	32	0%	
Number of RAMB8BWERs	1	64	1%	
Number of BUFIO2/BUFIO2_2CLKs	0	32	0%	
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	1	16	6%	
Number used as BUFGs	1			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	4	0%	
Number of ILOGIC2/ISERDES2s	0	200	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	200	0%	
Number of OLOGIC2/OSERDES2s	0	200	0%	
Number of BSCANs	0	4	0%	
Number of BUFHs	0	128	0%	
Number of BUFPLLs	0	8	0%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	0	16	0%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	0	2	0%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	5.01			





تاخیر مدار نیز به صورت معکوس عدد روبرو خواهد بود.

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    6.271|         |         |         |
---------------+---------+---------+---------+---------+
```