

به نام خدا

آزمایش پنجم - واحد حافظه

امین ساوه دورودی - محمدرضا اسکینی - کامیاب عابدی

شرح اولیه

هدف از این آزمایش، آشنایی با حافظه ها و نحوه تولید، زمان بندی خواندن و نوشتن در آن ها و شبیه سازی آنها می باشد.

حافظه های بلوکی (Block RAM)، منابع مستقلی در FPGA ها هستند که در بخش هایی از آن از قبل به صورت سخت افزاری تعبیه شده اند. این حافظه ها در اندازه های مشخصی وجود دارند و میتوانند توسط پیاده ساز ها به کار گرفته شوند. این حافظه ها به صورت بلوک هایی با ظرفیت ذخیره سازی چند کیلوبیتی روی FPGA ها، قابل فرخوانی و در دسترس هستند. با توجه به نوع FPGA مورد استفاده، ظرفیت و تعداد بلوک ها متفاوت است.

نوع دیگر حافظه ها، حافظه های توزیع شده (Distributed RAM) هستند که با متصل شدن چند LUT به هم تشکیل میشوند. و میتوانند در هر قسمتی از مدار توزیع شوند. این حافظه ها قابلیت ذخیره سازی تعداد محدودی بیت را دارند، به همین خاطر در جهت ذخیره سازی حجم کمی از داده ها، از این نوع استفاده میشود. مانند حافظه های بلوکی، انواع مختلف و کارکرد های مختلفی دارند.

در پیاده سازی ماژول های مختلف، ابزار سنتز به صورت خودکار بهترین حافظه را با توجه به کارکرد ماژول انتخاب و سنتز میکند. پس نکته مهم در پیاده سازی ماژول ها، این است که به روش درست و بهینه کد نویسی انجام شود تا بهترین حافظه با توجه به عملکرد ماژول، سنتز شود.

نکته مهم دیگر، سنکرون بودن ورودی و آسنکرون بودن خروجی در حافظه های توزیع شده (Distributed RAM) میباشد که میتوان خروجی ها با روش هایی، به صورت آسنکرون درآورد که سبب افزایش کارایی این نوع حافظه میشود.

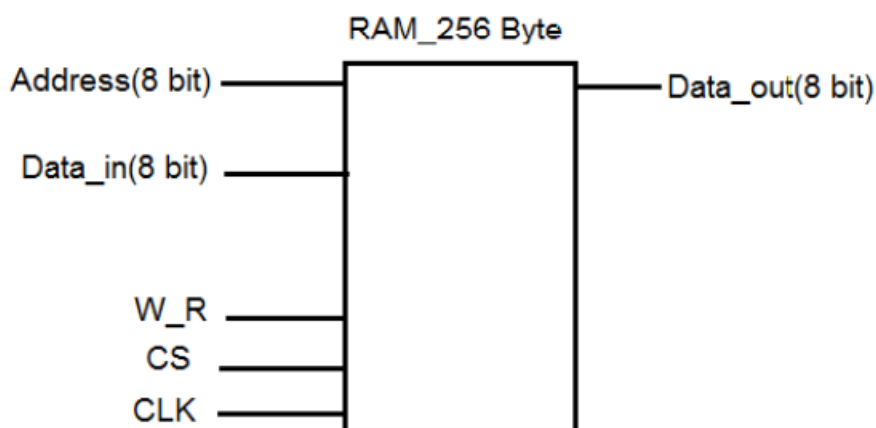
در Xilinx FPGAs، حافظه های توزیع شده به خاطر اینکه به نسبت به حافظه های بلوکی، حجم کمتری داده نگه میدارند، سریعتر میباشند و همچنین حجم کمتری نیز به خود اختصاص میدهند.

همچنین حافظه های بلوکی در سه روش میتواند استفاده شود. یعنی این امکان را به پیاده ساز میدهد که به روش های مختلف از این حافظه استفاده کند. در صورتی که حافظه ای اشتراکی این ویژگی را ندارند و تنها در یک حالت میتوانند اجرا شوند.

پیاده سازی

1- حافظه RAM 256 بایتی سنکرون

واحد حافظه برای خواندن و نوشتن داده در آدرسی مشخص استفاده میشود. حافظه طراحی شده طبق صورت گزارش کار , پنج ورودی و یک خروجی دارد. تصویر (1)



تصویر 1 – ساختار RAM

Address : آدرس جهت خواندن یا نوشتن داده روی آن آدرس از حافظه

Data_in : داده ای که قصد نوشتن آن را روی آدرس مشخص شده در حافظه داریم

CLK : با توجه به سنکرون بودن, همه کارها در لبه بالارونده (Posedge) انجام میشود.

CS : انتخاب کننده چیپ, در صورتی که 1 باشد, چیپ فعال است و در غیر این صورت خروجی برابر High impedance (Z) میباشد.

W_R : در صورتی که یک باشد به معنی نوشتن Data_in روی آدرس مشخص شده از حافظه و در صورتی که صفر باشد به معنی خواندن داده روی آدرس مشخص شده از حافظه میباشد.

```
module RAM_256 #(parameter ADDR_WIDTH = 8, DATA_WIDTH = 8, DEPTH = 256)(  
    input wire CLK,  
    input wire [ADDR_WIDTH-1:0] Address,  
    input wire W_R,  
    input wire [DATA_WIDTH-1:0] Data_in,  
    input wire CS,  
    output reg [DATA_WIDTH-1:0] Data_out);  
    reg [DATA_WIDTH-1:0] memory_array [0:DEPTH-1];  
    always @ (posedge CLK)  
    begin  
        if(!CS) Data_out <= 8'bzzzzzzzz;  
        else  
            begin  
                if(W_R)  
                    memory_array[Address] <= Data_in;  
                else  
                    Data_out <= memory_array[Address];  
            end  
        end  
    end  
endmodule
```

طبق تصویر(1) ورودی و خروجی های این ماژول تعریف میشود. سپس یک آرایه حافظه تعریف شده است. از آنجا که این حافظه سنکرون میباشد، تمامی اتفاقات در لبه بالارونده کلاک (posedge) رخ بدهد. پس با توجه به شرط هایی که در بخش پیاده سازی ذکر شده است، کد پیاده سازی شده است.

تست

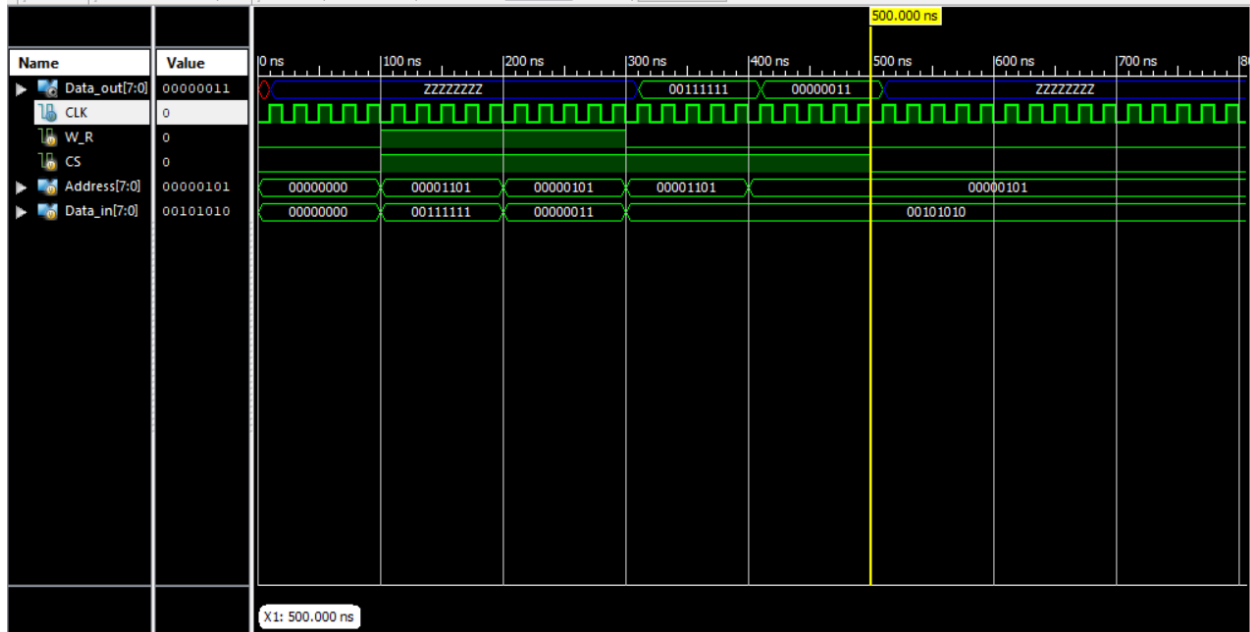
در این بخش به تست مازول پیاده سازی شده می پردازیم. کد تست بنچ نوشته شده در زیر آمده است:

```
begin
  CLK = ~CLK; #10;
end
initial
begin
  CLK = 0;
  W_R = 0;
  CS = 1'b0;      //chip gheir faal
  Address = 8'b0;  // khorooji z
  Data_in = 8'b0;
  #100;
  CS = 1'b1;
  W_R = 1'b1;
  Address = 8'b1101;  // write
  Data_in = 8'b111111;
  #100;
  Address = 8'b101;   // write
  Data_in = 8'b11;
  #100;
  W_R = 0;
  Data_in = 8'b101010; // read
  Address = 8'b1101;
  #100;
  Address = 8'b101;   //read
  #100;
  CS = 1'b0;
end
endmodule
```

طبق تست بنچ نوشته شده در بالا، انتظار داریم که ابتدا چیپ غیرفعال باشد و خروجی به ما Z را نشان دهد سپس فعال و دو مقدار مشخص شده را در دو آدرس جداگونه در حافظه بنویسد. سپس دو مقدار را از روی آدرس های مشخص شده در حافظه بخواند و در آخر غیرفعال شود و خروجی Z را به ما نشان دهد.

نکته مهم این است که در لبه بالا رونده کلاک تمامی این تغییرات اتفاق بیفتد.

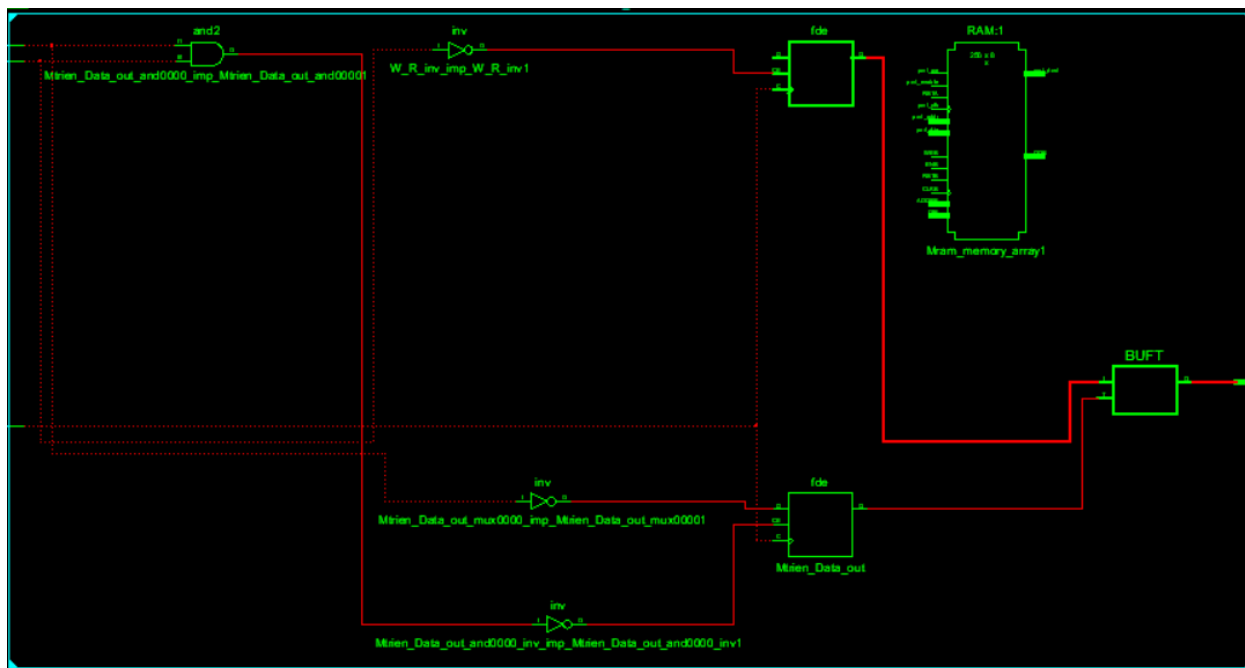
تصویر(2) , نتیجه شبیه سازی را به ما نشان میدهد که مطابق انتظار ما بوده است و نشان دهنده درستی پیاده سازی تمامی موارد بالا میباشد.



تصویر 2 - شبیه سازی

شماتیک مدار سنتز شده

تصویر (3) نیز شماتیک این حافظه را نشان میدهد.



تصویر 3 - RTL حافظه

خلاصہ طراحي

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	1	1,920	1%	
Number of occupied Slices	1	960	1%	
Number of Slices containing only related logic	1	1	100%	
Number of Slices containing unrelated logic	0	1	0%	
Total Number of 4 input LUTs	1	1,920	1%	
Number of bonded IOBs	27	83	32%	
IOB Flip Flops	8			
Number of RAMB16s	1	4	25%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	1.63			

تصویر 3-1

جدول تاخیر زمانی

Setup/Hold to clock CLK

Source	Max Setup to clk (edge)	Max Hold to clk (edge)	Internal Clock(s)	Clock Phase
Address<0>	1.097 (R)	0.457 (R)	CLK_BUF GP	0.000
Address<1>	1.394 (R)	0.220 (R)	CLK_BUF GP	0.000
Address<2>	1.383 (R)	0.228 (R)	CLK_BUF GP	0.000
Address<3>	1.129 (R)	0.421 (R)	CLK_BUF GP	0.000
Address<4>	1.086 (R)	0.466 (R)	CLK_BUF GP	0.000
Address<5>	1.648 (R)	0.016 (R)	CLK_BUF GP	0.000
Address<6>	1.435 (R)	0.186 (R)	CLK_BUF GP	0.000
Address<7>	1.140 (R)	0.422 (R)	CLK_BUF GP	0.000
CS	2.335 (R)	0.984 (R)	CLK_BUF GP	0.000
Data_in<0>	1.157 (R)	0.324 (R)	CLK_BUF GP	0.000
Data_in<1>	1.132 (R)	0.344 (R)	CLK_BUF GP	0.000
Data_in<2>	0.492 (R)	0.856 (R)	CLK_BUF GP	0.000
Data_in<3>	0.519 (R)	0.835 (R)	CLK_BUF GP	0.000
Data_in<4>	0.300 (R)	1.010 (R)	CLK_BUF GP	0.000
Data_in<5>	0.553 (R)	0.807 (R)	CLK_BUF GP	0.000
Data_in<6>	0.659 (R)	0.723 (R)	CLK_BUF GP	0.000
Data_in<7>	0.502 (R)	0.848 (R)	CLK_BUF GP	0.000
W_R	2.998 (R)	-0.080 (R)	CLK_BUF GP	0.000

Clock CLK to Pad

Destination	clk (edge) to PAD	Internal Clock(s)	Clock Phase
Data_out<0>	9.523 (R)	CLK_BUF GP	0.000
Data_out<1>	10.173 (R)	CLK_BUF GP	0.000
Data_out<2>	10.030 (R)	CLK_BUF GP	0.000
Data_out<3>	10.912 (R)	CLK_BUF GP	0.000
Data_out<4>	10.277 (R)	CLK_BUF GP	0.000
Data_out<5>	9.668 (R)	CLK_BUF GP	0.000
Data_out<6>	10.551 (R)	CLK_BUF GP	0.000
Data_out<7>	9.580 (R)	CLK_BUF GP	0.000

ROM -2

ROM پیاده سازی شده دارای هشت خط آدرس و هشت بیت دیتا میباشد. پس این ماژول، دارای دو ورودی و یک خروجی است.

کارکرد این ماژول این است که تنها یک بار روی آن داده ها با توجه به آدرس ها ذخیره میشوند و بعد از آن تنها میتوان داده های موجود در آدرس ها را خواند و دیگر نمیتوان روی آدرس های آن چیزی نوشت. البته امروزه، ROM هایی وارد بازار شده اند که توانایی نوشتن روی آنها فراهم است. برای نوشتن اولیه دو راه وجود دارد. روش اول مقدار دهی آدرس های ROM از روی فایلی از پیش ساخته شده میباشد و روش دیگر استفاده از Case برای نوشتن دستی داده ها روی آدرس های حافظه میباشد.

```
Module ROM(  
    input wire [7:0] Address,  
    input wire CE,  
    output wire [7:0] out  
);  
reg [255:0] data;[7:0]  
initial  
// first method  
$ readmemb("C:\Users\Mohammad\Desktop\Memory\data.txt", data, 0);  
    if(CE) assign out = data[Address];  
    else assign out = 8'bzzzzzzzz;  
    /* second method  
    case (Address)  
        : 0out = 8'b0;  
        : 1out = 8'b0;  
        : 2out = 8'b0;  
        : 3out = 8'b0;  
        : 4out = 8'b0;  
        : 5out = 8'b0;  
        : 6out = 8'b0;  
        : 7out = 8'b0;  
    endcase  
    /*  
Endmodule
```


تست

در این بخش به تست ماژول پیاده سازی شده می پردازیم. کد تست بنچ نوشته شده در زیر آمده است:

initial

begin

Address = 0;//0

#100;

Address = 8'b1;//1

#100;

Address = 8'b10;//2

#100;

Address = 8'b11;//3

#100;

Address = 8'b100;//4

#100;

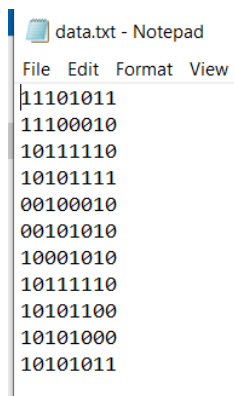
Address = 8'b101;//5

#200;

End

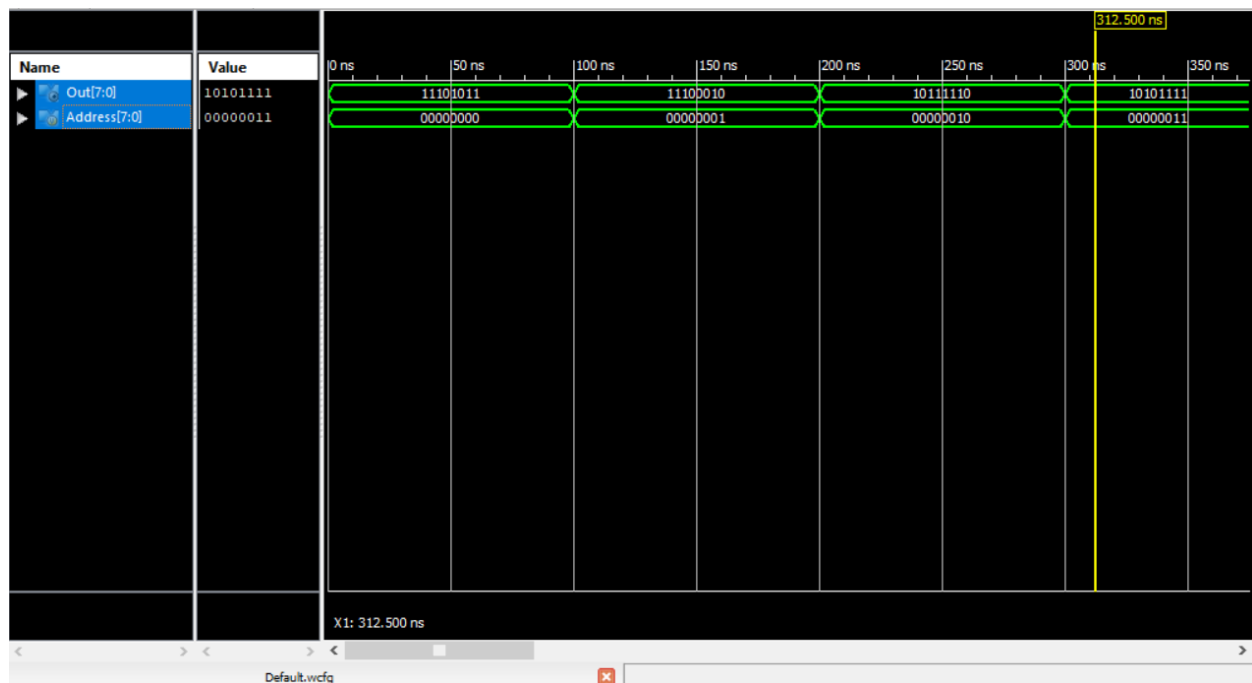
با توجه به تست بنچ بالا انتظار داریم ابتدا ROM با استفاده از داده های موجود در فایل از پیش نوشته شده ، مقدار دهی اولیه شود و سپس با استفاده از آدرس ها به داده ها دسترسی داشته باشیم.

در تصویر (4)، داده های اولیه موجود در فایل data.text نشان داده شده است که انتظار داریم شبیه سازی تست بنچ ما با توجه به آدرس خروجی یکسانی داشته باشد.



```
data.txt - Notepad
File Edit Format View
11101011
11100010
10111110
10101111
00100010
00101010
10001010
10111110
10101100
10101000
10101011
```

تصویر 4 – data.txt



تصویر 5 - شبیه سازی تست بنچ

همانطور که مشخص است , به درستی داده های موجود در ROM خوانده شده است. پس این ماژول ما نیز به درستی کار میکند.


تفاوت طراحی ROM و RAM در FPGA

رم نوعی حافظه موقتی برای نگهداری مقداری دیتا می باشد. دسترسی به این نوع حافظه، به محل دیتا بستگی ندارد. به این نوع حافظه، حافظه با دسترسی تصادفی گفته میشود به همین دلیل است که RAM به صورت **Random Access Memory** شناخته شده است. اما در صورت قطع شدن منبع تغذیه همه داده ها از دست میروند. به صورت کلی این رم ها در دو مدل ارائه میشوند:

دسته اول رم های استاتیک (SRAM) و دسته دوم رم های دینامیک (DRAM) میباشد.

نوع دیگری از حافظه ها یعنی ROM ها یک حافظه برای ذخیره سازی اطلاعات به صورت دائمی و دو دویی میباشد. میدانیم که این حافظه ها در ابتدا مقادیر را در حافظه خود قرار میدهند و سپس فقط میتوان آن ها را خواند. و این امر سبب می شود تا آن ها به راحتی قابل اصلاح نباشند. برای نگهداری داده ها به صورت بلند مدت مناسب هستند. در سیستم های کامپیوتری، حافظه مورد نیاز برای بوت شدن، از همین مدل حافظه ها میباشد. برای مثلا بایوس رایانه یک PROM است که دارای اطلاعات و برنامه اولیه برای راه اندازی را دارد.

خلاصه طراحی

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	7	1,920	1%	
Number of occupied Slices	4	960	1%	
Number of Slices containing only related logic	4	4	100%	
Number of Slices containing unrelated logic	0	4	0%	
Total Number of 4 input LUTs	7	1,920	1%	
Number of bonded IOBs	11	83	13%	
Average Fanout of Non-Clock Nets	2.60			

تاخير

Source Pad	Destination Pad	Delay
Address<0>	out<0>	6.305
Address<0>	out<2>	5.853
Address<0>	out<3>	5.570
Address<0>	out<4>	6.081
Address<0>	out<5>	5.552
Address<1>	out<0>	6.294
Address<1>	out<2>	5.809
Address<1>	out<3>	5.537
Address<1>	out<4>	6.071
Address<1>	out<5>	5.603
Address<1>	out<6>	6.041
Address<1>	out<7>	6.041
Address<2>	out<0>	6.860
Address<2>	out<2>	6.800
Address<2>	out<3>	6.159
Address<2>	out<4>	7.057
Address<2>	out<5>	6.461
Address<2>	out<6>	5.769
Address<2>	out<7>	5.735

تصوير 5-2

Cache -3

در طراحی Cache میدانیم که آدرس حافظه، از دو بیت اندیس، 6 بیت تگ ساخته شده است. این حافظه یک ماتریس دو بعدی است. در ادامه با توجه به این امر که آیا آدرس داده شده، به اصطلاح hit میشود یا نه میتوان دو حالت را متصور شد:

اگر hit نشود، مقدار رم، به عنوان دیتا نوشته میشود. و valid نیز برابر 1 میباشد. در صورتی که نیاز باشد تا دیتای در حافظه بنویسیم باید آن دیتا در کش نوشته شود. و مجدداً valid برابر با 1 شود. در صورتی که آدرس حافظه hit شود، باید دیتا را از کش خوانده و خروجی دهیم. و همانطور که بالاتر گفته شد، در غیر این صورت باید از رم خوانده شود.

Module CACHE(

```
input clk, input wire [7:0] address,
input wire [7:0] data, input wire w_r,
output wire hit, output wire [7:0] out);
reg [14:0]cache[7:0]; wire [7:0] mem_out;
wire [1:0] set; wire [5:0] cache_tag;
wire [5:0] input_tag; wire [7:0] cache_data;
wire valid;

assign set = address[1:0];
assign input_tag = address[7:2];
assign cache_data = cache[set][7:0];
assign cache_tag = cache[set][13:8];
assign valid = cache[set][14];

RAM_256 ram(clk, address, w_r, data, ~hit, mem_out);
assign hit=valid & (cache_tag == input_tag);

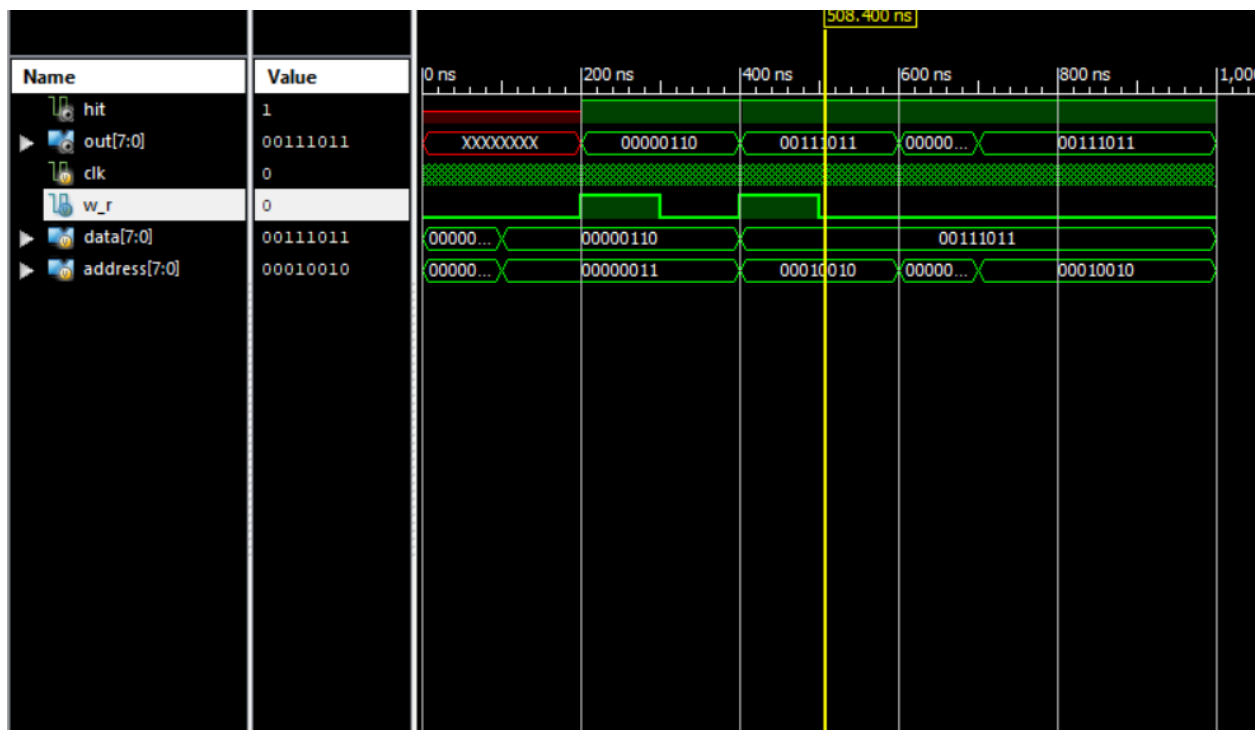
always @ (posedge clk)
begin
    if (w_r)
        begin
            cache[set] = {1'b1, input_tag, data};
        end
    else if (~hit)
        cache[set]= {1'b1, input_tag, mem_out};
    end

    assign out = hit ? cache_data : mem_out;
endmodule
```

تست

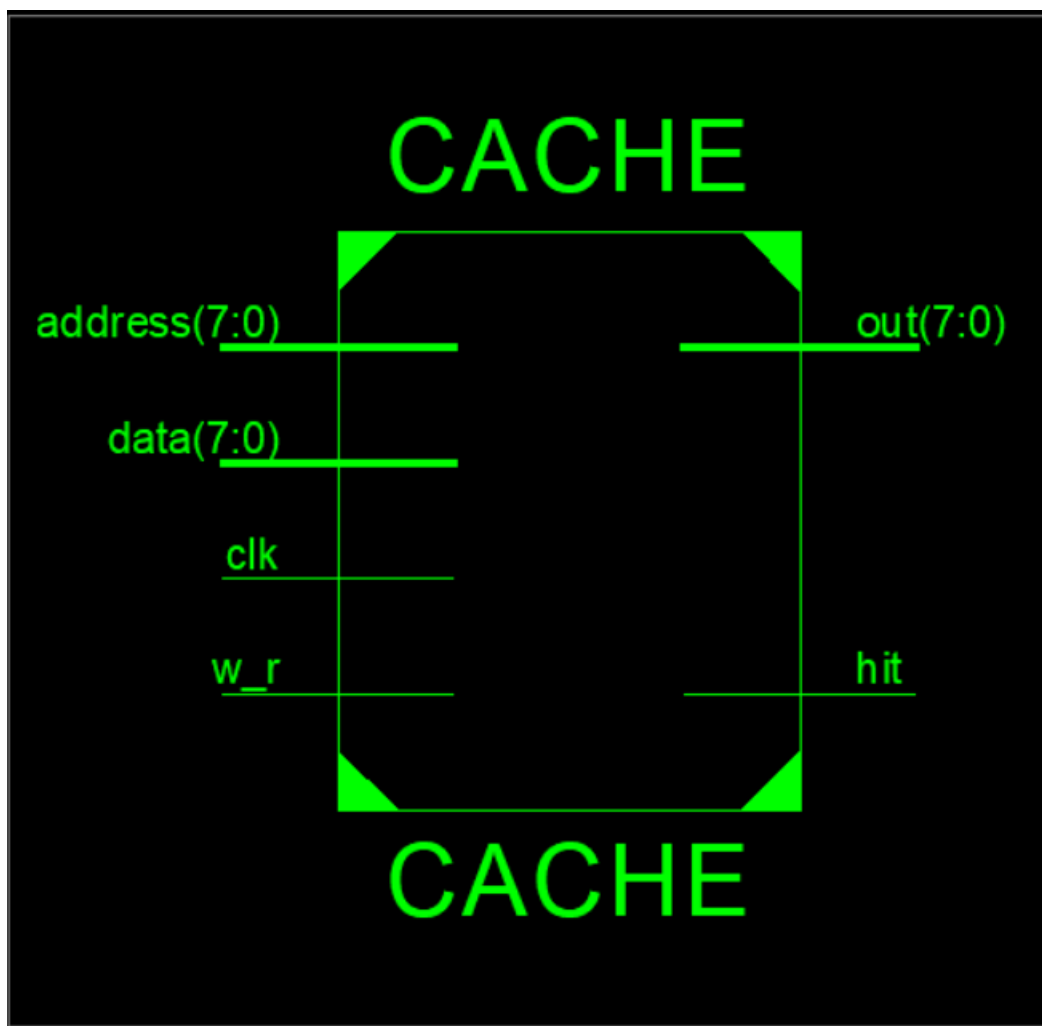
در این بخش به تست مازول پیاده سازی شده می پردازیم. کد تست بنچ نوشته شده در زیر آمده است:

```
initial
begin
    clk = 0;
    address = 0;
    data = 0;
    w_r = 0;
    #100;
    address = 3;
    data = 6;
    #100;
    w_r = 1;
    #100;
    w_r = 0;
    #100;
    w_r = 1;
    address = 18;
    data = 59;
    #100;
    w_r = 0;
    #100;
    address = 3;
    #100;
    address = 18;
    #100;
End
```



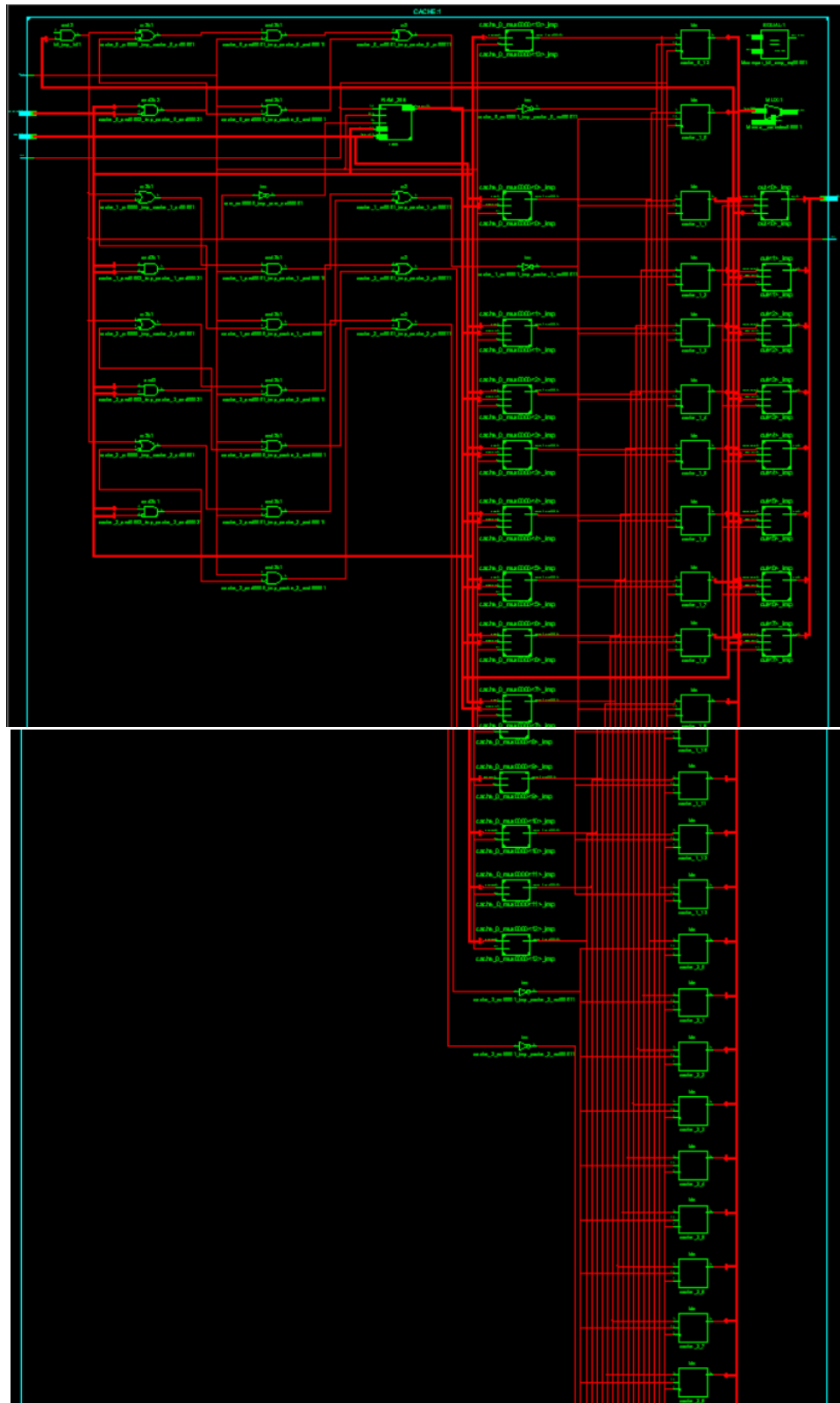
تصویر 6 - شبیه سازی تست بنچ

تصویر(6) نتیجه شبیه سازی تست بنچ را نشان میدهد که مطابق انتظارات ما بوده است. پس میتوان گفت که این ماژول نیز به درستی پیاده سازی شده است.



تصویر 7 - RTL سطح بالای ماژول

تصاویر (7و8) RTL این ماژول پیاده سازی شده را نشان میدهند.



تصویر 8

خلاصه طراحی

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	48	960	5%	
Number of Slice Flip Flops	57	1920	2%	
Number of 4 input LUTs	55	1920	2%	
Number of bonded IOBs	27	83	32%	
Number of BRAMs	1	4	25%	
Number of GCLKs	1	24	4%	

تصویر 9

تصویر (9) گزارش کلی و خلاصه از طراحی ما را نشان میدهد. همچنین میزان فضای اشغال شده نیز در تصویر بالا نمایان است.

جدول تاخیر

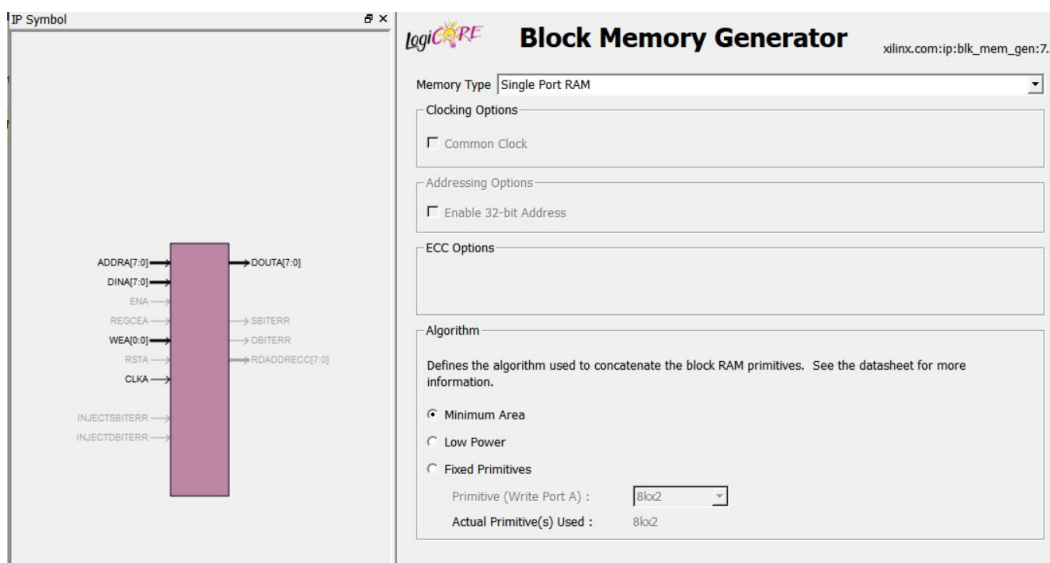
Pad to Pad		
Source Pad	Destination Pad	Delay
address<0>	hit	9.890
address<0>	out<0>	11.802
address<0>	out<1>	11.534
address<0>	out<2>	11.774
address<0>	out<3>	12.462
address<0>	out<4>	11.773
address<0>	out<5>	11.561
address<0>	out<6>	11.414
address<0>	out<7>	11.789
address<1>	hit	9.737
address<1>	out<0>	11.242
address<1>	out<1>	10.975
address<1>	out<2>	11.215
address<1>	out<3>	11.903
address<1>	out<4>	11.214
address<1>	out<5>	11.002
address<1>	out<6>	10.855
address<1>	out<7>	11.230
address<2>	hit	7.420
address<2>	out<0>	8.909
address<2>	out<1>	8.641
address<2>	out<2>	8.881
address<2>	out<3>	9.569
address<2>	out<4>	8.880
address<2>	out<5>	8.668
address<2>	out<6>	8.521
address<2>	out<7>	8.896
address<3>	hit	8.080
address<3>	out<0>	9.569
address<3>	out<1>	9.301
address<3>	out<2>	9.541
address<3>	out<3>	10.229
address<3>	out<4>	9.540
address<3>	out<5>	9.328
address<3>	out<6>	9.181
address<3>	out<7>	9.556
address<4>	hit	7.751
address<4>	out<0>	9.257
address<4>	out<1>	8.989

تصویر 9-1

ROM و RAM -4

جهت ساختن Block RAM و Block ROM , ابتدا (Core Generator & Architecture Wizard) IP را انتخاب کرده و طبق تصاویر (10 و 11) پیش می رویم.

برای پر کردن حافظه نیز یک فایل با پسوند coe ساخته و آن را مانند تصویر (12) پر میکنیم.



تصویر 10

Memory Size

Write Width	8	Range: 1..4608	Read Width:	8
Write Depth	256	Range: 2..9011200	Read Depth:	256

تصویر 11

```

MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
00,
11,
aa,
bb,
cc,
dd,
44;

```

تصویر 12

تست

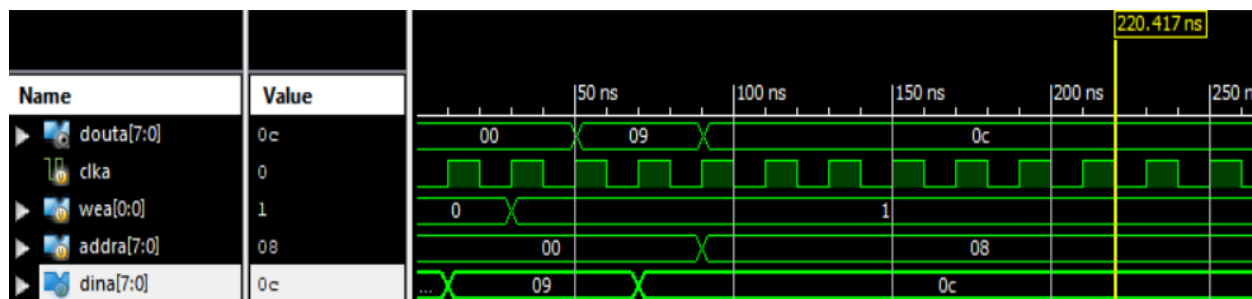
سپس تست بنچی برای آن نوشته و صحت عملکرد آن را تست میکنیم.

Initial begin

```
clka = 0;  
wea = 0;  
addra = 0;  
dina = 0;  
#10;  
dina=8'b1001;  
wea=0;  
#20;  
wea=1;  
#20;  
dina=8'b1100;  
#20;  
addra=8'b1000;  
#20;
```

End

نتیجه شبیه سازی در تصویر (13) نشان داده شده است که به درستی عمل میکند.



تصویر 13

خلاصه طراحی

Device Utilization Summary (estimated values)				
Logic Utilization	Used	Available	Utilization	
Number of Slices	48	960	5%	
Number of Slice Flip Flops	57	1920	2%	
Number of 4 input LUTs	55	1920	2%	
Number of bonded IOBs	27	83	32%	
Number of BRAMs	1	4	25%	
Number of GCLKs	1	24	4%	

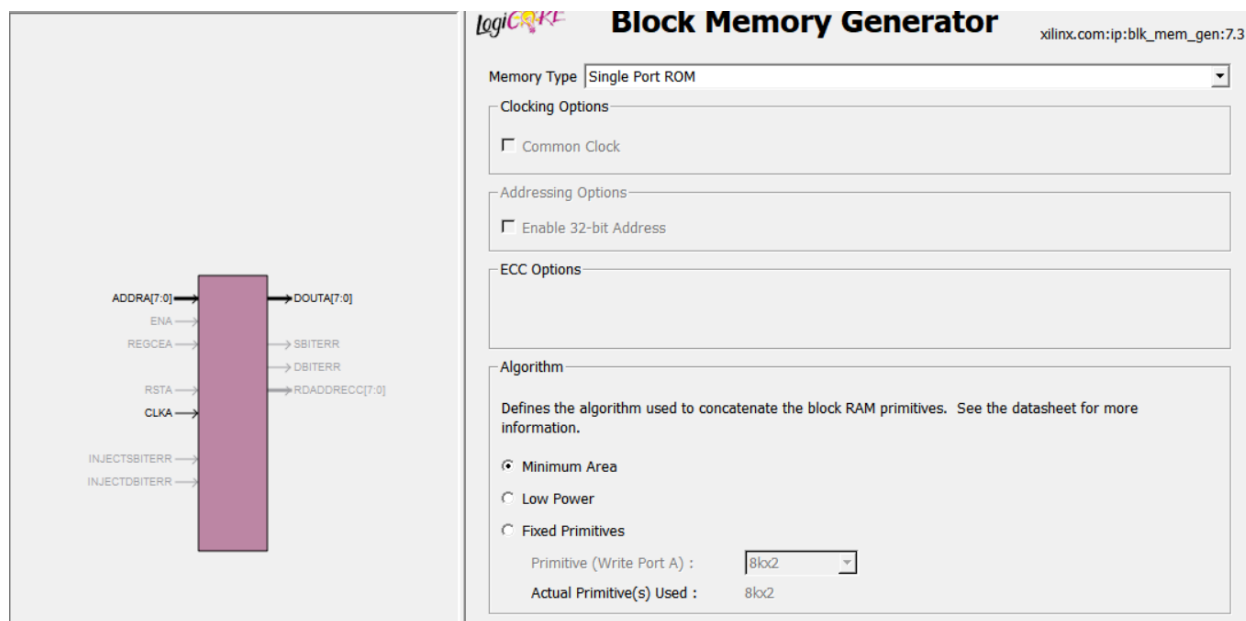
تصویر 13-1

جدول تاخیر زمانی

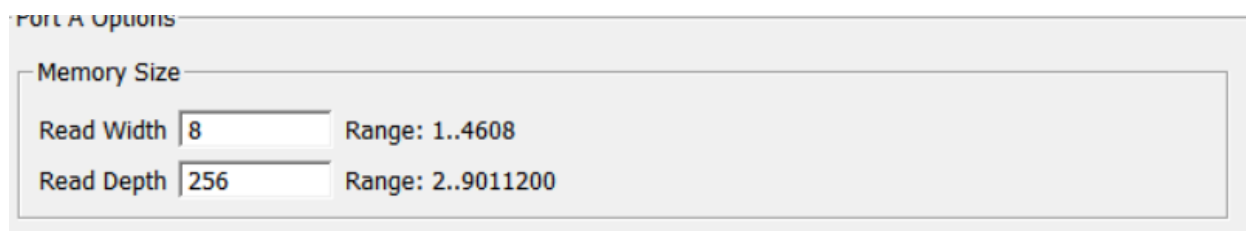
Setup/Hold to clock CLK				
Source	Max Setup to clk (edge)	Max Hold to clk (edge)	Internal Clock(s)	Clock Phase
Address<0>	1.097 (R)	0.457 (R)	CLK_BUFDP	0.000
Address<1>	1.394 (R)	0.220 (R)	CLK_BUFDP	0.000
Address<2>	1.383 (R)	0.228 (R)	CLK_BUFDP	0.000
Address<3>	1.129 (R)	0.431 (R)	CLK_BUFDP	0.000
Address<4>	1.086 (R)	0.466 (R)	CLK_BUFDP	0.000
Address<5>	1.648 (R)	0.016 (R)	CLK_BUFDP	0.000
Address<6>	1.435 (R)	0.186 (R)	CLK_BUFDP	0.000
Address<7>	1.140 (R)	0.422 (R)	CLK_BUFDP	0.000
CS	2.335 (R)	0.984 (R)	CLK_BUFDP	0.000
Data_in<0>	1.157 (R)	0.324 (R)	CLK_BUFDP	0.000
Data_in<1>	1.132 (R)	0.344 (R)	CLK_BUFDP	0.000
Data_in<2>	0.492 (R)	0.856 (R)	CLK_BUFDP	0.000
Data_in<3>	0.519 (R)	0.835 (R)	CLK_BUFDP	0.000
Data_in<4>	0.300 (R)	1.010 (R)	CLK_BUFDP	0.000
Data_in<5>	0.553 (R)	0.807 (R)	CLK_BUFDP	0.000
Data_in<6>	0.659 (R)	0.723 (R)	CLK_BUFDP	0.000
Data_in<7>	0.502 (R)	0.848 (R)	CLK_BUFDP	0.000
W_R	2.998 (R)	-0.080 (R)	CLK_BUFDP	0.000

تصویر 13-2

جهت ساختن Block ROM نیز همانند بالا عمل میکنیم. تصاویر (14 و 15) نحوه ایجاد آن را نشان میدهند.



تصویر 14



تصویر 15

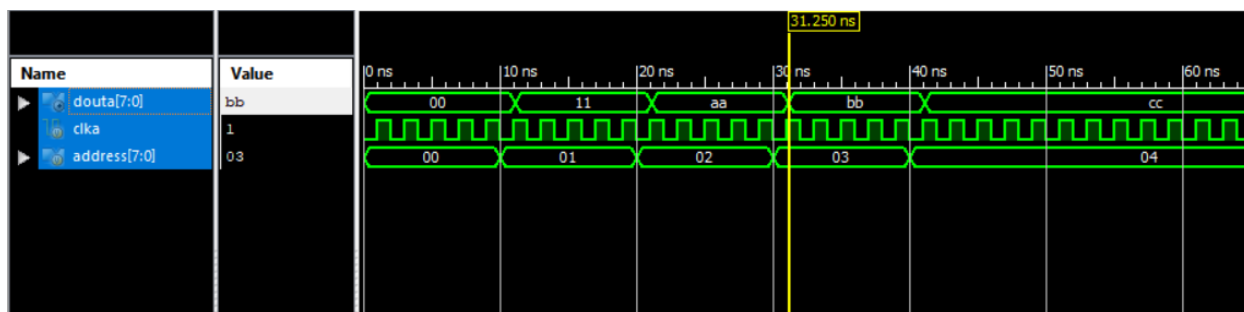
همچنین فایل memory نیز همان تصویر (12) میباشد.

تست

سپس تست بنچی برای آن نوشته و صحت عملکرد آن را تست میکنیم.

```
Initial
begin
  clka =0;
  address=8'b0;
  ;#1
  address=8'b1;
  ;#1
  address=8'b10;
  ;#1
  address=8'b11;
  ;#1
  address=8'b100;
  ;#1
end
endmodule
```

تصویر(16) نتیجه شبیه سازی را نشان میدهد که مطابق با انتظارات ما بوده است.



خلاصه طراحی

Device Utilization Summary				
Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs	1	1,920	1%	
Number of occupied Slices	1	960	1%	
Number of Slices containing only related logic	1	1	100%	
Number of Slices containing unrelated logic	0	1	0%	
Total Number of 4 input LUTs	1	1,920	1%	
Number of bonded IOBs	27	83	32%	
IOB Flip Flops	8			
Number of RAMB16s	1	4	25%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	1.63			

تصویر 2-16

جدول تاخیر زمانی

Source Pad	Destination Pad	Delay
Address<0>	out<0>	6.305
Address<0>	out<2>	5.853
Address<0>	out<3>	5.570
Address<0>	out<4>	6.081
Address<0>	out<5>	5.552
Address<1>	out<0>	6.294
Address<1>	out<2>	5.809
Address<1>	out<3>	5.527
Address<1>	out<4>	6.071
Address<1>	out<5>	5.603
Address<1>	out<6>	6.041
Address<1>	out<7>	6.041
Address<2>	out<0>	6.860
Address<2>	out<2>	6.800
Address<2>	out<3>	6.159
Address<2>	out<4>	7.057
Address<2>	out<5>	6.461
Address<2>	out<6>	5.769
Address<2>	out<7>	5.735

تصویر 3-16

نتیجه گیری

با نگاه به جدول های خلاصه طراحی، در هر دو شیوه Number of bonded IOBs یکسان میباشد. همچنین میتوان فهمید که نسخه های بلوکی (ازپیش ساخته شده) بهینه تر میباشند.

همچنین با توجه به جدول های تاخیر زمانی نیز میتوانیم پی ببریم که مقدار max setup to clk در پیاده سازی بلوکی، بسیار سریعتر میباشد ولی در مقدار clk to pad پیاده سازی انجام شده توسط ما سریعتر میباشد.

در نتیجه میتوانیم به این پی ببریم که متناسب با کد ما، نسخه بلوکی سریعتر میباشد چرا که سریعترین و بهینه ترین نوع حافظه را پیدا و پیاده سازی میکند.