به نام خدا

تمرین عملی پنجم طراحی سیستم های دیجیتال

دکتر علیرضا اجلالی

طراحی SRAM آسنکرون و CONTORLLER سنکرون

طه ایزدی محصل

402110543

ساختار کلی

ساختار کلی و functionality این ماژول به طور کامل در داک آمده. در ادامه به تحلیل کد زده شده برای این دو ماژول و نحوه ی پیاده سازی آن می پردازیم...

ماژول SRAM آسنکرون

ابتدا در کد زیر ، ماژول sram آسنکرون را بررسی می کنیم:

```
4  // asyncron sram module
5  module sram(
6      input [8 :0] addr,
7      inout [15:0] data,
8      input        CE, OE, WE, UB, LB
9  );
10     // internal memory array
11     reg [15:0] memory [0:511];
12
13     // READ
14     assign #(15, 15, 4) data = (CE == 0 && WE == 1 && OE == 0) ?
15                                {
16                                 (UB == 0 ? memory[addr][15:8] : 8'hzz),
17                                 (LB == 0 ? memory[addr][7:0]  : 8'hzz)
18                                } :
19                                16'hzzzz;
20
21     // WRITE
22     always @(*)
23     begin
24         if (CE == 0 && WE == 0)
25         begin
26             if (LB == 0)
27                 memory[addr][7:0] <= #15 data[7:0];
28             if (UB == 0)
29                 memory[addr][15:8] <= #15 data[15:8];
30         end
31     end
32 endmodule
```

همانطور که در کد مشخص است ، ابتدا یک آرایه ی 512 تایی از آرایه های 16 بیتی تعریف کردیم که همان حافظه ی ذخیره سازی sram ما می باشد.

سپس دو بلاک READ و WRITE را تعریف کردیم:

- READ : در اینجا ابتدا به سیگنال های کنترلی نگاه می کند که شرط خواندن را ارضا کرده باشند. اگر سیگنال ها برای خواندن تنظیم شده بودند ، نصفه ی اول و دوم را به هم concatenate می کند و روی data برای خواندن میریزد. توجه شود که هر نصفه با توجه به سیگنال LB یا UB نوشته می شود و یا z روی آن قرار می گیرد.

- WRITE : این بلاک از کد به صورت رفتاری زده شده و با تغییر هر کدام از ورودی ها ، چک می کند که سگینال ها نوشتن را نتیجه می دهند یا نه. اگر سیگنال ها برای نوشتن تنظیم شده باشند ، به سیگنال های کنترلی هر نصفه نگاه می کنند و اگر low بودند ، مقدار موجود روی data که کاربر آن را قرار داده را در حافظه می نویسد.

● تست بنچ SRAM

حال به بررسی تست بنچ این ماژول می پردازیم که ساختار آن کامل در داک توضیح داده شده و طبق آن نوشته شده:

ابتدا سیم ها و متغیر های واسط با sram را تعریف می کنیم که شامل سیگنال ها و data می شود که مقدار یا قطع بودن آن نیز توسط خط 55 با توجه به WE هندل می شود. همچنین از ماژول sram یک instance برای تست گرفته ایم:

```
35  module sram_tb;
36
37      // sram signals
38      reg  [8:0]  addr;
39      reg         CE;
40      reg         OE;
41      reg         WE;
42      reg         UB;
43      reg         LB;
44
45      // 2 way data pass
46      wire [15:0] data;
47
48      // loop counter
49      integer count = 0;
50
51      // variable for writing on data
52      reg  [15:0] data_driver_reg;
53
54      // writing on data based on WE signal
55      assign data = (WE == 0) ? data_driver_reg : 16'hzzzz;
56
57      // instance
58      sram my_sram (
59          .data(data),
60          .addr(addr),
61          .CE(CE),
62          .OE(OE),
63          .WE(WE),
64          .UB(UB),
65          .LB(LB)
66      );
```

در ادامه در initial که روند اصلی تست بنچ دنبال می شود ، به ترتیبی که در داک گفته شده 30 عدد را به وسیله ی لوپ در sram نوشتم:

```
// 1. writing address on lower byte for first 10
$display("Step 1: Writing address to lower byte for addresses 0-9...");
while(count < 10) begin
    addr = count;
    data_driver_reg = {8'h00, count[7:0]};
    $display("data is: %h", data_driver_reg);
    LB = 0;
    UB = 1;
    WE = 0;
    #20;
    WE = 1;
    #5;
    count = count + 1;
end
```

اعداد 0 تا 9 در نصفه اول

```
// 2. writing log2 of address on upper byte for second 10
$display("Step 2: Writing $clog2(address) to upper byte for addresses 10-19...");
while(count < 20) begin
    addr = count;
    data_driver_reg = {$clog2(count), 8'h00};
    LB = 1;
    UB = 0;
    WE = 0;
    #20;
    WE = 1;
    #5;
    count = count + 1;
end
```

لگاریتم 10 تا 19 آدرس در نصفه دوم

```
// 3. writing randoms on third 10
$display("Step 3: Writing random data to addresses 20-29...");
while(count < 30) begin
    addr = count;
    data_driver_reg = $random;
    LB = 0;
    UB = 0;
    WE = 0;
    #20;
    WE = 1;
    #5;
    count = count + 1;
end
```

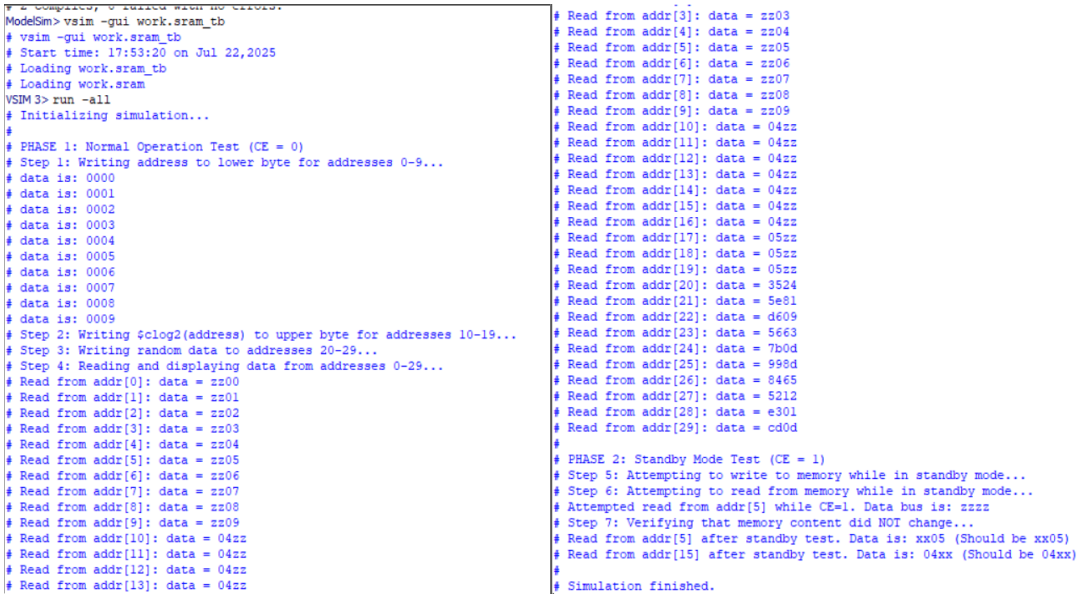
ده عدد رندوم

و در نهایت در یک لوپ 30 تایی تمامی این اعداد نوشته شده ، خوانده و verification **می شوند:**

```
// 4. reading first 30
$display("Step 4: Reading and displaying data from addresses 0-29...");
OE = 0;

count = 0;
while(count < 30) begin
    addr = count;

    // handling upper & lower
    if (count < 10) begin
        LB = 0;
        UB = 1;
    end
    if ((count > 9) && (count < 20)) begin
        LB = 1;
        UB = 0;
    end
    if (count > 19) begin
        LB = 0;
        UB = 0;
    end

    #20;
    $display("Read from addr[%0d]: data = %h", count, data);
    count = count + 1;
end
OE = 1;
```

و در نهایت نیز خواندن و نوشتن در حالت standby **انجام می شود که در ادامه ی تست بنچ آمده است.**

در ادامه نتیجه ی شبیه سازی این تست بنچ را مشاهده می کنیم:



```
ModelSim> vsim -gui work.sram_tb
# vsim -gui work.sram_tb
# Start time: 17:53:20 on Jul 22,2025
# Loading work.sram_tb
# Loading work.sram
VSIM 3> run -all
# Initializing simulation...
#
# PHASE 1: Normal Operation Test (CE = 0)
# Step 1: Writing address to lower byte for addresses 0-9...
# data is: 0000
# data is: 0001
# data is: 0002
# data is: 0003
# data is: 0004
# data is: 0005
# data is: 0006
# data is: 0007
# data is: 0008
# data is: 0009
# Step 2: Writing $clog2(address) to upper byte for addresses 10-19...
# Step 3: Writing random data to addresses 20-29...
# Step 4: Reading and displaying data from addresses 0-29...
# Read from addr[0]: data = zz00
# Read from addr[1]: data = zz01
# Read from addr[2]: data = zz02
# Read from addr[3]: data = zz03
# Read from addr[4]: data = zz04
# Read from addr[5]: data = zz05
# Read from addr[6]: data = zz06
# Read from addr[7]: data = zz07
# Read from addr[8]: data = zz08
# Read from addr[9]: data = zz09
# Read from addr[10]: data = 04zz
# Read from addr[11]: data = 04zz
# Read from addr[12]: data = 04zz
# Read from addr[13]: data = 04zz
# Read from addr[3]: data = zz03
# Read from addr[4]: data = zz04
# Read from addr[5]: data = zz05
# Read from addr[6]: data = zz06
# Read from addr[7]: data = zz07
# Read from addr[8]: data = zz08
# Read from addr[9]: data = zz09
# Read from addr[10]: data = 04zz
# Read from addr[11]: data = 04zz
# Read from addr[12]: data = 04zz
# Read from addr[13]: data = 04zz
# Read from addr[14]: data = 04zz
# Read from addr[15]: data = 04zz
# Read from addr[16]: data = 04zz
# Read from addr[17]: data = 05zz
# Read from addr[18]: data = 05zz
# Read from addr[19]: data = 05zz
# Read from addr[20]: data = 3524
# Read from addr[21]: data = 5e81
# Read from addr[22]: data = d609
# Read from addr[23]: data = 5663
# Read from addr[24]: data = 7b0d
# Read from addr[25]: data = 998d
# Read from addr[26]: data = 8465
# Read from addr[27]: data = 5212
# Read from addr[28]: data = e301
# Read from addr[29]: data = cd0d
#
# PHASE 2: Standby Mode Test (CE = 1)
# Step 5: Attempting to write to memory while in standby mode...
# Step 6: Attempting to read from memory while in standby mode...
# Attempted read from addr[5] while CE=1. Data bus is: zzzz
# Step 7: Verifying that memory content did NOT change...
# Read from addr[5] after standby test. Data is: xx05 (Should be xx05)
# Read from addr[15] after standby test. Data is: 04xx (Should be 04xx)
#
# Simulation finished.
```

که همانطور که مشاهده می شود نتیجه ی تست موفقیت آمیز است و تمامی مقادیر نوشته شده ، بعد از خوانده شدن تغییری نکرده اند و در همان آدرس تعیین شده قرار دارند.

همچنین عملیات نوشتن در حالت standby تغییری در مقدار قبلی نمی دهد و داده ها را حفظ می کند.

● ماژول SRAM CONTROLLER سنکرون

این ماژول کنترل کننده ی سنکرون sram ما می باشد که با کلاک کار می کند. در ابتدا محاسبات مربوط به حداقل کلاک ممکن برای مسیر بحرانی را انجام دادیم که همان خواندن یا نوشتن روی sram می باشد ، انجام دادیم و تعداد چرخه های مورد نیاز را بدست آوردیم:

```
// clock period
localparam CLK_PERIOD = 1000 / freq;

// sram critical path delay
localparam SRAM_DELAY = 15;

// ceiling division (A + B - 1) / B
localparam CYCLES = (SRAM_DELAY + CLK_PERIOD - 1) / CLK_PERIOD;
```

در قدم بعدی به تعریف localparam ها یا نشانگر های حالت فعلی سیستم می پردازیم که پایه و اساس FSM می باشد:

```
// FSM
localparam S_IDLE                = 4'b0000;
localparam S_WRITE_LOWER         = 4'b0001;
localparam S_WRITE_UPPER         = 4'b0010;
localparam S_READ_LOWER_SETUP    = 4'b0100;
localparam S_READ_LOWER_CAPTURE  = 4'b0101;
localparam S_READ_UPPER_SETUP    = 4'b0110;
localparam S_READ_UPPER_CAPTURE  = 4'b0111;
```

که هرحالت:

- **S_IDLE** : حالتیست که هیچ کاری انجام نمی شود و سیگنالی برای خواندن یا نوشتن فعال نیست
- **S_WRITE_LOWER** : حالتیست که می خواهیم 16 بیت اول مقدارمان را در یک خانه sram بنویسیم
- **S_WRITE_UPPER** : حالتیست که میخواهیم 16 بیت دوم مقدارمان را در خانه ی بعدی sram بنویسیم
- **S_READ_LOWER_SETUP** : حالتیست که دستور خواندن 16 بیت اول را به controller داده ایم که از sram بخواند (setup) در این حالت بر خلاف حالت نوشتن ، باید منتظر جواب sram بمانیم و آن را نمایش بدهیم (capture)
- **S_READ_LOWER_CAPTURE** : حالتیست که controller دارد 16 بیت اول را از sram می خواند و در نهایت نمایشش می دهد
- **S_READ_UPPER_SETUP + S_READ_UPPER_CAPTURE** : همانند دو حالت قبلی فقط برای 16 بیت دوم

حال با low کردن سیگنال های sram مطابق داک ، به پیاده سازی بلاک های بعدی FSM پرداختیم. ابتدا بلاک transition را تعریف کردیم:

```
// state transition
always @(posedge clk or posedge rst)
begin
    if (rst)
        current_state <= S_IDLE;
    else
        current_state <= next_state;
end
```

این بلاک مسئولیت هندل کردن transition حالت ها است و همواره next state را در current state میریزد و همچنین reset را هندل می کند. در ادامه اینکه next state با توجه به سیگنال ها چگونه تعیین می شود را بررسی می کنیم:

```
81      // combinational next state
82      always @(*)
83      begin
84          // defult of next state
85          next_state = current_state;
86
87          // choosing next state based on controller signals
88          case (current_state)
89              S_IDLE: begin
90                  if (memWrite)
91                      next_state = S_WRITE_LOWER;
92                  else if(memRead)
93                      next_state = S_READ_LOWER_SETUP;
94              end
95              S_WRITE_LOWER: begin
96                  if (wait_counter == 0)
97                      next_state = S_WRITE_UPPER;
98              end
99              S_WRITE_UPPER: begin
100                 if (wait_counter == 0)
101                     next_state = S_IDLE;
102             end
103             S_READ_LOWER_SETUP: begin
104                 if (wait_counter == 0)
105                     next_state = S_READ_LOWER_CAPTURE;
106             end
107             S_READ_LOWER_CAPTURE: begin
108                 next_state = S_READ_UPPER_SETUP;
109             end
110             S_READ_UPPER_SETUP: begin
111                 if (wait_counter == 0)
112                     next_state = S_READ_UPPER_CAPTURE;
113             end
114             S_READ_UPPER_CAPTURE: begin
115                 next_state = S_IDLE;
116             end
117         endcase
118     end
```

همانطور که مشخص است به صورت پیش فرض حالت بعدی را همان حالت فعلی قرار می دهد و در یک switch case با توجه حالت فعلی ، سیگنال ها و میزان باقی مانده از cycle های باقی مانده برای انجام عملیات ، حالت بعدی را تعیین می کند. حال این که تعداد cycle های عملیات ها چگونه تنظیم می شوند و چگونه کم می شوند در بلاک زیر قابل مشاهده است:

```
64      // wait counter
65      always @(posedge clk or posedge rst)
66      begin
67          // if it's reset
68          if (rst)
69              wait_counter <= 0;
70          // setting wait counter
71          else if (next_state != current_state)
72          begin
73              if (next_state == S_WRITE_LOWER || next_state == S_WRITE_UPPER ||
74                  next_state == S_READ_LOWER_SETUP || next_state == S_READ_UPPER_SETUP)
75                  wait_counter <= CYCLES - 1;
76          // decreasing wait counter during operation
77          end else if (wait_counter > 0)
78              wait_counter <= wait_counter - 1;
79      end
```

که بدین صورت عمل می کند که اگر reset باشد ، آن را صفر می کند. اگر اول شروع خواندن یا نوشتن در sram باشیم ، آن را به اندازه ای که قبلا محاسبه کردیم ست می کند و در غیر این صورت یعنی وسط انجام یک عملیات هستیم و counter را یکی کم می کند.

در نهایت نیز یک بلاک برای تنظیم خروجی ها در هر لحظه بر اساس حالتی که درونش قرار داریم نیاز داریم که شکل زیر پیاده سازی شده:

```
120     // output
121     always @(posedge clk or posedge rst)
122     begin
123         // if it's reset
124         if (rst) begin
125             ready <= 1'b1;
126             dataOut <= 32'b0;
127             addr <= 9'b0;
128             data_reg <= 16'b0;
129         end else begin
130             // default values
131             ready <= (next_state == S_IDLE && current_state == S_IDLE);
132 
133             if (current_state == S_READ_LOWER_CAPTURE)
134                 dataOut[15:0] <= data;
135             else if (current_state == S_READ_UPPER_CAPTURE)
136                 dataOut[31:16] <= data;
137 
138 
139             // set outputs based on next state
140             case (next_state)
141                 S_WRITE_LOWER: begin
142                     WE <= 1'b0;
143                     addr <= addrTarget;
144                     data_reg <= dataIn[15:0];
145                 end
146                 S_WRITE_UPPER: begin
147                     WE <= 1'b0;
148                     addr <= addrTarget + 1;
149                     data_reg <= dataIn[31:16];
150                 end
151                 S_READ_LOWER_SETUP: begin
152                     WE <= 1'b1;
153                     addr <= addrTarget;
154                 end
155                 S_READ_UPPER_SETUP: begin
156                     WE <= 1'b1;
157                     addr <= addrTarget + 1;
158                 end
159             endcase
160         end
161     end
162 endmodule
```

که در ابتدا چک می کند که اگر reset باشد خروجی ها را صفر می کند. در غیر این صورت با استفاده از if-else و switch case مقدار خروجی ها را بر اساس حالتی که در آن قرار داریم تعیین می کند و در اینجا پیاده سازی ماژول ما تکمیل می شود.

● تست بنچ SRAM CONTROLLER

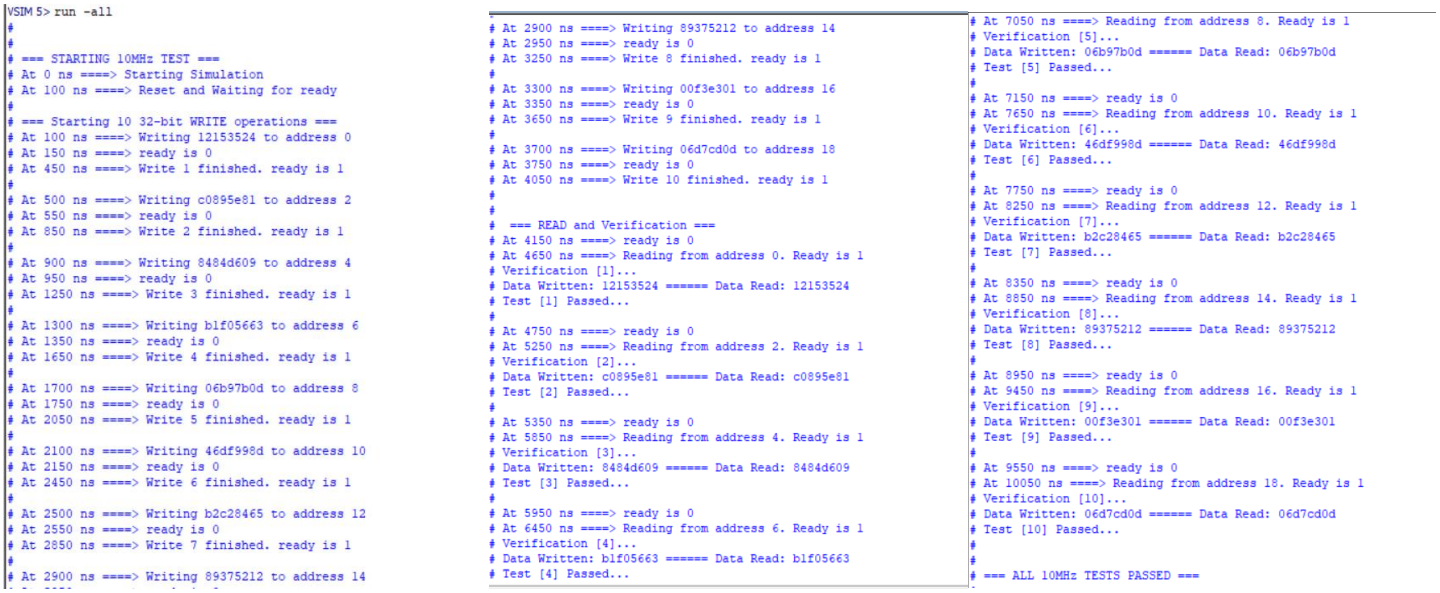
این تست بنچ مطابق داک دو بخش دارد یکی مربوط به کلاک 10MHz و دومی مربوط به کلاک 200MHz می باشد. برای هر کدام ابتدا ماژول های لازمه را instance گرفتیم و متغیرها را تعریف کردیم. سپس ده مقدار رندوم در sram نوشته شده که فرایند هر نوشتن با هندل کردن سیگنال ها و سیگنال ready جلو رفته:

```
// initializing
clk_10  = 0;
clk_200 = 0;
global_rst = 1;
test_select = 0;

// ****************** 10MHz Test ******************
$display("\n\n=== STARTING 10MHz TEST ===");
rst_10 = 1;
flag_10 = 0;
memRead_10 = 0;
memWrite_10 = 0;
$display("At %0t ns ====> Starting Simulation", $time);

#100;
global_rst = 0;
rst_10 = 0;
$display("At %0t ns ====> Reset and Waiting for ready\n", $time);
wait(ready_10);

// Write Operations
$display("=== Starting 10 32-bit WRITE operations ===");
for (i = 0; i < 10; i = i + 1)
begin
    @(negedge clk_10);
    memWrite_10 = 1;
    dataIn_10 = $random;
    written_data_10[i] = dataIn_10;
    addrTarget_10 = 2 * i;
    $display("At %0t ns ====> Writing %h to address %0d", $time, dataIn_10, addrTarget_10);

    @(posedge clk_10);
    memWrite_10 = 0;

    wait(!ready_10);
    $display("At %0t ns ====> ready is %d", $time, ready_10);
    wait(ready_10);
    $display("At %0t ns ====> Write %0d finished. ready is %d\n", $time, i+1, ready_10);
end
```

سپس بعد از آن مقادیر را می خوانیم و verification انجام می دهیم تا از صحت درستی ماژول هایمان اطمینان حاصل کنیم:

```
// Read Operations & Verification
$display("\n === READ and Verification ===");
for (j = 0; j < 10; j = j + 1)
begin
    @(negedge clk_10);
    memRead_10 = 1;
    addrTarget_10 = 2 * j;

    @(posedge clk_10);
    memRead_10 = 0;

    wait(!ready_10);
    $display("At %0t ns ====> ready is %d", $time, ready_10);

    wait(ready_10);
    $display("At %0t ns ====> Reading from address %0d. Ready is %d",$time, addrTarget_10, ready_10);

    // verification
    $display("Verification [%0d]...", j+1);
    $display("Data Written: %h ====== Data Read: %h", written_data_10[j], dataOut_10);

    if (dataOut_10 === written_data_10[j])
        $display("Test [%0d] Passed...\n", j+1);
    else
    begin
        $display("Test [%0d] Failed...\n", j+1);
        flag_10 = 1'b1;
    end
end

if (!flag_10)
    $display("\n=== ALL 10MHz TESTS PASSED ===");
else
    $display("\n=== 10MHz TESTS FAILED ===");
```

و همین فرایند برای حالت بعدی کلاک نیز تکرار شده. در شکل زیر نتیجه شبیه سازی تست بنچ را می بینید:



```
VSIM 5> run -all
#
#
# === STARTING 10MHz TEST ===
# At 0 ns ====> Starting Simulation
# At 100 ns ====> Reset and Waiting for ready
#
# === Starting 10 32-bit WRITE operations ===
# At 100 ns ====> Writing 12153524 to address 0
# At 150 ns ====> ready is 0
# At 450 ns ====> Write 1 finished. ready is 1
#
# At 500 ns ====> Writing c0895e81 to address 2
# At 550 ns ====> ready is 0
# At 850 ns ====> Write 2 finished. ready is 1
#
# At 900 ns ====> Writing 8484d609 to address 4
# At 950 ns ====> ready is 0
# At 1250 ns ====> Write 3 finished. ready is 1
#
# At 1300 ns ====> Writing b1f05663 to address 6
# At 1350 ns ====> ready is 0
# At 1650 ns ====> Write 4 finished. ready is 1
#
# At 1700 ns ====> Writing 06b97b0d to address 8
# At 1750 ns ====> ready is 0
# At 2050 ns ====> Write 5 finished. ready is 1
#
# At 2100 ns ====> Writing 46df998d to address 10
# At 2150 ns ====> ready is 0
# At 2450 ns ====> Write 6 finished. ready is 1
#
# At 2500 ns ====> Writing b2c28465 to address 12
# At 2550 ns ====> ready is 0
# At 2850 ns ====> Write 7 finished. ready is 1
#
# At 2900 ns ====> Writing 89375212 to address 14
```

```
# At 2900 ns ====> Writing 89375212 to address 14
# At 2950 ns ====> ready is 0
# At 3250 ns ====> Write 8 finished. ready is 1
#
# At 3300 ns ====> Writing 00f3e301 to address 16
# At 3350 ns ====> ready is 0
# At 3650 ns ====> Write 9 finished. ready is 1
#
# At 3700 ns ====> Writing 06d7cd0d to address 18
# At 3750 ns ====> ready is 0
# At 4050 ns ====> Write 10 finished. ready is 1
#
#
#  === READ and Verification ===
# At 4150 ns ====> ready is 0
# At 4650 ns ====> Reading from address 0. Ready is 1
# Verification [1]...
# Data Written: 12153524 ====== Data Read: 12153524
# Test [1] Passed...
#
# At 4750 ns ====> ready is 0
# At 5250 ns ====> Reading from address 2. Ready is 1
# Verification [2]...
# Data Written: c0895e81 ====== Data Read: c0895e81
# Test [2] Passed...
#
# At 5350 ns ====> ready is 0
# At 5850 ns ====> Reading from address 4. Ready is 1
# Verification [3]...
# Data Written: 8484d609 ====== Data Read: 8484d609
# Test [3] Passed...
#
# At 5950 ns ====> ready is 0
# At 6450 ns ====> Reading from address 6. Ready is 1
# Verification [4]...
# Data Written: b1f05663 ====== Data Read: b1f05663
# Test [4] Passed...
```

```
# At 7050 ns ====> Reading from address 8. Ready is 1
# Verification [5]...
# Data Written: 06b97b0d ====== Data Read: 06b97b0d
# Test [5] Passed...
#
# At 7150 ns ====> ready is 0
# At 7650 ns ====> Reading from address 10. Ready is 1
# Verification [6]...
# Data Written: 46df998d ====== Data Read: 46df998d
# Test [6] Passed...
#
# At 7750 ns ====> ready is 0
# At 8250 ns ====> Reading from address 12. Ready is 1
# Verification [7]...
# Data Written: b2c28465 ====== Data Read: b2c28465
# Test [7] Passed...
#
# At 8350 ns ====> ready is 0
# At 8850 ns ====> Reading from address 14. Ready is 1
# Verification [8]...
# Data Written: 89375212 ====== Data Read: 89375212
# Test [8] Passed...
#
# At 8950 ns ====> ready is 0
# At 9450 ns ====> Reading from address 16. Ready is 1
# Verification [9]...
# Data Written: 00f3e301 ====== Data Read: 00f3e301
# Test [9] Passed...
#
# At 9550 ns ====> ready is 0
# At 10050 ns ====> Reading from address 18. Ready is 1
# Verification [10]...
# Data Written: 06d7cd0d ====== Data Read: 06d7cd0d
# Test [10] Passed...
#
#
# === ALL 10MHz TESTS PASSED ===
```

که همانطور که مشخص است تمامی تست ها به درستی قبول شده اند و در هر خواندن یا نوشتن سیگنال ready دنبال شده است. نتیجه ی تست کلاک 200MHz ای نیز مانند بالا می باشد و تمامی تست ها قبول می شوند.

برای اطمینان بیشتر فایل vcd هر دو تست بنچ sram و controller آن فرستاده شده تا نتایج به صورت واضح تر قابل مشاهده باشند.

● بخش امتیازی: تاخیر واقعی SRAM

با توجه به دیتاشیتی که در داک موجود است ، برای این sram دو مدل معرفی شده که یک مدل سریع تر (10 نانوثانیه) و دیگری کند تر (12 نانوثانیه) است. برای هر کدام حداقل تاخیر های خواندن و نوشتن و تغییر به z به شرح زیر می باشد...

مدل سریعتر:

- **نوشتن**
  - **چرخه نوشتن: 10 نانوثانیه**
  - **عوض شدن سیگنال WE : 8 نانوثانیه**
  - **آماده سازی آدرس تا قبل از نوشتن: 8 نانوثانیه**
  - **آماده سازی داده تا پایان نوشتن: 6 نانوثانیه**
  - **زمان نگهداری داده (حفظ روی گذرگاه برای ثبت): 0 نانوثانیه (می تواند همزمان با پایان نوشتن تغییر کند)**
- **خواندن**
  - **چرخه خواندن: 10 نانوثانیه**
  - **زمان دسترسی به آدرس: 10 نانوثانیه**
  - **زمان دسترسی از طریق CE : 10 نانوثانیه**

- زمان دسترسی از طریق OE : 4 نانوثانیه

- تغییر حالت گذرگاه داده
  - تغییر حالت به z
    - از طریق CE : 4 نانوثانیه
    - از طریق OE : 4 نانوثانیه
    - از طریق WE : 5 نانوثانیه
  - تغییر حالت از z
    - از طریق CE : 3 نانوثانیه
    - از طریق OE : 0 نانوثانیه
    - از طریق WE : 2 نانوثانیه

مدل کندتر:

- نوشتن
  - چرخه نوشتن: 12 نانوثانیه
  - عوض شدن سیگنال WE : 8 نانوثانیه
  - آماده سازی آدرس تا قبل از نوشتن: 8 نانوثانیه
  - آماده سازی داده تا قبل از نوشتن: 6 نانوثانیه
  - زمان نگهداری داده: 0 نانوثانیه
- خواندن
  - چرخه خواندن: 12 نانوثانیه
  - زمان دسترسی به آدرس: 12 نانوثانیه
  - زمان دسترسی از طریق CE: 12 نانوثانیه
  - زمان دسترسی از طریق OE: 5 نانوثانیه
- تغییر حالت گذرگاه داده
  - تغییر حالت به z
    - از طریق CE: 6 نانوثانیه
    - از طریق OE: 5 نانوثانیه
    - از طریق WE: 6 نانوثانیه
  - تغییر حالت از z
    - از طریق CE: 3 نانوثانیه
    - از طریق OE: 0 نانوثانیه
    - از طریق WE: 2 نانوثانیه