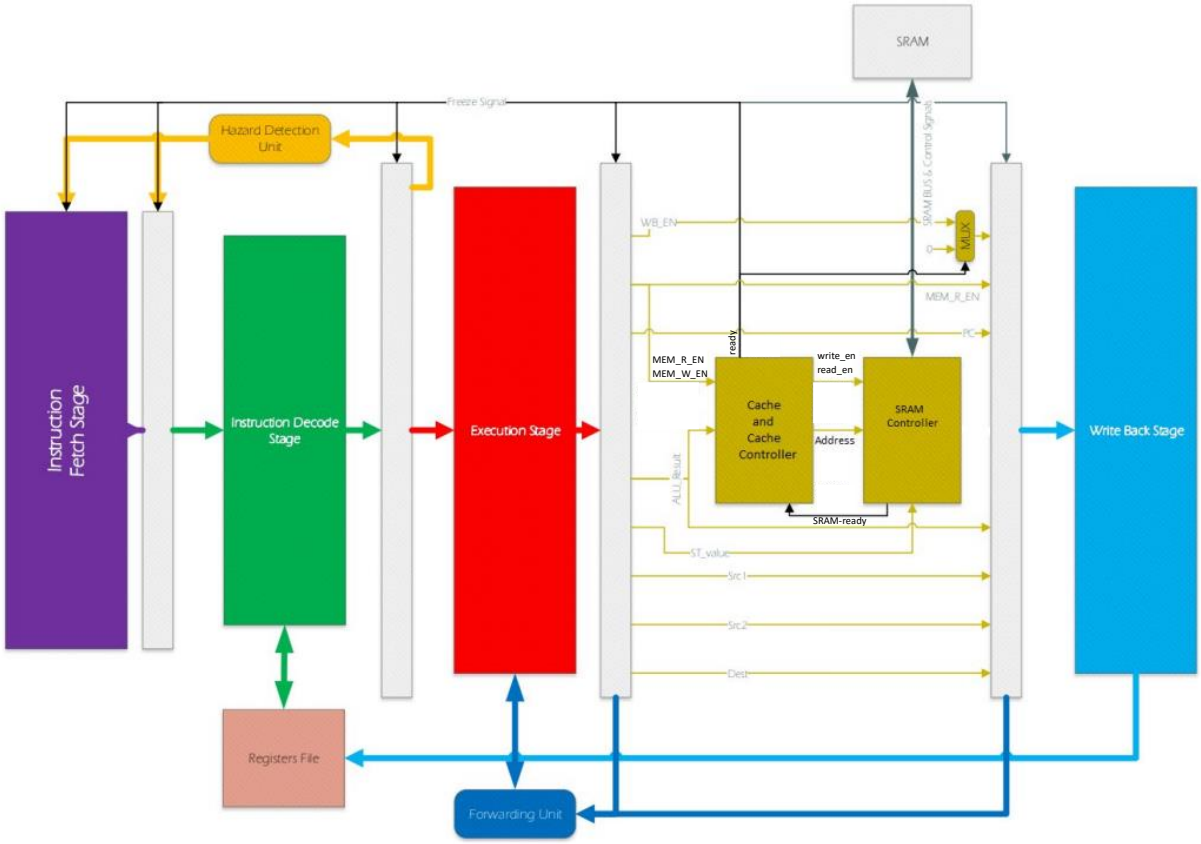
آوا میرمحمدمهدی 810199501 نسا عباسی مقدم 810199457

در جلسه هفتم آزمایشگاه معماری قصد داریم با اضافه کردن حافظه نهان(cache)، کارایی پردازنده را افزایش دهیم. در جلسه قبل دیدیم که با اضافه شدن SRAM، کارایی پردازنده کاهش یافت و راه حل آن، طراحی پردازنده با استفاده از cache پیشنهاد شد. حافظه نهان یک حافظه بسیار سریع است که از نوع پردازنده ساخته می شود. در صورتیکه داده موردنظر را داشته باشد آن را در یک سیکل کلاك به پردازنده میدهد و در غیر اینصورت، داده را از حافظه اصلی(SRAM) می‌خواند که در این حال، تعداد سیکل مورد نیاز، به تعداد کلاک های SRAM بستگی دارد. از این رو اگر به جای دسترسی به حافظه اصلی به حافظه نهان مراجعه شود، آنگاه نیاز به متوقف کردن پردازنده برای انجام علمیات حافظه نیست و کارایی پردازنده افزایش خواهد داشت.



Cache_controller:

حافظه نهان دارای دو way است که هر یک دارای 64 خط است. در هر خط 64 بیت داده است که به صورت دو داده 32 بیتی ذخیره شده است. علاوه بر آن، دارای 3 بیت offset، 6 بیت index، 10 بیت tag، یک بیت validو 64 خط LRU(last recently used) است که مجموعا برای تعیین offset، index و tag به 19 بیت احتیاج داریم. هر زمان که آدرس داده مورد نظر ما با آدرس داده ذخیره شده مطابقت داشت و همچنین بیت valid برابر یک بود، hit اتفاق می‌افتد. درصورتیکه حتی یکی از این دو شرط برقرار نباشد، miss رخ خواهد داد.

Offset: برای انتخاب بایت های داده است. در این پروژه تنها به بیت آخر ([2]offset) نیاز داریم تا از بین داده های 32 بیتی، یکی را انتخاب کنیم.

Index: برای انتخاب بین 64 خط (set) است.

Tag: برای مقدار آدرس داده است.

Valid: برای تعیین معتبر بودن یا نبودن داده است.

Tag										Index						Offset		
18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

عملیات خواندن و نوشتن در cache

- **عملیات خواندن**

در صورتیکه داده مورد نظر در cache موجود باشد و بیت valid آن یک باشد، می‌توان به داده دسترسی پیدا کرد. در این حالت بیت LRU را آپدیت می‌کنیم. بطوریکه اگر داده از way0 خوانده شده باشد، LRU را برابر صفر و اگر از way1 خوانده شده باشد، LRU را برابر یک می‌کنیم.

- **عملیات نوشتن**

در صورتیکه داده مورد نظر در cache نباشد و یا بیت valid آن یک نباشد، باید داده را از SRAM خوانده و بیت valid و LRU را آپدیت کنیم. بطوریکه بیت valid را یک کرده و بیت LRU را در در صورتیکه در way0 نوشته باشیم، برابر یک و درصورتیکه در way1 نوشته باشیم، برابر صفر می‌کنیم.

```
module Cache_controller (clk, rst, address, write_data, MEM_R_EN, MEM_W_EN, read_data, ready,
sram_address, sram_write_data, sram_write, sram_read, sram_read_data, sram_ready);

    input clk, rst, sram_ready, MEM_R_EN, MEM_W_EN;
    input [31:0] address;
    input [31:0] write_data;
    input [63:0] sram_read_data;

    output ready, sram_write, sram_read;
```

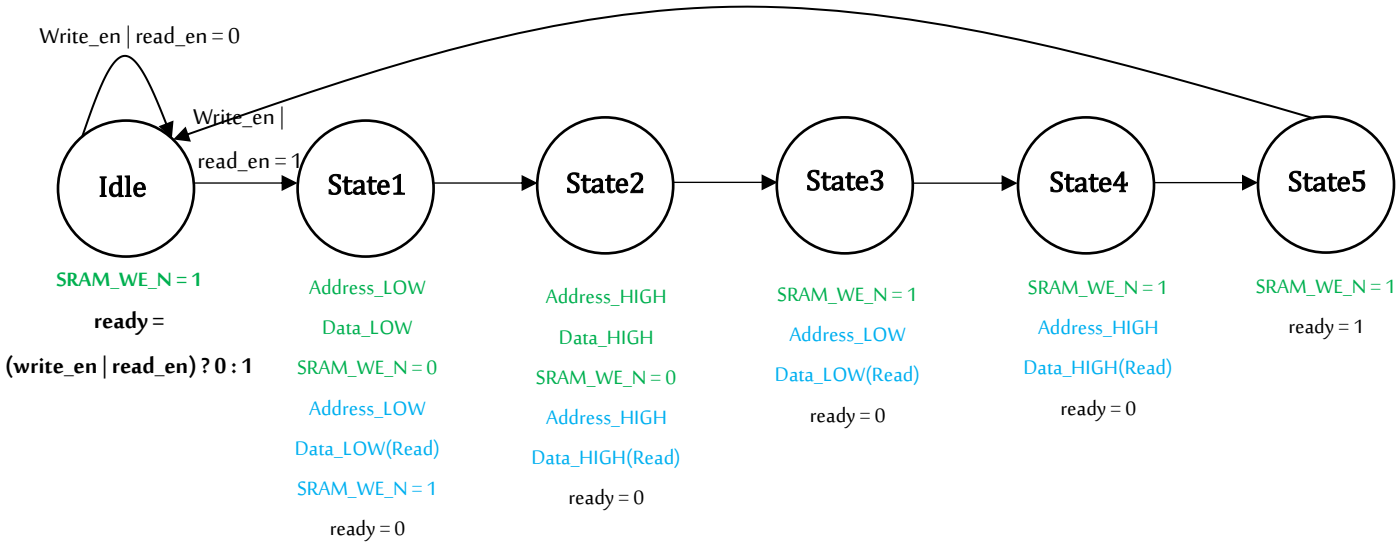
```
    output [31:0] read_data, sram_address, sram_write_data;

    wire hit;
    assign ready = sram_ready;

    wire [2:0] offset;
    wire [5:0] index;
    wire [9:0] tag;
    assign offset = address[2:0];
    assign index = address[8:3];
    assign tag = address[18:9];

    reg [31:0] way0_data0 [63:0];
    reg [31:0] way0_data1 [63:0];
    reg [31:0] way1_data0 [63:0];
    reg [31:0] way1_data1 [63:0];

    reg way0_valid [63:0];
    reg way1_valid [63:0];

    reg [9:0] way0_tag [63:0];
    reg [9:0] way1_tag [63:0];

    reg LRU [63:0];

    wire hit_way0, hit_way1;
    assign hit_way0 = (way0_valid[index]) & (way0_tag[index] == tag);
    assign hit_way1 = (way1_valid[index]) & (way1_tag[index] == tag);
    assign hit = hit_way0 | hit_way1;
    assign sram_read = MEM_R_EN & ~hit;

    wire [31:0] read_data_temp;
    wire [31:0] result_hit0, result_hit1, result_low_high;

    Mux2To1 #(32) mux_hit_way0 (result_hit0, way0_data0[index], way0_data1[index], offset[2]);
    Mux2To1 #(32) mux_hit_way1 (result_hit1, way1_data0[index], way1_data1[index], offset[2]);
    Mux2To1 #(32) mux_result_low_high (result_low_high, sram_read_data[31:0],
sram_read_data[63:32], offset[2]);

    assign read_data_temp = hit ? (hit_way0 ? result_hit0 : hit_way1 ? result_hit1 : 32'bz) :
sram_ready ? result_low_high : 32'bz;

    integer i;
    always @(posedge rst, posedge clk) begin
        if (rst)
            for(i = 0; i < 64; i = i + 1)
                LRU[i] <= 0;

        else if (MEM_R_EN) begin
            if (hit_way0)
                LRU[index] <= 0;

            if (hit_way1)
                LRU[index] <= 1;
        end
```

```
        else if (~hit & sram_ready) begin
            if (way0_valid[index] == 1'b1 & LRU[index] == 1'b1)
                LRU[index] <= 1'b0;

            else if (way1_valid[index] == 1'b1 & LRU[index] == 1'b0)
                LRU[index] <= 1'b1;
        end
    end

    assign read_data = MEM_R_EN ? read_data_temp : 32'bz;

    always @(posedge rst, posedge clk) begin
        if (rst)begin
            for (i = 0; i < 64; i = i + 1) begin
                {way0_valid[i], way1_valid[i]} <= 0;
            end
        end

        else if (MEM_R_EN & ~hit & sram_ready) begin
            if (LRU[index] == 1) begin
                {way0_data1[index], way0_data0[index]} <= sram_read_data;
                way0_valid[index] <= 1;
                way0_tag[index] <= tag;
            end

            else if (LRU[index] == 0) begin
                {way1_data1[index], way1_data0[index]} <= sram_read_data;
                way1_valid[index] <= 1;
                way1_tag[index] <= tag;
            end
        end

        else if (MEM_W_EN) begin
            if (hit_way0) begin
                way0_valid[index] <= 0;
            end

            if (hit_way1) begin
                way1_valid[index] <= 0;
            end
        end
    end

    assign sram_address = address;
    assign sram_write_data = write_data;
    assign sram_write = MEM_W_EN;


endmodule
```

SRAM_controller:

همانطور که در قسمت قبل توضیح داده شد، دسترسی به SRAM زمانی رخ می‌دهد که در حافظه نهان، miss رخ داده باشد. در این حالت داده مورد نظر و همچنین داده مجاور آنرا خواند و در cache ذخیره می‌کنیم. بنابراین بجای 32 بیت داده، 64 داده باید خوانده شود. جلسه قبل دیدیم که برای خواندن 32 بیت، به دو کلاک سایکل نیاز داریم که در هر کلاک 16 بیت خوانده می‌شود. در نتیجه برای 64 بیت داده به 4 کلاک سایکل نیاز داریم. از آنجایی که SRAM جلسه قبل با 6 کلاک سایکل طراحی شده بود، بنابراین در این قسمت احتیاجی به اضافه کردن تعداد کلاک ها نیست و می‌توان با همان تعداد استیت عملیات خواندن را انجام داد.

Diagram کنترلر SRAM به شکل زیر تغییر خواهد کرد:




```
module SRAM_Controller(clk, rst, write_en, read_en, address, write_data, read_data, ready,
SRAM_DQ, SRAM_ADDR, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N);
    input clk, rst;
    input write_en, read_en;
    input [31:0] address;
    input [31:0] write_data;
    output reg [63:0] read_data;
    output reg ready;

    inout [15:0] SRAM_DQ;
    output reg[17:0] SRAM_ADDR;
    output  SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N;
    output reg SRAM_WE_N;
    assign SRAM_UB_N = 1'b0;
    assign SRAM_LB_N = 1'b0;
    assign SRAM_CE_N = 1'b0;
    assign SRAM_OE_N = 1'b0;

    wire [31:0] mem_address;
```

```
    assign mem_address = address - 32'd1024;
    wire [15:0] DataBusRead;
    reg [15:0] data_to_write;

    assign SRAM_DQ = (write_en) ? data_to_write : 16'bz;
    assign DataBusRead = SRAM_DQ;
    reg [2:0] ps, ns;

    parameter [3:0] Idle = 0, State1 = 1, State2 = 2, State3 = 3, State4 = 4, State5 = 5;

    always @(*) begin
        {SRAM_WE_N, ready} = 2'b10;
        case (ps)
            Idle: begin
                ready = (write_en|read_en) ? 1'b0 : 1'b1;
            end

            State1: begin
                if (write_en) begin
                    SRAM_ADDR = {mem_address[18:2], 1'b0};
                    SRAM_WE_N = 1'b0;
                    data_to_write = write_data[15:0];
                end
                else begin
                    SRAM_ADDR = {mem_address[18:3], 2'b00};
                    read_data[15:0] <= DataBusRead;
                end
            end

            State2: begin
                if (write_en) begin
                    SRAM_ADDR = {mem_address[18:2], 1'b1};
                    SRAM_WE_N = 1'b0;
                    data_to_write = write_data[31:16];
                end
                else begin
                    SRAM_ADDR = {mem_address[18:3], 2'b01};
                    read_data[31:16] <= DataBusRead;
                end
            end

            State3: begin
                if (write_en) begin
                    SRAM_WE_N = 1'b1;
                end
                else begin
                    SRAM_ADDR = {mem_address[18:3], 2'b10};
                    read_data[47:32] <= DataBusRead;
                end
            end

            State4: begin
                if (read_en) begin
                    SRAM_ADDR = {mem_address[18:3], 2'b11};
                    read_data[63:48] <= DataBusRead;
```

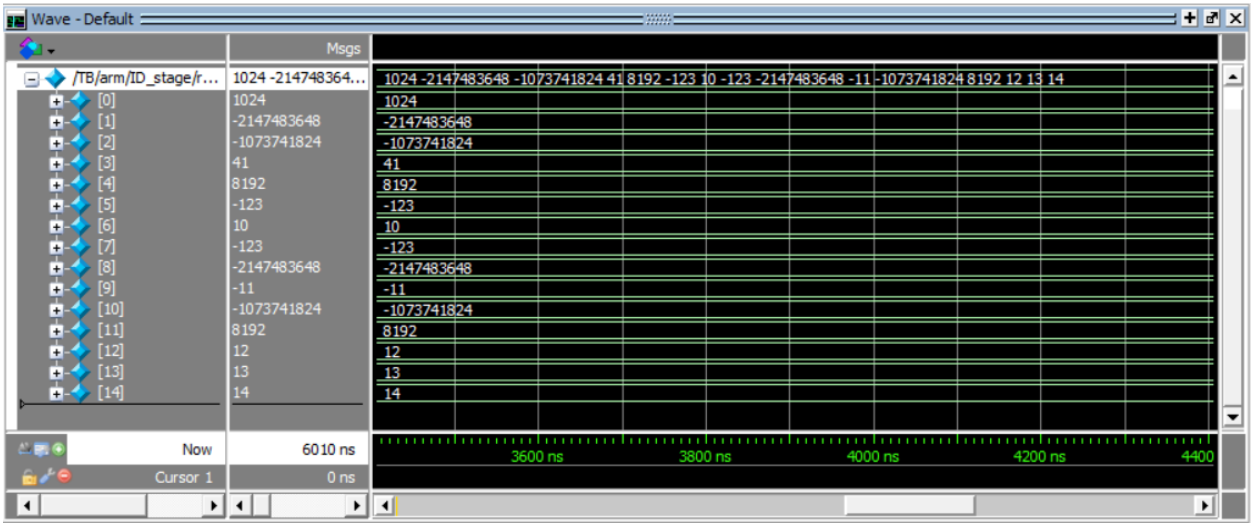
```
                end
            end

            State5: begin
                ready = 1'b1;
            end
        endcase
    end

    always @(*)begin
        case (ps)
            Idle: begin
                ns = (write_en|read_en) ? State1 : Idle;
            end

            State1: begin
                ns = State2;
            end

            State2: begin
                ns = State3;
            end

            State3: begin
                ns = State4;
            end

            State4: begin
                ns = State5;
            end

            State5: begin
                ns = Idle;
            end
        endcase
    end

    always @(posedge clk) begin
        if (rst) begin
            ps <= Idle;
        end
        else begin
            ps <= ns;
        end
    end

endmodule
```

MEM Stage

واحد حافظه نهان، به بخش MEM_Stage اضافه می‌شود. ورودی و خروجی ها ثابت میمانند و wire ها و اتصالات SRAM و cache انجام خواهد شد.

```
module Mem_Stage (clk, rst, dst, ALU_res, val_Rm, mem_read, mem_write, WB_en, dst_out,
ALU_res_out, mem_out, mem_read_out, WB_en_out,
                  ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_ADDR,
SRAM_DQ);
    input clk, rst;
    input [`REG_FILE_ADDRESS_LEN-1:0] dst;
    input [`WORD_WIDTH-1:0] ALU_res;
    input [`WORD_WIDTH-1:0] val_Rm;
    input mem_read, mem_write, WB_en;

    output [`REG_FILE_ADDRESS_LEN-1:0] dst_out;
    output [`WORD_WIDTH-1:0] ALU_res_out;
    output [`WORD_WIDTH-1:0] mem_out;
    output mem_read_out, WB_en_out;
    output ready, SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N;
    output [17:0] SRAM_ADDR;
    inout [15:0] SRAM_DQ;

    assign dst_out = dst;
    assign mem_read_out = mem_read;
    assign ALU_res_out = ALU_res;
    wire freeze;
    assign freeze = ~ready;

    wire sram_writeEn, sram_readEn, sram_ready;
    wire [`DATA_MEM_SIZE-1:0] sram_read_data;
    SRAM_Controller sram_controller(.clk(clk), .rst(rst), .write_en(sram_writeEn),
.read_en(sram_readEn), .address(ALU_res), .write_data(val_Rm),
                                    .read_data(sram_read_data), .ready(sram_ready),
.SRAM_DQ(SRAM_DQ), .SRAM_ADDR(SRAM_ADDR), .SRAM_UB_N(SRAM_UB_N), .SRAM_LB_N(SRAM_LB_N),
                                    .SRAM_WE_N(SRAM_WE_N), .SRAM_CE_N(SRAM_CE_N),
.SRAM_OE_N(SRAM_OE_N));

    Cache_controller cache_controller(.clk(clk), .rst(rst), .address(ALU_res),
.write_data(val_Rm), .MEM_R_EN(mem_read), .MEM_W_EN(mem_write), .read_data(mem_out),
                                      .ready(ready), .sram_write(sram_writeEn),
.sram_read(sram_readEn), .sram_read_data(sram_read_data), .sram_ready(sram_ready));

    Mux2To1 #(1) MUX_2_to_1_WB_EN (.out(WB_en_out), .in1(WB_en), .in2(1'b0), .sel(freeze));

endmodule
```

نمایش خروجی waveform در modelsim :

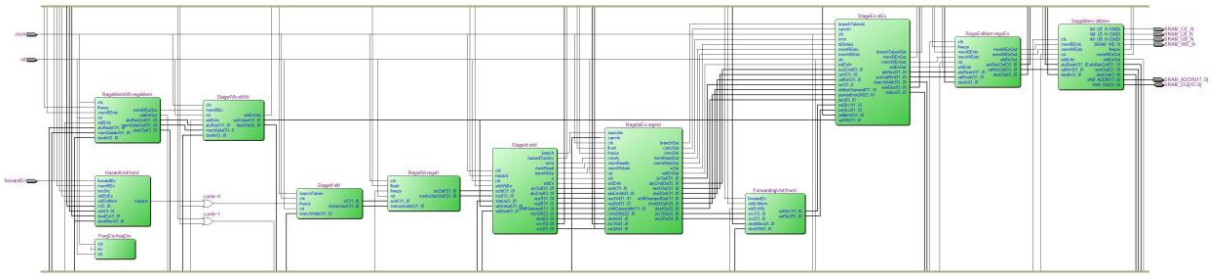


خروجی برنامه Quartus:

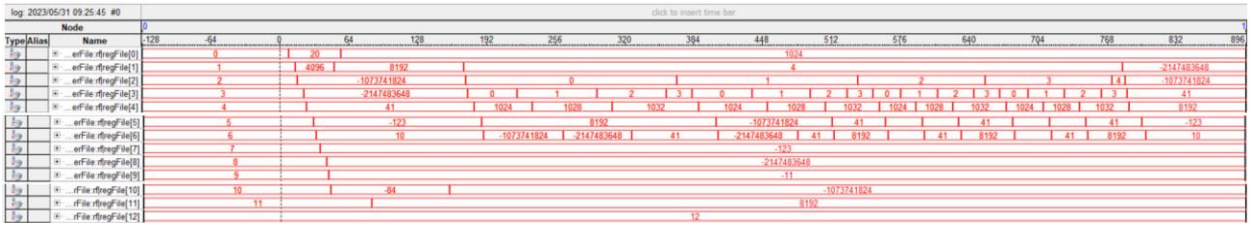
خروجی Flow Summary:

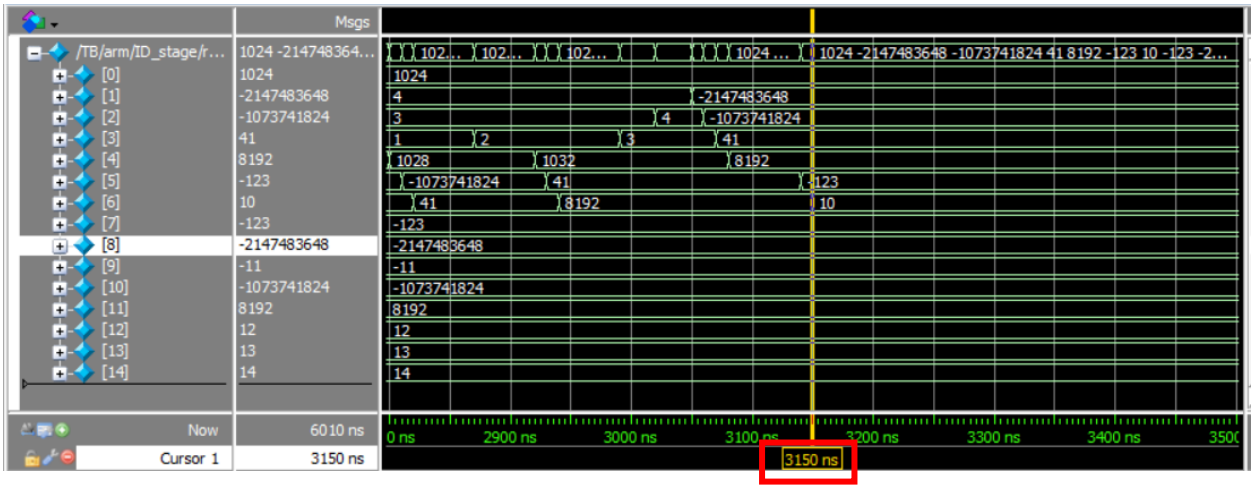
Flow Summary	
Flow Status	Successful - Wed May 31 09:30:51 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	arm
Top-level Entity Name	arm
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	6,933 / 33,216 (21 %)
Total combinational functions	4,772 / 33,216 (14 %)
Dedicated logic registers	4,394 / 33,216 (13 %)
Total registers	4394
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	176,384 / 483,840 (36 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

نمایش RTL:



خروجی Signal Tab:





میزان افزایش کارایی:

$$performance1 = \frac{1}{Execution\ time1} = \frac{1}{430}$$

$$performance2 = \frac{1}{Execution\ time2} = \frac{1}{315}$$

$$speedUp = \frac{p2}{p1} = \frac{430}{315} \approx 1.36 \rightarrow 36\%$$

میزان هزینه سخت‌افزاری:

$$\frac{Logic\ Elements2}{Logic\ Elements1} = \frac{6{,}933}{5{,}177} \approx 1.34 \rightarrow 34\%$$

میزان کارایی بر هزینه:

$$\left.\begin{aligned} Cost\ per\ performance1 &= \frac{5{,}177}{\frac{1}{430}} \\ Cost\ per\ performance2 &= \frac{6{,}933}{\frac{1}{315}} \end{aligned}\right\} \quad \frac{6{,}933 \times 315}{5{,}177 \times 430} \approx 0.98 \rightarrow 2\%$$

با توجه به محاسبات باال نتیجه میگیریم که با اضافه شدن Cache، هزینه سخت افزاری افزایش یافته است اما کارایی نیز بهبود داشته است. نسبت هزینه به افزایش کارایی نشان میدهد که افزودن Cache بطور کلی باعث بهبود عملکرد سیستم به نسبت هزینه شده است که این نتیجه مطابق با انتظار ما است.

سه حالتی که در این سه جلسه مورد بررسی قرار داده‌ایم را یکدیگر مقایسه میکنیم:

	کارایی	هزینه	کارایی بر هزینه
حافظه داخلی	195	7928	195×7928=1,545,960
SRAM	430	5177	430×5177=2,226,110
SRAM + Cache	315	6933	315×6933=2,183,895

همانطور که در جدول قابل مشاهده است، کارایی در حالتی که حافظه درون پردازنده قرار دارد، نسبت به دو طراحی دیگر، بهتر است اما هزینه نیز بسیار زیاد است. هنگامی که SRAM را اضافه می‌کنیم، کارایی بسیار کاهش می‌یابد اما در طرف دیگر هزینه نیز کاهش چشمگیری دارد. با اضافه شدن Cache، کارایی نسبت به حالت قبل بهبود یافته اما هزینه افزایش می‌یابد. بنابراین در هر مرحله می‌بینیم که tradeoff بین هزینه و کارایی وجود دارد. در نهایت با بررسی و مقایسه های انجام شده، می‌توان نتیجه گرفت که طراحی سوم، بهترین گزینه برای پیاده سازی خواهد بود.

مشکلات:

در ابتدا کد روی مادلسیم درست کار نمی‌کرد. با بررسی دوباره و دوباره کد ها در نهایت متوجه شدیم که wire های جدید که باید در MEM_Stage برای ورودی و خروجی ماژول Cache_controller به این قسمت اضافه می‌کردیم، تعریف نشده اند.

در هنگام پیاده سازی طراحی روی FPGA در ابتدا مقادیر صحیح روی signal tab جابجا نمایش داده می‌شدند، که با اضافه کردن frequency divider که کد آن در جلسه قبل توضیح داده شده است، این مشکل رفع شد.

بخش امتیازی:

برای افزایش کارایی پردازنده ها روش های گوناگونی وجود دارد که در این جا به معرفی چند تایی از آنها می‌پردازیم:

استفاده نکردن از Inout Bus:

با جایگزین کردن باس دو طرفه می‌توان بطور همزمان داده‌های بیشتری را انتقال داد. از آنجاییکه در استفاده از bus، هر ماژول بر اساس اولویت، می‌تواند از bus استفاده کند، تاخیر در سیستم بوجود خواهد آمد که در صورتیکه بین ماژول ها، سیم کشی مستقیم داشته باشیم، سیستم سریعتر عمل می‌کند.

استفاده از الگوریتم های مناسب:

Most Recently Used:

برخلاف LRU، در MRU داده هایی که اخیراً استفاده شده اند، اولویت بیشتری برای drop شدن دارند.

First in First out:

در این الگوریتم حافظه پنهان مانند صف FIFO عمل می‌کند. به این معنا که بلوک‌ها را به ترتیبی که اضافه شده اند drop می‌کند.

First in Last out or Last in First out:

در این الگوریتم، حافظه نهان مانند یک پشته عمل می‌کند. به این معنا که ابتدا بلوکی را که اخیراً اضافه شده است، بدون توجه به اینکه چند بار یا چند بار قبلاً به آن دسترسی داشته است، drop می کند.

و دیگر الگوریتم هایی که در ادامه نام آنها گفته شده است:

Least-frequently used

Least frequent recently used

Random Replacement

Time Aware Least Recently Used

LFU with dynamic aging

Pseudo-LRU