

گزارش کار آزمایشگاه معماری کامپیوتر

۱ MIPS پردازنده سازی

در این بخش پردازنده‌ای طراحی کردیم که می‌تواند به ۱۸ نوع دستور مختلف پاسخ منطقی بدهد. همچنین طراحی بر مبنای معماری pipeline‌ای و در ۵ مرحله صورت گرفته است که عبارتند از: EXE_Stage، ID_Stage، IF_Stage، Mem_Stage و WB_Stage. در ادامه به اختصار وظیفه‌ی هر یک از این مراحل توضیح داده خواهد شد. همچنین مازول‌های رجیسترها می‌یابیم که ترتیب عبارتند از: Mem_Reg، Exe_Reg، ID_Stage_Reg، IF_Stage_Reg و Mem_Reg است. همچنین نگهداری داده‌ها را برای حداقل یک کلاک سایکل به عهده دارد.

• مازول IF_Stage

این مازول به طور کلی وظیفه‌ی تغییر PC (که شامل افزایش ۴ واحدی آن یا ثابت نگه داشتن آن یا به طور کل تغییر آن به یک مقدار دیگر به دلیل وجود branch است، می‌باشد) و خواندن دستورهای مناسب از RAM را دارد. بنابراین یکی از مازول‌های اصلی که در IF_Stage inst_mem است. inst_mem بر حسب مقدار PC، در هر کلاک سایکل، محتوای آدرسی از RAM را برمی‌گرداند که به صورت زیر است:

```
assign addr_temp = (PC>>2)<<2;
assign out = {ram[addr_temp], ram[addr_temp+1], ram[addr_temp+2], ram[addr_temp+3]};
```

تصویر ۱: خروجی

• مازول ID_Stage

وظیفه‌ی کلی آن تنظیم سیگنال‌های کنترلی دستور فعلی در این استیج با استفاده از مازول controlUnit، پیدا کردن مقدار رجیسترها داخل دستور از طریق مازول RegisterFile و هندل کردن مقدار متغیرهایی که به استیج بعدی داده می‌شود بر حسب immediate بودن یا نبودن دستور (که از چند مالتی‌پلکس برای این پیاده‌سازی استفاده شده است).

• مازول RegisterFile: شامل ۳۲ رجیستر ۳۲ بیتی است که رجیستر اول مقدار ثابت صفر دارد. این مازول ۳ ورودی ۵ بیتی دارد که به ترتیب Dest_wb، src2 و src1 نامیده شده‌اند. دو رجیستر اول رجیسترها مبدأ و Dest_wb، رجیستر مقصود است که خروجی محاسبات (Result_WB که ۳۲ بیتی است) در آن نوشته می‌شود. نکته حائز اهمیت آنکه، نوشتن در این مازول با کلاک و خواندن از آن بدون کلاک است.

• مازول controlUnit: قابل مشاهده است تنظیم می‌کند. این مازول بدون کلاک کار می‌کند.

```
output reg WB_en, isImmediate, memRead, memWrite, output reg[3:0]executeCMD, output reg[1:0]BR, output reg twoSourced
controlUnit
```

تصویر ۲: سیگنال‌های کنترلی تنظیم شده توسط

^۱ برای مشاهده جدول همه نتایج به صفحه‌ی آخر گزارش مراجعه شود.

• مازول :EXE_Stage

وظیفه‌ی کلی آن شامل انجام محاسبات با کمک مازول ALU (که بدون کلاک کار می‌کند) و هندل کردن دستورات addPC و condCheck با مازول branch می‌باشد.

- مازول ALU: با توجه به سیگنال ۴ بیتی controlUnit که execCMD تنظیم کرده است عمل ریاضی خاصی بر دو مقدار ۳۲ بیتی ورودی این مازول انجام می‌شود که در تصویر ۳ می‌تواند مشاهده کرد.
- مازول BRTYPE: condCheck که ورودی ۲ بیتی این مازول است مشخص می‌کند که آیا وضعیتی که دو متغیر val1 و src2val نسبت به هم دارند شرط branch را برقرار کرده یا خیر. در صورت برقراری شرط، خروجی یک بیتی این مازول که out است را یک کرده و در غیر این صورت صفر می‌کنیم.
- مازول addPC: در صورتی که condCheck گفته شده که taken هست، با کمک این مازول PC به آدرسی که باید به آن جهش رخ دهد تغییر می‌کند.

```
module ALU (input[31:0] in1,in2, input[3:0]exeCMD, output reg[31:0]out);
    always@(exeCMD, in1, in2)begin
        case(exeCMD)
            4'd1: out <= in1+in2;
            4'd2: out <= in1-in2;
            4'd3: out <= in1&in2;
            4'd4: out <= in1|in2;
            4'd5: out <= ~(in1 | in2);
            4'd6: out <= in1^in2;
            4'd9: out <= $signed(in1)>>>in2;
            4'd10: out <= $signed(in1)>>>in2;
            4'd7: out <= in1<<in2;
            4'd8: out <= in1<<in2;
        endcase
    end
endmodule
```

تصویر ۳: مازول ALU

• مازول :Mem_Stage

دو ورودی ۳۲ بیتی به نامهای address و data دارد و دو ورودی ۱ بیتی controlUnit و MEMRead و MEMwrite (که آن‌ها را بر حسب opcode دستور تنظیم کرده است). همچنین یک خروجی ۳۲ بیتی به نام out دارد. این مازول در خود یک حافظه‌ی ۳۲*۶۴ بیتی دارد که می‌توانیم از آن خوانده یا در آن چیزی را بنویسیم. در صورتی که MEMRead یک باشد یعنی می‌خواهیم در آدرس حافظه‌ی address مقدار data را بنویسیم. در صورتی که MEMwrite یک باشد، می‌خواهیم از آدرس حافظه‌ی address مقدارش را خوانده و در out قرار دهیم. نکته‌ی آن که خواندن بدون کلاک و نوشتن با کلاک انجام می‌شود.

• مازول :WB_Stage

در این مرحله می خواهیم داده های را در یک رجیستر بنویسیم. تنها آن که داده می تواند از حافظه خوانده شده باشد یا نتیجه هی محاسبات ALU در EXE_Stage باشد. انتخاب بین این دو مقدار با سیگنال MEMRead که در استیچ قبلی به خاطر یک بودن آن از حافظه خواندیم، تعیین می شود. بنابراین در این مرحله تنها یک مالتی پلکسر برای انتخاب مقدار درست نیاز داریم.

- :IF_Stage_Reg

رجیستری است که با هر سیکل کلاک، سیگنال های ID_Stage را از freez, flush, PC, instruction منتقل می کند.

- :ID_Stage_Reg

رجیستری است که با هر سیکل کلاک، سیگنال های flush, Br_taken, MEM_R_En, MEM_W_En, WB_En_in, EXE_Stage را از EXE_CMD, Dest, Reg2, Val1, Val2, PC منتقل می کند.

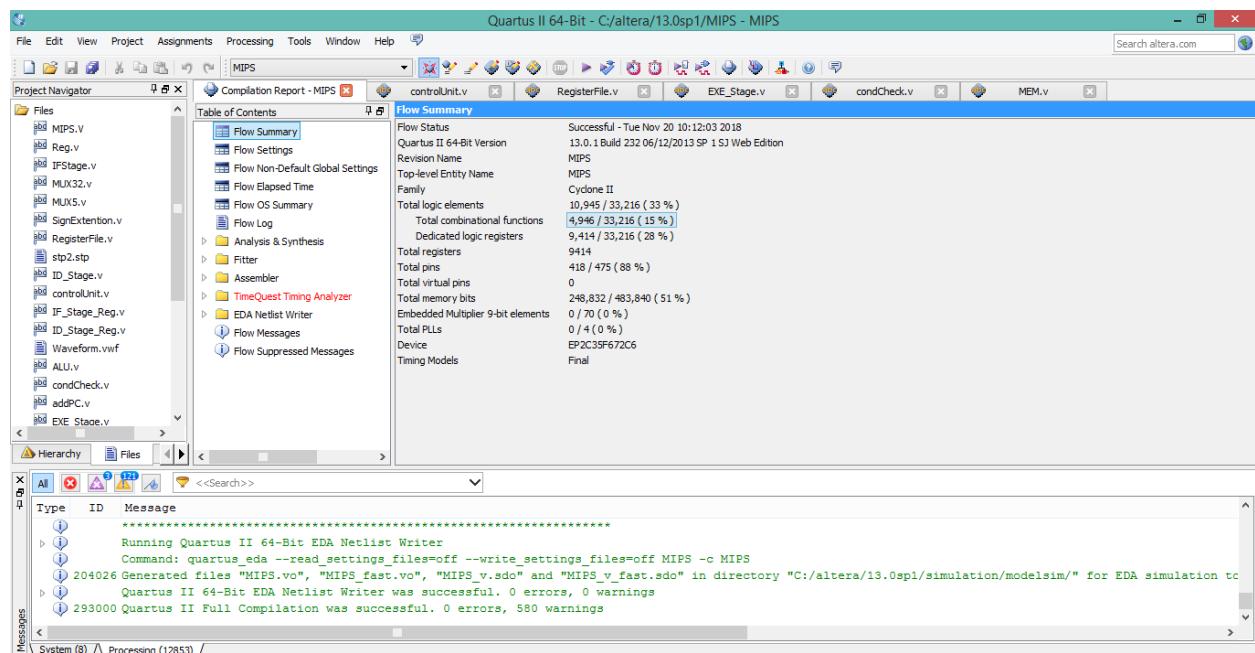
- :Exe_Reg

رجیستری است که با هر سیکل کلاک، سیگنال های MEM_R_En, MEM_W_En, WB_En_in, ALU_result, Mem_Stage به EXE_Stage منتقل می کند.

- :Mem_Reg

رجیستری است که با هر سیکل کلاک، سیگنال های MEM_R_En, WB_En_in, ALU_result, Mem_read_value, WB_Stage به Mem_Stage منتقل می کند.

برای جلوگیری از انواع hazard، در این مرحله بین هر دو دستوری که وابستگی آن ها خطایجاد می کرد، دو دستور NOP قرار دادیم تا نتیجه هی نهایی درست باشد. در مجموع تعداد کلاک سایکل ها شده ۲۷۳ شد (تصویر ۲۱ در پایان گزارش). در تصویر ۴ نیز می توان گزارش زمان کامپایل کد که شامل تعداد المنت های استفاده شده است را مشاهده کرد.



تصویر ۴: گزارش زمان کامپایل

مشکلاتی که با آن‌ها مواجه شدیم:

- روش‌های دیباگ کردن کد. از جمله چطور هر سیگنال ورودی به یک ماژول را دقیقاً به همنام آن assign کنیم.

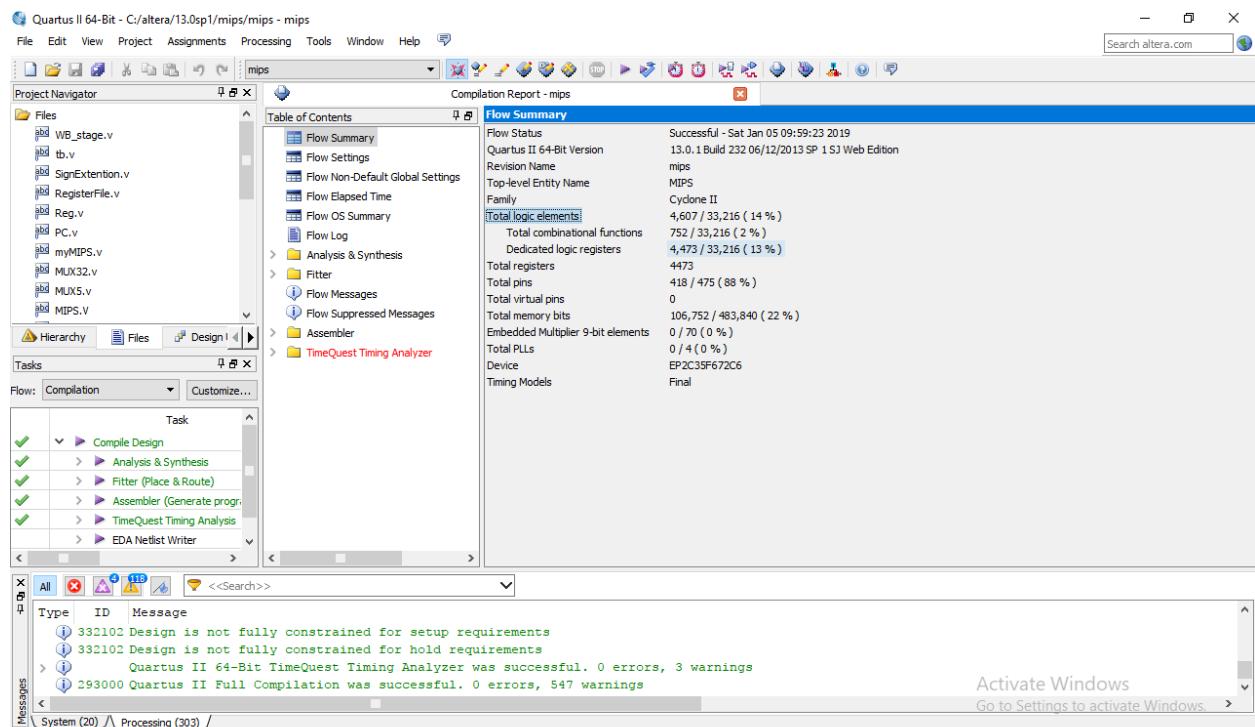
MIPS mips(.CLOCK_50(clk), .SW(sw));

تصویر ۵: assign یک به یک ورودی‌های ماژول به آن

- روش نوشتن Testbench برای پردازندeman و تست درستی آن در Modelsim. در نهایت همانند تصویر ۶ آن را اجرا دادیم.

```
module tb();
    reg clk,rst;
    myMIPS my(clk,rst);
    initial begin
        clk = 0;
        rst = 0;
        #25 rst = ~rst;
        #30 rst = ~rst;
        repeat(600) #50 clk = ~clk;
    end
endmodule
```

تصویر ۶



تصویر ۷: گزارش کامپایل

اضافه نمودن ماژول Hazard Detection

در این بخش با طراحی یک مازول Hazard Detection، در صورت بروز هazard سیگنال‌های PC و IF_Stage، در صورت freeze را در Reg Instruction ID_Stage کنترلی در ID_Stage را در صورت وجود هazard صفر می‌کنیم. با اضافه کردن این مازول دیگر نیازی به استفاده از دستورهای NOP نیست.

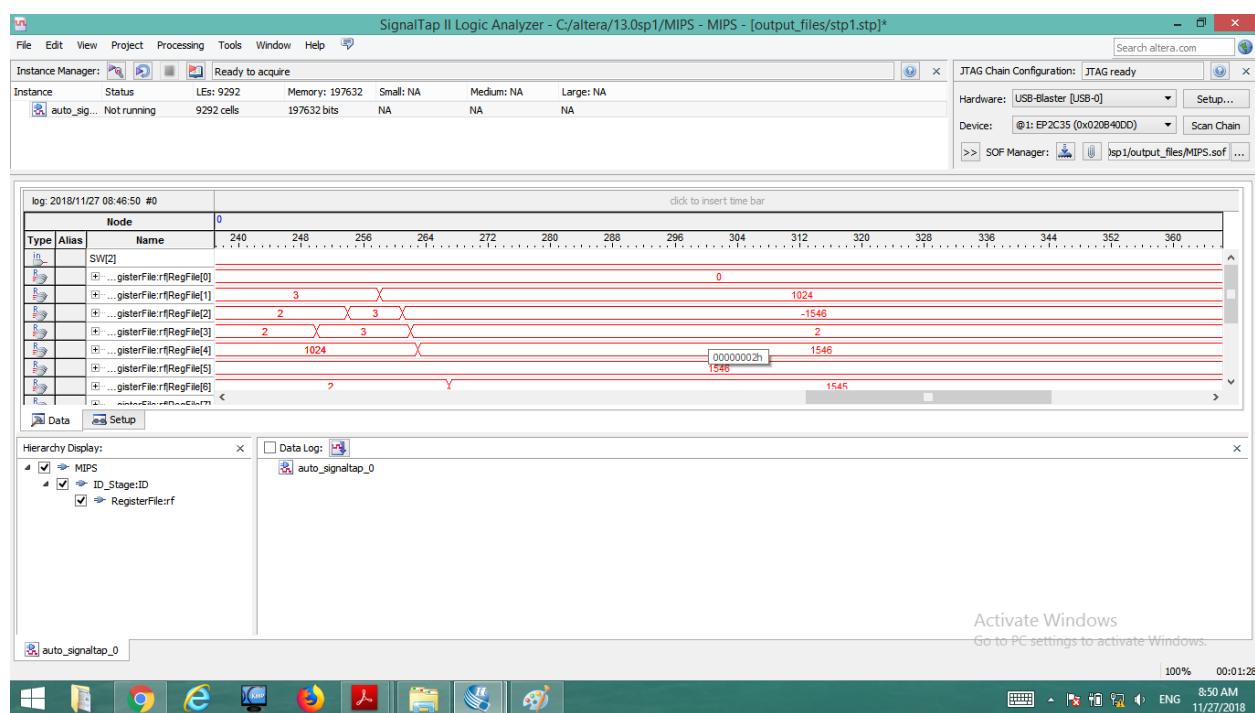
پیاده‌سازی مازول Hazard Detection در تصویر ۸ قابل مشاهده است. این مازول دو رجستر مبدا دستوری که به ID_Stage وارد شده (به ترتیب آن‌ها را src1 و src2 بنامیم) را به عنوان ورودی گرفته با رجستر مقصد دستورهایی که در MEM_Stage قرار دارد مقایسه می‌کند. همچنین سیگنال Mem_write مربوط به دستوراتی که در EXE_Stage و MEM_Stage را نیز به عنوان ورودی دریافت می‌کند.

در صورتی که سیگنال EXE_Stage در Mem_write یک باشد و رجیستر مقصد آن با یکی از دو مقدار src1 و src2 یکی باشد در آن صورت هazard داریم و باید سیگنال hazardDetected این مازول عدد یک را برگرداند. همچنین در صورتی که Mem_Stage در Mem_write یک باشد و رجیستر مقصد آن با یکی از دو مقدار src1 و src2 یکی باشد باز هم hazardDetected یک است.

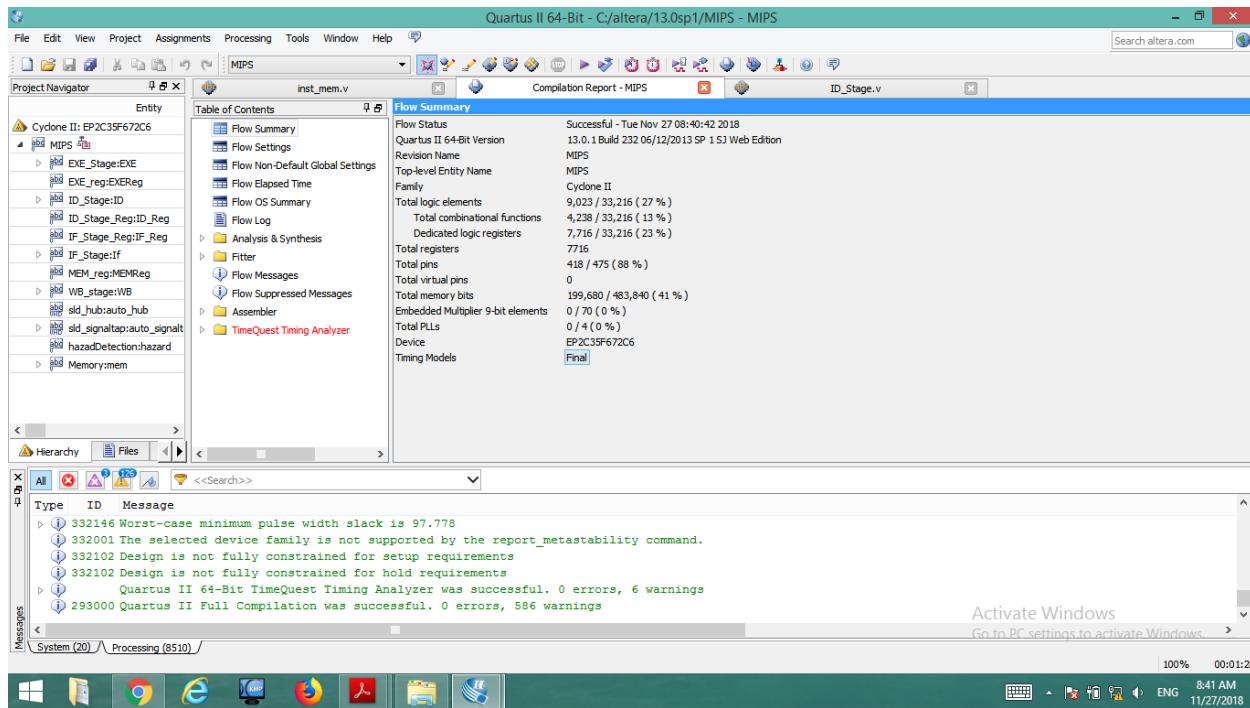
```
module hazardDetection(input[4:0]src1,src2,dest_EXE,dest_mem, input twoSources, wbEN_EXE, wbEN_mem, output hazardDetected);
    assign hazardDetected = !twoSources && (wbEN_EXE && dest_EXE == src1)? 1
    :!twoSources && (wbEN_mem && dest_mem == src1)? 1
    :twoSources && (wbEN_EXE && (dest_EXE == src1 || dest_EXE == src2))?1
    :twoSources && (wbEN_mem && (dest_mem == src1 || dest_mem == src2))? 1 :0;
endmodule
```

تصویر ۸: پیاده‌سازی مازول

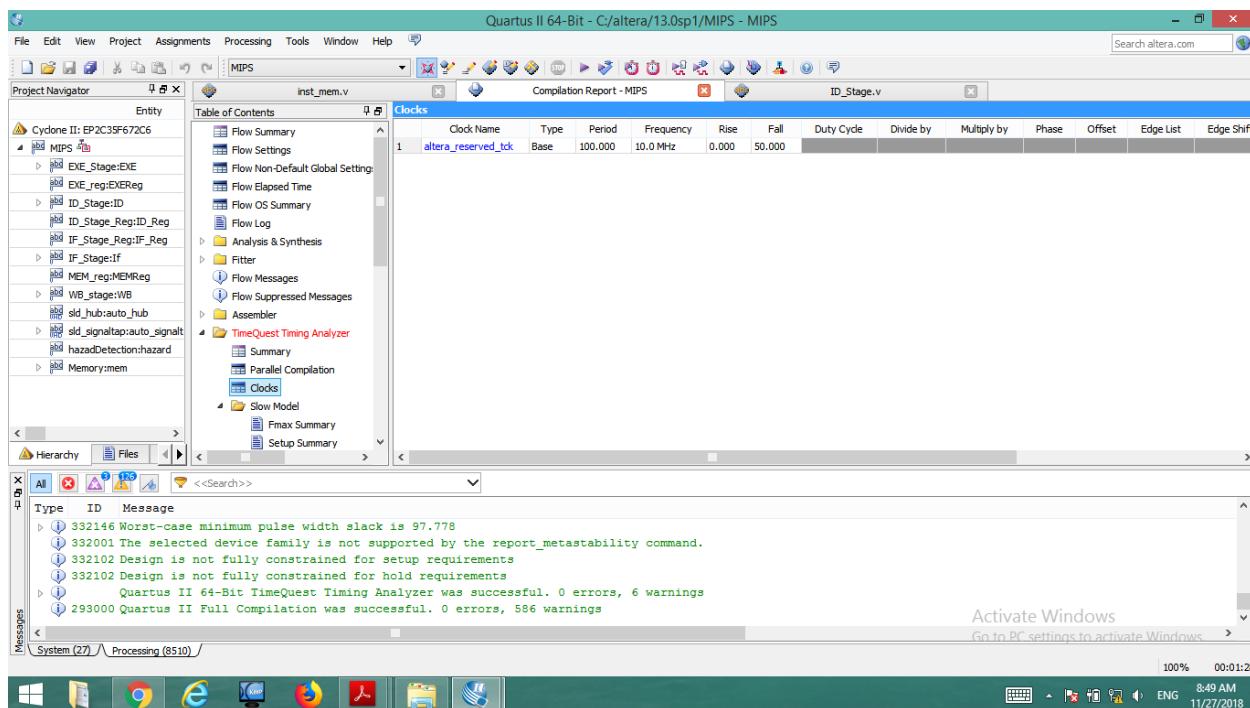
همان طور که در شکل ۹ قابل مشاهده است، با اضافه کردن مازول هazard تغییری در خروجی‌های نهایی نداریم. در حدود کلاک ۲۶۳ به خروجی مطلوب (پیش از دستور JMP) رسیدیم که این عدد تقریباً برابر با تعداد کلاک‌ها در روشی است که مازول هazard را نداشتیم و از NOP برای جلوگیری از خطا استفاده می‌کردیم. دلیل یکسان بودن تقریبی زمانی که هazard را اضافه کردیم هر جا که خطا داریم مازول هazard دستورات پیشین را stall می‌کند که معادل با افزودن دستور NOP در جاهایی است که خطا را پیش‌بینی کرده بودیم.



تصویر ۹: تعداد کلاک‌ها تا رسیدن به خروجی بعد از افزودن مازول هazard



تصویر ۱۰: گزارش زمان کامپایل مربوط به تعداد المنتهای اسفاده شده



تصویر ۱۱: گزارش زمان کامپایل مربوط به کلاک

پیاده سازی تکنیک Forwarding

در این بخش با طراحی یک مازول Forwarding، می‌توانیم خروجی‌های MEM_Stage و WB_Stage را مستقیماً به EXE_Stage (مازول ALU) بدهیم تا در تعداد کل‌اک‌ها صرفه‌جویی شود. با استفاده از روش Forwarding دیگر نیازی نیست تا داده‌ها در رجستر یا مموری ذخیره شده و دوباره از EXE_Stage آنجا خوانده و به داده شود که در نتیجه تا حدی تعداد هزاردها و به تبع از آن تعداد stall‌ها را نیز کاهش می‌دهد.

پیاده سازی مازول Forwarding در تصویر ۱۲ قابل مشاهده است. این مازول رجیسترها مبدأ و مقصد ID_Stage، رجیسترها مقصد MEM_Stage و WB_Stage مربوط به WB_en، سیگنال WB_Stage و MEM_Stage را به عنوان ورودی دریافت می‌کند.

```
module forwarding(input enable, input[4:0] IDRegDest, IDRegsrc1, IDRegsrc2, EXERegDest,
    MEMRegDest, input EXERegWBen, MEMRegWBen, output [1:0] selVal1, selVal2, selImm);
    assign selVal1 = (EXERegDest == IDRegsrc1 && EXERegWBen) && (!enable) ? 2'd1 :
        (MEMRegDest == IDRegsrc1 && MEMRegWBen) && (!enable) ? 2'd2 : 2'd0;
    assign selVal2 = (EXERegDest == IDRegsrc2 && EXERegWBen) && (!enable) ? 2'd1 :
        (MEMRegDest == IDRegsrc2 && MEMRegWBen) && (!enable) ? 2'd2 : 2'd0;
    assign selImm = (EXERegDest == IDRegDest && EXERegWBen) && (!enable) ? 2'd1:
        (IDRegDest == MEMRegDest && MEMRegWBen) && (!enable) ? 2'd2 : 2'd0;
endmodule
```

تصویر ۱۲: پیاده سازی مازول Forwarding

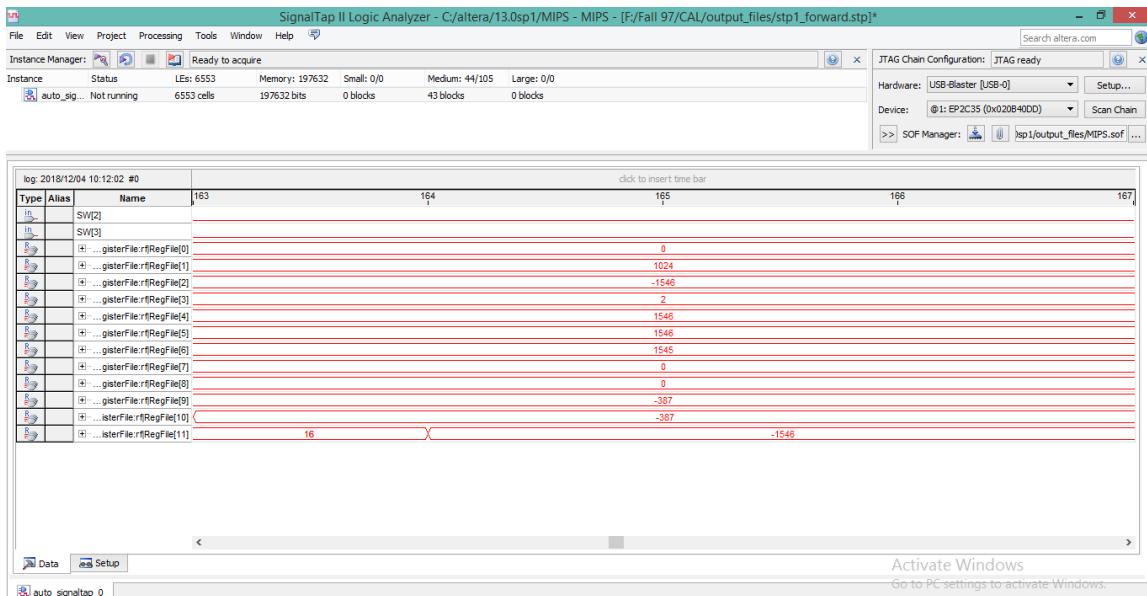
خروجی‌های این مازول به عنوان سلکتور ۳ مالتی‌پلکسرهای جدید در EXE_Stage است. مالتی‌پلکسرهای جدید به صورت نمایش داده شده در تصویر ۱۳ هستند (خطوط ۷ تا ۹) که مالتی‌پلکسرهای داده‌ای وارد شده از ID_Stage داده‌های وارد شده از MEM_Stage یا WB_Stage را بر حسب مقدار سلکتور از خود عبور می‌دهند.

```
1 module EXE_Stage(input clk, input[1:0] selVal1, selVal2, selImm, input[3:0] EXE_CMD,
2     input[31:0]memInput, WBInput, val1, val2, valSrc2, PC, input[1:0] BRTypE,
3     output[31:0] ALU_result, Br_addr, output Br_taken);
4     wire[31:0]ALUInput1, ALUInput2, IMMmuxOutput;
5
6 // new muxs for forwarding
7 MUX32_3input mux1(val1, memInput, WBInput, selVal1, ALUInput1);
8 MUX32_3input mux2(val2, memInput, WBInput, selVal2, ALUInput2);
9 MUX32_3input mux3(valSrc2, memInput, WBInput, selImm, IMMmuxOutput);
10
11 //ALU (input[31:0] in1,in2, input[3:0]exeCMD, output[31:0]out);
12 ALU alu(ALUInput1, ALUInput2, EXE_CMD, ALU_result);
13
14 //condCheck(input[31:0]val1, src2Val, input[1:0]BRTypE, output out);
15 condCheck cond_check(ALUInput1, IMMmuxOutput, BRTypE, Br_taken);
16
17 //addPC(input[31:0]PC, offset, output[31:0]out);
18 addPC add_PC(PC, ALUInput2, Br_addr);
19 endmodule
```

تصویر ۱۳: ۳ مالتی‌پلکسراضافه شده در خطوط ۷ تا ۹ EXE_Stage

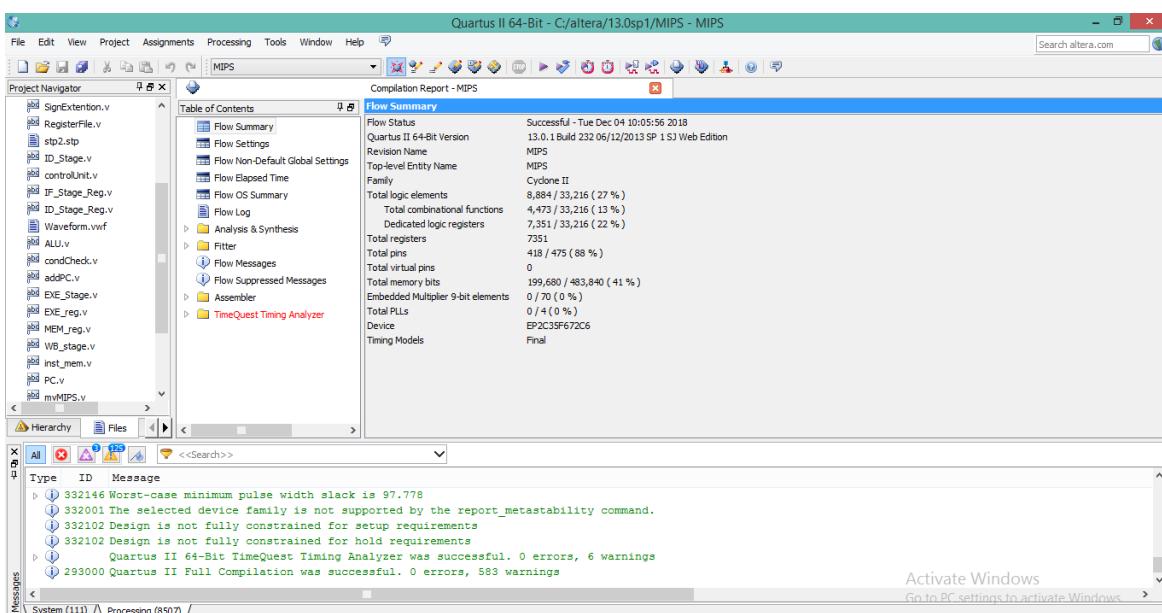
همچنین مازول هazard را اندکی تغییر می‌دهیم. بدین صورت که با قرار دادن یک flag آن را فعال یا غیرفعال می‌کنیم. در صورتی که مازول Forwarding را بخواهیم استفاده کنیم، مازول هazard را غیرفعال می‌کنیم. برای تنظیم این flag از یکی از کلیدهای روی FPGA، [3] SW استفاده می‌کنیم.

همان طور که در شکل ۱۴ قابل مشاهده است، با اضافه کردن مازول Forwarding تغییری در خروجی‌های نهایی نداریم. با فعال کردن مازول جدید و غیرفعال کردن مازول هazard در حدود کلاک ۱۶۳ به خروجی مطلوب (پیش از دستور JMP-1) رسیدیم که کمتر از تعداد سیکل‌های مراحل قبلی است.



تصویر ۱۴: تعداد کلاک‌ها تا رسیدن به خروجی بعد از افزودن مازول Forwarding

گزارش‌های زمان کامپایل بعد از افزودن مازول Forwarding را می‌توان در تصویر ۱۵ مشاهده کرد.



تصویر ۱۵: گزارش زمان کامپایل مربوط به تعداد المنتهای استفاده شده

در مقایسه‌ی این روش با روش هازارد داریم:

- از لحاظ میزان کارایی داریم:

$$\frac{Hazard}{Forwarding} = \frac{263}{163} = 1.61$$

- از لحاظ هزینه‌های سختافزاری داریم:

$$\frac{Hazard}{Forwarding} = \frac{9023}{8884} = 1.01$$

در نهایت میزان کارایی بر هزینه به صورت زیر است:

$$\frac{Performance}{Cost} = \frac{1.61}{1.01} = 1.59$$

استفاده از SRAM به عنوان حافظه اصلی

در این بخش می‌خواهیم از حافظه‌ی SRAM که FPGA است و حافظه‌ی بیشتری را در اختیار ما قرار می‌دهد، استفاده کنیم. در اینجا تعریف می‌کنیم که خواندن و نوشتمن در SRAM شش کلک سایکل زمان می‌برد بنابراین انتظار داریم که با استفاده از این حافظه‌ی خارجی برنامه‌ی ما کندتر خواهد شد.

برای استفاده از SRAM به این صورت عمل می‌کنیم که MEM_Stage داده‌های لازم را به مازول SRAMCtrl می‌دهد و این مازول مراحل لازم برای خواندن و نوشتمن را هندل می‌کند. مازول را می‌توان در تصویر ۱۶ مشاهده کرد.

```
1 module SRAMCtrl(input clk, rst , Wr_EN, Read_EN, input[31:0] ADDR, writeData,
2   output reg[31:0] readData, output freeze, inout[15:0] SRAM_DQ, output reg [17:0] SRAM_ADDR,
3   output reg SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N);
4   reg[2:0] counter;
5   reg[15:0] tempData;
6   reg[15:0] dataDQ;
7   initial begin
8     counter=3'd0; SRAM_UB_N=0; SRAM_LB_N=0; SRAM_WE_N=0; SRAM_CE_N=0; SRAM_OE_N=0;
9   end
10 always@(posedge clk)begin
11   if(rst)begin
12     SRAM_UB_N=0; SRAM_LB_N=0; SRAM_WE_N=0; SRAM_CE_N=0; SRAM_OE_N=0;counter <= 0;
13   end
14   else begin
15     if(Wr_EN)begin
16       counter = counter + 1;
17       case(counter)
18         3'd1:begin
19           SRAM_ADDR <= {ADDR[18:2],1'd0};
20           dataDQ <= writeData[15:0];
21           SRAM_WE_N <= 1'd0;
22         end
23         3'd2:begin
24           SRAM_ADDR <= {ADDR[18:2],1'd1};
25           dataDQ <= writeData[31:16];
26           SRAM_WE_N <= 1'd0;
27         end
28         3'd3: SRAM_WE_N <= 1'd1;
29         3'd4: SRAM_WE_N <= 1'd1;
30         3'd5: SRAM_WE_N <= 1'd1;
31         3'd6: begin counter <= 3'd0;SRAM_WE_N <= 1'd1; end
32       endcase
33     end
34   end
35 end
```

```

33
34     end else if(Read_EN) begin
35         SRAM_WE_N <= 1;
36         counter = counter + 1;
37         case(counter)
38             3'd1:begin
39                 SRAM_ADDR <= {ADDR[18:2],1'd0};
40             end
41             3'd2:begin
42                 SRAM_ADDR <= {ADDR[18:2],1'd1};
43                 tempData <= SRAM_DQ;
44             end
45             3'd3:begin
46                 readData <= {SRAM_DQ, tempData};
47             end
48             3'd4: SRAM_WE_N <= 1'd1;
49             3'd5: SRAM_WE_N <= 1'd1;
50             3'd6: counter <= 3'd0;
51         endcase
52     end else begin
53         counter <= 0;
54         SRAM_WE_N <= 1;
55     end
56 end
57 end
58 assign freeze = (Wr_EN || Read_EN)&(counter < 5) ? 1 : 0;
59 assign SRAM_DQ = Wr_EN ? dataDQ : 16'bxxxxxxxxxxxxxx;
60
61 endmodule

```

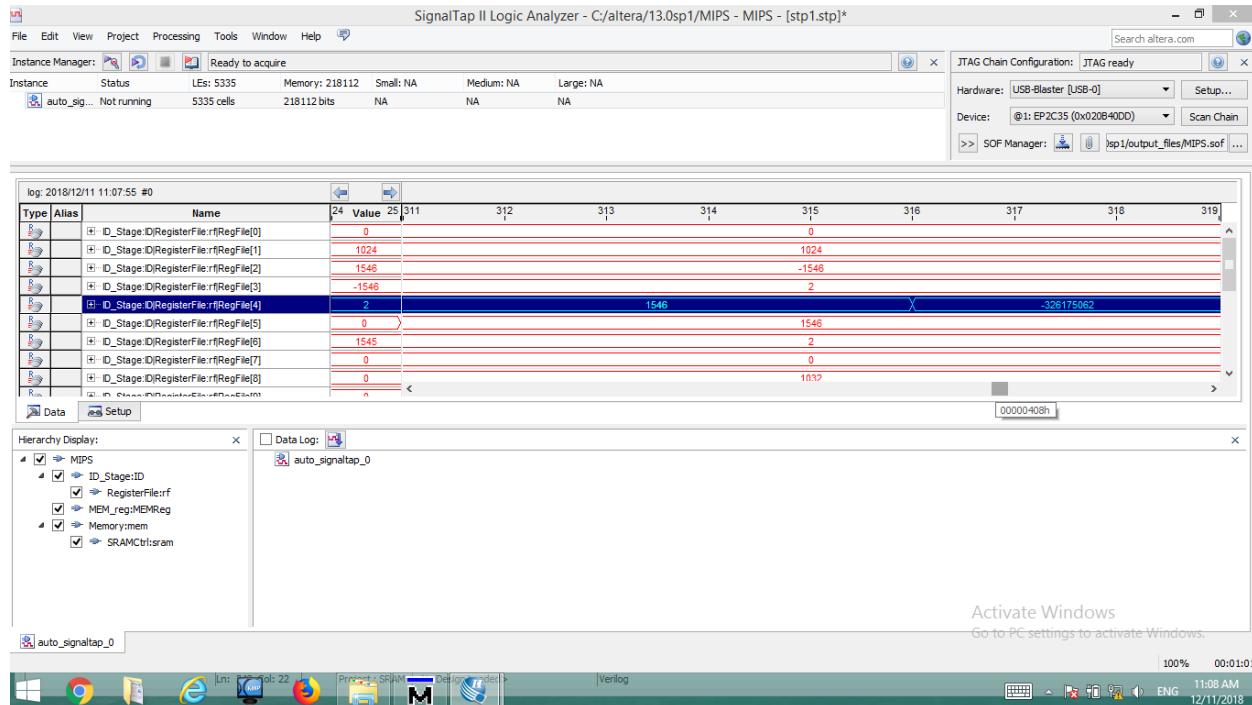
تصویر ۱۶: پیاده‌سازی ماثول SRAMCtrl

همانطور که پیش‌تر گفته شد، خواندن و نوشتن در ۶ کلاک صورت می‌گیرد. برای حساب کردن تعداد کلاک سایکل‌ها از یک شمارنده به نام counter استفاده می‌کیم. بدین صورت که در ابتدای ماثول شمارنده را مقدار دهی اولیه می‌کیم (counter = 0). سپس هرگاه خواستیم بخوانیم یا بنویسیم (یعنی Read_EN یا Wr_EN یک بودند) مقدار این شمارنده را یکی افزایش می‌دهیم (خطوط ۱۶ و ۳۶ نشان‌داده شده در شکل ۱۶).

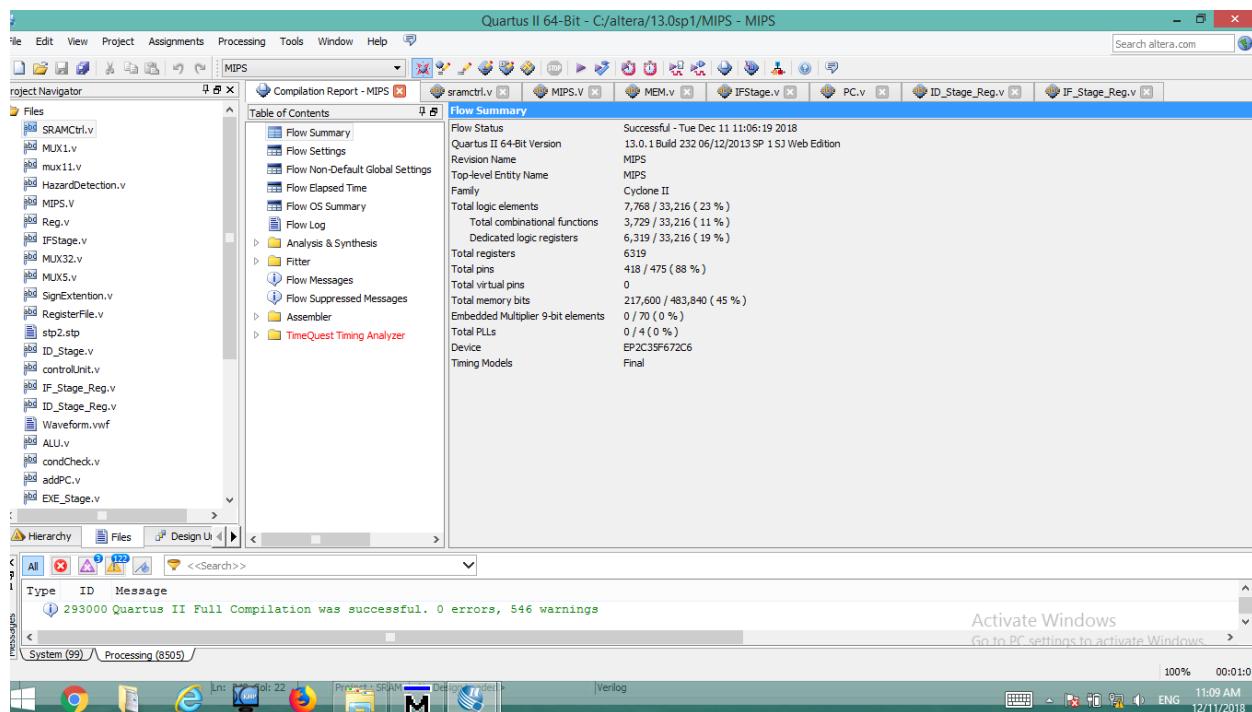
برای نوشتن: در اولین سیکل نوشتن، آدرس و بخش کمازش‌تر داده را قرار می‌دهیم و SRAM_WE_EN را صفر می‌کنیم که بداند می‌خواهیم بنویسیم. در سیکل دوم بخش پارازش‌تر داده را قرار می‌دهیم. برای خواندن: در اولین سیکل خواندن، آدرس اشاره کننده به خانه کمازش‌تر را قرار می‌دهیم. در سیکل دوم داده‌ی سیکل قبل را دریافت کرده و آدرس اشاره کننده به خانه بالازش‌تر را قرار می‌دهیم. در سیکل سوم داده سیکل قبل را دریافت می‌کنیم. در انتهای زمانی که خواستیم داده را به ماثول ببرونی بدهیم، داده‌ی برگردانده شده در سیکل دوم و سوم خواندن را با هم ترکیب کرده و در SRAM_DQ قرار می‌دهیم.

همچنین از آنجا که استفاده از SRAM بسیار زمان‌بر است باید به stall/freeze کردن دستورها توجه کنیم. زمانیکه که شمارنده عددی بین ۱ تا ۶ داشته باشد یعنی در حال خواندن یا نوشتن هستیم و باید دستورات stall شوند. این سیگنال که با نام MEM_Stage مشخص شده است به freeze ای‌ایی که قبلاً به رجسترها بین استیچ‌ها و PC داده شده بود ترکیب می‌شود.

همان طور که در شکل ۱۷ قابل مشاهده است، با استفاده از SRAM تغییری در خروجی‌های نهایی نداریم. در حدود کلاک ۳۱۱ به خروجی مطلوب (پیش از دستور JMP-1) رسیدیم.



تصویر ۱۷: تعداد کلاک‌ها تا رسیدن به خروجی بعد از استفاده از SRAM



تصویر ۱۸: گزارش زمان کامپایل مربوط به تعداد المنت‌های استفاده شده

در مقایسه‌ی این روش با روش هازارد داریم:

- از لحاظ میزان کارایی داریم:

$$\frac{Hazard}{SRAM} = \frac{263}{311} = 0.84$$

- از لحاظ هزینه‌های سختافزاری داریم:

$$\frac{Hazard}{SRAM} = \frac{9023}{7768} = 1.16$$

در نهایت میزان کارایی بر هزینه (هزینه‌ی هر المنت سختافزاری در یک کلک) به صورت زیر است:

$$\frac{Performance}{Cost} = \frac{0.84}{1.16} = 0.72$$

پیاده سازی Cache

در این بخش می‌خواهیم با پیاده‌سازی یک مازول Cache استفاده از SRAM را تسريع بخشیم. بدین صورت که زمانیکه داده‌ای از SRAM خوانده می‌شود آنرا در Cache نیز نگهداری می‌کنیم. در هنگام نوشتمن در SRAM نیز داده را از Cache پاک می‌کنیم.

ماژول جدیدی به نام CacheCtrl را همانند نشان‌داده شده در تصویر ۱۹ به پردازنده اضافه می‌کنیم.

```
1 module CacheCtrl(input clk, rst, sram_ready, input[63:0] sram_rdata, input[31:0] addr, wdata,
2   input MEM_R_EN, MEM_W_EN, output reg[31:0] rdata, output reg freeze,
3   output reg[31:0] sram_addr, sram_wdata, output reg sram_w_en, sram_r_en);
4   reg[63:0] data0[0:63];
5   reg[63:0] data1[0:63];
6   reg[8:0] tag0[0:63];
7   reg[8:0] tag1[0:63];
8   reg[63:0] valid0;
9   reg[63:0] valid1;
10  reg[63:0] LRU;
11  integer i;
12  reg hit0, hit1, hit, ready, cache_ready;
13  wire temp;
14  reg[63:0] temp1;
15  always@(*)begin
16    hit0 <= (addr[17:9] == tag0[addr[8:3]] && valid0[addr[8:3]]) ? 1 : 0;
17    hit1 <= (addr[17:9] == tag1[addr[8:3]] && valid1[addr[8:3]]) ? 1 : 0;
18
19    hit <= MEM_W_EN ? 0 : hit0 || hit1 || (!(MEM_R_EN || MEM_W_EN));
20    freeze <= !(sram_ready || hit); //l-> freeze in registers
21
22    if(hit)begin
23      if(hit0)begin
24        temp1 <= data0[addr[8:3]];
25        if(addr[2] == 0)
26          rdata <= temp1[31:0];
27        else
28          rdata <= temp1[63:32];
29      end
30      else if(hit1)begin
31        temp1 <= data1[addr[8:3]];
32        if(addr[2] == 0)
33          rdata <= temp1[31:0];
34      end
35    end
36  end
```

```

34         else
35             rdata <= temp1[63:32];
36         end
37     end
38     else if(sram_ready && MEM_R_EN)begin
39         if(addr[2] == 0)
40             rdata <= sram_rdata[31:0];
41         else
42             rdata <= sram_rdata[63:32];
43
44     end
45
46 end
47
48 always@(posedge clk) begin
49     if (rst) begin
50         sram_addr <= 32'd0;
51         sram_wdata <= 32'd0;
52         sram_w_en <= 1'd0;
53         sram_r_en <= 1'd0;
54         for(i = 0; i < 64; i=i+1 )begin
55             tag0[i] <= 9'd0;
56             tag1[i] <= 9'd0;
57             data0[i] <= 64'd0;
58             data1[i] <= 64'd0;
59             valid0[i] <= 0;
60             valid1[i] <= 0;
61             LRU[i] <= 0;
62         end
63     end
64
65
66     else if (MEM_W_EN) begin
67         sram_addr <= addr;
68         sram_wdata <= wdata;
69         sram_w_en <= MEM_W_EN;
70         sram_r_en <= MEM_R_EN;
71
72         if (hit0) begin
73             valid0[addr[8:3]] <= 1'd0;
74         end
75         else if(hit1)begin
76             valid1[addr[8:3]] <= 1'd0;
77         end
78     end
79
80     else if (MEM_R_EN) begin
81         if (!hit)
82             sram_addr <= addr;
83             sram_w_en <= MEM_W_EN;
84             sram_r_en <= MEM_R_EN;
85         end
86
87         else if(hit)begin
88             if (hit0)
89                 LRU[addr[8:3]] <= 1'd1;
90             else
91                 LRU[addr[8:3]] <= 1'd0;
92             end
93         end
94     if(sram_ready)begin
95         sram_w_en <= 0;
96         sram_r_en <= 0;

```

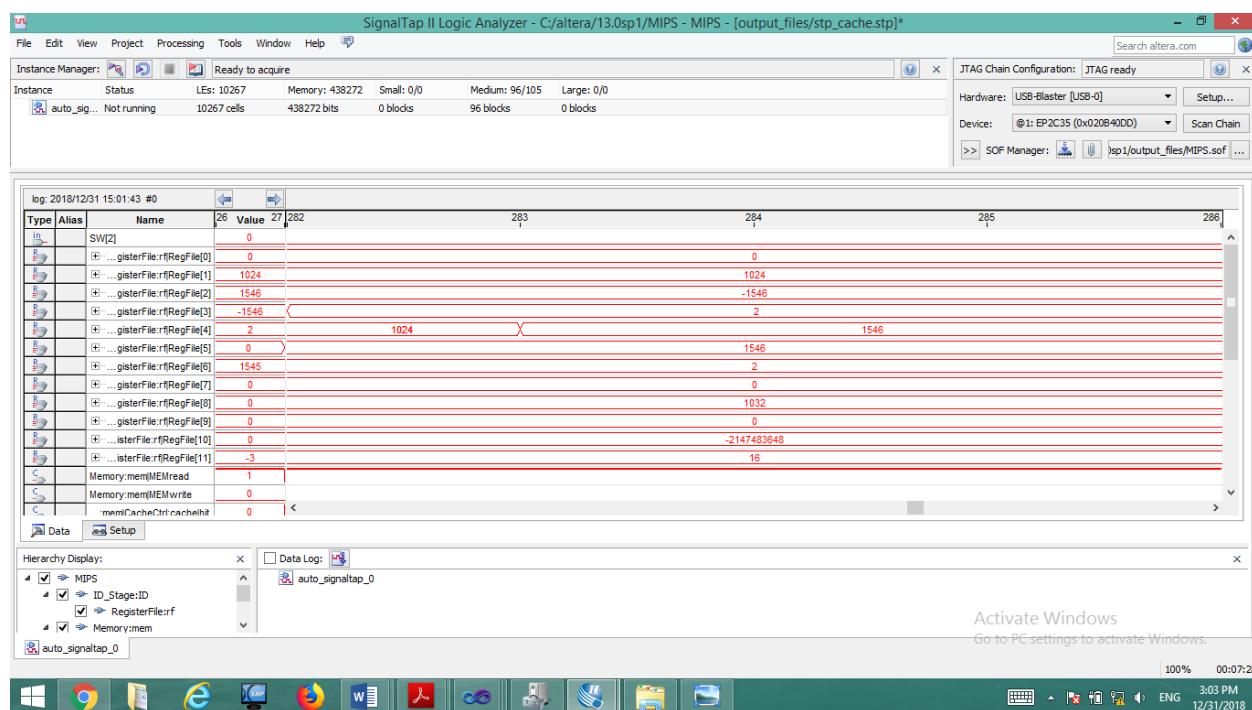
```

97    end
98
99    if (LRU[addr[8:3]] == 0 && MEM_R_EN && !hit && sram_ready) begin
100        data0[addr[8:3]] <= sram_rdata;
101        LRU[addr[8:3]] <= 1'd1;
102        valid0[addr[8:3]] <= 1'd1;
103        tag0[addr[8:3]] <= addr[17:9];
104    end
105    else if (LRU[addr[8:3]] == 1 && MEM_R_EN && !hit && sram_ready) begin
106        data1[addr[8:3]] <= sram_rdata;
107        LRU[addr[8:3]] <= 1'd0;
108        valid1[addr[8:3]] <= 1'd1;
109        tag1[addr[8:3]] <= addr[17:9];
110    end
111 end
112 end
113 endmodule

```

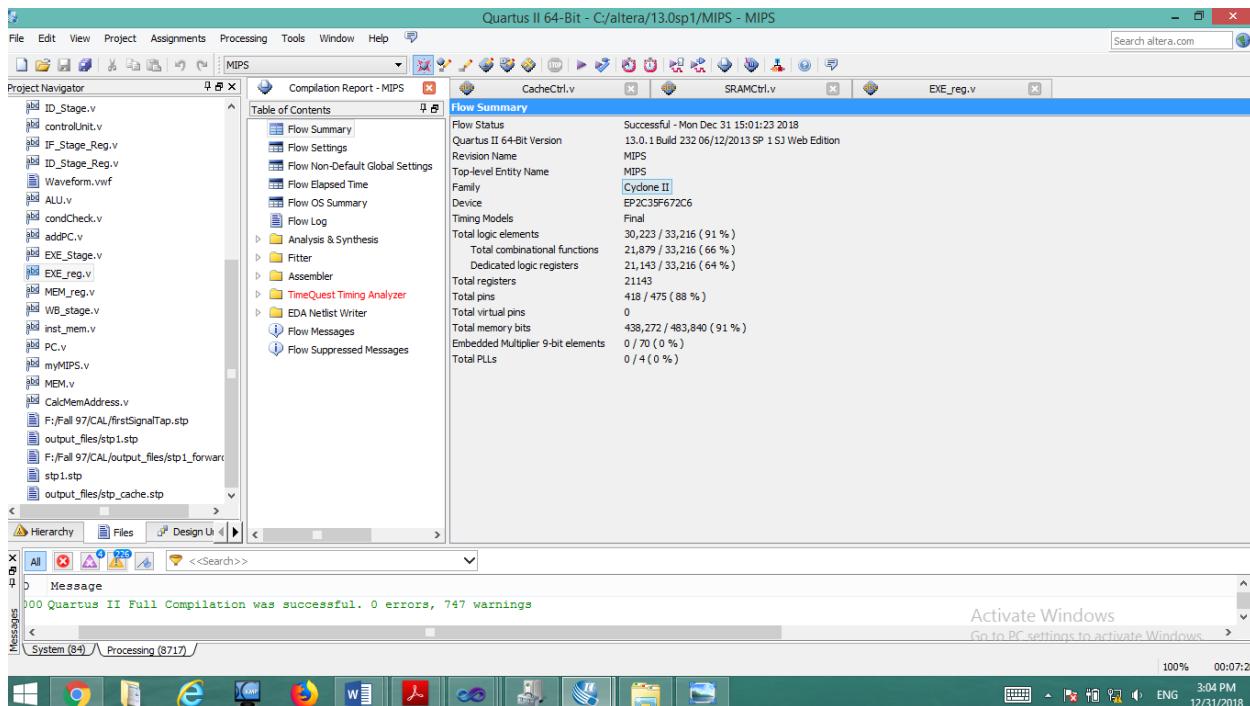
تصویر ۱۹: پیاده‌سازی مازول CacheCtrl

همانطور که در شکل ۲۰ قابل مشاهده است، استفاده از Cache باعث شده در ۲۸۳ کلک سایکل برنامه به نتیجه برسد.



تصویر ۲۰: نتیجه‌ی استفاده از Cache

گزارش زمان کامپایل را نیز می‌توان در تصویر ۲۱ مشاهده کرد.



تصویر ۲۰: گزارش زمان کامپایل

حال می‌خواهیم نتایج سه آزمایش آخر را با یکدیگر مقایسه کنیم.

- حالت ۱: استفاده از حافظهٔ داخلی
- حالت ۲: استفاده از SRAM موجود در برد
- حالت ۳: استفاده از Cache و SRAM

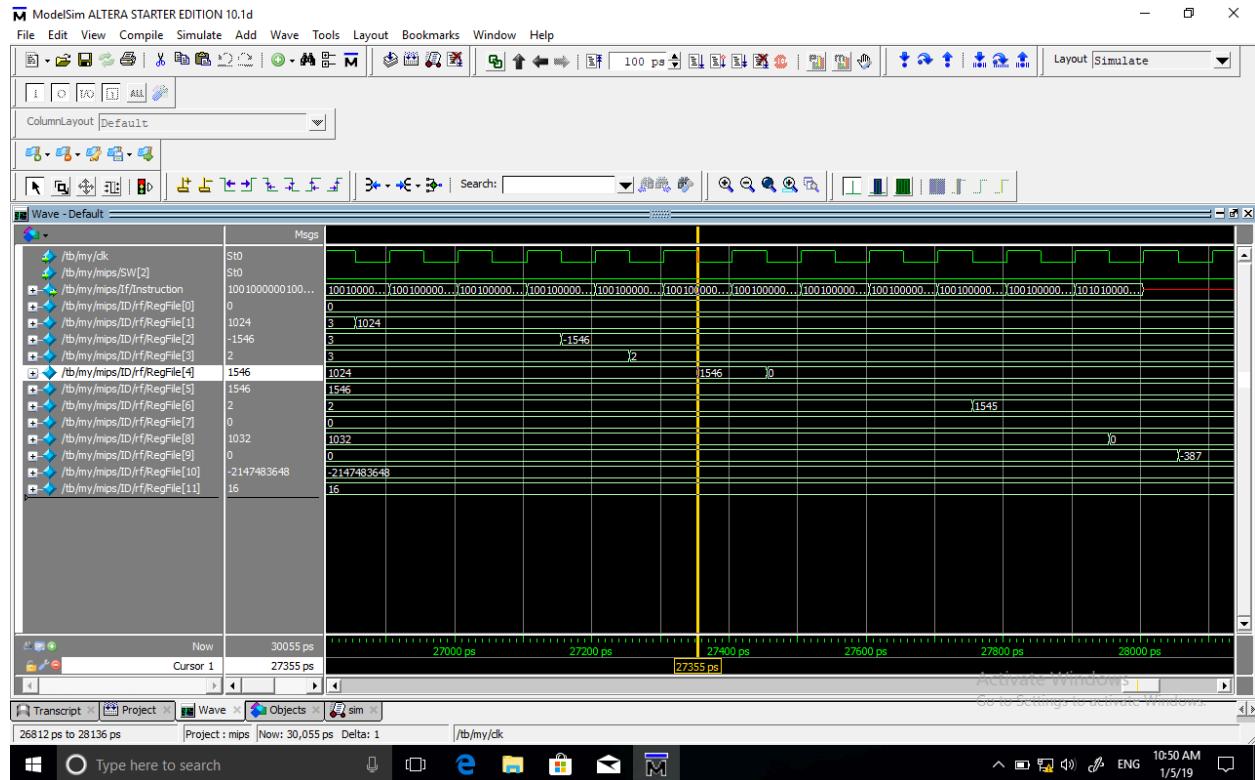
در بخش‌های بالاتر حالت ۱ و ۲ با یکدیگر مقایسه شده‌اند و می‌دانیم که گرچه استفاده از حافظهٔ داخلی محدودیت دارد اما بسیار سریع‌تر از زمانی است که از حافظهٔ SRAM استفاده می‌شود.

با توجه به جدول زیر می‌تواند گفت گزینه‌ی بهینهٔ ثابتی وجود ندارد، بلکه بر حسب کارکردی که از سیستم‌مان انتظار داریم می‌توانیم یکی از سه روش زیر را انتخاب کنیم. در صورتی که حافظهٔ زیاد می‌خواهیم و زمان بریمان مهم است استفاده از حالت ۳ پیشنهاد می‌شود. اگر می‌خواهیم هزینهٔ سخت‌افزاری‌مان کمتر باشد اما نیاز به حافظهٔ زیاد داریم حالت ۲ مناسب است و اگر نیاز به حافظهٔ چندانی نداریم استفاده از حالت ۱ پیشنهاد می‌شود.

	Number of Clock Cycles	Total Logic Elements
Internal Memory (Hazard)	263	9023
SRAM without Cache	311	7768
SRAM with Cache	283	30223

جدول مقایسه کلی همهی آزمایش‌ها

	Number of Clock Cycles	Total Logic Elements	Performance per Cost (each stage compared to its previous)
MIPS	273	4607	-
Hazard	263	9023	0.491881350021658
Forwarding	163	8884	0.629468878986104
SRAM	311	7768	2.1820872795739
SRAM with Cache	283	30223	0.233882481060133



تصویر ۲۱: تعداد کلاک سایکل‌های یک MIPS