گزارش درس آزمایشگاه معماری

عرفان عسگری – ۸۱۰۱۹۹۴۶۰

سالار صفردوست - ۸۱۰۱۹۹۴۵۰

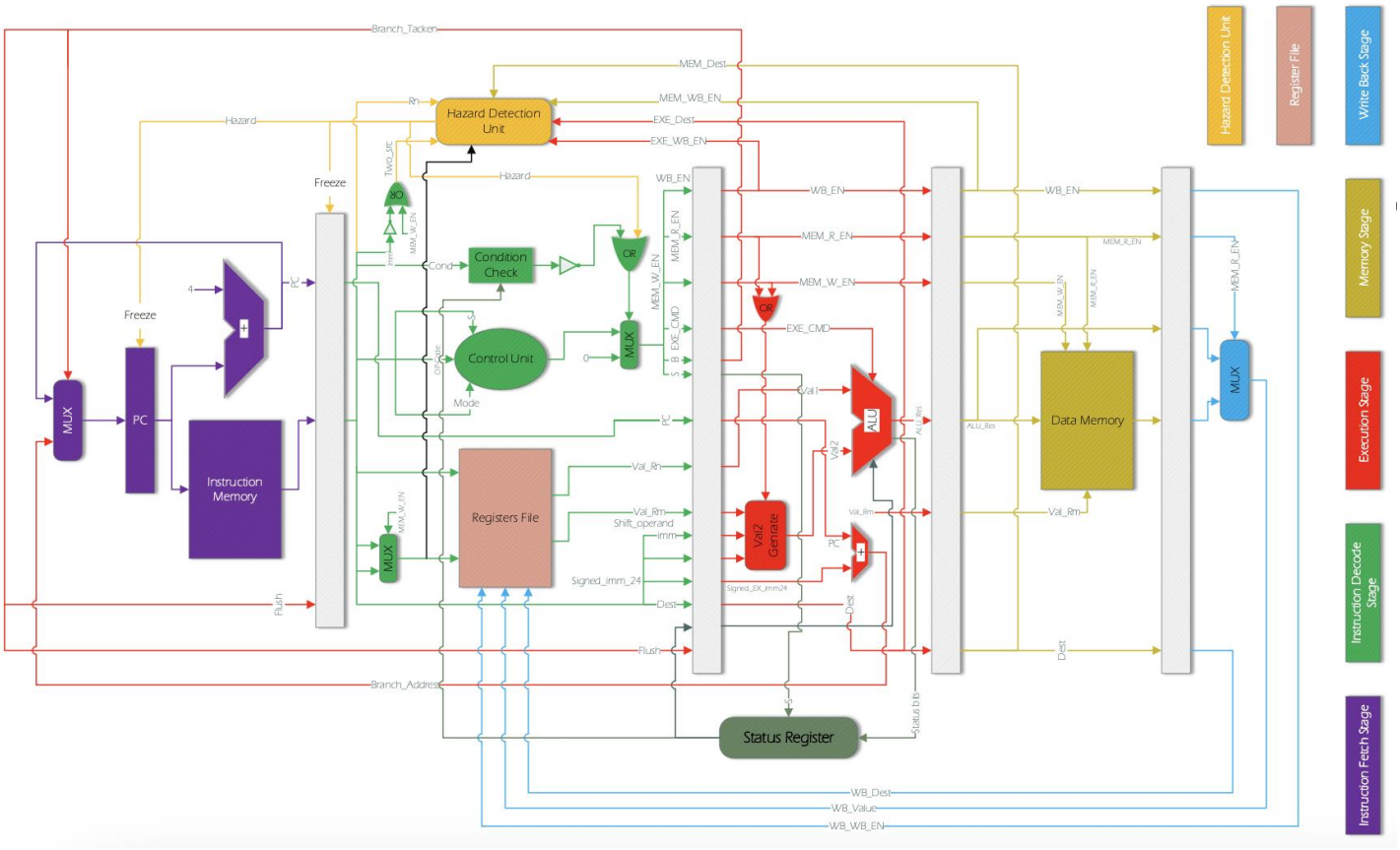
تیر ۱۴۰۳

دانشکده‌ی مهندسی برق و کامپیوتر،

دانشکده‌ی فنی، دانشگاه تهران

پیاده سازی پردازنده ARM

هدف پیاده‌سازی سخت افزاری پردازنده ARM مطابق شکل ۱-۱ است.



شکل 1- پردازنده *ARM*

استیج Instruction Fetch

برای این کار ابتدا به سراغ پیاده سازی Instruction Fetch Stage می‌رویم. برای این کار تمام قسمت‌های بنفش شکل ۱-۱ را باید پیاده‌سازی کنیم. این بخش شامل رجیستر PC است همچنین هر بار ۴ واحد به مقدار PC افزوده می شود برای انجام دستور بعدی. این بخش یک Instruction Memory هم دارد که دستورات پردازنده در آن ذخیره می‌شوند. سپس۴ استیج باقی‌مانده Pipe و ۵ رجیستر پردازنده را بدون محتویات داخلشان پیاده سازی می‌کنیم تا شاهد حرکت PC باشیم.

پیاده سازی IF به صورت زیر است:

```
module IF_Stage(
    input clk, rst, freeze, branchTaken,
    input[31:0] branchAddress,
    output[31:0] pc, instruction);

    wire[31:0] pcIn, pcOut;

    Mux_32b mux_32b(branchTaken, pc, branchAddress, pcIn);
    PC_Reg pc_reg(clk, rst, freeze, pcIn, pcOut);
    PC_Adder pc_adder(pcOut, pc);
    Instruction_Memory instruction_memory(pcOut, instruction);
endmodule
```

```
module IF_Stage_Reg(
    input clk, rst, freeze, flush,
    input[31:0] pcIn, instructionIn,
    output reg[31:0] pc, instruction
);
    always @(posedge clk or posedge rst) begin
        if(rst || flush) begin
            pc = 32'b0;
            instruction = 32'b0;
        end
        else begin
            if(freeze==1'b0) begin
                pc = pcIn;
                instruction = instructionIn;
            end
        end
    end
endmodule
```

```
module Instruction_Memory(
    input[31:0] address,
    output reg[31:0] instruction
    );

    always @(address)
    case (address)
      32'd0: instruction <= 32'b1110_00_1_1101_0_0000_0000_000000010100; //MOV
      32'd4: instruction <= 32'b1110_00_1_1101_0_0000_0001_101000000001; //MOV
      32'd8: instruction <= 32'b1110_00_1_1101_0_0000_0010_000100000011; //MOV
      32'd12: instruction <= 32'b1110_00_0_0100_1_0010_0011_000000000010; //ADDS
```

شکل 2- IF Modules

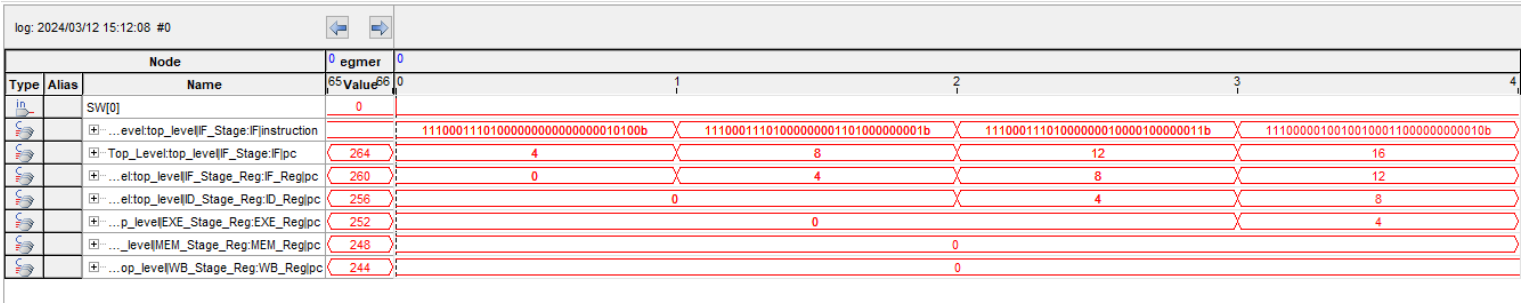
این ماژول را به همراه ماژول های خالی دیگر وارد TopLevel می‌کنیم:

Top_Level.v

```
module Top_Level(
    input clk, rst
);
    wire [31:0] pc_IF, pc_IF_Reg, pc_ID, pc_ID_Reg, pc_EXE, pc_EXE_Reg, pc_MEM, pc_MEM_Reg, pc_WB, pc_WB_Reg;
    wire [31:0] inst_IF, inst_IF_Reg;
    IF_Stage        IF(clk, rst, 1'b0, 1'b0, 32'b0, pc_IF, inst_IF);
    IF_Stage_Reg    IF_Reg(clk, rst, 1'b0, 1'b0, pc_IF, inst_IF, pc_IF_Reg, inst_IF_Reg);
    ID_Stage        ID(clk, rst, pc_IF_Reg, pc_ID);
    ID_Stage_Reg    ID_Reg(clk, rst, pc_ID, pc_ID_Reg);
    EXE_Stage       EXE(clk, rst, pc_ID_Reg, pc_EXE);
    EXE_Stage_Reg   EXE_Reg(clk, rst, pc_EXE, pc_EXE_Reg);
    MEM_Stage       MEM(clk, rst, pc_EXE_Reg, pc_MEM);
    MEM_Stage_Reg   MEM_Reg(clk, rst, pc_MEM, pc_MEM_Reg);
    WB_Stage        WB(clk, rst, pc_MEM_Reg, pc_WB);
    WB_Stage_Reg    WB_Reg(clk, rst, pc_WB, pc_WB_Reg);
endmodule
```

شکل 3- Top Level (IF)

با پیاده سازی این بخش در کوارتز مشاهده می‌شود که PC با گام های 4 تایی افزوده می‌شود و دستورات مختلف خوانده می‌شود:



شکل 4- Signal Tap (IF)

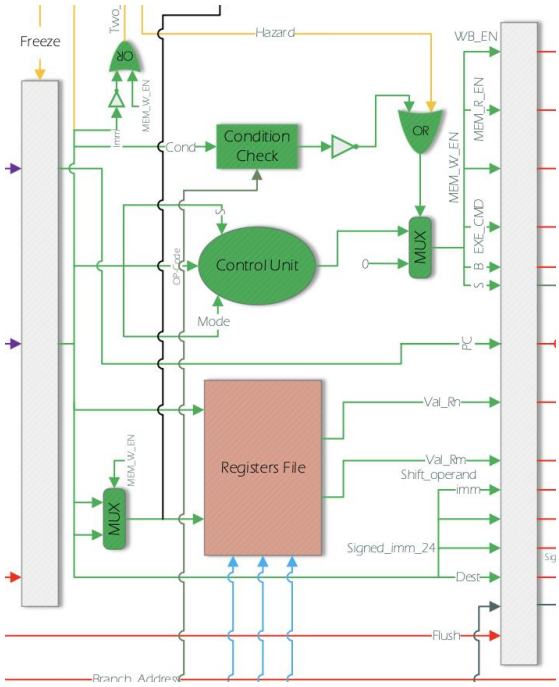
گزارش پیاده سازی:

Flow Summary	
Flow Status	Successful - Tue Mar 12 15:11:40 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Test
Top-level Entity Name	Test
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	4,582 / 33,216 (14 %)
Total combinational functions	1,658 / 33,216 (5 %)
Dedicated logic registers	4,282 / 33,216 (13 %)
Total registers	4282
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	28,928 / 483,840 (6 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 5- Flow Summary

استیج Instruction Decode

در این استیج باید دستوراتی که در مرحله قبل از ممووری خواندیم را decode کنیم.



شکل 6- ID

برای این کار ابتدا به پیاده‌سازی ماژول Register File می‌پردازیم. مقادیر اولیه رجیسترها را برابر شماره رجیسترها قرار می‌دهیم. در صورت یک بودن writeBackEn مقدار ورودی را در آدرس دریافت شده قرار می‌دهد و پیوسته خروجی را برابر آدرس‌های src1, src2 قرار می‌دهد:

```
module Register_File(
    input clk, rst,
    input[3:0] src1, src2, destWB,
    input[31:0] resultWB,
    input writeBackEn,
    output[31:0] reg1, reg2
);
    integer i=0;
    wire overIndex;
    reg[31:0] registerFile[14:0];

    assign overIndex = (src1==4'b1111) | (src2==4'b1111);
    assign reg1 = registerFile[overIndex? 4'b0:src1];
    assign reg2 = registerFile[overIndex? 4'b0:src2];

    initial begin
        for(i=0; i<15; i=i+1) begin
            registerFile[i] <= i;
        end
    end

    always @(negedge clk or posedge rst) begin
        if(rst) begin
            for(i=0; i<15; i=i+1) begin
                registerFile[i] <= i;
            end
        end
        else if(writeBackEn) begin
            registerFile[destWB] <= resultWB;
        end
    end
endmodule
```

شکل 7- Register File

حال به سراغ پیاده‌سازی Control Unit می‌رویم. این ماژول با گرفتن opCode, sIn , mode از instruction مشخص می‌کند که دستور از چه نوعی از لحاظ خواندن و نوشتن روی مموری و ... است و هم چنین مشخص می‌کند که چه دستوری باید به ALU استیج بعدی برود (ExecuteCommand) تا بر اساس آن ALU عمل مربوطه را انجام دهد. و خروجی دیگر مربوط به دستور Branch می‌باشد که مشخص میکند پرش داریم یا خیر که همگی در controlOut تجمیع شده اند.

```
module Control_Unit(
    input[1:0] mode,
    input[3:0] opCode,
    input sIn,
    output[8:0] controlOut
);
    reg[3:0] exeCmd_ID;
    reg memReadEn_ID, memWriteEn_ID, writeBackEn, b, sOut;

    always @(mode or opCode or sIn) begin
        {exeCmd_ID, memReadEn_ID, memWriteEn_ID, writeBackEn, b, sOut} = 9'b0;
        case(mode)
            2'b00: begin
                sOut <= sIn;
                case(opCode)
                    4'b1101: {exeCmd_ID, writeBackEn} <= {4'b0001, 1'b1}; //MOVE
                    4'b1111: {exeCmd_ID, writeBackEn} <= {4'b1001, 1'b1}; //MVN
                    4'b0100: {exeCmd_ID, writeBackEn} <= {4'b0010, 1'b1}; //ADD
                    4'b0101: {exeCmd_ID, writeBackEn} <= {4'b0011, 1'b1}; //ADC
                    4'b0010: {exeCmd_ID, writeBackEn} <= {4'b0100, 1'b1}; //SUB
                    4'b0110: {exeCmd_ID, writeBackEn} <= {4'b0101, 1'b1}; //SBC
                    4'b0000: {exeCmd_ID, writeBackEn} <= {4'b0110, 1'b1}; //AND
                    4'b1100: {exeCmd_ID, writeBackEn} <= {4'b0111, 1'b1}; //ORR
                    4'b0001: {exeCmd_ID, writeBackEn} <= {4'b1000, 1'b1}; //EOR
                    4'b1010: exeCmd_ID <= 4'b0100;                        //CMP
                    4'b1000: exeCmd_ID <= 4'b0110;                        //TST
                endcase
            end

            2'b01: begin
                exeCmd_ID <= 4'b0010;
                sOut <= 1'b1;
                case(sIn)
                    1'b1: {memReadEn_ID, writeBackEn} <= {1'b1, 1'b1}; //LDR
                    1'b0: memWriteEn_ID <= 1'b1;                       //STR
                endcase
            end

            2'b10:
                b <= 1'b1;

        endcase
    end
    assign controlOut = {exeCmd_ID, memReadEn_ID, memWriteEn_ID, writeBackEn, b, sOut};
endmodule
```

شکل 8- Control Unit

در ماژول Condition Check با دریافت بیت های condition از instruction و بیت‌های statusReg یک بیت خروجی out تولید می‌کنیم که با بیت hazard مشخص کننده‌ی این می‌شوند که سیگنال‌های کنترلی باید به استیج بعد بروند یا همه آنها صفر شوند.

```
module Condition_Check (
    input [3:0] condition,
    input [3:0] statusReg,
    output reg out
);
    wire N, Z, C, V;
    assign {N, Z, C, V} = statusReg;
    always @(condition, statusReg) begin
        case(condition)
            4'b0000: out = Z;
            4'b0001: out = ~Z;
            4'b0010: out = C;
            4'b0011: out = ~C;
            4'b0100: out = N;
            4'b0101: out = ~N;
            4'b0110: out = V;
            4'b0111: out = ~V;
            4'b1000: out = C && ~Z;
            4'b1001: out = ~C || Z;
            4'b1010: out = (N==V);
            4'b1011: out = ~(N==V);
            4'b1100: out = ~Z && (N==V);
            4'b1101: out = Z && ~(N==V);
            4'b1110: out = 1'b1;
            4'b1111: out = 1'b1;
        endcase
    end
endmodule
```

شکل 9- Condition Check

ماژول‌های معرفی شده در بالا را در ID نمونه گیری میکنیم:

```
module ID_Stage(
    input clk, rst,
    //from IF stage
    input[31:0] instruction,
    //from WB stage
    input[31:0] resultWB,
    input[3:0] destWB,
    input writeBackEnWB,
    //from HazardDetect module
    input hazard,
    //from Status register
    input[3:0] statusReg,
    //to ID stage register
    output[31:0] valRn, valRm,
    output[23:0] signedImm24,
    output[11:0] shiftOperand,
    output[3:0] exeCmd, destID,
    output imm, writeBackEnID, memReadEn, memWriteEn, b, s,
    //to HazardDetect module
    output[3:0] src1, src2,
    output twoSrc
);
    wire[8:0] controlOut;
    wire conditionOut;
    assign signedImm24 = instruction[23:0];
    assign shiftOperand = instruction[11:0];
    assign destID = instruction[15:12];
    assign imm = instruction[25];
    assign src1 = instruction[19:16];
    assign twoSrc = ~imm || memWriteEn;

    Mux_4b mux_4b(memWriteEn, instruction[3:0], destID, src2);
    Register_File register_file(clk, rst, src1, src2, destWB, resultWB, writeBackEnWB, valRn, valRm);
    Control_Unit control_unit(instruction[27:26], instruction[24:21], instruction[20], controlOut);
    Condition_Check condition_check(instruction[31:28], statusReg, conditionOut);
    Mux_9b mux_9b(~conditionOut || hazard, controlOut, 9'b0, {exeCmd, memReadEn, memWriteEn, writeBackEnID, b, s});
endmodule
```

شکل 10- ID Stage

رجیستر ID:

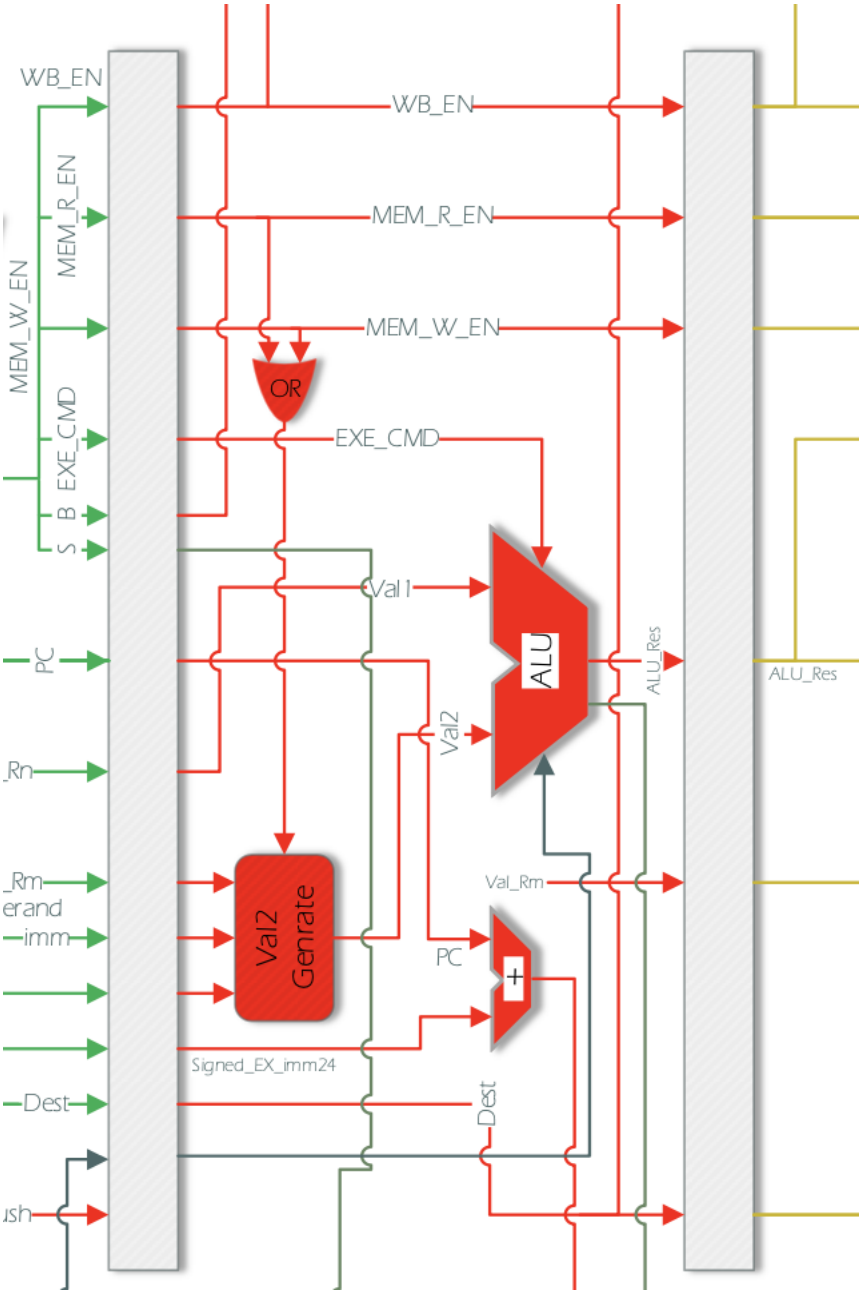
```
module ID_Stage_Reg(
    input clk, rst, flush,
    input writeBackEnIn, memReadEnIn, memWriteEnIn, bIn, sIn,
    input[3:0] exeCmdIn,
    input[31:0] pcIn, valRnIn, valRmIn,
    input immIn,
    input[11:0] shiftOperandIn,
    input[23:0] signedImm24In,
    input[3:0] destIn,
    input[3:0] statusRegIn,

    output reg writeBackEn, memReadEn, memWriteEn, b, s,
    output reg[3:0] exeCmd,
    output reg[31:0] pc, valRn, valRm,
    output reg imm,
    output reg[11:0] shiftOperand,
    output reg[23:0] signedImm24,
    output reg[3:0] dest,
    output reg[3:0] statusReg
);
    always @(posedge clk or posedge rst) begin
        if(rst || flush) begin
            {writeBackEn, memReadEn, memWriteEn, b, s, imm} <= 6'b0;
            {exeCmd, dest, statusReg} <= 12'b0;
            shiftOperand <= 12'b0;
            signedImm24 <= 24'b0;
            {pc, valRn, valRm} <= 96'b0;
        end
        else begin
            {writeBackEn, memReadEn, memWriteEn, b, s, imm} <= {writeBackEnIn, memReadEnIn, memWriteEnIn, bIn, sIn, immIn};
            {exeCmd, dest, statusReg} <= {exeCmdIn, destIn, statusRegIn};
            shiftOperand <= shiftOperandIn;
            signedImm24 <= signedImm24In;
            {pc, valRn, valRm} <= {pcIn, valRnIn, valRmIn};
        end
    end
endmodule
```

شکل 11 - ID Register

استیج Execution

در این استیج قرار است محاسبات دستورات مختلف انجام شود.



شکل 12- Execution

در ابتدا به سراغ درست کردن ماژول ALU میرویم.

```
module ALU(···
);
    reg V, C;
    wire N, Z;

    always @(aluCmd, in1, in2, c) begin
        {V, C, resultALU} = 34'b0;
        case(aluCmd)
            4'b0001: resultALU = in2;
            4'b1001: resultALU = ~in2;
            4'b0010: begin
                {C, resultALU} = in1 + in2;
                V = (resultALU[31] != in1[31]) && (in1[31]==in2[31]);
            end
            4'b0011: begin
                {C, resultALU} = in1 + in2 + c;
                V = (resultALU[31] != in1[31]) && (in1[31]==in2[31]);
            end
            4'b0100: begin
                {C, resultALU} = in1 - in2;
                V = (resultALU[31] != in1[31]) && (in1[31]==~in2[31]);
            end
            4'b0101: begin
                {C, resultALU} = in1 - in2 - {31'b0, ~c};
                V = (resultALU[31] != in1[31]) && (in1[31]==~in2[31]);
            end
            4'b0110: resultALU = in1 & in2;
            4'b0111: resultALU = in1 | in2;
            4'b1000: resultALU = in1 ^ in2;
            4'b0110: resultALU = in1 & in2;
        endcase
    end

    assign N = resultALU[31];
    assign Z = (resultALU == 32'b0);
    assign statusBits = {N, Z, C, V};

endmodule
```

شکل 13- ALU

در این ماژول با توجه به aluCmd که از بخش ID دریافت می‌کنیم. اقدام به انجام یک عمل می‌کند که نوع عمل در سوییچ کیس کد بالا مشخص شده است و بعد از انجام عمل خروجی مربوطه تولید می‌شود. همچنین با توجه به نوع عملیات و عملوندها و نتیجه عملیات، بیت‌های N,V,C,Z مقدار دهی می‌شوند و پس از تجمیع در statusBits به Status Register فرستاده می‌شود.

در گام بعد ماژول ValGenerator را پیاده‌سازی می‌کنیم. این ماژول برای مشخص کردن وردی دوم ALU استفاده می‌شود. اگر دستور از نوع 32bit immediate باشد با توجه به shiftOperand مقدار imm مشخص می‌شود. اگر دستور imm باشد مقدار valRm را شیفت می‌دهیم.

```
module Val_Generator(
    input [31:0] valRm,
    input imm, isMem,
    input [11:0] shiftOperand,
    output reg[31:0] valOut
);
    integer shiftValue=0;
    always@(valRm, imm, shiftOperand, isMem)
    begin
        valOut = 32'b0;
        if(isMem) begin
            valOut = shiftOperand;
        end
        else if(imm) begin
            shiftValue = (shiftOperand[11:8])*2;
            valOut = (shiftOperand[7:0]<<(32-shiftValue)) | (shiftOperand[7:0]>>(shiftValue));
        end
        else if(~shiftOperand[4]) begin
            shiftValue = (shiftOperand[11:7]);
            case(shiftOperand[6:5])
                2'b00: valOut = valRm << shiftValue;
                2'b01: valOut = valRm >> shiftValue;
                2'b10: valOut = valRm >>> shiftValue;
                2'b11: valOut = (valRm >> shiftValue) | (valRm << (32-shiftValue));
            endcase
        end
    end

endmodule
```

شکل 14- Val Generator

ماژول ها را به هم متصل میکنیم و در EXE قرار می‌دهیم:

```
module EXE_Stage(
    //from ID stage
    input clk, rst,
    input[3:0] exeCmd,
    input memReadEn, memWriteEn,
    input[31:0] pc,
    input[31:0] valRn, valRm,
    input imm,
    input[11:0] shiftOperand,
    input[23:0] signedImm24,
    input[3:0] statusRegID,
    //to MEM stage
    output [31:0]resultALU, branchAddress,
    //to Status register
    output [3:0] statusRegEXE
);
    wire[31:0] val1, val2;
    wire C, isMem;

    assign val1 = valRn;
    assign C = statusRegID[3];
    assign isMem = memReadEn || memWriteEn;

    Val_Generator val_generator(valRm, imm, isMem, shiftOperand, val2);
    Branch_Adder branch_adder(pc, signedImm24, branchAddress);
    ALU alu(val1, val2, C, exeCmd, resultALU, statusRegEXE);
endmodule
```

شکل 15- EXE Stage

رجیستر EXE:

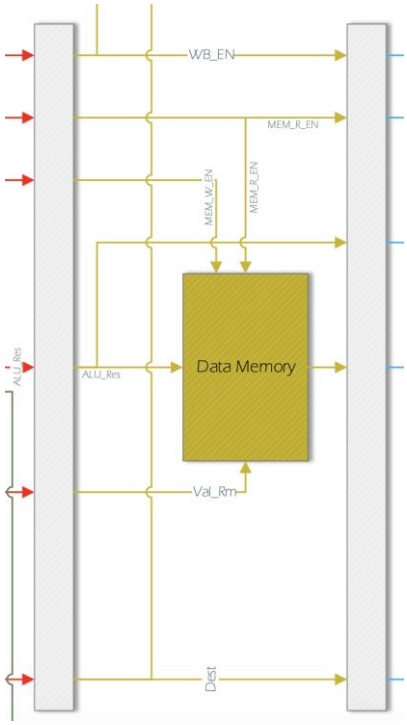
```
module EXE_Stage_Reg(
    input clk, rst,
    input writeBackEnIn, memReadEnIn, memWriteEnIn,
    input[31:0] resultALUIn, storeValIn,
    input[3:0] destIn,

    output reg writeBackEn, memReadEn, memWriteEn,
    output reg[31:0] resultALU, storeVal,
    output reg[3:0] dest
);
    always @(posedge clk or posedge rst) begin
        if(rst) begin
            {writeBackEn, memReadEn, memWriteEn} <= 3'b0;
            {resultALU, storeVal} <= 64'b0;
            dest <= 4'b0;
        end
        else begin
            {writeBackEn, memReadEn, memWriteEn} <= {writeBackEnIn, memReadEnIn, memWriteEnIn};
            {resultALU, storeVal} <= {resultALUIn, storeValIn};
            dest <= destIn;
        end
    end
endmodule
```

شکل 16- EXE Register

استیج Memory:

اکنون بخش حافظه را به پردازنده اضافه می‌کنیم:



شکل 17- Memory

مموری شامل ۶۴ خانه ۳۲ بیتی می‌باشد که با توجه به سیگنال‌های ورودی به آن عملیات خواندن از یا نوشتن روی مموری انجام می‌شود. ۱۰۲۴ تا از ادرسی که به عنوان ورودی گرفتیم کم میکنیم و دو بار به شمت راست شیفت میدهیم تا ادرسی که داده مورد نظر در حافظه دارد مشخص شود.

```
module MEM_Stage(
    input clk, rst, memRead, memWrite,
    input [31:0] address, data,
    output [31:0] memResult
);
    integer i=0;
    reg[31:0] memory[0:63];
    wire[31:0] temp;

    initial begin
        for(i=0; i<64; i=i+1) begin
                memory[i] <= 0;
        end
    end
    assign memResult = memRead ? memory[(address-32'd1024)>>2]: 32'b0;
    always @(posedge clk) begin
        if(memWrite) memory[(address-1024)>>2] <= data;
    end
endmodule
```

شکل 18- MEM Stage

```
module MEM_Stage_Reg(
    input clk, rst, wbEnIn, memReadEnIn,
    input [31:0] aluResultIn, memReadValueIn,
    input [3:0] dstIn,
    output reg wbEn, memReadEn,
    output reg [31:0] aluResult, memReadValue,
    output reg [3:0] dst
);

    always @(posedge clk or posedge rst) begin
        if(rst) begin
            {wbEn, memReadEn, aluResult, memReadValue, dst} = 70'b0;
        end
        else begin
            {wbEn, memReadEn, aluResult, memReadValue, dst} <= {wbEnIn, memReadEnIn, aluResultIn, memReadValueIn, dstIn};
        end
    end

endmodule
```

شکل19 - MEM Register

استیج Write Back:

این استیج شامل یک mux است که با توجه به مقدار memReadEn مشخص می‌شود که کدام دیتا را باید برای نوشته شده در رجیستر فایل ارسال کنیم.

```
module WB_Stage(
    input clk, rst,
    input [3:0] dstIn,
    input [31:0] memResult, aluResult,
    input memReadEn, wbEnIn,
    output [3:0] wbDst,
    output [31:0] wbValue,
    output wbEn
);
    assign wbDst = dstIn;
    assign wbEn = wbEnIn;
    assign wbValue = memReadEn ? memResult : aluResult;
endmodule
```

شکل 20- WB Stage

ثبات وضعیت (Status Register):

این رجیستر، وضعیت را در لبه پایین رونده clk و در صورت ۱ بودن S اپدیت میکند. بیت‌های انتقال پیدا کرده شامل فلگ های zero, negative, overflow, carry می‌باشند.

```
module Status_Reg(
    input clk, rst,
    input s,
    input [3:0] statusIn,

    output reg[3:0] status
);

    always @(negedge clk or posedge rst) begin
        if(rst) begin
            status = 4'b0;
        end
        else if(s) begin
            status = statusIn;
        end
    end
endmodule
```

شکل21 - Status Register

واحد تشخیص مخاطره (Hazard Detection Unit)

این ماژول وابستگی داده‌ای را مشخص می کند. در این ماژول سعی داریم مخاطره خواندن پس از نوشتن را تشخیص دهیم و در صورت مشاهده آن، Pipe را stall کنیم. در دو دستور متوالی برای مثال اگر دستور دوم به نتیجه دستور اول نیاز داشته باشد باید تا اتمتم کامل دستور اول صبر کنیم.

```
module Hazard_Unit(
    input [3:0] src1, src2,
    input twoSrc,
    input wbEn_EXE, wbEn_MEM,
    input [3:0] wbDst_EXE, wbDst_MEM,
    output freeze
);
    wire cond1, cond2, cond3 cond4;

    assign cond1 = wbEn_EXE && (src1==wbDst_EXE);
    assign cond2=  wbEn_EXE && twoSrc && (src2==wbDst_EXE);
    assign cond3 = wbEn_MEM && (src1==wbDst_MEM);
    assign cond4 = wbEn_MEM && twoSrc && (src2==wbDst_MEM);
    assign freeze = cond1 || cond2 || cond3 || cond4;

endmodule
```

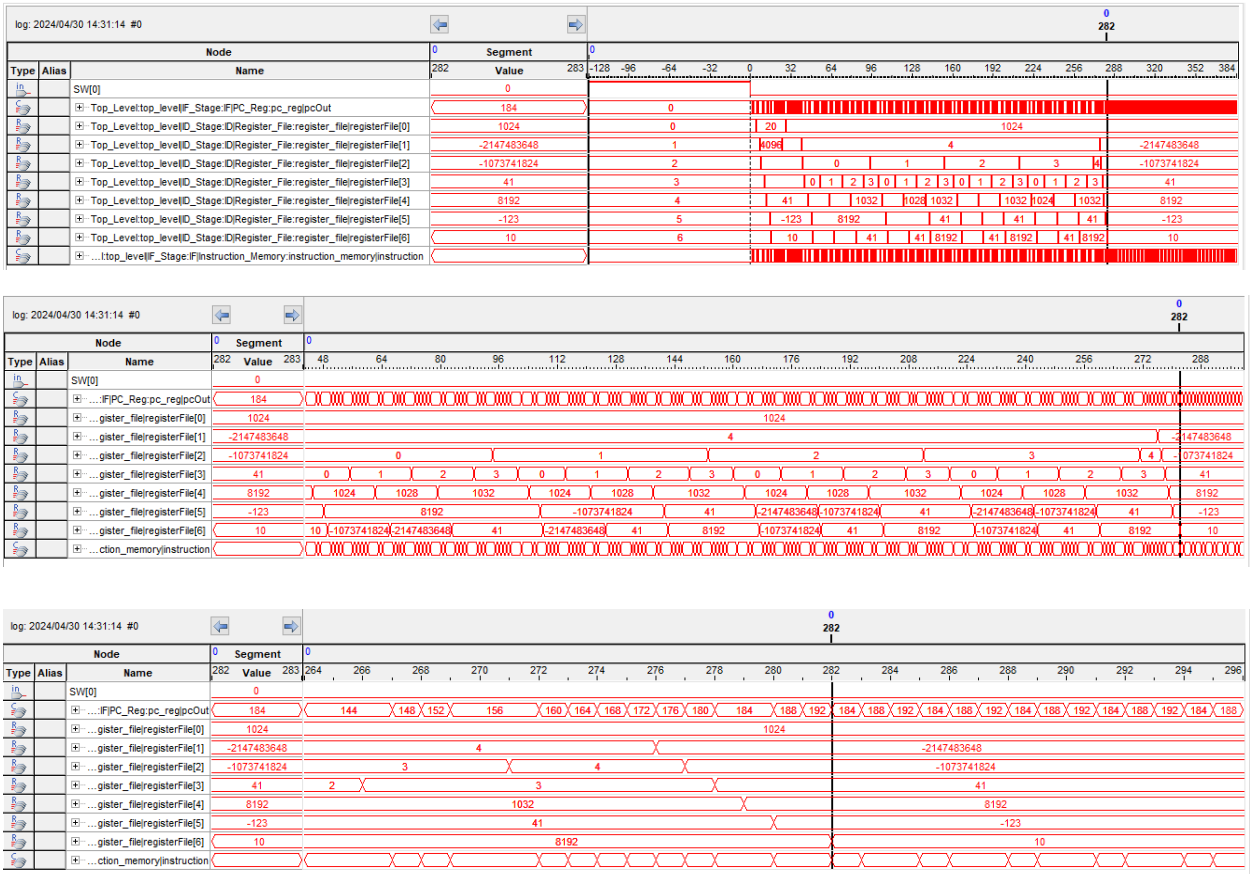
شکل 22- Hazard Detection Unit

تست پردازنده ARM:

تمام قسمت‌های پردازنده را به هم متصل میکنیم و آن را در کوارتز پیاده سازی می‌کنیم:

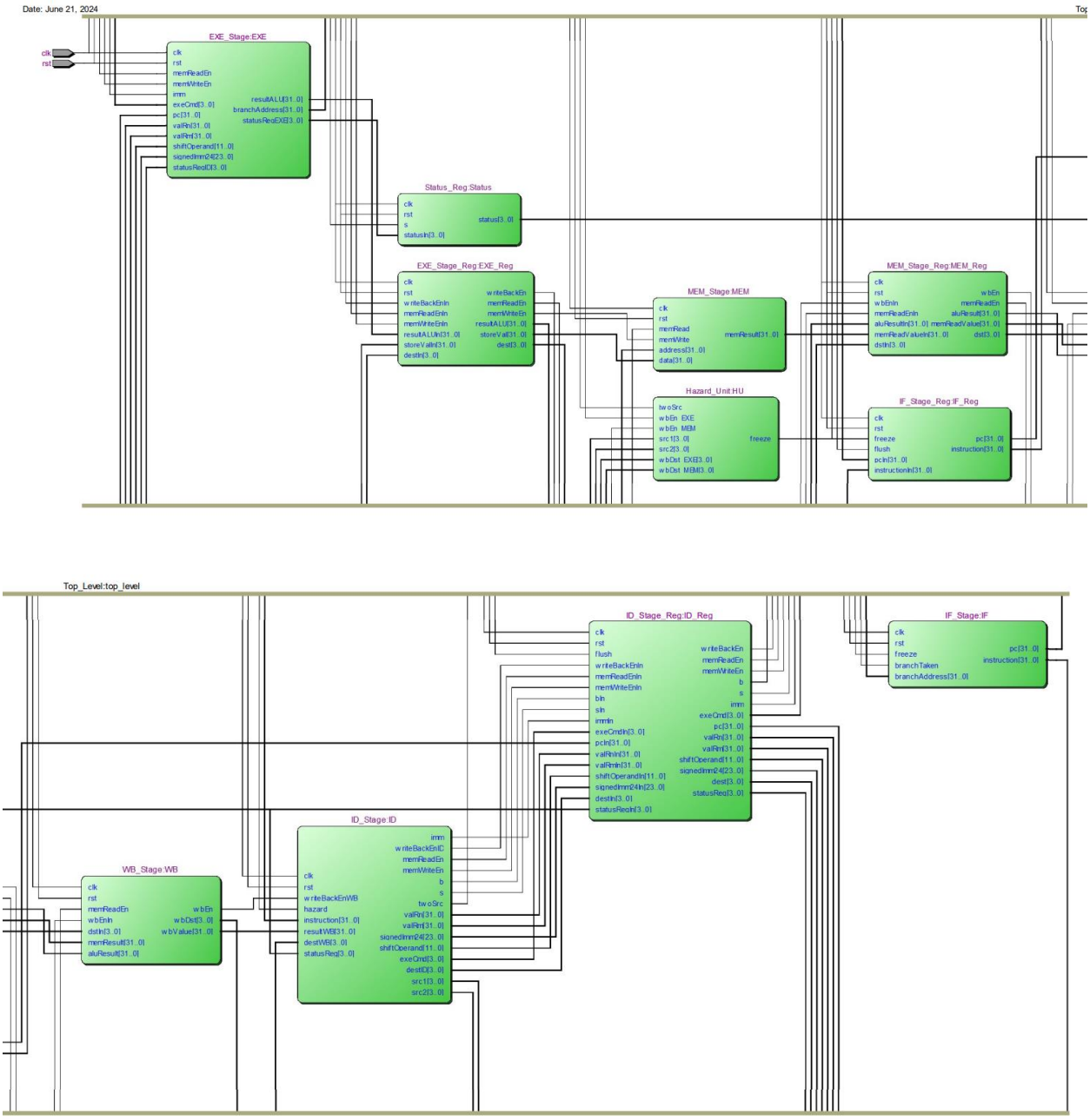
Flow Summary	
Flow Status	Successful - Tue Apr 30 14:30:51 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Test
Top-level Entity Name	Test
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7,878 / 33,216 (24 %)
Total combinational functions	4,152 / 33,216 (13 %)
Dedicated logic registers	6,193 / 33,216 (19 %)
Total registers	6193
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	299,008 / 483,840 (62 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 23- ARM Flow Summary



شکل 24- تصاویری از Signal Tap از اجرای ARM

همانطور که مشاهده می‌شود در 282 کلاک دستورات اجرا شده‌اند:



شکل 25- ARM RTL

بخش forwarding

در بخش قبل ساختار پردازنده ARM را تکمیل کردیم. با توجه به ساختار pipeline، وجود data dependency های زیاد سبب می شد در بسیاری از مراحل واحد hazard detection پایپ لاین را متوقف کند. برای جلوگیری از این کار، باید دیتای مورد نظر را زودتر به واحد exe برسانیم تا hazard به وجود نیاید و از داده ی بروز استفاده شود. دو حالت وجود دارد که باعث ایجاد hazard و در نتیجه متوقف شدن پایپ لاین می شود:

- دستور قبلی در مرحله memory و دستور حال حاضر در مرحله exe باشد.
- دستور قبلی در مرحله Write Back و دستور حال حاضر در مرحله exe باشد.

توجه شود که در هر دو حالت باید data dependency وجود داشته باشد یعنی دستوری که در پایپ لاین جلوتر است روی یکی از دیتاهای دستوری که در exe است ، تاثیر بگذارد.

برای رفع این مشکل، واحد Forwarding Unit را به مدار اضافه می کنیم. یک ورودی forwardingEn هم اضافه می کنیم که روشن یا خاموش بودن این حالت را کنترل کنیم. ساختار این component به صورت زیر است:

```
module Forwarding_Unit(
    input forwardingEn,
    input [3:0] src1, src2,
    input [3:0] wbDst_MEM, wbDst_WB,
    input wbEn_MEM, wbEn_WB,
    output [1:0] selSrc1, selSrc2
);
    assign selSrc1 = forwardingEn ?
        ((wbEn_MEM && src1 == wbDst_MEM) ? 2'd1 : (wbEn_WB && src1 == wbDst_WB) ? 2'd2 : 2'd0)
        : 2'd0;
    assign selSrc2 = forwardingEn ?
        ((wbEn_MEM && src2 == wbDst_MEM) ? 2'd1 : (wbEn_WB && src2 == wbDst_WB) ? 2'd2 : 2'd0)
        : 2'd0;
endmodule
```

شکل 26- Forwarding Unit

ورودی های دوم و سوم بیانگر ورودی هایی هستند که از بخش decode در مرحله ID به دست آمده اند. ورودی wbEn_MEM نشان می دهد آیا دستور مورد نظر در مرحله memory قرار است رجیستری را در رجیسترفایل بروزرسانی کند؟ همچنین ورودی wbEn_WB نشان می دهد آیا دستوری که در حال حاضر در مرحله write back قرار دارد رجیستر فایل را

بروزرسانی می کند؟ خروجی های selSrc2 و selSrc1 نشان دهنده ورودی کنترل کننده مالتی پلکسر های ورودی های واحد ALU هستند که دیتای ورودی به این واحد را کنترل می کنند.

حالت ۰۰ برای هر دو مالتی پلکسر حالت پیش فرض است. در صورتی که ورودی اول با مقصد دستور مورد نظر در مرحله memory برابر باشد و wbEn_MEM فعال باشد، یعنی دستوری که در مرحله memory است قرار است رجیستر مورد نظر را آپدیت کند و در صورت فعال بودن forwardingEn ، مقدار به روز شده را به جای خروجی مرحله ID وارد ALU می کنیم. در صورتی که ورودی اول با مقصد دستور مورد نظر در مرحله Write Back برابر باشد و wbEn_WB فعال باشد، یعنی دستوری که در مرحله WB است قرار است رجیستر مورد استفاده در دستوری که در مرحله exe است را آپدیت کند که در صورت فعال بودن forwardingEn ، مقدار به روز شده را به جای خروجی مرحله ID وارد ALU می کنیم. همین حالت برای ورودی دوم ALU هم صادق است.

علاوه بر تغییرات بالا، واحد Hazard Detection Unit هم دچار تغییر می شود. با در نظر گرفتن اینکه واحد forwarding برخی از hazard ها را برطرف می کند، نیاز به بررسی hazard های کمتری داریم.

```
module Hazard_Unit(
    input forwardingEn,
    input [3:0] src1, src2,
    input twoSrc,
    input wbEn_EXE, wbEn_MEM, memReadEn_EXE,
    input [3:0] wbDst_EXE, wbDst_MEM,
    output freeze
);
    wire cond1, cond2, cond3, cond4, cond5, cond6;

    assign cond1 = wbEn_EXE && (src1==wbDst_EXE);
    assign cond2 = wbEn_EXE && twoSrc && (src2==wbDst_EXE);
    assign cond3 = wbEn_MEM && (src1==wbDst_MEM);
    assign cond4 = wbEn_MEM && twoSrc && (src2==wbDst_MEM);

    assign cond5 = memReadEn_EXE && (src1==wbDst_EXE);
    assign cond6 = memReadEn_EXE && twoSrc && (src2==wbDst_EXE);

    assign freeze = forwardingEn ? (cond5 || cond6) : (cond1 || cond2 || cond3 || cond4);

endmodule
```

شکل 27- Hazard Detection Unit تغییر یافته

یک سوئیچ به reset و و یک سوئیچ به forwardingEn وصل شده است.

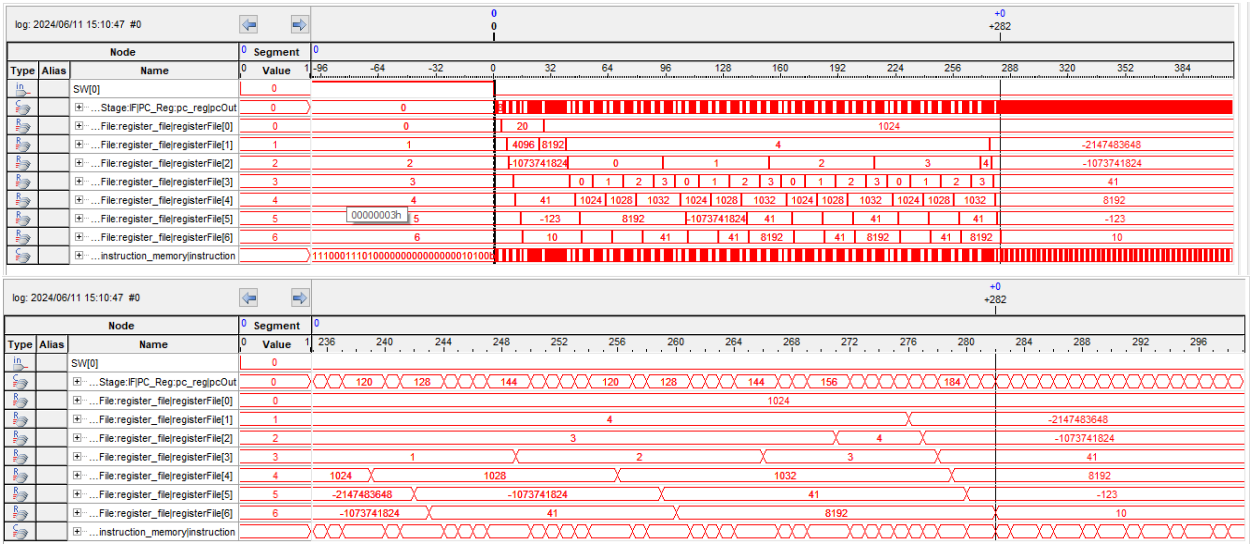
Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Flow Messages
- Flow Suppressed Messages
- Assembler
- TimeQuest Timing Analyzer

Flow Summary

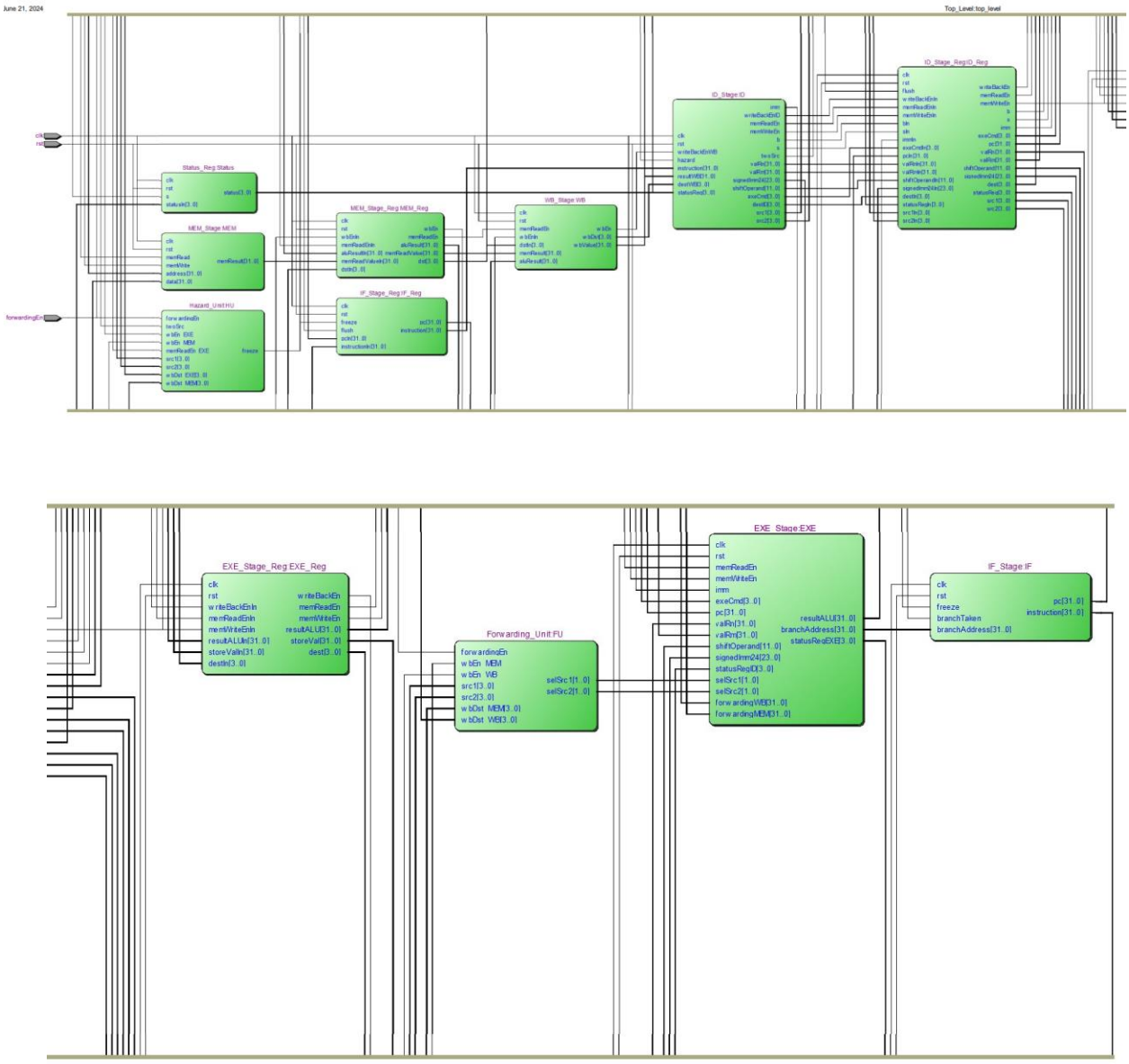
Flow Status	Successful - Tue Jun 11 14:56:30 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Test
Top-level Entity Name	Test
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	8,067 / 33,216 (24 %)
Total combinational functions	4,341 / 33,216 (13 %)
Dedicated logic registers	6,198 / 33,216 (19 %)
Total registers	6198
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	299,008 / 483,840 (62 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 28- Forwarding Flow Summary



شکل 29- نتیجه Signal Tap در حالت خاموش بودن سوئیچ ForwardingEn

شکل 30- نتیجه Signal Tap در حالت روشن بودن سوئیچ ForwardingEn



شکل 31- Forwarding RTL

همانطور که مشاهده می‌شود در 194 کلاک دستورات اجرا شده‌اند:

افزایش کارایی:

$$\frac{282 - 194}{282} = 45.3\%$$

افزایش هزینه:

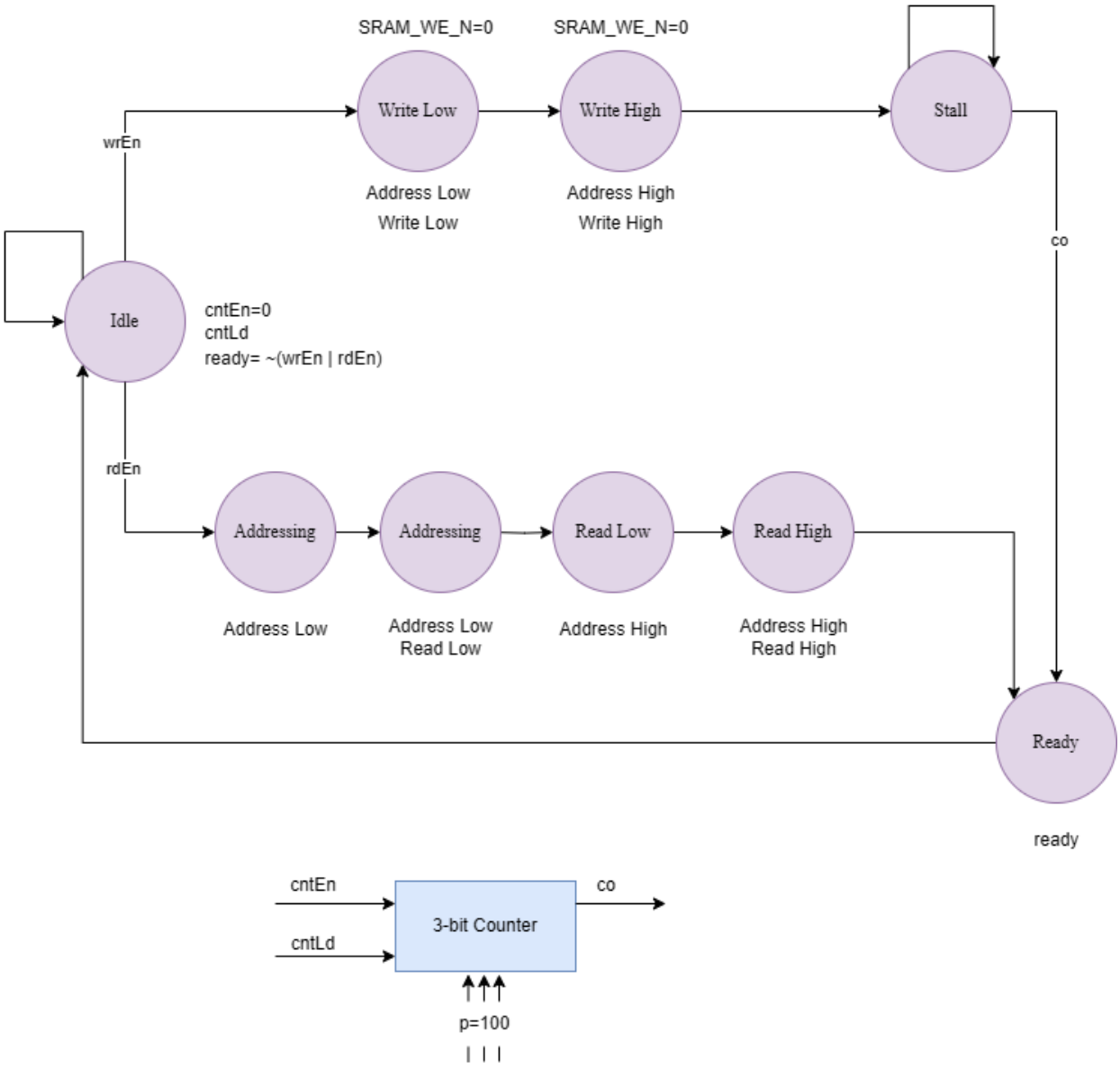
$$\frac{8067 - 7878}{7878} = 2.4\%$$

میزان افزایش کارایی بر حسب هزینه:

$$\frac{45.3}{2.4} = 18.88$$

بخش SRAM

در این قسمت به طراحی بخش SRAM می پردازیم. از آنجایی که تعداد Logical Element در FPGA محدود است در بورد ها بخشی را به واحد حافظه اختصاص داده اند. برای دسترسی به این حافظه، باید کنترلری طراحی کنیم که بتواند به درستی اطلاعات ذخیره شده روی SRAM را بخواند و به درستی اطلاعات را در آن ذخیره کند. شکل زیر واحد کنترلر یک SRAM را نشان میدهد:



شکل32 - SRAM Controller State Machine Diagram

```
module SRAM_Controller (…
);
    reg [3:0] ps, ns;
    reg cntEn, cntLd;
    reg [15:0] dataLow, dataHigh;
    wire [16:0] sramAddress;
    wire co;

    Counter_3b cnt3b(clk, rst, cntEn, cntLd, 3'b100, co);

    assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'b0;

    assign sramAddress = ((address-1024))>>2;

    always @(ps, wrEn, rdEn, co) begin
        ns = `IDLE;
        case(ps)
            `IDLE :         ns = wrEn ? `WRITE_LOW : rdEn ? `ADDR_LOW : `IDLE;
            `WRITE_LOW :    ns = `WRITE_HIGH;
            `WRITE_HIGH :   ns = `STALL;
            `ADDR_LOW :     ns = `READ_LOW;
            `READ_LOW :     ns = `ADDR_HIGH;
            `ADDR_HIGH :    ns = `READ_HIGH;
            `READ_HIGH :    ns = `READY;
            `STALL :        ns = co ? `READY : `STALL;
            `READY :        ns = `IDLE;
        endcase
    end

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

```
    always @(ps, wrEn, rdEn) begin
        ready = 1'b0;
        cntEn = 1'b1;
        cntLd = 1'b0;
        SRAM_ADDR = 18'b0;
        SRAM_WE_N = 1'b1;
        case(ps)
            `IDLE :         begin ready=~(wrEn|rdEn); cntEn=1'b0; cntLd=1'b1; end
            `WRITE_LOW :    begin SRAM_WE_N=1'b0; SRAM_ADDR={sramAddress, 1'b0}; end
            `WRITE_HIGH :   begin SRAM_WE_N=1'b0; SRAM_ADDR={sramAddress, 1'b1}; end
            `ADDR_LOW :     begin SRAM_ADDR={sramAddress, 1'b0}; end
            `READ_LOW :     begin SRAM_ADDR={sramAddress, 1'b0}; dataLow=SRAM_DQ; end
            `ADDR_HIGH :    begin SRAM_ADDR={sramAddress, 1'b1}; end
            `READ_HIGH :    begin SRAM_ADDR={sramAddress, 1'b1}; dataHigh=SRAM_DQ; end
            `READY :        begin ready=1; end
        endcase
    end

    assign SRAM_DQ = ps==`WRITE_LOW ? writeData[15:0] : ps==`WRITE_HIGH ? writeData[31:16] : 16'bz;
    assign readData = {dataHigh, dataLow};

endmodule
```

شکل 33- ماژول کنترلر SRAM

در قسمت بعد، واحد کنترلر را به جای حافظه قرار می دهیم و طراحی را روی بورد تست می کنیم:

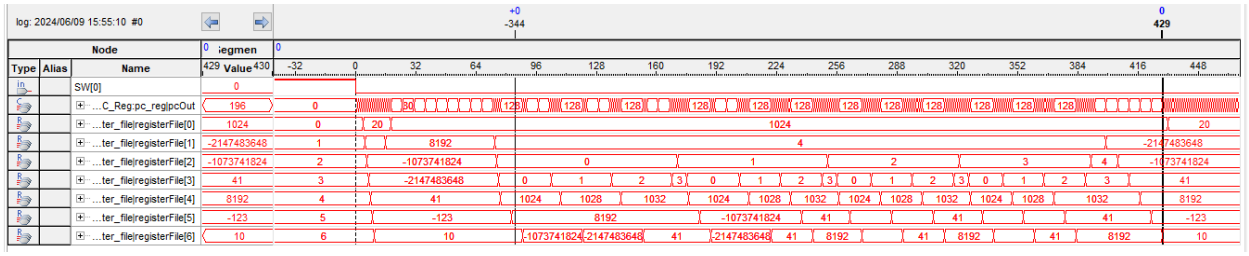
Table of Contents

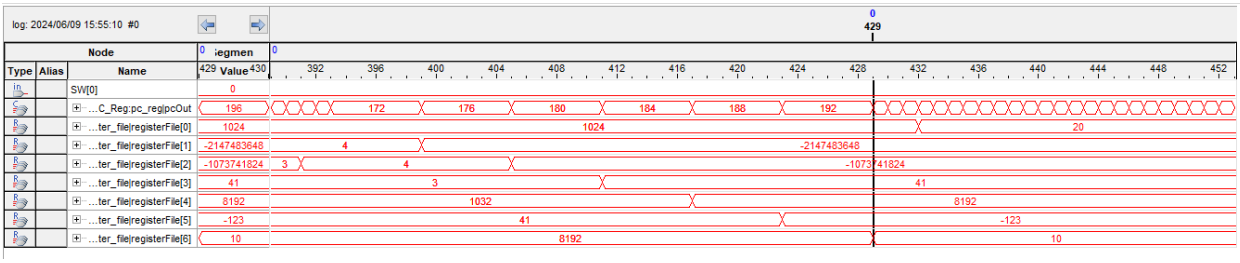
- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Flow Messages
- Flow Suppressed Messages
- Assembler
- TimeQuest Timing Analyzer

Flow Summary

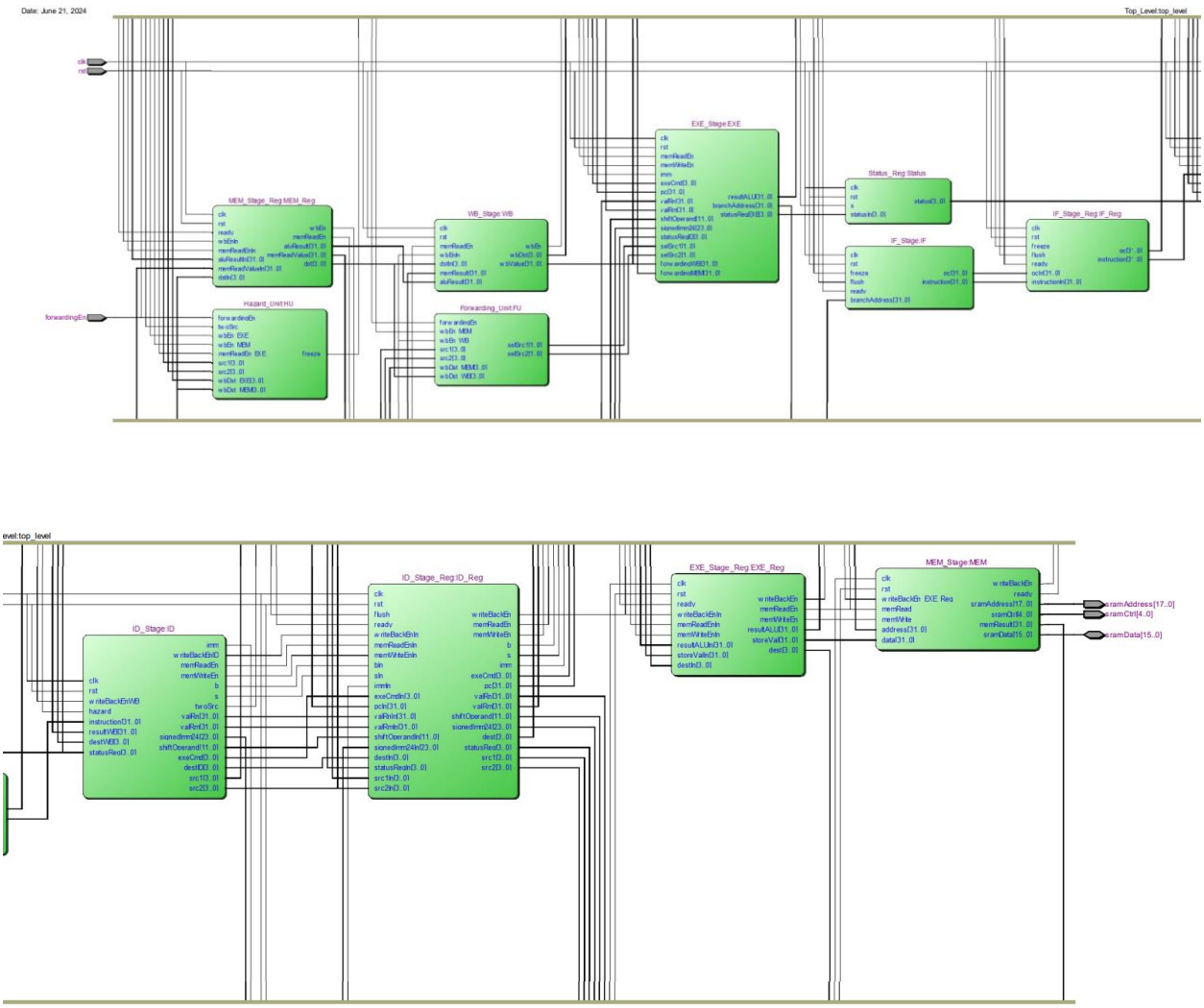
Flow Status	Successful - Sun Jun 09 16:47:52 2024
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	Test
Top-level Entity Name	Test
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	7,465 / 33,216 (22 %)
Total combinational functions	4,175 / 33,216 (13 %)
Dedicated logic registers	5,593 / 33,216 (17 %)
Total registers	5593
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	264,192 / 483,840 (55 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

شکل 34- SRAM Flow Summary





شکل 35- SRAM Signal Tap Results



شکل 36- SRAM RTL

همانطور که مشاهده می‌شود در 429 کلاک دستورات اجرا شده‌اند:

کاهش کارایی:

$$\frac{429 - 194}{429} = 54.7\%$$

کاهش هزینه:

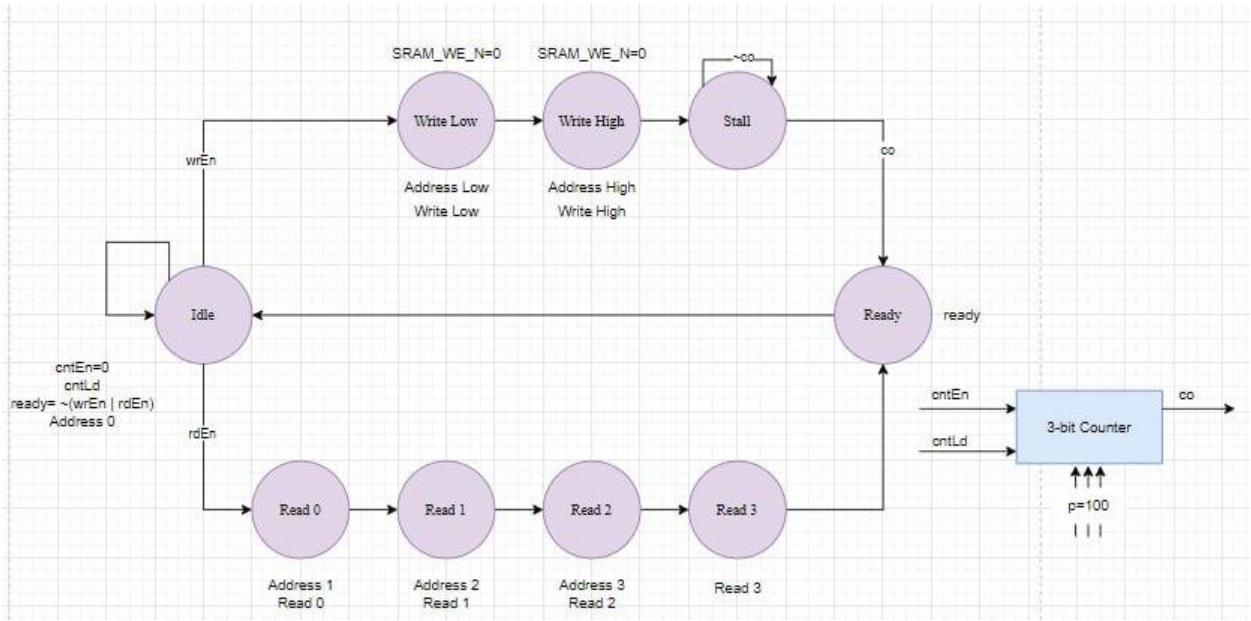
$$\frac{7878 - 7465}{7465} = 5.5\%$$

میزان کاهش کارایی بر حسب هزینه:

$$\frac{54.7}{5.5} = 9.94$$

بخش Cache

بر اساس توضیحات ویدیو، از Cache ۲ طرفه و مبنای least recently used استفاده می کنیم. هر ردیف حافظه ۶۴ بیت است که معادل دو داده است. کنترلر SRAM برای پیاده سازی این بخش دستخوش تغییراتی شده است تا بتواند در ۶ کلاک، دو کلمه ۳۲ بیتی از حافظه بخواند.



شکل 37- SRAM Controller adapted for Cache

```
module Cache_Controller( ···
);

    // ADDRESS DECODE
    wire word = address[2];
    wire [5:0] index = address[8:3];
    wire [9:0] tag = address[18:9];

    // CACHE MEMORY
    reg [63:0] dataWay0 [0:63];
    reg [63:0] dataWay1 [0:63];
    reg [9:0] tagWay0 [0:63];
    reg [9:0] tagWay1 [0:63];
    reg [0:63] validWay0;
    reg [0:63] validWay1;
    reg [0:63] LRU;

    // COMBINATIONAL LOGIC
    initial begin
        validWay0 = 64'b0;
        validWay1 = 64'b0;
        LRU = 64'b0;
    end

    wire hit, hitWay0, hitWay1;
    assign hitWay0 = (tagWay0[index] == tag) & validWay0[index];
    assign hitWay1 = (tagWay1[index] == tag) & validWay1[index];
    assign hit = hitWay0 | hitWay1;
    assign ready = ~(wrEn | rdEn) | (wrEn & sramReady) | (rdEn & (hit | sramReady));

    wire [63:0] readBlock;
    assign readBlock = hitWay0 ? (dataWay0[index]) : (hitWay1 ? dataWay1[index] : sramReadData);
    assign readData = word ? readBlock[63:32] : readBlock[31:0] ;
```

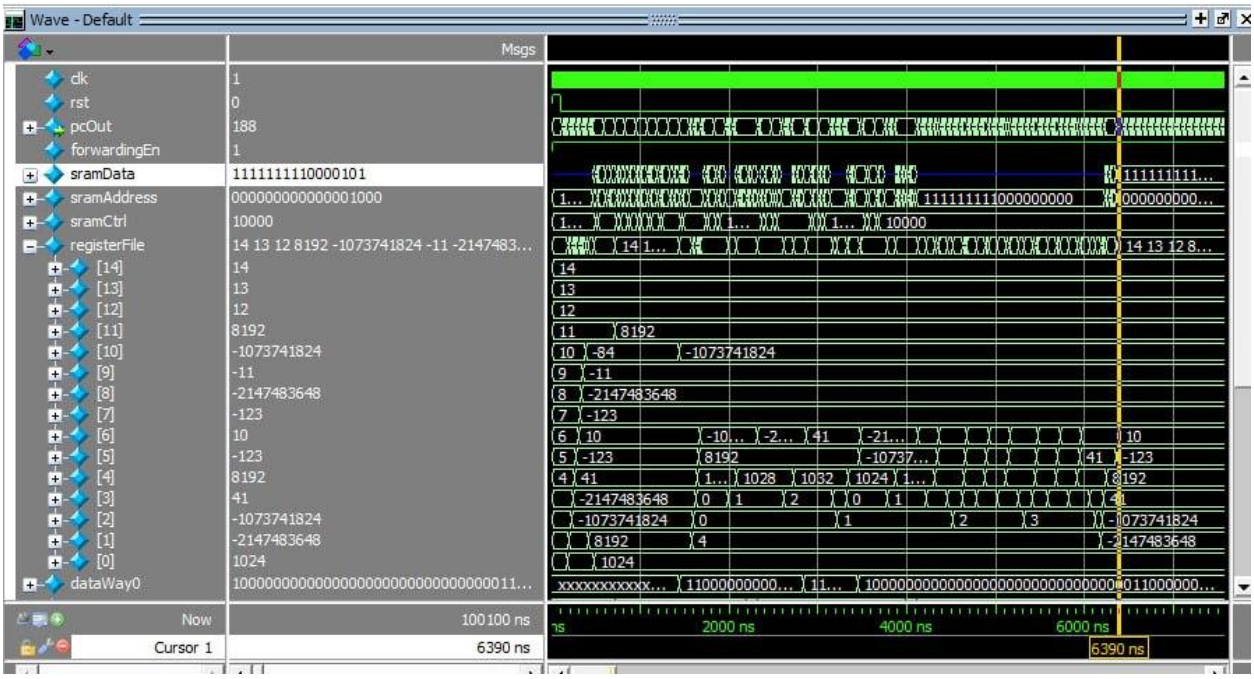
```
    assign sramRdEn = rdEn & ~hit;
    assign sramWrEn = wrEn;
    assign sramAddress = address;
    assign sramWriteData = writeData;

    always @(posedge clk) begin
        if(rst) begin
            validWay0 = 64'b0;
            validWay1 = 64'b0;
            LRU = 64'b0;
        end
        else if(wrEn & hit) begin
            validWay0[index] <= hitWay0 ? 1'b0 : validWay0[index];
            validWay1[index] <= hitWay1 ? 1'b0 : validWay1[index];
        end
        else if(rdEn & hit) begin
            LRU[index] <= hitWay0 ? 1'b0 : 1'b1;
        end
        else if(rdEn & sramReady & ~hit) begin
            if(LRU[index]==1'b0) begin
                dataWay1[index] <= sramReadData;
                tagWay1[index] <= tag;
                validWay1[index] <= 1'b1;
                LRU[index] <= 1'b1;
            end
            else begin
                dataWay0[index] <= sramReadData;
                tagWay0[index] <= tag;
                validWay0[index] <= 1'b1;
                LRU[index] <= 1'b0;
            end
        end
    end

endmodule
```

شکل 38- ماژول Cache Controller

نتایج شبیه سازی Cache در تصویر زیر آمده است:



شکل 39- Cache Modelsim Simulation

زمان اجرا 6400 نانوثانیه می‌باشد. هر کلاک 20 نانو ثانیه می‌باشد بنابراین تعداد کلاک از رابطه زیر بدست می‌آید:

$$total\ clocks = \frac{6400}{20} = 320$$

همانطور که مشاهده می‌شود در 320 کلاک دستورات اجرا شده‌اند.

افزایش کارایی:

$$\frac{429 - 320}{320} = 34.1\%$$

مشکلات و خطاها:

از بخش Forwarding به بعد طول کلاک های 50مگاهرتزی برای انجام برخی قسمت ها مانند status register و forwarding کافی نبود بنابراین از یک frequency divider در کوارتز استفاده کردیم و کلاک را به 25مگاهرتز کاهش دادیم.