

به نام خدا گزارش گروه ۳، پروژه نهایی درس معماری کامپیوتر

مراحل طراحی

ابتدا واحد کنترل و مسیر داده با صورت جدگانه طراحی شد و بعد پردازندۀ تشکیل شد. سپس دستورات اختیاری اضافه شد.

چالش های طراحی

از آنجایی که روند طراحی با احتیاط و به صورت مرحله ای انجام شد با چالش مشخصی رو به رو نشدیم. تنها نکته قابل ذکر این است که ابتدا فکر می کردیم خروجی مسیر داده شامل aluResult و باقی خروجی ها ای است که در پردازندۀ تک سیکل وجود داشت، ولی بعد متوجه شدیم بهتر است خروجی مسیر داده adr, writeData Instruction باشد و برای این تغییرات باید در مازول های مرتبط هم اصلاحاتی رو انجام دهیم. اصلاح کردن این مشکلات هم به کمک نگاه کردن به شماتیک و پیدا کردن اشکالات آن انجام شد. چالش بعدی هم این بود که مقدار 4^4 که به مالتیپلیکسر متصل می شود را ۸ بیت در نظر گرفته بودیم که مشکل ایجاد کرد چون ورودی باید 16^4 بیت می بود و pc به جای چهار با عددی جمع می شد که ۸ بیت دیگر شد. پس این مشکل رو هم حل کردیم.

طراحی control unit

instrDec ○

```
module instrDec (input logic [6:0] op, output logic [1:0] immSrc);

    always_comb begin

        case (op)
            7'b0110011: immSrc = 2'b00;
            7'b0010011: immSrc = 2'b00;
            7'b00000011: immSrc = 2'b00;
            7'b0100011: immSrc = 2'b01;
            7'b1100011: immSrc = 2'b10;
            7'b1101111: immSrc = 2'b11;
            default: immSrc = 2'bx;
        endcase

    end

endmodule
```

با نوجه به جدول ۳ مختص به immSrc، به op مقدار می دهیم. دیفالت هم x است که بتوان ارور ها را بهتر تشخیص داد.

aluDec o

```
module aluDec (input logic [1:0] aluop,
                input logic op5, funct7b5,
                input logic [2:0] funct3,
                output logic [2:0] alucontrol);

    always_comb begin

        case(aluop)
            2'b00: alucontrol = 3'b000;
            2'b01: alucontrol = 3'b001;
            2'b10: case (funct3)
                3'b000: if({op5, funct7b5} == 2'b11)
                    alucontrol = 3'b001;
                else
                    alucontrol = 3'b000;
                3'b010: alucontrol = 3'b101;
                3'b110: alucontrol = 3'b011;
                3'b111: alucontrol = 3'b010;
                default: alucontrol = 3'bx;
            endcase
            default: alucontrol = 3'bx;
        endcase
    end

endmodule
```

با توجه به جدول ۲، و با توجه به مقدار aluop, func3, op5, func7b5 محاسبه کرده و در خروجی بگذاریم. دیفالت هم دوباره به همان علت x است.

controller o

```
module controller(input logic clk,
                  input logic reset,
                  input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic funct7b5,
                  input logic zero,
                  output logic [1:0] immsrc,
                  output logic [1:0] alusrca, alusrcb,
                  output logic [1:0] resultsrc,
                  output logic adrsrc,
                  output logic [2:0] alucontrol,
                  output logic irwrite, pcwrite,
                  output logic regwrite, memwrite);

    logic branch, pcupdate;
    logic [1:0] aluop;

    fsm MainFSM(clk, reset, op, branch, pcupdate, regwrite,
                memwrite, irwrite, resultsrc, alusrcb, alusrca, adrsrc, aluop);
    aluDec AluDecoder(aluop, op[5], funct7b5, funct3, alucontrol);
    instrDec InstrDecoder(op, immsrc);

    assign pcwrite = (branch & zero) | pcupdate;

endmodule
```

ماژول کنترلر شامل mainFSM , aluDec, instrDec می باشد.

```

module fsm (input logic clk,rst,
            input logic [6:0] op,
            output logic branch, pcupdate, regwrite, memwrite, irwrite,
            output logic [1:0] resultsrc, alusrch, alusrca,
            output logic adrsrc,
            output logic [1:0] aluop);

typedef enum logic [3:0] {s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11} statetype; // sll used for error detection
statetype state, nextstate;
// state register
always_ff @(posedge clk, posedge rst) begin
    if(rst)
        state <= s0;
    else
        state <= nextstate;
end
// nextstate logic
always_comb begin
    case(state)
        s0: nextstate = s1;
        s1: case(op)
                7'b0110011: nextstate = s6;
                7'b0010011: nextstate = s8;
                7'b0000011: nextstate = s2;
                7'b0100011: nextstate = s2;
                7'b1100011: nextstate = s10;
                7'b1101111: nextstate = s9;
                default: nextstate = s11;
            endcase
        s2: if(op[5])
                nextstate = s5;
            else
                nextstate = s3;
        s3: nextstate = s4;
        s4: nextstate = s0;
        s5: nextstate = s0;
        s6: nextstate = s7;
        s7: nextstate = s0;
        s8: nextstate = s7;
        s9: nextstate = s7;
        s10: nextstate = s0;
        s11: nextstate = s11;
    endcase
end
// output logic
always_comb begin
    case(state)
        s0: begin
            branch = 1'b0 ;
            pcupdate = 1'b1 ;

```

```

// output logic
always_comb begin
    case(state)
        s0: begin
            branch = 1'b0 ;
            pcupdate = 1'b1
            regwrite = 1'b0
            memwrite = 1'b0
            irwrite = 1'b1 ;
            resultsrc = 2'b1
            alusrcb = 2'b10
            alusrca = 2'b00
            adrsrc = 1'b0 ;
            aluop = 2'b00 ;
        end
    s1: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b0
            memwrite = 1'b0
            irwrite = 1'b0 ;
            resultsrc = 2'b0
            alusrcb = 2'b01
            alusrca = 2'b01
            adrsrc = 1'b0 ;
            aluop = 2'b00 ;
        end
    s2: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b0
            memwrite = 1'b0
            irwrite = 1'b0 ;
            resultsrc = 2'b0
            alusrcb = 2'b01
            alusrca = 2'b10
            adrsrc = 1'b0 ;
            aluop = 2'b00 ;
        end
    s3: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b0
            memwrite = 1'b0
            irwrite = 1'b0 ;
            resultsrc = 2'b0
            alusrcb = 2'b00
            alusrca = 2'b00
            adrsrc = 1'b1 ;
            aluop = 2'b00 ;
        end
    s4: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b1 ;
            memwrite = 1'b0 ;
            irwrite = 1'b0 ;
            resultsrc = 2'b01 ;
            alusrcb = 2'b00 ;
            alusrca = 2'b00 ;
            adrsrc = 1'b0 ;
            aluop = 2'b00 ;
        end
    s5: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b0
            memwrite = 1'b1 ;
            irwrite = 1'b0 ;
            resultsrc = 2'b00 ;
            alusrcb = 2'b00
            alusrca = 2'b00
            adrsrc = 1'b1 ;
            aluop = 2'b00 ;
        end
    s6: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b0
            memwrite = 1'b0
            irwrite = 1'b0 ;
            resultsrc = 2'b00
            alusrcb = 2'b00
            alusrca = 2'b10
            adrsrc = 1'b0 ;
            aluop = 2'b10 ;
        end
    s7: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b1 ;
            memwrite = 1'b0 ;
            irwrite = 1'b0 ;
            resultsrc = 2'b00 ;
            alusrcb = 2'b00
            alusrca = 2'b00
            adrsrc = 1'b0 ;
            aluop = 2'b00 ;
        end
    s8: begin
            branch = 1'b0 ;
            pcupdate = 1'b0
            regwrite = 1'b0
            memwrite = 1'b0
            irwrite = 1'b0 ;
            resultsrc = 2'b00
            alusrcb = 2'b00
            alusrca = 2'b00
            adrsrc = 1'b0 ;
            aluop = 2'b00 ;
        end
    end

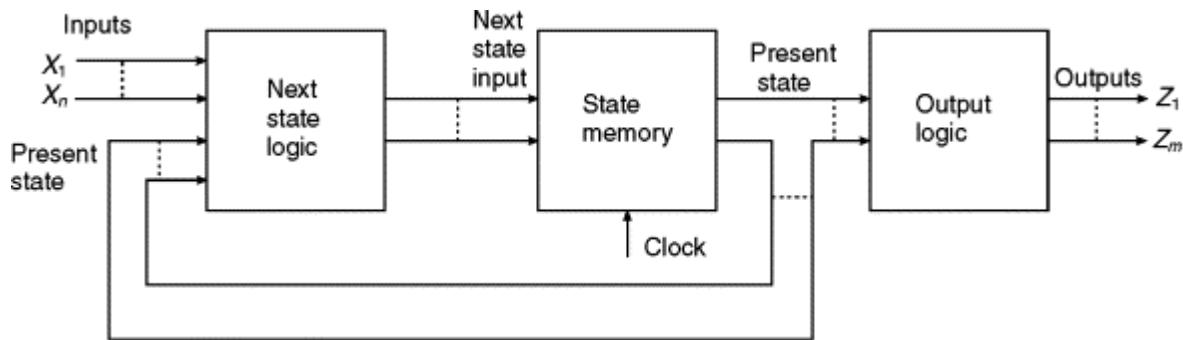
```

```

        branch = 1'b0 ;
        pcupdate = 1'b0 ;
        regwrite = 1'b0 ;
        memwrite = 1'b0 ;
        irwrite = 1'b0 ;
        resultsrc = 2'b00 ;
        alusrcb = 2'b01 ;
        alusrca = 2'b10 ;
        adrsrc = 1'b0 ;
        aluop = 2'b10 ;
    end
s9: begin
        branch = 1'b0 ;
        pcupdate = 1'b1 ;
        regwrite = 1'b0 ;
        memwrite = 1'b0 ;
        irwrite = 1'b0 ;
        resultsrc = 2'b00 ;
        alusrcb = 2'b10 ;
        alusrca = 2'b01 ;
        adrsrc = 1'b0 ;
        aluop = 2'b00 ;
    end
s10: begin
        branch = 1'b1 ;
        pcupdate = 1'b0 ;
        regwrite = 1'b0 ;
        memwrite = 1'b0 ;
        irwrite = 1'b0 ;
        resultsrc = 2'b00 ;
        alusrcb = 2'b00 ;
        alusrca = 2'b10 ;
        adrsrc = 1'b0 ;
        aluop = 2'b01 ;
    end
s11: begin
        branch = 1'bx ;
        pcupdate = 1'bx ;
        regwrite = 1'bx ;
        memwrite = 1'bx ;
        irwrite = 1'bx ;
        resultsrc = 2'bx ;
        alusrcb = 2'bx ;
        alusrca = 2'bx ;
        adrsrc = 1'bx ;
        aluop = 2'bx ;
    end
endcase
end

```

با توجه به شکل ۷.۴۸ کتاب، ابتدا متغیر stateType را تعریف می کنیم که از استیت ۱ تا ۱۱ را در برابر می گیرد. سپس state , nextState تعريف می شود.
هم برای تشخیص اروری هست که مربوط به opcode نا معتبر است ، برای راحت شدن شبیه سازی است.



با توجه به این شماتیک منطق برنامه باید شامل:
nextState (combinational),
stateMemory (sequential),
(به علت اینکه ماشین مور است) Output logic (combinational)
باشد.

طراحی Datapath

Adder ○

```
module adder (input [31:0] a, b, output [31:0] y);
assign y = a+b;
endmodule
```

این مژول دو ورودی ۳۲ بیتی را به عنوان ورودی گرفته و نتیجه جمع آن ها را به عنوان خروجی می دهد.

Flopr ○

```
module flopr #(parameter WIDTH = 8)
  (input logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else q <= d;
endmodule
```

این مژول پارامتر width را با مقدار ۸ تعریف می کند و سه ورودی (8 bits) clk, reset, d را گرفته و در لبه بالا رونده clock و reset ، اگر reset فعال بود مقدار ۰ را در خروجی می ریزد و اگر نه، ورودی d را در خروجی قرار می دهد.

Flopnr ○

```
module flopnr #(parameter WIDTH = 8)
  (input logic clk, reset, en, input logic [WIDTH-1:0] d, output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else if (en) q <= d;
endmodule
```

این مژول شبیه مژول قبل است فقط با این تفاوت که ورودی دیگری به اسم en (enable) هم دارد که فقط وقتی en فعال باشد مقدار d در خروجی قرار می گیرد.

Mux3 ○

```
module mux3 #(parameter WIDTH = 8)
  (input logic [WIDTH-1:0] d0, d1, d2,
   input logic [1:0] s,
   output logic [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

این مژول سه ورودی به طول width (در اینجا ۸) بیت می گیرد و با توجه به ورودی select (s) که دو بیتی است، با کمک عملیات سه عملوندی، از بین ورودی ها یکی را انتخاب کرده و روی خروجی قرار می دهد.

```

module mux2 #(parameter WIDTH = 8)
    (input logic [WIDTH-1:0] d0, d1, input logic s, output logic [WIDTH-1:0] y);
begin
    assign y = s ? d1 : d0;
endmodule

```

Mux2 ○

همانند مازول قبل است فقط به جای سه ورودی، دو ورودی می‌گیرد و از بین آن دو انتخاب می‌کند.

```

module extend (input logic [31:7] instr, input logic [1:0] immsrc, output logic [31:0] immext);
begin
    always_comb
        case(immsrc) //controller produces immsrc signal
            //I
            2'b00: immext = {{20{instr[31]}}, instr[31:20]};
            //S
            2'b01: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            //B
            2'b10: immext = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
            //J
            2'b11: immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
            default: immext = 32'bx; // undefined ?
        endcase
endmodule

```

Extend ○

این مازول بیت‌های لازم از instruction را به همراه immsrc دو بیتی که نشان دهنده روش Extend کردن است، به عنوان ورودی می‌گیرد و با استفاده از الگوی type دستور، (intr[31:7]) را به ۳۲ بیت گسترش داده و در خروجی می‌گذارد.

```

module alu (input logic [31:0] src1, src2, input logic [2:0] aluc, output logic [31:0] out, output logic zero);
begin
    always_comb begin
        case (aluc)
            3'b000: begin
                out = src1 + src2;
            end
            3'b001: begin
                out = src1 - src2;
            end
            3'b010: begin
                out = src1 & src2;
            end
            3'b011: begin
                out = src1 | src2;
            end
            3'b101: begin
                if (src1 < src2)
                    out = 1;
                else
                    out = 0;
            end
            default: begin
                out = 0;
            end
        endcase
        if (out==0)
            zero = 1;
        else
            zero = 0;
    end
endmodule

```

ALU ○

این مازول دو عدد ۳۲ بیتی src1, src2 را به همراه سیگنال کنترل aluc جهت تعیین نوع عملیات دریافت می‌کند.

بر حسب aluc خروجی که همان `out` هست را محاسبه کرده و خروجی دوم هم که `zero` است در نهایت اگر `out` صفر بود یک می شود و در غیر این صورت صفر می شود. (صفر بودن `out` به معنی برابر بودن ورودی ها است).
این `alu` عملیات `+`, `-`, `&`, `|`، و مقایسه را انجام می دهد.

```
module memory(input logic      clk, we,
              input logic [31:0] a, wd,
              output logic [31:0] rd);
    logic [31:0] RAM[63:0];
    initial
        $readmemh("riscvtest.txt", RAM);
    assign rd = RAM[a[31:2]]; // word aligned
    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;
endmodule
```

Mem ○

این ماثول ورودی های clock, A, wd (Write Data), we (Write Enable) را گرفته و در لبه بالا رونده کلک اگر Write Enable فعال بود مقدار Data را در حافظه ای ذخیره می کند که مربوط به A هست. این مموری دستورات را از فایل riscvtest.txt می خواند.
همچنین خروجی آن (Read data) rd است که برابر مقداری قرار می گیرد که به آن اشاره دارد.

```
module regFile (input logic clk, input logic we, input logic [4:0] a1, a2, a3, input logic [31:0] wd, output logic [31:0] rd1, rd2);
    logic [31:0] regf [31:0];
    always_ff @(posedge clk)
    begin
        if (we && (a3!=0))
            regf[a3] <= wd;
    end
    assign rd1 = a1==0 ? 0 : regf[a1];
    assign rd2 = a2==0 ? 0: regf[a2];
endmodule
```

regFile ○

ورودی ها:
Clock

We: اگر فعال باشد مقدار Wd روی رجیستری که در A3 ذخیره شده قرار می گیرد. (همچنین A3 باید صفر نباشد).
A1: رجیستر مبدا اول که از instruction گرفته شده.
A2: رجیستر مبدا دوم که از instruction گرفته شده.
A3: رجیستر مقصد
wd: مقداری که باید نوشته شود.
خروجی ها:
rd1: رجیستر مبدا اول
rd2: رجیستر مبدا اول

```

module dataPath (input logic clk, reset,
                 input logic [1:0] ImmSrc,
                 input logic [2:0] ALUControl,
                 input logic [1:0] ResultSrc,
                 input logic IRWrite,
                 input logic RegWrite,
                 input logic [1:0] ALUSrcA, ALUSrcB,
                 input logic AdrSrc,
                 input logic PCWrite,
                 input logic [31:0] ReadData,
                 output logic Zero,
                 output logic [31:0] Adr,
                 output logic [31:0] WriteData,
                 output logic [31:0] instr);

logic [31:0] Result, ALUOut, ALUResult;
logic [31:0] RD1, RD2, A, SrcA, SrcB, Data;
logic [31:0] ImmExt;
logic [31:0] PC, OldPC;

//pc
flopnr #(32) pcFlop(clk, reset, PCWrite, Result, PC);

//regFile
regFile rf(clk, RegWrite, instr[19:15], instr[24:20], instr[11:7], Result, RD1, RD2);
extend ext(instr[31:7], ImmSrc, ImmExt);
flop # (32) regF( clk, reset, RD1, A );
flop # (32) regF_2( clk, reset, RD2, WriteData );

//alu
mux3 #(32) srcAmux(PC, OldPC, A, ALUSrcA, SrcA);
mux3 #(32) srcBmux(WriteData, ImmExt, 32'd4, ALUSrcB, SrcB);
alu alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
flop # (32) aluReg (clk, reset, ALUResult, ALUOut);
mux3 #(32) resultMux(ALUOut, Data, ALUResult, ResultSrc, Result );

//mem
mux2 #(32) adrMux(PC, Result, AdrSrc, Adr);
flopnr #(32) memFopl(clk, reset, IRWrite, PC, OldPC);
flopnr #(32) memFlop2(clk, reset, IRWrite, ReadData, instr);
flop # (32) memDataFlop(clk, reset, ReadData, Data);

endmodule

```

Datapath ○

Inputs and outputs

(۱) در این قسمت اگر pcWrite یک باشد نتیجه محاسبه pc روی رجیستر قرار می‌گیرد.

(۲) رجیستر فایل با ورودی‌ها مقدار دهی شده و به ما rd1, rd2 را می‌دهد. مقدار عددی اکستند می‌شود و rd1 و rd2 در دو رجیستر بدون Enable ذخیره می‌شوند.

(۳) در این بخش ورودی‌های alu توسط دو تا مالتیپلکسر تعیین شده و سپس نتیجه عملیات در یک رجیستر ذخیره می‌شود. نتیجه نهایی توسط یک مالتیپلکسر مشخص می‌شود که از بین aluResult, Data, aluOut, جواب نهایی یا همان Result را انتخاب می‌کند.

(۴) ابتدا آدرس مورد نیاز توسط یک مالتیپلکسر انتخاب می‌شود و سپس PC در رجیستر‌های ذخیره می‌شوند. ReadData,

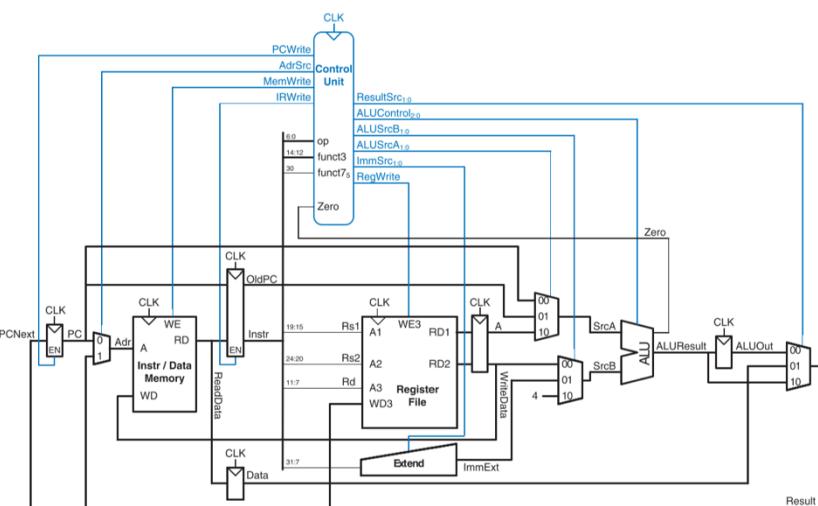


Figure 7.27 Complete multicycle processor

به طور کلی طراحی مسیر داده توصیف شکل رویه رو است.

Riscv_MultiCycle ○

```

module riscV_MultiCycle (input logic clk, reset,
                        input logic [31:0] ReadData,
                        output logic [31:0] Adr,
                        output logic MemWrite,
                        output logic [31:0] WriteData);

    logic [1:0] ResultSrc, ImmSrc , ALUSrcA, ALUSrcB;
    logic adrSrc, Zero;
    logic [2:0] alucontrol;
    logic irwrite, pcwrite;
    logic regwrite;
    logic [31:0] Instr;

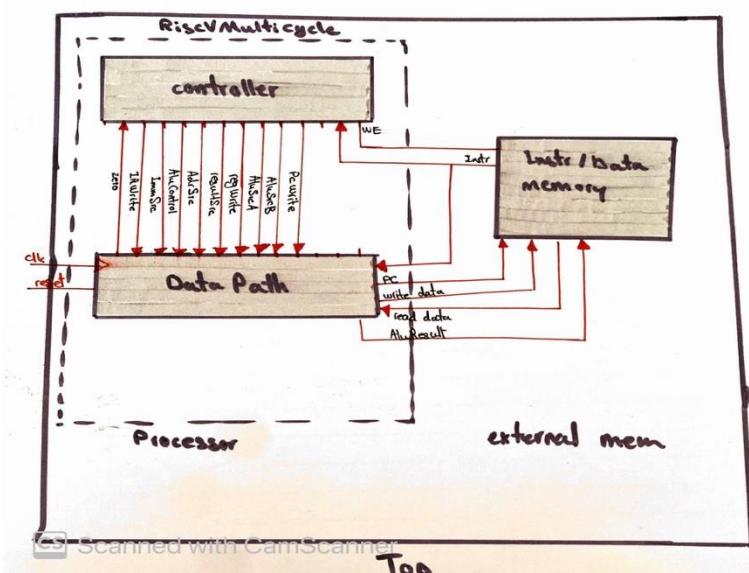
    controller c (clk, reset, Instr[6:0], Instr[14:12], Instr[30], Zero, ImmSrc, ALUSrcA, ALUSrcB, ResultSrc, adrSrc, alucontrol, irwrite, pcwrite, regwrite,
    dataPath dp (clk, reset, ImmSrc, alucontrol, ResultSrc, irwrite, regwrite, ALUSrcA, ALUSrcB, adrSrc, pcwrite, ReadData, Zero, Adr, WriteData, Instr);

endmodule

```

در این مازول از طرح زیر استفاده کردیم که نشان می دهد Riscv_MultiCycle شامل دو مازول مسیر داده و واحد کنترل است که ورودی ها و خروجی ها در آن مشخص شده اند.. مازول تاپ هم بر اساس همین ورودی ها و خروجی ها طراحی شده است. فقط به جای FristStage می شود تا کار راحت تر شود.

multi cycle processor



```

Scanned with CamScanner
Top

module top(input logic      clk, reset,
            output logic [31:0] WriteData, DataAddr,
            output logic          MemWrite);

    logic [31:0] ReadData;

    riscV_MultiCycle rvMulti(clk, reset, ReadData, DataAddr, MemWrite, WriteData);
    memory mem(clk, MemWrite, DataAddr, WriteData, ReadData);

endmodule

```

PC	lstr	state	result	Result notes
3 00	n/a	Fetch	4	PC+4
4 4	"	decode	X	old PC + 2mn
5 4	"	ExecuteL	X	Alu result = (0) + 5 = 5
6 4	"	ALUWB	5	result = ALUout
7 4	"	Fetch	8	PC+4
8 8	00 C00193	Decode	X	old PC + 2mn
9 8	"	ExecuteL	X	Alu result = (0) + 12 = 12
10 8	"	ALU WB	12	result = ALUout = 12
11 8	"	Fetch	12	PC+4
12 12	FF718393	Decode	X	old PC + 2mn
13 12	"	ExecuteL	X	Alu result = 12 - 4 = 8
14 12	"	ALUWB	3	result = ALUout = 3
15 12	"	Fetch	16	PC+4
16 16	0023E233	decode	X	old PC + 2mn
17 16	"	ExecuteR	X	Alu result = 3 or 5 = 7
18 16	"	ALUWB	(3 or 5) 7	result = aluout
19 16	"	Fetch	20	PC+4
20 20	0041F2B3	decode	X	old PC + 2mn
21 20	"	ExecuteR	X	Alu result = 12 and 7 = 4
22 20	"	ALUWB	(12 and 7) 4	result = ALUout = 4
23 20	"	Fetch	24	PC+4
24 24	004282B3	decode	X	old PC + 2mn

Pc	Instr	sstate	result	result note
25	24	"	X	Alu result = $x_5 = 4 + 7 = 11$
26	24	"	11	result = Alu Out
27	24	"	28	PC+4
28	28	02728863	decode	X
29	28	"	BEQ	X
30	28	"	Fetch	32
31	32	0041A233	decode	X
32	32	"	ExecuteR	X
33	32	"	AluWB	0
34	32	"	Fetch	36
35	36	00020463	decode	X
36	36	"	BEQ	X
37	40	"	Fetch	44
38	44	0023A233	decode	X
39	44	"	ExecuteR	X
40	44	"	AluWB	1
41	44	"	Fetch	48
42	48	005203B3	decode	X
43	48	"	ExecuteR	X
44	48	"	AluWB	12
45	48	"	Fetch	52
46	52	402383B3	decode	X

PC	Instr	state	result	result note
47	52	"	ExecuteR	X
48	52	"	AluWB	7
49	52	"	Fetch	56
50	56	0471AA23	decode	X
51	56	"	Mem Addr	X
52	56	"	Mem write	84 + [m ₃]
53	56	"	Fetch	60
54	60	06002103	decode	X
55	60	"	Mem Addr	X
56	60	"	Mem read	96 + [m ₀]
57	60	"	Mem WB	(96 + [m ₀]) _{issuing}
58	60	"	Fetch	64
59	64	005104B3	decode	X
60	64	"	ExecuteR	X
61	64	"	AluWB	18
62	64	"	Fetch	68
63	68	008001EF	decode	X
64	72	"	Jal	oldpc + 4
65	72	"	AluWB	"
66	72	"	Fetch	76
67	76	009101B3	decode	X
68	76	"	ExecuteR	X
69	76	"	AluWB	25

```
module multicycle_tb();

logic clk;
logic reset;
logic [31:0] WriteData, DataAddr;
logic MemWrite;

// instantiate device to be tested
top dut(clk, reset, WriteData, DataAddr, MemWrite);

// initialize test
initial
begin
    reset <= 1; # 22; reset <= 0;
end

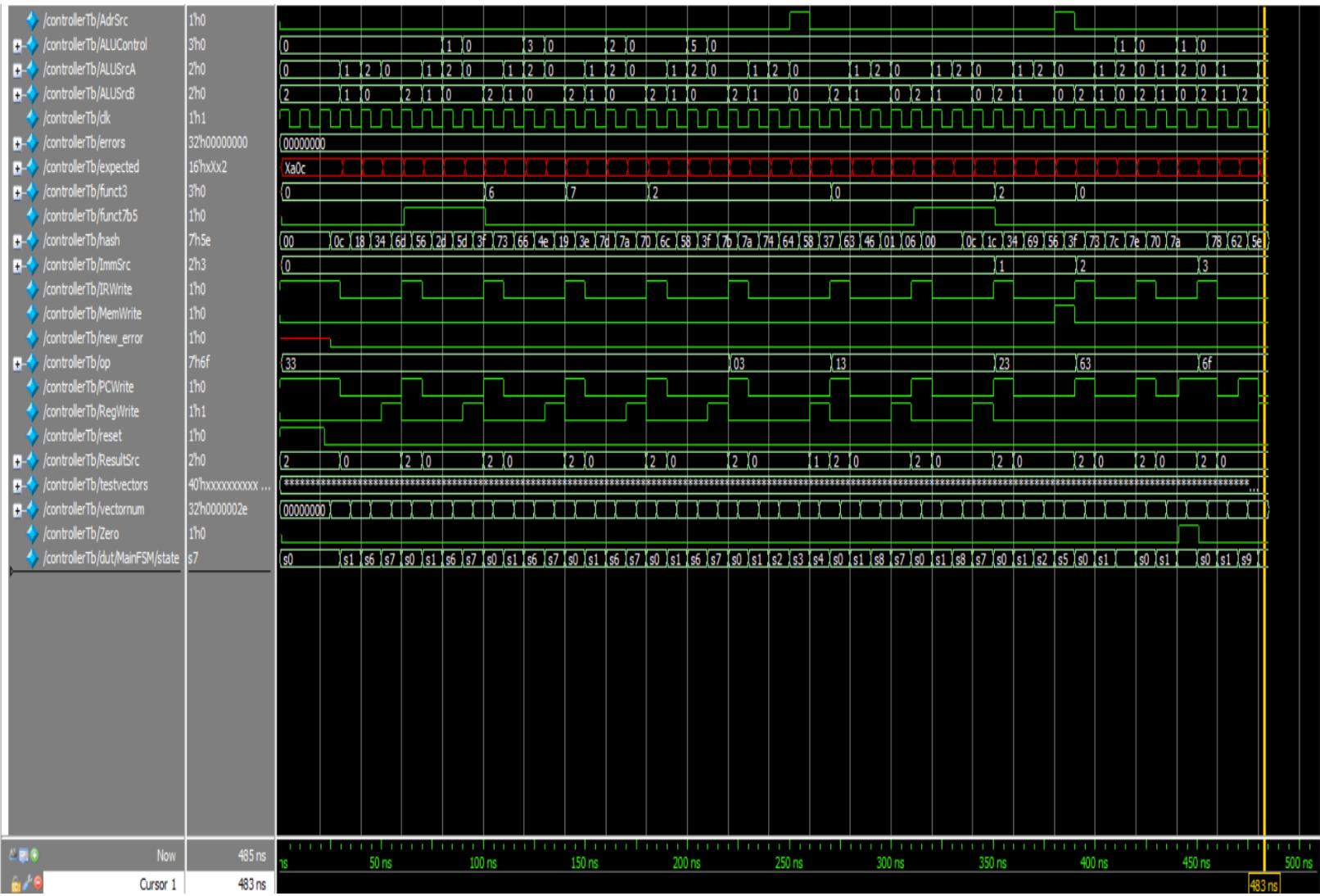
// generate clock to sequence tests
always
begin
    clk <= 1; # 5; clk <= 0; # 5;
end

// check results
always @(negedge clk) begin
if(MemWrite)
begin
if(DataAddr == 100 & WriteData == 25)
begin
$display("Simulation succeeded");
$stop;
end else if (DataAddr != 96) begin
$display("Simulation failed");
$stop;
end
end
end
endmodule
```

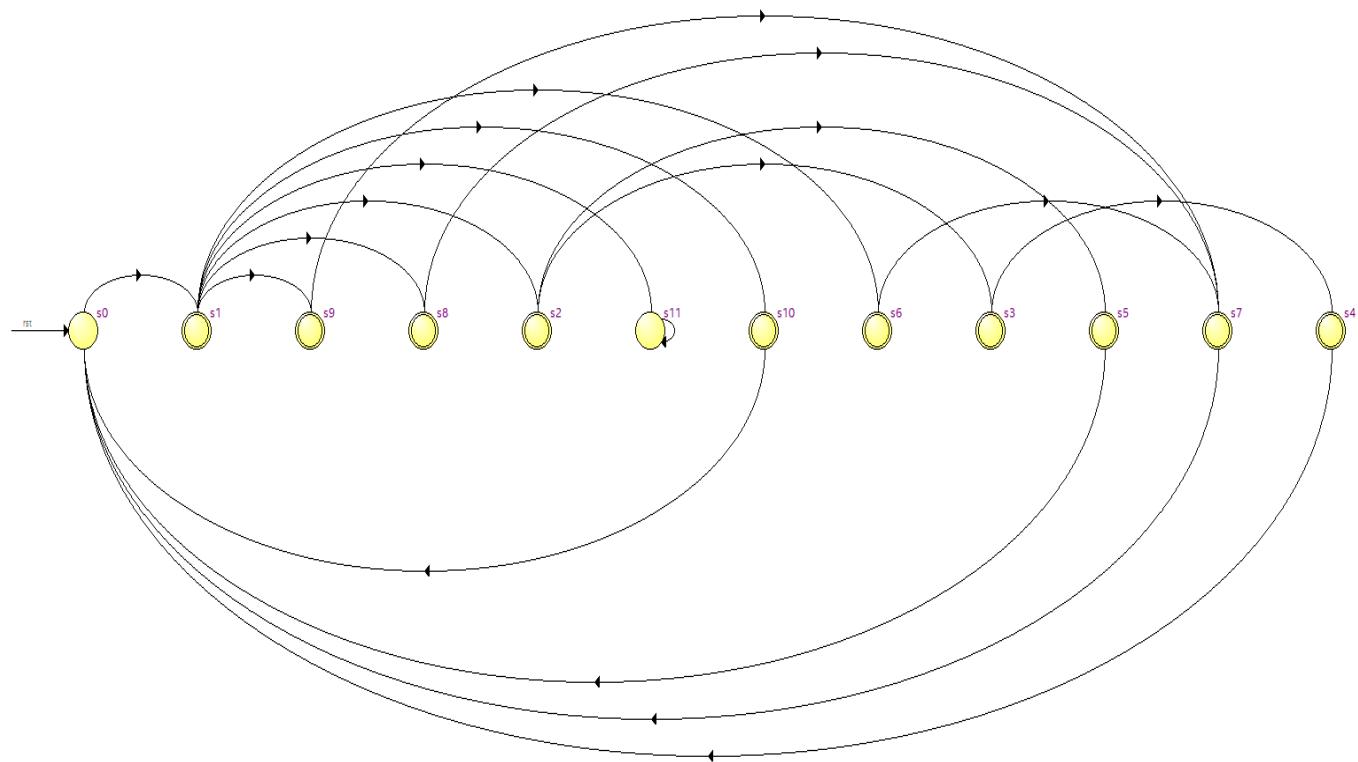
```

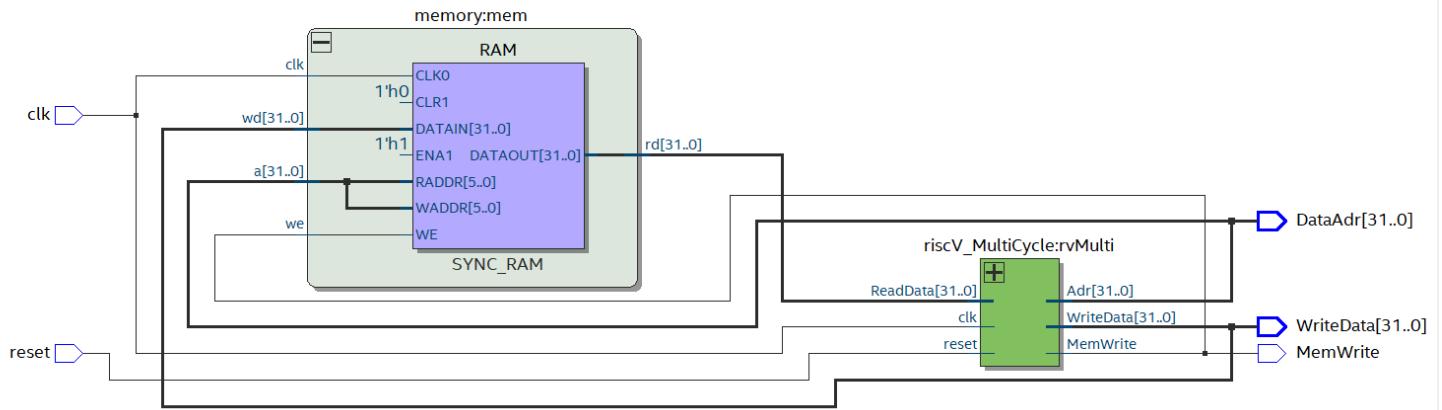
#      47 tests completed with      0 errors
# hash = 39
# ** Note: $stop    : C:/Users/Arsalan/Desktop/memProj/controllerTb.sv(106)
#   Time: 485 ns Iteration: 1 Instance: /controllerTb
# Break in Module controllerTb at C:/Users/Arsalan/Desktop/memProj/controllerTb.sv line 106

```

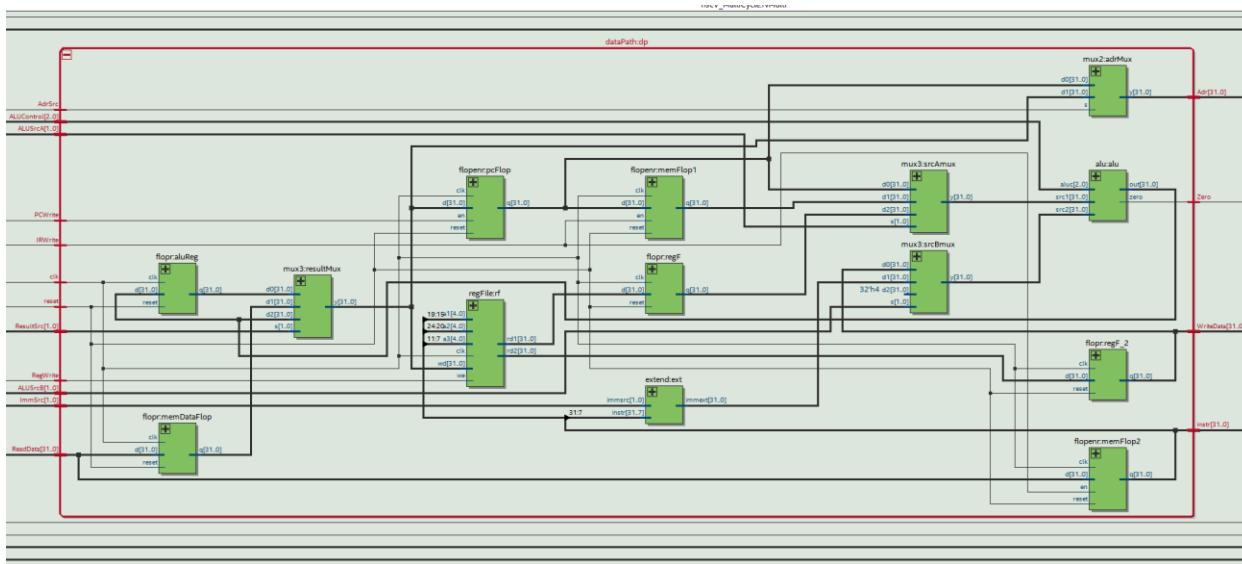
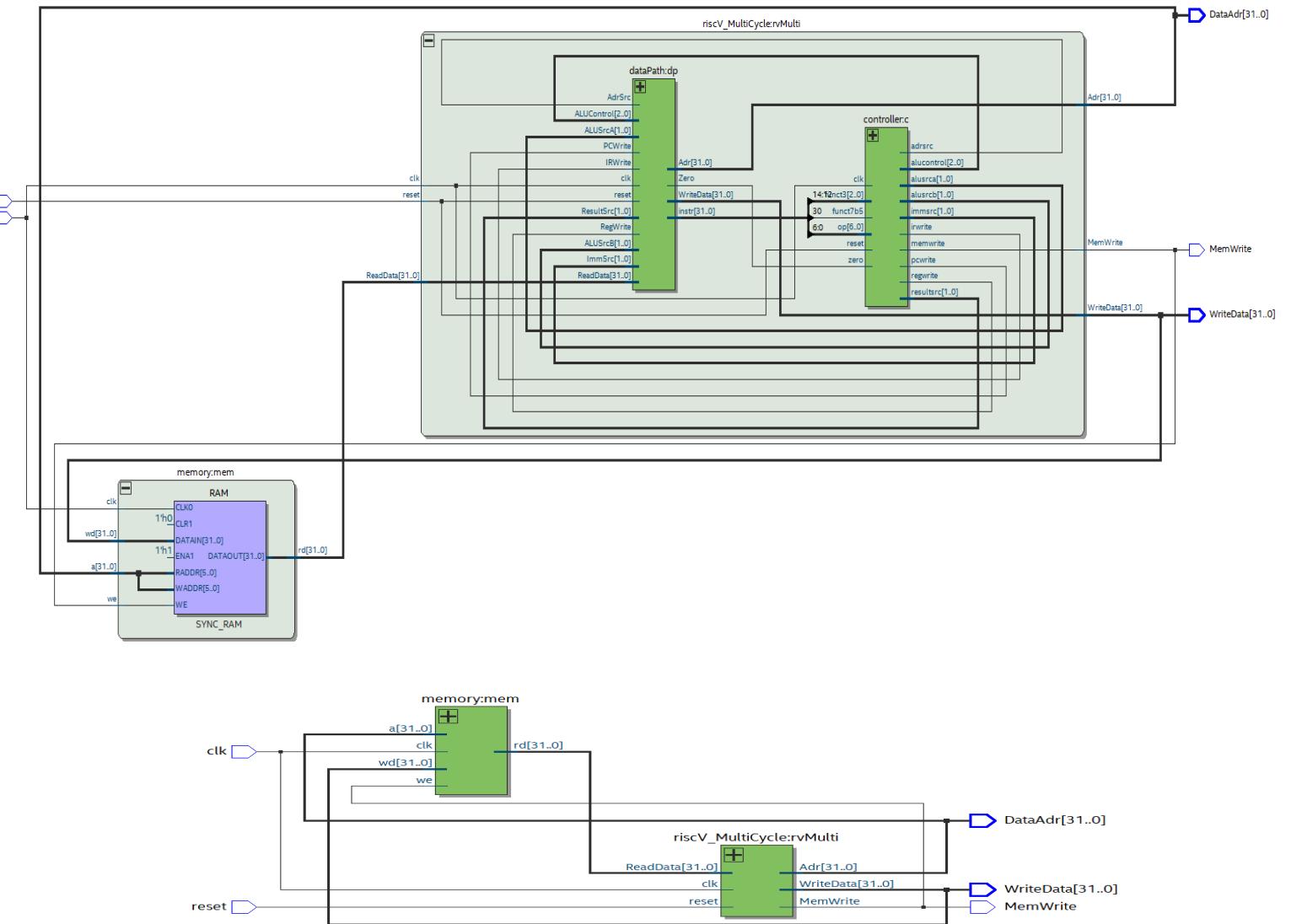


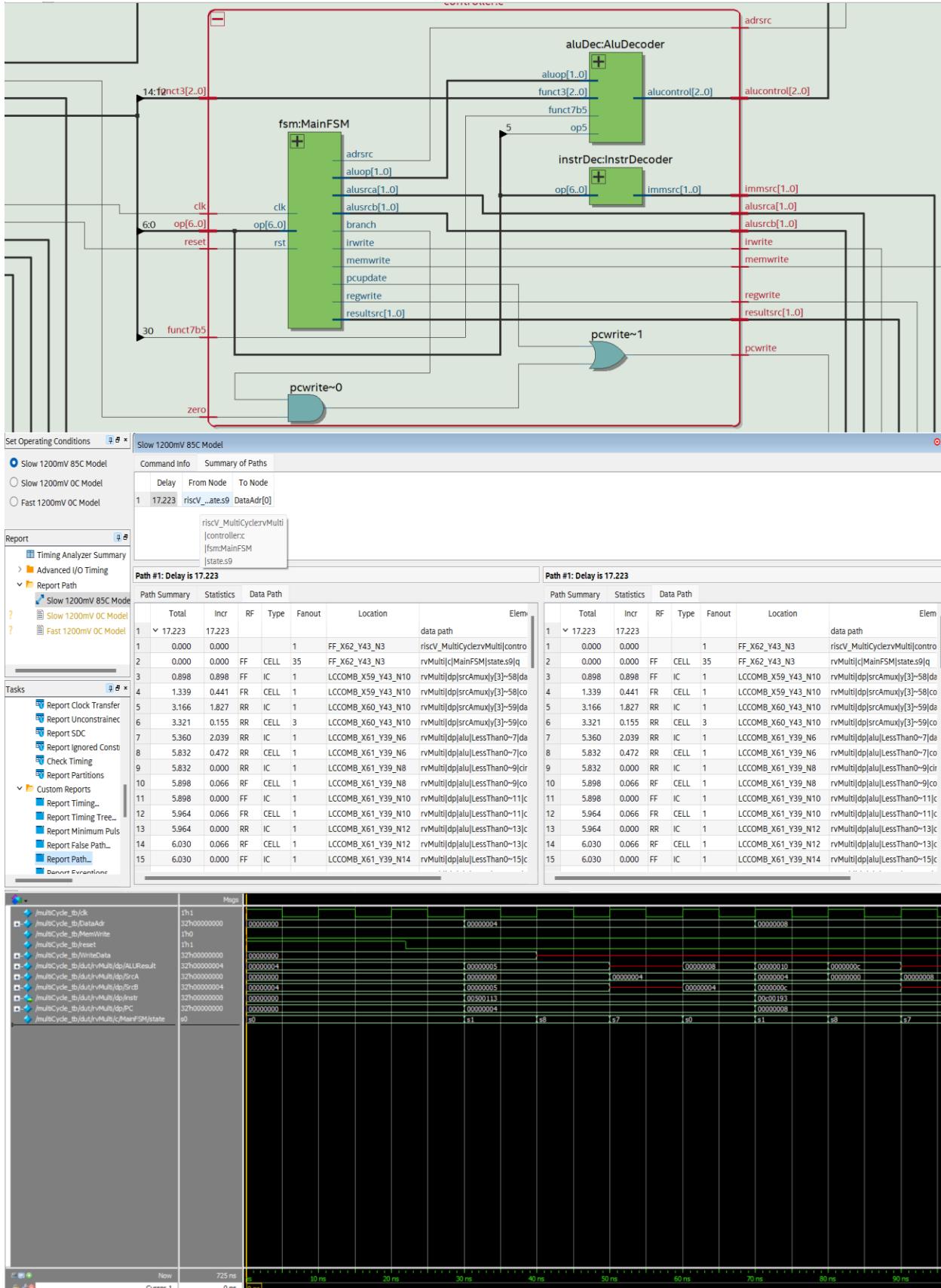
Flow Summary	
 <<Filter>>	
Flow Status	Successful - Sun Jul 03 17:54:51 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,097 / 114,480 (3 %)
Total registers	2372
Total pins	67 / 529 (13 %)
Total virtual pins	0
Total memory bits	2,048 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

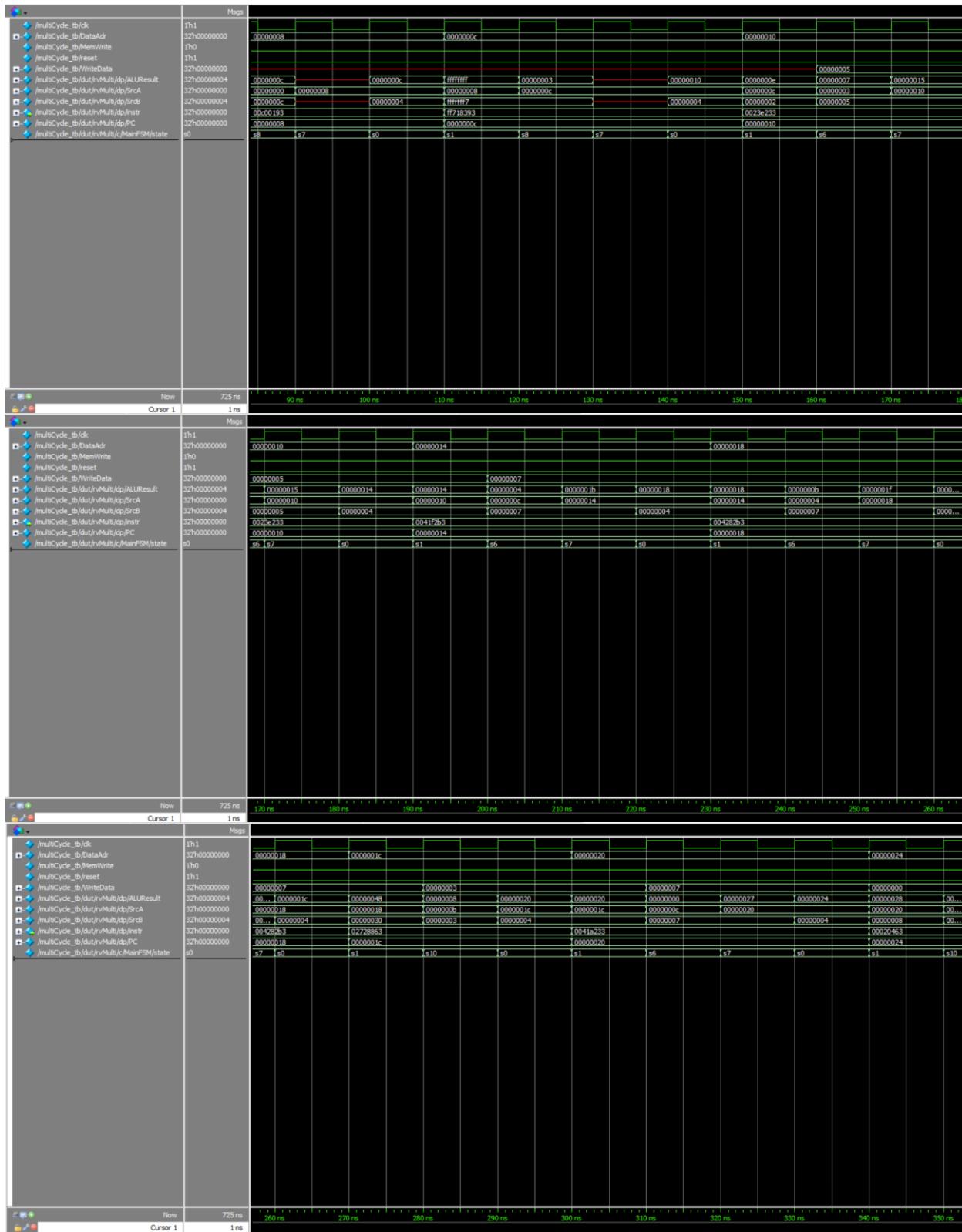


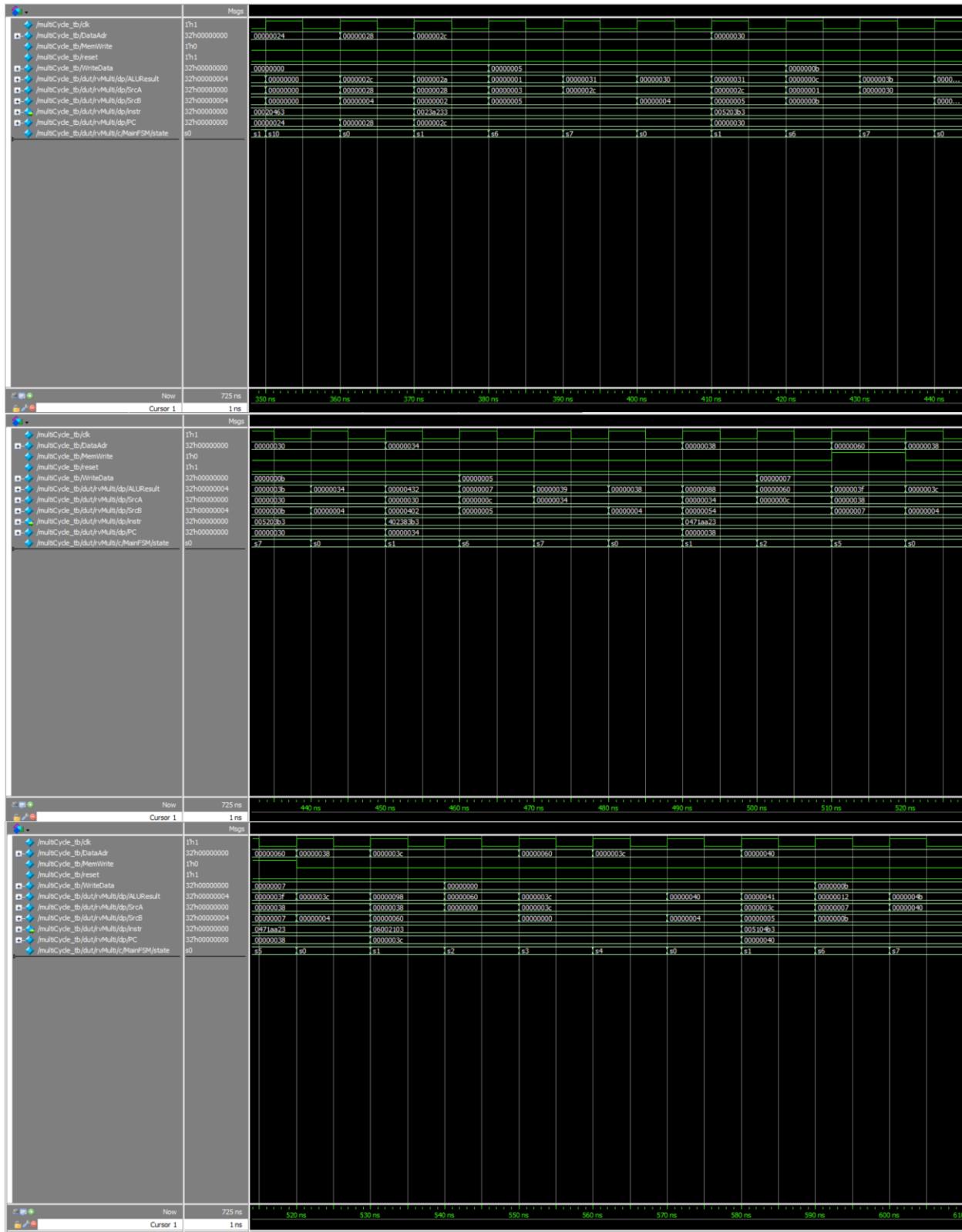


	Source State	Destination State	Condition
1	s0	s1	
2	s1	s11	(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0).(!Decoder0)
3	s1	s10	(Decoder0)
4	s1	s9	(Decoder0)
5	s1	s8	(Decoder0)
6	s1	s6	(Decoder0)
7	s1	s2	(nextstate)
8	s2	s5	(op[5])
9	s2	s3	(!op[5])
10	s3	s4	
11	s4	s0	
12	s5	s0	
13	s6	s7	
14	s7	s0	
15	s8	s7	
16	s9	s7	
17	s10	s0	
18	s11	s11	







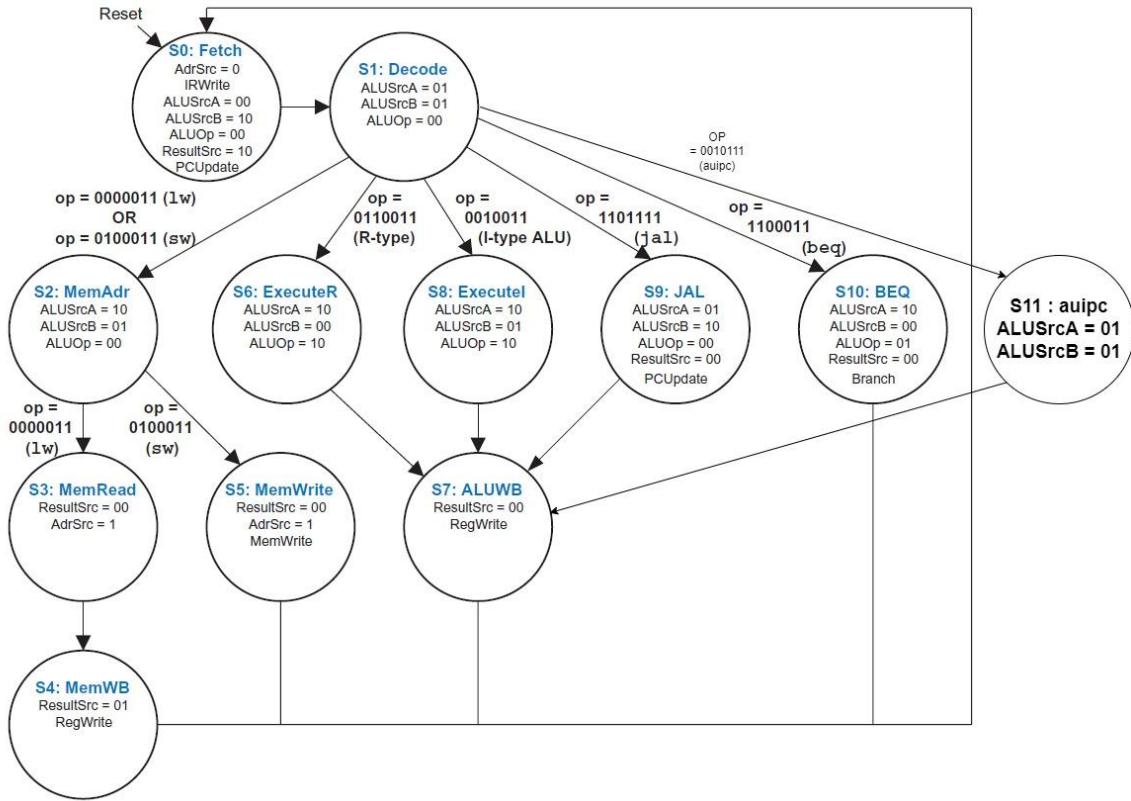




بخش امتیازی :

: auipc rd, imm_{31:20}

این دستور مقدار 32 بیتی که 20 بیت بالایی آن **imm_{31:20}** است را به **pc** اضافه کرده و حاصل را در **rd** می‌نویسد. برای انجام اینکار نیازمند ایجاد تغییر در واحد کنترل هستیم و نیازی به تغییر **dataPath** نیست.



پس بعد از مرحله Decode این دستور باید جمع انجام دهد و با توجه به این جمع در ALUSrcA باید مقدار رجیستر بباید و در ALUSrcB مقدار imm همچنین باید به یک حالت جدید اضافه کنیم :

```
module extend (input logic [31:7] instr, input logic [2:0] immsrc, output logic [31:0] immext);
    always_comb
        case(immsrc) //controller produces immsrc signal
            //I
            3'b000: immext = {{20{instr[31]}}, instr[31:20]};
            //S
            3'b001: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
            //B
            3'b010: immext = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
            //J
            3'b011: immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
            //U
            3'b100: immext = {instr[31:12], 12'b0};
            default: immext = 32'bx; // undefined ? ?
        endcase
    endmodule
```

```

1  module instrDec ( input logic [6:0] op,
2                   output logic [2:0] immsrc);
3
4   always_comb begin
5
6     case(op)
7       7'b0110011: immsrc = 3'b000;
8       7'b0010011: immsrc = 3'b000;
9       7'b00000011: immsrc = 3'b000;
10      7'b0100011: immsrc = 3'b001;
11      7'b1100011: immsrc = 3'b010;
12      7'b1101111: immsrc = 3'b011;
13      7'b0010111: immsrc = 3'b100;
14      7'b0110111: immsrc = 3'b100;
15      7'b1100111: immsrc = 3'b000;
16      default: immsrc = 3'bx;
17   endcase
18
19 end
20
21
22 endmodule

```

```

11  always_ff @(posedge clk, posedge rst) begin
12    if(rst)
13      state <= s0;
14    else
15      state <= nextstate;
16  end
17  // nextstate logic
18  always_comb begin
19    case(state)
20      s0: nextstate = s1;
21      s1: case(op)
22        7'b0110011: nextstate = s6;
23        7'b0010011: nextstate = s8;
24        7'b00000011: nextstate = s2;
25        7'b0100011: nextstate = s2;
26        7'b1100011: nextstate = s10;
27        7'b1101111: nextstate = s9;
28        7'b0010111: nextstate = s11;
29        7'b0110111: nextstate = s12;
30        7'b1100111: nextstate = s2;
31        default: nextstate = s13;
32    endcase
33
34    aluop = 2'b01;
35  end
36
37  s11 : begin
38    branch = 1'b0 ;
39    pcupdate = 1'b0 ;
40    regwrite = 1'b0 ;
41    memwrite = 1'b0 ;
42    irwrite = 1'b0 ;
43    resultsrc = 2'b00 ;
44    alusrcb = 2'b01 ;
45    alusrca = 2'b01 ;
46    adrsrc = 1'b0 ;
47    aluop = 2'b00 ;
48  end
49
50  s12 : begin
51    branch = 1'b0 ;
52    pcupdate = 1'b0 ;
53    regwrite = 1'b1 ;
54  end

```

پس برای دستورات U-type هم حالتی ایجاد شد که به درستی عمل کند.

پس از انجام تغییرات طراحی نوبت تست این دستور است ، اول همان riscvtest را امتحان میکنیم تا ببینیم دستور اضافه شده بر عملکرد دیگر دستورات ناثیر نداشته باشد .

```

# Simulation succeeded
# ** Note: $stop    : C:/Users/Arsalan/Desktop/memProj/multiCycle_tb.sv(31)
#   Time: 725 ns  Iteration: 1  Instance: /multiCycle_tb
# Break in Module multiCycle_tb at C:/Users/Arsalan/Desktop/memProj/multiCycle_tb.sv line 31

```

حال نوبت طراحی تست مجزا و اجرای آن است :

```

#0 aulpc x2, 1           //x2 = 00000000000000000000000000000000 = 0x1000 = 4096 | 00000000000000000000000000000001-00010-0010111 | 0x00001117
#4 sw x2, 100(x0)        //mem[100] = 0x1000 = 4096 | 0000011-00010-00000-010-00100-0100011 | 0x06202223

```

کد های ماشین را در فایلی جدا به نام `auiptest.txt` مینویسیم و مسیر خواندن داده در `memory` را به این فایل تغییر میدهیم ، یک تست بنج جدا نوشته و شبیه سازی را شروع میکنیم :

```

module auiptb();

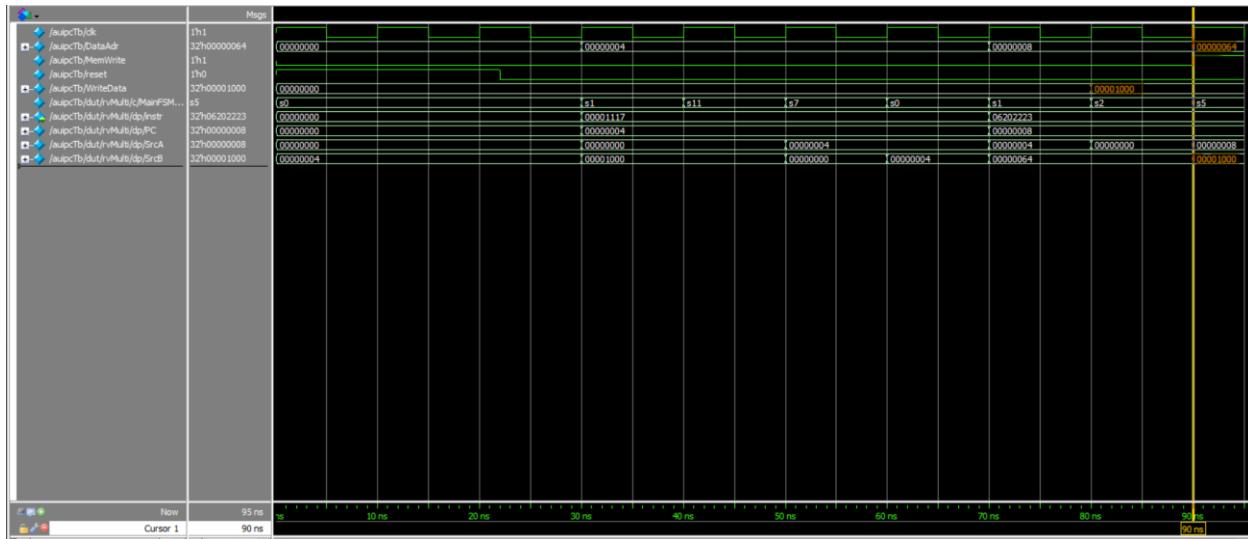
logic clk;
logic reset;
logic [31:0] WriteData, DataAdr;
logic MemWrite;

// instantiate device to be tested
top dut(clk, reset, WriteData, DataAdr, MemWrite);

// initialize test
initial
begin
    reset <= 1; # 22; reset <= 0;
end

// generate clock to sequence tests
always
begin
    clk <= 1; # 5; clk <= 0; # 5;
end
always @(negedge clk) begin
    if(MemWrite)
        begin
            if(DataAdr == 100 & WriteData == 4096)
                begin
                    $display("Simulation succeeded");
                    $stop;
                end
            else
                begin
                    $display("Simulation failed");
                    $stop;
                end
        end
    end
end
endmodule

```



```

# Simulation succeeded
# ** Note: $stop : C:/Users/Arsalan/Desktop/memProj/auipcTb.sv(29)
#   Time: 95 ns Iteration: 1 Instance: /auipcTb
# Break in Module auipcTb at C:/Users/Arsalan/Desktop/memProj/auipcTb.sv line 29

```

تست این دستور موفقیت آمیز بود ، حال نوبت به دستورات shift میرسد . این دستورات R_type هستند پس نیازی به تغییر مسیر داده و واحد کنترل نیست فقط باید حالت محاسبه شیفت به alu و حالت دستور آن در aluDec اضافه شود :

```

  end
4'b0100: begin // sra
  out = $signed(src1) >>> $unsigned(src2[4:0]);
  cout = 0;
end
4'b0101: begin
  if ($signed(src1) < $signed(src2))
    out = 1;
  else
    out = 0;
  cout = 0;
end
4'b0110: begin // srl
  out = src1 >> src2[4:0];
  cout = 0;
end
4'b0111: begin // sll
  out = src1 << src2[4:0];
  cout = 0;
end
  .
  .
  .

```

```

case(aluop)
2'b00: alucontrol = 4'b0000;
2'b01: alucontrol = 4'b0001;
2'b10: case (funct3)
    3'b000: if({op5, funct7b5} == 2'b11)
        alucontrol = 4'b0001;
    else
        alucontrol = 4'b0000;
    3'b001: // sll
        alucontrol = 4'b0111;
    3'b010: alucontrol = 4'b0101;
    3'b011:// sltu
        alucontrol = 4'b1000;
    3'b100:// xor
        alucontrol = 4'b1001;
    3'b101: if(funct7b5) // sra
        alucontrol = 4'b0100;
    else // srl
        alucontrol = 4'b0110;
    3'b110: alucontrol = 4'b0011;
    3'b111: alucontrol = 4'b0010;
    default: alucontrol = 4'bx;
endcase

```

حال نوبت انجام تست riscvtest است :

```

# Simulation succeeded
# ** Note: $stop : C:/Users/Arsalan/Desktop/memProj/multiCycle_tb.sv(31)
#   Time: 725 ns Iteration: 1 Instance: /multiCycle_tb
# Break in Module multiCycle_tb at C:/Users/Arsalan/Desktop/memProj/multiCycle_tb.sv line 31

```

موفقیت آمیز بود ، نوبت طراحی تست مجزا برای این دستورات است :

```

#0 addi x1, x0, 2
#1 addi x2, x0, -3
#2 sra x2, x2, x1 // x2 = -1
#3 sw x2, 100(x0) // mem[100] = -1
#4 sll x2, x2, x1 // x2 = -4
#5 sw x2, 104(x0) // mem[104] = -4
#6 srl x2, x2, x1 // x2 = 001111 1111 1111 1111 1111 1111 11 = 0xFFFFFFFF
#7 sw x2, 108(x0) // mem[108] = 0xFFFFFFFF

```

کد های ماشین را در فایل جدایی shiftstest.txt مینویسیم و مسیر خوانش داده در memory را تغییر میدهیم :

```
module shiftTb();

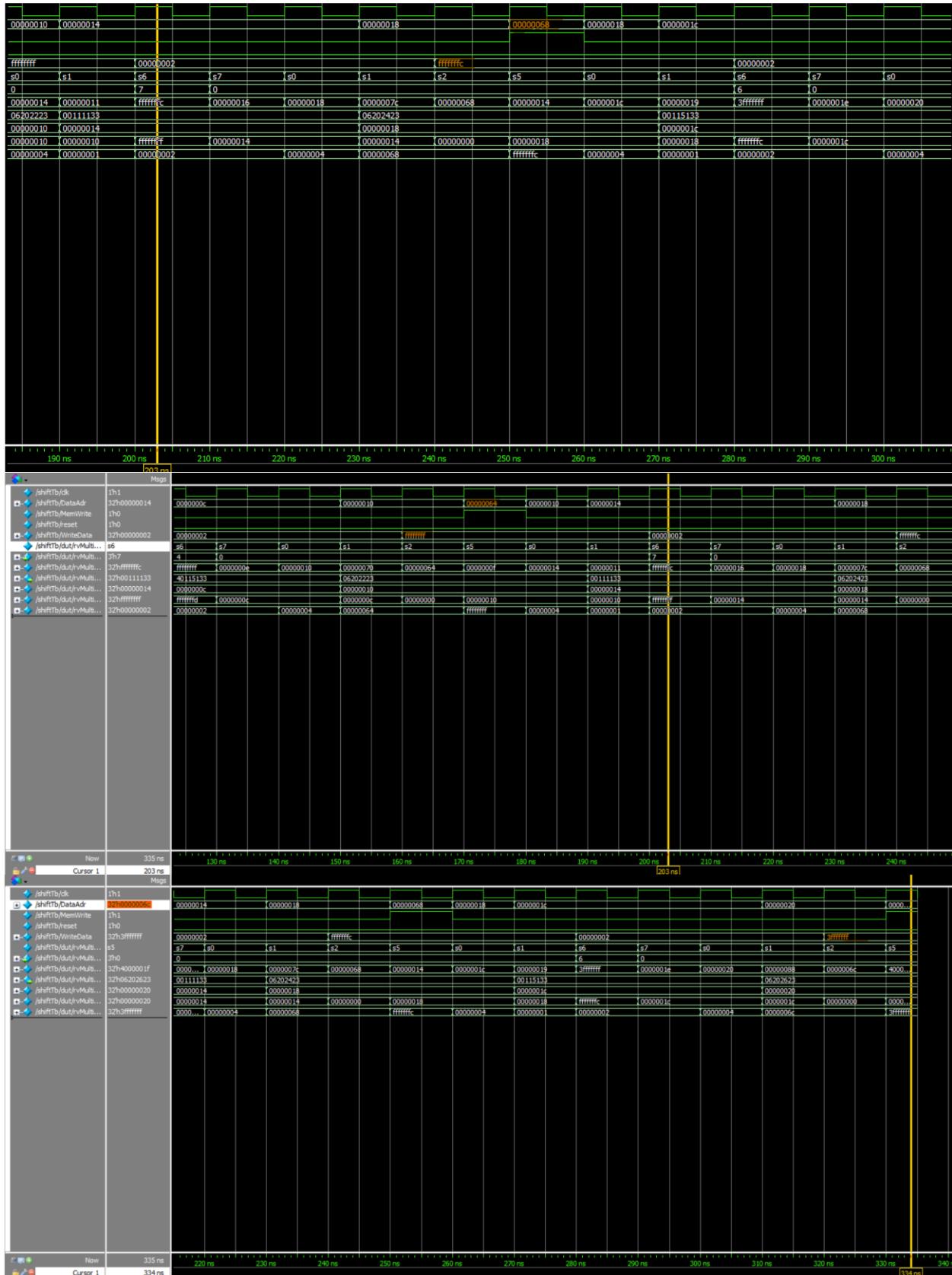
logic clk;
logic reset;
logic [31:0] WriteData, DataAddr;
logic MemWrite;

// instantiate device to be tested
top dut(clk, reset, WriteData, DataAddr, MemWrite);

// initialize test
initial
begin
    reset <= 1; # 22; reset <= 0;
end

// generate clock to sequence tests
always
begin
    clk <= 1; # 5; clk <= 0; # 5;
end
always @ (negedge clk) begin
    if (MemWrite)
        begin
            if (DataAddr == 108 & WriteData == 32'h3FFFFFFF)
                begin

                    $display("Simulation succeeded");
                    $stop;
                end
            else if (DataAddr != 100 & DataAddr != 104)
                begin
                    $display("Simulation failed");
                    $stop;
                end
        end
end
end
endmodule
```

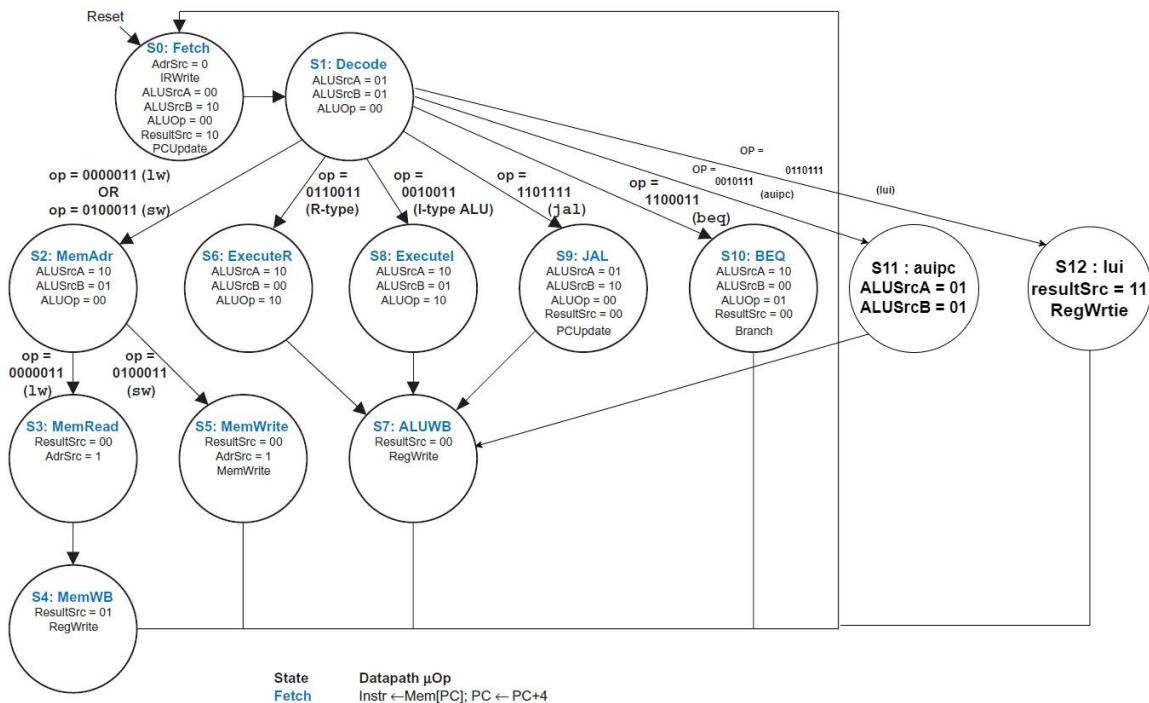


```

VSIM 71> run 3us
# Simulation succeeded
# ** Note: $stop      : C:/Users/Arsalan/Desktop/memProj/shiftTb.sv(29)
#   Time: 335 ns Iteration: 1 Instance: /shiftTb
# Break in Module shiftTb at C:/Users/Arsalan/Desktop/memProj/shiftTb.sv line 29

```

موفقیت آمیز بود . دستور بعدی که میتوان اضافه کرد lui هست اما برای این دستور نیاز داریم ماشین حالت را تغییر دهیم:



تغییر به این گونه است که ابتدا دستور در هنگام دیکوڈ شدن مقدار immediate را تبدیل به 32 بیت می کند ، سپس در سیکل بعد این مقدار در رجیستر مقصد نوشته میشود ، اما برای این کار نیازمند یک تغییر در مسیر داده هستیم :

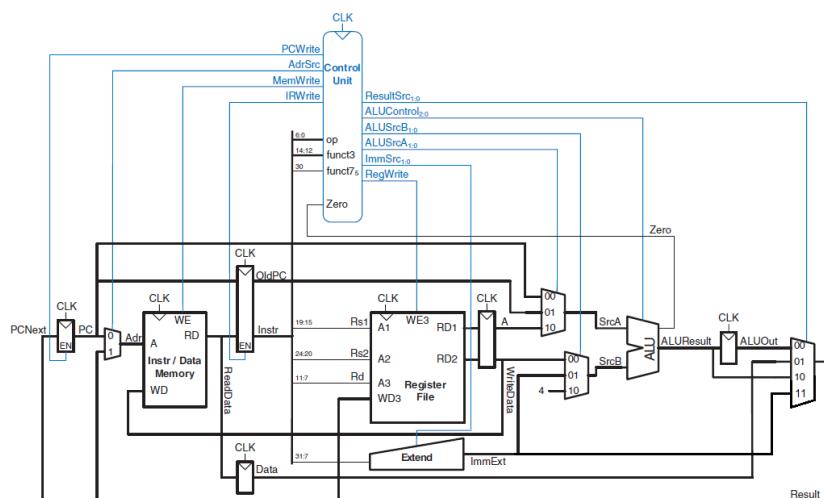


Figure 7.27 Complete multicycle processor

با توجه به تغییرات شکل گرفته ، مسیر داده و منطق واحد کنترل را تغییر میدهیم :

```
module mux4 #(parameter WIDTH = 8)
  (input logic [WIDTH-1:0] d0, d1, d2, d3,
   input logic [1:0] s,
   output logic [WIDTH-1:0] y);
  assign y = s[1] ? (s[0] ? d3 : d2):(s[0] ? d1 : d0 ) ;
endmodule

logic [31:0] Result , ALUOut, ALUResult;
logic [31:0] RD1, RD2, A , SrcA, SrcB, Data;
logic [31:0] ImmExt;
logic [31:0] PC, OldPC;

//pc
flopnr #(32) pcFlop(clk, reset, PCWrite, Result, PC);

//regFile
regFile rf(clk, RegWrite, instr[19:15], instr[24:20], instr[11:7], Result, RD1, RD2
extend ext(instr[31:7], ImmSrc, ImmExt);
flop #(32) regF( clk, reset, RD1, A);
flop #(32) regF_2( clk, reset, RD2, WriteData);

//alu
mux3 #(32) srcAmux(PC, OldPC, A, ALUSrcA, SrcA);
mux3 #(32) srcBmux(WriteData, ImmExt, 32'd4, ALUSrcB, SrcB);
alu alu(SrcA, SrcB, ALUControl, ALUResult, Zero, cout, overflow, sign);
flop #(32) aluReg( clk, reset, ALUResult, ALUOut);
mux4 #(32) resultMux(ALUOut, Data, ALUResult, ImmExt, ResultSrc, Result );

-----`nextstate = s1;
s1: case(op)
  7'b0110011: nextstate = s6;
  7'b0010011: nextstate = s8;
  7'b0000011: nextstate = s2;
  7'b0100011: nextstate = s2;
  7'b1100011: nextstate = s10;
  7'b1101111: nextstate = s9;
  7'b0010111: nextstate = s11;
  7'b0110111: nextstate = s12;
  7'b1100111: nextstate = s2;
  default: nextstate = s13;
endcase
```

```

module instrDec ( input logic [6:0] op,
                  output logic [2:0] immsrc);

  always_comb begin

    case(op)
      7'b0110011: immsrc = 3'b000;
      7'b0010011: immsrc = 3'b000;
      7'b0000011: immsrc = 3'b000;
      7'b0100011: immsrc = 3'b001;
      7'b1100011: immsrc = 3'b010;
      7'b1101111: immsrc = 3'b011;
      7'b0010111: immsrc = 3'b100;
      7'b0110111: immsrc = 3'b100;
      7'b1100111: immsrc = 3'b000;
      default: immsrc = 3'bx;
    endcase

  end

endmodule

```

لازم به ذکر است که برای ایجاد حالت جدید extend unit مجبور هستیم immsrc را 3 بیتی کنیم ، بعد از تغییرات ، دستورات branch را به پردازنده اضافه میکنیم :
برای این دستورات نیاز به دریافت یک سری flag از alu هستیم که بتوان مقایسه بین دو عملوند را بتوان تشخیص داد و نتیجه branch را به درستی مشخص کرد :

```

module alu (input logic [31:0] src1, src2, input logic [3:0] aluc, output logic [31:0] out, output logic zero,
            output logic cout, overflow, sign);
  always_comb begin
    case (aluc)
      4'b0000: begin
        {cout, out} = {1'b0, src1} + {1'b0, src2};
      end
      4'b0001: begin
        {cout, out} = {1'b0, src1} + {1'b0, ~src2} + 9'b1;
      end
      4'b0010: begin
        out = src1 & src2;
        cout = 0;
      end
      4'b0011: begin
        out = src1 | src2;
        cout = 0;
      end
      4'b0100: begin // sra
        out = $signed(src1) >> $unsigned(src2[4:0]);
        cout = 0;
      end
      4'b0101: begin
        if ($signed(src1) < $signed(src2))
          out = 1;
        else
          out = 0;
        cout = 0;
      end
      4'b0110: begin // sr1
        out = src1 >> src2[4:0];
        cout = 0;
      end
      4'b0111: begin // sll
        out = src1 << src2[4:0];
      end
    endcase
  end
endmodule

```

```

        out = src1 << src2[4:0];
        cout = 0;
    end
} 4'b1000: begin // sltu
    if($unsigned(src1) < $unsigned(src2))
        out = 1;
    else
        out = 0;
    cout = 0;
end
} 4'b1001: begin // xor
    out = src1 ^ src2;
    cout = 0;
end
} default: begin
    out = 0;
    cout = 0;
end
endcase
if (out==0)
    zero = 1;
else
    zero = 0;
sign = out[31];
overflow = (~((src1[31] ^ src2[31]) ^ aluc[0])) & (src1[31] ^ sign) & ~aluc[1];
end
endmodule

```

این flag ها را به مسیر داده و از آن به واحد کنترل منتقل میکنیم :

```

module controller(input logic clk,
                  input logic reset,
                  input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic funct7b5,
                  input logic zero, cout, overflow, sign,
                  output logic [2:0] immsrc,
                  output logic [1:0] alusrcA, alusrcB,
                  output logic [1:0] resultsrc,
                  output logic adrsrc,
                  output logic [3:0] alucontrol,
                  output logic irwrite, pcwrite,
                  output logic regwrite, memwrite);

logic beq, bne, blt, bge, bltu, bgeu, branch, pcupdate;
logic [1:0] aluop;

fsm MainFSM(clk, reset, op, branch, pcupdate, regwrite,
             memwrite, irwrite, resultsrc, alusrcB, alusrcA, adrsrc, aluop);
aluDec AluDecoder(aluop, op[5], funct7b5, funct3, alucontrol);
instrDec InstrDecoder(op, immsrc);
branchDec BranchDecoder(op, funct3, branch, beq, bne, blt, bge, bltu, bgeu);

assign pcwrite = (beq & zero) | (bne & ~zero) | (bgeu & cout) | (bltu & ~cout)
| (bge & (sign == overflow)) | (blt & (sign != overflow)) | pcupdate;

endmodule

```

برای اینکه ماشین حالت را تغییر ندهیم یک دیکود کننده branch میسازیم که زمانی که یک branch شروع به کار کرده و سیگنال های مورد نظر (beq, bne ...) راست میکند :

```

module branchDec(input logic [6:0] op,
                  input logic [2:0] funct3,
                  input logic branch,
                  output logic beq, bne, blt, bge, bltu, bgeu);

    assign beq = (op == 7'b1100011) & (funct3 == 3'b000) & branch ;
    assign bne = (op == 7'b1100011) & (funct3 == 3'b001) & branch ;
    assign blt = (op == 7'b1100011) & (funct3 == 3'b100) & branch ;
    assign bge = (op == 7'b1100011) & (funct3 == 3'b101) & branch ;
    assign bltu = (op == 7'b1100011) & (funct3 == 3'b110) & branch ;
    assign bgeu = (op == 7'b1100011) & (funct3 == 3'b111) & branch ;

endmodule

```

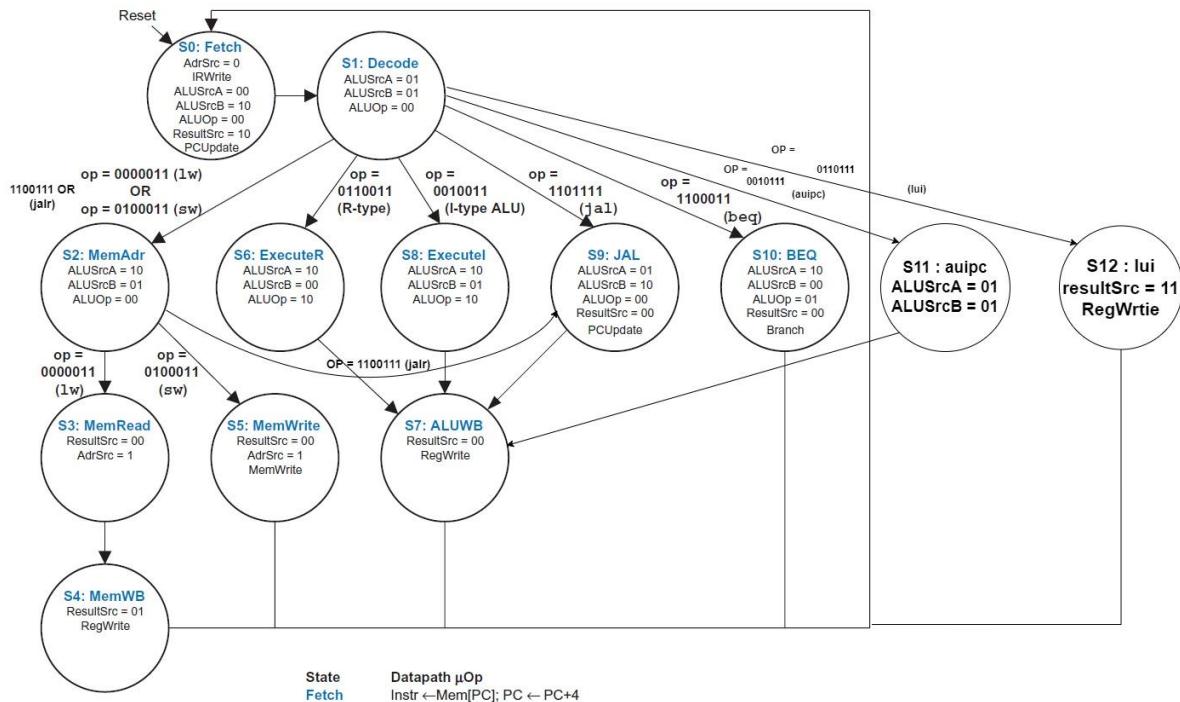
پس از آن با استفاده از این flag ها و سیگنال های branch نتیجه beq, bne ... مشخص میشود :

```

assign pcwrite = (beq & zero) | (bne & ~zero) | (bgeu & cout) | (bltu & ~cout)
| (bge & (sign == overflow)) | (blt & (sign != overflow)) | pcupdate;

```

بعد از این دستور ، میتوان jalr را هم اضافه کرد :



اول آدرس مقصد حساب میشود ، بعد وارد PC میشود بعد 4 PC + 4 در رجیستر مقصد نوشته میشود :

```

s2: if(op[5]) begin
    if(op[6])
        nextstate = s9;
    else
        nextstate = s5;
end
else

```

```

// nextstate logic
always_comb begin
    case(state)
        s0: nextstate = s1;
        s1: case(op)
            7'b0110011: nextstate = s6;
            7'b0010011: nextstate = s8;
            7'b0000011: nextstate = s2;
            7'b0100011: nextstate = s2;
            7'b1100011: nextstate = s10;
            7'b1101111: nextstate = s9;
            7'b0010111: nextstate = s11;
            7'b0110111: nextstate = s12;
            7'b1100111: nextstate = s2;
        default: nextstate = s13;
    endcase

```

پس از همه این تغییر ها دستورات `_type` خود به خود به طراحی اضافه میشوند چراکه هم مسیر داده و هم منطق واحد کنترل فراهم است فقط حال نوبت به تست این دستورات اضافه شده میرسد :

```

# Simulation succeeded
# ** Note: $stop      : C:/Users/Arsalan/Desktop/memProj/finalTb.sv(29)
#   Time: 1025 ns  Iteration: 1  Instance: /finalTb
# Break in Module finalTb at C:/Users/Arsalan/Desktop/memProj/finalTb.sv line 29
# UOTM EAN 1

```

تست اصلی موفقیت آمیز بود :

```
1      j start
2 labelEr: addi x1, x0, 0 # x1 = 0
3          sw x1, 252(x0) # put the wrong value in the memory
4 start:  lui x1, 1 # x1 = 4096
5          addi x2, x0, 2 #x2 = 2
6          addi x3, x0, -3 #x3 = -3
7          bge x2, x3, label1 #should be taken
8          addi x1, x0, 0 # x1 = 0
9          sw x1, 252(x0) # put the wrong value in the memory
10 label1: sltu x4, x2, x3 # x4 = 1
11         bne x4, x2, label2 #should be taken
12         addi x1, x0, 0 #x1 = 0
13         sw x1, 252(x0) # put the wrong value in the memory
14 label2: xor x5, x2, x3 #x5 = -1
15         blt x5, x3, labelEr #should not be taken
16         sltu x6, x5, x3 # x6 = 0
17         bltu x2, x5, label3 #should be taken
18         addi x1, x0, 0 # x1 = 0
19         sw x1, 252(x0) # put the wrong value in the memory
20 label3: slli x7, x5, 4 #x7 = -16
21         bgeu x5, x7, label4 # should be taken
22         addi x1, x0, 0 # x1 = 0
23         sw x1, 252(x0) # put the wrong value in the memory
```

```

24 label4: srai x1, x1, 2 # x1 = 1024
25     sltiu x1, x1, -2048 # x1 = 1
26     slli x1, x1, 13 # x1 = 8192
27     srli x1, x1, 1 # x1 = 4096
28     srai x7, x7, 4 # x7 = -1
29     xori x7, x7, -1 # x7 = 0
30     addi x7, x7, 176
31     bge x5, x5, label5 #should be taken
32     addi x1, x0, 0 # x1 = 0
33     sw x1, 252(x0) # put the wrong value in the mamory
34 label5: bge x3, x0, labelEr #should not be taken
35     bgeu x5, x5, label6 #should be taken
36     addi x1, x0, 0 # x1 = 0
37     sw x1, 252(x0) # put the wrong value in the mamory
38 label6: blt x5, x0, label7 #should be taken
39     addi x1, x0, 0 # x1 = 0
40     sw x1, 252(x0) # put the wrong value in the mamory
41 label7: bltu x5, x0, labelEr
42     bgeu x0, x5, labelEr #both should not be taken
43     jalr x8, x7, 0
44     addi x1, x0, 0 # x1 = 0
45     sw x1, 252(x0) # if x1 == 4096 => correct else not correct

```

در این تست سعی شده همه حالت ها در نظر گرفته شود و بررسی شود ، کد های ماشین را در مسیر finaltest.txt مینویسیم و مسیر memory را تغییر میدهیم در نهایت با بررسی نتیجه تست بنج و مقدار رجیستر ها درستی طراحی را نتیجه میگیریم :

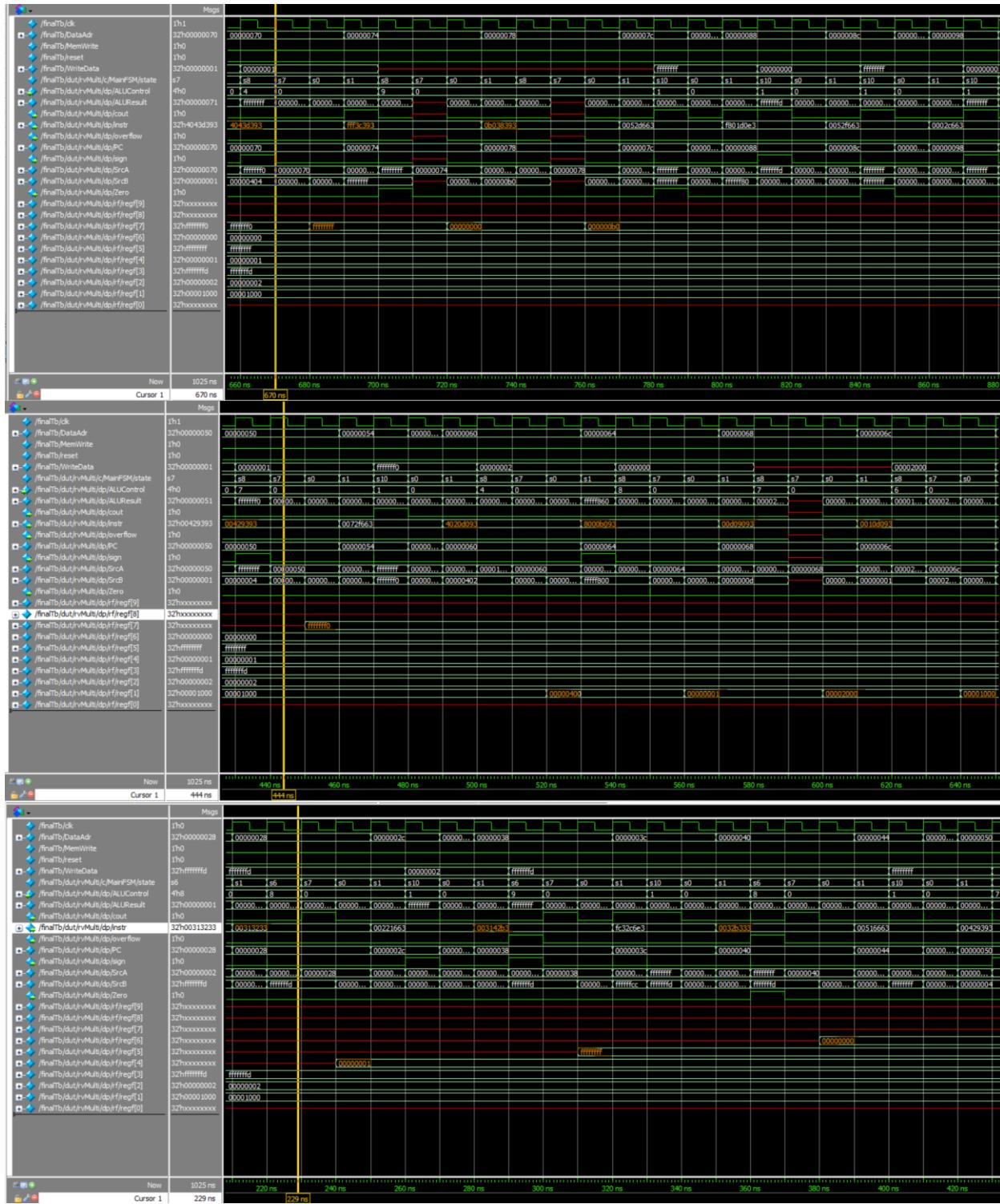
```
module finalTb();

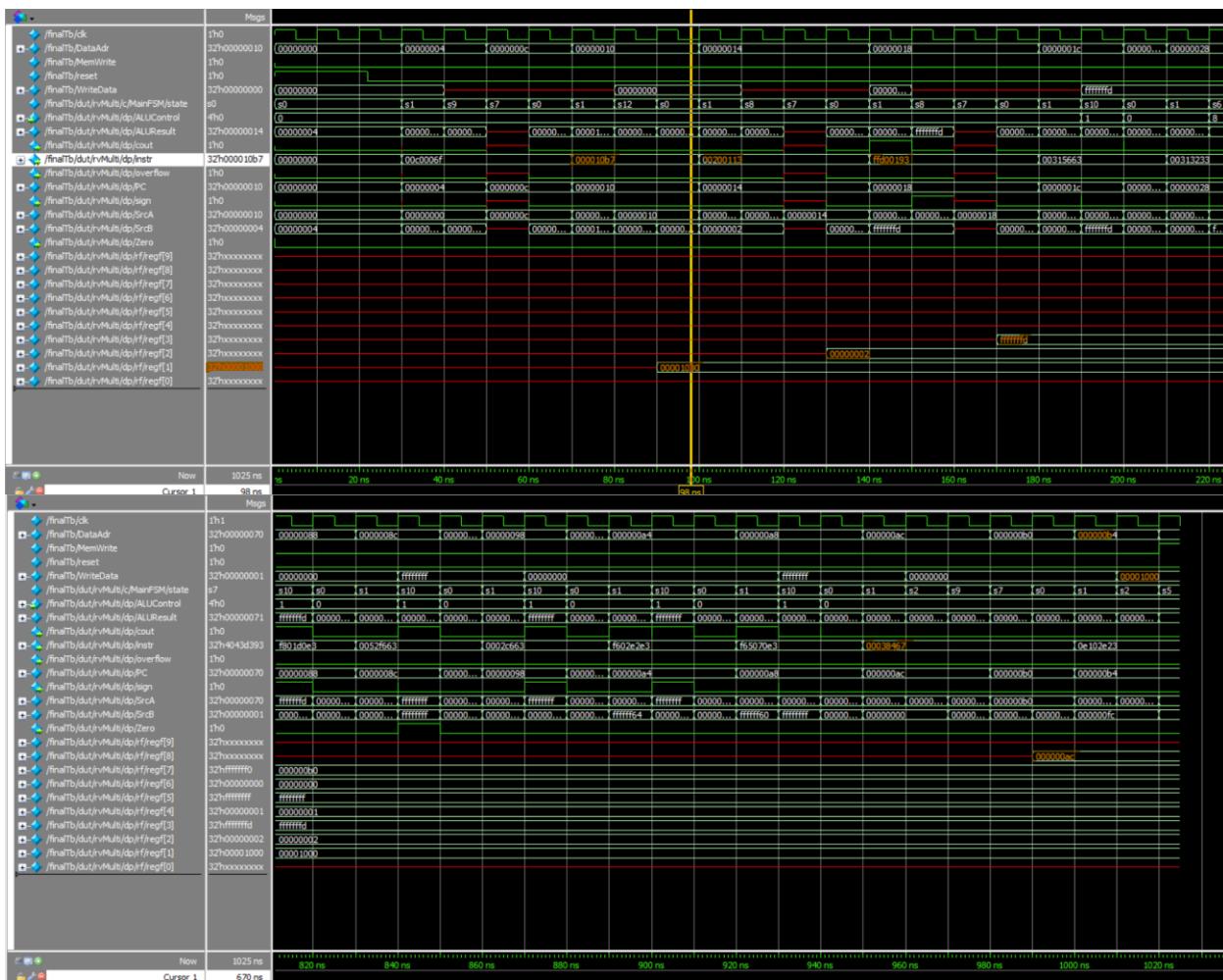
logic clk;
logic reset;
logic [31:0] WriteData, DataAddr;
logic MemWrite;

// instantiate device to be tested
top dut(clk, reset, WriteData, DataAddr, MemWrite);

// initialize test
initial
begin
    reset <= 1; # 22; reset <= 0;
end

// generate clock to sequence tests
always
begin
    clk <= 1; # 5; clk <= 0; # 5;
end
always @{negedge clk} begin
    if(MemWrite)
        begin
            if(DataAddr == 252 & WriteData == 32'h00000100)
                begin
                    $display("Simulation succeeded");
                    $stop;
                end
            else
                begin
                    $display("Simulation failed");
                    $stop;
                end
        end
end
end
endmodule
```





```

# Simulation succeeded
# ** Note: $stop      : C:/Users/Arsalan/Desktop/memProj/finalTb.sv(29)
#   Time: 1025 ns Iteration: 1 Instance: /finalTb
# Break in Module finalTb at C:/Users/Arsalan/Desktop/memProj/finalTb.sv line 29

```

LATCHES 1

پس طراحی درست انجام شده است.

پایان