به نام خدا

آزمایش پنجم _ SingleCycle

امین ساوه دورودی – محمد رضا اسکینی – کامیاب عابدی

شرح اولیه

هدف از این آزمایش, پیاده سازی پردازنده ساده تک چرخه ای است. شماتیک پردازنده ساده تک چرخه ای در تصویر زیر (1) نشان داده شده است.

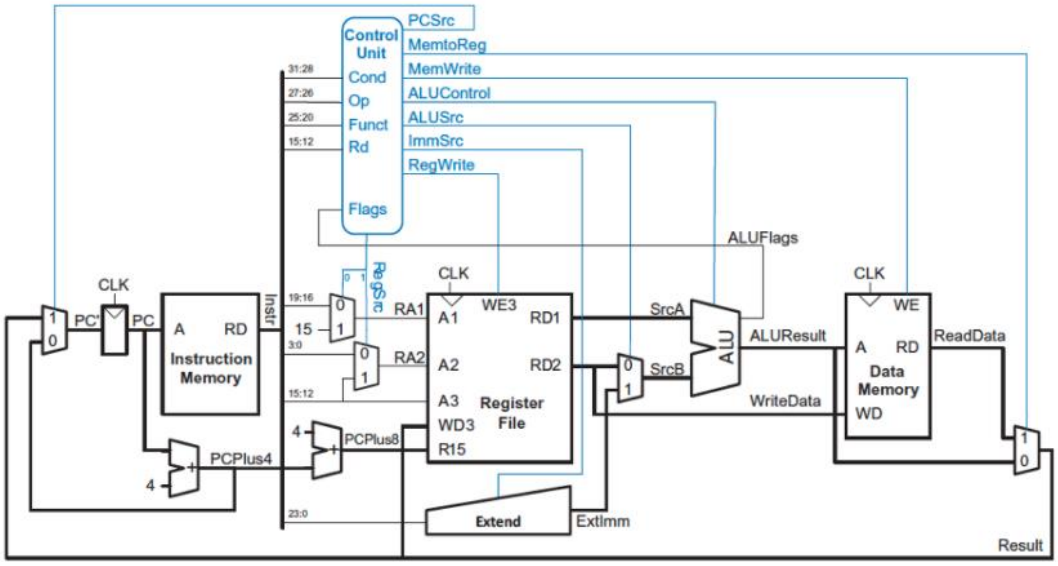


Figure 7.13 Complete single-cycle processor

تصویر 1 – پردازنده ساده تک چرخه ای

پردازنده ساده تک چرخه ای از دو بخش کنترل(Control) و بخش مسیر داده (DataPath) تشکیل شده است. برای اجرای دستورات در این پردازنده ابتدا نیاز است به صورت تک به تک از حافظه انتخاب شوند که این مرحله به کمک حافظه دستور (Instruction Memory) انجام میشود. همچنین برای دسترسی آدرس از داده ها از حافظه داده (Data Memory) استفاده میشود. واحد کنترل نیز وظیفه مشخص کردن نوع عملکرد هر بخش را با استفاده از سیگنال هایی بر عهده دارد.

پردازنده ساده تک چرخه ای هر دستور را در یک چرخه اجرا میکند. پردازنده ساده تک چرخه ای, ساده ترین نوع پردازنده میباشد. زمان هر چرخه توسط طولانی ترین دستور (در اینجا LDR) مشخص میشود که به همین سبب باعث میشود نسبت به دیگر پردازنده های پیشرفته تر, در مدت زمان بیشتری تعداد دستورات واحد را اجرا کند. در ادامه به پیاده سازی ماژول های این پردازنده می پردازیم. و در آخر پردازنده پیاده سازی شده را توسط تست بنچ هایی مورد آزمایش قرار می دهیم.

پیاده سازی

ماژول top

```
module top(
	input clk , reset,
	output [15:0] Instr,
	output [3:0] Flags,
	output [7:0] ALUResult,
	output [7:0] SrcA, SrcB,
	output PCSrc
	);
	wire [7:0] ReadData;
	wire [7:0] PC;
	wire [7:0] WriteData , DataAdr;
	//ReadData = Data from memory
	//WriteData = Data from second port of register file
	//Control = ALUControl
	//Flags = ALUFlags
	cpu cpu(clk, reset, PC, Instr, DataAdr, ALUResult, WriteData, ReadData, Flags, SrcA, SrcB, PCSrc);
	imem imem(PC, Instr);
	dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

این ماژول, وظیفه نمونه گیری از ماژول های اساسی پردازنده را بر عهده دارد. این ماژول به عنوان ماژول مادر شناخته میشود که در آن یک ماژول cpu که شامل Datapath و Controller میباشد, نمونه گیری شده است. همچنین برای بارگذاری دستور ها نیاز به یک imem(Instruction Memory) است و در اخر برای پیدا کردن ادرس داده ها بر روی حافظه نیاز به یک dmem(Data Memory) میباشد.

ماژول imem

```
module imem(
            input [7:0] a,
            output [15:0] rd
            );
            reg [15:0] RAM[255:0];
            initial
            $readmemh("C:/Users/pixel/Desktop/src/memfile.dat",RAM);
            assign rd = RAM[a[7:2]];
endmodule
```

این ماژول, وظیفه خواندن دستور ها از فایل و تحویل آن به سیگنال Instr واقع در ماژول top را برعهده دارد. همانطور که در گزارش کار آمده است, حافظه (RAM) 256 خانه 16 بیتی دارد. همچنین سیگنال PC که نشان دهنده Program Counter برنامه است, هشت بیتی میباشد که دو بیت اول آن به صورت Word Offset هستند و از آنجایی که نیازی به آن نداریم پس برای هر بارگذاری هر دستور تنها به بیت های 7:2, PC نیاز داریم. همچنین دستور ها در فایلی با پسوند dat. به صورت هگزادسیمال قرار داده میشوند که از طریق دستور readmemh میتوان دستورات را خواند و به RAM انتقال داد.

ماژول dmem

```
module dmem(
            input clk, we,
            input [7:0] a, wd,
            output [7:0] rd
            );
  reg [7:0] RAM[255:0];
  assign rd = RAM[a[7:2]];
  always @(posedge clk)
    if(we) RAM[a[7:2]] <= wd;
endmodule
```

از این ماژول, برای پیدا کردن داده های مورد نظر در حافظه با استفاده از آدرس شان استفاده میشود. تمامی موارد ماژول imem نیز در این ماژول به همانگونه وجود دارد. همچنین با توجه به سیگنال we, امکان نوشتن داده بر روی آدرسی از حافظه نیز برقرار است.

ماژول cpu

```
module cpu(
		input clk, reset,
		output [7:0] PC,
		input [15:0] Instr,
		output [7:0] DataAdr,
		output [7:0] ALUResult, WriteData,
		input [7:0] ReadData,
		output [3:0] FLAGS,
		output [7:0]  SrcA, SrcB,
		output PCSrc
		);
		wire [7:0] ALUResult2;
		wire RegWrite, RegWrite2;
		wire [4:0] ALUControl;
		wire ALUSrc;
		wire LM, LI;
		wire ImmSrc;
		wire [3:0] ALUFlags;
		controller c(clk, reset, Instr, ALUFlags, RegWrite, RegWrite2, ALUSrc, ALUControl, PCSrc, LM,ImmSrc, LI);
		datapath dp(clk, reset, RegWrite, RegWrite2, ALUSrc, ALUControl,
				PCSrc, ALUFlags, PC, Instr, ALUResult, ALUResult2, WriteData,
				LM, DataAdr, ReadData, ImmSrc, SrcA , SrcB, LI);
		assign FLAGS = ALUFlags;
endmodule
```

این ماژول, ماژول اصلی پردازنده ما میباشد که همانطور که در شرح اولیه ذکر شده است, از دو بخش کنترل و مسیر داده تشکیل شده است. با توجه به شماتیک کلی هر بخش(تصویر 1) سیگنال های ورودی و خروجی مشخص شده است.

ماژول datapath

```
module datapath(
		input clk, reset,
		input RegWrite, RegWrite2,
		input ALUSrc,
		input [4:0] ALUControl,
		input PCSrc,
		output [3:0] ALUFlags,
		output [7:0] PC,
		input [15:0] Instr,
		output [7:0] ALUResult, ALUResult2, WriteData,
		input LM,
		output [7:0] DataAdr,
		input [7:0] ReadData,
		input ImmSrc,
		output [7:0]  SrcA, SrcB,
		input LI
		);
		wire [7:0] PCNext, PCPlus4;
		wire [7:0] Result;
		wire [7:0] ExtImm;
		wire [2:0] destinationReg;
		// next PC logic
		mux2 #(8) pcmux(PCPlus4, Instr[7:0], PCSrc, PCNext);
		flopr #(8) pcreg(clk, reset, PCNext, PC);
		adder #(8) pcadd1(PC, 8'b100, PCPlus4);
		// register file logic
		mux2 #(3) destinationMux(Instr[5:3], Instr[10:8], LI, destinationReg);
		regfile rf(clk, RegWrite, RegWrite2, destinationReg, Instr[2:0], Result, ALUResult2, SrcA, WriteData);
		extend ext(Instr[2:0], Instr[7:0], ImmSrc, ExtImm);
		mux2 #(8) resmux(ALUResult, ReadData, LM, Result);
		// ALU logic
		mux2 #(8) srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
		alu alu(ALUControl, SrcA, SrcB, ALUResult, ALUResult2, ALUFlags);
		//For LM instruction
		mux2 #(8) DataAdrMux(8'b00000000 , Instr[7:0], LM , DataAdr);
endmodule
```

این ماژول وطیفه اجرای عملیات های خواسته شده در دستور را بر عهده دارد. همانطور که در تصویر(1) مشخص است, نیاز به مالتی پلکسر و جمع کننده و regfile و ALU و extend و .. داریم. که با همگام با تصویر(1) پیش میرویم و تک به تک آن ها را پیاده سازی میکنیم و سیگنال ها را مرتبط با ورودی و خروجی هر کدام به آن ها میدهیم.

ماژول regfile

```
module regfile(
        input clk,
        input we3, we4,
        input [2:0] ra1, ra2,
        input [7:0] wd3, wd4,
        output [7:0] rd1 , rd2
        );
       reg [7:0] rf[7:0];
        always @(posedge clk)
        begin
                if (we3) rf[ra1] <= wd3;
                if (we4)    rf[ra2] <= wd4;
        end
        assign rd1 =  rf[ra1] ;
        assign rd2 =  rf[ra2] ;
endmodule
```

این پردازنده 8 رجیستر دارد که در این ماژول نگه داری میشوند. این ماژول وظیفه نگهداری رجیستر ها و نوشتن و خواندن مقادیر رجیستر ها را بر عهده دارد.

ماژول ALU

```
module alu (
        input [4:0] ALUControl,
        input [7:0] srcA, srcB,
        output reg [7:0] ALUResult,
        output reg [7:0] ALUResult2,
        output reg [3:0] ALUFlags
        );
        reg xn;
        always @(ALUControl, srcA, srcB)
        case(ALUControl)
                //7-XCHG
                5'b00111: ALUResult2 <= srcA;
                default: ALUResult2 <= 8'b0;
        endcase
        always @(*)
        case(ALUControl)
                //1-ADD
                5'b00001:
                begin
                        {ALUFlags[1], ALUResult} <= srcA + srcB;
                        ALUFlags[3] <= ALUResult[7];
                        ALUFlags[2] <= ~(|ALUResult);
                        xn <= ~(1'b0 ^ srcA[7] ^ srcB[7]);
                        ALUFlags[0] <= ((ALUResult[7] ^ srcA[7]) & xn);
                end
```

```
        5'b00010: //AND
                begin
                        ALUResult <= srcA & srcB ;
                        ALUFlags[3] <= ALUResult[7];
                        ALUFlags[2] <= ~(|ALUResult);
                        ALUFlags[1] <= 1'b0;
                        ALUFlags[0] <= 1'b0;
                end
                5'b00011: //SUB
                begin
                        {ALUFlags[1], ALUResult} <= srcA - srcB;
                        ALUFlags[3] <= ALUResult[7];
                        ALUFlags[2] <= ~(|ALUResult);
                        xn <= ~(1'b1 ^ srcA[7] ^ srcB[7]);
                        ALUFlags[0] <= ((ALUResult[7] ^ srcA[7] ) & xn);
                end
                5'b00100: //OR
                begin
                        ALUResult <= srcA | srcB;
                        ALUFlags[3] <= ALUResult[7];
                        ALUFlags[2] <= ~(|ALUResult);
                        ALUFlags[1] <= 1'b0;
                        ALUFlags[0] <= 1'b0;
                end
```

در این ماژول, با توجه به سیگنال های دریافتی از بخش کنترل, نوع عملیات منطقی مشخص میشود و در این ماژول عملیات های منطقی بر روی داده های ورودی اعمال میشوند. همچنین با توجه به جدول گزارش کار , هر عملیات فلگ های به خصوصی را بروز رسانی میکند که در این ماژول با توجه به جدول, فلگ های مورد نظر نیز بروزرسانی میشوند.

(تنها بخشی از کد ALU به عنوان نمونه آورده شده است)

ماژول mux2

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

این ماژول همان مالتی پلکسر هست که یک ورودی به عنوان سیگنال کنترل میگیرد و با توجه به مقدار سیگنال کنترل, خروجی مشخص میشود. برای شرط از شرط سه عملوندی (ternary operator) استفاده شده است. از این ماژول در تمام جاهایی که در تصویر(1) از مالتی پلکسر استفاده شده است, استفاده میشود.

ماژول adder

```
module adder #(parameter WIDTH = 8)(
        input  [WIDTH-1:0] a , b,
        output  [WIDTH-1:0] y
        );
       assign y = a + b ;
endmodule
```

یک ماژول جمع کننده ساده است که از آن برای محاسبه PC+4 که در تصویر (1) نمایان است, استفاده میشود.

ماژول extend

```
module extend(
            input [2:0] inputImm,
            input [7:0] ImmLi,
            input ImmSrc,
            output reg [7:0] outputImm
            );
  always @(ImmSrc, ImmLi, inputImm)
  case(ImmSrc)
    1'b0: outputImm = {5'b0 , inputImm};
    1'b1: outputImm = ImmLi ;
    default: outputImm =  7'bx ;
  endcase
endmodule
```

مقادیر ثابت ما حداکثر هشت بیتی میباشند که با توجه جدول دستورکار آزمایش میتوانند سه بیتی یا هشت بیتی باشند. این ماژول, وظیفه آن را دارد که با توجه به دستور, خروجی را به عنوان هشت بیتی به بیرون پاس دهد.

ماژول controller

واحد کنترل از ماژول های مختلفی تشکیل شده است که در تصویر(2), شماتیک آن آورده شده است.

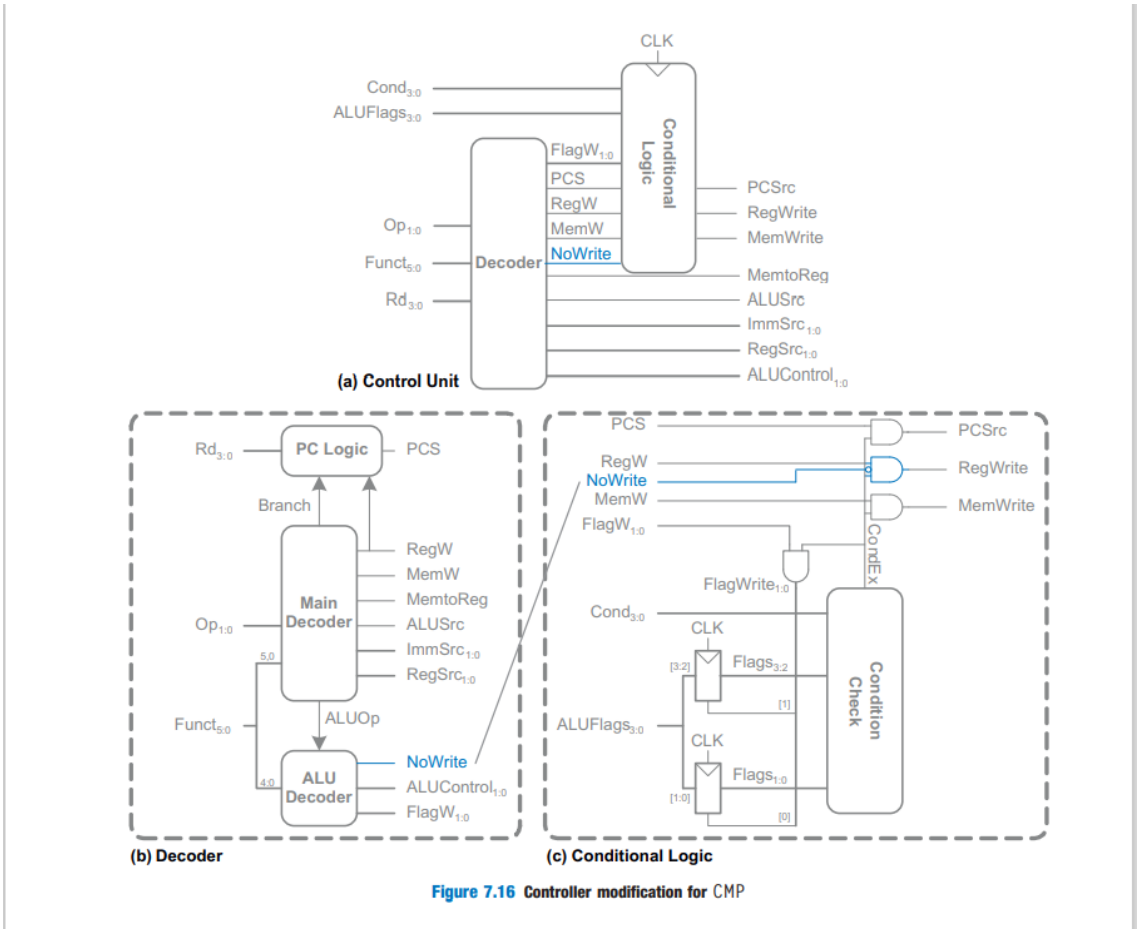


Figure 7.16 Controller modification for CMP

تصویر2 – بخش کنترل پردازنده تک چرخه ای

این ماژول با تحلیل دستور در ابتدا , سیگنال و بیت های مورد نیاز هر ماژول رو در اختیارش میگذارد و در آخر خروجی این ماژول سیگنال های کنترلی را به بخش datapath میفرستد تا مشخص شود هر ماژول datapath به چه نحوی عمل کند.

```
module controller(
        input clk, reset,
        input [15:0] Instr,
        input [3:0] ALUFlags,
        output RegWrite, RegWrite2,
        output ALUSrc,
        output [4:0] ALUControl,
        output PCSrc,
        output LM,
        output ImmSrc,
        output LI);
        wire pcS;
        wire [1:0] FlagW;
        wire [2:0] branchType;
        decoder dec(Instr[15], Instr[14:6], FlagW, pcS, RegWrite, RegWrite2, ALUSrc,  ALUControl, branchType, LM, ImmSrc,
LI);
        condlogic cl(clk, reset, ALUFlags, FlagW, pcS, branchType, PCSrc);
endmodule
```

در این ماژول طبق تصویر(2) ابتدا نمونه ای از ماژول decoder و condlogic ساخته میشود و سیگنال های ورودی و خروجی متناظر با هر کدام از ماژول ها به آنها داده میشود.

ماژول decoder خود از سه بخش تشکیل شده است. تصویر(2)

بخش main decoder

```
always @(Op, Funct)
        case(Op)
                1'b0:
                begin
                        // CMP NOP ShowR ShowRSegment
                        if(Funct[4:0] == 5'b10100 | Funct[4:0] == 5'b00000 |
                                Funct[4:0] == 5'b10010 | Funct[4:0] == 5'b10011)
                                controls <= 10'b1000000000 ;
                        // Immidiate instructions INC DEC
                        else if (Funct[3]) controls <= 10'b1110000000;
                        //XCHG
                        else if(Funct[4:0] == 5'b00111) controls <= 10'b1011000000;
                        // Register Instructions
                        else controls <= 10'b1010000000;
                end
                2'b1:
                begin
                        if(Funct[8:5] == 4'b0000) controls <= 10'b0000001010;   // JE
                        else if(Funct[8:5] == 4'b0001) controls <= 10'b0000010010; // JB
                        else if(Funct[8:5] == 4'b0010) controls <= 10'b0000011010; // JA
                        else if(Funct[8:5] == 4'b0011) controls <= 10'b0000100010; // JL
                        else if(Funct[8:5] == 4'b0100) controls <= 10'b0000101010; // JG
                        else if(Funct[8:5] == 4'b0101) controls <= 10'b0000110010; // JMP
                        else if(Funct[8:5] == 4'b0110) controls <= 10'b1110000011; // LI
                        else if(Funct[8:5] == 4'b0111) controls <= 10'b1010000110; // LM
                        else controls <= 10'bx;
                end
                // Unimplemented
                default: controls <= 10'bx;
        endcase
        assign {ALUOp, ALUSrc, RegWrite, RegWrite2, branchType, LM, ImmSrc, LI} = controls;
```

این بخش با توجه به نوع دستور سیگنال های کنترلی را تعیین میکند.

بخش alu decoder

```
always @(*)
                if(ALUOp & Op & Funct[8:5] == 4'b0110) //LI   begin
                        ALUControl = 5'b10101;
                        FlagW = 2'b00;              end
                else if(ALUOp & Op & Funct[8:5] == 4'b0111) //LM
                begin
                        ALUControl = 5'b10110 ;
                        FlagW = 2'b00;
                end
                else if (ALUOp)
                begin
                        case(Funct[4:0])
                                5'b00001: ALUControl = 5'b00001;    //ADD
                                5'b00010: ALUControl = 5'b00010;    //AND
                                5'b00011: ALUControl = 5'b00011;    //SUB
                                5'b00100: ALUControl = 5'b00100;    //OR
                                5'b00101: ALUControl = 5'b00101;    //XOR
                                5'b00110: ALUControl = 5'b00110;    //MOV
                                5'b00111: ALUControl = 5'b00111;    //XCHG
                                5'b01000: ALUControl = 5'b01000;    //NOT
                                5'b01001: ALUControl = 5'b01001;    //SAR
                                5'b01010: ALUControl = 5'b01010;    //SLR
                                5'b01011: ALUControl = 5'b01011;    //SAL
                                5'b01100: ALUControl = 5'b01100;    //SLL
                                5'b01101: ALUControl = 5'b01101;    //ROL
                                5'b01110: ALUControl = 5'b01110;    //ROR
                                5'b01111: ALUControl = 5'b01111;    //INC
                                5'b10000: ALUControl = 5'b10000;    //DEC
                                5'b00000: ALUControl = 5'b00000;    //NOP
                                5'b10010: ALUControl = 5'b10010;    //ShowR
                                5'b10011: ALUControl = 5'b10011;    //ShowRSeg
                                5'b10100: ALUControl = 5'b10100;    //CMP
                                default:  ALUControl = 5'b00000;    //unimplemented  endcase
```

در این بخش با توجه به نوع دستور, دستور کنترلی ALU مشخص میشود که مشخص میکند چه نوع عملیاتی انجام شود.

```
// Flag[1] is SF and ZF
                    // Flag[0] is CF and OF
                    //SF = negative
                    //ZF = zero
                    //CF = carry
                    //OF = overflow
                    FlagW[1]= ALUControl == 5'b00001 |  ALUControl == 5'b00011 |
                                ALUControl == 5'b00010 |  ALUControl == 5'b00100 |
                               ALUControl == 5'b00101 |  ALUControl == 5'b01001 |
                              ALUControl == 5'b01010 |  ALUControl == 5'b01011 |
                             ALUControl == 5'b01100 |  ALUControl == 5'b01101 |
                           ALUControl == 5'b01110 |  ALUControl == 5'b01111 |
                         ALUControl == 5'b10000 |  ALUControl == 5'b10100 ;
                    FlagW[0]= ALUControl == 5'b00001 |  ALUControl == 5'b00011 |  ALUControl == 5'b00010 |
                                ALUControl == 5'b00100 |              ALUControl == 5'b00101 |
                               ALUControl == 5'b01001 |  ALUControl == 5'b01010 |
                              ALUControl == 5'b01011 |     ALUControl == 5'b01100 |
                             ALUControl == 5'b01101 |  ALUControl == 5'b01110 |
                           ALUControl == 5'b01111 |        ALUControl == 5'b10000|
 ALUControl == 5'b10100 ;  end  else  begin
            ALUControl = 5'b0001;// add for non-DP instructions
                               FlagW = 2'b00;// don't update Flags
                    end
//PC logic
  assign PCS = (Op & Funct[8:5] != 4'b0110 & Funct[8:5] != 4'b0111);
```

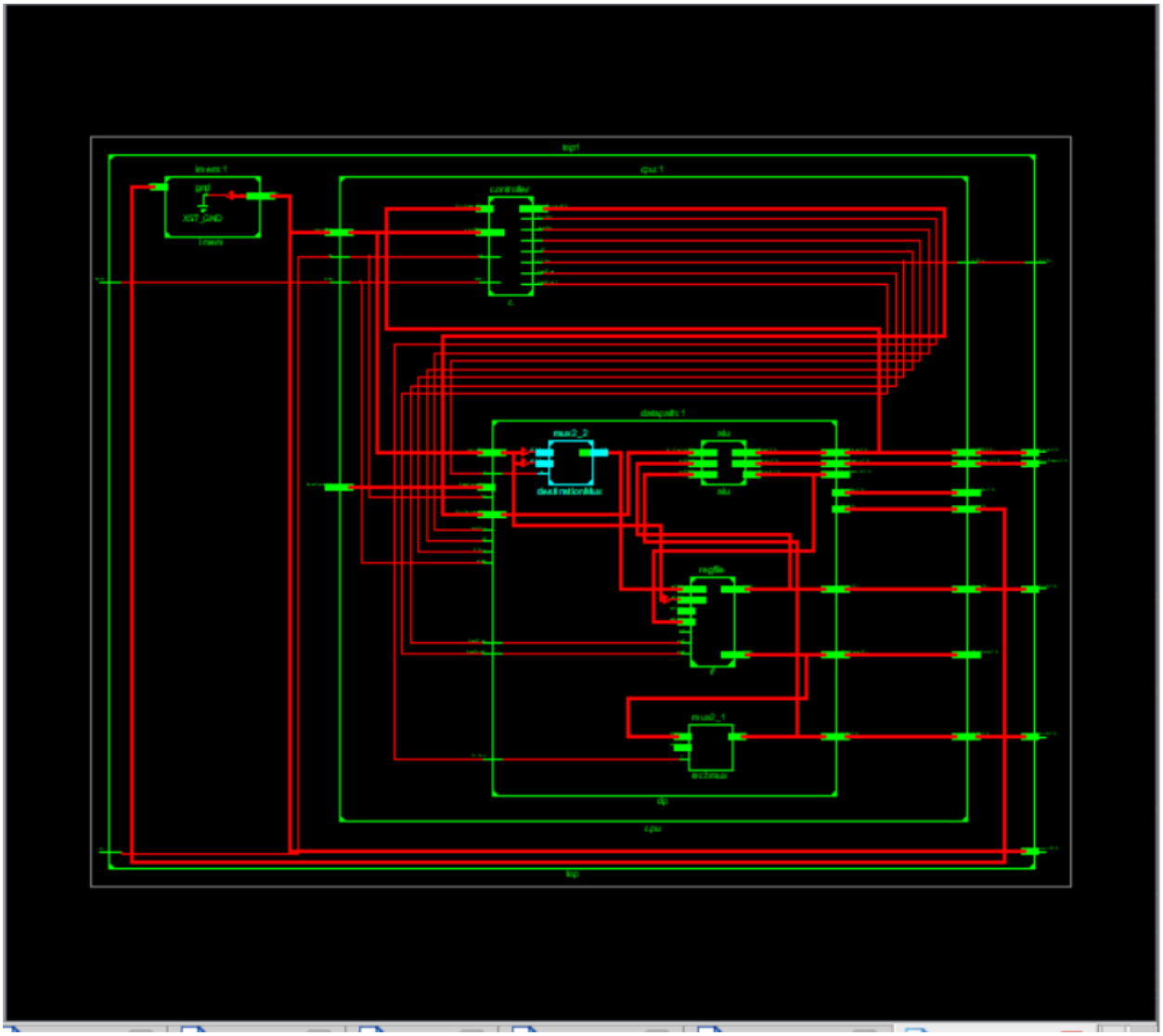
همانطور که در دستورکار آمده است, فلگ هایی داریم که با توجه به نوع عملیات , آپدیت میشوند یا خیر. این تکه کد از بخش alu decoder تعیین کننده آپدیت شدن فلگ ها است. همچنین در آخر نیز بخش pc logic پیاده سازی شده است که با توجه به نوع دستور میتواند 0 یا 1 باشد.

ماژول condlogic

```
module condlogic(
		input clk, reset,
		input [3:0] AluFlags,
		input [1:0] FlagW,
		input pcS,
		input [2:0] branchType,
		output pcSrc
		);
  wire [3:0] Flags;
  wire CondEx;
  flopenr #(2) flagreg1(clk, reset, FlagW[1],
			AluFlags[3:2], Flags[3:2]);
  flopenr #(2) flagreg0(clk, reset, FlagW[0],
			AluFlags[1:0], Flags[1:0]);
  condcheck cc(branchType , Flags , CondEx);
  assign pcSrc = pcS & CondEx ;
endmodule
module condcheck(
		input [2:0] branchType,
		input [3:0] AluFlags,
		output reg CondEx
		);
		always @(*)
		case(branchType)
				// JE  3'b001: CondEx = AluFlags[2];
				// JB 3'b010: CondEx = AluFlags[1];
				// JA 3'b011: CondEx= ( (~AluFlags[1]) & (~AluFlags[2]) );
				//JL 3'b100: CondEx = (AluFlags[1] == ~AluFlags[2]) ? 1'b1 : 1'b0 ;
				//JG  3'b101: CondEx =( (AluFlags[3] == AluFlags[0]) &  (~AluFlags[2]) );
				//JMP   3'b110: CondEx = 1'b1 ;
				default: CondEx = 1'b0 ;
		endcase
endmodule
```

در این ماژول با استفاده از فلگ های گرفته شده از بخش alu decoder, مشخص میشود که آیا در دستور شرطی وجود داشته است و آیا شرط ها برقرار هستند یا خیر. با چک کردن فلگ ها, تعیین میشود که چه سیگنال های کنترلی باید فعال باشند که مشخص کنند مسیر اجرایی پردازنده چگونه باشد.

ماژول های توضیح داده شده در بخش بالا تشکیل دهنده پردازنده ساده تک چرخه ای ما میباشند.



تصویر3- RTL

تصویر(3) شماتیک کلی پردازنده پیاده سازی شده را نشان میدهد.

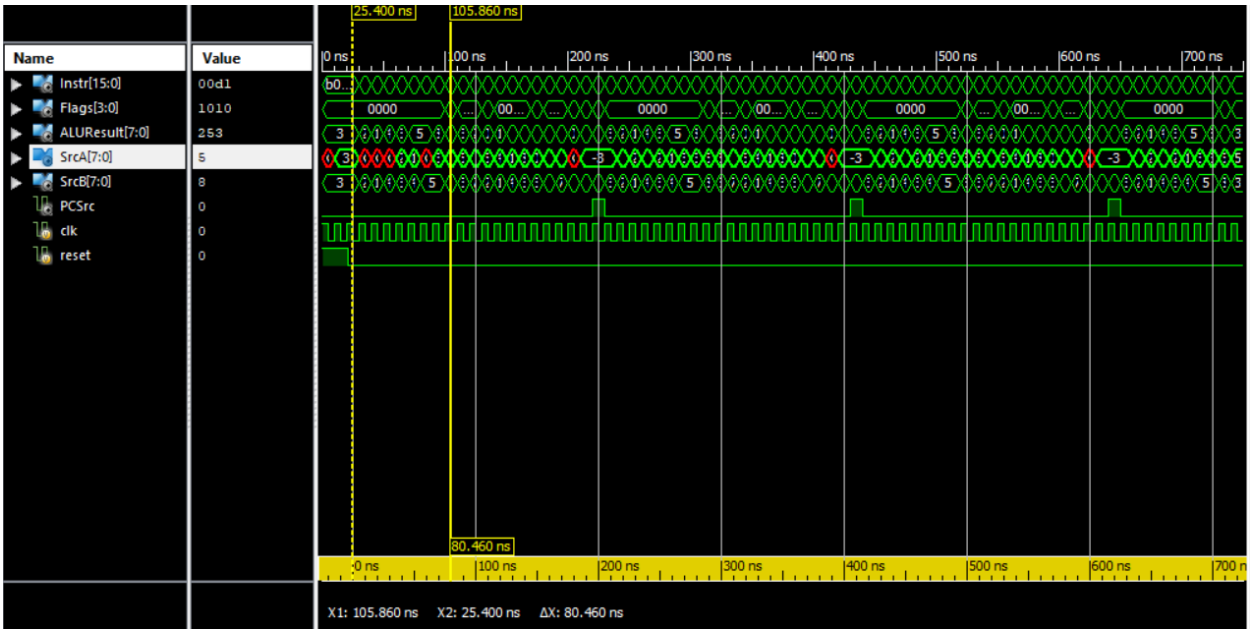
تست

برای تست صحت عملکرد پردازنده ابتدا دستوراتی را آماده میکنیم.

```
LI R0 #3 //R0 = 3
LI R1 #2 //R1 = 2
LI R2 #1 //R2 = 1
LI R3 #4 //R3 = 4
OR R1 R0 //R1 = 3
XOR R2 R3 //R2 = 5
MOV R5 R2 //R5 = 5
ADD R1 R2 //R1 = 8
SUB R2 R1 //R2 = -3
XCHG R5 R0 // R0 = 5 R5 = 3
NOT R2 // R2 = 11111101 R2 = 00000010 = 2
JL to start //not accept
SAR R0 #2 //R0 = 1
SLR R3 #1 //R3 = 2
SAL R3 #4 //R3 = 32
SLL R1 #3 //R1 = 64
DEC R0 #1 // R0 = 0
DEC R0 #1 // R0 = -1
INC R0 #1 // R0 = 0
NOP
CMP R0 R1 //R0-R1 = 0-64=-64
LI R4 #5 // R4 = 5
LI R6 #15 // R6 = 15
ShowR R4
JA to start //not accept
ShowR R6
AND R6 R4 // R6 = 5
ShowRseg R4
ShowRseg R6
CMP R6 R4
JE to JMP0 //accept so R7 doesn't affect
LI R7 #7
JMP to 0
```

تصویر4 – دستورهای تست

دستورهای تست(تصویر(4)) را با استفاده از جدول موجود در دستورکار به هگزادسیمال تبدیل میکنیم و درون فایل memfile.dat قرار میدهیم تا حافظه دستور بتواند دستورات را از آن فایل بخواند.

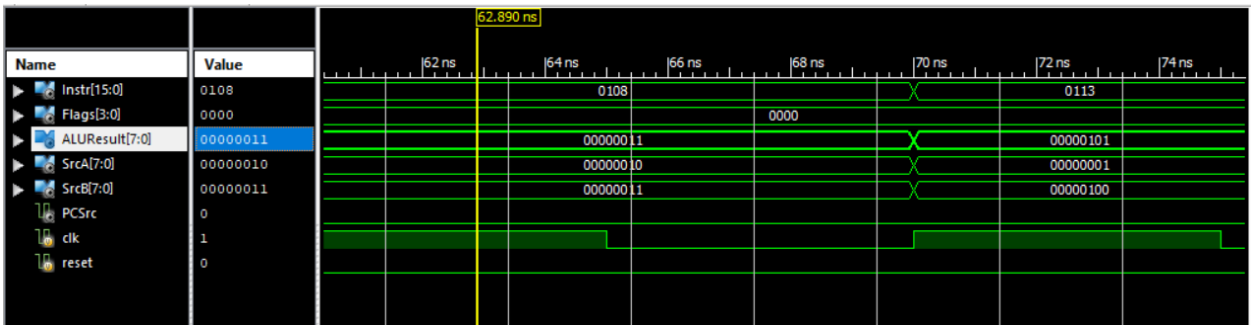


تصویر5- نتیجه شبیه سازی

تصویر(5) نتیجه شبیه سازی پردازنده ساده تک چرخه ای را نشان میدهد که در ادامه به بررسی صحت عملکرد چند دستور میپردازیم.

```
OR R1 R0 //R1 = 3
XOR R2 R3 //R2 = 5
```

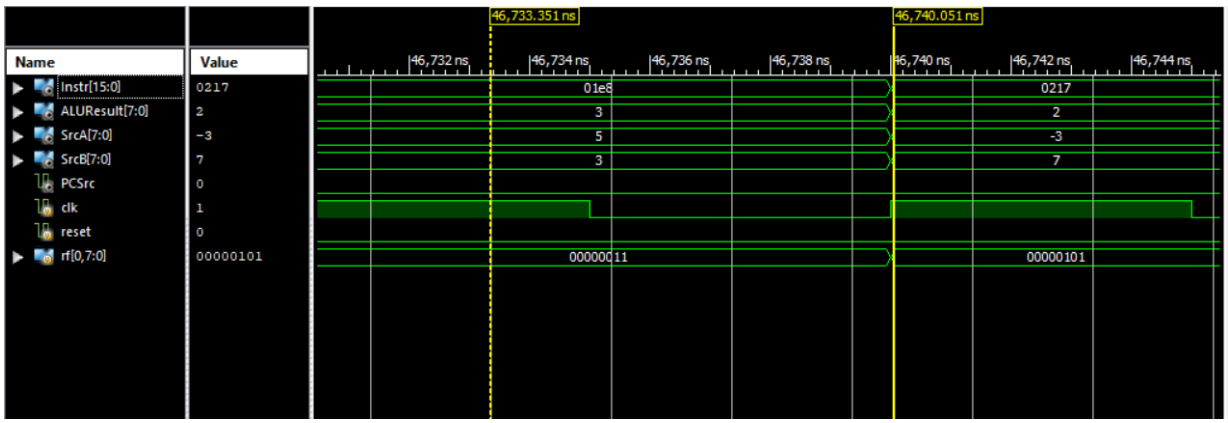
برای تست عملکرد پردازنده ابتدا دو دستور بالا رو مورد بررسی قرار میدهیم.



تصویر6

همانطور که در تصویر(6) مشاهده میکنیم انتظار داریم ALUResult ما در چرخه اول برابر با 3 باشد که همانطور که میبینیم عملیات or بر روی دو علموند انجام شده است و نتیجه 3 در خروجی ALU قرار گرفته است. همچنین در چرخه بعدی نیز بر روی دو عملوند عملیات xor انجام شده است که نشان دهنده این است هر دو بخش کنترل(تعیین نوع عملیات) و بخش مسیر داده(بارگذاری رجیستر ها) به درستی عمل کرده اند.

```
XCHG R5 R0 // R0 = 5 R5 = 3
```

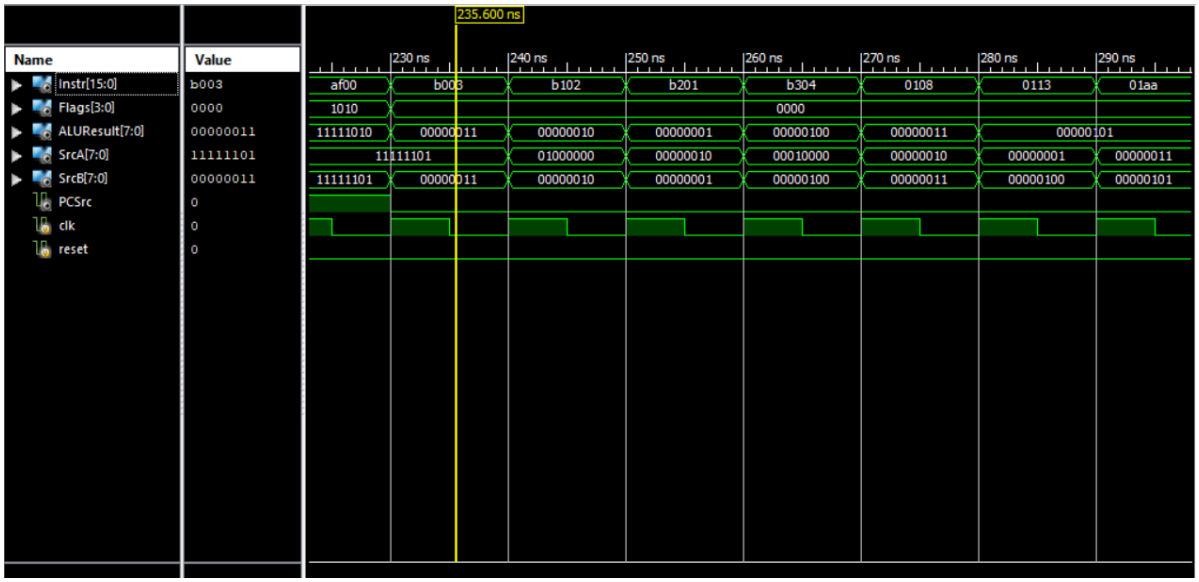


تصویر7

دستور مشخص شده , مقدار دو رجیستر را با هم عوض میکند. همانطور که در تصویر(7) مشخص است, در لبه بالارونده کلاک عملیات نوشتن روی رجیستر انجام میشود و در اتمام چرخه مقدار 5 بر روی رجیستر R0 نوشته میشود. تصویر(7)

```
JMP to 0
```

این دستور, Pc مساوی صفر قرار میدهد به طوری که دستور اول حافظه خوانده میشود و اجرا میشود.

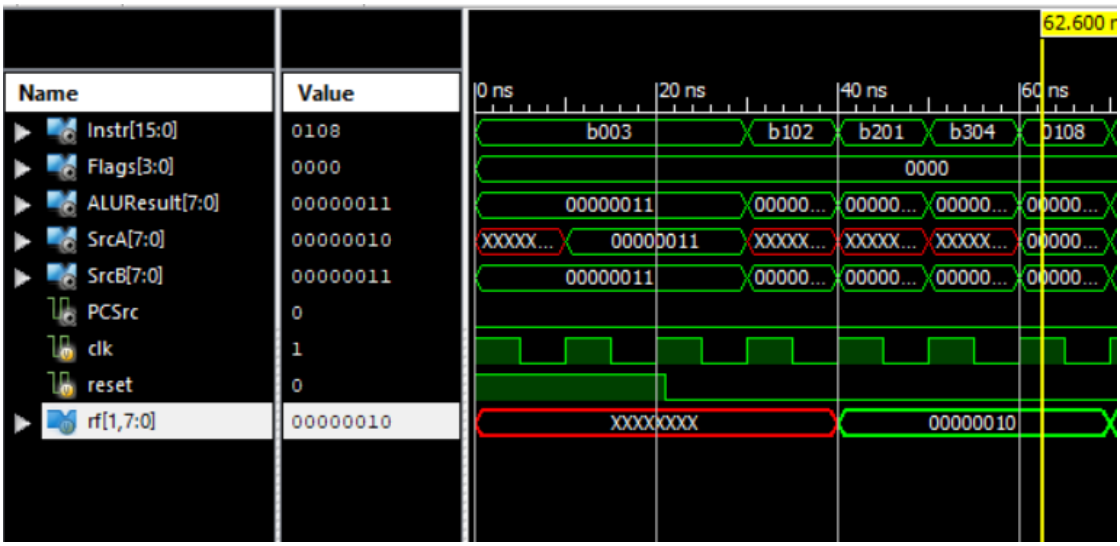


تصویر8

همانطور که در شبیه سازی مشخص است بعد از تمام شدن دستور JMP , دستور اول فایل اجرا میشود. تصویر(8)

```
LI R1 #2 //R1 = 2
```

این دستور مقدار ثابت 2 را بر روی رجیستر R1 کپی میکند.

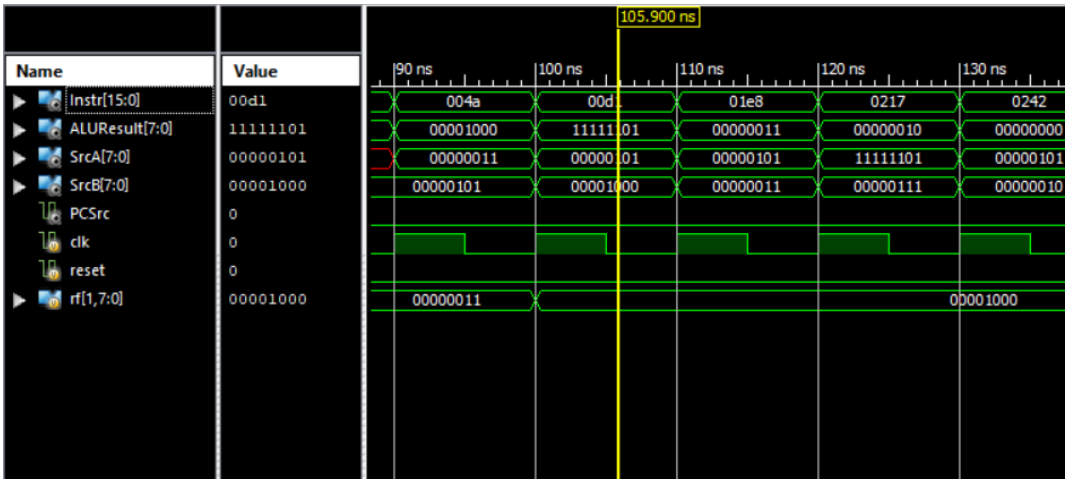


تصویر9

همانطور که مشخص است در لبه بالارونده کلاک مقدار 2 بر روی رجیستر R1 نوشته شده است. تصویر(9)

```
ADD R1 R2 //R1 = 8
```

در این دستور دو رجیستر R1,R2 جمع میشوند و در اخر انتظار داریم مقدار R1 برابر با 8 شود.



تصویر10

در تصویر(10) مشخص است که نتیجه عملیات جمع (ALUResult) در اخر چرخه و همزمان با لبه بالارونده کلاک بر روی رجیستر R1 نوشته میشود.

نتیجه گیری

در این آزمایش, پردازنده ساده تک چرخه ای پیاده سازی شد. عملکرد ماژول های مورد استفاده در این پردازنده, به صورت مختصر شرح داده شد و در ادامه با طراحی تست , به تحلیل صحت عملکرد پردازنده پرداختیم. طبق تست های انجام شده , میتوان صحت عملکرد پردازنده پیاده سازی شده را تایید کرد.