طراحی پردازنده چند سیکل risc-v

درس معماری کامپیوتر - نیمسال دوم ۱۴۰۱

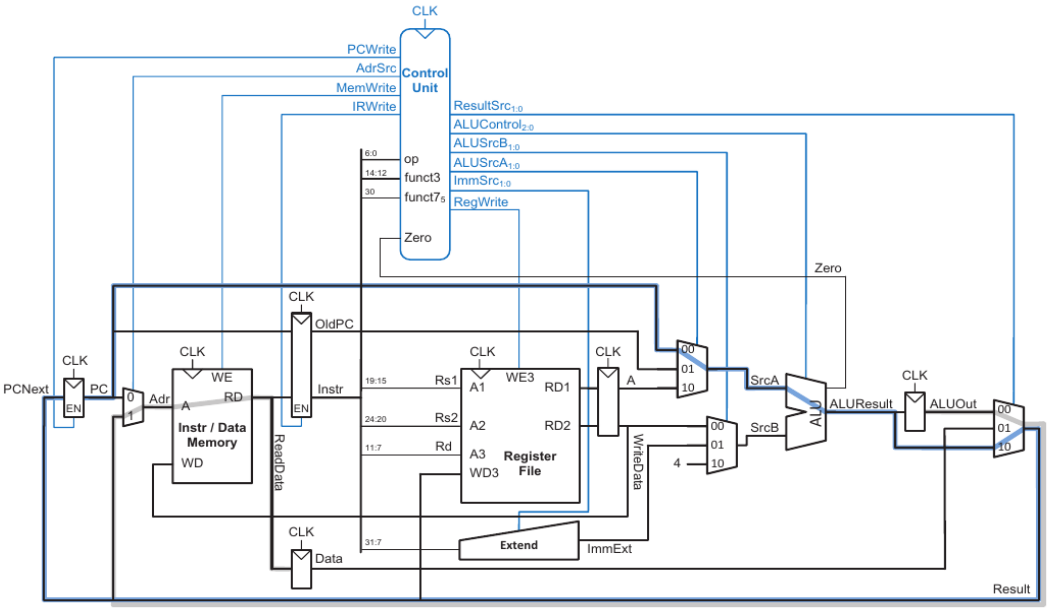
دکتر عطار زاده

امیرحسین منصوری - مائده دهقان

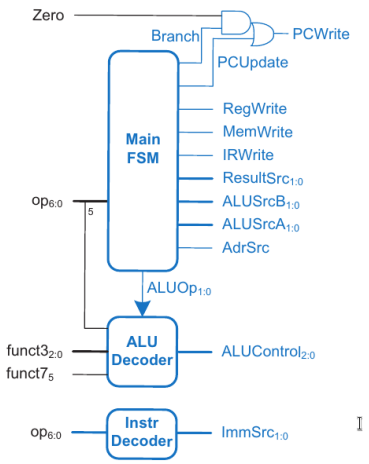
پروژه پایانی

مقدمه

در این پروژه هدف ما طراحی پردازنده چندسیکل معماری RISC-V است.
ساختار پردازنده ما در نهایت به شکل زیر است.



همانطور که در شکل هم میبینیم، پردازنده ما دارای یک واحد کنترل و یک مسیرداده است که به یک واحد حافظه خارجی متصل شده است.
واحد کنترل این پردازنده از سه بخش Main FSM، ALU Decoder و Instr Decoder تشکیل شده است.



این پردازنده هر دستور را در چند سیکل اجرا میکند و داشتن Main FSM به آن کمک میکند که سیگنال های خروجی ترتیبی را در هر مرحله متناسب با مرحله ای که دستور در آن قرار دارد، تولید کند و به مسیر داده بفرستد.

مسیر داده شامل بخش های مختلفی مثل رجیستر فایل و واحد محاسبه است که عملیات خواسته شده را انجام میدهد.

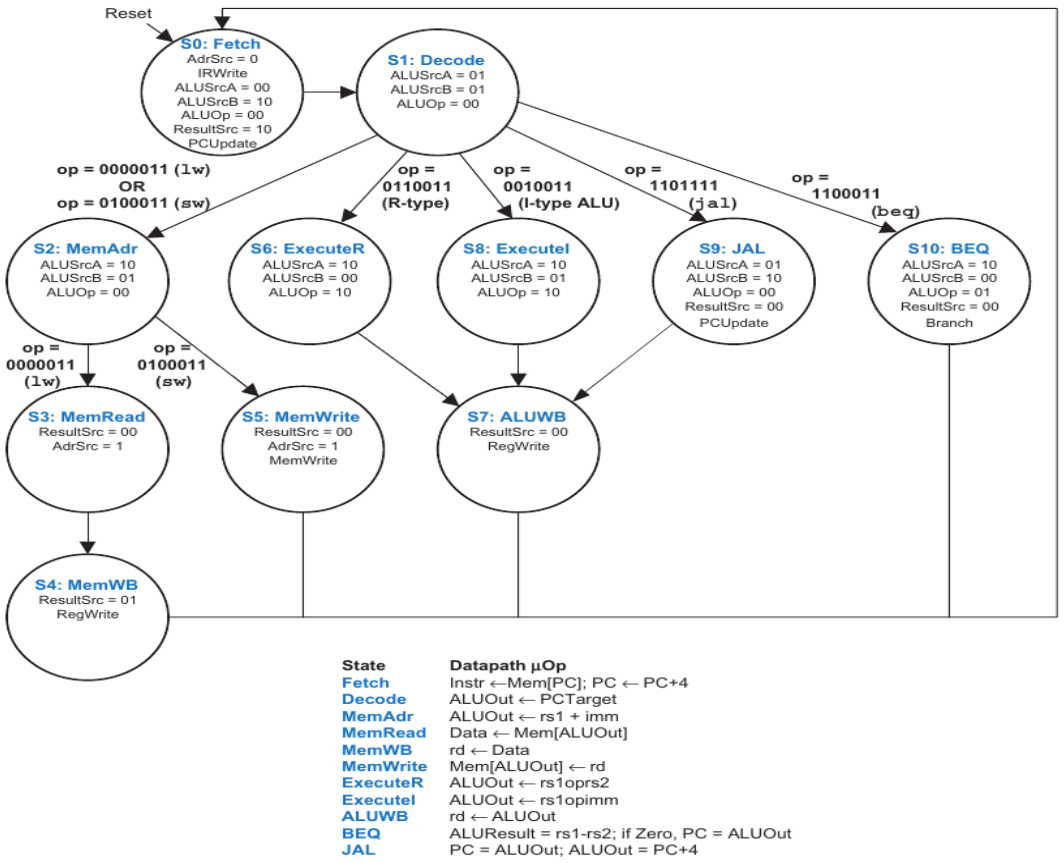
طراحی پردازنده:

طراحی واحد کنترل:

برای طراحی بخش کنترل، ۳ قسمت آن که شامل ماشین حالت اصلی، ALU Decoder، و Instruction Decoder است باید پیاده شود. قسمت ALU Decoder دقیقا مشابه حالت تک سیکل است که در تمرین‌های قبلی پیاده شده است و برای پروژه نیز همان پیاده‌سازی استفاده شده است. قسمت Instruction Decoder نیز یک جدول درستی ساده است و به سادگی پیاده می‌شود.

پیچیده‌ترین قسمت، ماشین حالت اصلی است. برای پیاده سازی این قسمت، از ماشین حالتی که در شکل ۷.۴۵ مرجع هریس آمده استفاده شده است. برای خوانایی بیشتر، حالات به صورت یک enum با اسم هر حالت در نظر گرفته‌شده‌اند و در بقیه کد، به جای کد حالت آن‌ها، اسم آن‌ها آمده است. در نهایت، منطق ترکیبی‌ای برای محاسبه حالت بعدی از روی حالت فعلی و ورودی‌ها، و منطق ترکیبی دیگری برای محاسبه خروجی‌ها از روی حالت فعلی در نظر گرفته شده است. همچنین برای حالت‌های نامعتبر (مثل ورودی‌های نامعتبر و غیرمنتظره)، خروجی به صورت don't care (مقدار x) در نظر گرفته شده تا در صورت وجود اشکال در کد، این موضوع در شبیه‌سازی آشکار شود.

استیت ها و سیگنال های نهایی پردازنده ما به شکل زیر است:



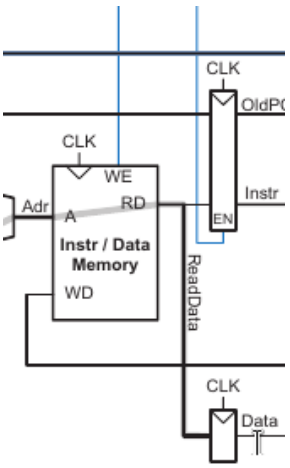
طراحی مسیر داده:

در مسیر داده ما باید سیگنال هایی را از واحد کنترل بگیریم و در مرحله بر اساس سیگنال های داده شده روی داده عملیاتی را انجام دهیم.

مسیر داده ما از واحد های ALU, Register File و تعدادی فلیپ فلاپ و مالتی پلکسر تشکیل شده است. ضمن پیاده سازی هر بخش، برقراری ارتباط مناسب بخش ها با یکدیگر و همچنین ارتباط با واحد کنترل و مموری از چالش های طراحی مسیر داده است.

باید توجه داشت که در هر مرحله پس از انجام عملیات باید خروجی ها را برای مراحل بعد در یک رجیستر مناسب ذخیره کرد.

برای مثال اگر در مرحله خواندن داده به مموری برای ذخیره دستور از فلیپ فلاپ با سیگنال فعال ساز استفاده نکنیم، با هر بار خواندن از حافظه مقدار instruction عوض میشود و دستور ها به طور کامل اجرا نمی شوند و حتی ممکن است داده ای که از مموری خوانده شده است به عنوان یک دستور برای واحد کنترل فرستاده شود!



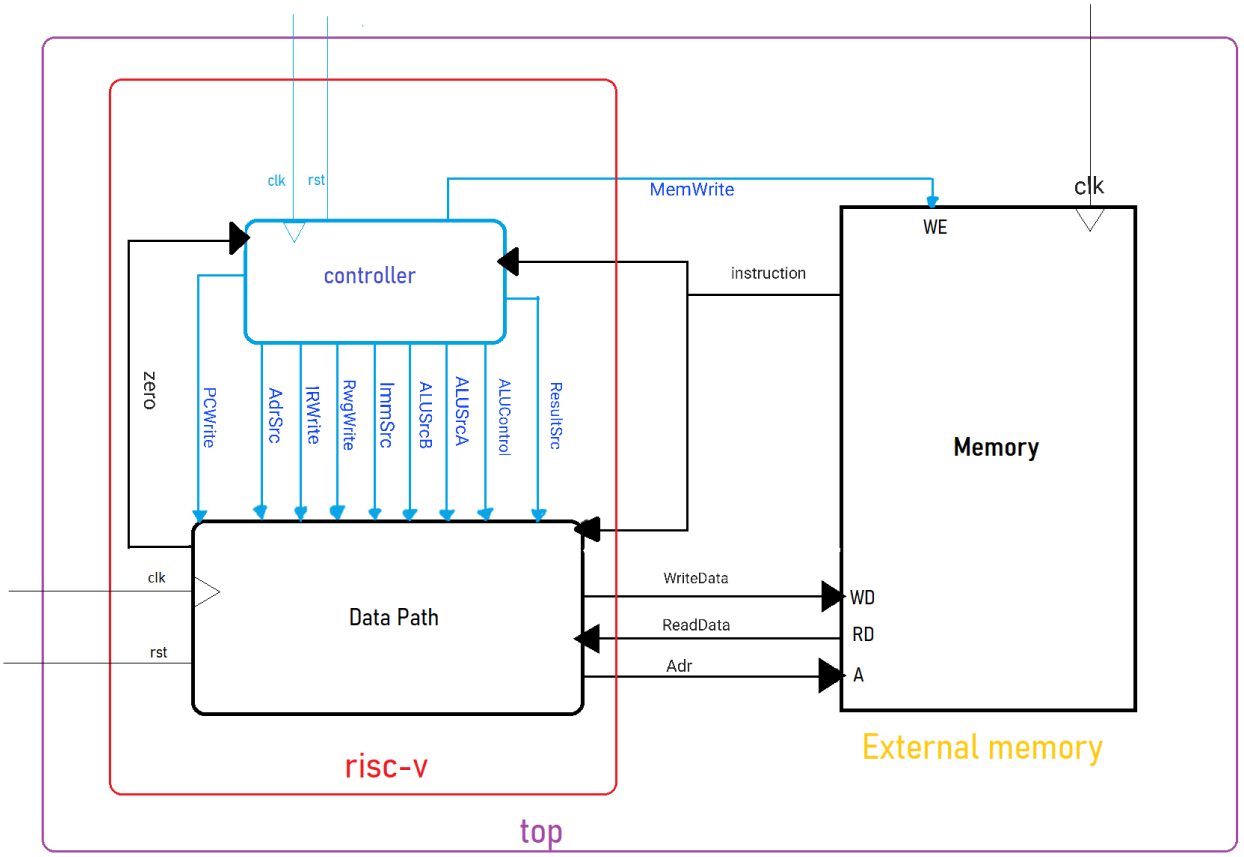
نمودار زیر سلسله مراتب بلوک حافظه، پردازنده riscvmulti ، و کنترلر و مسیر داده و تمام سیگنال های بین آنها را نشان میدهد.

در نمودار زیر ما ارتباط بین سه بخش Data path، controller و memory را مشاهده میکنیم.

مبینیم که controller و Data Path در کنار یکدیگر risc-v و پردازنده ما را تشکیل میدهند. این محدوده با خط قرمز مشخص شده است.

همچنین بخش حافظه را داریم که با سیگنال هایی که در شکل میبینیم به بخش پردازنده متصل شده است. توجه داریم که در پردازنده چند سیکلی، حافظه یکپارچه است.

این دو بخش در کنار یکدیگر top را تشکیل میدهند. در این پروژه هدف غایی ما، پیاده سازی top است به گونه ای که پردازنده در کنار مموری قابلیت انجام یک سری دستور ذخیره شده در حافظه، را داشته باشد.



توصیف سلسله مراتبی کد systemVerilog ی پروژه به شرح زیر آمده است:

همانطور که در قسمت های قبل نیز اشاره شد ما ابتدا بخش top را طراحی میکنیم و سپس به اجزای آن میپردازیم.

ماژول top:

همانطور که گفته شد، این ماژول هدف غایی ما از طراحی است که ارتباط بین پردازنده و موری را نشان میدهد.

این ماژول clkو reset را وروردی میگیرد و سیگنال MemWrite , DataAdr و WriteDat را خروجی میدهد.

همانطور که میبینیم این ماژول دارای نمونه پردازنده (rvmulti) و یک نمونه memory است.

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

  logic [31:0] ReadData;

  // instantiate
  riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                     WriteData, ReadData);
  mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);

endmodule
```

ماژول rvmulti سیگنال MemWrite را خروجی میدهد که وضعیت اجازه نوشتن را مشخص میکند. این سیگنال به عنوان ورودی به موری داده میشود. DataAdr و WriteDAta از پردازنده خارج و به عوان ورودی به موری داده میشوند تا در صورت فعال بودن سیگنال MemWrite، داده در ادرس مذکور در موری نوشته شود.

یک ReadData تعریف کردیم که درواقع مقدار خوانده شده از موری است و و به عنوان ورودی به پردازنده داده میشود.

حال هر کدام از ماژول های پردازنده (rvmulti) و موریmemoryرا بررسی میکنیم.

الف) ماژول mem:

این ماژول که در صورت پروژه نیز به ما داده شده بود، یک مموری را نشان میدهد. یک کلاک و ریست و یک آدرس و یک دیتا به عنوان ورودی میگیرد و یک دیتا به عنوان خروجی پس میدهد.

```
1  module mem(input  logic        clk, we,
2             input  logic [31:0] a, wd,
3             output logic [31:0] rd);
4
5    logic [31:0] RAM[63:0];
6
7    initial
8        $readmemh("../../riscvtest.txt",RAM);
9
10   assign rd = RAM[a[31:2]]; // word aligned
11
12   always_ff @(posedge clk)
13     if (we) RAM[a[31:2]] <= wd;
14 endmodule
```

در خط ۵م میبینیم که مموری ما ساخته شده از ۶۴ داده ۳۲ بیتی است.
در خط ۷م داده ها از فایل riscvtese.txt خوانده میشوند و درون مموری ما قرار میگیرند.
سپس مقدار خروجی rd را از روی آدرس a1 مشخص میکنیم.
رد لبه بالارونده کلاک، اگر سیگنال we فعال بود، مقدار داده ورودی wd را در آدرس ورودی a1 در مموری مینویسم.

ب) ماژول پردازنده (riscvmulti) :

همانطور که در قسمت های پیشین نیز توضیح داده شد، پردازنده ما یک کلاک و یک ریست و یک داده ReadData به عنوان ورودی میگیرد و یک آدرس Adr، یک دیتا برای نوشته شدن در مموری WriteData و یک سیگنال MemWrite به عنوان خروجی میدهد. همانطور که گفته شد در صورت فعال بودن این سیگنال داده در مموری نوشته میشود.

میبینیم که ی نمونه DataPath و یک نمونه controller داریم.

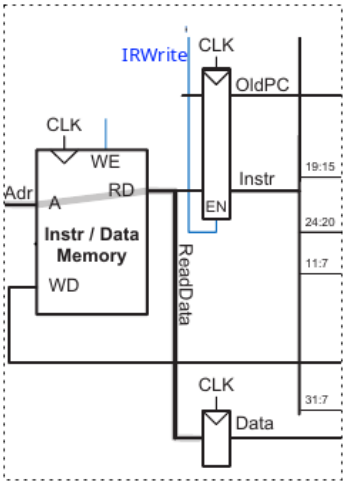
```
module riscvmulti(input  logic        clk, reset,
                  output logic        MemWrite,
                  output logic [31:0] Adr, WriteData,
                  input  logic [31:0] ReadData);

    logic [1:0] ResultSrc;
    logic [2:0] ALUControl;
    logic [1:0] ALUSrcA;
    logic [1:0] ALUSrcB;
    logic [1:0] ImmSrc;
    logic RegWrite, AdrSrc;
    logic IRWrite, PCWrite;
    logic Zero;
    logic [31:0] instruction;
    logic [31:0] Data;


    flopenr#(32) instructionReg(clk,reset,IRWrite,ReadData,instruction);
    flopr#(32) dataSaver(clk, reset, ReadData, Data);

    controller c(clk,reset, instruction[6:0],instruction[14:12],instruction[30],Zero,ImmSrc,ALUSrcA,
                 ALUSrcB,ResultSrc, AdrSrc,ALUControl,IRWrite, PCWrite, RegWrite, MemWrite);

    DataPath dp(clk, reset, ResultSrc, ALUControl,ALUSrcA,ALUSrcB,ImmSrc, RegWrite,
                AdrSrc, IRWrite, PCWrite, instruction, Data, Zero, Adr, WriteData);


endmodule
```

در خطوط ۶م تا ۱۵م داده های مورد نیاز برای ارتباط بین دو نمونه مذکور تعدادی داده تعریف کردیم.

در خط ۱۸م، دیتای ورودی به پردازنده را (ReadData)، در یک رجیستر ذخیره کردیم. نکته مهم این است که عوض شدن مقدار رجیستر instructionReg جدا از rst، به سیگنال فعال کننده IRWrite نیز بستگی دارد به این معنا که تا زمانی که IRWrite از خروجی های واحد کنترل نیز هست، فعال نباشد مقدار این رجیستر ثابت میماند. توجه داریم که این رجیستر دستور را نگه میدارد و این نوع طراحی کمک میکند که بتوانیم از مموری استفاده کنیم بدون آنکه نگران پریدن مقدار پیشین آن(دستوری که در حال اجرا است) باشیم و مقدار دستور جز با اجازه سیگنال فعال کننده تغییری نکند.

د رخط ۱۹م همین دیتا را در رجیستر ذخیره میکنیم که سیگنال فعال کننده ندارد!

به شکل زیر نگاه کنید:



این شکل بخشی از پردازنده چند سیکل risc-v را نشان میدهد.

میبینیم که دیتای خارج شده از مموری روی دو سیم قرار میگیرد که یکی instruction تعبیر میشود و تا پایان اجرای کامل آن و شروع دستور بعدی، این مقدار باید ثابت بماند ولی برای سیم دیگر اینطور نیست و در طول اجرای دستور نیز ممکن است ما از مموری چیزی بخوانیم که این داده در رجیستر instructionReg نوشته نمیشود ولی در saveData نوشته شده و مقدار جدید به مسیر داده فرستاده میشود.

میبینیم که ی نمونه DataPath و یک نمونه controller داریم.

هرکدام از این ماژول ها به طور کامل در بخش بعدی شرح داده میشوند:

الف) واحد کنترل (controller):

این واحد از ۳ قسمت ماشین حالت اصلی، ALU Decoder، و Instruction Decoder تشکیل شده است.

```
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;

  assign PCWrite = (Zero & Branch) | PCUpdate;

  mainfsm fsm (clk, reset, op, Branch, PCUpdate, RegWrite, MemWrite, IRWrite, ResultSrc, ALUSrcA, ALUSrcB, AdrSrc, ALUOp);

  aludec dec (op[5], funct3, funct7b5, ALUOp, ALUControl);

  instrdec idec (op, ImmSrc);

  endmodule
```

واحدهای ALU Decoder و Instruction Decoder نیز به صورت زیر هستند (قسمت ماشین حالت اصلی، به دلیل طولانی بودن کد در گزارش آورده نشده‌اند).

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

always_comb begin
    case (op)
        'b0000011: ImmSrc = 2'b00;
        'b0100011: ImmSrc = 2'b01;
        'b0110011: ImmSrc = 2'bxx;
        'b1100011: ImmSrc = 2'b10;
        'b0010011: ImmSrc = 2'b00;
        'b1101111: ImmSrc = 2'b11;
        default: ImmSrc = 2'bxx;
    endcase
end

endmodule
```

```
module aludec(input logic opb5,
    input logic [2:0] funct3,
    input logic funct7b5,
    input logic [1:0] ALUOp,
    output logic [2:0] ALUControl);

logic RtypeSub;
assign RtypeSub = funct7b5 & opb5; // TRUE for Rtype subtract

always_comb
    case(ALUOp)
        2'b00: ALUControl = 3'b000; // addition
        2'b01: ALUControl = 3'b001; // subtraction
        default: case(funct3) // Rtype or Itype ALU
                3'b000: if (RtypeSub)
                            ALUControl = 3'b001; // sub
                        else
                            ALUControl = 3'b000; // add, addi
                3'b010: ALUControl = 3'b101; // slt, slti
                3'b110: ALUControl = 3'b011; // or, ori
                3'b111: ALUControl = 3'b010; // and, andi
                default: ALUControl = 3'bxxx; // ???
            endcase
    endcase
endmodule
```

همانطور که پیش‌تر اشاره شد، ماژول aludec همان ماژولی است که برای پردازنده تک سیکل در مرجع هریس استفاده شده است.

ب) طراحی مسیر داده(*Data Path*):

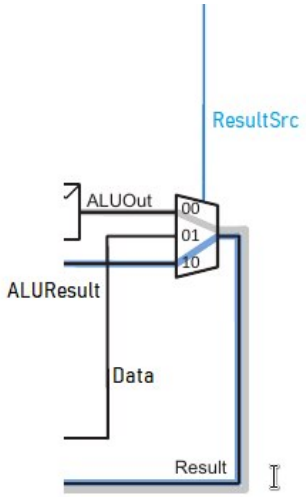
۱-ماژول مسیرداده

ماژول مسیرداده سیگنال های کنترلی خود را از کنترلر میگیرد. (ورودی های خط ۲ تا ۸)؛ همچنین داده و instruction را از مموری میگیرد و سیگنال zero و آدرس و داده ای که باید در حافظه نوشته شود را برمیگرداند.

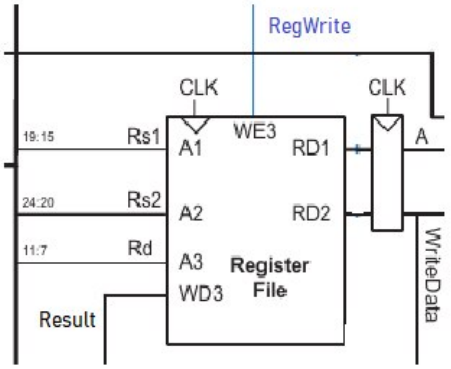
```
1  module DataPath(input  logic clk, rst,
2                  input  logic [1:0] ResultSrc,
3                  input  logic [2:0] ALUControl,
4                  input  logic [1:0] ALUSrcA,
5                  input  logic [1:0] ALUSrcB,
6                  input  logic [1:0] ImmSrc,
7                  input  logic RegWrite,
8                  input  logic AdrSrc,
9                  input  logic IRWrite, PCWrite,
10                 input  logic [31:0] Instr,
11                 input  logic [31:0] Data,
12                 output logic zero,
13                 output logic [31:0] Adr, WriteData
14                 );
15
16    logic [31:0] Result;
17    logic [31:0] rd1,rd2;
18    logic [31:0] A_a; //rd1
19    logic [31:0] immExt;
20    logic [31:0] SrcB;
21    logic [31:0] SrcA;
22    logic [31:0] ALUOut;
23    logic [31:0] PC,oldPc;
24    logic [31:0] ALUResult;
25
26    flopenr#(32) pcSaver(clk,rst,IRWrite, PC,oldPc);
27    flopenr#(32) nextPcReg(clk,rst,PCWrite,Result, PC);
28
29    mux3#(32) resultChoser(ALUOut, Data ,ALUResult, ResultSrc ,Result);
30    regFile rf(clk, RegWrite, Instr[19:15], Instr[24:20], Instr[11:7], Result, rd1,rd2); //instance of register file
31    flopr#(32)  rd1Saver(clk,rst, rd1, A_a);
32    flopr#(32)  rd2Saver(clk,rst, rd2, WriteData);
33
34    extend extendUnit(Instr[31:7], ImmSrc, immExt);
35
36    mux3#(32) muxSrcB(WriteData, immExt, 'd4, ALUSrcB,SrcB ); //srcb mux
37    mux3#(32) muxsrcA(PC,oldPc,A_a ,ALUSrcA,SrcA); //srcA mux
38    alu mainAlu(SrcA,SrcB, ALUControl, ALUResult, zero); //instance of alu
39    flopr#(32)  aluSaver(clk,rst, ALUResult, ALUOut);
40
41    mux2#(32) address(PC,Result, AdrSrc, Adr);
42 endmodule
```

در خط ۲۶ و ۲۷ دو رجیستر تعریف کردیم. رجیستر PcSaver مقدار اولیه PC را ذخیر میکند. سیگنال enable این رجیستر IRWrite است که از سمت کنترلر و فقط در مرحله Fetch فعال میشود. رجیستر nextPcReg نتیجه انتخاب شده توسط مالتی پلکسر resultChooser را ذخیره می کند. سیگنال enable این رجیستر PCWrite است.

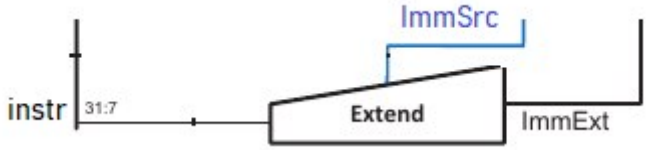
در خط ۲۹ یک مالتی پلکسر قرار دادیم که با استفاده از سیگنال ResultSrc بین داده ای که از مموری آمده(Data) و نتیجه در لحظه واحد محاسبه(ALUResult) و نتیجه واحد محاسبه در کلاک قبلی (ALUOut), داده Result را انتخاب میکند.



در خط ۳۰ از رجیسترفایل یک موجودیت تشکیل دادیم و در ۲ خط بعد نتایج بیرون آمده از آن را در دو رجیستر ذخیره کردیم.

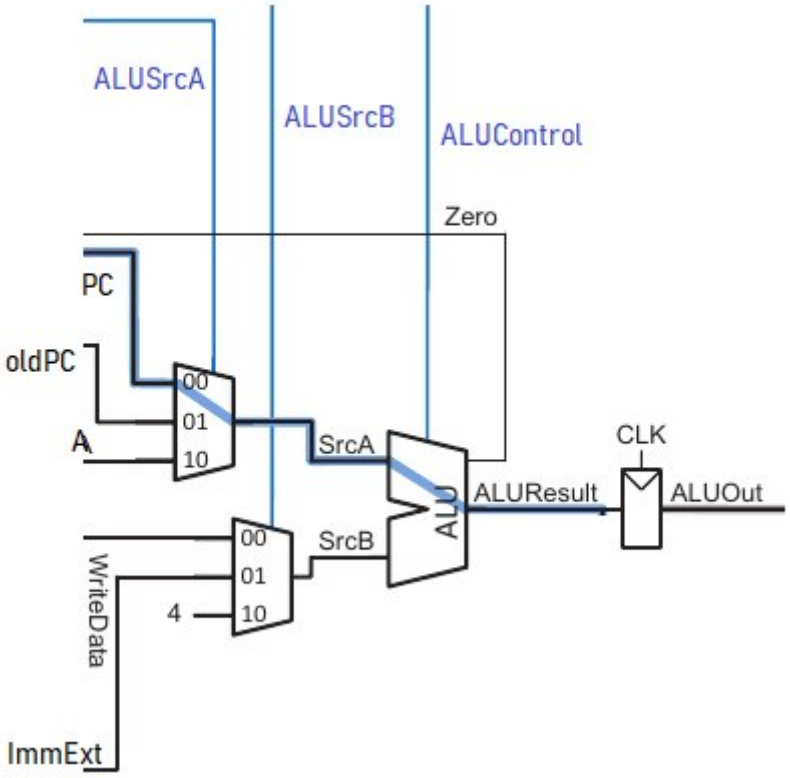


سپس extendUnit که نمونه ای از extend است را ساخته ایم. این واحد بخشی از دستور را گرفته و با استفاده از ImmSrc خروجی ImmExt متناسب با نوع دستور را تولید میکند.

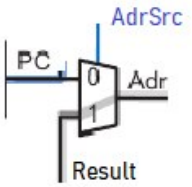


سپس دو مالتی پلکسر داریم که با استفاده از سیگنالی که از سمت کنترلر می اید نتیجه درست برای ورودی ALU را انتخاب میکنند.

یک نمونه از ALU میسازیم و خروجی دو مالتی پلکسر مذکور را به آن میدهیم. سیگنال ALUControl نیز از سمت کنترلر وارد این بخش می شود و نتیجه مورد نظر ALUResult بر این اساس تولید میشود و در رجیستر alusaver ذخیره میشود. اگر ALUResult مساوی صفر باشد سیگنال zero که از مسیرداده به واحد کنترل میرود فعال میشود.



در آخر با گذاشتن مالتی پلکسری آدرسی که باید از مموری خوانده شود را انتخاب میکنیم. این انتخاب با استفاده از سیگنال AdrSrc از بین نتیجه مالتی پلکسر resultChoser و PCانتخاب میشود.



حال نوبت به بررسی اجزای مسیرداده میرسد.

۲- ماژول بانک رجیستر(Register File):

این ماژول سه ادرس (a0,a1,a2)و یک دیتا (result)ورودی میگیرد. همچنین سیگنال RegWrite را به عنوان ورودی به این ماژول میدهیم که مانند enable برای پورت نوشتن عمل میکند. خروجی این ماژول دو دیتا است که در ادرس های a1 و a2 وجود داشته اند.

```
1  module regFile( input logic clk,
2                  input logic RegWrite,
3                  input logic [4:0] a1,
4                  input logic [4:0] a2,
5                  input logic [4:0] a3,
6                  input logic [31:0] result,
7                  output logic [31:0] Rd1,
8                  output logic [31:0] Rd2);
9   //creating array of registers
10     logic [31:0] rf[31:0];
11     always @(posedge clk)
12         if(RegWrite) rf[a3] <= result;
13
14     assign Rd1 =  a1 == 0 ? 0 : rf[a1];
15     assign Rd2 =  a2 == 0 ? 0 : rf[a2];
16
17 endmodule
```

در خط ۱۰ میبنیم که رجیستر فایل یک آرایه از رجیستر ها است. در ایجا ما ۳۲ رجیستر ۳۲ بیتی را در نظر گرفتیم. در بلوک always میبینیم که در لبه بالارونده کلاک اگر RegWrite فعال بود مقدارresult در ادرس a3 نوشته میشود.

همچنین خروجی های RD1 و RD2 برابر با مقدار ذخیره شده در آدرس های a1 و a2 قرار می گیرند.

۳- ماژول گسترش دهنده (extend):

این ماژول بخشی از دستور و همچنین سیگنال ImmSrc را از واحد کنترل دریافت میکند و با استفاده از منطقی ترکیبی خروجی immExt را تولید میکند. این منطق با استفاده از *جدول B.1* کتاب *(Digital Design and Computer Architecture-riscvEdition (Sarah L. Harris ,David Harris* پیاده سازی شده است.

```
module extend(input logic [31:7] instr, input logic [1:0] immsrc, output logic [31:0] immext);
    always_comb
    case(immsrc)
        2'b00 : immext = {{20{instr[31]}}, instr[31:20]};
        2'b01: immext  = {{20{instr[31]}}, instr[31:25], instr[11:7]};
        2'b10: immext  = {{20{instr[31]}} , instr[7], instr[30:25], instr[11:8], 1'b0};
        2'b11: immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
        default immext = 2'b00;
    endcase

endmodule
```

۴-ماژول واحد محاسبه(alu):

این ماژول دو ورودی و یک سیگنال کنترل از سمت واحد کنترل میگیرد و خروجی z را تولید و سیگنال zero را به واحد کنترل ارسال میکند.

```
module alu(input logic [31:0] a, b,
                input logic [2:0] control,
                output logic [31:0] z,
                output logic zero);

    always_comb
        case (control)
            3'b000: z = a + b;
            3'b001: z = a - b;
            3'b010: z = a & b;
            3'b011: z = a | b;
            3'b101: z = a < b ? 1 : 0;
            default: z = 32'bx;
        endcase

    assign zero = z == 0;

endmodule
```

در واقع سیگنال control مشخص میکند که ALU باید چه عملیاتی را انجام دهد. منطق ترکیبی پیاده سازی شده برای مشخص کردن مقدار z، با استفاده از جدول داده شده در صورت پروژه، نوشته شده است.
سیگنال zero زمانی مقدار ۱ میگیرد که خروجی محاسباتی ALU یا همان z، مساوی صفر باشد و در غیر اینصورت، مقدار آن صفر است.

۵- فلیپ فلاپ (flopr):

این ماژول کلاک و ریست و یک دیتا به عنوان ورودی میگیرد و یک دیتا را به عنوان خروجی برمیگرداند.

```
1  module flopr# (parameter WIDTH = 8 )
2                    (
3                    input logic clk, rst,
4                    input logic [WIDTH-1:0] d,
5                    output logic [WIDTH-1:0] q);
6
7      always_ff@(posedge clk, posedge rst)
8          if(rst) q<=0;
9          else    q<=d;
10
11
12 endmodule
```

در لبه بالارونده کلاک و ریست، اگر ریست فعال بود، مقدار خروجی برابر صفر قرار داده میشود و در غیر این صورت، مقدار ورودی در خروجی ریخته میشود.
در پیاده سازی این ماژول از قابلیت Generic در زبان SystemVerilog استفاده کردیم. به این معنا که منطق ماژول برای مقدار های مختلف متغیر WIDTH یکسان است و برای مثال برای پیاده سازی یک فلیپ فلاپ با ظرفیت ۳۲ بیت نیاز به تعریف ماژول جدید نداریم بلکه میتوانیم مقدار WIDTH را مساوی ۳۲ قرار دهیمو د راینجا مقدار پیش فرض WIDTH، هشت میباشد.

۶-فلیپ فلاپ دارای قابلیت فعال شدن(flopenr):

ویژگی های این فلیپ فلاپ و نحوه کار آن شبیه نوع قبلی است ولی یک تفوات وجود دارد.
این ماژول سیگنال enableی نیز به عنوان ورودی دریافت میکند و در صورتی مقدار خروجی آن، برابر با مقدار ورودی میشود که سیگنال enb فعال باشد.

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

    always_ff@(posedge clk, posedge rst)
        if(rst) q<=0;
        else if(enb)    q<=d;


endmodule
```

از این نوع فلیپ فلاپ در ذخیره کردن مقدار PC و Instruction استفاده کردیم.

۷-مالتی پلکسر دو به یک(mux2):

این ماژول دو ورودی و یک سیگنال تعیین کننده میگیرد و یکی از آن دو ورودی را بر اساس سیگنال تعیین کننده به عنوان مقدار خروجی در نظر میگیرد.

```
module mux2 #(parameter WIDTH = 8)(
                  input logic [WIDTH-1:0] a0,a1,
                  input logic S,
                  output [WIDTH-1:0] result);

   assign result  = S? a1 : a0;
endmodule
```

همانطور که در خط ششم مشخص است با استفاده از عملگر سه عملوندی، گفتیم اگر S مساوی ۱ بود، result را برابر a1 و در غیر اینصورت آن را مساوی a0 قرار بده.

جدول درستی آن به شکل زیر است:

Result	S
a1	1
a0	0

در پیاده سازی این ماژول از قابلیت Generic که در قسمت های قبل توضیح داده شد، استفاده شده است.

۸-مالتی پلکسر سه به یک (mux3):

عملکرد این ماژول، شبیه مورد قبل است. البته ۳ ورودی میگیرد و سیگنال تعیین کننده آن ۲ بیتی است.

```
1 module mux3 #(parameter WIDTH = 8)(
2                 input logic [WIDTH-1:0] a0,a1,a2,
3                 input logic [1:0] s,
4                 output logic [WIDTH-1:0] result);
5
6     assign result = s[1] ? a2 : (s[0] ? a1 : a0);
7 endmodule
```

در خط ۶، میبینیم که خروجی با استفاده از ترکیب دو عملگر شرطی سه عملوندی، مشخص میشود.

پیش‌بینی سیگنال‌ها (جدول ۱)

جدول ۱ برای سه دستور اول به صورت زیر است.

Step	PC	Instr	State	Result	Result Notes
1	00	N/A	Fetch (S0)	4	PC += 4
2	04	0x00500113	Decode (S1)	X	OldPC + Immediate
3	04	0x00500113	Executel (S8)	X	ALUResult = x0

					+ 5 = 5
4	04	0x00500113	ALUWB (S7)	5	Result = ALUOut
5	04	0x00500113	Fetch (S0)	8	PC += 4
6	08	0x00C00193	Decode (S1)	X	OldPC + Immediate
7	08	0x00C00193	ExecuteI (S8)	X	ALUResult = x0 + 12 = 12
8	08	0x00C00193	ALUWB (S7)	12	Result = ALUOut
9	08	0x00C00193	Fetch (S0)	4	PC += 4
10	12	0xFF718393	Decode (S1)	X	OldPC + Immediate
11	12	0xFF718393	ExecuteI (S8)	X	ALUResult = x3 + (-9) = 12 - 9 = 3
12	12	0xFF718393	ALUWB (S7)	3	Result = ALUOut

شبیه‌سازی و تست

تست کنترلر

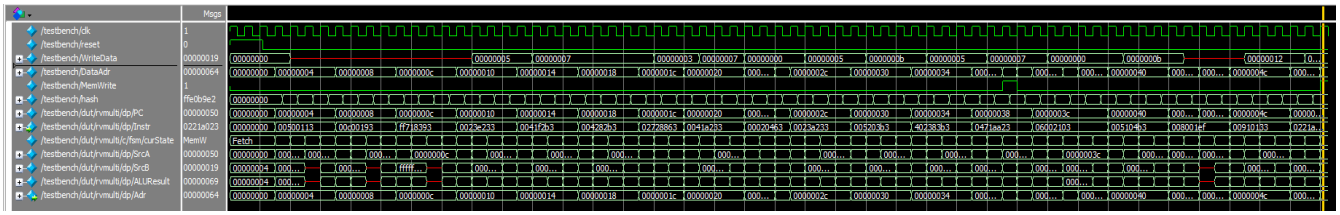
تست کنترلر به این شکل کار می‌کند که مجموعه‌ای از ورودی‌ها و خروجی مورد انتظار آن‌ها را که در فایل controller.tv قرار دارد را به شکل یک بردار تست خوانده، ورودی‌ها را در لبه بالارونده کلاک به کنترلر می‌دهد و در لبه پایین رونده، قسمت‌های مختلف آن را جداگانه مقایسه می‌کند و در صورت وجود تفاوت، آن را گزارش می‌دهد. همچنین هر جفت ورودی و خروجی، برای یک کلاک داده شده و درواقع خروجی‌ها را در هر حالتِ ماشین حالت جداگانه بررسی می‌کند؛ برای همین برای هر دستور، به تعداد CPI آن دستور تست داریم.
نتیجه اجرای تست کنترلر به صورت زیر است.

```
VSIM 20> run -all
#         47 tests completed with          0 errors
# hash = 39
# ** Note: $stop    : C:/Users/Mans/Documents/CA HW/riscv-multicycle/controller_tb.sv(113)
#    Time: 485 ps  Iteration: 1  Instance: /controller_tb
# Break in Module controller_tb at C:/Users/Mans/Documents/CA HW/riscv-multicycle/controller_tb.sv line 113
```

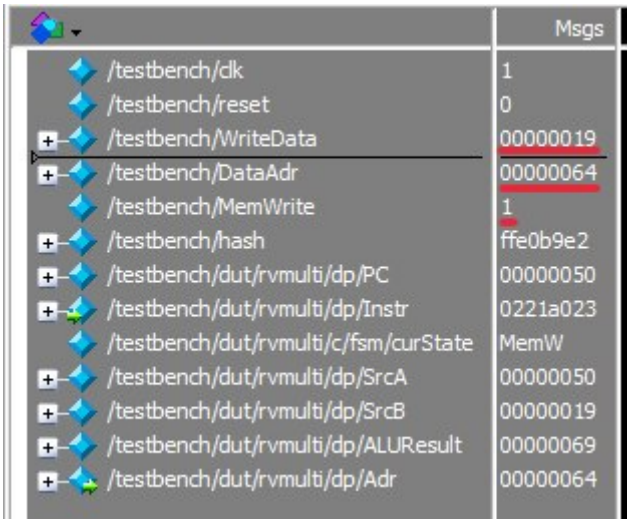
پس از اجرای تست، انجام ۴۷ تست موجود در بردار تست بدون خطا گزارش می‌شود؛ در نتیجه کنترلر از تست عبور می‌کند.

تست کلی پردازنده

با اجرای فایل starter.sv، و قرار دادن فایل riscvtest.txt که شامل کد ماشین برنامه تست است، پردازنده را تست می‌کنیم (لازم به ذکر است که ماژول‌هایی که در فایل به صورت پیش‌فرض تعریف شده بود، به فایل‌های جداگانه‌ای منتقل شده‌اند و در نتیجه فایل starter.sv تنها شامل ماژول تست‌بنچ است).
شیوه کارکرد تست به این صورت است که کد ماشین را در ممووری (ماژول mem) بارگذاری می‌کند و سپس با دادن سیگنال clock، شروع به اجرای برنامه می‌کند. درصورت کارکرد درست پردازنده، تنها یک بار باید مقدار ۲۵ در آدرس ۱۰۰ نوشته شود. این مورد در تست‌بنچ بررسی شده و در صورت وقوع چنین نوشتنی، پیامی مبنی بر موفقیت تست چاپ می‌شود و تست متوقف می‌شود.
شکل موج شبیه‌سازی به صورت زیر است.



مقادیر سیگنال‌ها در آخرین سیکل اجرای برنامه به صورت زیر است.



با توجه به مقادیر مشخص شده، مقدار 0x19=25 به درستی در آدرس 0x64=100 نوشته می‌شود؛ در نتیجه کد پردازنده با موفقیت تست را پاس می‌کند.

مشخصات سیستم سنتز شده

این پردازنده، برای دستگاه Cyclone IV مدل EP4CE115F29C7 سنتز شده است.

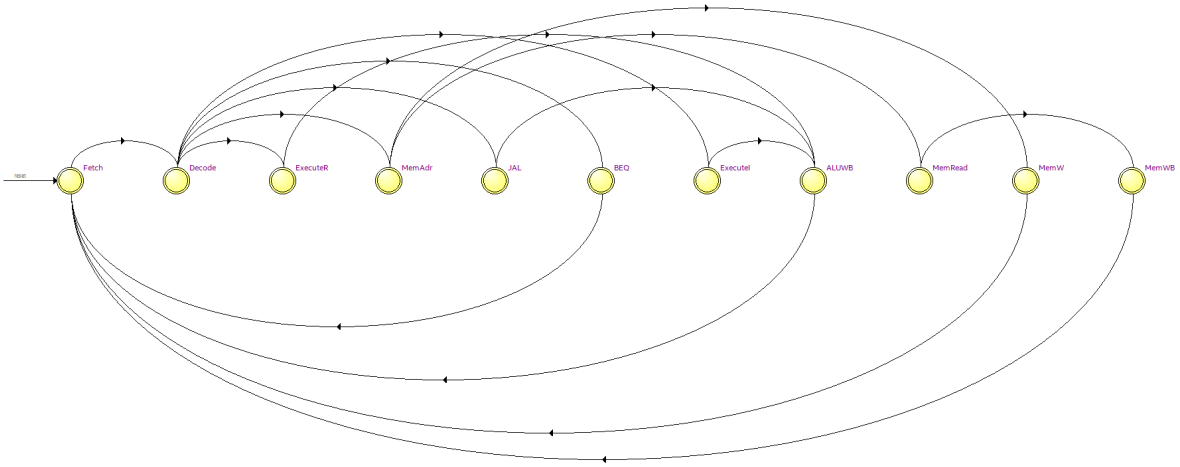
مشخصات منابع مصرفی پس از سنتز، به صورت زیر است.

Flow Status	Successful - Sun Jul 03 18:41:54 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	riscv
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,143 / 114,480 (3 %)
Total registers	2435
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 %)

حداکثر فرکانس پردازنده روی این FPGA نیز با استفاده از گزارش Fmax، برابر ۶۶٫۷۴MHz به دست می‌آید.

	Fmax	Restricted Fmax	Clock Name	Note
1	66.74 MHz	66.74 MHz	clk	

خروجی State Machine Viewer نیز به صورت زیر است.



شکل بالا، مطابق ماشین حالت موجود در کتاب هریس است (هرچند که شکلی کمی متفاوت دارد).

خروجی RTL Viewer نیز (به دلیل حجم زیاد خروجی) ضمیمه گزارش شده است.

بخش امتیازی:

در این بخش برای پوشش دستورات بیشتر توسط پردازنده، نیاز است که تغییراتی را در کدمان اعمال کنیم.

در این بخش دستورات زیر اضافه شده اند:

xor-xori-sll-slli-srl-srli-sra-sari-mul-mulh-mlusu-mulhu-div-rem

برای انجام این دستورات نیاز داریم که فقط جدول درستی ALUDecoder و عملیات مرتبط با هر دستور را در واحد محاسبه زیاد کنیم.

با اضافه کردن این دستودارات دیگر داشتن ۳ بیت برای سیگنال ALUControl که فقط ۸ حالت را پوشش میدهد، کافی نیست. پس سیگنال ALUControl را ۴ بیتی میکنیم.

همچنین aludecoder برای پوشش دستورات بیشتری که بیت پنجم آنها وبیت پنجم funct7 و funct3 آنها یکسان است، به بیت اول funct7 که همان instruction[25]s است نیز نیاز دارد. پس ان را به ورودی های ماژول controller و aludecoder اضافه میکنیم.

همچنین همانطور که گفته شد تعداد بیت های سیگنال ALUControl را نیز ۴ بیت ثرار میدهیم.

```
module controller(input  logic       clk,
                  input  logic       reset,
                  input  logic [6:0] op,
                  input  logic [2:0] funct3,
                  input  logic       funct7b5,
                  input  logic       funct7b0,
                  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;

assign PCWrite = (Zero & Branch) | PCUpdate;

mainfsm fsm (clk, reset, op, Branch, PCUpdate, RegWrite, MemWrite, IRWrite, ResultSrc, ALUSrcA, ALUSrcB, AdrSrc, ALUOp);

aludec dec (op[5], funct3, funct7b5, funct7b0,ALUOp, ALUControl);

instrdec idec (op, ImmSrc);

endmodule
```

جدول منطقی ALUDecoder برای باقی دستورات به شکل زیر خواهد بود:

ALUop	funct3	op[5]funct7[5]	funct7[0]	instruction	ALUcontrol
10	100	x	0	xor,xori	0100
10	001	x	0	sll,slli	0110
10	101	00,01	0	srl,srli	0111
10	101	01,11	0	sra,sari	1000
10	000	x	1	mul	1001
10	001	x	1	mulh	1010
10	010	x	1	mulhsu	1011
10	011	x	1	mulhu	1100
10	100	x	1	div	1101
10	101	x	1	rem	1110

همانطور که گفتیم با اضافه کردن این دستورات بیت صفرم funct7 نقش تعیین کننده ای به خود میگیرد. زیرا آنهایی که funct3 یکسانی دارند را جدا میکند. در نتیجه کد ما به شکل زیر خواهد بود:

ماژول aludecoder:

```
module aludec(input logic opb5,
    input logic [2:0] funct3,
    input logic funct7b5,
    input logic funct7b0,
    input logic [1:0] ALUOp,
    output logic [3:0] ALUControl);

    logic RtypeSub;
    assign RtypeSub = funct7b5 & opb5; // TRUE for Rtype subtrac

always_comb begin
      case(ALUOp)
         2'b00: ALUControl = 4'b000; // addition
         2'b01: ALUControl = 4'b001; // subtraction
         default: if (funct7b0) begin
            case(funct3)
               3'b000: ALUControl = 4'b1001; //mul ==>extra
               3'b001: ALUControl = 4'b1010; //mulh ==>extra
               3'b010: ALUControl = 4'b1011; //mulhsu ==>extra
               3'b011: ALUControl = 4'b1100; //muluu ==>extra
               3'b100: ALUControl = 4'b1101; //div ==>extra
               3'b110: ALUControl = 4'b1110; //rem ==>extra
               default: ALUControl = 4'bxxxx;
            endcase
         end
         else case(funct3) // Rtype or Itype ALU
                  3'b000: if (RtypeSub)
                     ALUControl = 4'b0001; // sub
                     else
                     ALUControl = 4'b0000; // add, addi
                  3'b010: ALUControl = 4'b0101; // slt, slti
                  3'b110: ALUControl = 4'b0011; // or, ori
                  3'b111: ALUControl = 4'b0010; // and, andi
                  3'b100: ALUControl = 4'b0100; //xor - xori ====> extra
                  3'b001: ALUControl = 4'b0110; //shift left left logical / immediate ===> extra
                  3'b101: if(funct7b5)
                              ALUControl = 4'b1000; //shift right arithmetic==>extra
                           else
                              ALUControl = 4'b0111; //shift right logical ==> extra
                   default: ALUControl = 4'bxxxx; // ???
               endcase
      endcase
   end
endmodule
```

ماژول alu:

```
module alu(input logic [31:0] a, b,
                input logic [3:0] control,
                output logic [31:0] z,
                output logic zero);

    logic [63:0] s_mul;
    logic [63:0] u_mul;
    logic [63:0] s_u_mul;
    always_comb begin
        s_mul = a * b;
        u_mul = $unsigned(a) * $unsigned(b);
        s_u_mul = $signed(a) * $unsigned(b);
    end
    always_comb
        case (control)
            4'b0000: z = a + b;
            4'b0001: z = a - b;
            4'b0010: z = a & b;
            4'b0011: z = a | b;
            4'b0101: z = a < b ? 1 : 0;
            4'b0100: z = a ^ b ; //xor-xori --> added
            4'b0110: z = a << b[4:0]; //sll--slli ---> added
            4'b0111: z = a >> b[4:0]; //srl--srli ---> added
            4'b1000: z = a >>> b[4:0]; //sla---> added
            4'b1001: z = s_mul[31:0]; //mul
            4'b1010: z = s_mul[63:32];//mulhss
            4'b1011: z = s_u_mul[63:32]; //mulhsu
            4'b1100: z = u_mul[63:32]; //mulhuu
            4'b1101:if (b == 0)
                    z = 32'b1;
                else if ((a == 32'h80000000) && (b == 32'b1))
                    z = 32'h80000000;
                else
                    z  = a / b; //division
            4'b1110: if (b == 0)
                    z = a;
                else if ((a == 32'h80000000) && (b == 32'b1))
                    z = 0;
                else
                    z =a % b; //rem
            default: z = 32'bx;
        endcase

    assign zero = z == 0;

endmodule
```

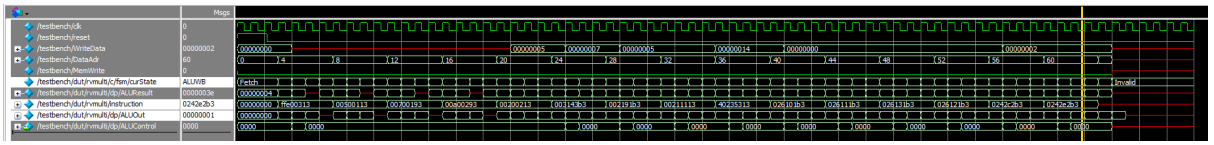
برنامه زیر برای تست پردازنده توسعه داده شده نوشته شده است:

```
main:
addi x6, x0, -2  #x6 = -2
addi x2, x0, 5   #x2 = 5
addi x3, x0, 7   #x3 = 7
addi x5, x0, 10  #x5 = 10
addi x4 , x0 , 2 #x4 = 2
xor x7, x2, x3   #x7 = 2
sll x3, x3, x2   #x3 = 224 -0xE0
slli x2, x2, 2   #X2 = 20
srai x6, x6, 2   #x6=-1
mul x3, x2, x6   #x3 = -20
mulh x3, x2, x6
mulhu x3, x2, x6
mulhsu x3, x2, x6
div x5, x5, x4  #x5 = 2
rem x5, x5,  x4  # x5= 1
```

کد ماشین برنامه مذکور به شکل زیر خواهد بود:

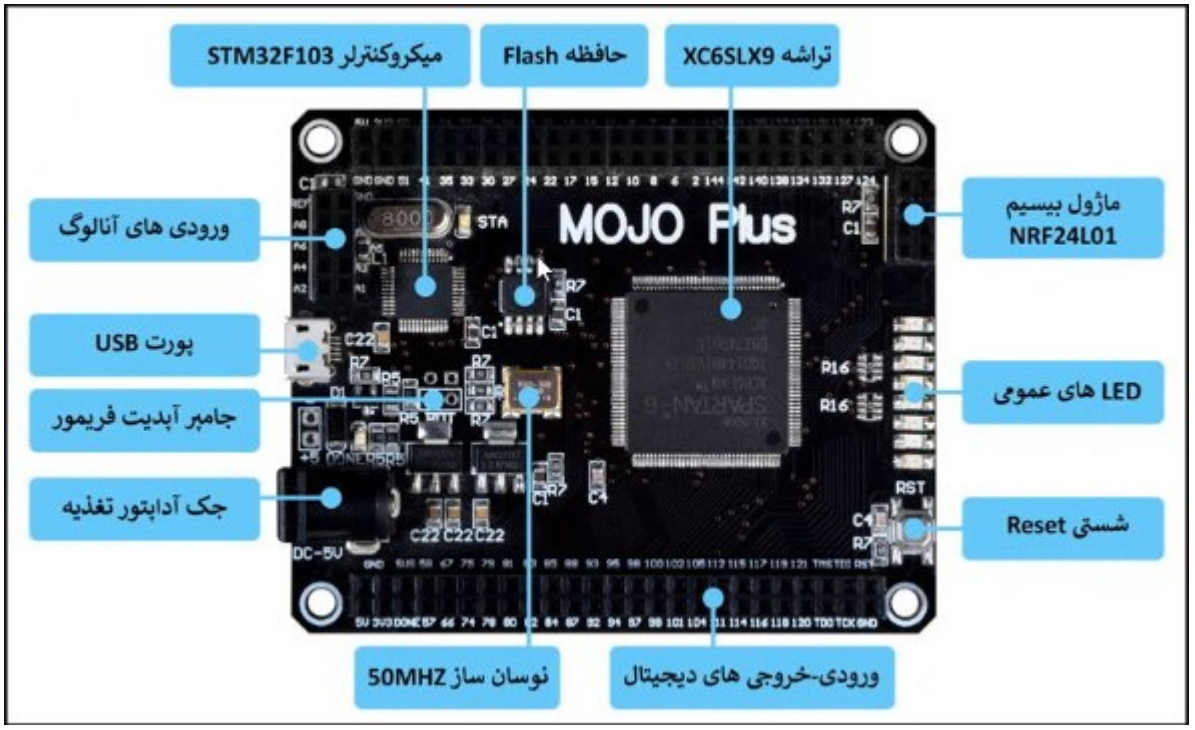
```
0xFFE00313
0x00500113
0X00700193
0X00A00293
0x00200213
0X003143B3
0x002191B3
0x00211113
0x40235313
0x026101B3
0x026111B3
0x026131B3
0x026121B3
0x0242C2B3
0x0242E2B3
```

نتیجه شبیه‌سازی این برنامه به صورت زیر خواهد بود.



مشخص است که مقدار ۱ (که در ALUOut ذخیره شده) به درستی در حین اجرای دستور آخر (0x0242E2B3) نوشته می‌شود.

بخش امتیازی(پیاده سازی روی fpga)



برای پیاده سازی روی fpga spartan6 از نرم افزار زایلینکس استفاده میکنیم.

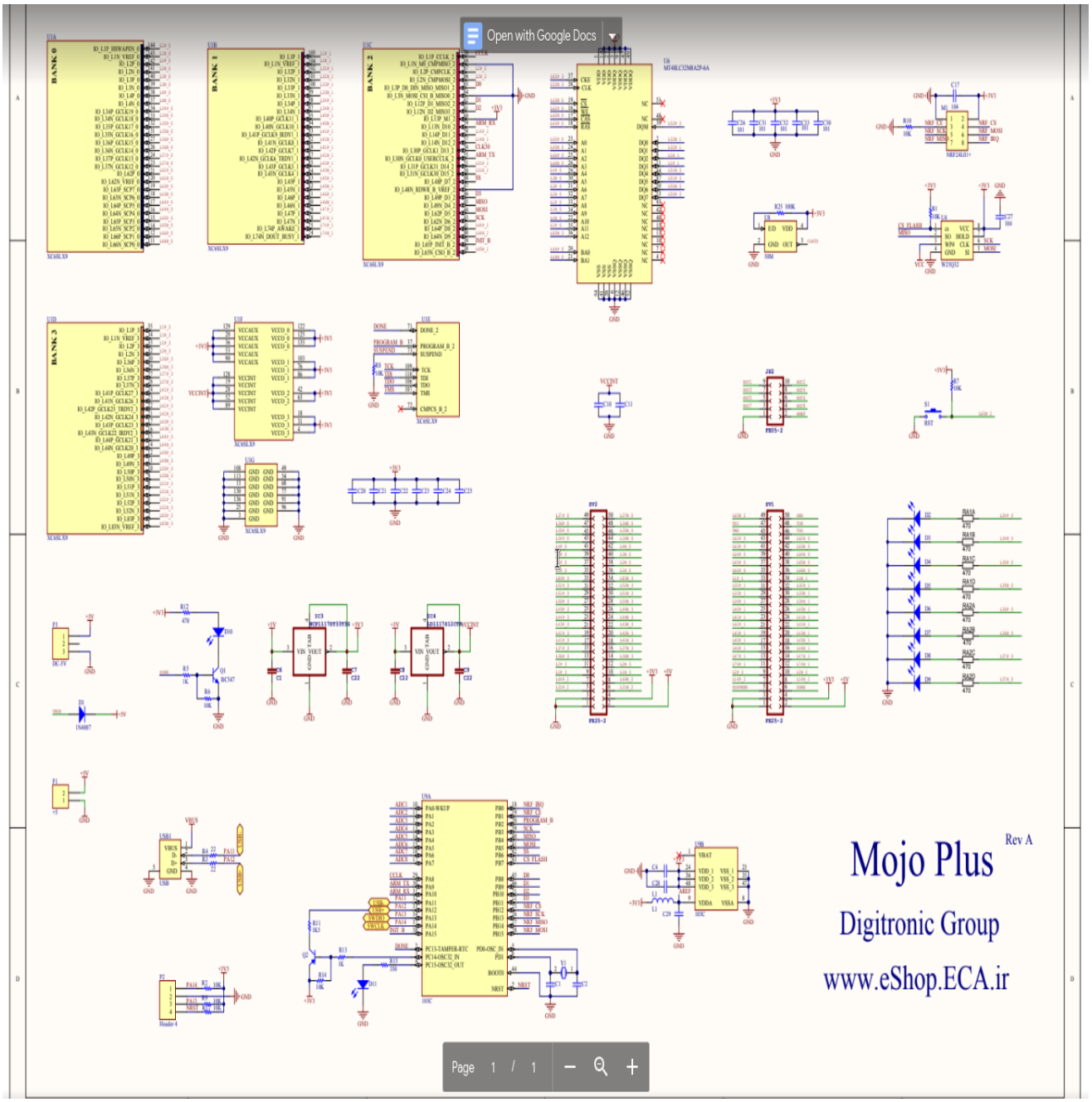
برای این کار نیازمندیم که فایل ها را به زبان وریلاگ تبدیل کنیم.

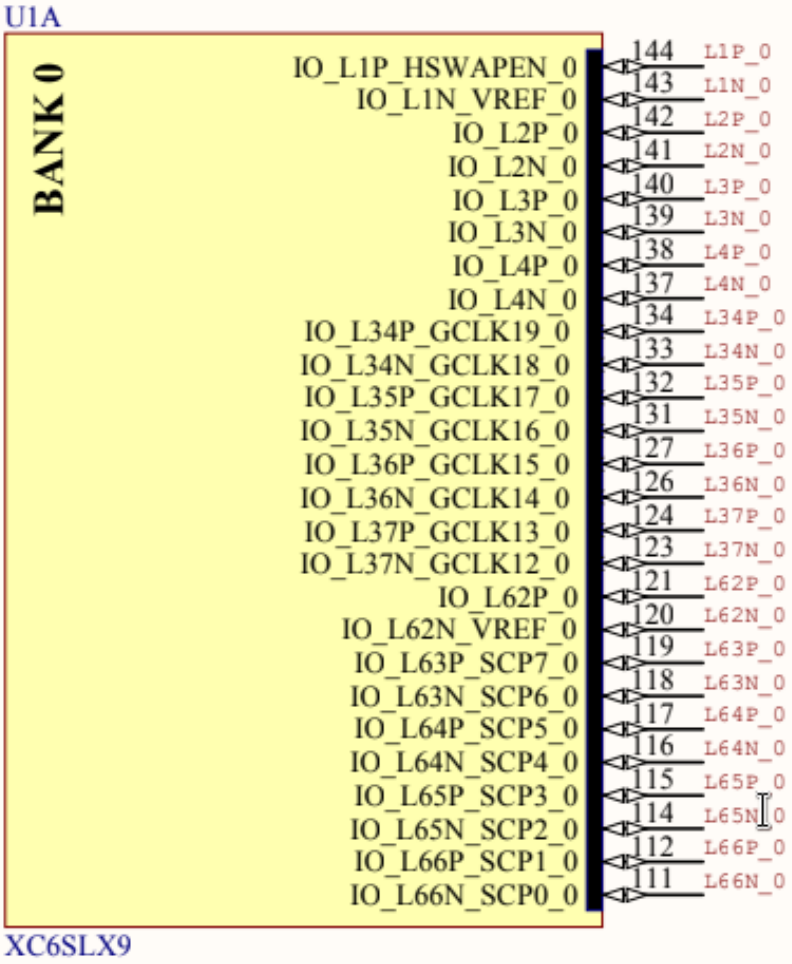
بعد از تبدیل آنها به زبان وریلاگ آن را سنتز میکنیم.

سپس یک فایلی که ورودی و خروجی ما را به fpga متصل کند میسازیم.

به دلیل کمبود تعداد پورت های fpga، میدانیم که در نهایت تست ما در نهایت عدد ۲۵ را در آدرس ۱۰۰ میریزد. پس متوان از ۵ بیت کم ارزش دیتا و ۷ بیت کم ارزش ادرس روی برد استفاده کرد. با کلاک و ریست و MemWrite جمعا به حداقل ۱۵ LED نیاز داریم اما fpga ما فقط ۸ led دارد پس با یک breadboard میتوانیم LED هایمان را به fpga متصل کنیم.

برای مشخص کردن اینکه هر led به کدام ورودی وصل شود، نیاز داریم که manual برد مورد نظر را مطالعه کنیم.

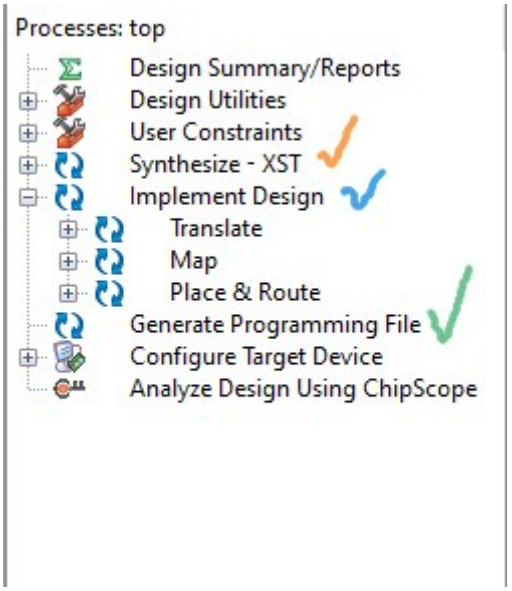



U1A
BANK 0
IO_L1P_HSWAPEN_0
IO_L1N_VREF_0
IO_L2P_0
IO_L2N_0
IO_L3P_0
IO_L3N_0
IO_L4P_0
IO_L4N_0
IO_L34P_GCLK19_0
IO_L34N_GCLK18_0
IO_L35P_GCLK17_0
IO_L35N_GCLK16_0
IO_L36P_GCLK15_0
IO_L36N_GCLK14_0
IO_L37P_GCLK13_0
IO_L37N_GCLK12_0
IO_L62P_0
IO_L62N_VREF_0
IO_L63P_SCP7_0
IO_L63N_SCP6_0
IO_L64P_SCP5_0
IO_L64N_SCP4_0
IO_L65P_SCP3_0
IO_L65N_SCP2_0
IO_L66P_SCP1_0
IO_L66N_SCP0_0
144 L1P_0
143 L1N_0
142 L2P_0
141 L2N_0
140 L3P_0
139 L3N_0
138 L4P_0
137 L4N_0
134 L34P_0
133 L34N_0
132 L35P_0
131 L35N_0
127 L36P_0
126 L36N_0
124 L37P_0
123 L37N_0
121 L62P_0
120 L62N_0
119 L63P_0
118 L63N_0
117 L64P_0
116 L64N_0
115 L65P_0
114 L65N_0
112 L66P_0
111 L66N_0
XC6SLX9

با استفاده از شماتیک های داده شده چند پین برای قرار دادن دیتا استفاده میکنیم:
فایل top.usf ما نهایتا به شکل زیر خواهد بود:
(کلاک و ریست را در خود فایل جنریت کردیم)

```
CONFIG VCCAUX=3.3;

NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 50 MHz HIGH 50%;

NET "clk" LOC = P56 | IOSTANDARD = LVTTL;
NET "rst_n" LOC = P38 | IOSTANDARD = LVTTL;


NET "WriteData<0>" LOC = P134 | IOSTANDARD = LVTTL;
NET "WriteData<1>" LOC = P133 | IOSTANDARD = LVTTL;
NET "WriteData<2>" LOC = P132 | IOSTANDARD = LVTTL;
NET "WriteData<3>" LOC = P131 | IOSTANDARD = LVTTL;
NET "WriteData<4>" LOC = P127 | IOSTANDARD = LVTTL;
NET "DataAdr<0>" LOC = P124 | IOSTANDARD = LVTTL;
NET "DataAdr<1>" LOC = P123 | IOSTANDARD = LVTTL;
NET "DataAdr<2>" LOC = P111 | IOSTANDARD = LVTTL;
NET "DataAdr<3>" LOC = P112 | IOSTANDARD = LVTTL;
NET "DataAdr<4>" LOC = P114 | IOSTANDARD = LVTTL;
NET "DataAdr<5>" LOC = P115 | IOSTANDARD = LVTTL;
NET "DataAdr<6>" LOC = P116 | IOSTANDARD = LVTTL;
NET "DataAdr<7>" LOC = P117 | IOSTANDARD = LVTTL;
```

سپس implement design را فشار داده و سپس با generate programming file فایل top.bit را که باید روی fpga ریخته شود تولید میکنیم.


Processes: top
Design Summary/Reports
Design Utilities
User Constraints
Synthesize - XST
Implement Design
Translate
Map
Place & Route
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope