



Computer Architecture

HW4 Report

G14 (Mohammad Bahrami & Navid Raeiszadeh)

سوال اول:

الف) بله لازم است یک multiplexer به datapath اضافه شود تا بتوان از بین مقدار رجیستر داده شده SrcA و PC مقدار رجیستر را انتخاب کرد تا با offset جمع شود تا PCTarget تولید شود.

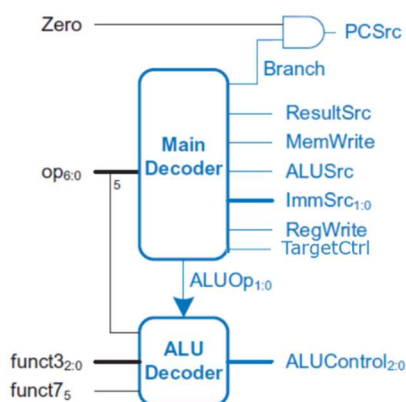
ب)

سیگنال کنترلی TargetCtrl که در شکل با رنگ قرمز نشان داده شده است برای انتخاب بین SrcA که مقدار رجیستر rs1 است و PC می باشد.

انشعابی از SrcA به مالتیپلکسری که با رنگ زرد نشان داده شده است می آوریم تا مقدارش پس از انتخاب با offset جمع شود.

(ج)

در main decoder سیگنال TargetCtrl را اضافه میکنیم.



(د)

جدول 7.6 اصلاح شده با اضافه کردن jalr و سیگنال کنترلی TargetCtrl

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	TargetCtrl
lw	0000011	1	00	1	0	01	0	00	0	1
sw	0100011	0	01	1	1	xx	0	00	0	1
R-type	0110011	1	xx	0	0	00	0	10	0	1
beq	1100011	0	10	0	0	xx	1	01	0	1
I-type ALU	0010011	1	00	1	0	00	0	10	0	1
jal	1101111	1	11	x	0	10	0	xx	1	1
jalr	1100111	1	00	x	0	10	0	xx	1	0

توضیحات تکمیلی:

1100111 (103)	000	-	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
---------------	-----	---	---	-------------------	------------------------	--------------------------------------

برای مشخص شدن مقدار PC باید رجیستر rs1 که در ورودی اول رجیستر فایل هست را با مقدار ساینده اکستند شده ی آفست جمع کرد که نوع ساینده اکستند از جنس I type هاست که کد 00 را به خود اختصاص داده است.

برای اینکار از جمع کننده PCTarget استفاده میکنیم و برای انتخاب مقدار SrcA نیاز به یک مالتیپلکسر داریم که در قسمت الف شرح داده شد.

مقدار PC + 4 که باید در rd قرار گیرد به مالتیپلکسر بعد از DataMemory وارد میشود و در آنجا انتخاب شده و به پورت رایت رجیستر فایل رفته تا در rd نوشته شود.

سوال دوم :

برای اضافه کردن سه دستور `addi, jal, jalr` کافی است `datapath` و `controller` رو مانند شکل سوال یک تغییر داد.

کد هایی که در منبع هریس برای مدل HDL پردازنده `single cycle` بود رو کپی کرده و تغییرات لازم رو اعمال میکنیم.

ماژول top :

```
module top(input logic clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic MemWrite);
    logic [31:0] PC, Instr, ReadData;
    // instantiate processor and memories
    riscvsingle rvsingle(clk, reset, PC, Instr, MemWrite,
                        DataAdr, WriteData, ReadData);
    imem imem(PC, Instr);
    dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

بدون تغییر باقی میماند.

ماژول riscvsingle :

```

1  module riscvsingle( input logic clk, reset,
2                      output logic [31:0] PC,
3                      input logic [31:0] Instr,
4                      output logic MemWrite,
5                      output logic [31:0] ALUResult, WriteData,
6                      input logic [31:0] ReadData);
7
8      logic ALUSrc, RegWrite, Jump, TargetCtrl, Zero;
9      logic [1:0] ResultSrc, ImmSrc;
10     logic [2:0] ALUControl;
11
12     controller c( Instr[6:0], Instr[14:12], Instr[30], Zero,
13                 ResultSrc, MemWrite, PCSrc,
14                 ALUSrc, RegWrite, Jump, TargetCtrl,
15                 ImmSrc, ALUControl);
16
17     datapath dp( clk, reset, ResultSrc, PCSrc,
18                 ALUSrc, TargetCtrl, RegWrite,
19                 ImmSrc, ALUControl,
20                 Zero, PC, Instr,
21                 ALUResult, WriteData, ReadData);
22
23 endmodule
24

```

سیگنال TargetCtrl که در سوال یک به عنوان سلکت مالتیپلکسر اضافه شده بود را به کد نیز اضافه میکنیم.

ماژول imem :

```

module imem(input logic [31:0] a,
            output logic [31:0] rd);
    logic [31:0] RAM[63:0];

    initial
        $readmemh("D:\\Lessons\\Computer Architecture\\HW\\HW4\\riscvtest.txt",RAM);

    assign rd = RAM[a[31:2]]; // word aligned

endmodule

```

آدرس فایل تکست ماشین کد دستورالعمل هارا در readmemh قرار میدهیم.

ماژول dmem :

```
module dmem(input logic clk, we,
            input logic [31:0] a, wd,
            output logic [31:0] rd);
    logic [31:0] RAM[63:0];

    assign rd = RAM[a[31:2]]; // word a

    always_ff @(posedge clk)
        if (we) RAM[a[31:2]] <= wd;

endmodule
```

بدون تغییر.

ماژول controller :

```
module controller( input logic [6:0] op,
                   input logic [2:0] funct3,
                   input logic funct7b5,
                   input logic Zero,
                   output logic [1:0] ResultSrc,
                   output logic MemWrite,
                   output logic PCSrc, ALUSrc,
                   output logic RegWrite, Jump, TargetCtrl,
                   output logic [1:0] ImmSrc,
                   output logic [2:0] ALUControl);
    logic [1:0] ALUOp;
    logic Branch;
    maindec md( op, ResultSrc, MemWrite, Branch,
                ALUSrc, RegWrite, Jump, ImmSrc, ALUOp, TargetCtrl);
    aludec ad( op[5], funct3, funct7b5, ALUOp, ALUControl);
    assign PCSrc = Branch & Zero | Jump;
endmodule
```

سیگنال TargetCtrl را به main decoder می‌دهیم.

ماژول datapath :


```

module datapath(input logic clk, reset,
               input logic [1:0] ResultSrc,
               input logic PCSrc, ALUSrc, TargetCtrl,
               input logic RegWrite,
               input logic [1:0] ImmSrc,
               input logic [2:0] ALUControl,
               output logic Zero,
               output logic [31:0] PC,
               input logic [31:0] Instr,
               output logic [31:0] ALUResult, WriteData,
               input logic [31:0] ReadData);

    logic [31:0] PCNext, PCPlus4, PCTarget;
    logic [31:0] ImmExt;
    logic [31:0] SrcA, SrcB;
    logic [31:0] Result;
    // the wire between targetmux and PCAdder
    logic [31:0] TargetWire;
    // next PC logic
    flopr #(32) pcreg(clk, reset, PCNext, PC);
    adder pcadd4(PC, 32'd4, PCPlus4);
    // added mux for jalr instruction
    mux2 #(32) targetmux(SrcA, PC, TargetCtrl, TargetWire);
    adder pcaddbranch(TargetWire, ImmExt, PCTarget);
    mux2 #(32) pcmux(PCPlus4, PCTarget, PCSrc, PCNext);
    // register file logic
    regfile rf( clk, RegWrite, Instr[19:15], Instr[24:20],
               Instr[11:7], Result, SrcA, WriteData);
    extend ext(Instr[31:7], ImmSrc, ImmExt);
    // ALU logic
    mux2 #(32) srcbmux(WriteData, ImmExt, ALUSrc, SrcB);
    alu alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
    mux3 #(32) resultmux( ALUResult, ReadData, PCPlus4,
                          ResultSrc, Result);
endmodule

```

باس TargetWire بعد از مالتیپلکسر قرار دارد و وارد adder pc target میشود

تغییر اصلی در دیتاپث اینجا اعمال شده که بین PC و SrcA یک مقدار برای جمع شدن با آفست انتخاب میشود که SrcA برای دستور jalr هست.

ماژول maindecoder :

```
1 module maindec( input logic [6:0] op,  
2                 output logic [1:0] ResultSrc,  
3                 output logic MemWrite,  
4                 output logic Branch, ALUSrc,  
5                 output logic RegWrite, Jump,  
6                 output logic [1:0] ImmSrc,  
7                 output logic [1:0] ALUOp,  
8                 output logic TargetCtrl);  
9     logic [11:0] controls;  
10    assign {RegWrite, ImmSrc, ALUSrc, MemWrite,  
11           ResultSrc, Branch, ALUOp, Jump, TargetCtrl} = controls;  
12    always_comb  
13    case(op)  
14        // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump_TargetCtrl  
15        7'b0000011: controls = 12'b1_00_1_0_01_0_00_0_1; // lw  
16        7'b0100011: controls = 12'b0_01_1_1_00_0_00_0_1; // sw  
17        7'b0110011: controls = 12'b1_xx_0_0_00_0_10_0_1; // R-type  
18        7'b1100011: controls = 12'b0_10_0_0_00_1_01_0_1; // beq  
19        7'b0010011: controls = 12'b1_00_1_0_00_0_10_0_1; // I-type ALU  
20        7'b1101111: controls = 12'b1_11_0_0_10_0_00_1_1; // jal  
21        7'b1100111: controls = 12'b1_00_0_0_10_0_00_1_0; // jalr  
22  
23        default: controls = 12'bx_xx_x_x_xx_x_xx_x_x; // ???  
24    endcase  
25 endmodule
```

سیگنال TargetCtrl را به controls کانکت میکنیم و که در این صورت سیگنال controls دوازده بیتی خواهد شد و بیت آخر مربوط به TargetCtrl خواهد بود.

جدول درستی در سوال یک آورده شده است.

ماژول aludec :

```

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 R-type subtract
    always_comb
        case(ALUOp)
            2'b00: ALUControl = 3'b000; // addition
            2'b01: ALUControl = 3'b001; // subtraction
            default: case(funct3) // R-type or I-type 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

```

بدون تغيير.

ماژول flopr و flopenr:

```

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

```

```

module flopenr #(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

```

هر دو بدون تغییر.

ماژول های mux2, mux3 :

```

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

    assign y = s ? d1 : d0;

endmodule

```

```

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

```

بدون تغییر.

ماژول extend :

بدون تغییر

```

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

```

ماژول adder :

```

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

```

بدون تغيير.

ماژول regfile :

```

module regfile(input logic clk,
               input logic we3,
               input logic [4:0] a1, a2, a3,
               input logic [31:0] wd3,
               output logic [31:0] rd1, rd2);
    logic [31:0] rf[31:0];
    // three ported register file
    // read two ports combinationaly (A1/RD1, A2/RD2)
    // write third port on rising edge of clock (A3/WD3/WE3)
    // register 0 hardwired to 0
    always_ff @(posedge clk)
        if (we3) rf[a3] <= wd3;

    assign rd1 = (a1 != 0) ? rf[a1] : 0;
    assign rd2 = (a2 != 0) ? rf[a2] : 0;

endmodule

```

ورودی های a1,a2,a3 باید 5 بیتی باشند که در کد کتاب 6 بیتی بودند.

ماژول alu :

```

module alu (input logic [31:0] SrcA, SrcB,
            input logic [2:0] ALUControl,
            output logic [31:0] ALUResult,
            output logic Zero);

    always_comb begin
        case(ALUControl)
            3'b000: ALUResult = SrcA + SrcB;
            3'b001: ALUResult = SrcA - SrcB;
            3'b010: ALUResult = SrcA & SrcB;
            3'b011: ALUResult = SrcA | SrcB;
            3'b101: ALUResult = SrcA < SrcB;
            default: ALUResult = 32'bx;
        endcase
    end

    assign Zero = ALUResult == 0;

endmodule

```

5 دستور مورد نیاز پردازنده که در جدول کتاب هم آمده است را پیاده سازی میکنیم.

تست بنچ:

در این تست فایل riscvtest.txt که حاوی ماشین کد اسمبلی مورد نظر ما میباشد را خوانده و اگر پس از اتمام خواندن در خانه 100 مموری مقدار 25 رایت شده باشد موفقیت آمیز خواهد بود.


```

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

```

تغییری نسبت به کد کتاب نداشته است .

کد اسمبلی:

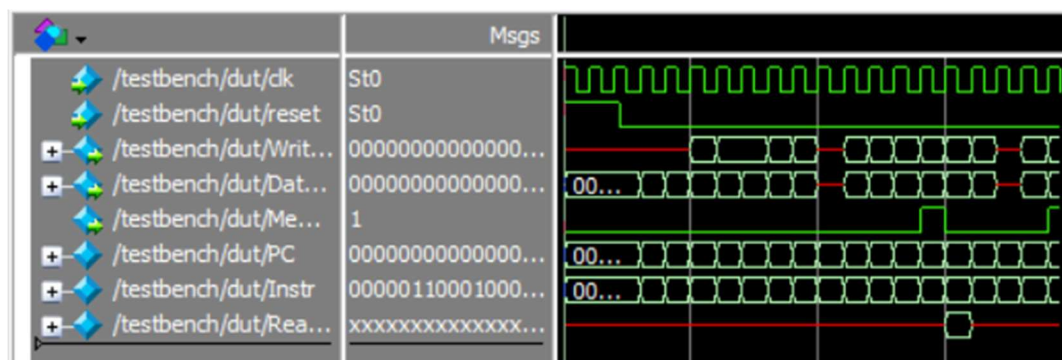

```

1 main:  addi x2, x0,5 #x2 = 5 0 00500113
2        addi x3, x0, 12 # x3 = 12 4 00C00193
3        addi x7, x3, -9 # x7 = (12 - 9) = 3 8 FF718393
4        or x4, x7, x2 # x4 = (3 OR 5) = 7 C 0023E233
5        and x5, x3, x4 # x5 = (12 AND 7) = 4 10 0041F2B3
6        add x5, x5, x4 # x5 = 4 + 7 = 11 14 004282B3
7        beq x5, x7, end #shouldn't be taken 18 02728863
8        slt x4, x3, x4 # x4 = (12 < 7) = 0 1C 0041A233
9        jalr ra, x3,28# should be taken. jump to around label and link the next line.
10       j done # will be executed after line (jalr x0, ra, 0)
11 around: slt x4, x7, x2 # x4 = (3 < 5) = 1 28 0023A233
12        add x7, x4, x5 # x7 = (1 + 11) = 12 2C 005203B3
13        sub x7, x7, x2 # x7 = (12 - 5) = 7 30 402383B3
14        sw x7, 84(x3) #[96] = 7 34 0471AA23
15        lw x2, 96(x0) #x2 = [96] = 7 38 06002103
16        add x9, x2, x5 # x9 = (7 + 11) = 18 3C 005104B3
17        jal x3, end # jump to end, x3 = 0x44 40 008001EF
18        addi x2, x0,1 #shouldn't execute 44 00100113
19 end:    add x2, x2, x9 # x2 = (7 + 18) = 25 48
20        sw x2, 100(x0) # [100] = 25 0221A023
21        jalr x0, ra, 0 # should be taken and go to line (j done) and not falling into infinite loop trap
22 loop:  beq x0, x0, loop # infinite loop. shouldn't be executed
23 done:  addi x0, x0, 0 # the end

```

در اینجا در خط 9 م برنامه با استفاده از دستور jalr مقدار رجیستر x3 که 12 میباشد را با 28 جمع کرده تا به PC به مقدار 40 برسد که آدرس شروع لیبل around است و ازین به بعد تا خط 21 مشابه کد کتاب است و پس از رایت کردن مقدار 25 در خانه 100 مموری با استفاده از دستور jalr x0, ra, 0 به خط 10 پرش کرده و از آنجا به لیبل done جامپ میکنیم و اگر jalr به درستی اجرا نشود به لیبل loop میرسیم که یک حلقه بی نهایت اجرا میشود.

شکل موج در سیمولیشن:



توقف اجرا برنامه و موفقیت آمیز بودن:

```
1 module testbench();
2     logic clk;
3     logic reset;
4     logic [31:0] WriteData, DataAdr;
5     logic MemWrite;
6     // instantiate device to be tested
7     top dut(clk, reset, WriteData, DataAdr, MemWrite);
8     // initialize test
9     initial
10         begin
11             reset <= 1; # 22; reset <= 0;
12         end
13     // generate clock to sequence tests
14     always
15         begin
16             clk <= 1; # 5; clk <= 0; # 5;
17         end
18     // check results
19     always @(negedge clk)
20         begin
21             if(MemWrite) begin
22                 if(DataAdr === 100 & WriteData === 25) begin
23                     $display("Simulation succeeded");
24                     $stop;
25                 end else if (DataAdr !== 96) begin
26                     $display("Simulation failed");
27                     $stop;
28                 end
29             end
30         end
31 endmodule
```

سوال سوم:

در این سوال از ما تغییرات T با تغییر زمان T_{alu} خواسته شده است.

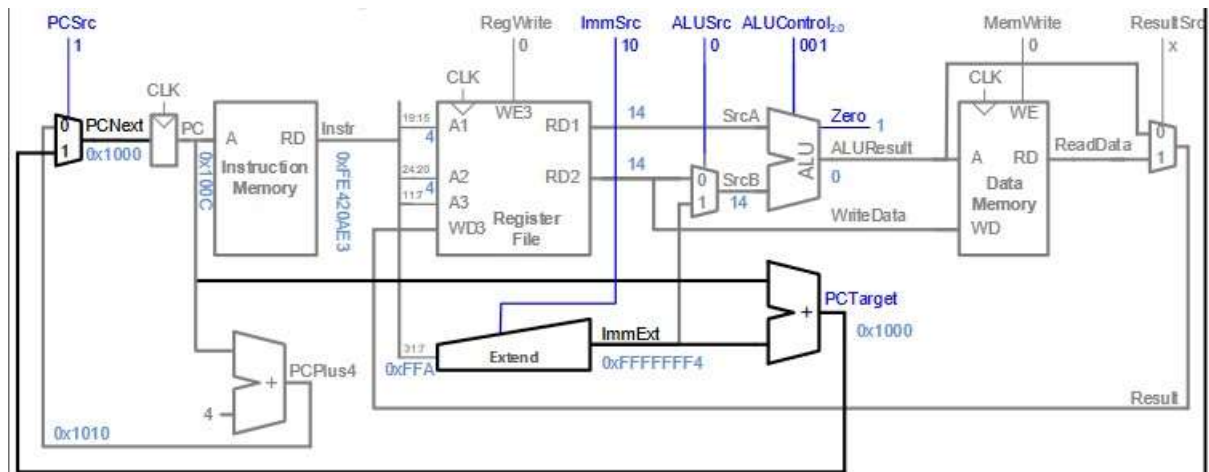
$$T_{c-single} = t_{pcq-pc} + 2 t_{mem} + t_{RF_{read}} + t_{alu} + t_{mux} + t_{RF_{setup}} \quad \text{سوال ۳}$$

$$f_o + f_{oo} + 100 + 30 + 90 + 90 = 720 \text{ ps}$$

$$\rightarrow 10 \times 10^9 \times 720 \times 10^{-12} = 72 \text{ s} \quad \text{برای پیکو}$$

$$\text{Execution time} = \text{instruction} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

```
: PC src
```



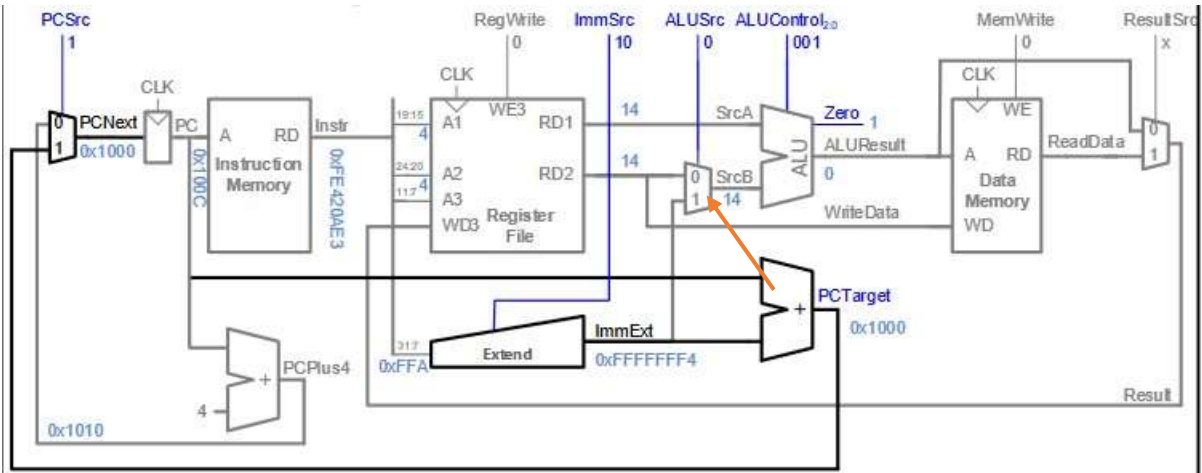
pc src : خب الان سواله ماميله انه اس سئل ان اسك بيشه ، دوني هميشه

مشخصاً تمامی دستوراتی که با PC سروکار دارن و تغییرش میدچار اخلال و تغییر مینن

jump ها، مانند `avl`، برای فراخوانی تابع دیگر است

ساز دارند.

- : ALU src



Alu Src : مطابق با data path می باشد ✓
Alu source می شود ✓

کنکری، enable، مالتی پلکسر، λ^2 کہ خروجی آن مستقیماً β ، src ، Alu ، واصل می شود.

و در حاشیه هم از imm extend، rdx است. در صورت 1 Always بدان آن دستور العمل ها

در عمل روش های ساده ای Alu چهار مشکل خواهد شد، اساساً rd_2 وارد Alu خواهد شد.

دستورات مرتبہ R ٹایپ، بالخصوص: `and`, `nor`, `add`, `sub`, `or`, ...

• بیت اول imm src :

• بیت اول imm src : در این صورت تمامی جاهایی که با immediate کار داریم، احتمالاً دچار مشکل خواهد شد. حالا چرا احتمالاً؟! چون مقدار imm را پردازنده به ۳۲ بیت افزایش می‌دهد و اگر آنها خود ۱۲ بیت باشند، اشتباه رخ خواهد داد. تمامی دستورات s type و type I که با immediate کار دارند به مانند: jalr, lui, andi, xori, addi, lw, ...

• بیت دوم result src :

• بیت دوم result src : سیگنال result src، سیگنال خط انتخاب (enable) هالی نامی است که ورودی آن Alu result و Read data از Data Memory است و خروجی آن هم در register File رایت می‌شود. بنابراین تمامی دستورات که مقدار است با مقدار نتیجه کارند می‌تواند غلط باشد. مانند R تاییپها: sra, or, xor, and, add, ...

