تمرین ۴

سوال ۱

با توجه به کارکرد این دستور نیاز به مسیری وجود دارد که `rs1+ sign extend(imm)` محاسبه و نتیجه روی `PCNext` قرار گیرد. همچنین مسیری باید وجود داشته باشد تا `PC+4` در `registr file` ذخیره شود (که این مسیر از طریق `result` موجود است.) با اضافه کردن یک `mux` و با منشعب کردن خروجی `ALU` و اضافه کردن سیگنال کنترلی `PCResultSrc` مسیر داده تکمیل می‌شود.

```
jalr x0,x1,0    # PC = rs1 + sign extend(imm), rd=PC+4
```

I-Type

hex	imm	rs1	funct3	rd	op	Assembly
00008067	000000000000	00001	000	00000	1100111	jalr zero,ra,0

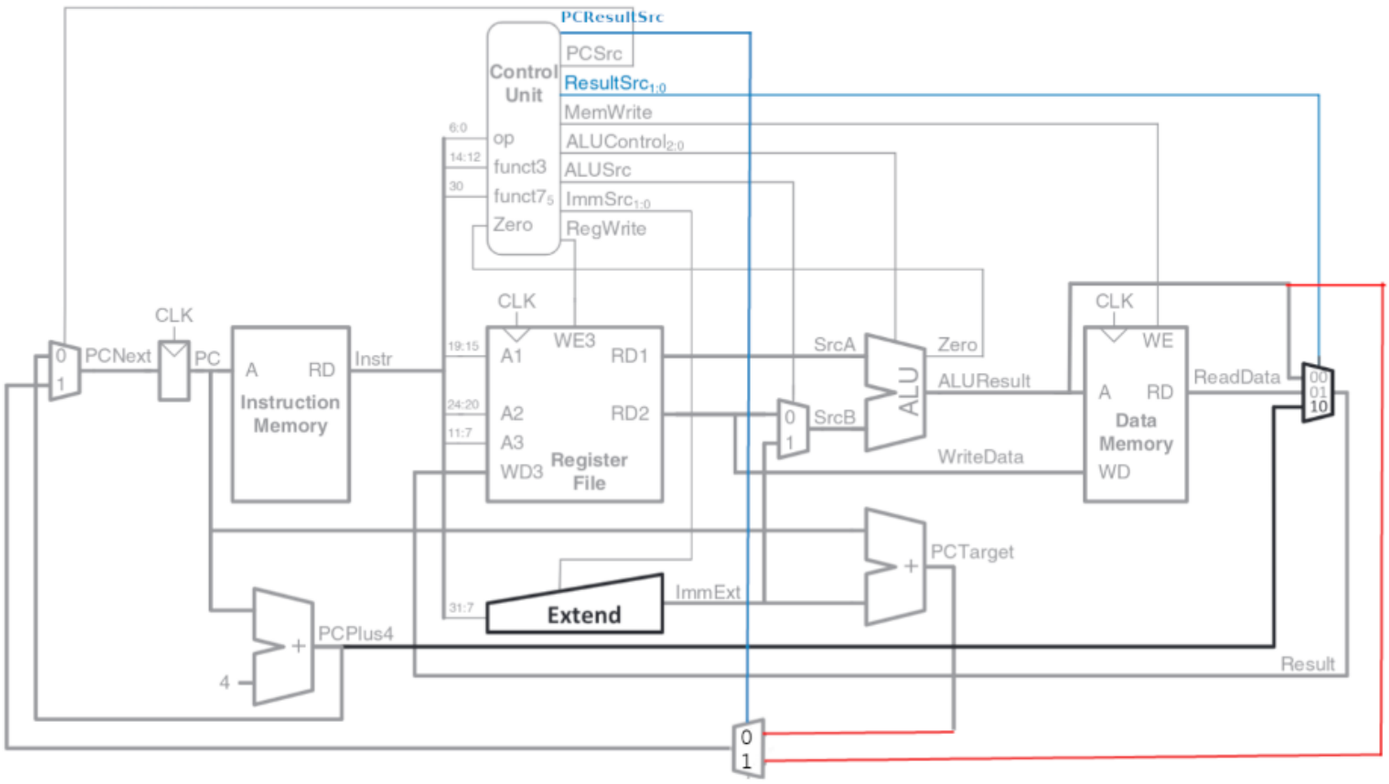


Table 7.3

ALUOp	funct3	{op5, funct75}	ALUControl	Instruction
00	x	x	000 (add)	lw, sw, jalr
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

Table 7.6

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	PCResul
lw	0000011	1	00	1	0	01	0	00	0	x
sw	0100011	0	01	1	1	xx	0	00	0	x
R-Type	0110011	1	xx	0	0	00	0	10	0	x
beq	1100011	0	10	0	0	xx	1	01	0	0
I-type ALU	0010011	1	00	1	0	00	0	10	0	x

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump	PCResul
jalr	1100111	1	00	1	0	10	0	00	1	1

سوال ۲

با استفاده از تغییرات موجود در سوال یک روی data path کد را تغییر می‌دهیم تا بتوانیم از jalr پشتیبانی کنیم. این تغییرات با کامنت‌های `// changed by me for jalr` قابل مشاهده هستند. به صورت کلی این تغییرات شامل اضافه کردن سیگنال کنترلی `PCResultSrc` و سیگنال میانی `PCResultTarget` و اضافه کردن `pcsrcmux` میباشد.

همچنین با توجه به جدول ۷.۶ مقادیر مورد تحلیل تابع `maindec` دستخوش تغییر شده اند.

شرح تغییرات در تصاویر زیر:

design.sv

```
module riscvsingle (input logic clk,reset,
            output  logic   [31:0]   PC,
                input   logic   [31:0]    Instr,
                output  logic              MemWrite,
                output  logic  [31:0]   ALUResult, WriteData,
                input     logic  [31:0]  ReadData);
    logic           ALUSrc, RegWrite, Jump, Zero;
    logic [1:0] ResultSrc, ImmSrc;
    logic [2:0] ALUControl;
    controller c(Instr[6:0], Instr[14:12], Instr[30], Zero,
                 ResultSrc, MemWrite, PCSrc,
            ALUSrc, RegWrite, Jump, ImmSrc,  ALUControl,PCResultSrc);   // changed by me for jalr
    datapath dp(clk, reset, ResultSrc, PCSrc,
            ALUSrc,  RegWrite, ImmSrc,
            ALUControl, Zero, PC, Instr,
            ALUResult, WriteData, ReadData,PCResultSrc);                // changed by me for jalr
endmodule
```

```
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, ALUSrc,
                      output logic           PCSrc,
                      output logic           RegWrite, Jump,
                      output logic [1:0] ImmSrc,
                  output logic [2:0] ALUControl,
                      output logic PCResultSrc);                // changed by me for jalr
    logic [1:0] ALUOp;
    logic           Branch;

    maindec md(op, ResultSrc, MemWrite, Branch,
                   ALUSrc, RegWrite, Jump, ImmSrc, ALUOp,jalr); // changed by me for jalr

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

  assign PCSrc = Branch & Zero | Jump;
  assign PCResultSrc = jalr;                                    // changed by me for jalr
endmodule
```

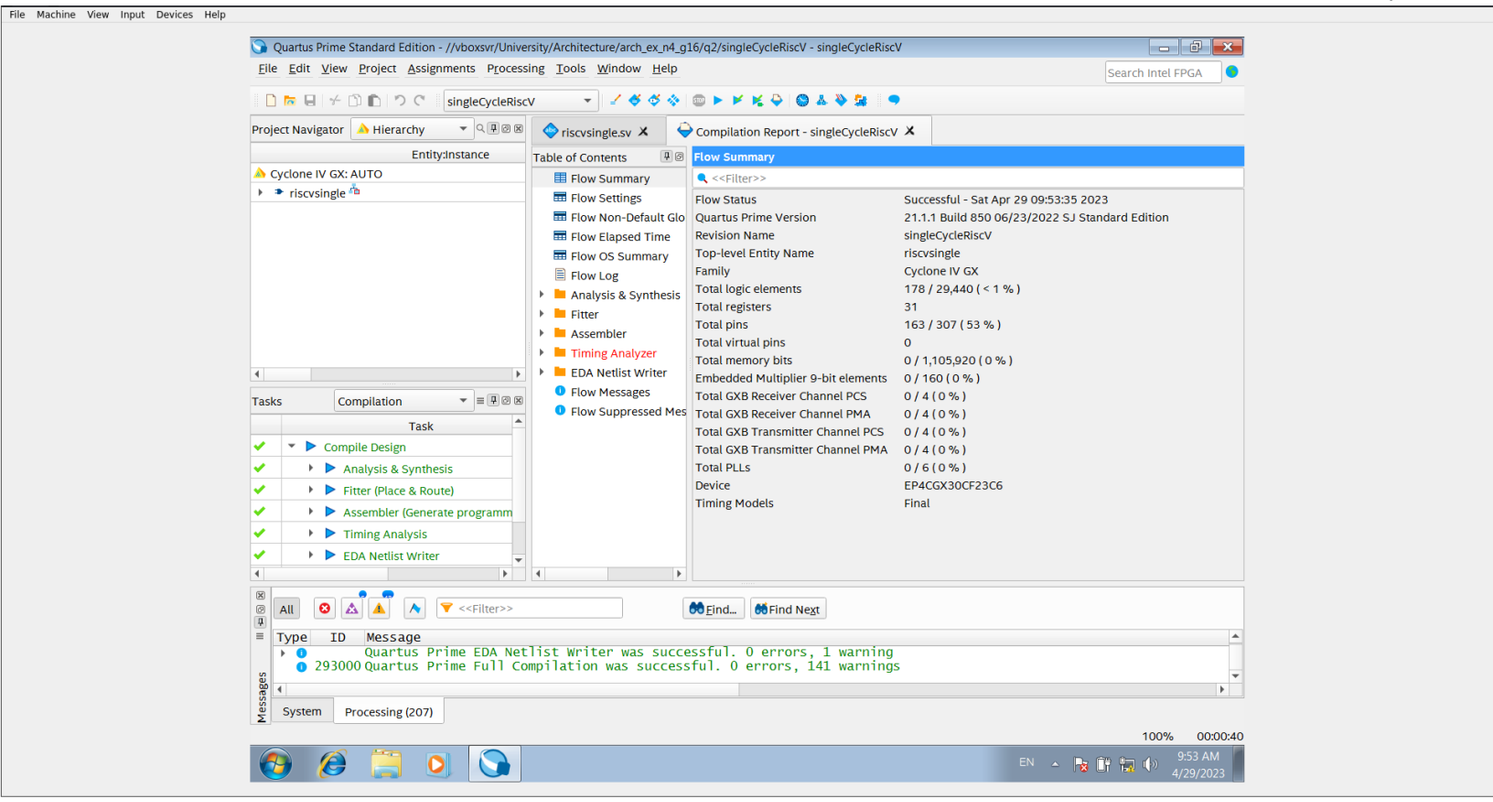
```
module datapath(input     logic             clk,
        reset,input     logic [1:0]    ResultSrc,
        input     logic PCSrc, ALUSrc,
        input     logic              RegWrite,
        input     logic [1:0]     ImmSrc,
        input     logic [2:0]     ALUControl,
        output logic Zero, output logic [31:0] PC,
        input     logic  [31:0]  Instr,
        output logic [31:0] ALUResult, WriteData,
        input     logic  [31:0]  ReadData,
                input   logic   PCResultSrc);           // changed by me for jalr

    logic [31:0] PCNext, PCPlus4, PCTarget;
    logic [31:0] ImmExt;
    logic [31:0] SrcA, SrcB;
    logic [31:0] Result;    // next PC logic

    flopr #(32) pcreg(clk, reset, PCNext, PC);

    adder           pcadd4(PC, 32'd4, PCPlus4);
    adder           pcaddbranch(PC, ImmExt, PCTarget);

    mux2 #(32)     pcmux(PCPlus4, PCResultTarget, PCSrc, PCNext);    // register file logic    // changed by me for jalr

    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);

  mux2 #(32)   pcsrcmux( ALUResult,PCTarget, PCResultSrc,PCResultTarget );    // changed by me for jalr

    alu             alu(SrcA, SrcB, ALUControl, ALUResult, Zero);
    mux3 #(32)   resultmux( ALUResult, ReadData, PCPlus4,ResultSrc, Result);

endmodule
```

```
module maindec(input     logic [6:0] op,output logic [1:0] ResultSrc,
               output logic           MemWrite,
               output logic           Branch, ALUSrc,
           output logic           RegWrite, Jump,
                output logic [1:0] ImmSrc,output logic [1:0] ALUOp,output logic jalr);   // changed by me for jalr
  logic [11:0] controls;                                    // changed by me for jalr
    assign {RegWrite, ImmSrc, ALUSrc, MemWrite,
    ResultSrc, Branch, ALUOp, Jump,jalr} = controls;    // changed by me for jalr
    always_comb                                         // changed by me for jalr
    case(op)
        // RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump
        7'b0000011: controls = 11'b1_00_1_0_01_0_00_0x; // lw
            7'b0100011: controls = 11'b0_01_1_1_00_0_00_0x; // sw
            7'b0110011: controls = 11'b1_xx_0_0_00_0_10_0x; // R?type
            7'b1100011: controls = 11'b0_10_0_0_00_1_01_00; // beq
      7'b0010011: controls = 11'b1_00_1_0_00_0_10_0x; // I?type ALU         // addi already supported(also check the ALU).
            7'b1101111: controls = 11'b1_11_0_0_10_0_00_10; // jal          // jal already support.
            7'b1100111: controls = 11'b1_11_0_0_10_0_00_11; // jalr
        default:        controls = 11'bx_xx_x_x_xx_x_xx_xx; // ???
    endcase
endmodule
```

باقی دستورات ذکر شده نظیر addi و jal در کد کتاب بدون ایجاد تغییرات پشتیبانی می‌شوند. برخی از این خطوط در کد با کامنت‌هایی مشخص شده است. این دستورات در کد اسمبلی تست هم مورد بررسی قرار گرفته اند و کار میکنند.

تغییر ما شامل یک jal تایید شده به یک برچسب با نام dummy می‌باشد. سپس با استفاده از

فضای compile کد:



کد مربوط به testbench و کدهای اسمبلی به پیوست در q2 قابل مشاهده هستند.

```
jal dummy                       # ra=PC+4, jumps to dummy
addi t1,t1,-8                   # set t1 to ok
jalr zero,t1,0                  # PC=t1+0
ok: sw x2, 0x20(x3)             # [100] = 250221A023
done: beq x2, x2, done          # infinite loop5000210063
dummy: jalr t1,ra,0             # PC=ra+0,t1=PC+4
```

این تکه کد jalr را به خوبی تست میکند. به این شکل که هر دو عملیات تغییر PC و تغییر rd به PC+4 تست می‌شود. در صورتی که هر یک از این عملیات موفق نشود مقدار نهایی در حافظه ذخیره نخواهد شد

simulation commands:

```
vlog -reportprogress 300 -work work //VBOXSVR/University/Architecture/arch_ex_n4_g16/q2/riscvsingle.sv
vlog -reportprogress 300 -work work //VBOXSVR/University/Architecture/arch_ex_n4_g16/q2/testBench.sv
vsim -voptargs=+acc=lprn testbench
add wave -position insertpoint sim:/testbench/*
```

سوال ۳

برطبق جدول ۷.۷ میدانیم که تاخیر واحد `ALU` برابر `120` پیکوثانیه است؛ اما اگر از جمع کننده `Prefix Adder` استفاده کنیم این مقدار به `90` پیکوثانیه کاهش می یابد.

برای بدست آوردن زمان چرخه ما ابتدا باید `Critical Path` را پیدا کنیم. در پردازنده `RISC-V` میدانیم که دستور `lw` طولانی ترین دستور است و `Critical Path` آن در شکل زیر نمایش داده شده است.

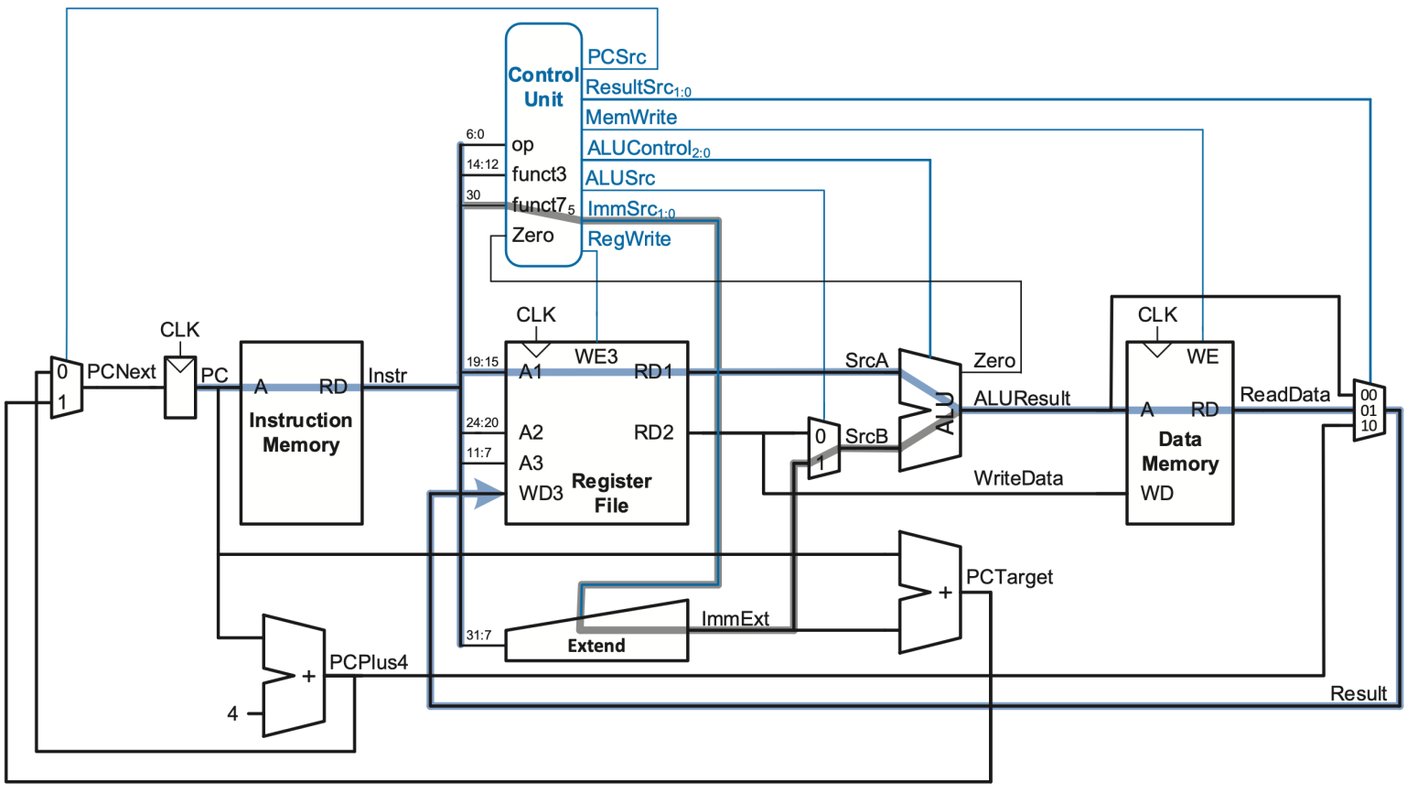


Figure 7.17 Critical path for `lw`

که در اینجا ما تاخیر چندین واحد از جمله `Data Path` و `Control Unit` را داریم که معادله زیر نشان دهنده این تاخیر ها در اجرای دستور `lw` است.

$$T_{c\_single} = t_{pcq\_PC} + t_{mem} + \max[t_{RFread}, \quad t_{dec} + t_{ext} + t_{mux}] + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

که با ساده سازی در نهایت به معادله زیر خواهیم رسید.

$$T_{c\_single} = t_{pcq\_PC} + 2t_{mem} + t_{RFread} + t_{ALU} + t_{mux} + t_{RFsetup}$$

حالا با داشتن معادله بالا و جدول ۷.۷ کتاب، زمان یک چرخه را محاسبه میکنیم.

```
T = 40 + 2(200) + 100 + 90 + 30 + 60 = 720 ps
```

همچنین قبل تر خواندیم که زمان اجرای یک برنامه، طبق فرمول زیر بدست می آید.

$$ExecutionTime = (\#instructions)\left(\frac{cycles}{instruction}\right)\left(\frac{seconds}{cycle}\right)$$

با جایگذاری مقادیر، در معادله زمان اجرای برنامه با `10` میلیارد دستور را خواهیم داشت.

```
Execution Time = (10 * 10^9 instruction) (1 cycle/instruction) (720 * 10^-12 s/cycle) = 7.2 seconds
```

سوال ۴

میدانیم که سیگنال کنترلی `PCSrc` ، ورودی `Istruction Memory` را انتخاب میکند که یا از `Extend` بیاید و یا از `PCPlus4` که چسبیده به ۱ بودن این سیگنال باعث میشود که این سیگنال همواره از واحد `Extend` بیاید و درواقع و دو دستور `jal` و `beq` انجام شوند که این باعث تداخل در برنامه مورد نظر میشود.

سیگنال کنترلی ALUSrc ، تصمیم میگیرد که کدام ورودی مالتیپلکسر قبل ALU ، روی یکی از ورودی های ALU قرار بگیرد؛ درواقع برای هر دستور ممکن است ورودی از قسمت Extend و یا Register File بیاید که به ترتیب سیگنال ImmExt و RD2 میشوند. که اگر سیگنال کنترلی ALUSrc چسبیده به ۱ باشد، باعث میشود که برای دستورات R-Type و lw و sw مشکل بوجود بیاید؛ چون همه این دستورات، ALUSrc برابر ۱ دارند و اختلال این سیگنال، باعث تداخل در برنامه و عدم اجرای درست آن میشود. جدای این دستورات، برای بقیه دستورات که سیگنال ALUSrc آنها برابر صفر است نیز مشکل بوجود میاید و تنها دستور jal به درستی اجرا میشود، چون که این سیگنال در این دستور فاقد اهمیت میباشد.

سیگنال ImmSrc دارای ۲ بیت است که نحوه extend کردن واحد Extend را نشان میدهد که بر اساس هر دستور چگونه باید اینکار را انجام دهد (Sign-Extend و یا ...)؛ چسبیده به ۱ بودن بیت اول این سیگنال باعث میشود که دو دستور jal و sw که بیت اول سیگنال ImmSrc آنها ۱ است، دچار تداخل شوند و عمل extend به درستی انجام نشود. در ضمن برای بقیه دستورات بجز دستورات R-Type که درواقع این سیگنال xx است، نیز مشکل ایجاد میشود چون آن دستورات نیز نمیتوانند اجرا شوند چون بیت اول تمام آنها صفر است.

برای سیگنال ResultSrc میدانیم که اگر همان مجموعه کم دستور را داشته باشیم، اصلا بیت دومی برای این سیگنال نخواهیم داشت. درصورتی که دستورات jal و I-Type اضافه شوند، سیگنال ResultSrc تبدیل به سیگنال ۲ بیتی میشود؛ که در اینجا نیز تنها زمانی که دستور jal انجام میشود، بیت دوم این سیگنال یعنی سیگنال ResultSrc برابر ۱ میشود. پس درواقع نتیجه میگیریم که درهر صورت بیت دوم این سیگنال اگر برابر ۱ باشد، برای تمامی دستورات، بجز دستور sw و beq که این سیگنال مهم نیست و بصورت xx است، مشکل ایجاد میشود.

در این سوال از جدول زیر و شکل مربوطه کمک گرفته شده است.

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