CS M151B - Homework 4

Kuruvilla Cherian, Jahan
UID: 104436427
Section 1B

February 12, 2017

Note that for the most of this homework we shall refer to the diagram below for analysis of single cycle datapaths.

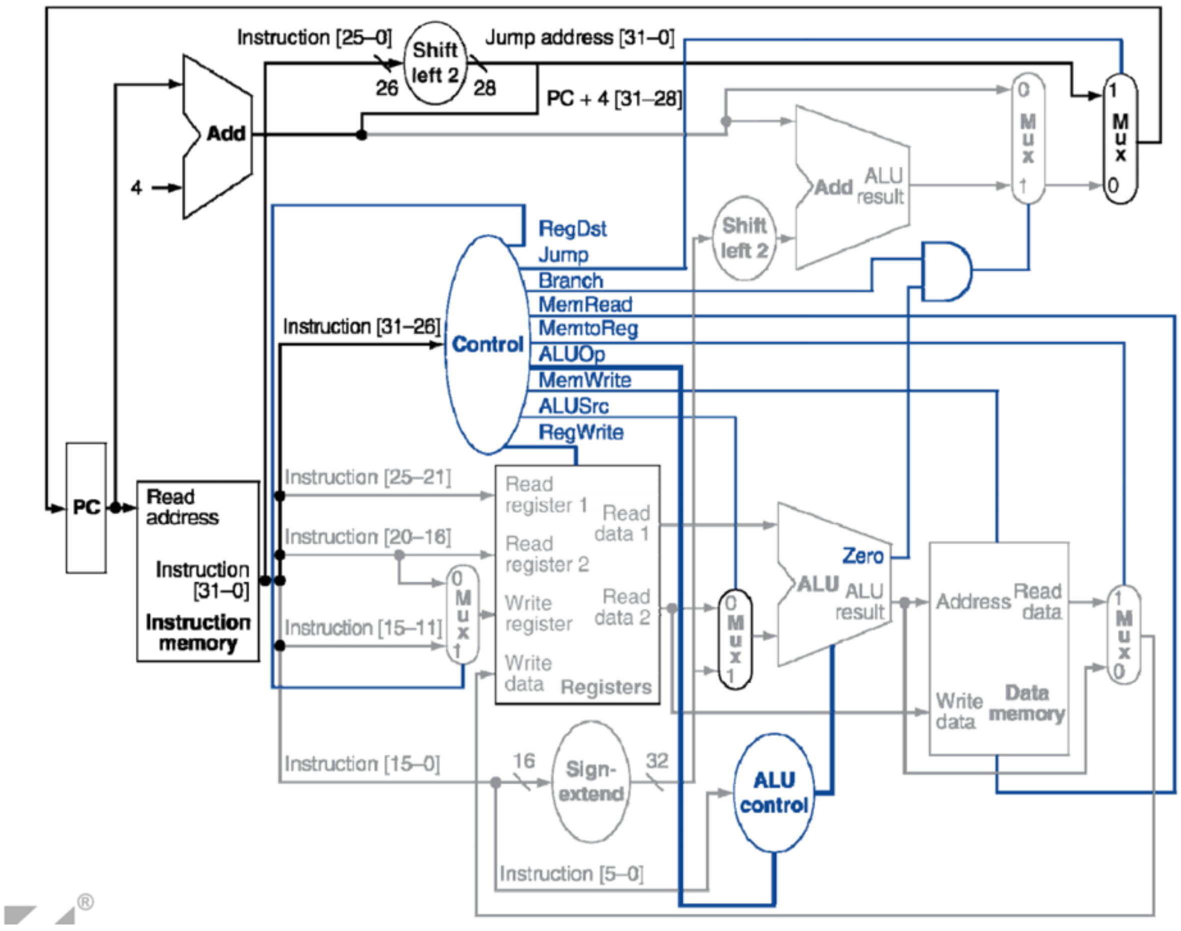

Figure 1: Single Cycle Data Path simple diagram

Problem 4.7

1

The lower 16 bits are extracted from the instruction word for sign extension. Because the MSB of the 16th bit is 0, we sign extend with all zeros leading us with:

00000000000000000000000000010100

For the jump shift, we extract the lower 26 bits and then left shift it giving us:

0001100010000000000001010000

2

The first thing we do is determine the type of instruction at hand. By looking at the most significant 6 bits we see that the bits *101011* refer to the **sw** I-type instruction. Thus on this basis we conclude that our ALU Control will receive two inputs:

ALUOp [1-0] = **00**
Instruction [5-0] = **010100**

3

Because we deduced that this instruction is just a store word I-type instruction, we know that the program counter (PC) will simply move to the next instruction as there is no conditional jumping involved. Thus the new PC address is **PC + 4**, with the path being

$$PC_{out} \rightarrow Add\,(PC+4) \rightarrow Branch\,MUX(0) \rightarrow Jump\,MUX(0) \rightarrow PC_{in}$$

4

There are a total of 5 Multiplexers throughout the processor design for the single cycle data path. These can be defined as:

$MUX_{WriteReg}$: The multiplexer determining the write register input in the register file
MUX_{ALUIn}: The multiplexer determining the input to the the ALU
$MUX_{WriteData}$: The multiplexer determining the input to the the Write Data port in the register file
MUX_{Branch}: The multiplexer that decides whether we have a branch instruction or not (takes the Branch control signal)
MUX_{Jump}: The multiplexer that decides if we have a J-type instruction (using the jump result as one input)

Based on Figure 1 and the various inputs to these MUX's we have the following values:

$MUX_{WriteReg}$ = Instruction [20-16] = **00010** = **2** OR **0** since RegDst is a don't care
MUX_{ALUIn} = Since ALUSrc is 1, we have the sign extended immediate =
00000000000000000000000000010100 = **20**
$MUX_{WriteData}$ = The output here is one we don't care about because the MemtoReg is a don't care. Therefore our output will either be a value **from data memory at address 17** or because the ALUOp code is 00, we add inside the ALU the Sign extended immediate (20) and the Read data 1 (r3 = -3) and get **17**.
MUX_{Branch}: Because we are not branching our output will just be **PC + 4**.
MUX_{Jump}: Because we are not jumping our output will just be **PC + 4**.

5

ALU: $Input_1$ = r3 = **-3** and $Input_2$ = SE(I) = **20**
Add_1: $Input_1$ = **PC** and $Input_2$ = **4**
Add_2: $Input_1$ = **PC + 4** and $Input_2$ = **20 « 2 = 80**

6

Read Register 1 = Instruction [25-21] = 00011 = **3**
Read Register 2 = Instruction [20-16] = 00010 = **2**
Write Register = Instruction [20-16] or Instruction [15-11] = 00010 or 00000 = **2** or **0**
Write Data = Some value from data memory at Address 17 or **17**
RegWrite = **0** as we don't want to write to register.

BLT Instruction

```
if (R[rs] < R[rt])
        PC = PC + 4 + SE(I)
else
        PC = PC + 4
```

The key difference between this and BEQ is that the comparison is slightly different. With the BLT we care about less than so that's the key focus. There a couple of approaches, we could either create a new ALUOp code that is 11 hardwired to the SLT and then use that, or we can just use the existing subtraction function and then use the most significant bit to control the rest.
Let's go with the latter approach. Thus we extract the most significant bit from the ALU result (if it's 1 that means we got a negative number of R[rs] – R[rt], which means R[rs] < R[rt], else vice versa). We then AND this with another BLT control signal, and then OR this result with the current output of the AND Gate from the Branch and the Zero bit, and use this result from the OR gate to control the MUX to change the PC. Thus we will have the following diagram:

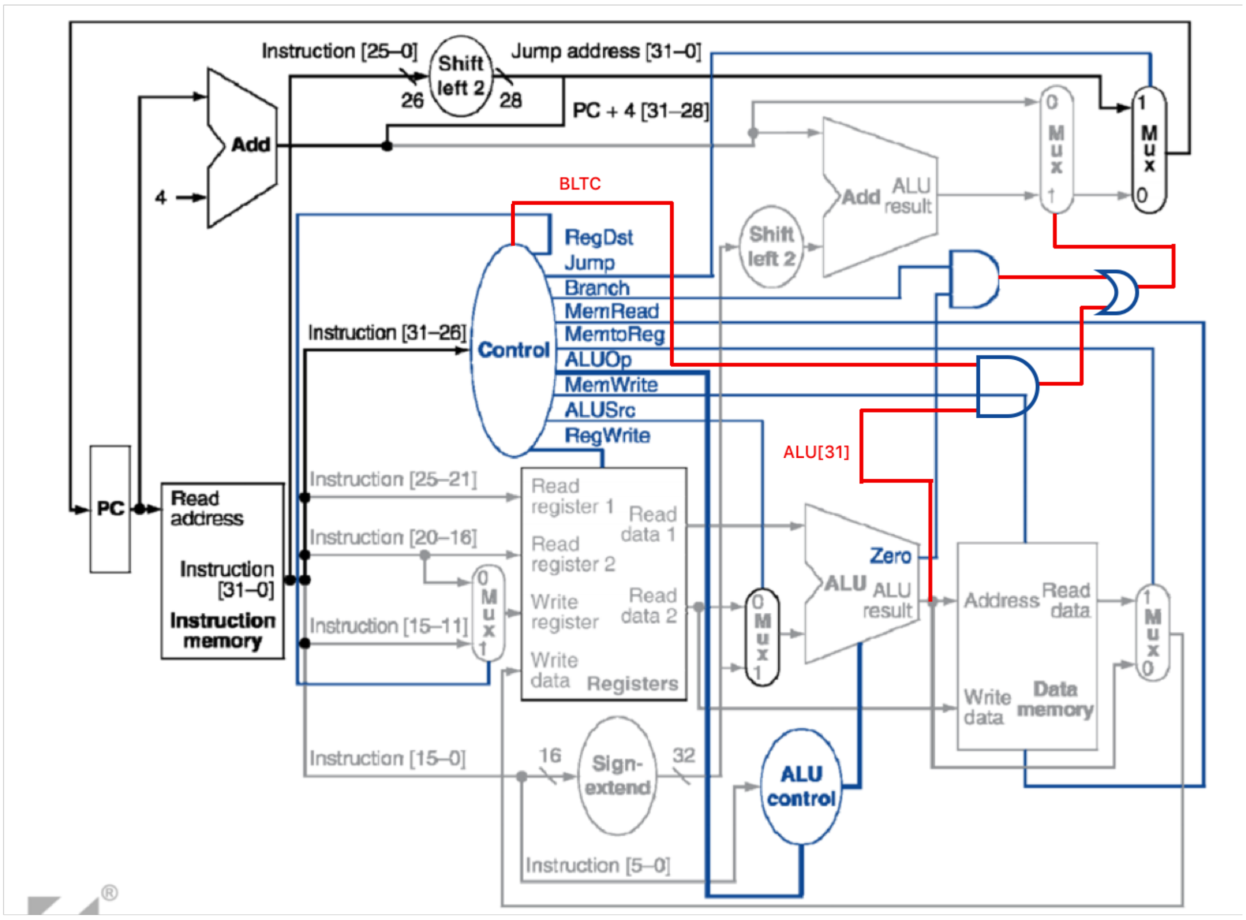

Figure 2: Modified data path to incorporate BLT

Note that the BLTC signal will be 0 for all other instructions supported, and that the Opcode for this instruction will be some I-type opcode.

Outputs	BLT
RegDst	X
ALUSrc	0
MemToReg	X
RegWrite	0
MemRead	0
MemWrite	0
Branch	0
ALUOp1	0
ALUOp2	1
Jump	0
BLTC	1

Table 1: The Controller outputs for the BLT Instruction support

JAL Instruction

```
R[$r31] = PC + 4
PC = [31...28](PC + 4) | [27...0](I << 2)
```

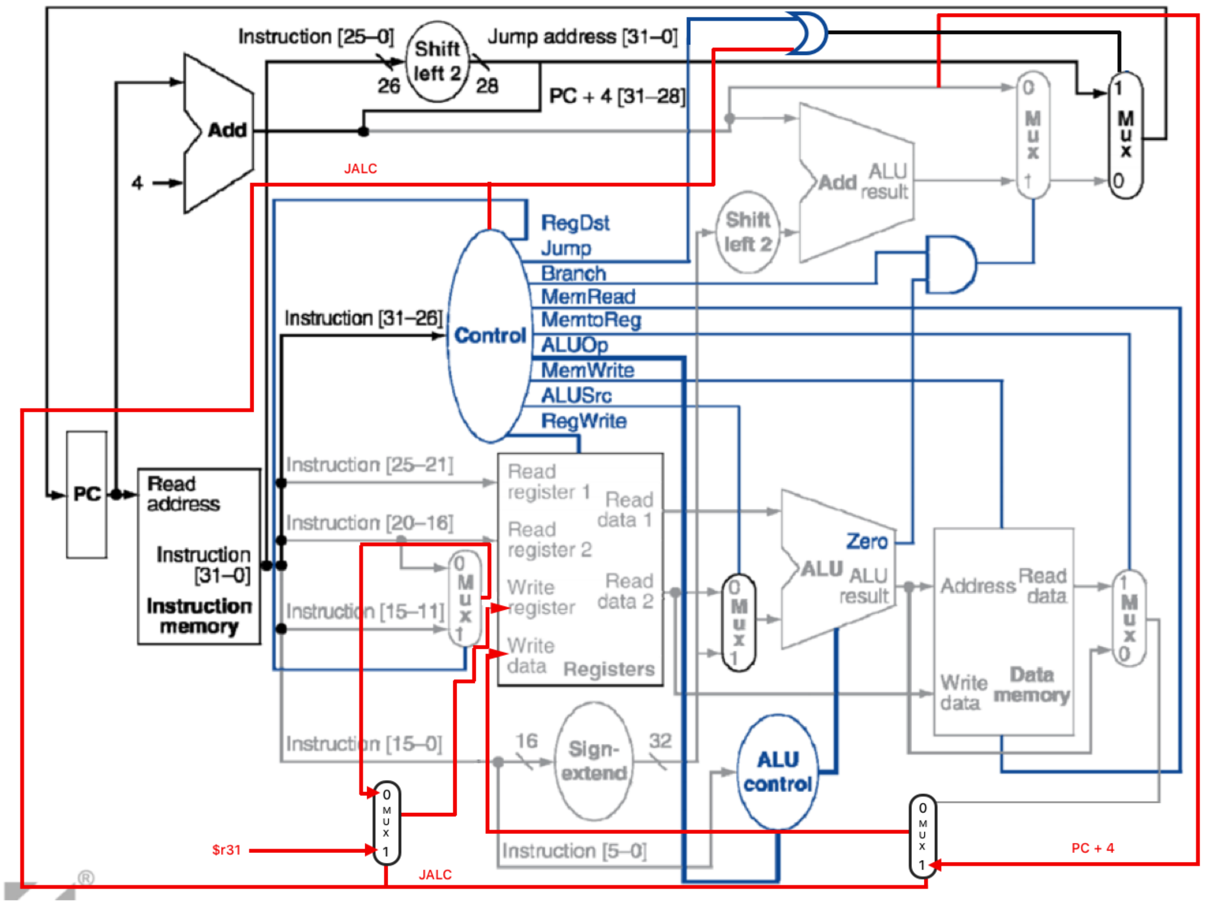

Figure 3: Modified Data Path to include JAL

The proposed JAL instruction is already extremely similar to our Jump instruction from Figure 1. The key difference has to do with the Write Register and Write Data ports for the Register file. The Write register port needs to now select between \$r31 or the output from the $MUX_{WriteReg}$ based on a JALC control signal. The Write Data port will have to select from PC + 4 or from the previous output of the $MUX_{WriteData}$, also based on JALC control signal. Finally because we could jump using either the Jump signal or this new JALC control we OR the results to use as the control to the MUX_{Jump}. This can be seen in Figure 3 above. The modified control inputs and outputs are shown in Table 2 below. Note that the JALC signal will be 0 for all other instructions

Outputs	**BLT**
RegDst	X
ALUSrc	X
MemToReg	X
RegWrite	1
MemRead	0
MemWrite	0
Branch	0
ALUOp1	X
ALUOp2	X
Jump	0
JALC	1

Table 2: The Controller outputs for the JAL Instruction support

supported, and that the Opcode for this instruction will be some J-type opcode.

JR Instruction

PC = R[rs]

The key thing to note here is that this instruction is an R-type instruction. So assuming we keep the opcode 000000. Therefore because of this restriction we cannot modify the main Controller output as they are the same for all R-type instructions, and so we will have to modify the ALU Controller. In order to do this we add the following new operation **jrc** as listed below:

opcode	**ALUOp**	**Operation**	**funct**	**ALU Function**	**ALU Control**
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add 0010	
beq	01	branch equal	XXXXXX	subtract	0110
R-type	01	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111
		jrc	011111	OR	0001

Table 3: Modified ALU Controller with jrc

Notice we set the function field to 011111 and let the ALU function simply be an OR since we don't care about the data that goes into the write register. Thus with these fields we can control the MUX_{Jump} using the AND of the NOT of the MSB of the funct field and the LSB of the ALU Control. We will use the data path without the Jump, so that we can use R[rs] as one of the inputs to the MUX_{Jump}. Thus we can ensure with this logic that PC will only be set to R[rs] when the funct field is set for jrc. The only last concern would be with RegWrite, since in this case because of the nature of the main control signals, we would be writing "junk" data into the write register. To counteract this we just AND RegWrite with NOT of our MUX_{Jump} control signal. This is shown clearly in Figure 4.

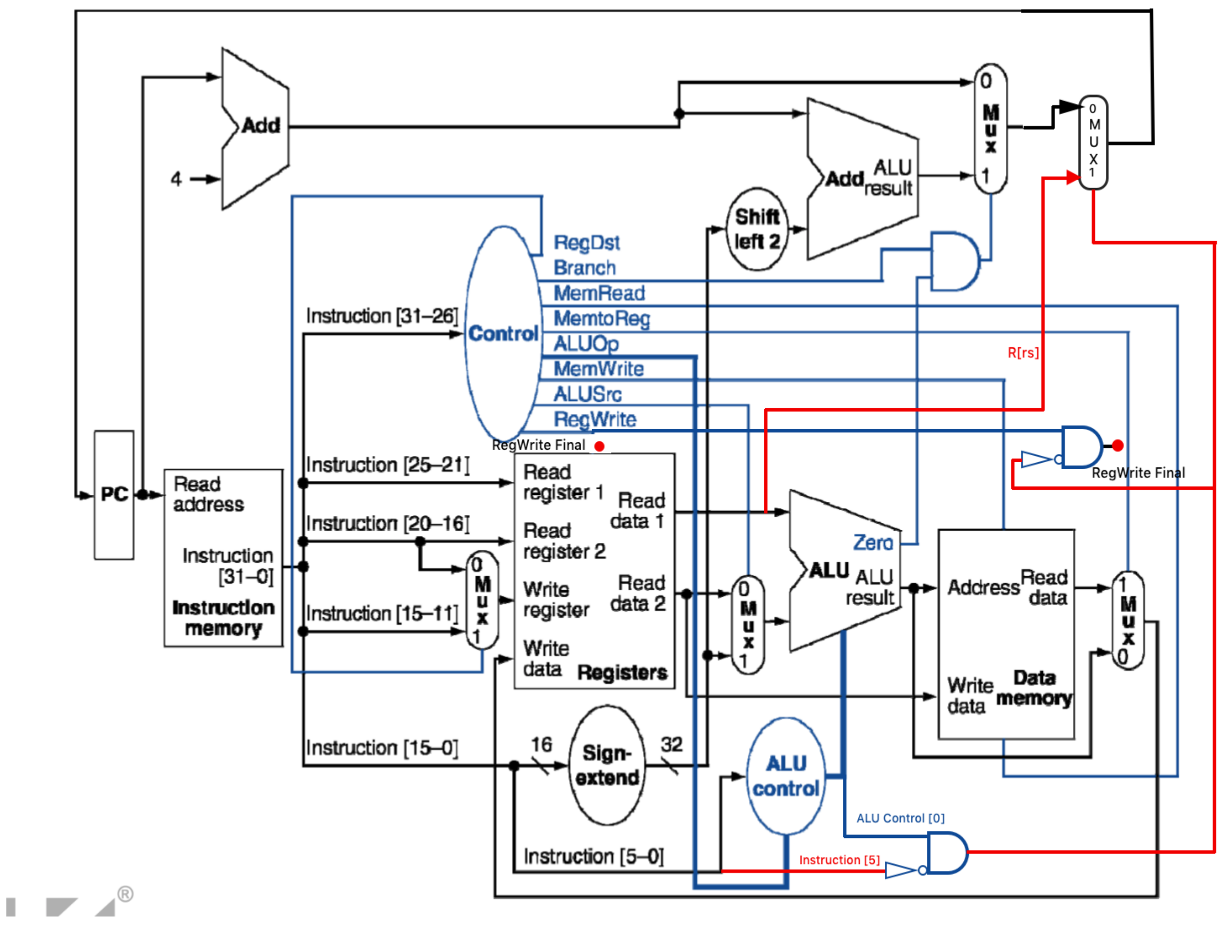

Figure 4: Modified Data Path to include JR