CS151B Winter 2022

Discussion(week 4)

CS151B Teaching Team

Agenda

1. Logistics
2. Datapath and control review
3. Practice (datapath control)
4. Q&A (if time allows)

Logistics

HW 4 due **Friday (Jan 28, 2022)** 11:59PM

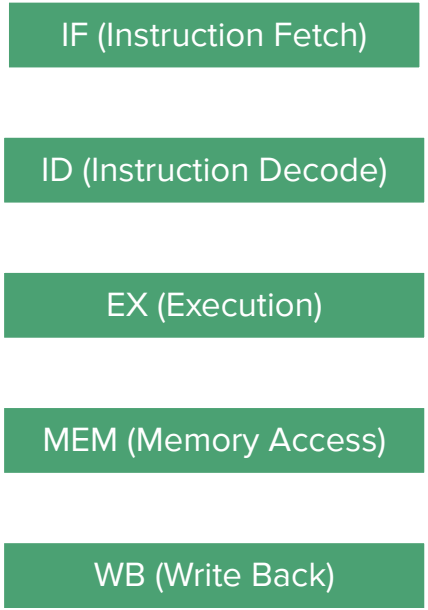
Midterm is next **Wednesday (Feb 2nd, 2022), 10 – 11:50 AM**

- Online exam is proctored via Zoom.
- You are required to keep your video on during the entire duration of the exam.
- Exam is open book, open notes, but **no collaboration** is allowed
- *If you have technical difficulties, please let us know as soon as possible*

PLEASE READ THE QUESTIONS CAREFULLY !! NO TLDR!!!

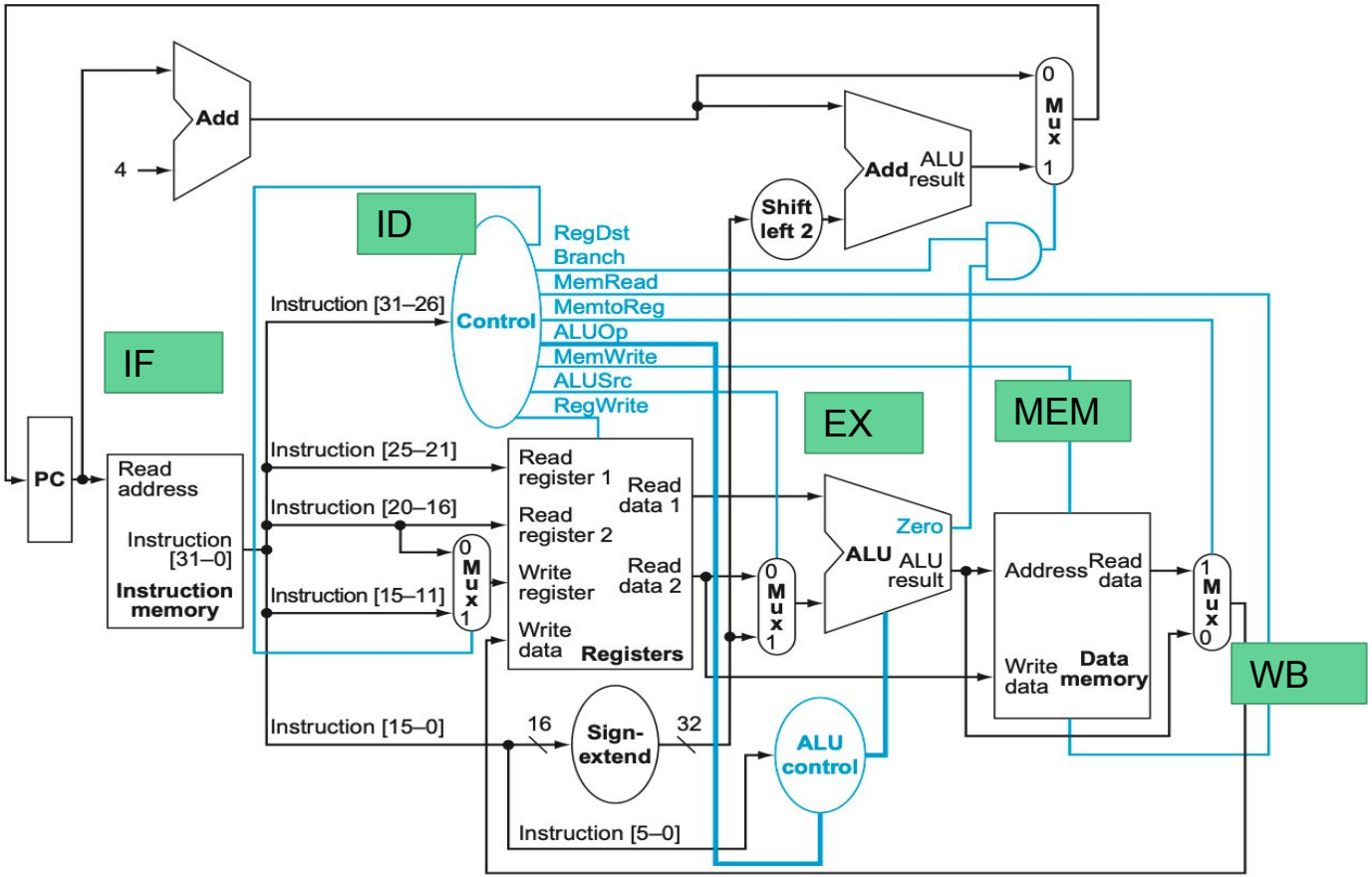
Cheating is not allowed, we take academic honesty very seriously

Our simple computation flow

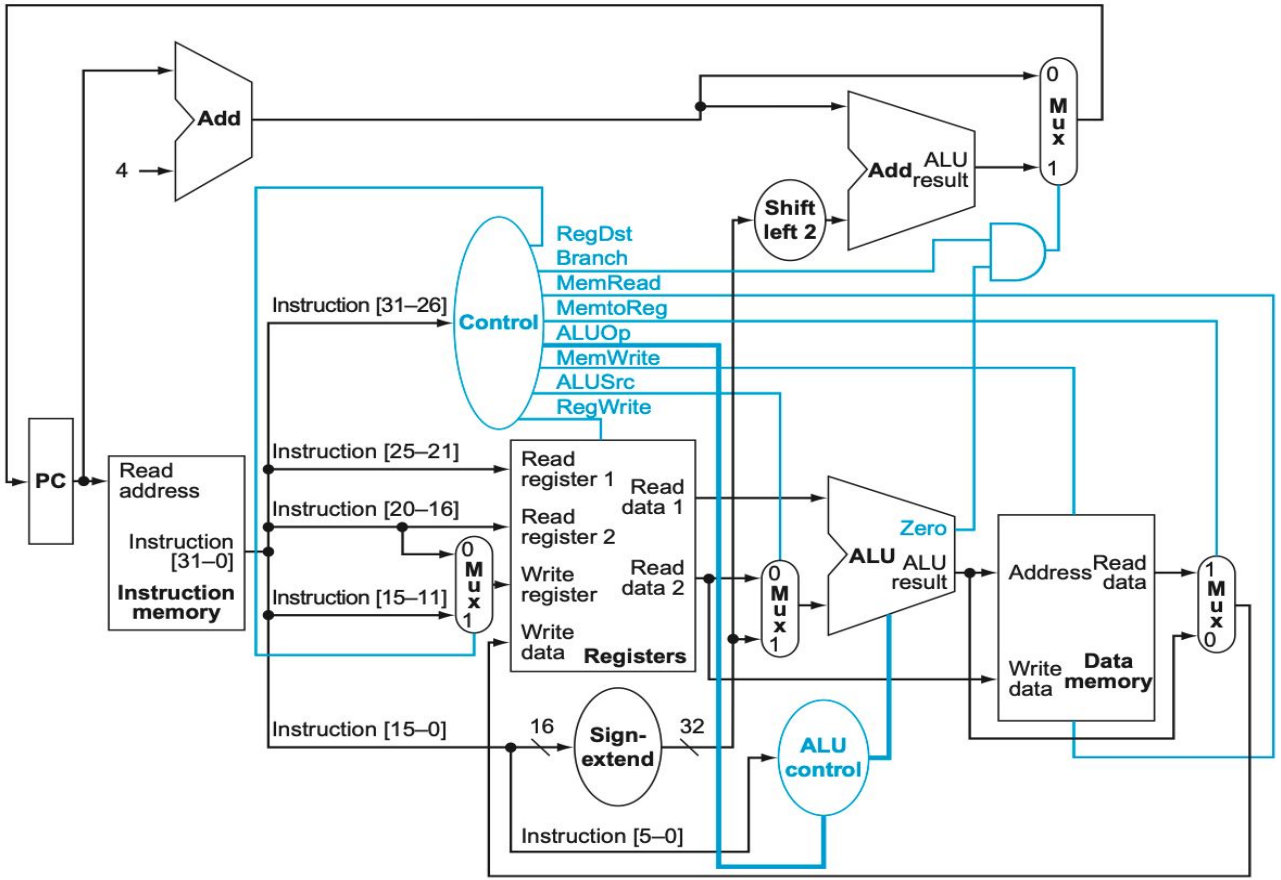


This will be the basis for our pipeline design in the coming weeks

Simple datapath with control unit



Simple datapath with control unit



MIPS instruction formats-revisited

Type	31 26	25 21	20 16	15 11	10 06	05 00
R-Type	opcode	$rs	$rt	$rd	shamt	funct
I-Type	opcode	$rs	$rt	imm		
J-Type	opcode	address				

R-type needs both `$rs` and `$rt` values as the inputs to the ALU computation. It needs to write the result to the `$rd` register (for the typical R-types).

I-type needs the `$rs` value and the `SE(imm)` value as its ALU operands. It will write its final result to the `$rt` register(in case of a **lw**).

J-type does not need any register values, it only needs to create its target address by shifting its address field by 2 bits and concatenating with the MS 4 bits of the opcode.

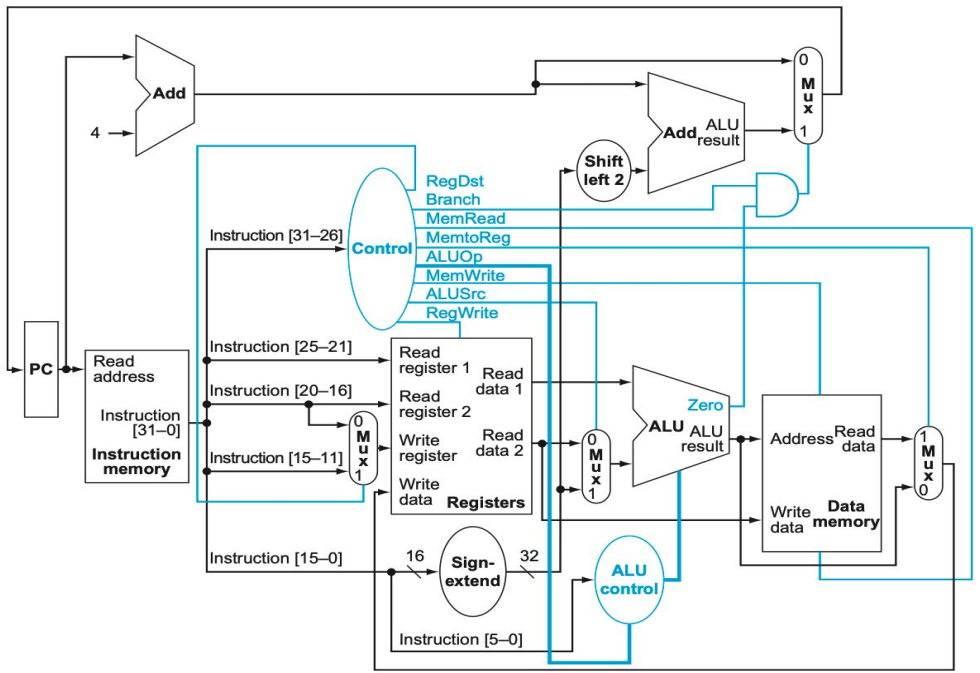
But it doesn't mean we don't read the values not needed (in register files), we instead use control logic to extract only the meaningful data for our specific instruction.

Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc (Branch)	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

R-type instruction (*add, subtract, and, or , set on less than*):

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0



Output	R-format
RegDst	1
ALUSrc	0
MemtoReg	0
RegWrite	1
MemRead	0
MemWrite	0
Branch	0
ALUOp1	1
ALUOp0	0

Q: Why can't we set the MemRead and MemWrite to don't-care?

1. Writing garbage values to the memory might pollute our memory.
2. Reading from random addresses (potentially not valid) addresses could cause exceptions

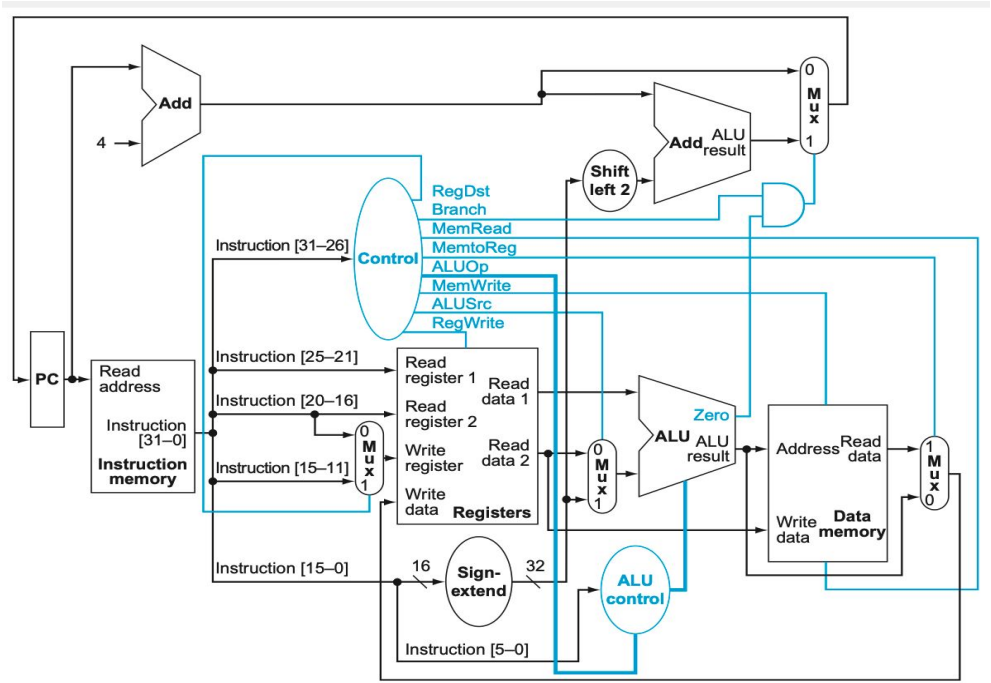
Input to the main controller:
000000 (0 in decimal)

Q: How does the ALU control distinguish different R-type instruction?

a. 6-bit Funct field is used.

I-type instruction (LW)

35 or 43	rs	rt	address
31:26	25:21	20:16	15:0

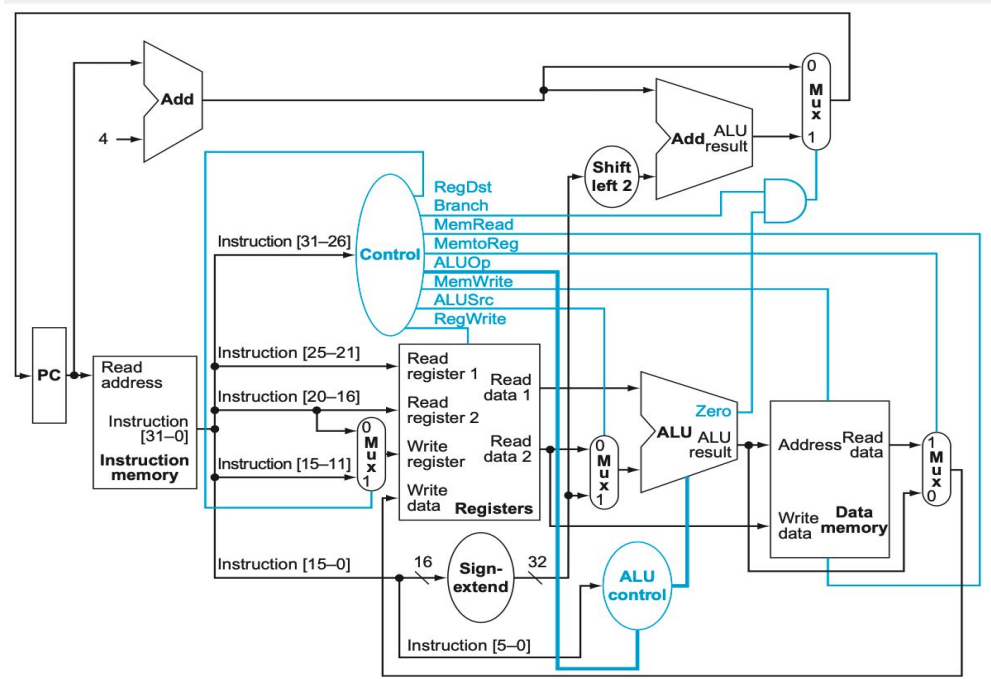


Output	LW
RegDst	0
ALUSrc	1
MemtoReg	1
RegWrite	1
MemRead	1
MemWrite	0
Branch	0
ALUOp1	0
ALUOp0	0

Input to the main controller: 100011 (35 in decimal)

I-type instruction (SW)

35 or 43	rs	rt	address
31:26	25:21	20:16	15:0



Output	SW
RegDst	X
ALUSrc	1
MemtoReg	X
RegWrite	0
MemRead	0
MemWrite	1
Branch	0
ALUOp1	0
ALUOp0	0

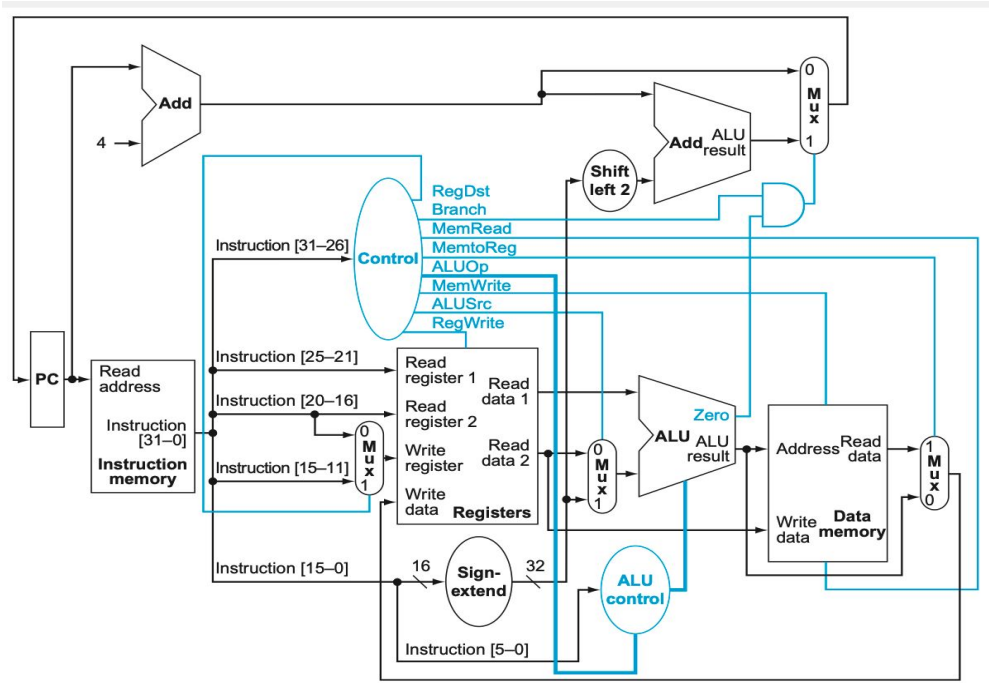
Q: Why are we using the same ALUOp code for both SW and LW?

A. Because we want to perform an add in both cases.

Input to the main controller: 101011 (43 in decimal)

Branch instructions (BEQ)

4	rs	rt	address
31:26	25:21	20:16	15:0



Output	BEQ
RegDst	X
ALUSrc	0
MemtoReg	X
RegWrite	0
MemRead	0
MemWrite	0
Branch	1
ALUOp1	0
ALUOp0	1

Q: Why are the RegDst and MemtoReg are X's?

a. Because we set the RegWrite to 0, so we don't write to the register file. This means their values have no effects on the register file.

Input to the main controller: 000100 (4 in decimal)

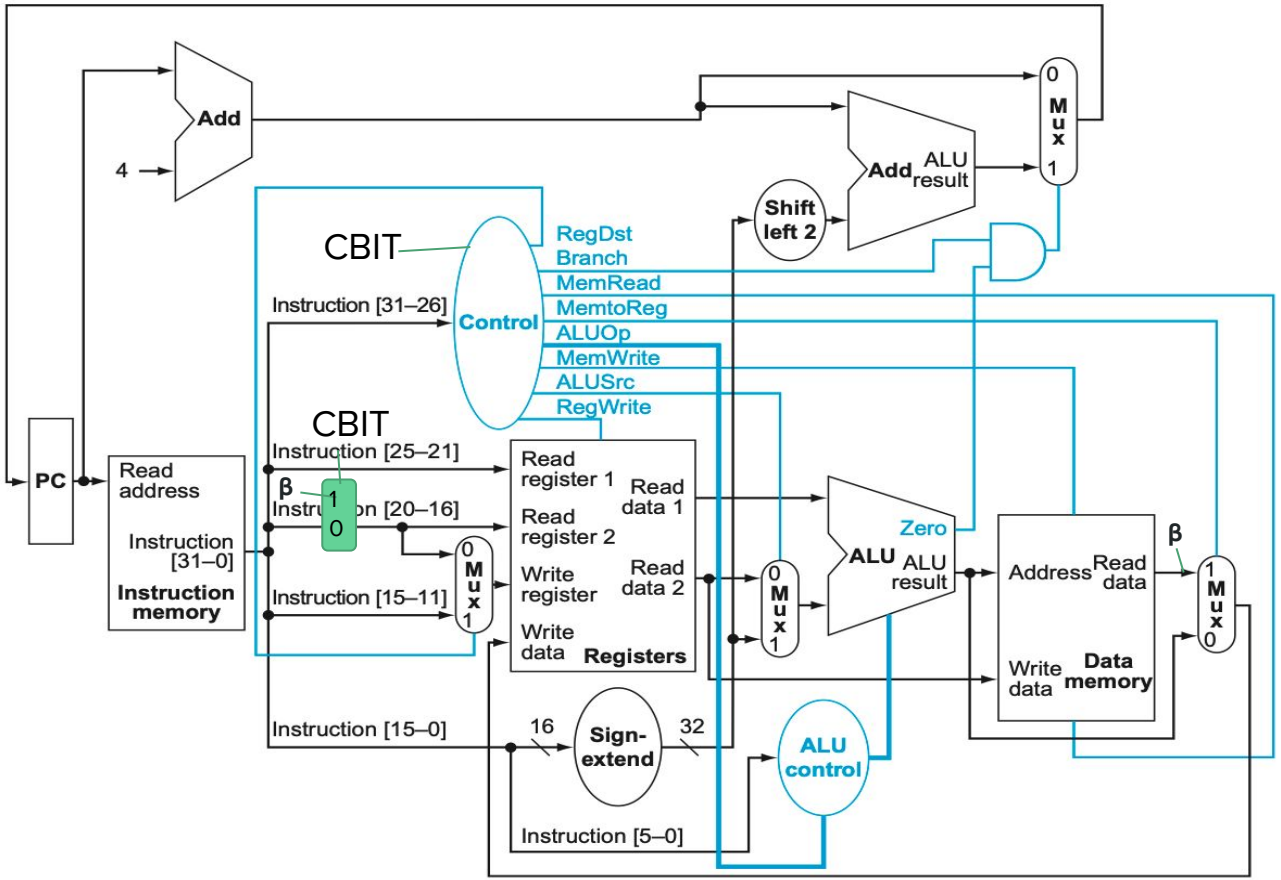
How to solve datapath/control modification problems?

Steps:

1. Identify which signals go where (e.g. what goes into ALU?)
2. Add **muxes+wires** if necessary, **use variables**
3. Decide what the control signals should be
   a. You can make your own! If I/J type then the **main control unit** produces it
   b. **If R type** then our opcode =0 so we need to produce it from functional field (**ALU controller**)
4. Add your control signal **to the table**
5. *Carefully follow the problem instructions on what's allowed*: e.g., whether you can use additional functional units such as an ALU, register files, or additional ports to the memory
6. Make sure the modified circuit will **not break the original** functionalities of the circuit.

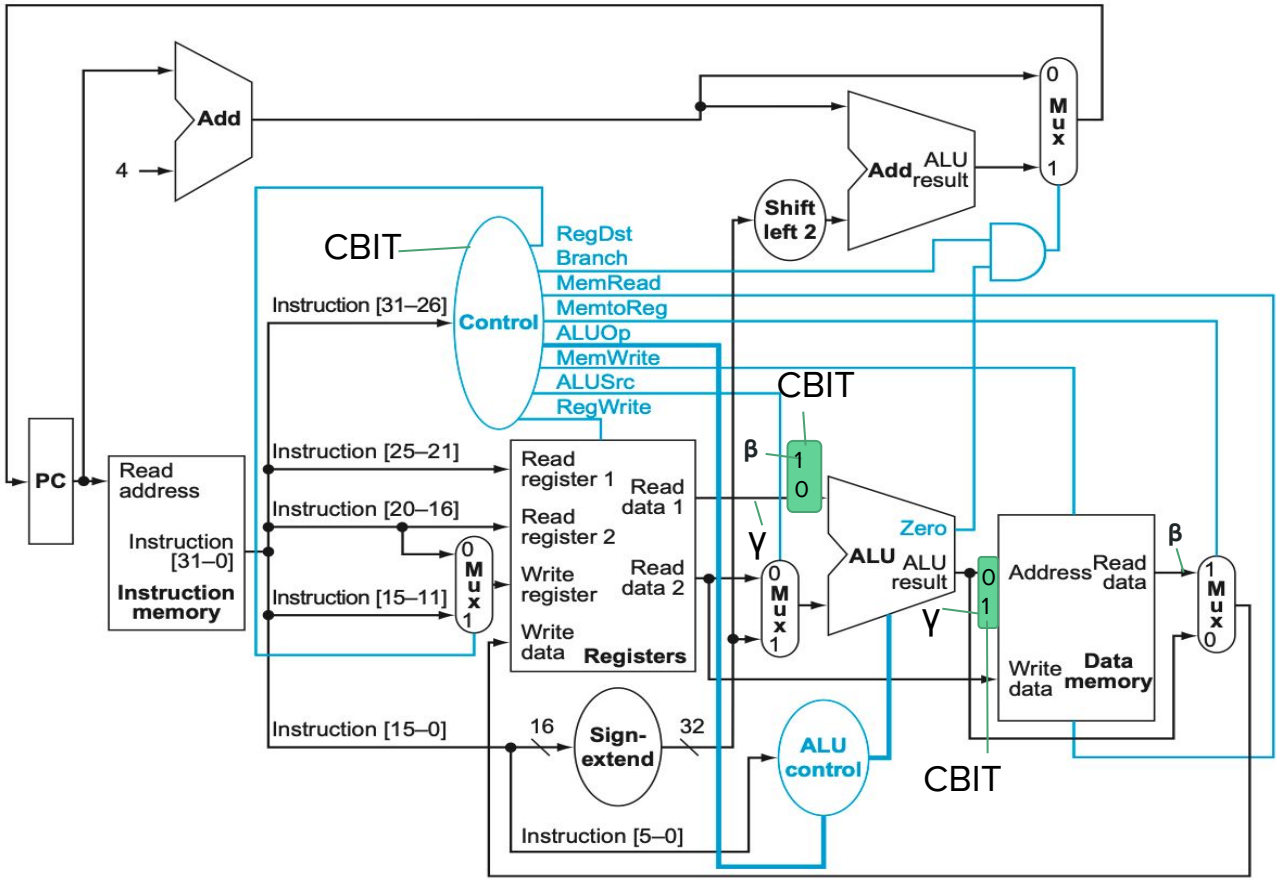
A problem can be solved in different ways!

Simple datapath with control unit



R type?
I Type?

Simple datapath with control unit



Is this allowed?

Practice Questions

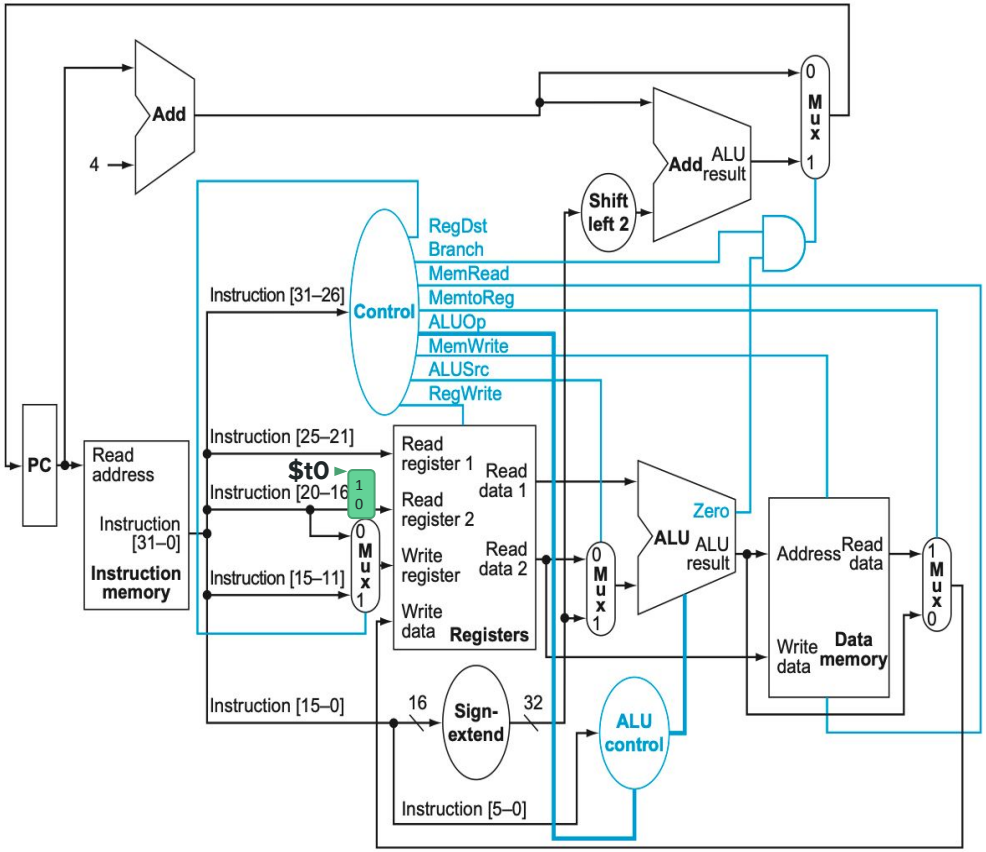
P1 - MLT instruction

Consider the single-cycle processor implementation from class. Your task will augment this datapath with a new instruction - MLT:

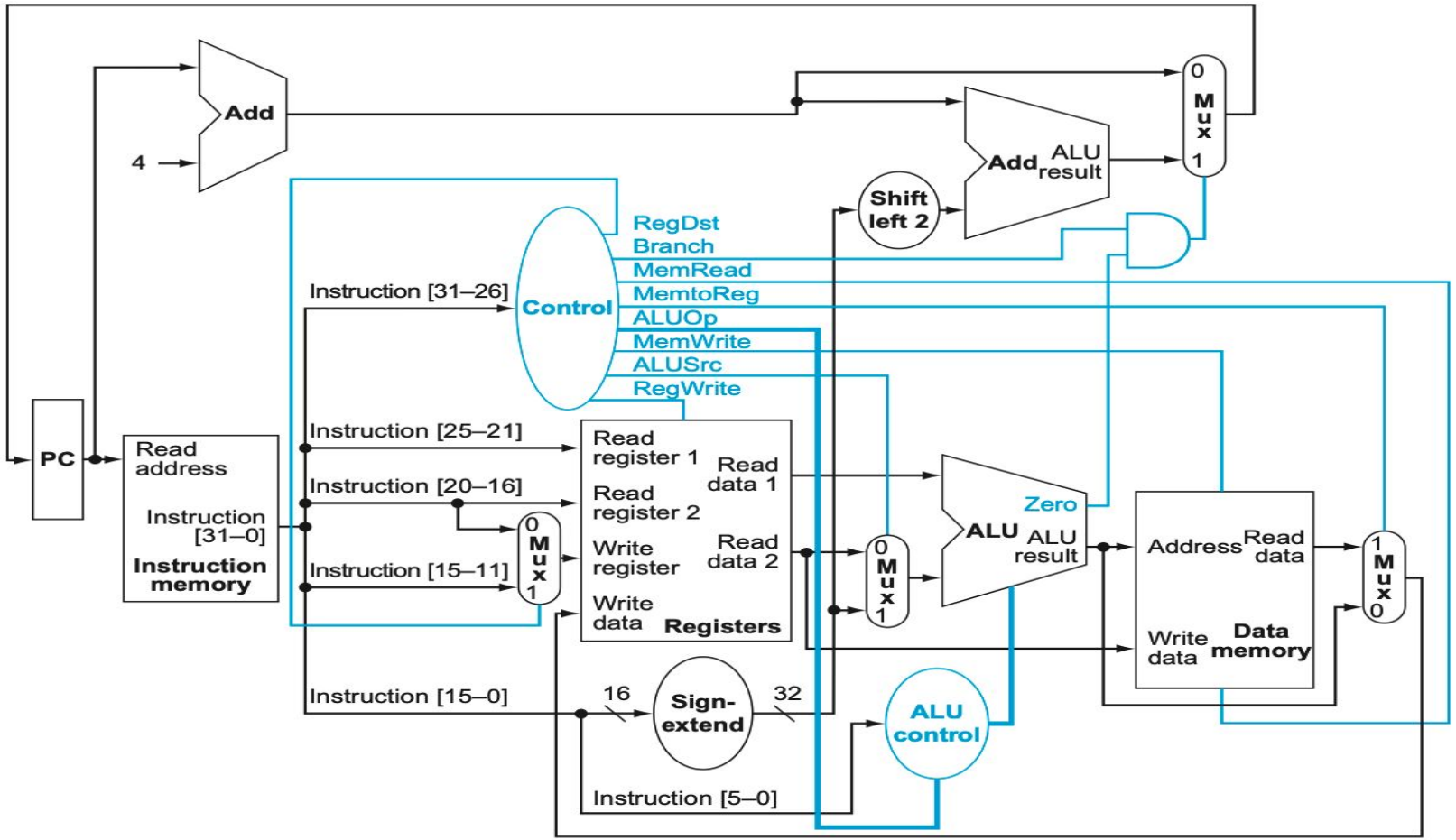
1. It is an I-type
2. Here is the pseudocode:

```
If (M[R[rs]] < R[$t0]):

        R[rt] = SE(I)
```

1. **$t0 is implicitly used, it is not encoded in I-type field**



```
If (M[R[rs]] < R[$t0]):   R[rt] = SE(I)
```



Main Controller

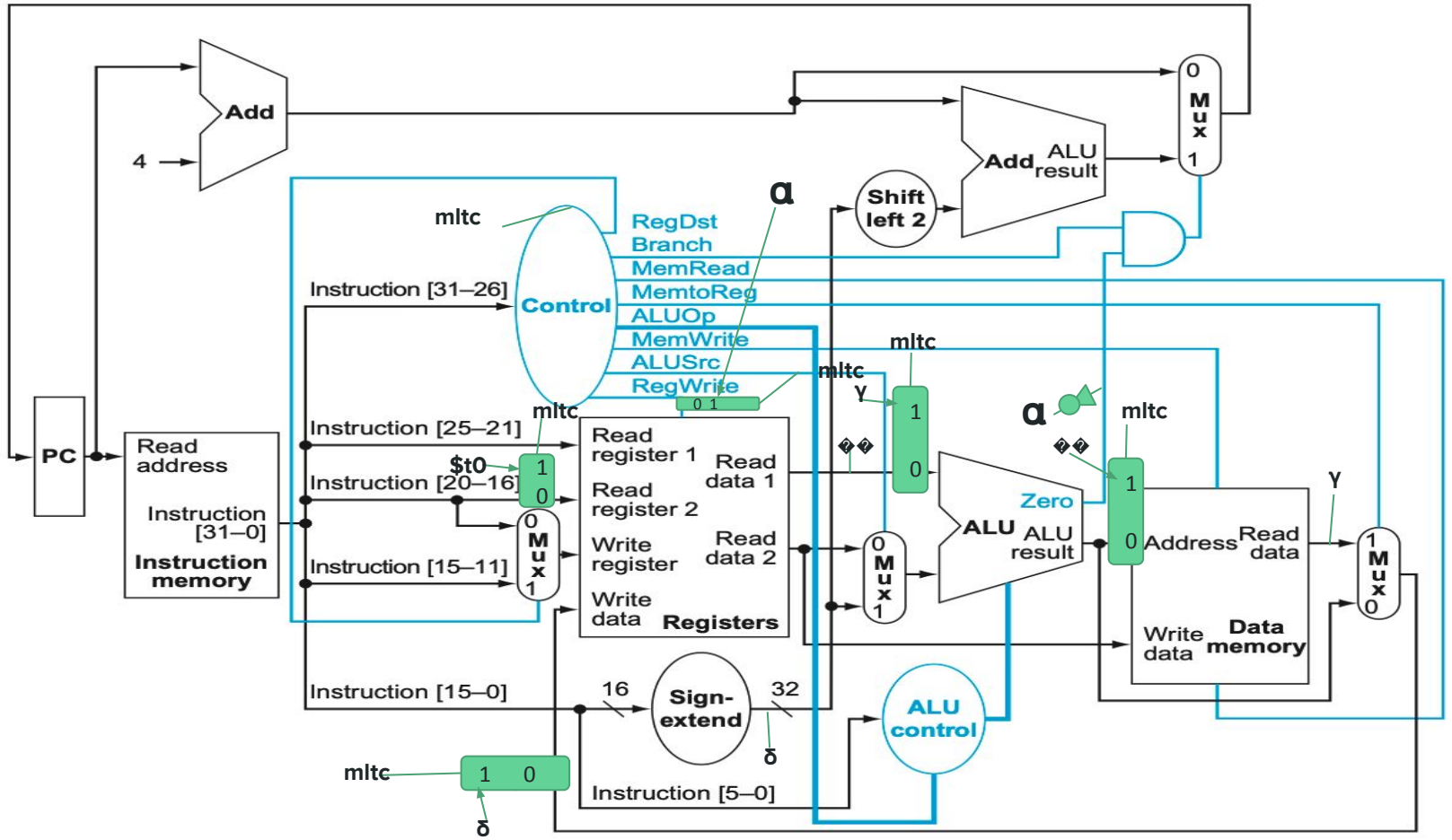
Input or Output	Signal Name	R-format	lw	sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

```
If (M[R[rs]] < R[$t0]):   R[rt] = SE(I)
```

ALU Controller

Opcode	ALUOp	instruction	function	ALU Action	ALUCtrl
Lw	00	load word	XXXXXX	add	010
Sw	00	store word	XXXXXX	add	010
Beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111

If (M[R[rs]] < R[$t0]): R[rt] = SE(I)



Main Controller

Input or Output	Signal Name	R-format	lw	sw	Beq	
Inputs	Op5	0	1	1	0	0
	Op4	0	0	0	0	1
	Op3	0	0	1	0	0
	Op2	0	0	0	1	0
	Op1	0	1	1	0	0
	Op0	0	1	1	0	0
Outputs	RegDst	1	0	X	X	0
	ALUSrc	0	1	1	0	0
	MemtoReg	0	1	X	X	x
	RegWrite	1	1	0	0	x
	MemRead	0	1	0	0	1
	MemWrite	0	0	1	0	0
	Branch	0	0	0	1	0
	ALUOp1	1	0	0	0	1
	ALUOp0	0	0	0	1	1

Mltc 0 0 0 0 1

ALU Controller

Opcode	ALUOp	instruction	function	ALU Action	ALUCtrl
Lw	00	load word	XXXXXX	add	010
Sw	00	store word	XXXXXX	add	010
Beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111

Mltc 11 mltc xxxx slt 111

P2

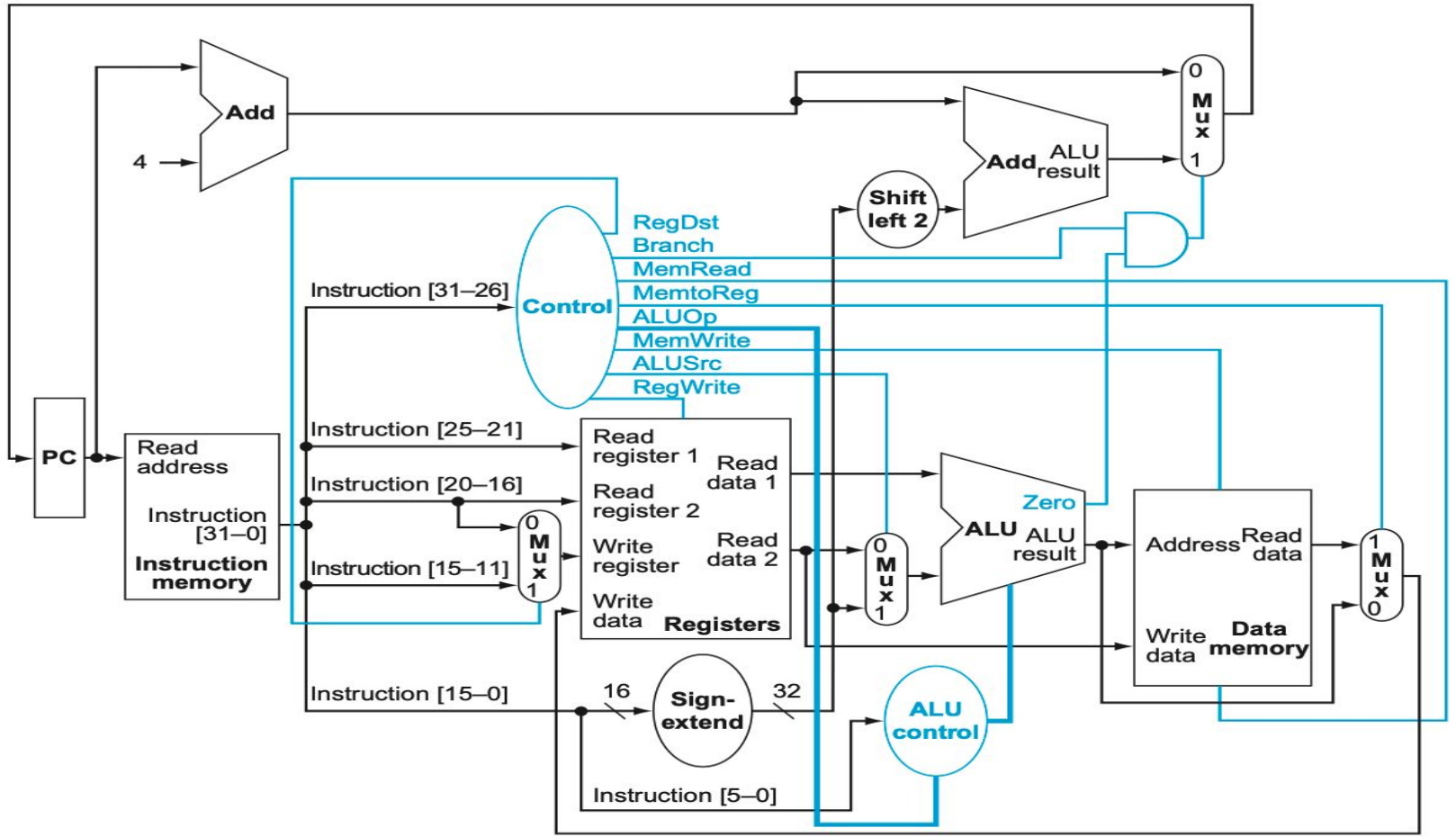
Consider the single-cycle processor implementation from class. Your task will be to augment this datapath with a new instruction - the branch on immediate *(**BOI**)* instruction. The ***BOI*** instruction is an **I-type** instruction. The ***BOI*** instruction will have the following pseudocode:

```
If ( RF[RT] < SE(I) ):

      PC = PC + 4 + M[ RF[RS] ]

Else:

      PC = PC + 4 + M[ RF[RT] ]
```

Where RF[] is the register file and M[] is memory.

All other instructions must still work correctly after your modifications. You should not add any new ALUs, register file ports, or ports to memory.

```
If ( RF[RT] < SE(I) ):        PC = PC + 4 + M[ RF[RS] ]

                Else:         PC = PC + 4 + M[ RF[RT] ]
```



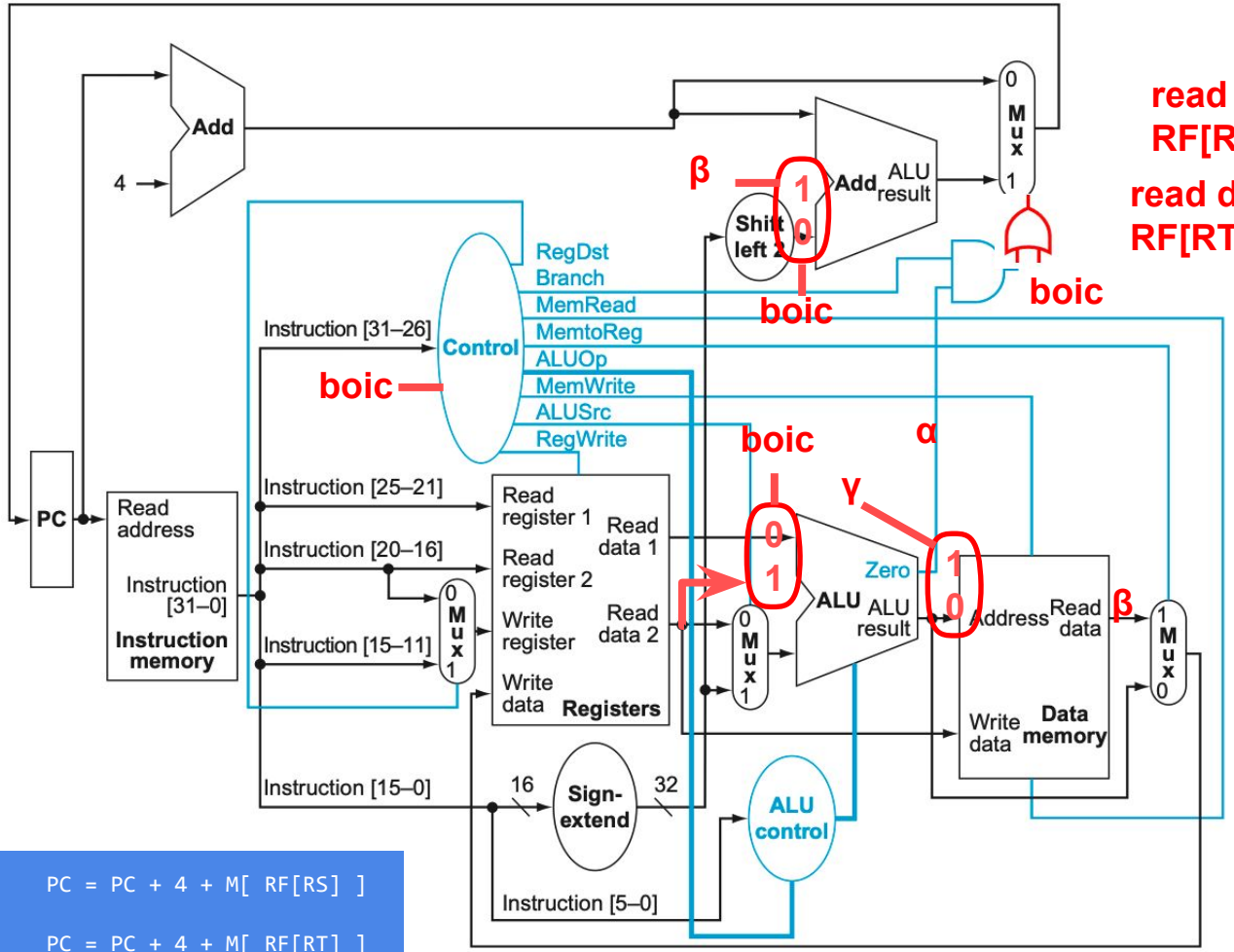
Main Controller

Input or Output	Signal Name	R-format	lw	sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

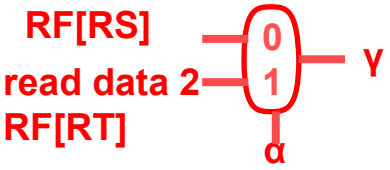
```
If ( RF[RT] < SE(I) ):        PC = PC + 4 + M[ RF[RS] ]

                Else:         PC = PC + 4 + M[ RF[RT] ]
```

ALU Controller

Opcode	**ALUOp**	**instruction**	**function**	**ALU Action**	**ALUCtrl**
Lw	00	load word	XXXXXX	add	010
Sw	00	store word	XXXXXX	add	010
Beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111



read data 1
RF[RS]
read data 2
RF[RT]




```
If ( RF[RT] < SE(I) ):      PC = PC + 4 + M[ RF[RS] ]
                Else:       PC = PC + 4 + M[ RF[RT] ]
```

Main Controller

Input or Output	Signal Name	R-format	lw	sw	Beq	BOI
Inputs	Op5	0	1	1	0	1
	Op4	0	0	0	0	1
	Op3	0	0	1	0	0
	Op2	0	0	0	1	0
	Op1	0	1	1	0	0
	Op0	0	1	1	0	0
Outputs	RegDst	1	0	X	X	X
	ALUSrc	0	1	1	0	1
	MemtoReg	0	1	X	X	X
	RegWrite	1	1	0	0	0
	MemRead	0	1	0	0	1
	MemWrite	0	0	1	0	0
	Branch	0	0	0	1	X
	ALUOp1	1	0	0	0	1
	ALUOp0	0	0	0	1	1
	BOIC	0	0	0	0	1

ALU Controller

Opcode	ALUOp	instruction	function	ALU Action	ALUCtrl
Lw	00	load word	XXXXXX	add	010
Sw	00	store word	XXXXXX	add	010
Beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111
BOI	11	BOI	XXXXXX	SLT	111

P3

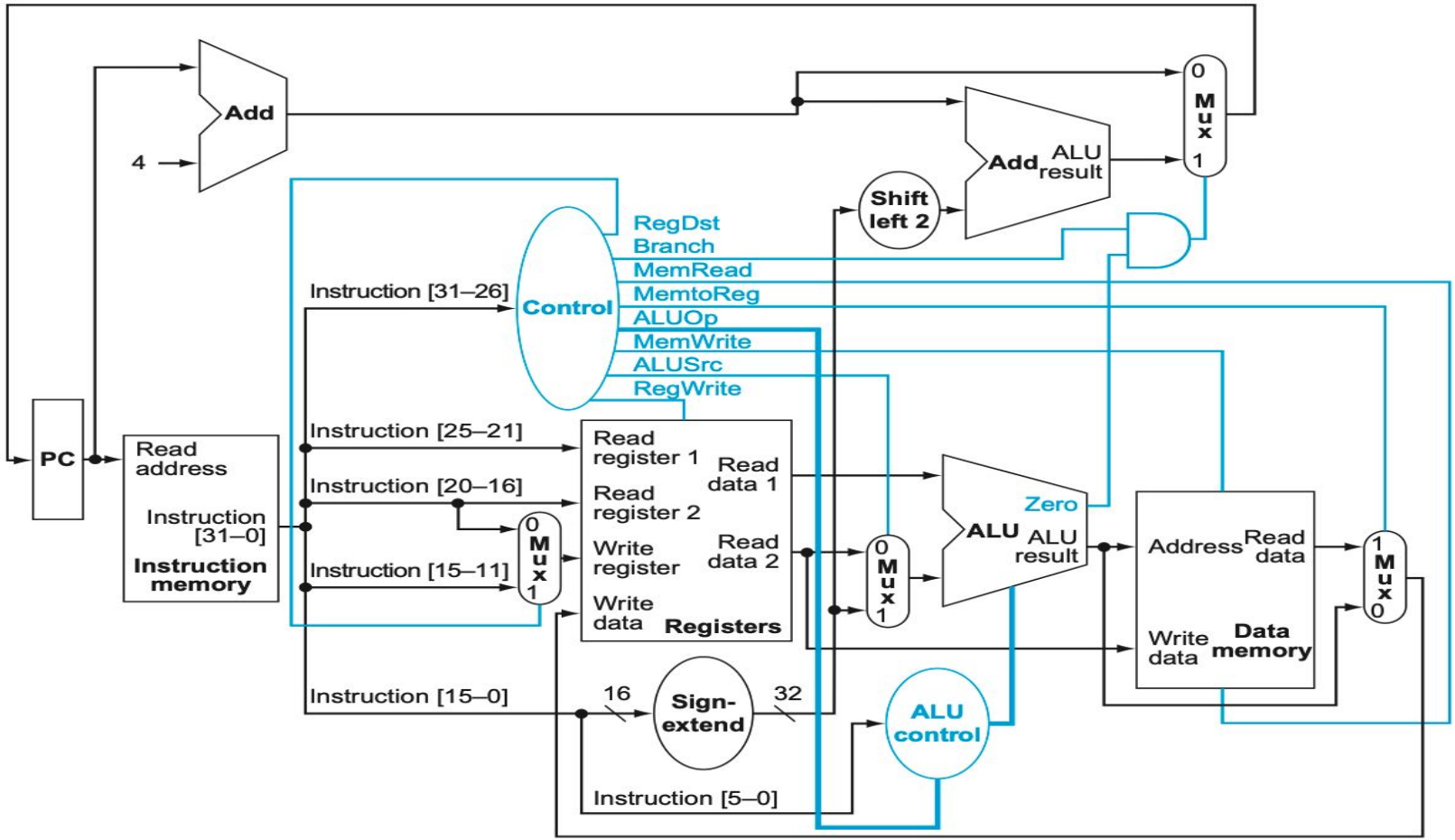
Consider the single-cycle processor implementation. Your task will be to augment this datapath with a new instruction: the ***funkyb*** instruction. This instruction will be an **R-Type** instruction, and will have the following effect:

```
If ( R[rs] < R[rt] ):
     PC = PC + 4 + R[rs]          //Note that these two statements
     R[rd] = R[rt]                   //will be concurrent
Else:
     PC = PC + 4 + R[rt]          //Note that these two statements
     R[rd] = R[rs]               //will be concurrent
```

Implement ***funkyb*** on the single cycle datapath. Use the **R-type** instruction format - so this instruction will have the same opcode as all other R-types. Use a unique function field to modify the ALU controller to implement this instruction, not the main controller.

All other instructions must still work correctly after your modifications. You should not add any new ALUs, register file ports, or ports to memory.

```
If ( R[rs] < R[rt] ):     PC = PC+4+R[rs]          R[rd] = R[rt]
                Else:     PC = PC+4+R[rt]          R[rd] = R[rs]
```



Main Controller

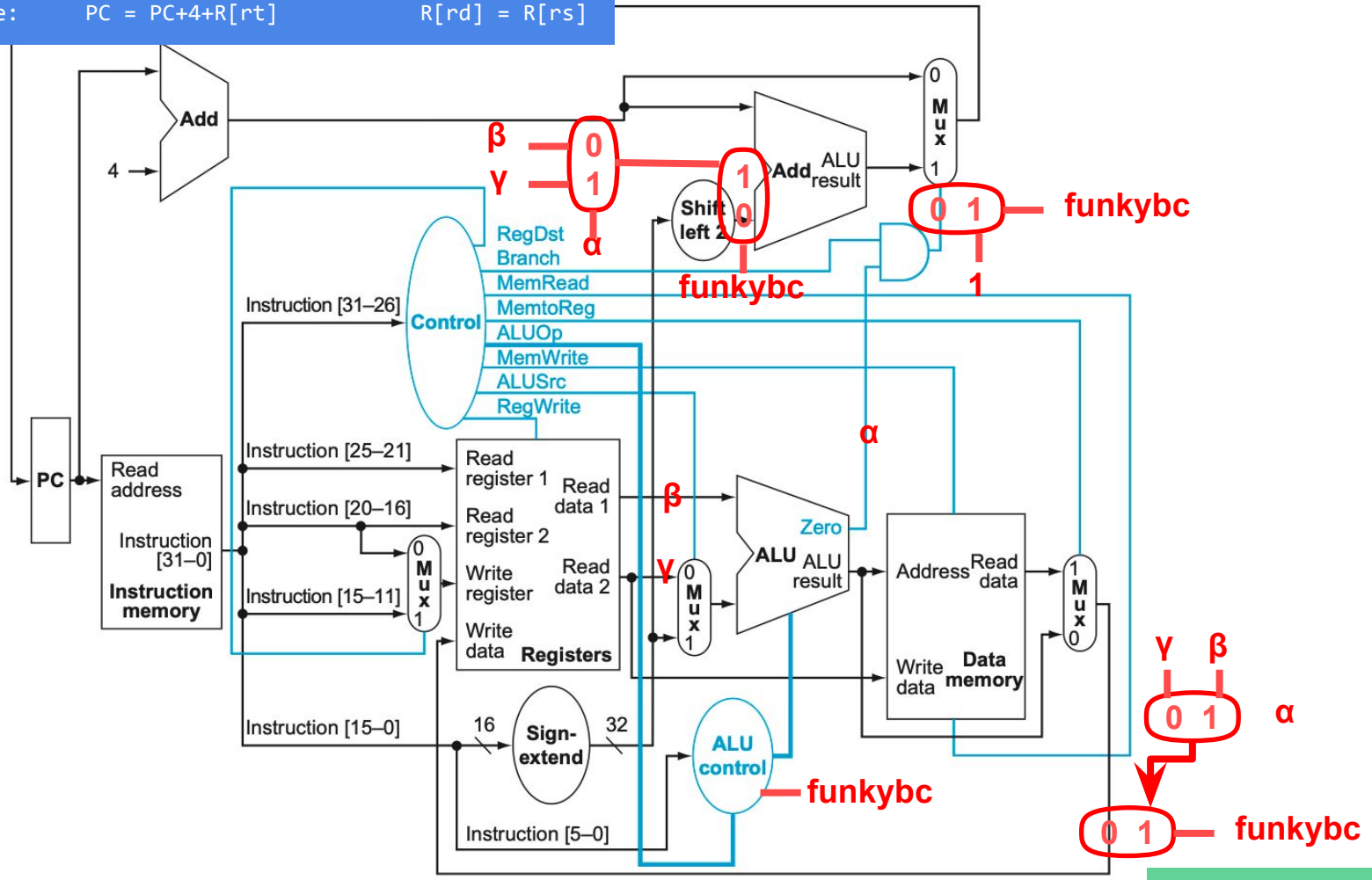
Input or Output	Signal Name	R-format	lw	sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

```
If ( R[rs] < R[rt] ):     PC = PC+4+R[rs]          R[rd] = R[rt]
                  Else:   PC = PC+4+R[rt]          R[rd] = R[rs]
```

ALU Controller

Opcode	ALUOp	instruction	function	ALU Action	ALUCtrl
Lw	00	load word	XXXXXX	add	010
Sw	00	store word	XXXXXX	add	010
Beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	AND	000
R-type	10	OR	100101	OR	001
R-type	10	SLT	101010	SLT	111

```
If ( R[rs] < R[rt] ):      PC = PC+4+R[rs]          R[rd] = R[rt]
                  Else:    PC = PC+4+R[rt]          R[rd] = R[rs]
```



Main Controller

Input or Output	Signal Name	R-format	lw	sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

ALU Controller

Opcode	ALUOp	instruction	function	ALU Action	ALUCtrl	funkybc
Lw	00	load word	XXXXXX	add	010	0
Sw	00	store word	XXXXXX	add	010	0
Beq	01	branch equal	XXXXXX	subtract	110	0
R-type	10	add	100000	add	010	0
R-type	10	subtract	100010	subtract	110	0
R-type	10	AND	100100	AND	000	0
R-type	10	OR	100101	OR	001	0
R-type	10	SLT	101010	SLT	111	0
R-type	10	funky	001011	SLT	111	1

Funkyb is an R-type instruction

Acknowledgement:

Patterson, David, and John Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 5th ed., Morgan Kaufmann, 2013.

Credit to Prof. (Glenn) Reinman (some practice questions were extracted from previous midterm exams)

Questions? Feedback?

Good luck on the exam!!!

Thanks!