Question 2: Multicycle CPU implementation

We would like to add a "scaled" addressing mode to the MIPS multicycle architecture:

lws rd, rs, rt # rd = Mem[rt + (4 × rs)]

For example, if $a0 contains 1000 and $a1 contains 10, then "lws $t0, $a1, $a0" loads $t0 with the value at address 1040 (1000 + 4×10).

You need to show the correct control signals necessary to implement the lws instruction, by *either*:
- completing the finite state machine diagram on page 4, *or*
- filling in the microprogramming table on page 5.

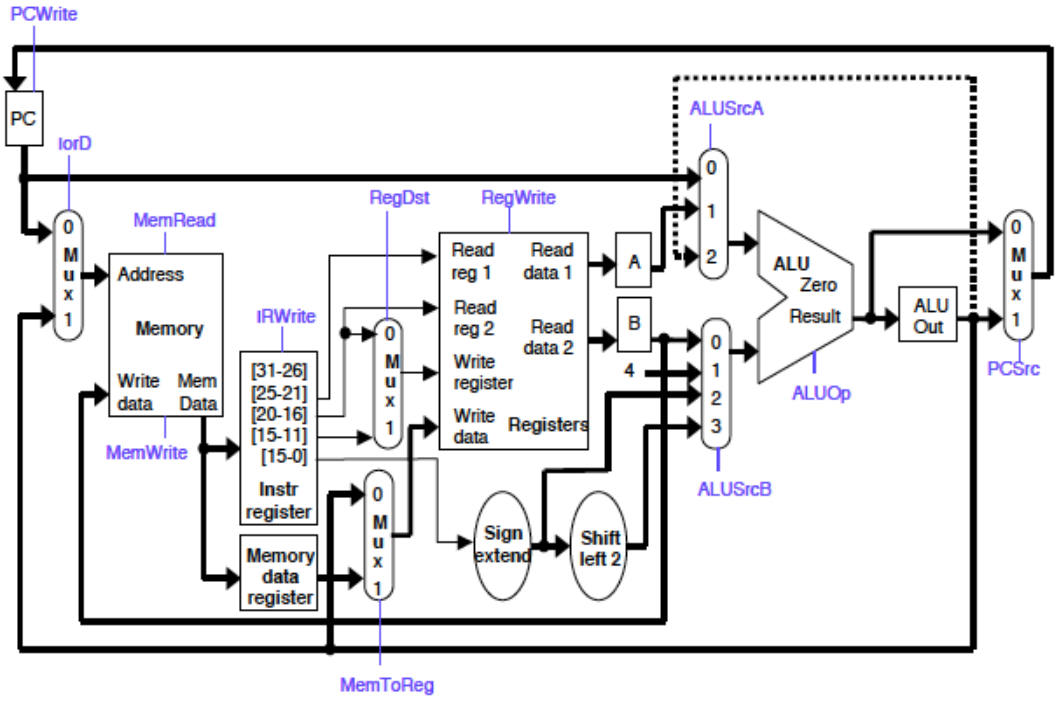
You only need to do one or the other, not both.

The multicycle datapath from lecture is shown below, with one important change: the ALUOut register is connected to the ALUSrcA mux (shown with a dotted line), which now has three inputs instead of two. No other changes to the datapath are needed.

You may assume that ALUOp = 100 performs an integer multiplication, and 010 performs an addition.

Finally, here is the instruction format, for your reference (shamt and func are not used):

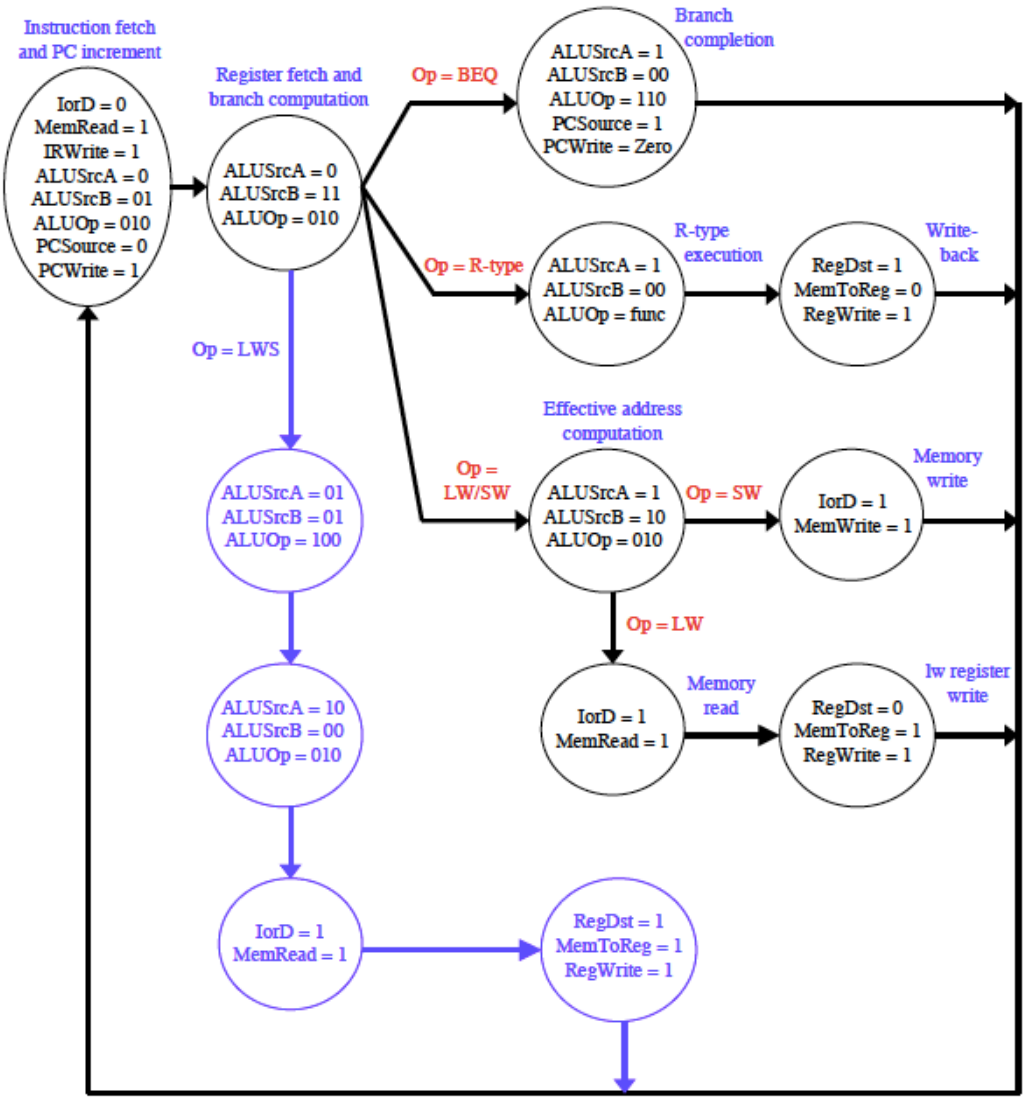
Field	op	rs	rt	rd	shamt	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0



Question 2 continued

Complete this finite state machine diagram for the lws instruction, *or* fill in the microprogramming table on the next page, *but not both!*

You can show the control values in either binary or decimal, whichever is more convenient for you.



See the comments on the next page.

Question 2 continued

Fill in this microprogramming table for the lws instruction, *or* complete the finite state machine diagram on the previous page, *but not both!*

You may need to make up new values for some of these fields, but just make sure your intentions are clear.

Label	ALU Control	Src1	Src2	Register control	Memory	PCWrite control	Next
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshift	Read			Dispatch 1
BEQ1	Sub	A	B			ALU-Zero	Fetch
Rtype1	Func	A	B				Seq
				Write ALU			Fetch
Mem1	Add	A	Extend				Dispatch 2
SW2					Write ALU		Fetch
LW2					Read ALU		Seq
				Write MDR			Fetch
LWS1	Multiply	A	4				Seq
	Add	ALU Out	B				Seq
					Read ALU		Seq
				Write MDR to register rd			Fetch

You should only need to add a few symbols such as "Multiply" to get the ALU to multiply, and "ALUOut" to set the value of the expanded ALUSrcA mux.

*Otherwise, the solution here is basically the same as the one on the previous page. The "lws" instruction has a more complicated addressing mode, so two cycles are required to compute the effective address. The first new cycle computes (4 * rs), and the second computes (4 * rs) + rt. The final two stages of the "lws" are almost the same as for the original "lw" instruction, but we need to store in register rd instead of rt.*

Since we've split this datapath into five distinct parts, you cannot combine any of the stages of the "lws" instruction together. For example, "Write MDR to register rd" in the microprogram requires an address to be supplied from the ALUOut register, but that must be produced in the previous clock cycle.

Finally, we said that the intermediate registers A, B, ALUOut and MDR are implicitly written to on every clock cycle. In this situation it's all right to use B (in "Add ALUOut B") two cycles after it's written (in the Register control "Read" stage)—within those two cycles, IR does not change, so none of the inputs to the register file change, so the register file outputs will not change either.

Question 2: Multicycle CPU implementation

MIPS is a register-register architecture, where arithmetic source and destinations must be registers. But let's say we wanted to add a register-memory instruction:

addm rd, rs, rt # rd = rs + Mem[rt]

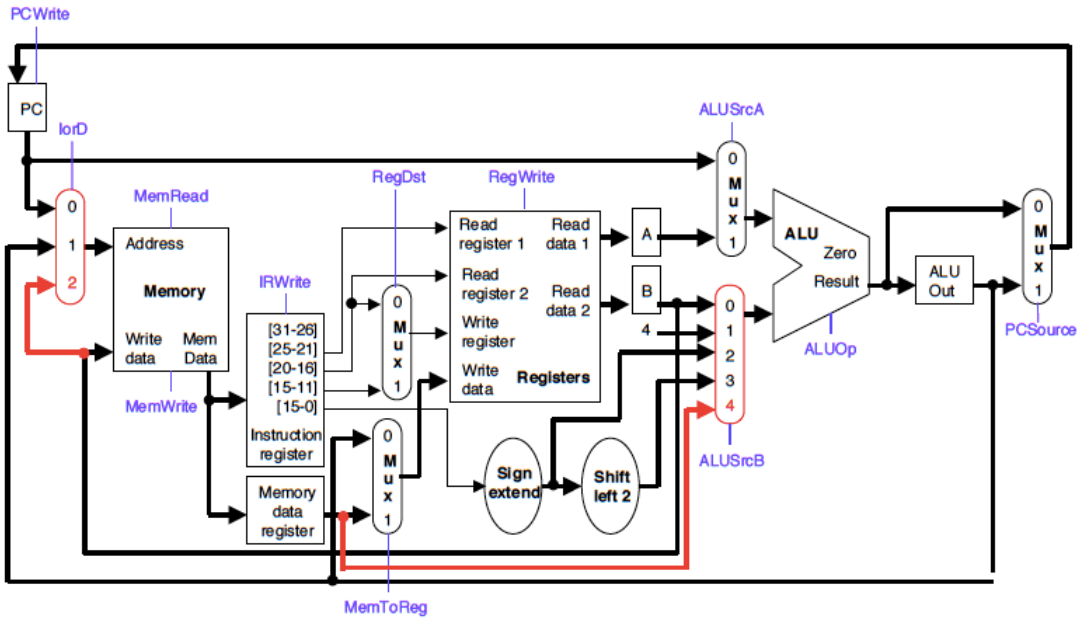
Here is the instruction format, for your reference (shamt and func are not used):

Field	op	rs	rt	rd	shamt	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

The multicycle datapath from lecture is shown below. You may assume that ALUOp = 010 performs an addition.

Part (a)
On the next page, show what changes are needed to support *addm* in the multicycle datapath. (10 points)

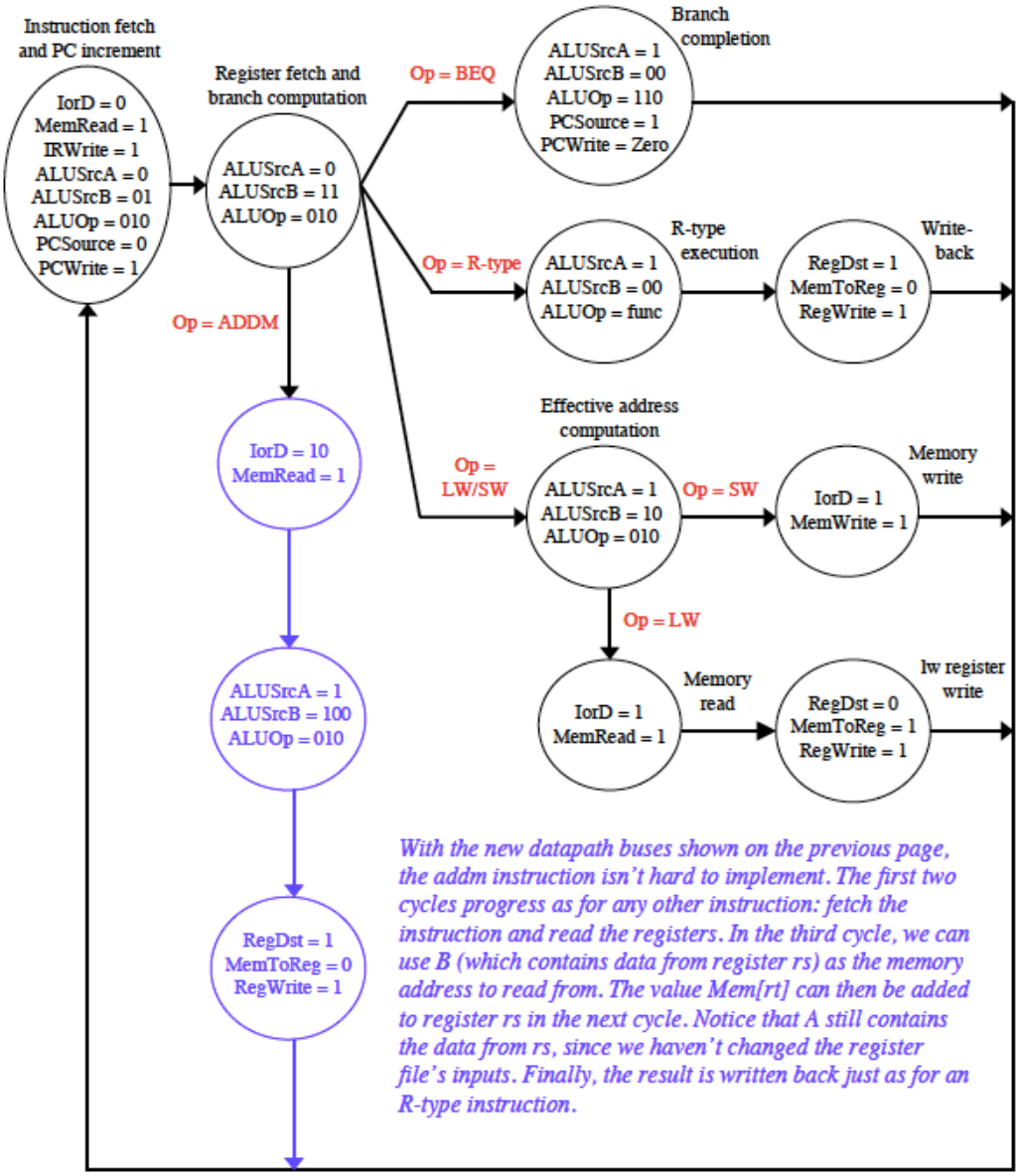
On the next page, we've connected the intermediate register B to the memory unit so we can read from address rt. We also feed MDR (which will contain Mem[rt]) into the ALU, for addition with register rs. These are probably the simplest set of changes; the control unit on the next page shows the details of how to get this to work.



Question 2 continued

Part (b)

Complete this finite state machine diagram for the *addm* instruction. Be sure to include any new control signals you may have added. (15 points)



Question 1: Multicycle CPU implementation (50 points)

Consider extending the MIPS architecture with the instruction below, which adds **three** registers together and stores the result in a register.

add3 rd, rs, rt, ru # rd = rs + rt + ru

This will use the same format as R-type instructions---shown here for reference---where the *shamt* field is used to hold *ru*.

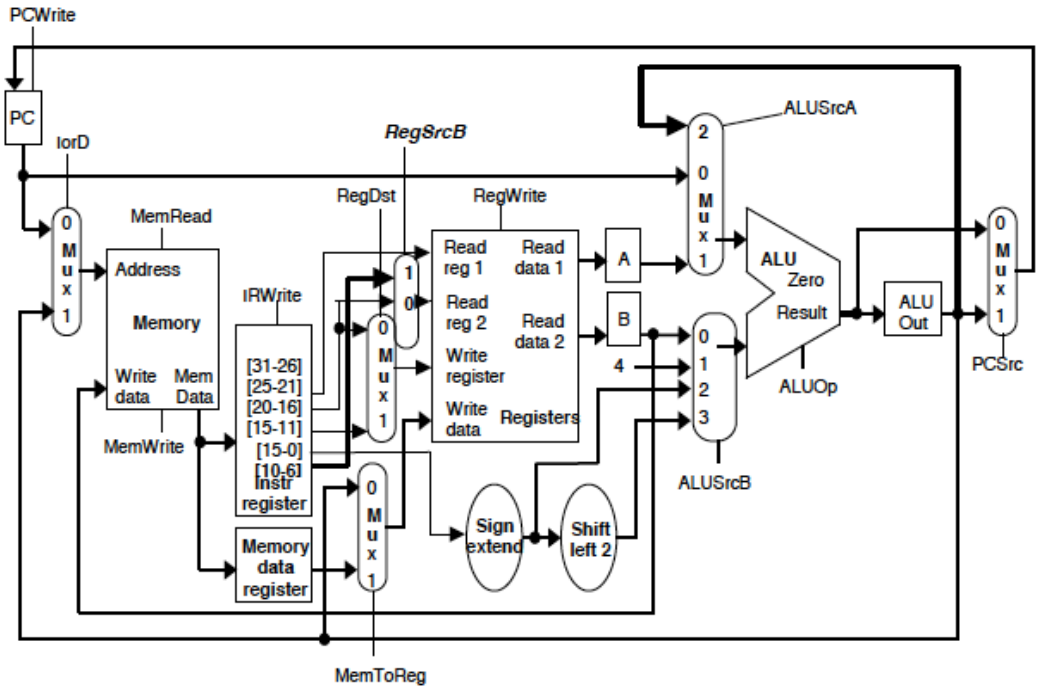
Field	op	rs	rt	rd	shamt/ru	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

An example of the usage of the *add3* instruction is shown in part (c) of question 1.

Part (a)

The multicycle datapath from lecture appears below. Show what changes are needed to support *add3*. You should only add wires and muxes to the datapath; do not modify the main functional units themselves (the memory, register file and ALU). Try to keep your diagram neat! (15 points)

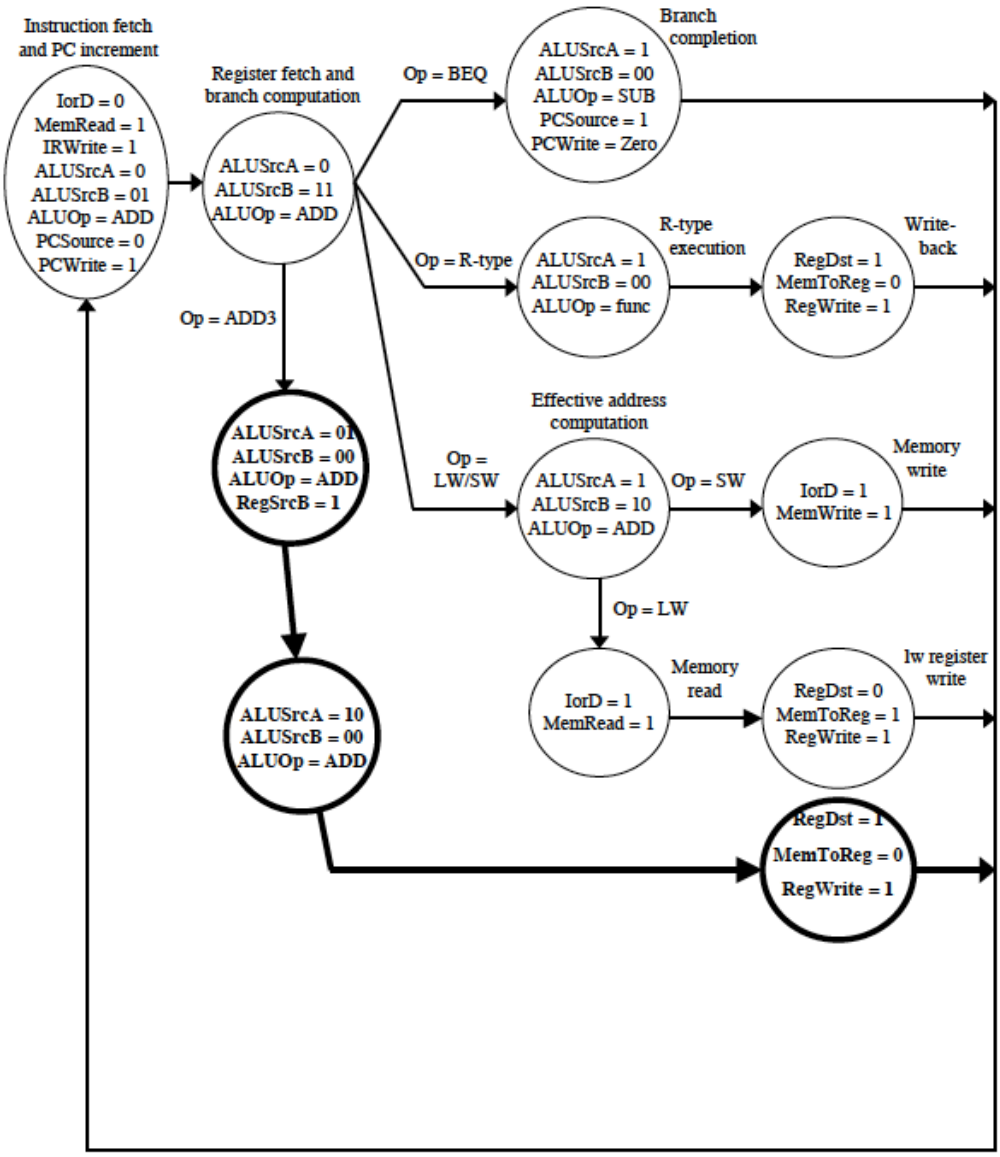
Note: While we're primarily concerned about correctness, full points will only be rewarded to solutions that use a minimal number of cycles and do not lengthen the clock cycle. Assume that the ALU, Memory and Register file all take 2ns, and everything else is instantaneous.



Question 1 continued

Part (b)

Complete this finite state machine diagram for the *add3* instruction. Control values not shown in each stage are assumed to be 0. Remember to account for any control signals that you added or modified in the previous part of the question! (20 points)



Question 1: Multicycle CPU implementation (35 points)

Consider extending the MIPS architecture with the instruction below, which loads two consecutive words of data from memory and stores them into two destination registers.

```
ld  rt, rd, rs       # rt = Mem[rs]; rd = Mem[rs + 4]
```

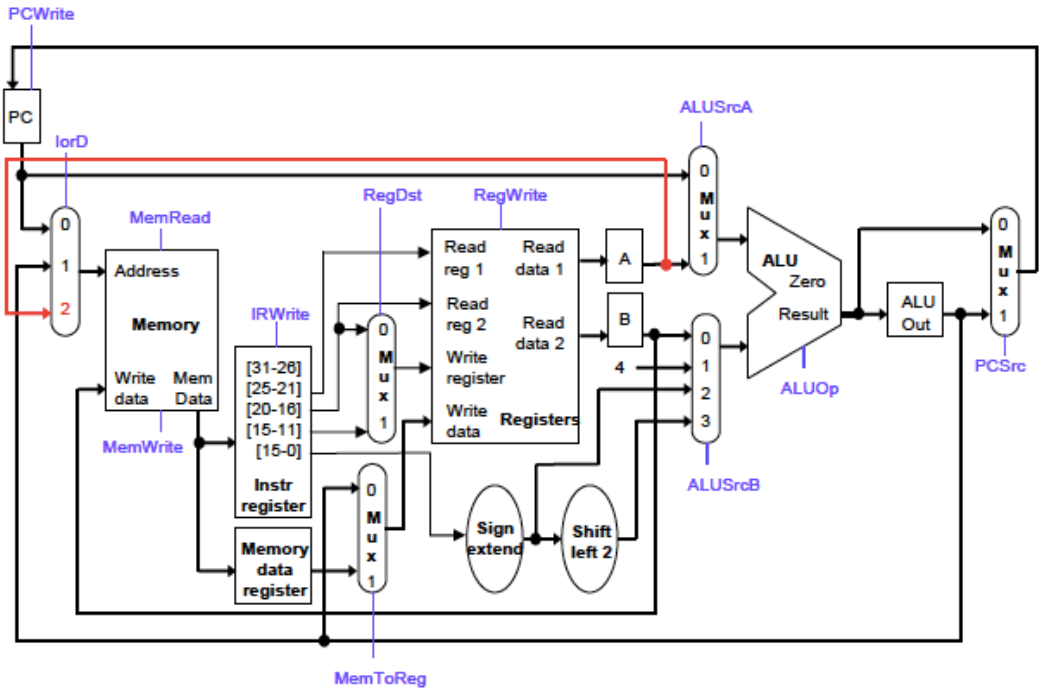
This will use the same format as R-type instructions, shown here for reference (shamt and func are not used).

Field	op	rs	rt	rd	shamt	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

Part (a)

The multicycle datapath from lecture appears below. Show what changes are needed to support *ld*. You should not need to modify the main functional units (the memory, register file and ALU), but you can make any other changes or additions necessary. Try to keep your diagram neat! (10 points)

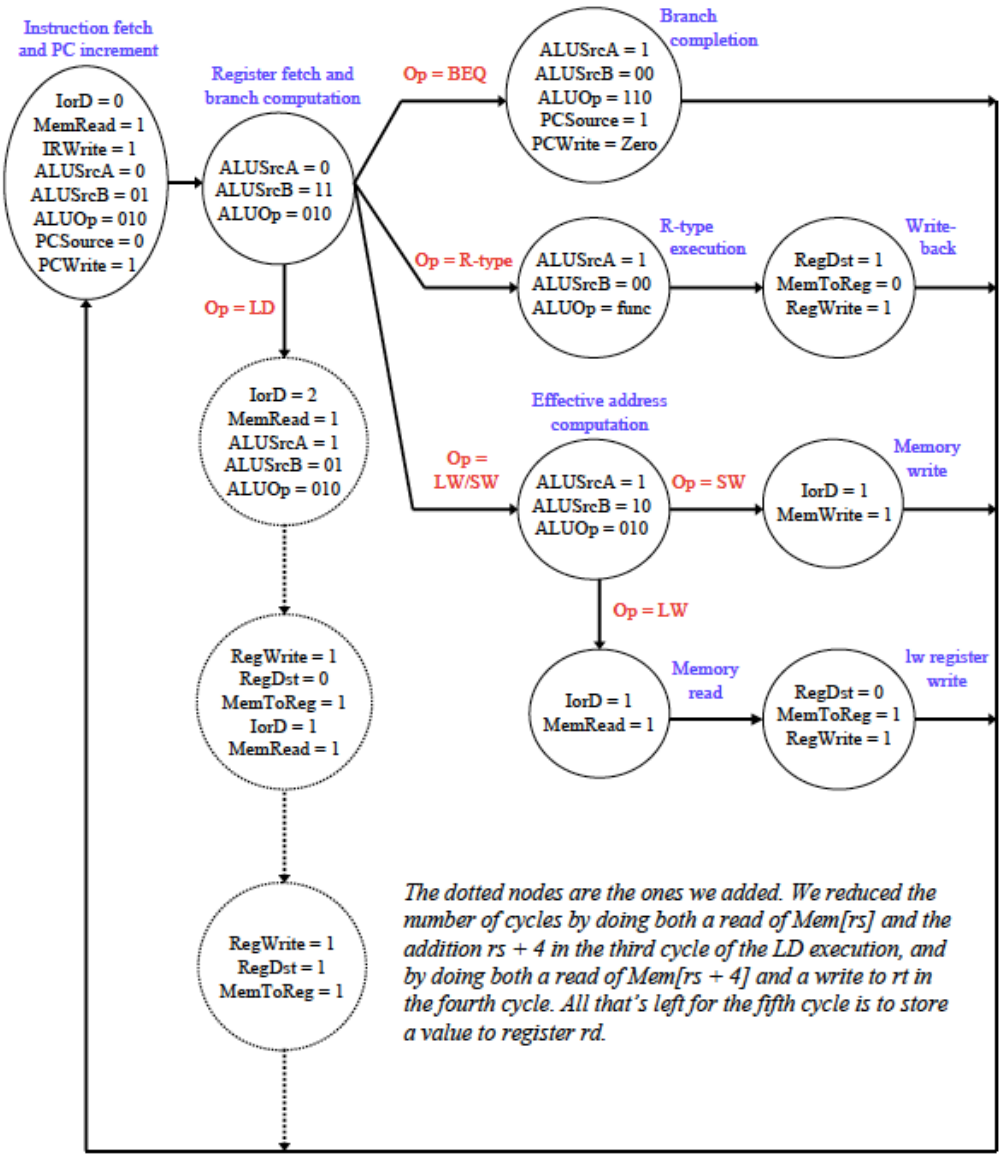
The ld instruction can be split into several smaller single-cycle operations: fetch and decode the instruction (as usual), read Mem[rs] and store it into register rt, compute rs + 4, and read Mem[rs + 4] and store that into rd. There are a few possible solutions, so we'll just show one of them. The sole datapath change we'll make is to connect A, which contains the value of register rs, to the memory's address input to let us read Mem[rs]. The IorD mux is expanded appropriately.



Question 1 continued

Part (b)

Complete this finite state machine diagram for the *ld* instruction. Control values not shown in each stage are assumed to be 0. Remember to account for any control signals that you added or modified in the previous part of the question! (25 points)



Question 2: Multi-cycle CPU implementation and its performance (50 points)

Assume that the ALU can perform the max2 operation (*i.e.*, return the greater of 2 inputs):

```
alu_result = (A_input > B_input) ? A_input : B_input;
```

ALUOp for this instruction is MAX2.

Given this improved ALU, implement the **max4** instruction that writes into register rd the largest value of 4 registers:

```
max4 rs, rt, rd, rm          # rd = max(rs, rt, rd, rm)
```

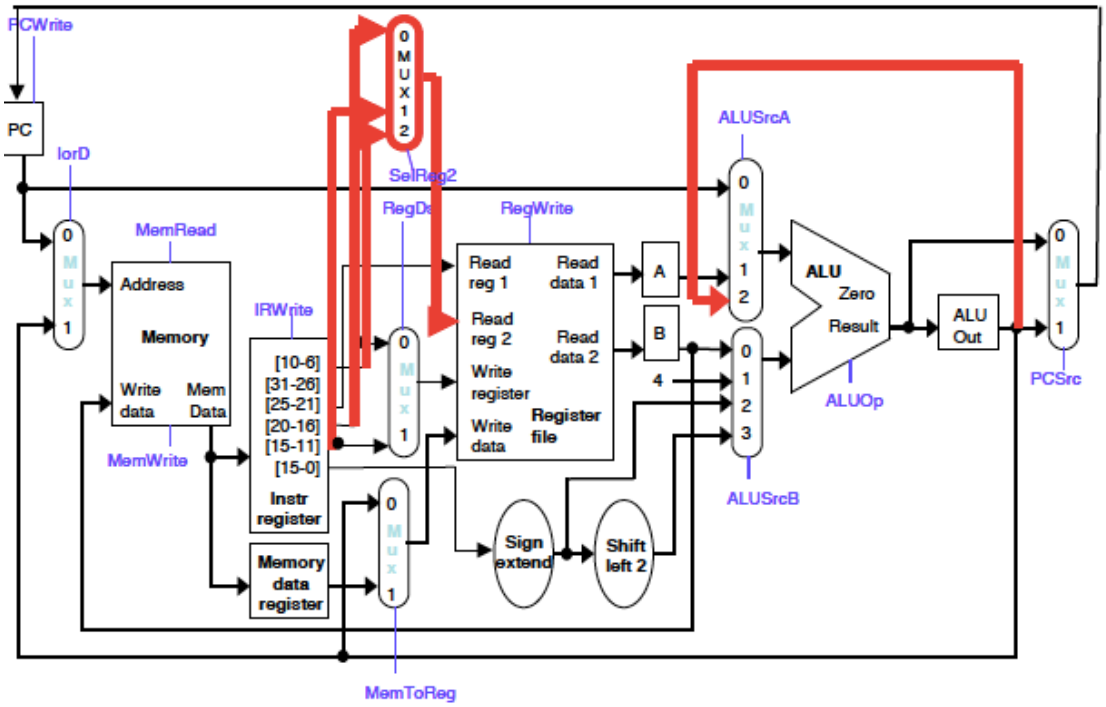
Note that register **rd** is both an *input* and an *output*. Instruction **max4** has the following format:

Field	op	rs	rt	rd	rm	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

Part (a)

The multicycle datapath from lecture appears below. Show what changes are needed to support **max4**. You should only add wires and muxes to the datapath; do not modify the main functional units themselves (the memory, register file, and ALU). Try to keep your diagram neat! (10 points)

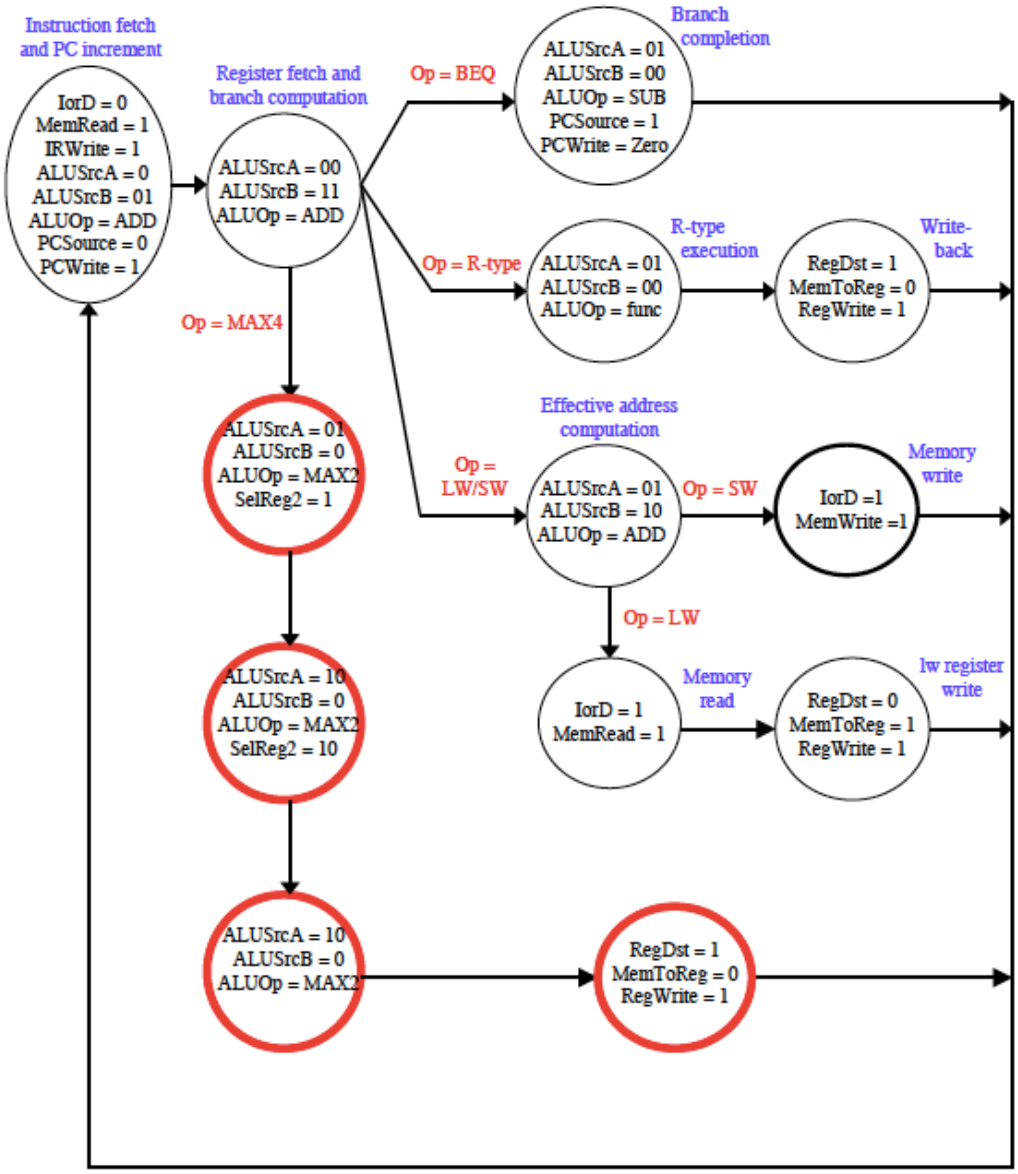
Note: While we're primarily concerned about correctness, full points will only be rewarded to solutions that use a minimal number of cycles and do not lengthen the clock cycle. Assume that the ALU, Memory and Register file all take 2ns, and everything else is instantaneous.



Question 2 continued

Part (b)

Complete this finite state machine diagram for the **max4** instruction. Control values not shown in each stage are assumed to be 0. Remember to account for any control signals that you added or modified in the previous part of the question! (15 points)



Question 2, Multi-cycle implementation (40 points)

The (imaginary) *jump memory* (jmem) instruction is like a *jump-and-link* (jal) instruction, except both the target is loaded from memory and the return address is saved to memory. The i-type format is used, as shown below. You can assume that R[rt] and (R[rs] + offset) are distinct (non-overlapping) addresses.

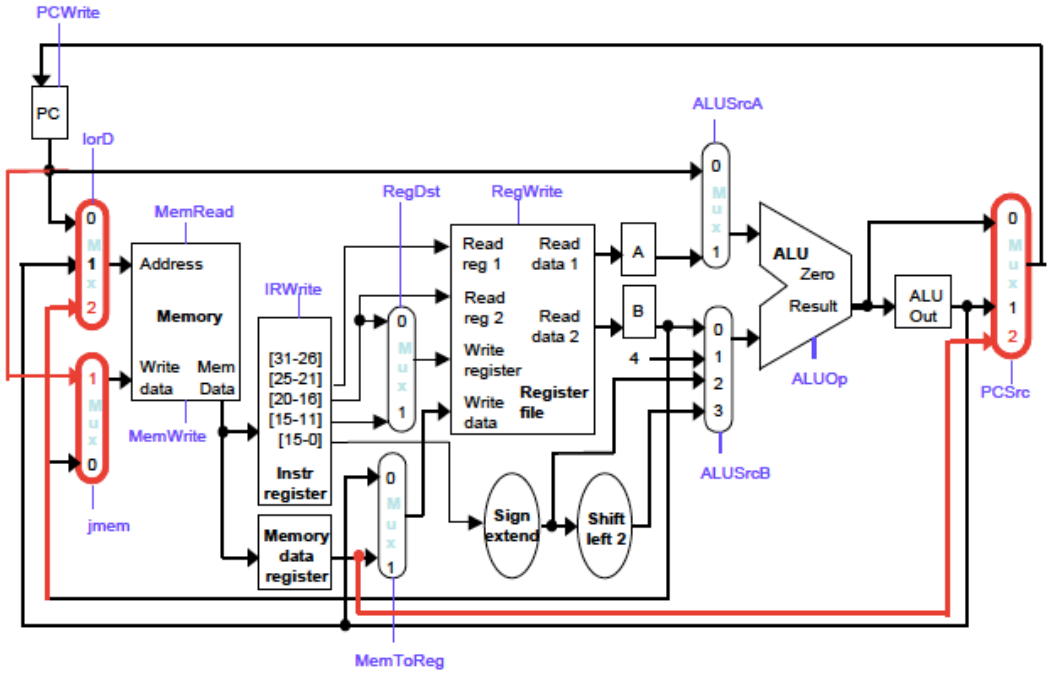
```
jmem (rt), offset(rs)          # Memory[R[rs]+offset] = PC+4;
                               # PC = Memory[R[rt]]
```

Field	op	rs	rt	imm
Bits	31-26	25-21	20-16	15-0

Part (a)
The multicycle datapath from lecture appears below. Show what changes are needed to support **jmem**. You should only add wires and muxes to the datapath; do not modify the main functional units themselves (the memory, register file, and ALU). Try to keep your diagram neat! (15 points)

Note: While we're primarily concerned about correctness, five (5) of the points will only be rewarded to solutions that use a minimal number of cycles and do not lengthen the clock cycle. Assume that everything besides the ALU, Memory and Register File is instantaneous.

Obviously there are many ways to implement this instruction. We show a solution that accomplishes it in 4 cycles. All solutions are going to require adding datapath from the PC register to the Write data port on the memory and from the MemData port on the memory to the PC.



Question 2, continued

Part (b)

Complete this finite state machine diagram for the **jmem** instruction. Control values not shown in each stage are assumed to be 0. Remember to account for any control signals that you added or modified in the previous part of the question! (25 points)