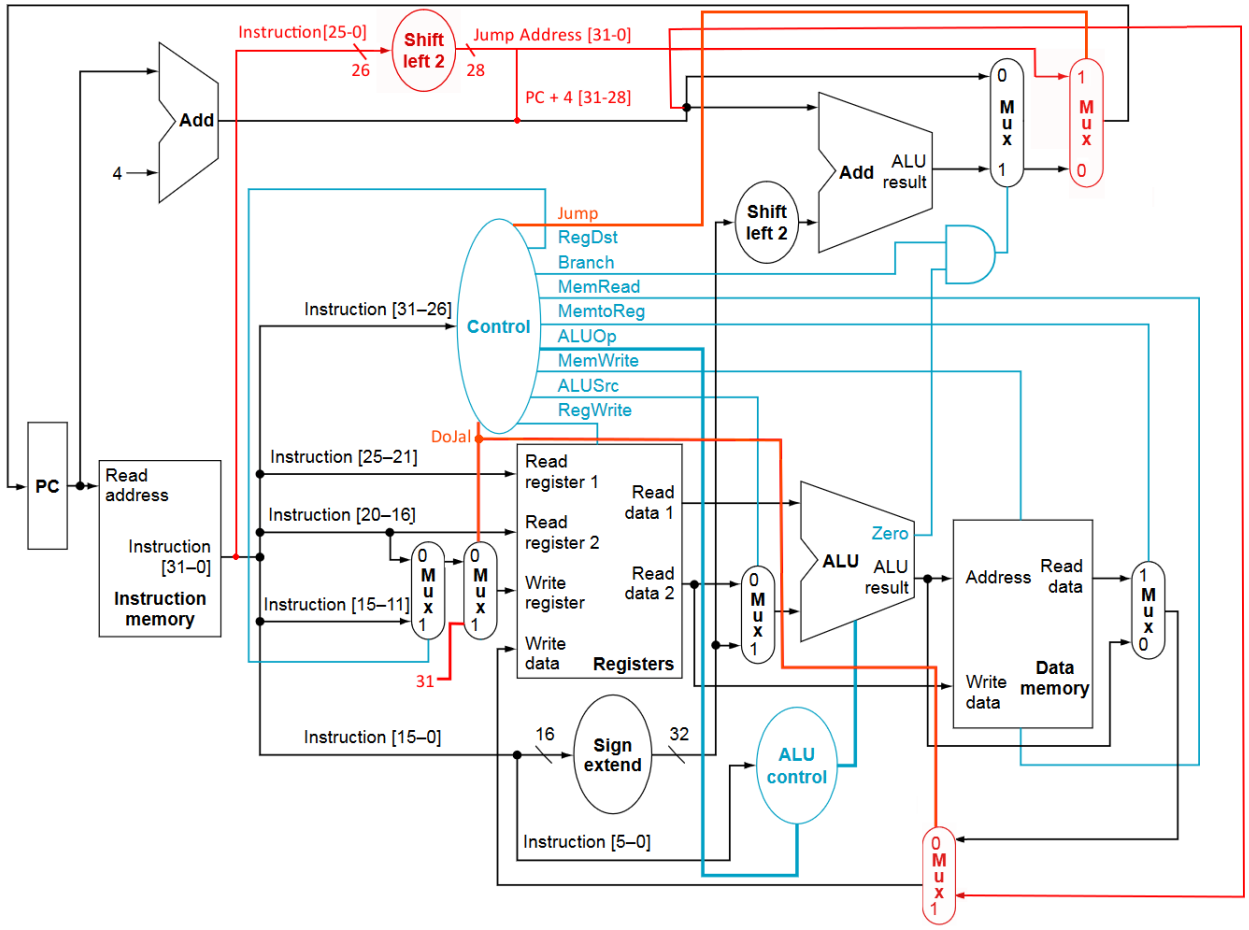
CS151B Spring 2017 Homework #3 Solutions

Problem (1)

A) The changes made for the "j" instruction (Fig. 4.24 in the book) have to be applied. Beyond that, we add two new MUXes:
1) A 5-bit two-input MUX that allows the constant 31 to be driven to the "Write register" input of the register file
2) A 32-bit two-input MUX that allows the incremented PC to be driven to the "Write data" input of the register file.



B) Two new control signals:
1) `Jump`: As in Fig. 4.24, when asserted causes the jump target to be driven to the PC inputs
2) `DoJal`: When asserted, drives the constant 31 to the "Write register" input of the register file and the incremented PC to the "Write data" input of the register file.

C)

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

Problem (2)

```
rfunc:      bgtz  $a0, $zero, rfunc0    # if (n <= 0) ...
            addi  $v0, $zero, 3         # ... return 3
            jr    $ra
rfunc0:     addi  $v0, $zero, 1
            bne   $a0, $v0, rfunc1      # if (n == 1) ...
            addi  $v0, $zero, 2         # ... return 2
            jr    $ra

rfunc1:     addi  $sp, $sp, -8          # push onto stack...
                                        ## Compute rfunc(⌊n/4⌋+1)
            sw    $ra, 4($sp)           # push return address
            sw    $s0, 0($sp)           # push $s0
            add   $s0, $a0, $zero       # save n in $s0
            srl   $a0, $a0, 2           # n = ⌊n/4⌋
            addi  $a0, $a0, 1           # pass argument ⌊n/4⌋+1
            jal   rfunc                 # ... to rfunc
                                        ## Compute rfunc(⌊n/8⌋−1)
            srl   $a0, $s0, 3           # n = ⌊n/8⌋
            addi  $a0, $a0, -1          # n = ⌊n/8⌋−1
```

```
add   $s0, $v0, $zero      # save rfunc(⌊n/4⌋ + 1) in $s0
jal   rfunc                # call rfunc(⌊n/8⌋ −1)
add   $v0, $s0, $v0        # return rfunc(⌊n/4⌋ +1) +rfunc(⌊n/8⌋ −1)
lw    $s0, 0($sp)          # pop $s0
lw    $ra, 4($sp)          # pop return address
addi  $sp, $sp, 8          # ... from stack
jr    $ra                  # return to caller
```

Problem (3)

A) Yes

B) In a single-cycle datapath, the cycle time is determined by the slowest instruction. In theory, there could be some implementation for which the increase in the operation time for an AND instruction causes it to become the slowest instruction. This would then require the cycle time to be increased for all instructions, thus slowing down the execution of the entire program.

C) No

D) Similar to slide 5.13 in the class notes, compare the operations performed by the AND instruction and the LW instrction:

AND	LW
Instruction fetch	Instruction fetch
Register file read	Register file read
ALU performs AND operation	ALU performs ADD operation
Register file write	Data memory read
	Register file write

The ALU performs addition using a carry propagate adder. Thus, a lower bound on the worst-case latency for addition is 32 gate delays. On the other hand, the latency if an AND operation is one gate delay (plus the delay of the MUXes which are also there for addition). Thus, even if the latency of the AND is doubled, the latency for AND is still less than the latency of addition. Furthermore, the LW also has to read from the data memory, which the AND instruction does not do. Thus, even with the new implementation, for any conceivable implementation, the LW instruction has a longer latency than the AND instruction. Since the cycle time is determined by the slowest instruction, the cycle time will not change in the new implementation. Since performance is determined by the instruction count and cycle time, program execution time will not change with the new implementation.

Problem (4)

The only result observed by a user/programmer will be that for a lw instruction, the value loaded into the Rt register will not be the value at the specified address in memory. Instead, the value loaded into Rt will appear random but will actually be the sum of the previous value of the Rt register and the address of the word following the lw instruction.

Based on Figure 5.37, the only state when MemtoReg has to be asserted is state 4 (the last cycle of a lw instruction). Hence, the fault will only affect the operation of lw, causing the contents in the ALUOut register, instead of the value in MDR, to be stored into register R[$IR_{20..16}$] (the lw destination register). Since ALUOp, ALUSrcA, and ALUSrcB are not listed in state 3 in Figure 5.37, they are all set to 0. Hence, in state 3, the ALU will add the value of the PC register (that has already been incremented in state 0) with the previous value of Rt (loaded into the B register in state 2).

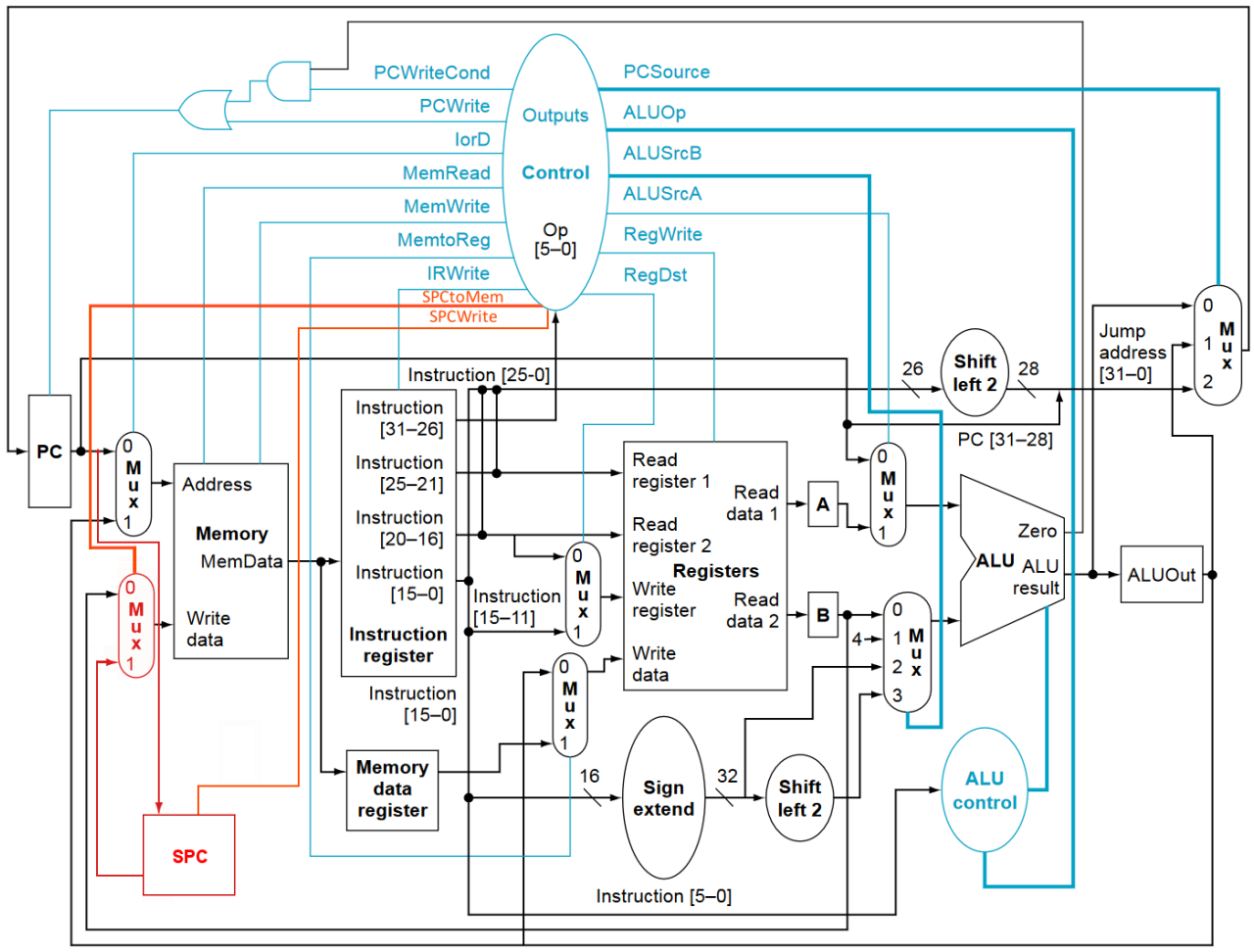
Problem (5)

The instruction format for `swpc` is:

`swpc rt, offset(rs)`

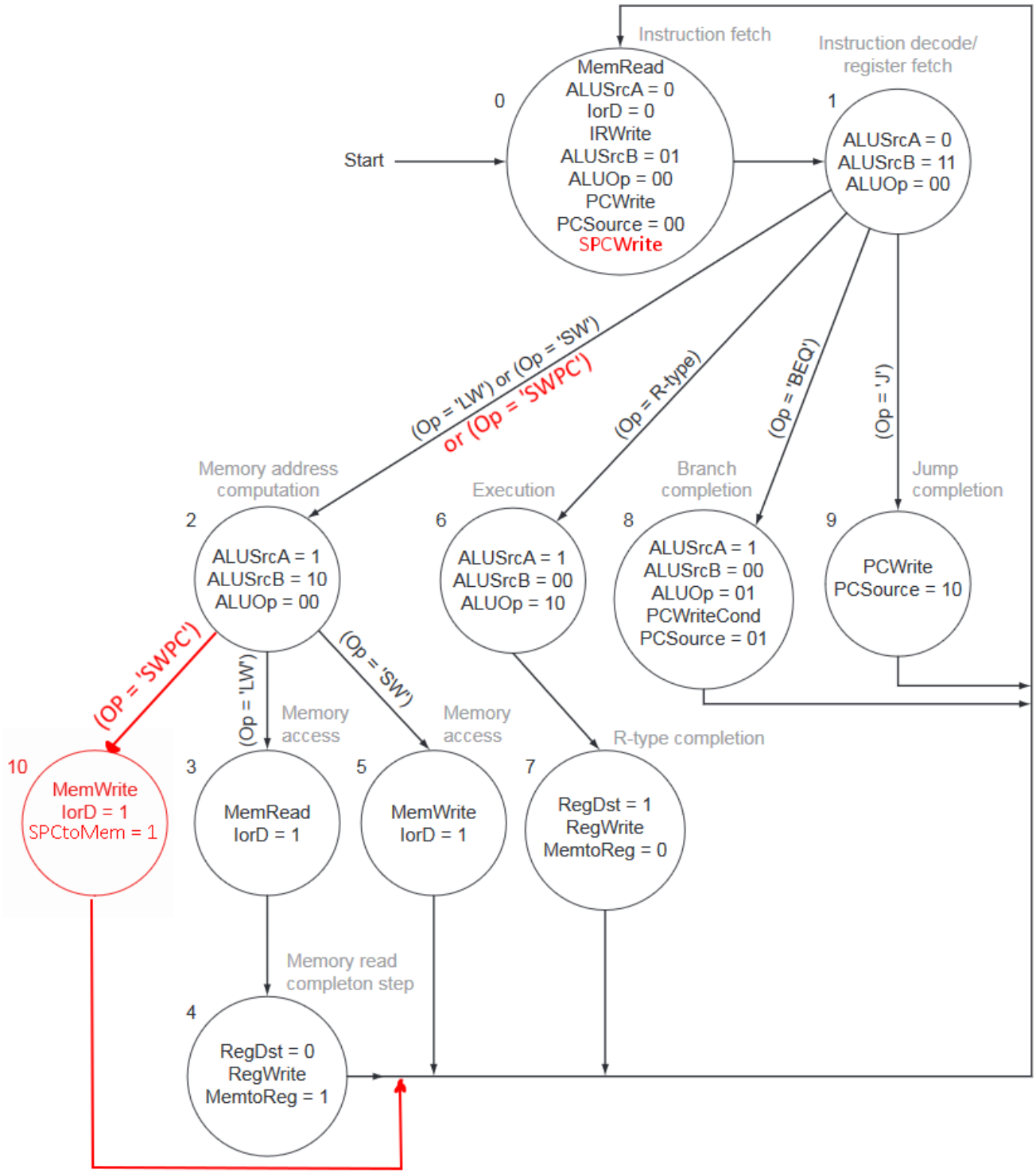
A) Since `swpc` stores the value of PC before it is incremented, a new register, SPC (Saved PC), is added to maintain that value speculatively, for all instructions. A new 32-bit two-input MUX is added to allow the value of SPC (instead of the value of the B register) to be written to memory.

B) The new SPC register is identical to the IR register.
The new SPCtoMem MUX is identical to the IorD MUX.



C) There are two new control signals.
1) `SPCWrite`: when asserted, the current value in PC is written to SPC.
2) `SPCtoMem`: when asserted, the current value in SPC is driven to the "Write data" input of the memory.

D) There are three changes:
1) SPCWrite is asserted in State 0.
2) When the instruction is SWPC, there is a state transition from State 1 to State 2.
3) A new state, State 10, is added, with a transition to this state from State 2 when the instruction is SWPC.



E) Four cycles.

Solutions to Homework #3 Practice Problems

Problem (6)

```
bfind:      ori    $t0,$zero,0x0062       # 0x0062 is the character
                                          # 'b' in ascii
notfound:   lbu    $t1,0($a0)             # load a character
            beq    $t1,$t0, found         # 'b' found
            beq    $t1,$zero,found        # 'NULL' found
            addi   $a0, $a0, 1            # increment pointer
            j      notfound               # examine the next character
found:      add    $v0, $a0, $zero        # set up return value
            jr     $ra                    # return to the caller
A more efficient implementation of the bfind procedure is:
bfind:      ori    $t0,$zero,0x0062       # 0x0062 is the character
                                          # 'b' in ascii
            j      load                   # examine the first character
next:       addi   $a0,$a0, 1             # increment pointer
load:       lbu    $t1,0($a0)             # load a character
            beq    $t1,$t0,found          # 'b' found
            bne    $t1,$zero,next         # 'NULL' found
found:      add    $v0,$a0,$zero          # set up return value
            jr     $ra                    # return to the caller
```

Problem (7)

```
ackerman:   bne    $a0, $zero, recurse1   # test for the base case m = 0
            addi   $v0, $a1, 1            # return n + 1
            jr     $ra
recurse1:   bne    $zero, $a1, recurse2   # test for case n = 0
            addi   $sp, $sp, -4
            sw     $ra, 0($sp)            # push $ra
            addi   $a0, $a0, -1           # compute m - 1
            addi   $a1, $zero, 1
            jal    ackerman               # ackerman(m-1,1)
            lw     $ra, 0($sp)            # pop $ra
            addi   $sp, $sp, 4
            jr     $ra                    # return ackerman(m-1,1)
recurse2:   addi   $sp, $sp, -8
            sw     $ra, 4($sp)            # push $ra
            sw     $a0, 0($sp)            # push $a0
            addi   $a1, $a1, -1           # compute n - 1
            jal    ackerman               # ackerman(m,n-1)
            add    $a1, $v0, $zero        # move return value to $a1
            lw     $a0, 0($sp)            # $a0 not preserved across call
            addi   $a0, $a0, -1           # compute m - 1
            jal    ackerman               # ackerman(m-1,ackerman(m,n-1))
            lw     $ra, 4($sp)            # pop $ra
            addi   $sp, $sp, 8
            jr     $ra                    # return ackerman(m-1,ackerman(m,n-1))
```

Problem (8)

The fault is manifested only when the specified values of RegWrite and the most-significant bit (MSB) of ALUSrcB and differ in the same cycle. Examining the state diagram of Figure 5.37, one sees that states 1, 2, 4, and 7 are impacted.
In states 4 and 7, RegWrite is 1 and the ALUSrcB should be 00. Instead, ALUSrcB will be 10 because of the fault. Neither state uses the output of the ALU. Furthermore, for both states 4 and 7, the succeeding state is state 0, and state 0 does not use the output of the ALUOut register. Therefore, changing the bottom ALU input in states 4 and 7 has no impact on the execution of programs by the processor.

In states 1, and 2, the MSB of ALUSrcB is 1 and RegWrite should be 0. Instead, RegWrite will be 1 because of the fault. The following additional state change will occur in these states: $R[IR_{20..16}] \leftarrow$ ALUOut.

For state 1, the previous state is state 0. During state 0, the ALU computes PC + 4 and that value is stored in ALUOut. Hence, during state 1, $R[IR_{20..16}] \leftarrow PC + 4$, where PC is the address of the current instruction.

For state 2, the previous state is state 1. During state 1, the branch target for the `beq` instruction is computed (regardless of the actual instruction being executed). Hence, during state 2, $R[IR_{20..16}] \leftarrow PC + 4 + SignExt(IR_{15..0} \,||\, 00)$ where PC is the address of the current instruction.

The effect of the fault on state 1 impacts all instructions.

The effect of the fault on state 2 impacts `lw` and `sw`.

Note that for `sw`, the impact of the fault in state 1 is masked by the impact of the fault in state 2. For `lw`, the fault is masked because $R[IR_{20..16}]$ is overwritten with the correct value in state 4.

Therefore, the user/programmer will observe the following incorrect behavior:

1) When a `sw` instruction is executed, if the Rt field is not 0, the value of the register specified in the Rt field will be overwritten by the value $PC + 4 + SignExt(Instruction_{15..0} \,||\, 00)$, where PC is the address of the `sw` instruction and $Instruction_{15..0}$ is the immediate field of the `sw` instruction.
2) When an R-type instruction is executed, if the Rt field is not 0, **and** the register numbers specified in the Rt and Rd fields are different, the value of the register specified in the Rt field will be overwritten by the value PC + 4, where PC is the address of the R-type instruction.
3) When a `beq` instruction is executed, if the Rt field is not 0, the value of the register specified in the Rt field will be overwritten by the value PC + 4, where PC is the address of the `beq` instruction.
4) When a `j` instruction is executed, if the $Instruction_{20..16}$ is not 0, the value of the register specified in $Instruction_{20..16}$ will be overwritten by the value PC + 4, where PC is the address of the `j` instruction.

Problem (9)

The instruction format for `lui` (page A-57) is:

`lui rt, Imm`

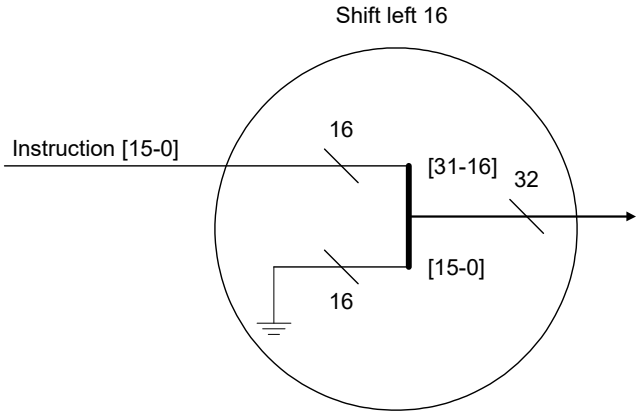
0xf	0	rt	Imm
6	5	5	16

A)

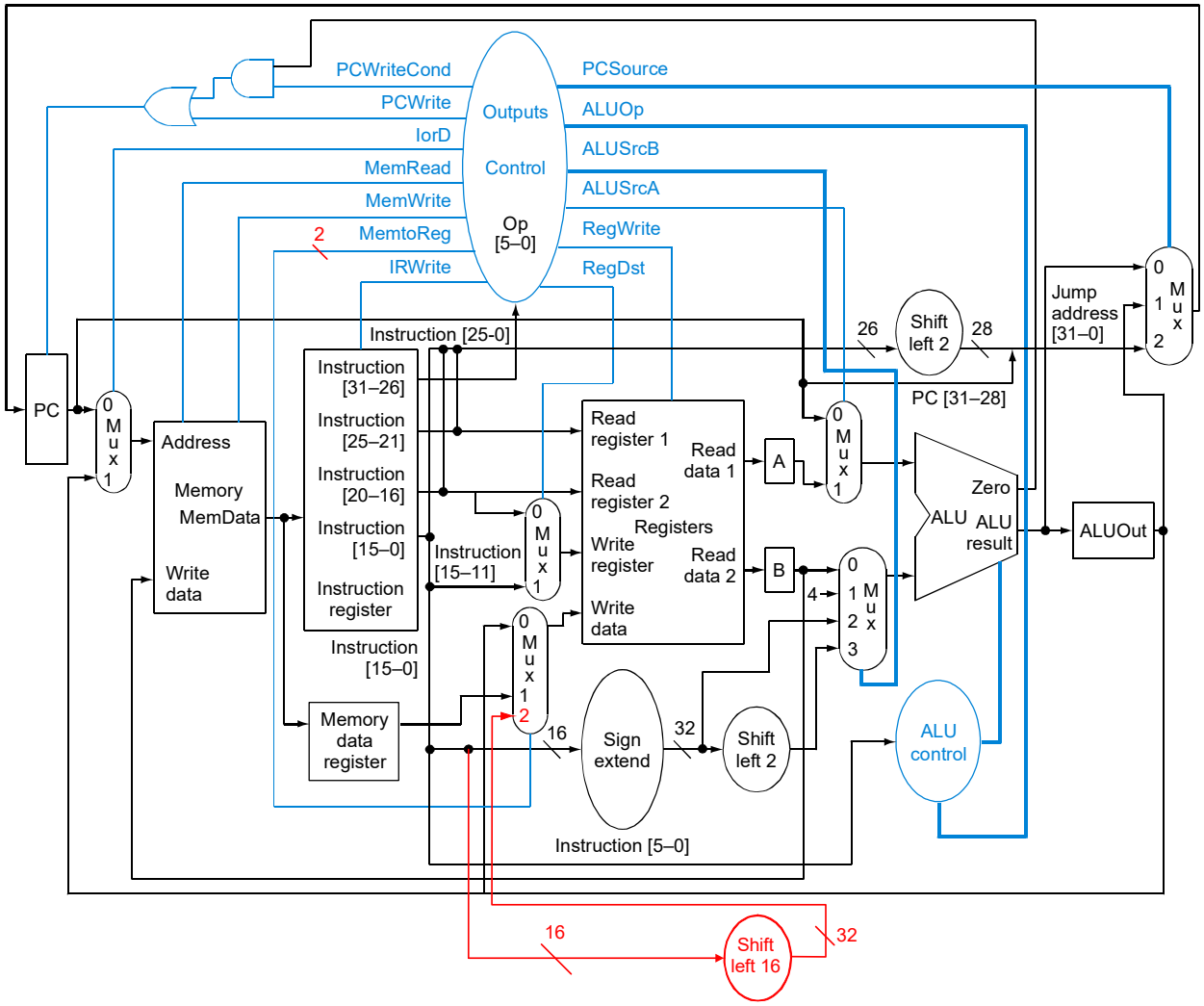
Instruction fetch (unchanged)
Instruction decode and register fetch (unchanged)
Register write: R[$IR_{20..16}$] = Shift-left-by-16([$IR_{15..0}$])

We add to the datapath a module called “Shift left 16” to perform the logical shift-left-by-16-bits of the immediate field, and use this as an additional input to the MemtoReg MUX.

B) The modified datapath requires replacing the MemtoReg MUX with a new three input MUX.
The modified datapath requires a new module -- "Shift Left 16". This new module is implemented as follows:



Problem (9) Shift Left 16 Module



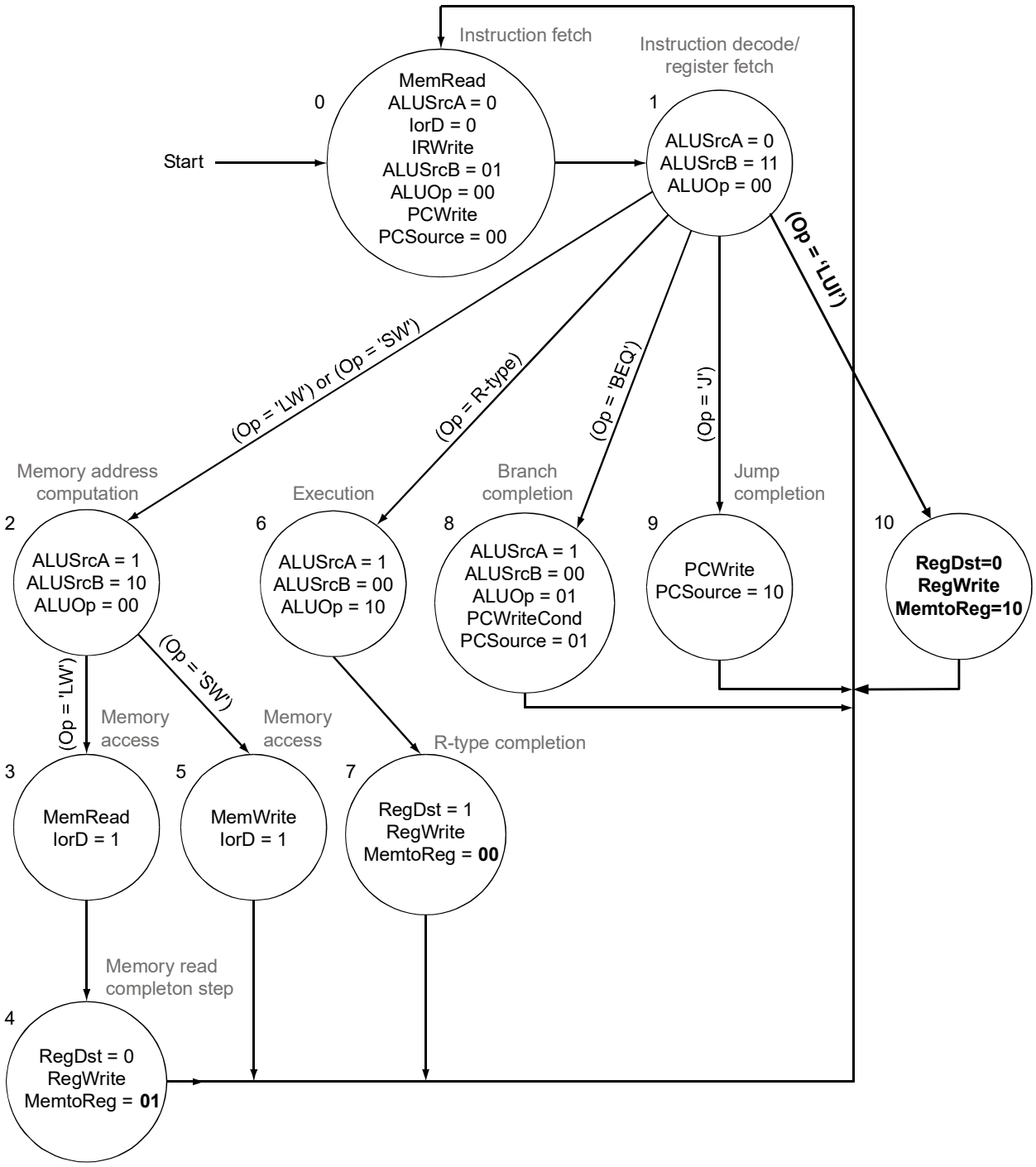
Problem (9) Datapath

C) Since the replacement MemtoReg MUX has three inputs, it requires two bits of control instead of one. Thus, the control unit must generate one more control signal bit. Collectively, the two control signals for the replacement MUX are now referred to as "MemtoReg".

MemtoReg1	MemtoReg0	MUX output
0	0	ALUOut
0	1	Memory Data Register
1	**0**	Shift-left-16 immediate

D)

The third cycle of `lui` differs from any cycle of any of the already implemented instructions. Thus, we must add one new state to our finite state machine (i.e., state10), a transition from state1 to state10 if Op='LUI', and a transition from state10 back to state0. In state10 we write the register, so RegWrite = 1, MemtoReg = 10, RegDst = 0. Note that we also must indicate the value of MemtoReg1 in states 4 and 7.



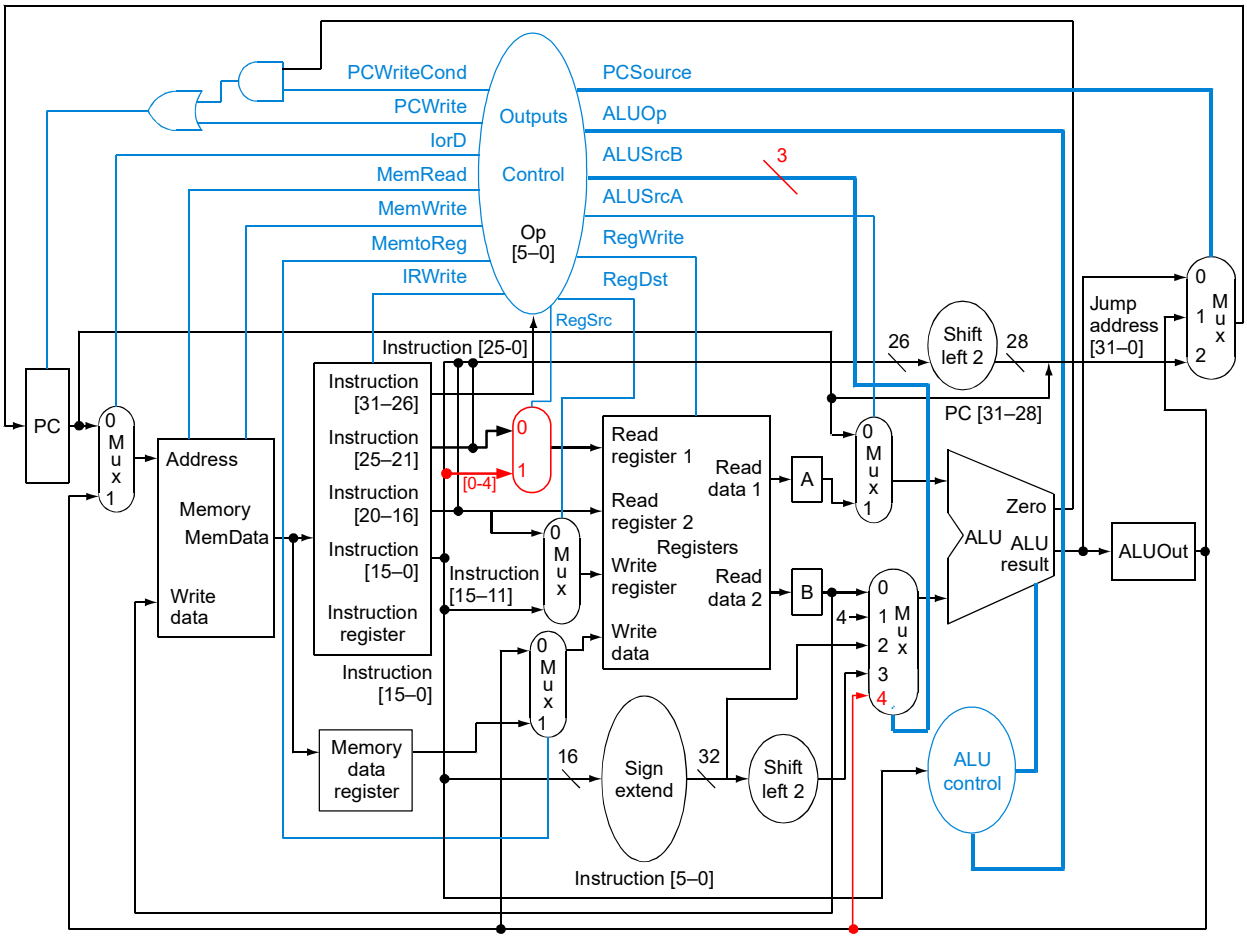
Problem (9) State Machine Diagram

E) Three cycles.

Problem (10)

A) The instruction requires using the ALU twice. The ALUSrcB multiplexor is replaced by a larger multiplexor that allows the ALU to use the previous result produced by the ALU (stored in ALUOut) as an input for the second addition. In order to read from the register file the register specified in IR[4-0], a new 5-bit two input MUX is added to drive the "Read Register 1" input of the register file from either IR[25..21] or IR[4..0]. The third operand for the summation is read from the register file during the cycle in which the first addition is performed.

B) The ALUSrcB MUX is replaced by a 32-bit five input MUX. A new 5-bit two-input MUX is added.

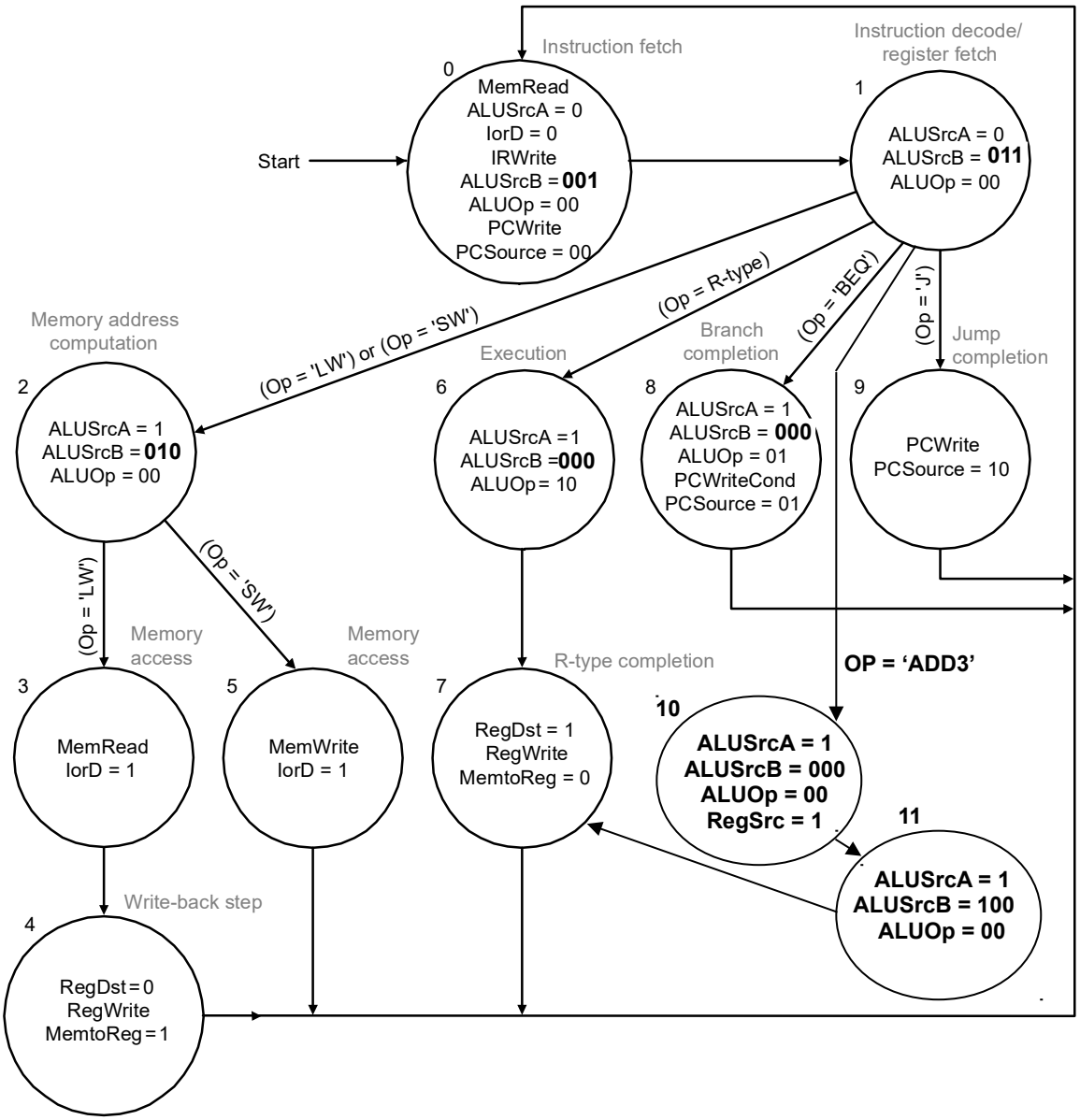


Problem (10) Datapath

C) Since the replacement ALUSrcB MUX has five inputs, it requires three bits of control instead of two. Thus, the control unit must generate one more control signal bit. Collectively, the three control signals for the replacement MUX are now referred to as "ALUSrcB".

RegSrc is a new control signal for the new MUX that drives the register file "Read Register 1" input.

D)



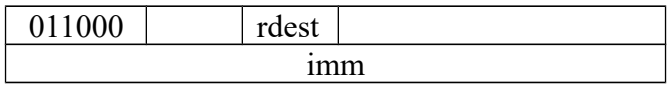
Problem (10) State Machine Diagram

E) Five cycles.

Problem (11)

Opcode “011000” can be used for this instruction.

`li rdest, imm`



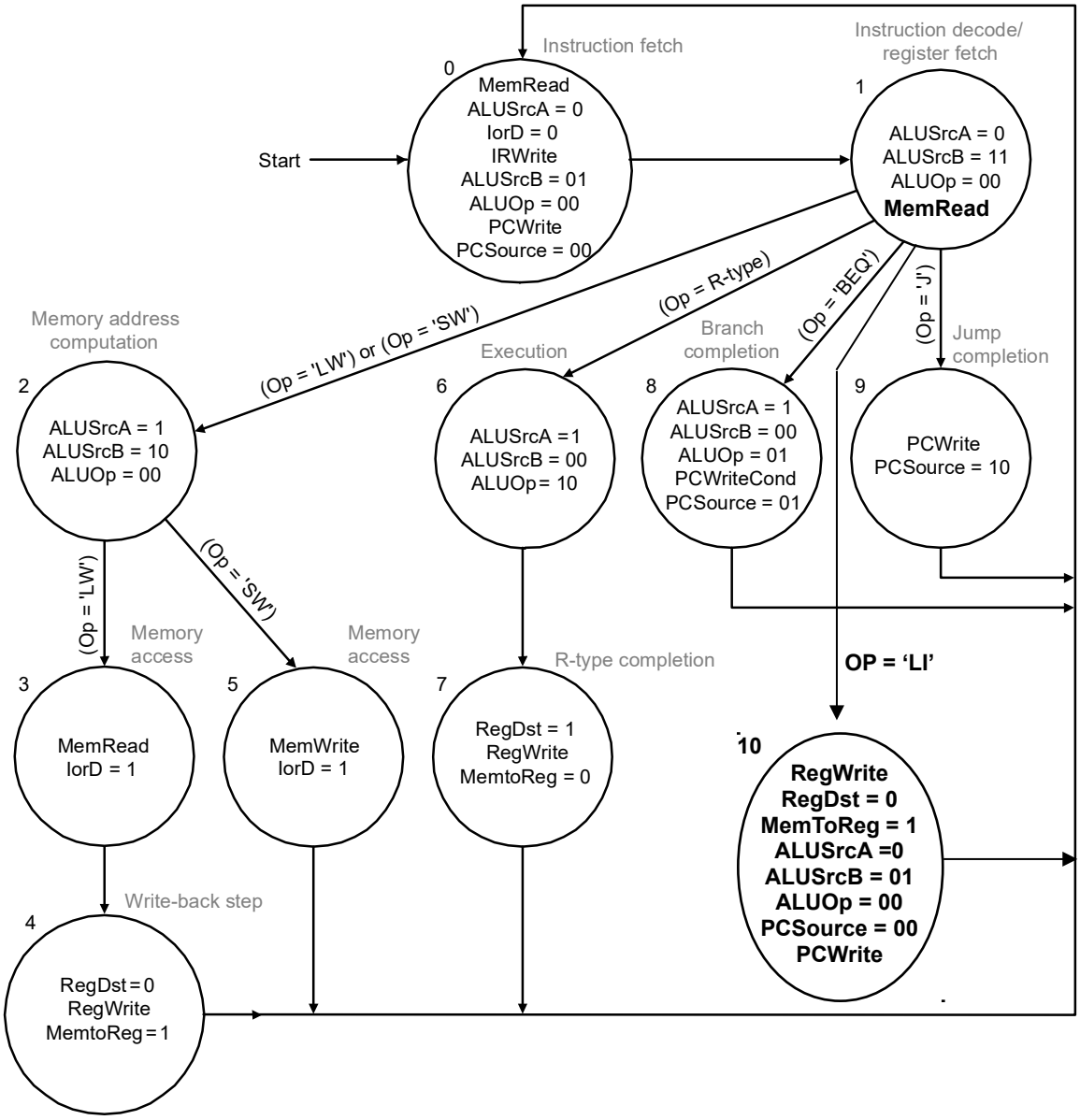
R[$rdest] ← imm;

A) No datapath changes are needed. The second word of the instruction is speculatively read from memory to the MDR during State 1 of all instructions. If the instruction is `li`, the content of MDR is copied to the destination register and the PC is incremented by 4 so that it contains the address of the next instruction.

B) The datapath does not need to be modified.

C) No new control signals are needed.

D)



E) Three cycles.

Problem (12)

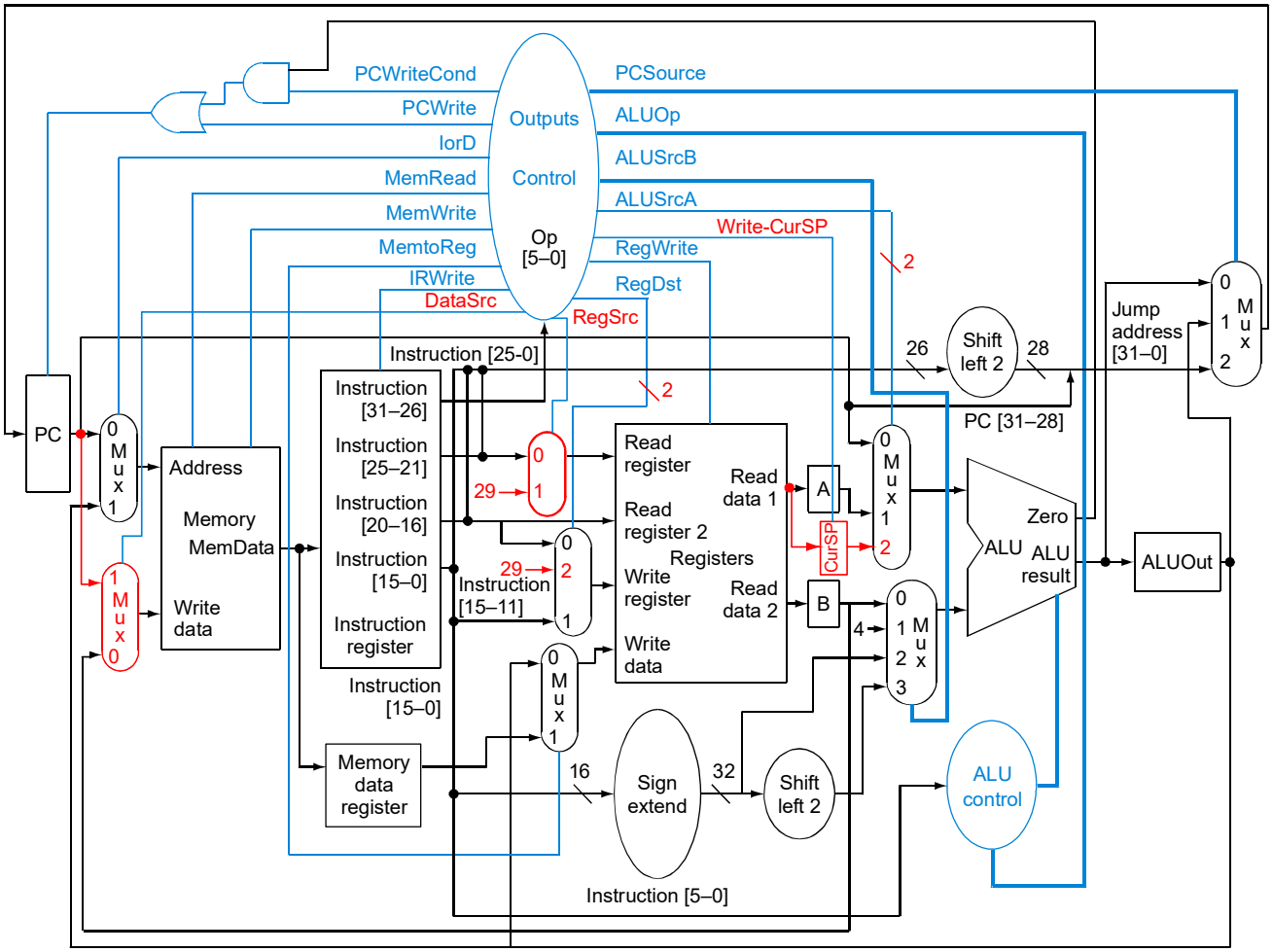
A) A straightforward implementation of the new `jal` instruction would result in an execution time of five cycles for `jal`. However, the problem states that the top priority is to minimize the execution time for `jal`. The execution time for `jal` can be cut to only four cycles by adding a new register, called CurSP (current SP) to the datapath. This register is loaded with the contents of register 29 (the stack pointer) in state 0 (before the instruction is known). In order to do that, a MUX is added to the "Read register 1" port, allowing the constant 29 to be driven onto that port.

The ALUSrcA MUX is expanded to three inputs so that CurSP can be selected as the A input to the ALU.

The RegDst MUX is expanded to three inputs so that the constant 29 can be selected in order to write the updated stack pointer value to register 29.

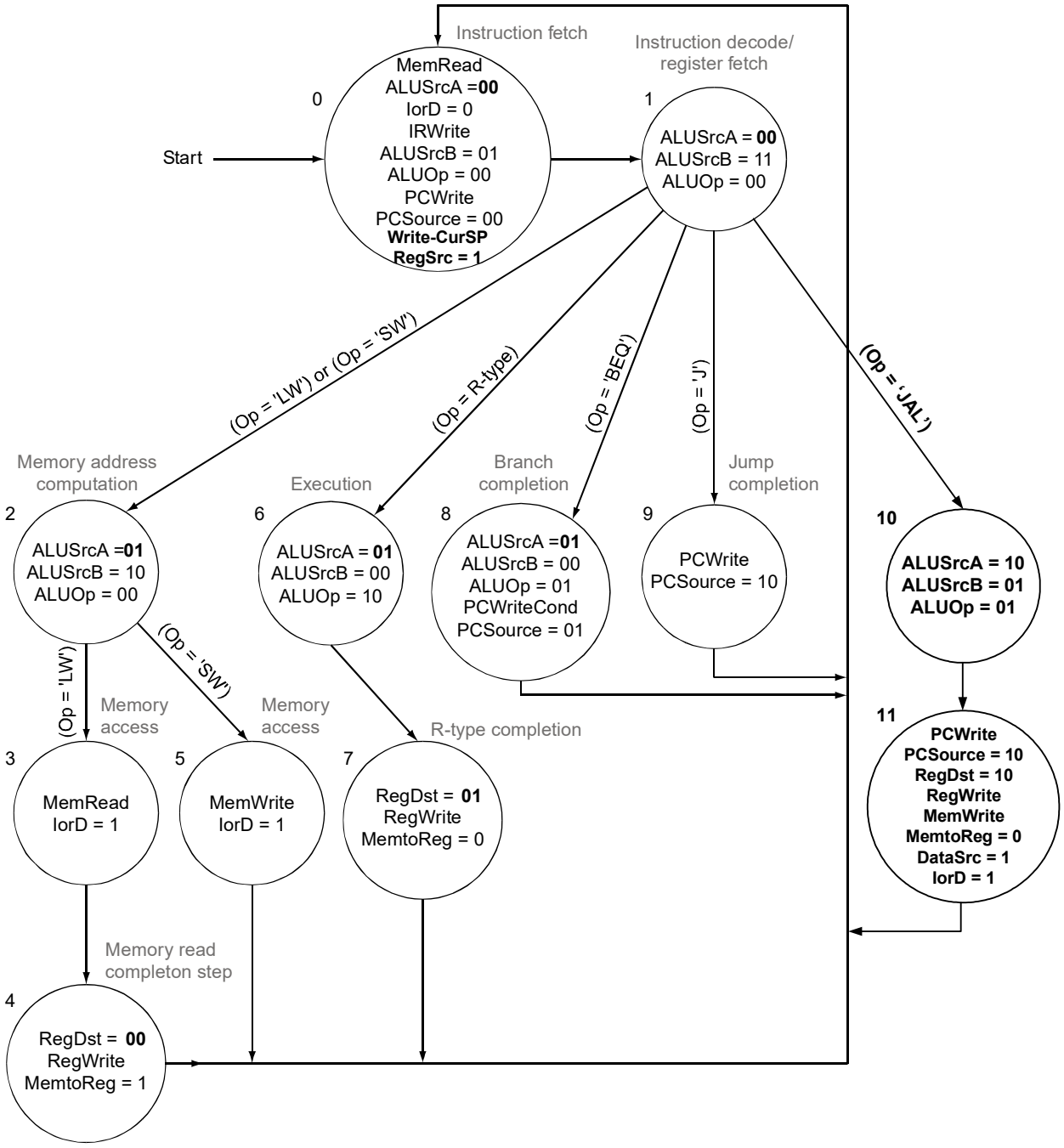
Since the `jal` instruction needs to store the value of the PC in memory (on the stack), a new DataSrc MUX is added to select whether the data to be written is from the B register or the PC.

Note: It is possible to further improve the performance of the `jal` instruction (to a latency of only three cycles), but at a very high additional implementation cost. Specifically, the operation that occurs during the third cycle of the `jal` implementation shown here (ALUOut ← CurSP–4) could be executed during the second cycle if the required resources are added to the datapath. This would require another ALU (actually, just a subtractor), another 32-bit register (ALUOut2), and a new MUX to select between the outputs of ALUOut and ALUOut2. This solution is not shown here due to its high implementation cost.



B) Three new controls are added: 1) RegSrc is used to choose rs or register 29 for read register 1 port, 2) Write-CurSP controls the writing into the CurSP register, and 3) DataSrc is used to select PC as an input to the memory. Two existing controls are modified: 1) ALUSrcA is expanded to 2 bits so CurSP can be selected, and 2) RegDst is now 2 bits to enable the selection of the 29th register.

C)



D) No, a new `return` instruction is not necessary. The return address can be loaded from the stack into a register and the `jr` instruction can be used to actually perform the return. We can use register $1, which is usually reserved for assembler temporaries for this purpose. Note that the `jr` instruction is not

supported by the implementation in Figure 5.28. Hence, while no new `return` instruction is needed, the `jr` instruction defined by the current MIPS ISA must be implemented.

```
lw    $1, 0($29)
addi  $29, $29, 4

jr    $1
```

Problem (13)

We need only to consider delays associated to major components in the datapath (ALU, memory, and register file), since the problem specifies that all other delays are negligible. From the state diagram for the multicycle implementation, we observe that only states 0 and 1 make use of more than one major component. However, in both cases the components can be used in parallel, and the constraint on the cycle length is given by the largest delay. The following table gives the constraints on the cycle length determined by each state.

State	Modules	Cycle length
0	memory, ALU	max (5, 3) = 5ns
1	ALU, reg-file	max (3, 6) = 6ns
2	ALU	3 ns
3	memory	5 ns
4	reg-file	6 ns
5	memory	5 ns
6	ALU	3 ns
7	reg-file	6 ns
8	ALU	3 ns
9	-	0 ns

Thus, we conclude that the minimum cycle time must be 6 ns, which determines the maximum clock frequency:

$$\begin{aligned} \text{clock freq.} &= \#\text{cycles / sec} \\ &= 1 / \text{cycle length} \\ &= 1 / (6 \times 10^{-9}) \\ &= 167 \text{ MHz} \end{aligned}$$