CDA 3103: Study Set 6

DATAPATH, CONTROL SIGNALS, PIPELINING

Review: Datapath

The MIPS ISA is called a RISC (reduced instruction set computer). We only developed the datapath for some of the MIPS instructions we know.

RISC Instruction Set:

R-Types: Add, Subtract, And, Or, Slt

I-Types: Add Immediate, And Immediate, Or Immediate, Load Word, Store Word, Branch if Equal

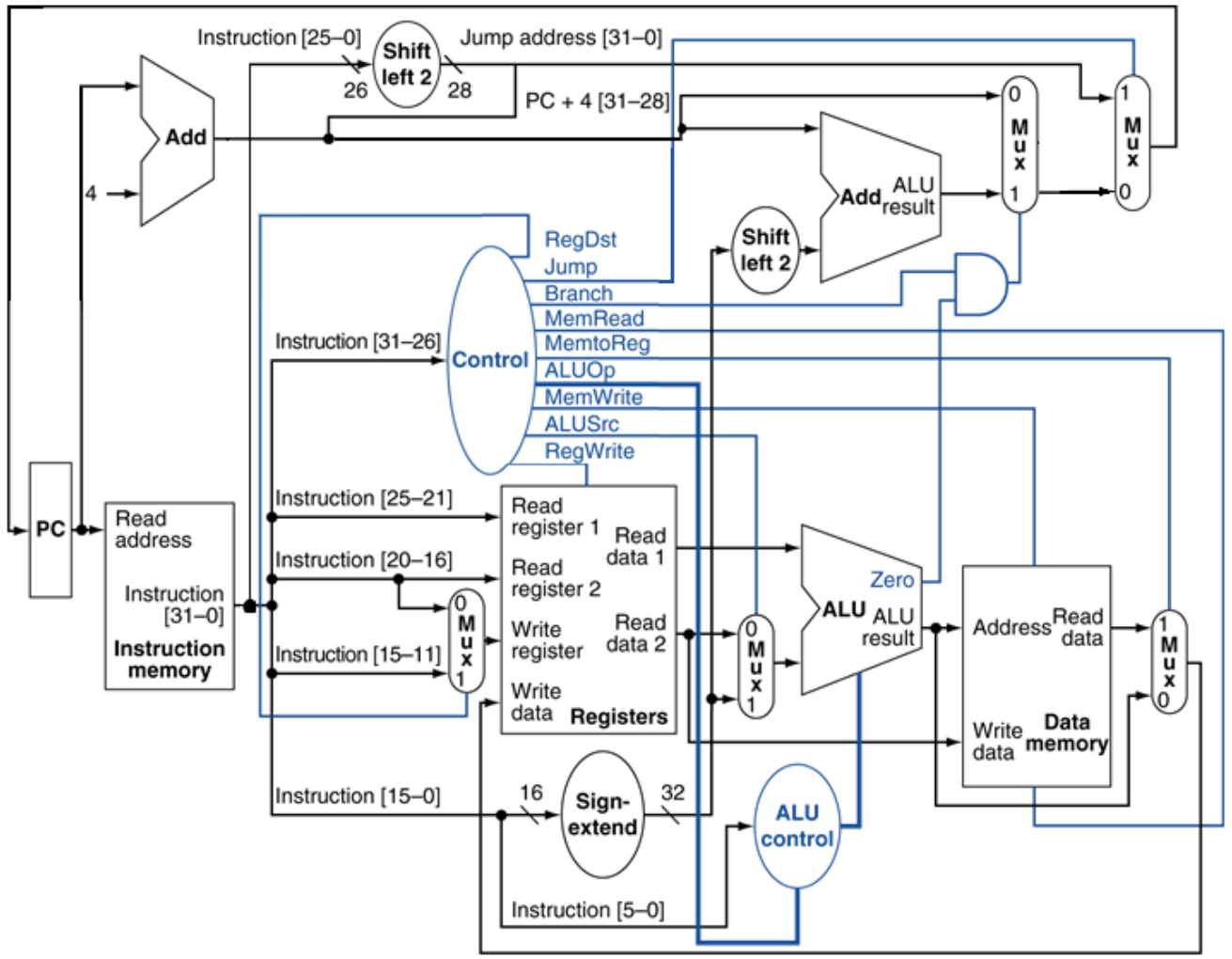
J-Types: Jump

Review: Datapath

In the single cycle datapath each instruction is fetched and processed in one clock cycle.

The Program Counter (PC) is a register that contains our current address in instruction memory. This is the location of our next instruction.

The instruction comes out of memory on a 32-bit bus. We split the wires of this bus to several different locations. The six most significant bits [bits 31-26] for all instructions types are the opcode, we send these bits to the control unit.

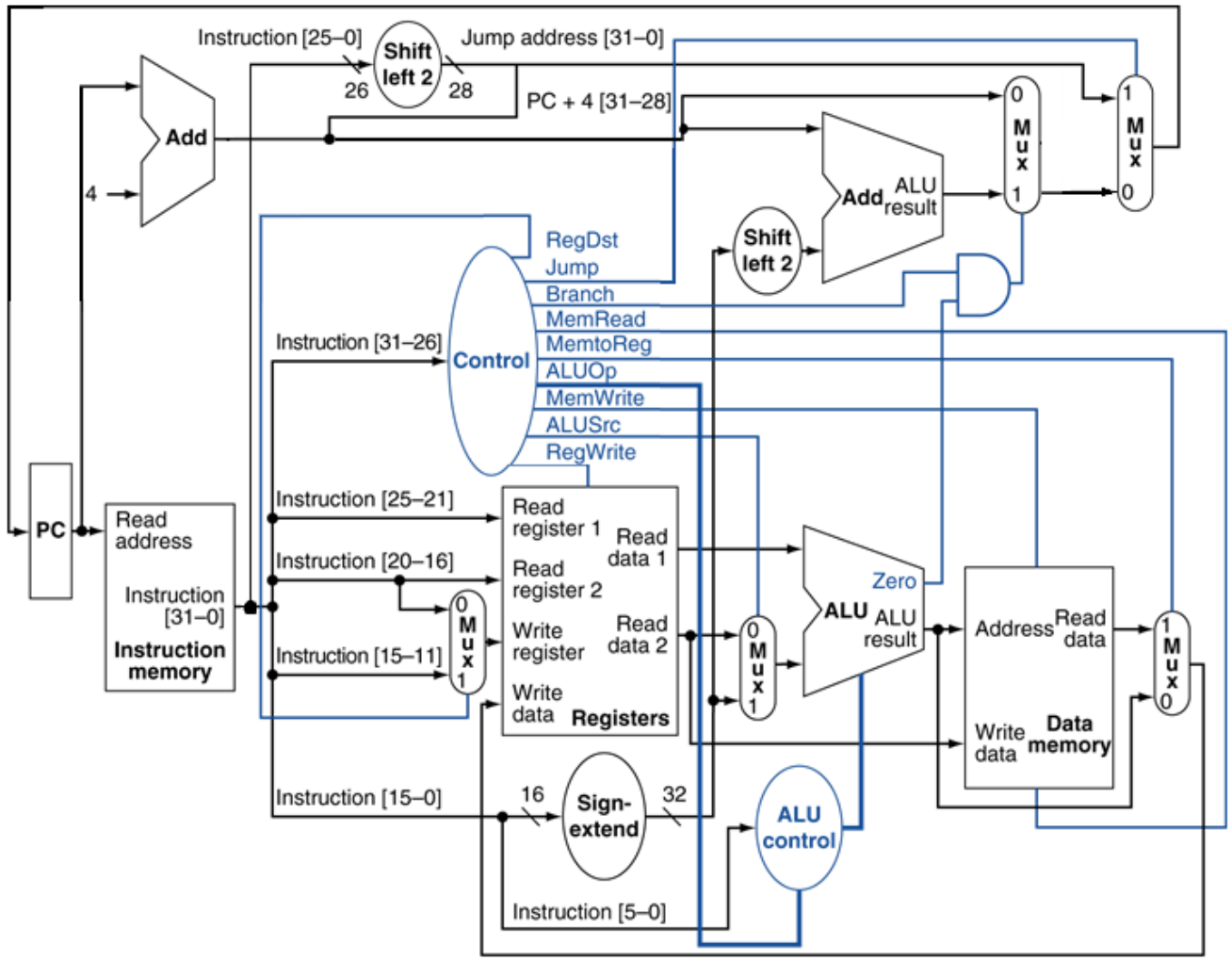


Review: Datapath

The next five bits [bits 25-21] are sent to the register file to identify the first register we will be reading from. We always do this, even if the instruction does not need to read any registers.

The same is true for the following five bits [bits 20-16]. In most of our instructions (add, sub, and, or, slt, sw, and beq) we will need the data from these two registers.

For a lw instruction, these five bits specify the destination. So we also send these five bits through a multiplexor to the write register.

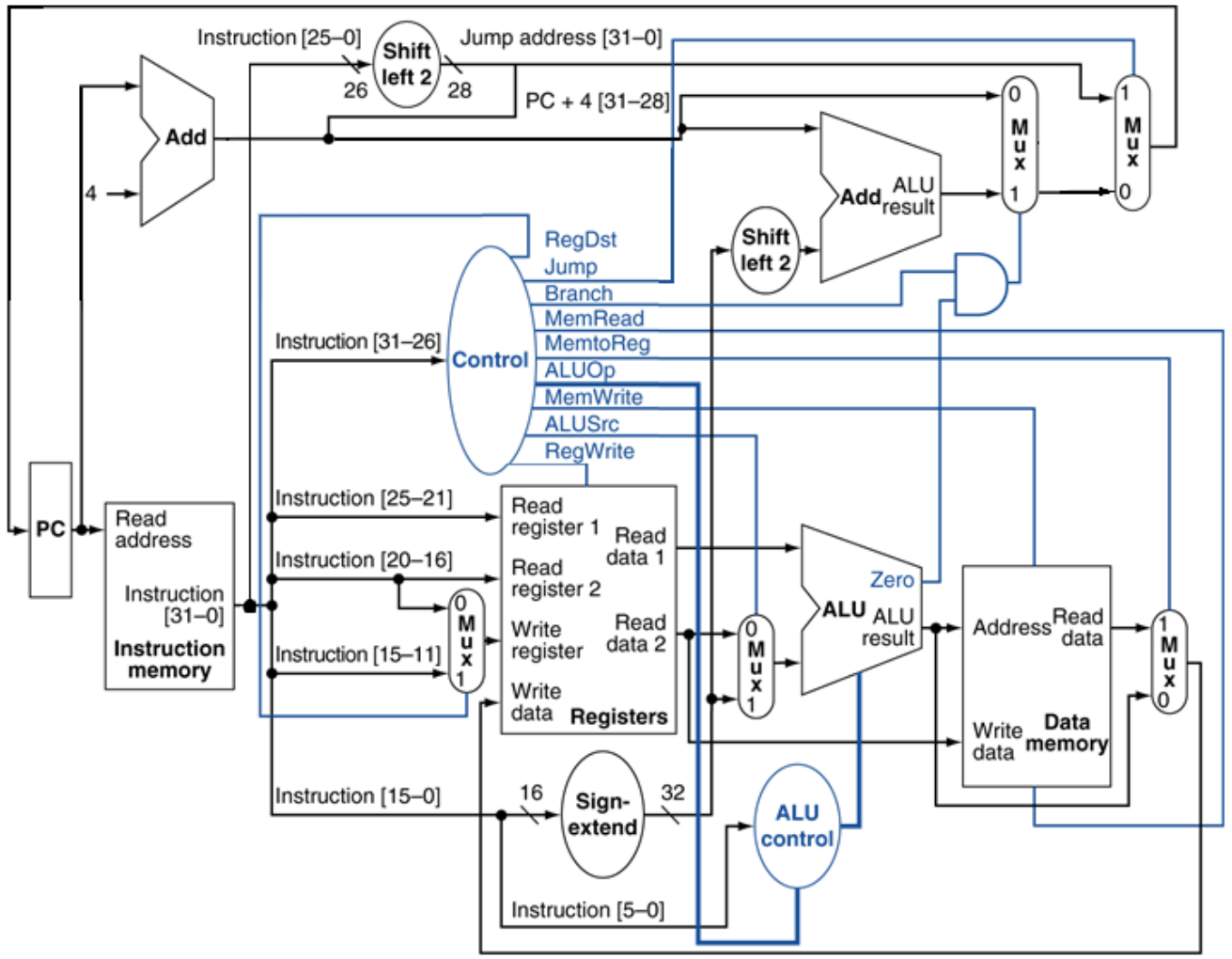


Review: Datapath

The next five bits [bits 15-11] are sent to the write register multiplexor. In R-Type instructions, this will be the register that receives a new value.

The 16 least significant bits [bits 15-0] are also sent to the sign extension unit to form a sign-extended immediate when its needed (addi, lw, sw, and beq).

The 6 least significant bits [bits 5-0] are also sent to the ALU control. These will only be used for decoding R-Type instructions.



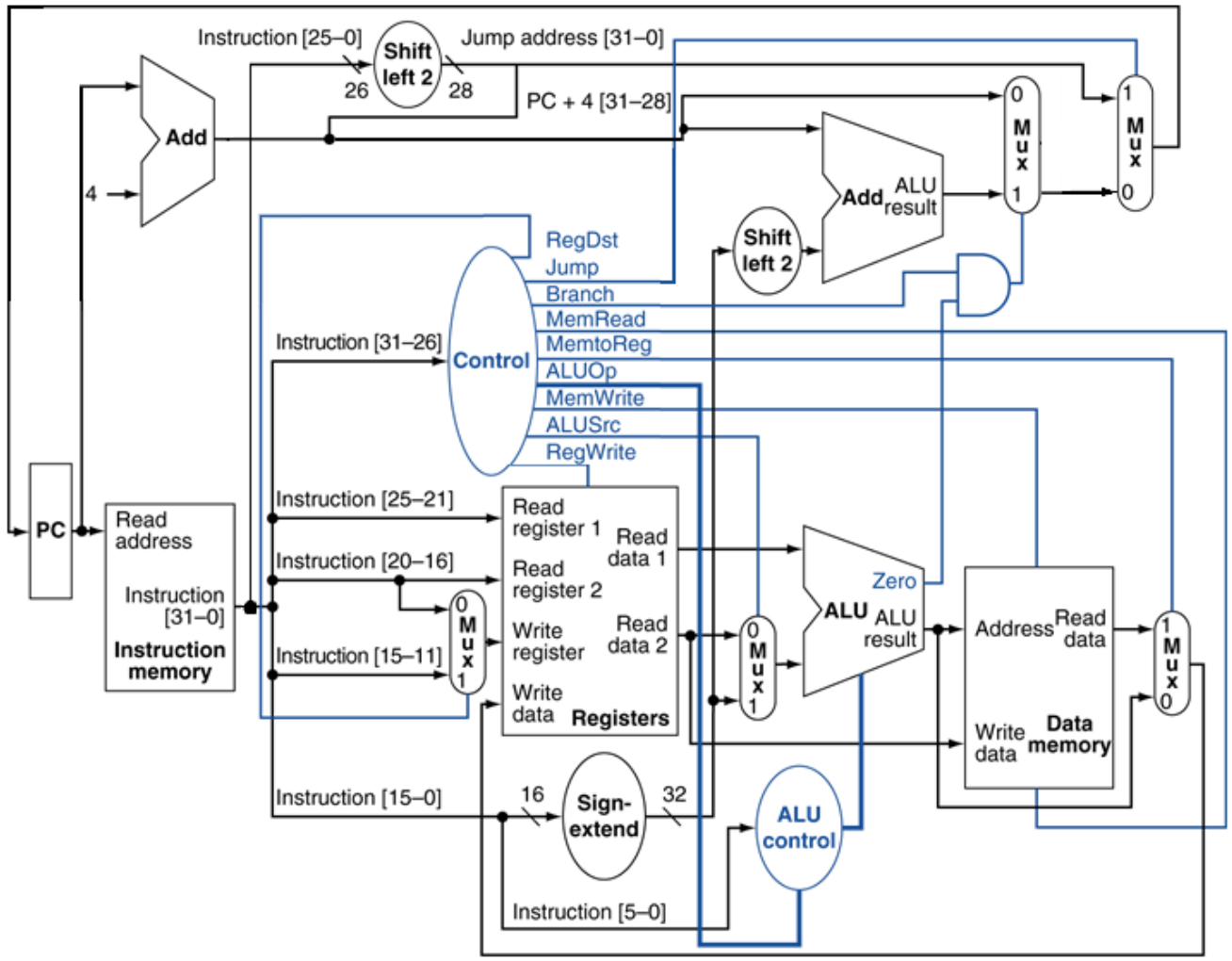
Review: Datapath

At this point, we add 4 to the PC to get set up for the next instruction.

If our current instruction is a branch or a jump, this value might change a second time.

In a jump, the 26 least significant bits [bits 25-0] will be shifted twice to the left. This is the same as multiplying by 4 and results in a 28 bit jump address.

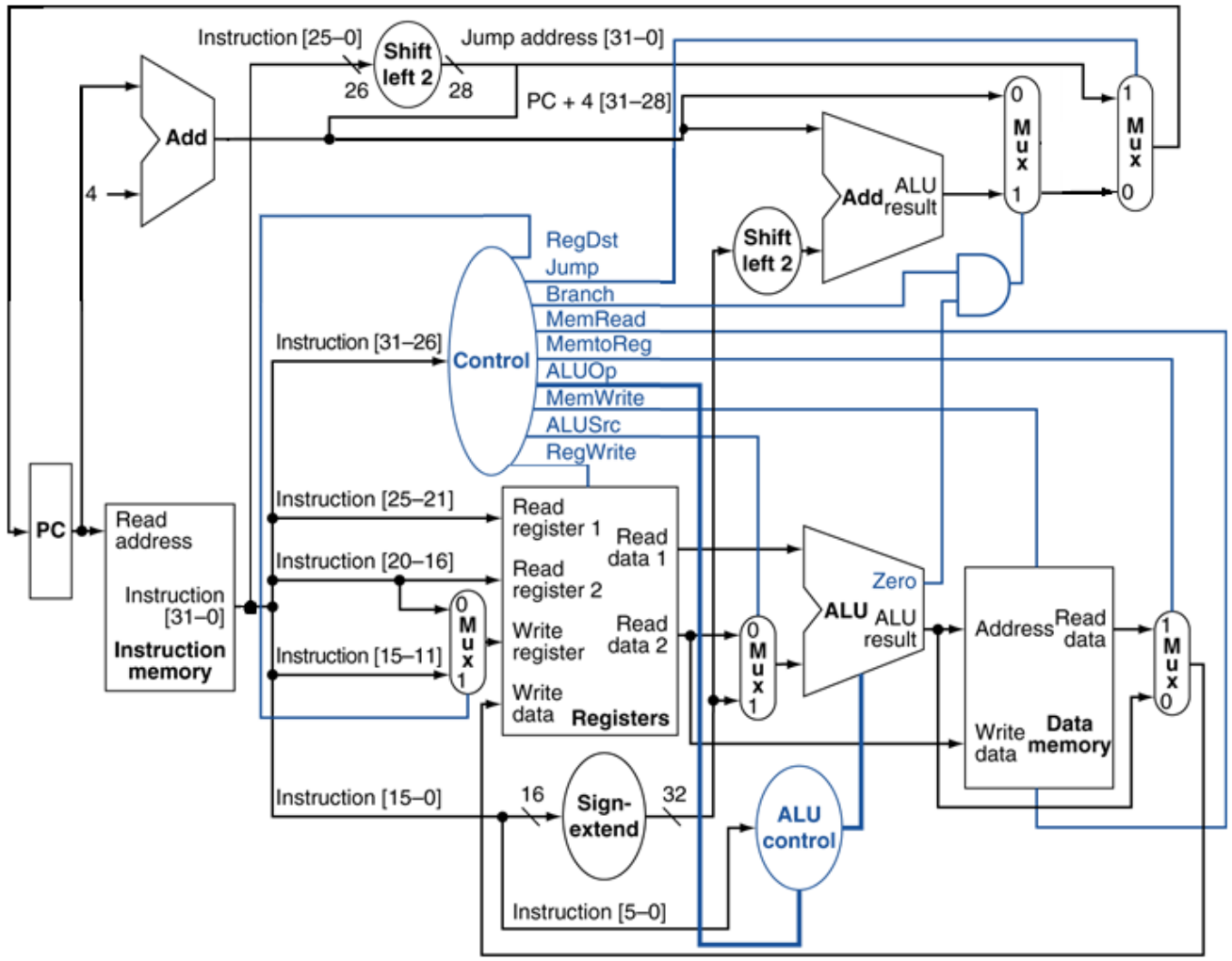
We then copy over the four most significant bits from the PC [bits 31-28] to form a complete new PC.



Review: Datapath

The register file will produce two values along the read data 1 and read data 2 buses. Read data 1 is always sent to the Arithmetic Logic Unit (ALU).

The second input to the ALU depends on the instruction. R-Type instructions and BEQ use both register values. I-Type instructions other than BEQ use only one register and the other input to the ALU is the immediate value. We use the multiplexor between the register file and the ALU to choose between these two values.

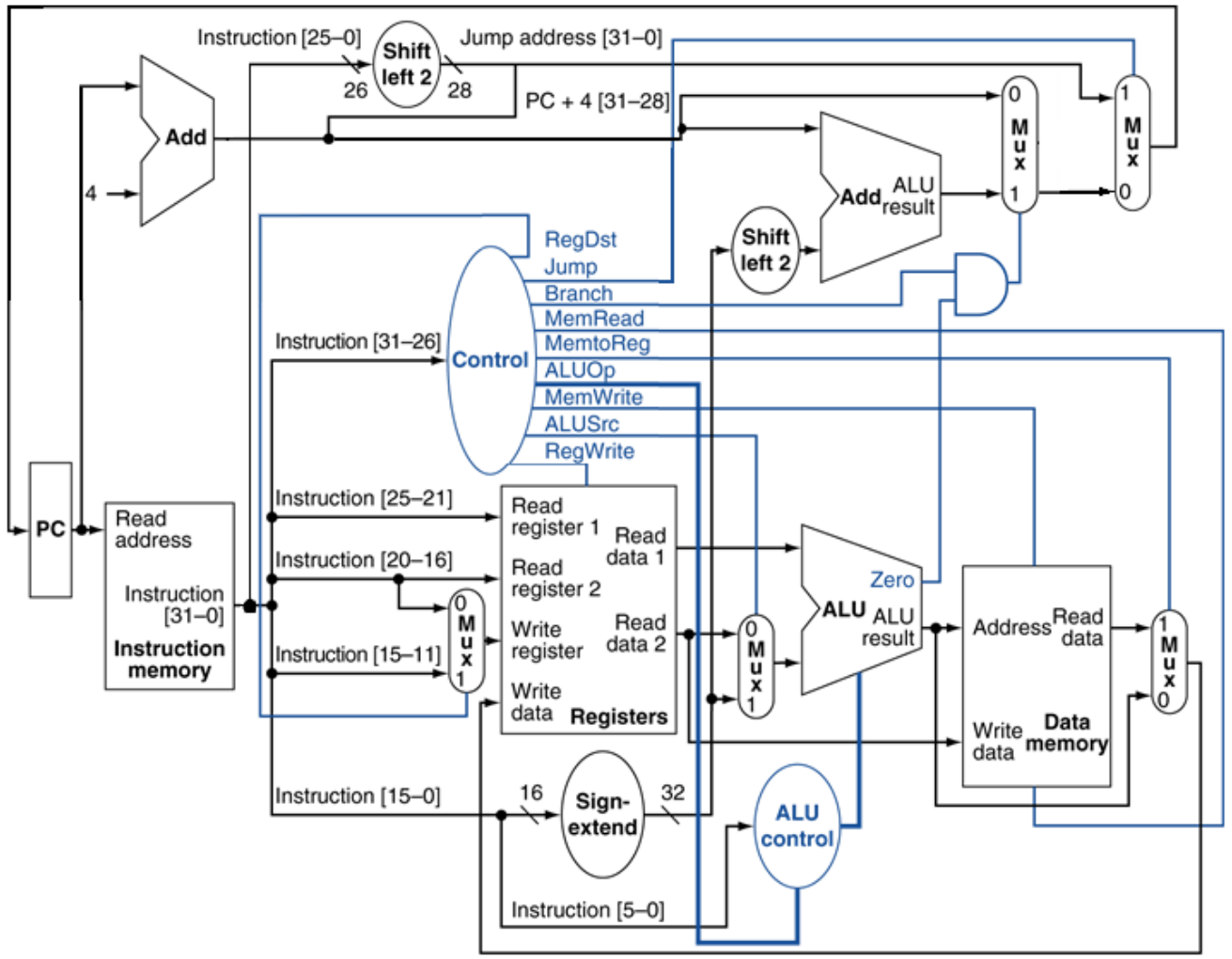


Review: Datapath

The ALU will produce a result based on the inputs provided and the ALU operation signal from the ALU control unit.

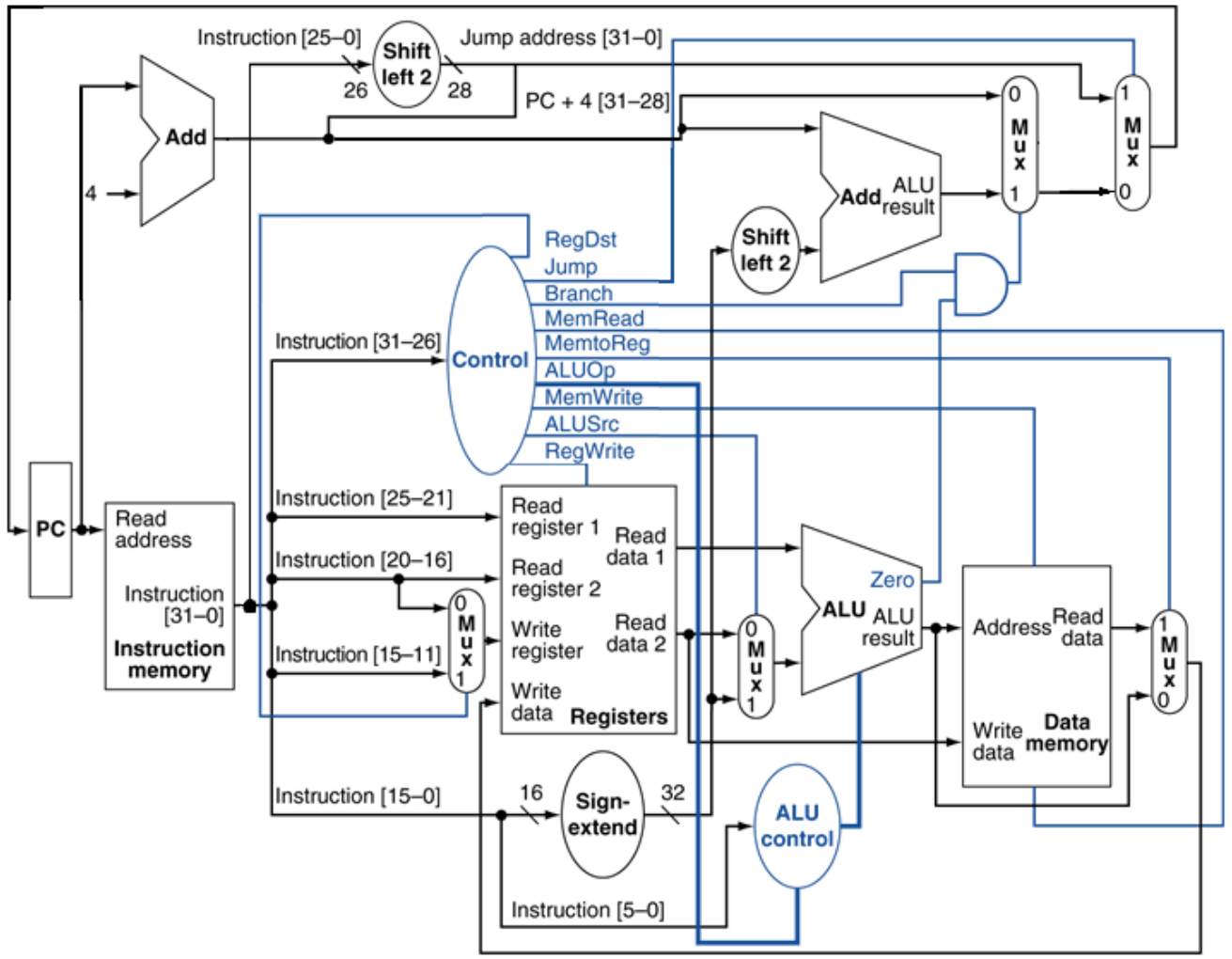
For most instructions, we send this result “around” the data memory and back to the register file.

For lw and sw, this result is our memory access address. In a sw, we take the value from the read data 2 bus and store into data memory.



Review: Datapath

In a lw instruction, we take a value out of memory and send it back to the register. This creates two possible values for write data. We use a multiplexor to choose between ALU result and read data.

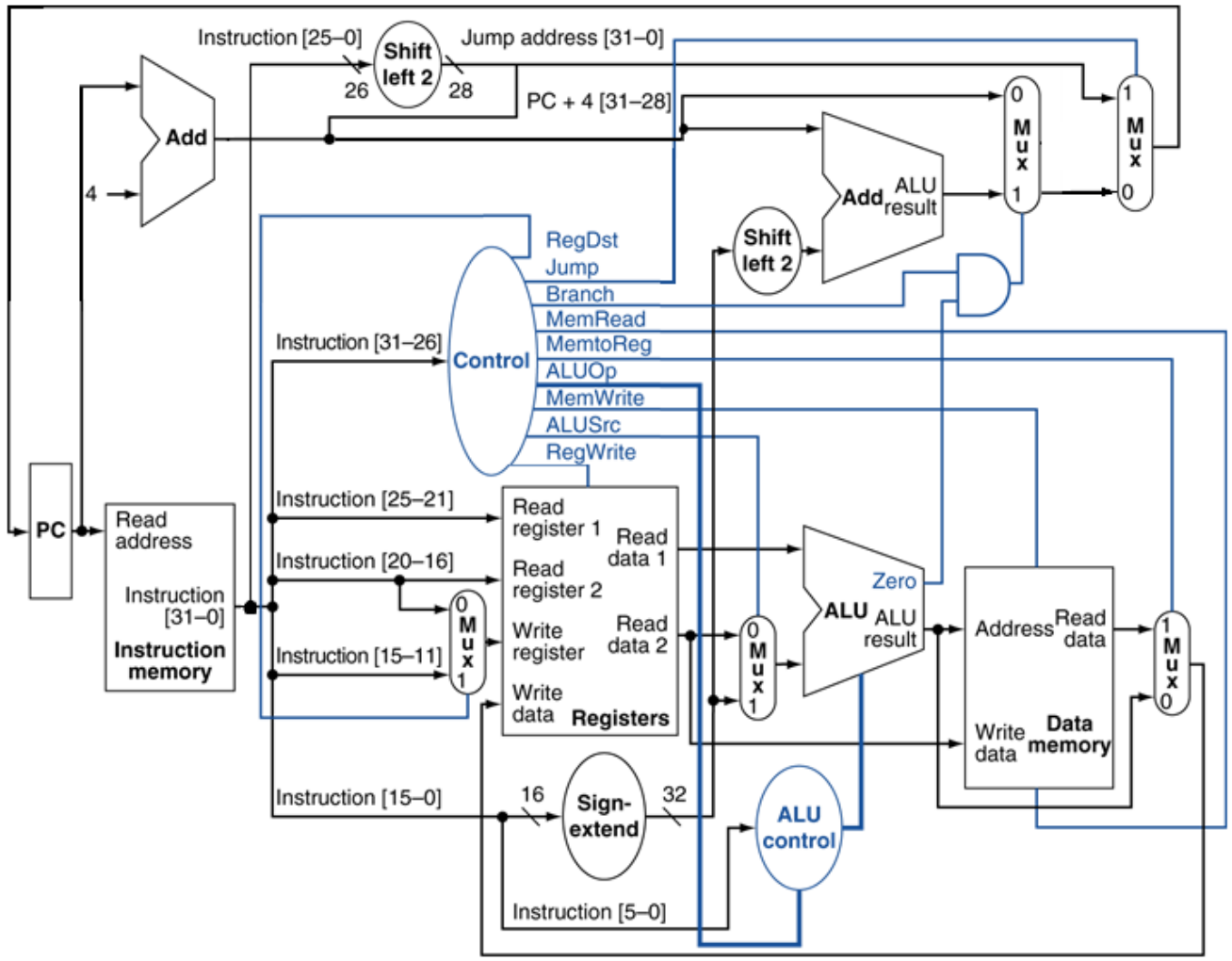


Review: Datapath

In a BEQ instruction, we calculate a branch target address by shifting our sign-extended immediate to the left twice. This is the same as multiplying it by four.

We then add this to the new PC value. If the two registers we are comparing actually are equal, then the Zero signal from the ALU will be asserted.

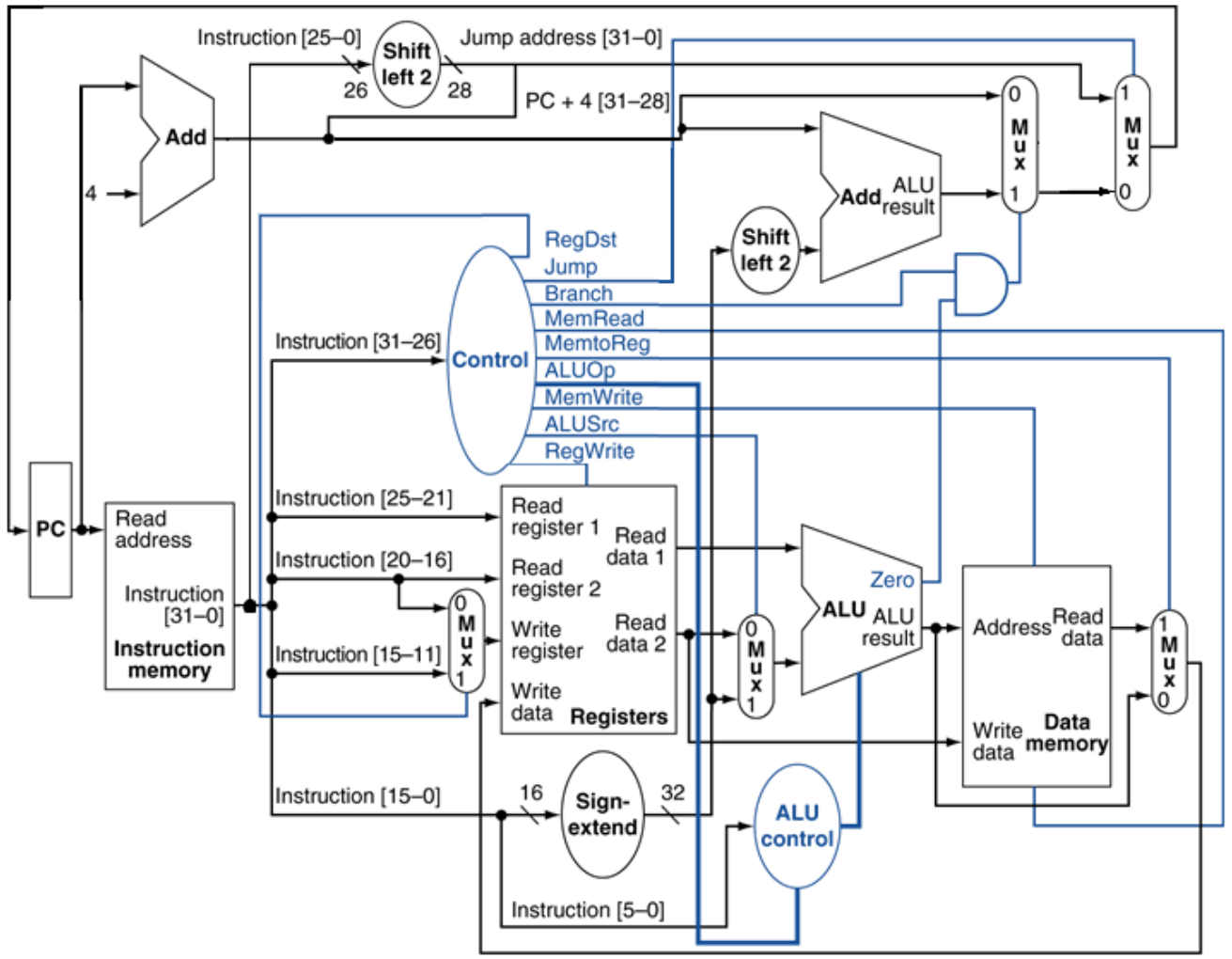
If both Branch and Zero are true, we select this branch target address to be our new Program Counter.



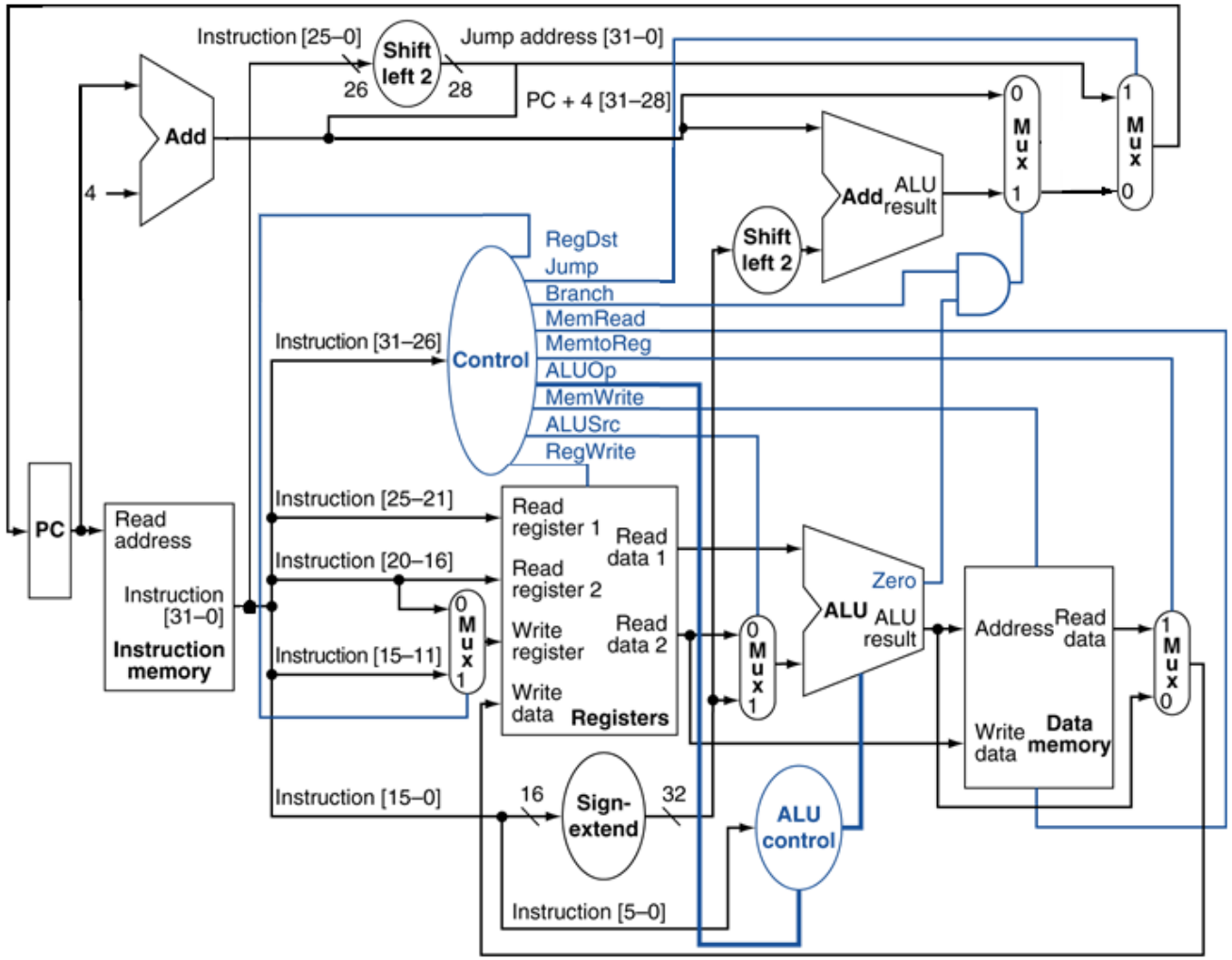
Review: Datapath

All of the decisions to be made on the datapath are controlled by multiplexors and control signals (shown in blue).

The Control Unit sets each of these signals based on the opcode in the instruction. These signals give the processor permission to read from data memory, write to data memory, and write to the register file. They also specify when to use read data 2 versus the sign extended immediate, which register to write to, when to use memory data versus the ALU result. They also determine where the next PC will come from.



Review: Control Signals



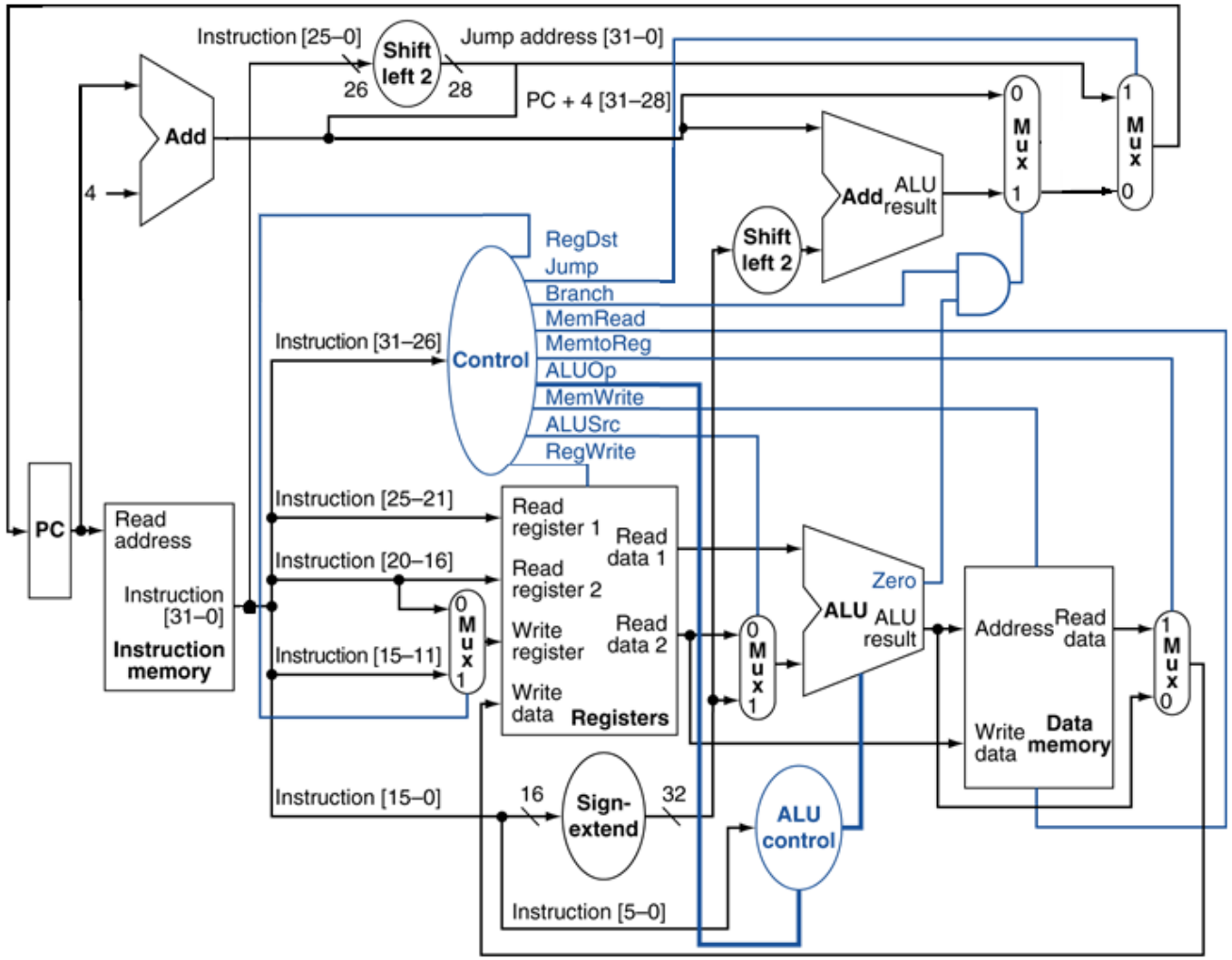
The Control Unit sets all the control signals based on the Opcode.

RegDst chooses which set of bits will determine the write register.

If RegDst is 0 (deasserted) we use bits 25-21 to determine the write register. We want to do this for I-Type instructions that write to the register file (addi, andi, ori, and lw).

If RegDst is 1 (asserted) we use bits 15-11 to determine the write register. We want to do this for all R-Type instructions.

Review: Control Signals



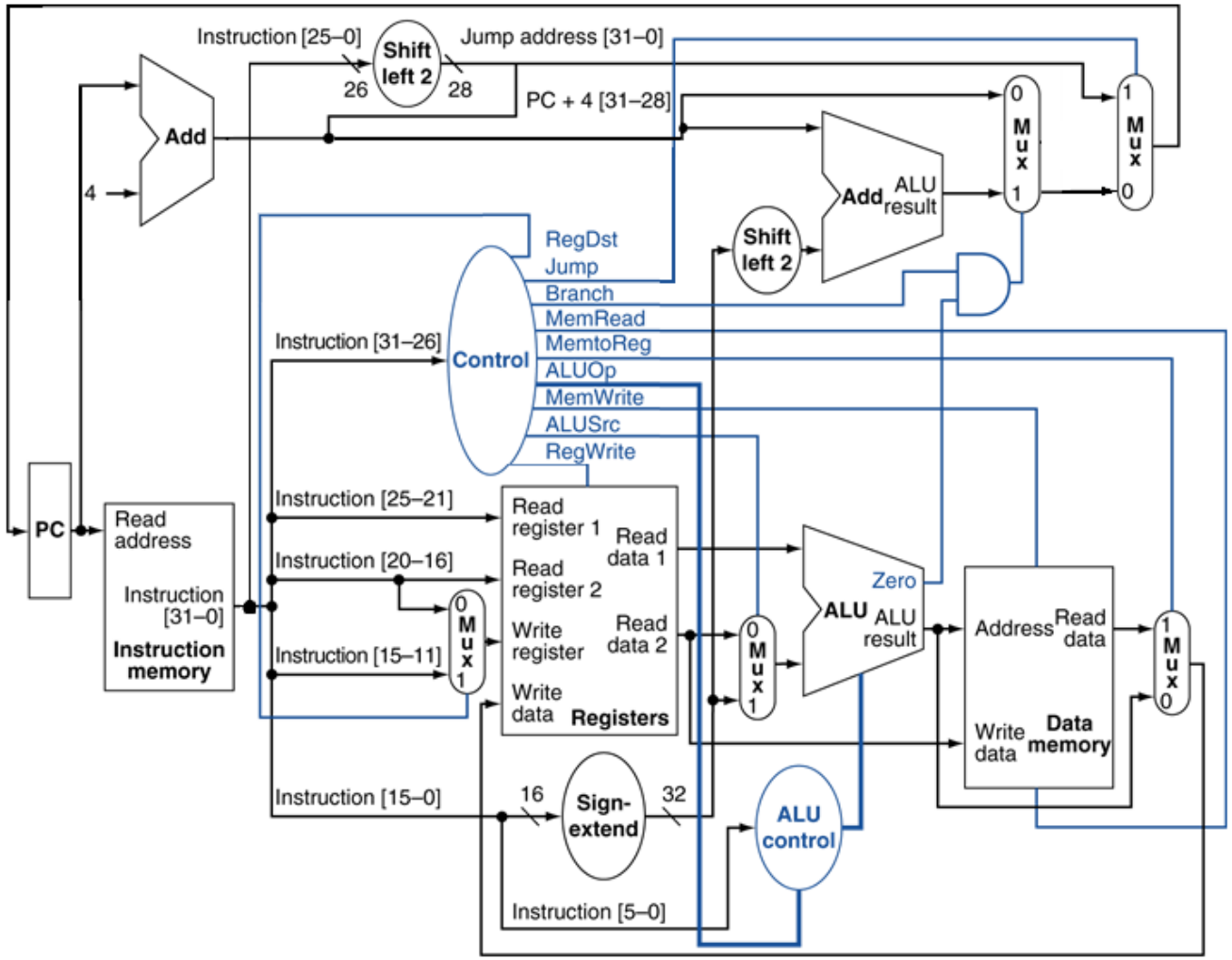
Jump determines our next Program Counter.

We want to assert Jump whenever our current instruction is a jump. Jump should be deasserted for all other instructions.

Branch is also used to determined the next Program counter.

We want to assert Branch whenever our current instruction is BEQ. Branch should be deasserted for all other instructions.

Review: Control Signals



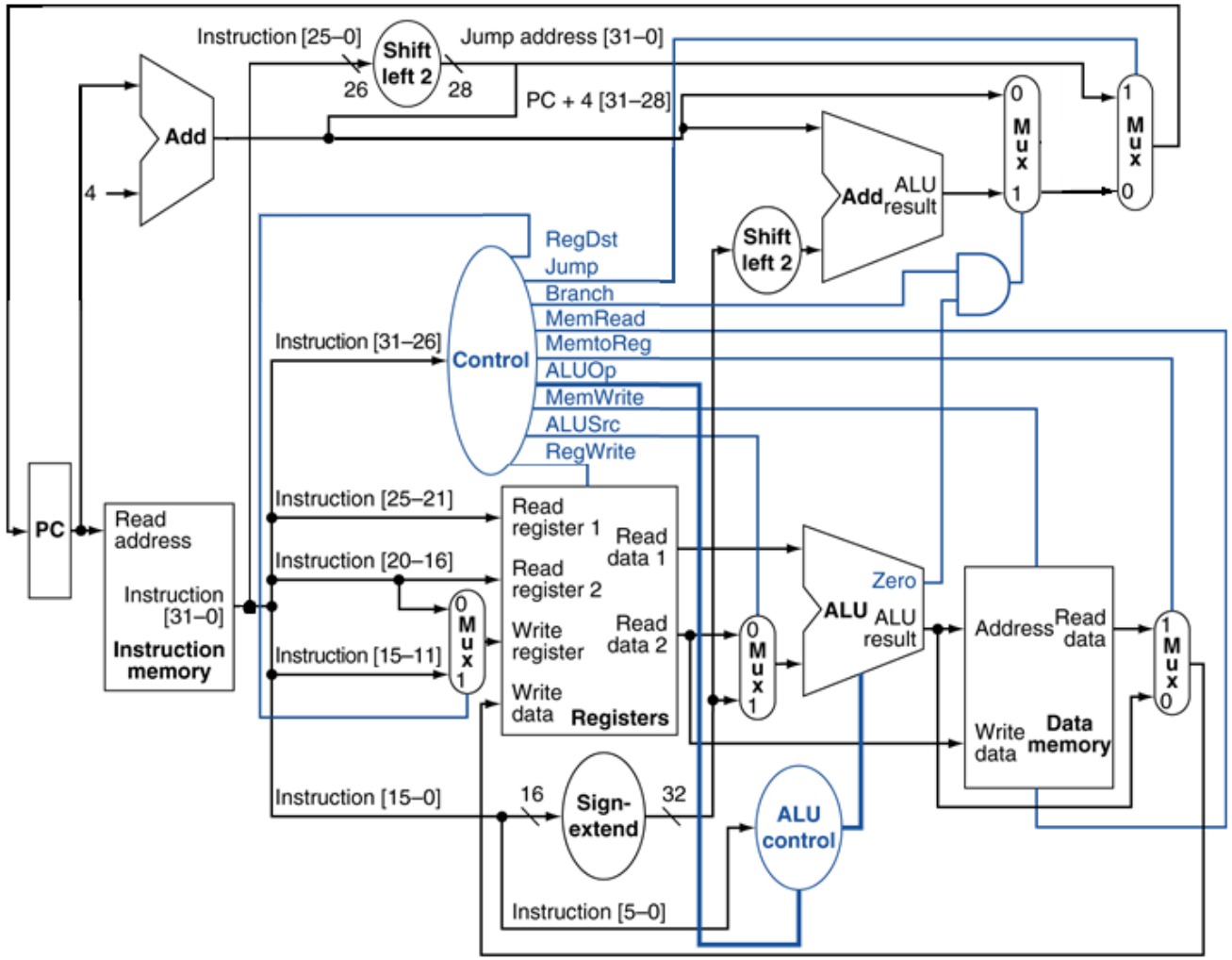
MemRead gives the processor permission to read from Data Memory.

We want to assert MemRead whenever the current instruction is a load word. It should be desasserted for all other instructions.

MemWrite gives the processor permission to write to Data Memory.

We want to assert MemWrite whenever the current instruction is a store word. It should be desasserted for all other instructions.

Review: Control Signals

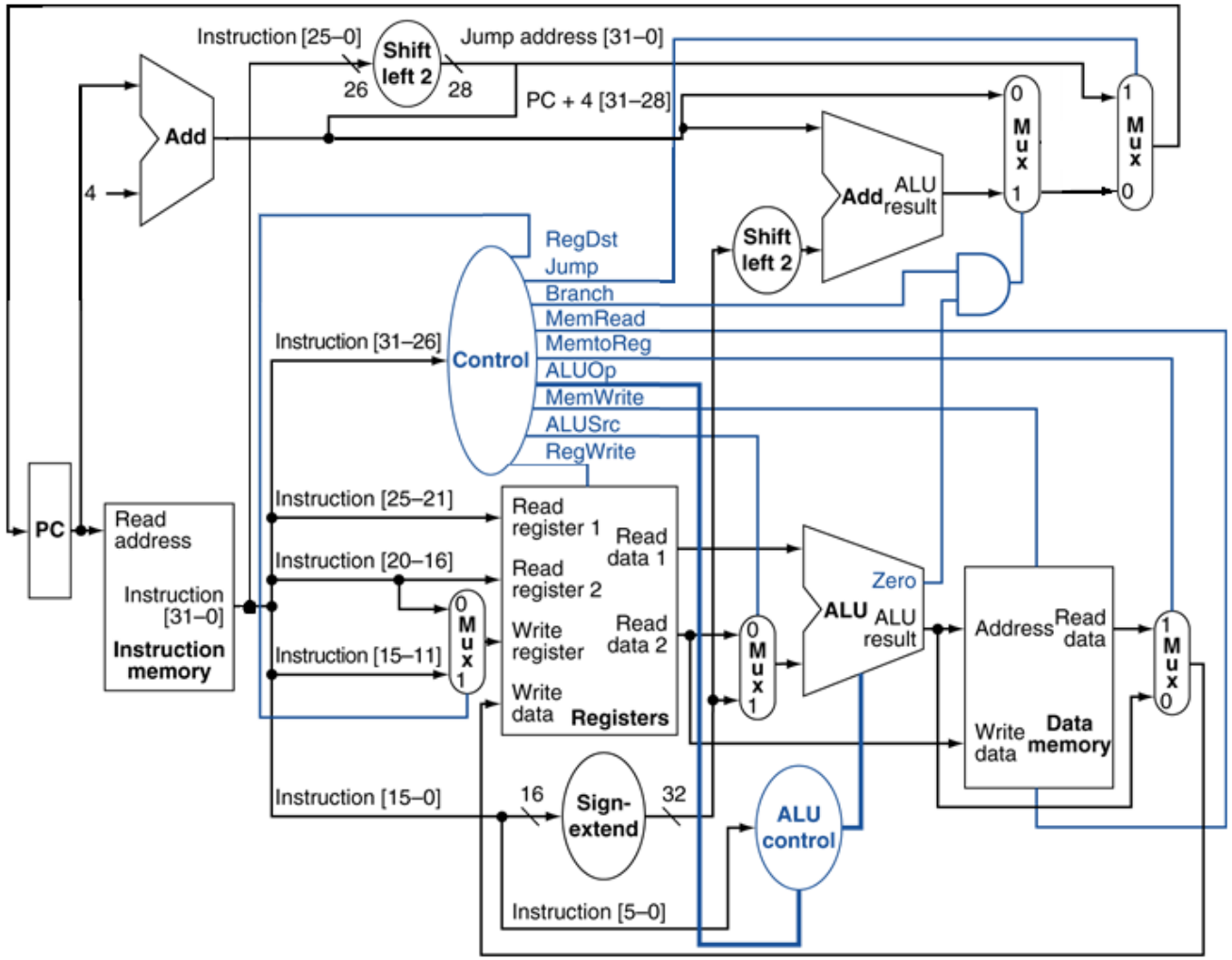


MemtoReg determines which result we are going to send back to the register file.

When MemtoReg is set to 0 (deasserted), the ALU result from the Arithmetic Logic Unit will be sent to the write data input of the register file. We want to do this for all R-Type instructions and any I-Type instructions that write to the register file.

When MemtoReg is asserted, the value from data memory will be sent to the register file. We want to do this if the instruction is a load word.

Review: Control Signals

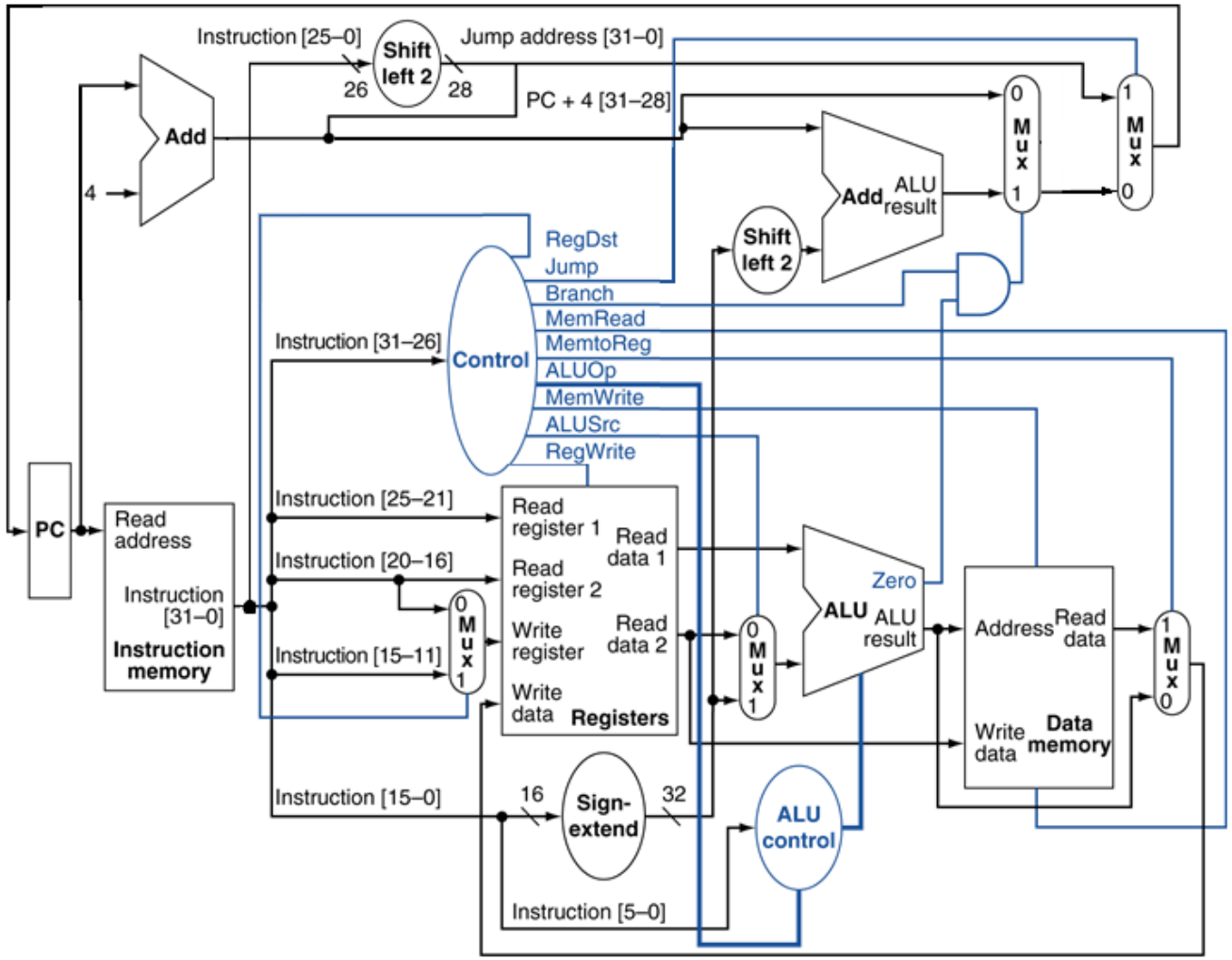


ALUSrc stands for ALU source. This determines the second input for the Arithmetic Logic Unit. It has to choose between read data 2 and the sign-extended immediate value.

If ALUSrc is set to 0 (deasserted) then we will choose read data 2. We want to do this for all R-Types and BEQ.

If ALUSrc is set to 1 (asserted) then we will choose the sign-extended immediate value. We want to do this for the remaining I-Types (addi, andi, ori, lw, and sw).

Review: Control Signals



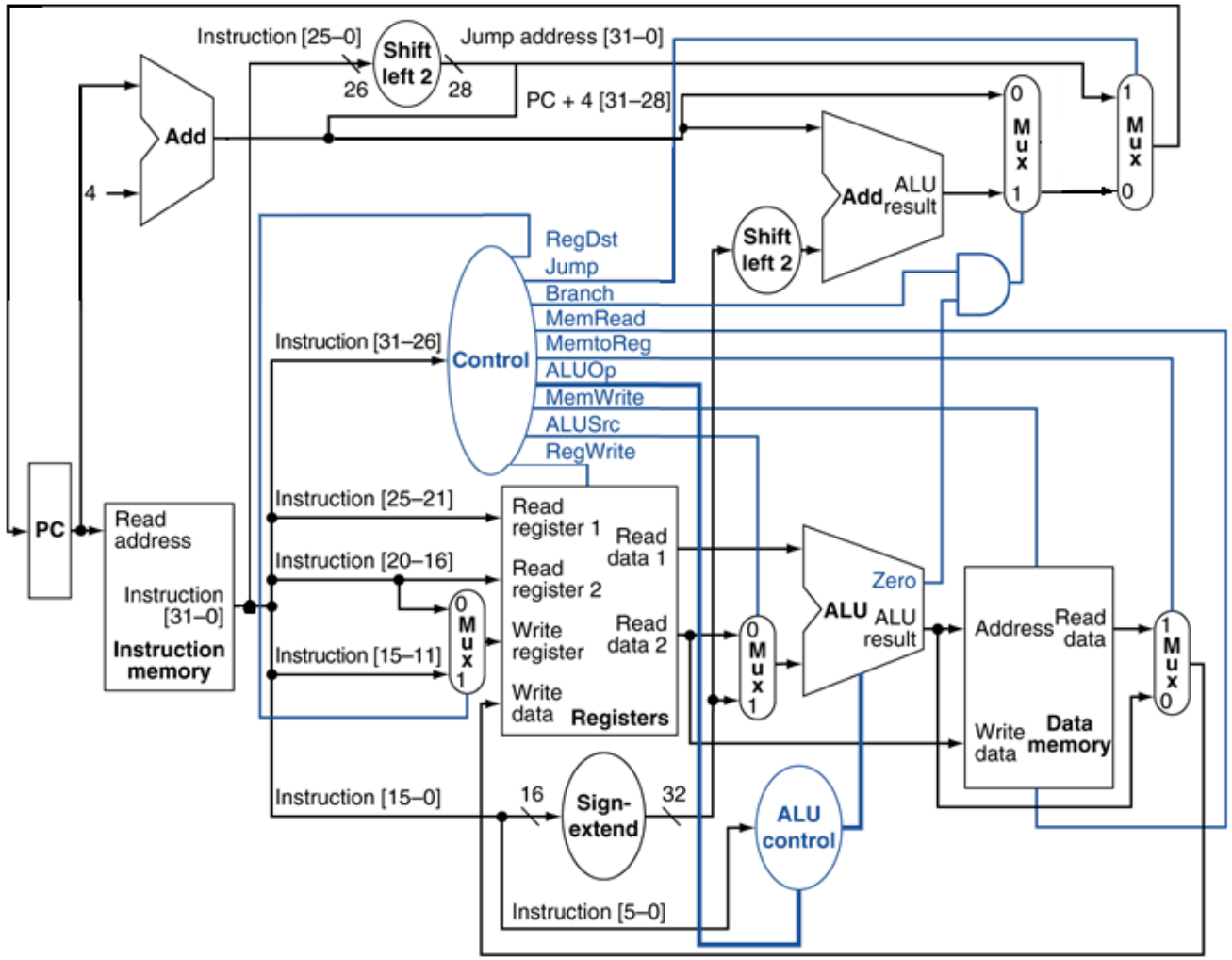
RegWrite gives the processor permission to write to the register file.

We want to assert this control signal whenever we need to write to the register file:

- All R-Types
- Add Immediate
- And Immediate
- Or Immediate
- Load Word

We want to deassert this signal for all other instructions.

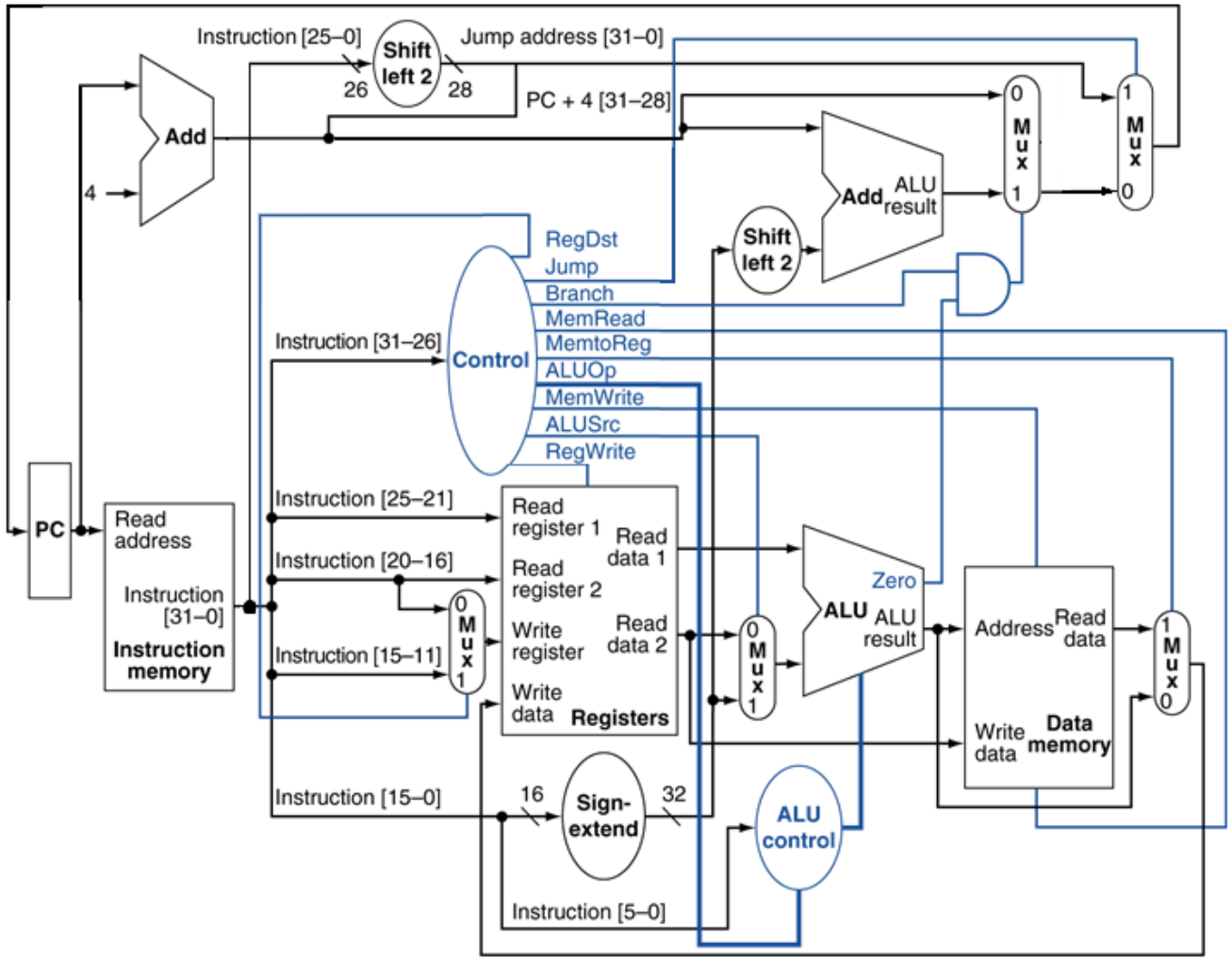
Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

The first control signal is RegDst. Our instruction – addi – is an I-Type instruction with the following format:

Partial Credit 1:

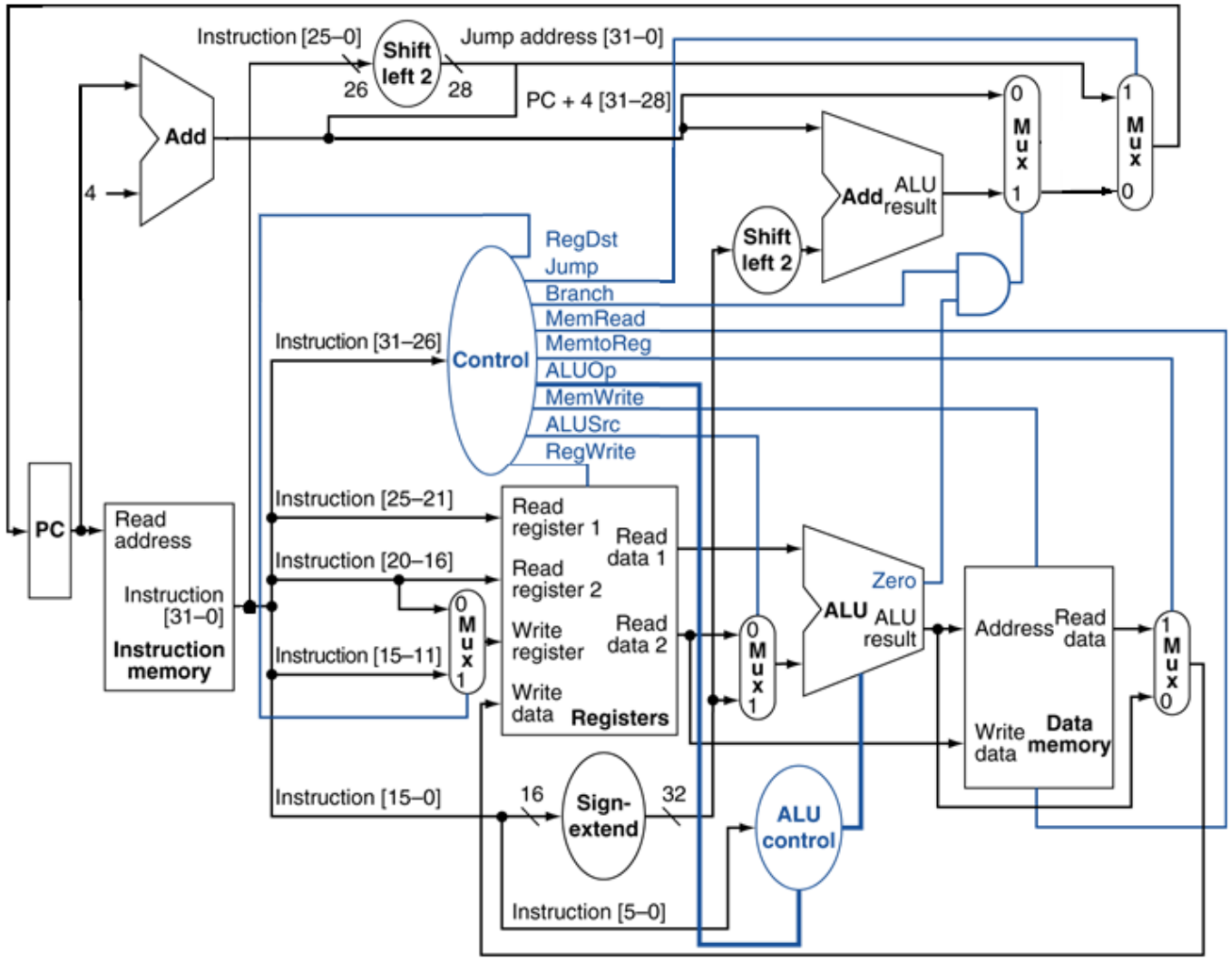
op	rs	rt	constant
6 bits	5 bits	5 bits	16 bits

This tells us the destination register is specified by bits 20-16. So we should set RegDst to 0 to let bits 20-16 specify the write register.

RegDst = 0.

Solution 1:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

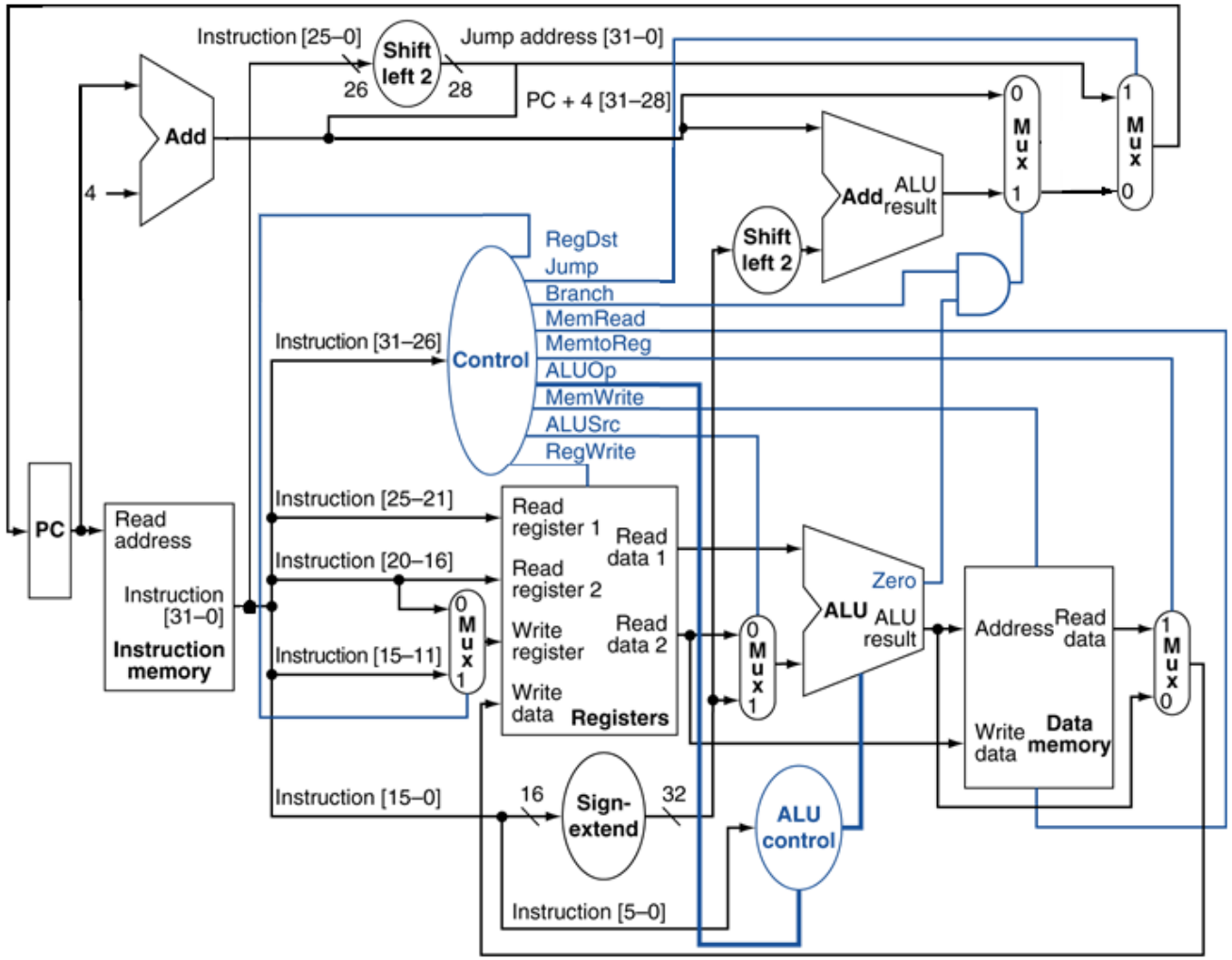
The next control signal is Jump. Since our instruction – addi – is not a jump instruction, this signal should be set to 0.

Partial Credit 2:

Jump = 0.

Solution 2:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

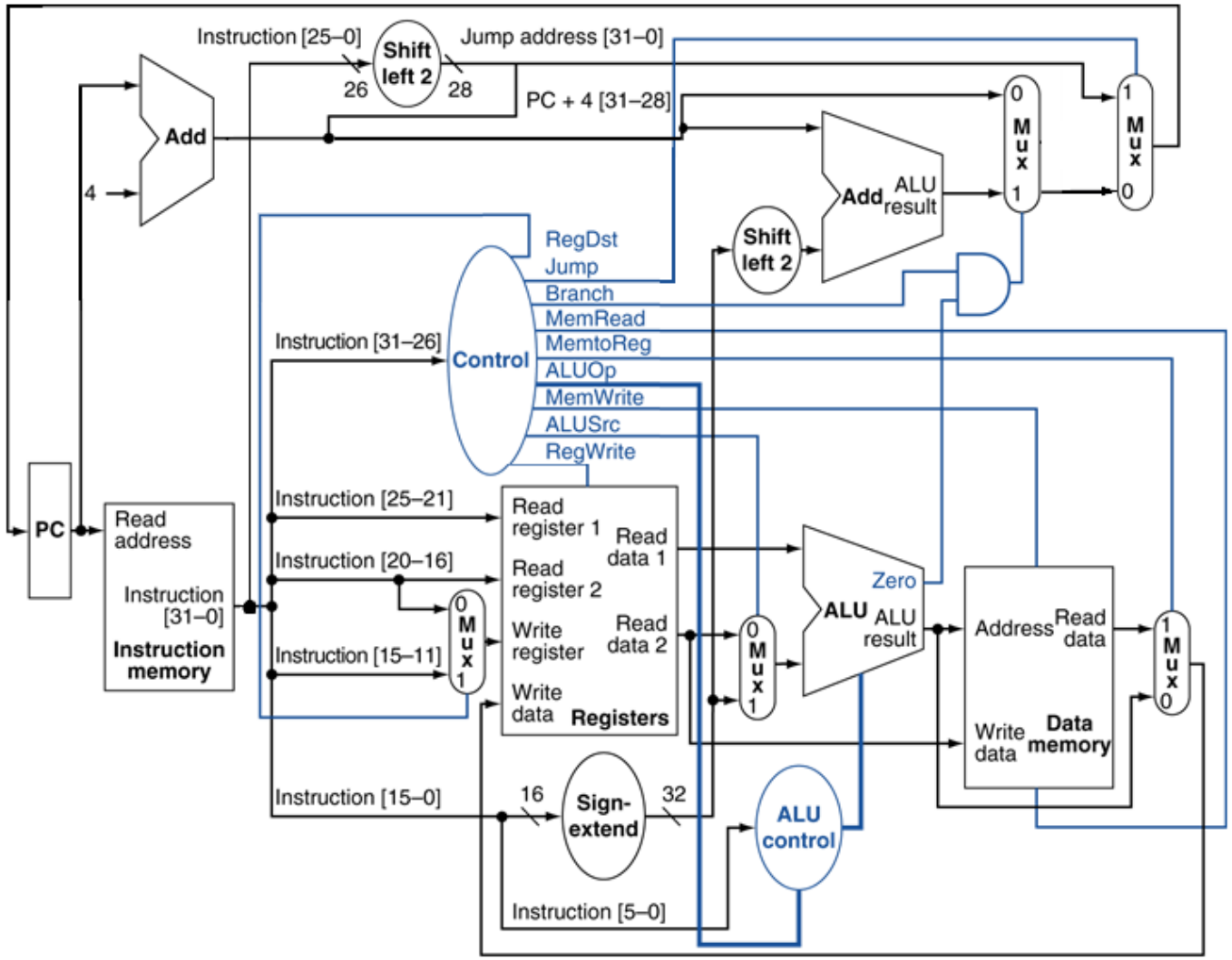
The next control signal is Branch. Since our instruction – addi – is not a branch instruction, this signal should be set to 0.

Partial Credit 3:

Branch = 0.

Solution 3:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

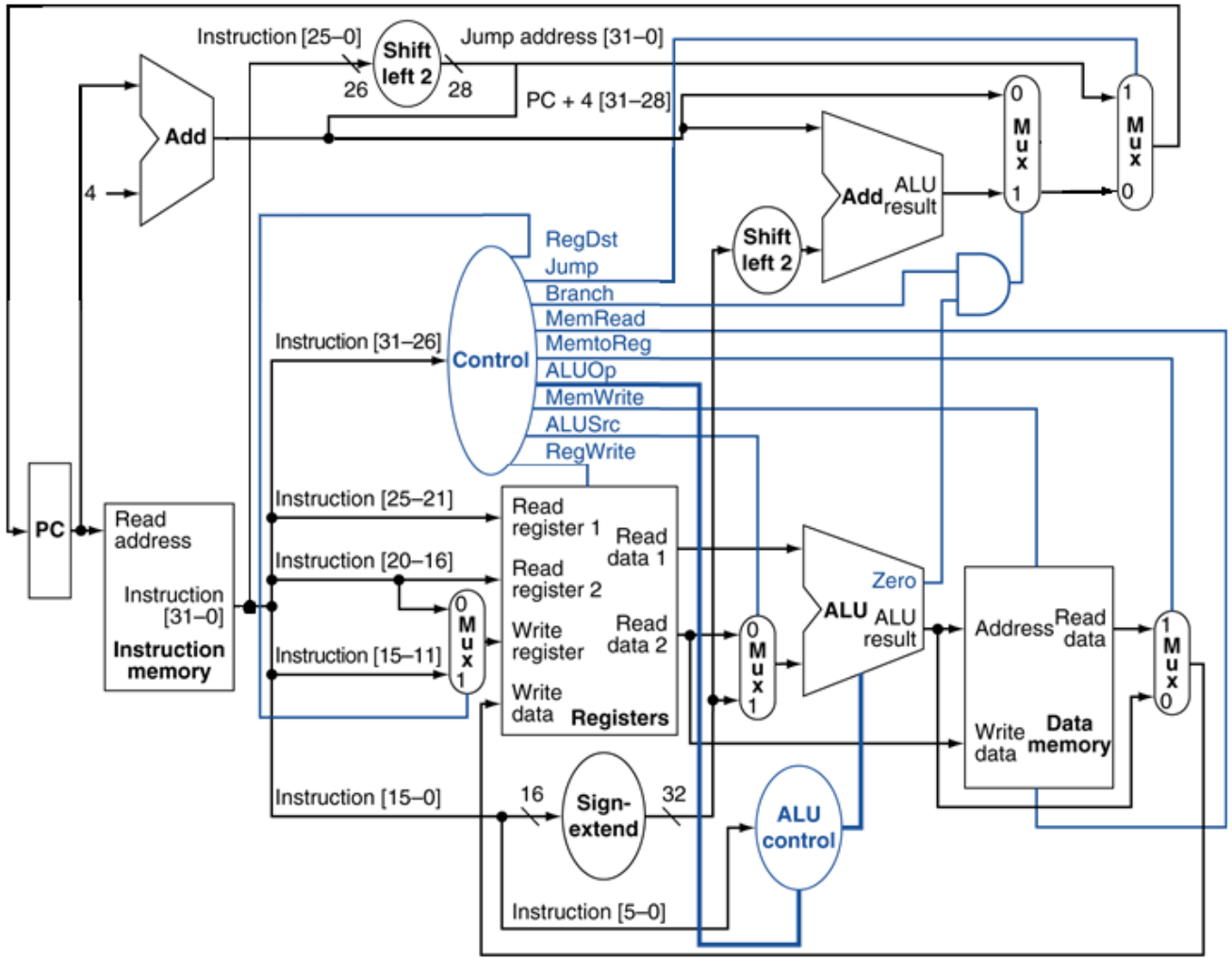
The next control signal is MemRead. Since our instruction – addi – is not a load word instruction, this signal should be set to 0.

Partial Credit 4:

MemRead = 0.

Solution 4:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

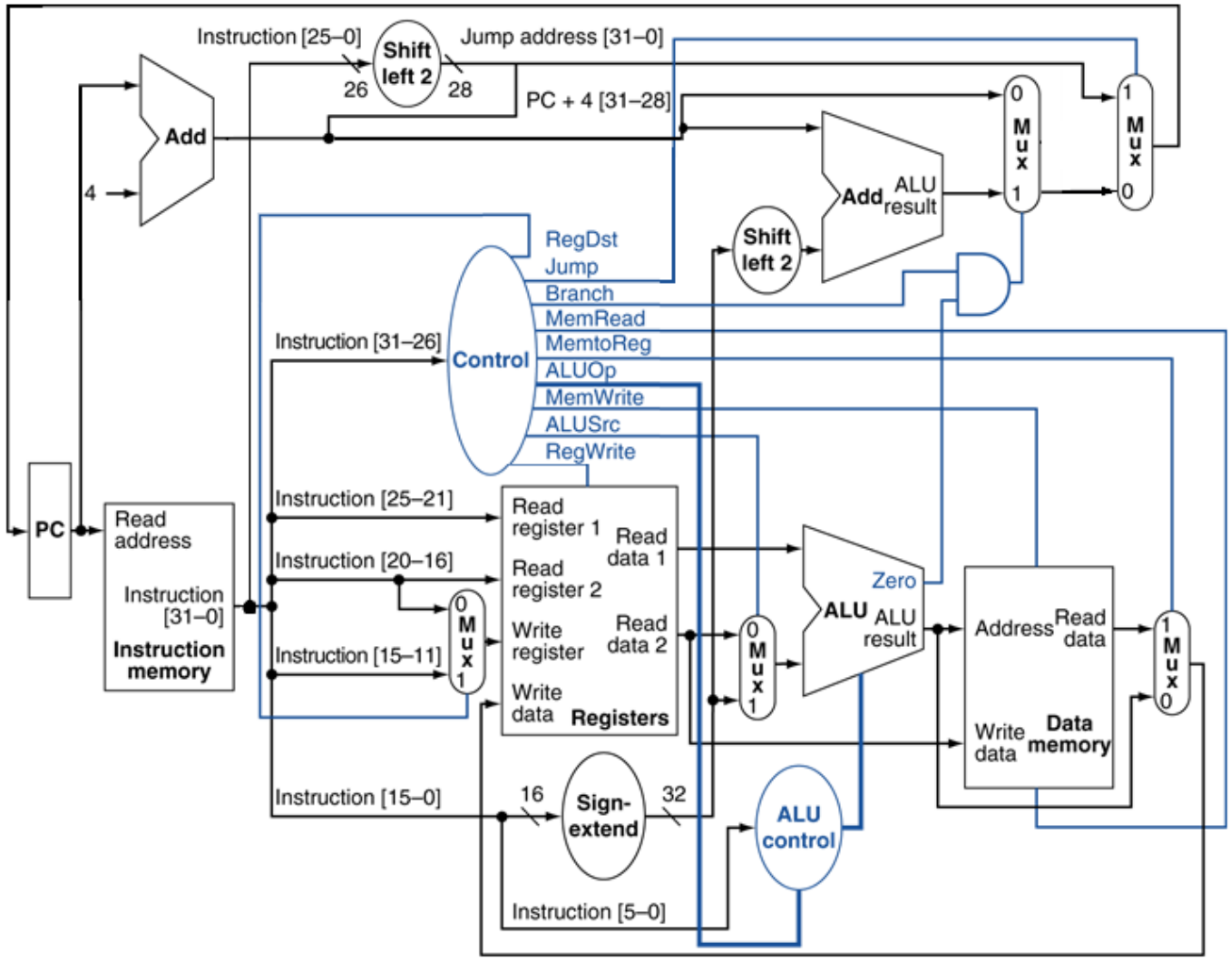
The next control signal is MemtoReg. Since our instruction – addi – is not a load word instruction, this signal should be set to 0.

Partial Credit 5:

MemtoReg = 0.

Solution 5:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

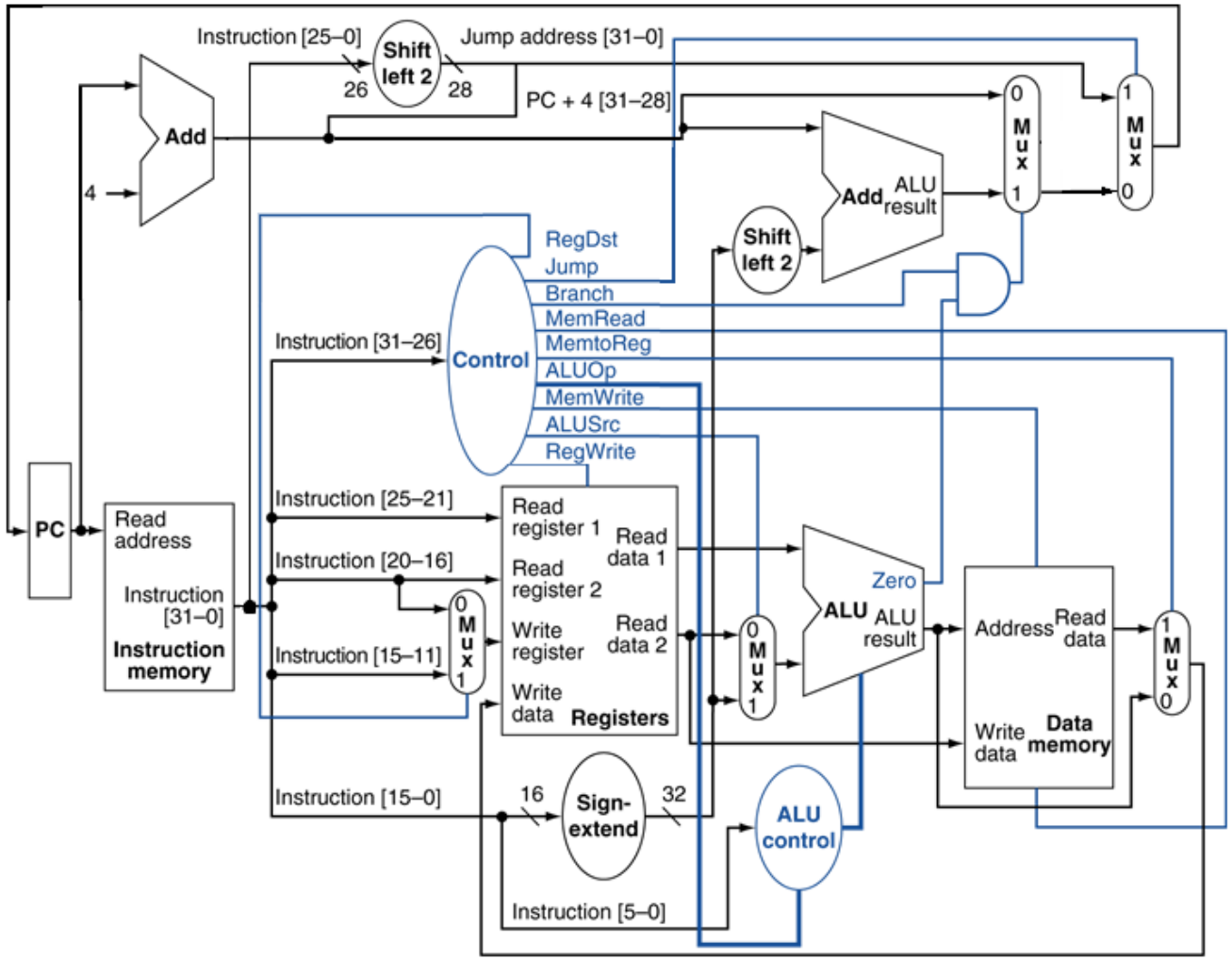
The next control signal is ALUOp. For this signal we need to consider what action the ALU should perform. With an addi instruction, we expect the ALU to perform an addition operation.

Partial Credit 6:

Solution 6:

ALUOp = add.

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

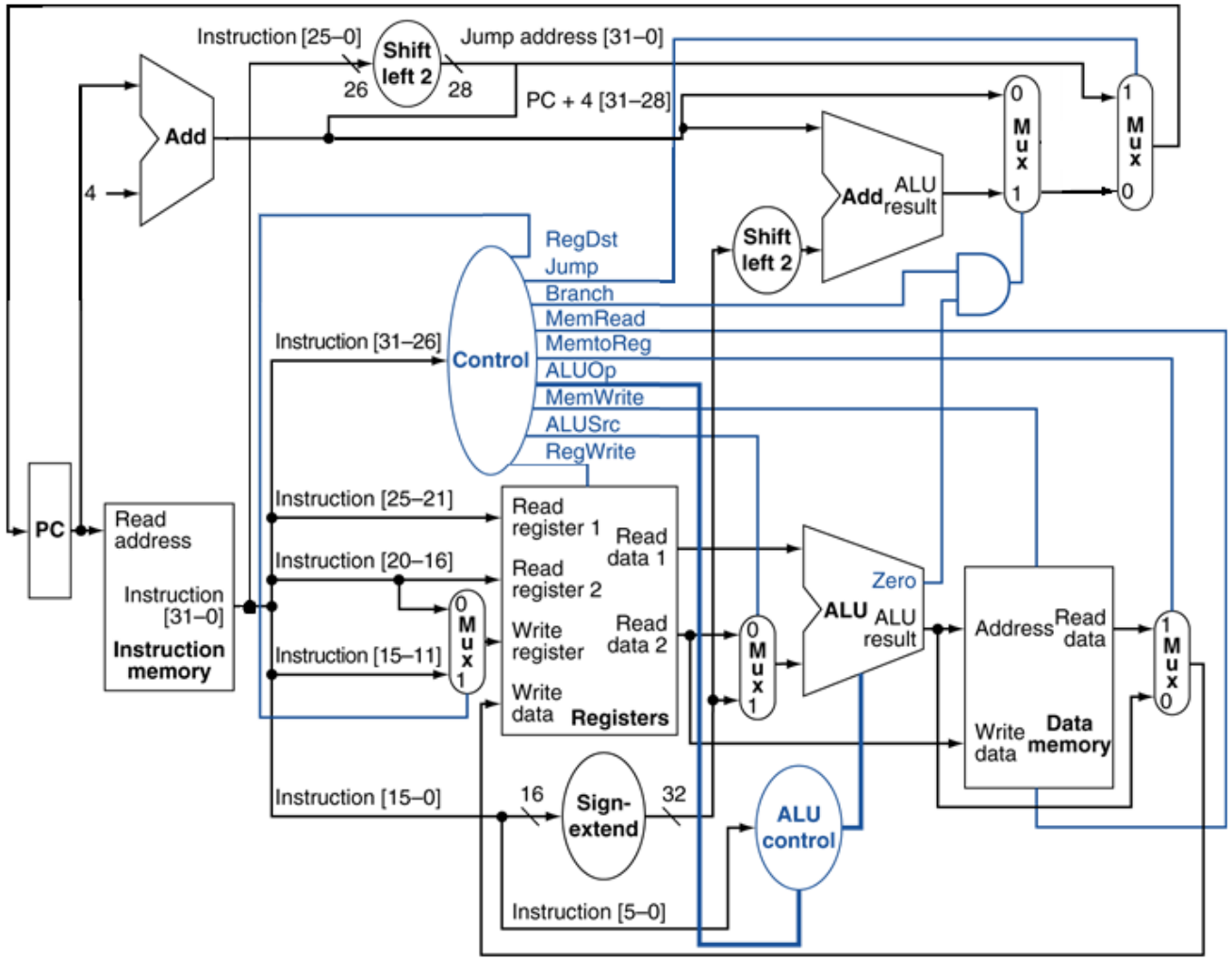
The next control signal is MemWrite. Since our instruction – addi – is not a store word instruction, this signal should be set to 0.

Partial Credit 7:

MemWrite = 0.

Solution 7:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

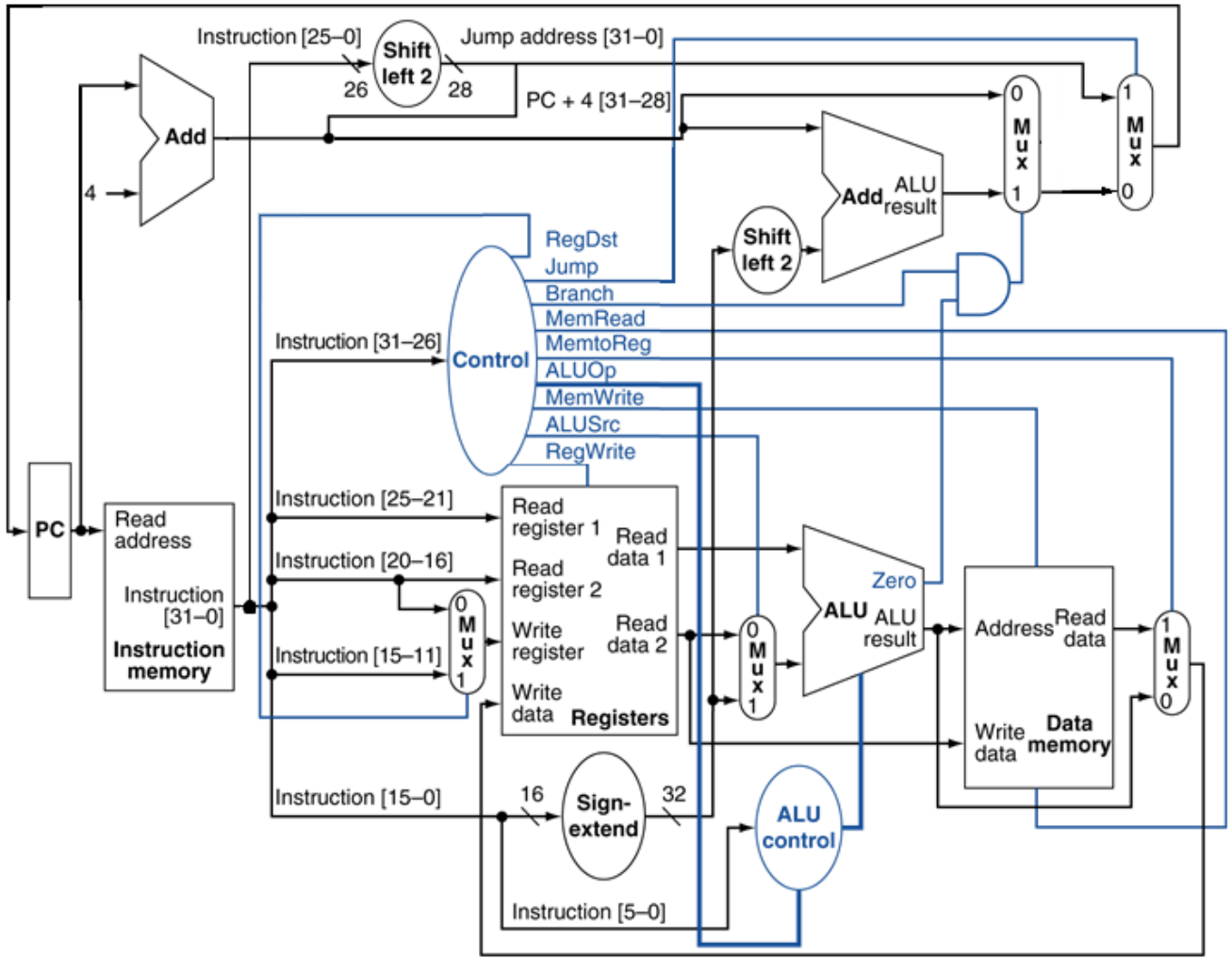
The next control signal is ALUSrc. Since our instruction – addi – seeks to add a register and a constant value together we should send the sign-extended immediate as the second input to the ALU. To do this we need to set ALUSrc to 1.

Partial Credit 8:

ALUSrc = 1.

Solution 8:

Example: Control Signals



How should the control signals be set for an addi instruction?

Given:

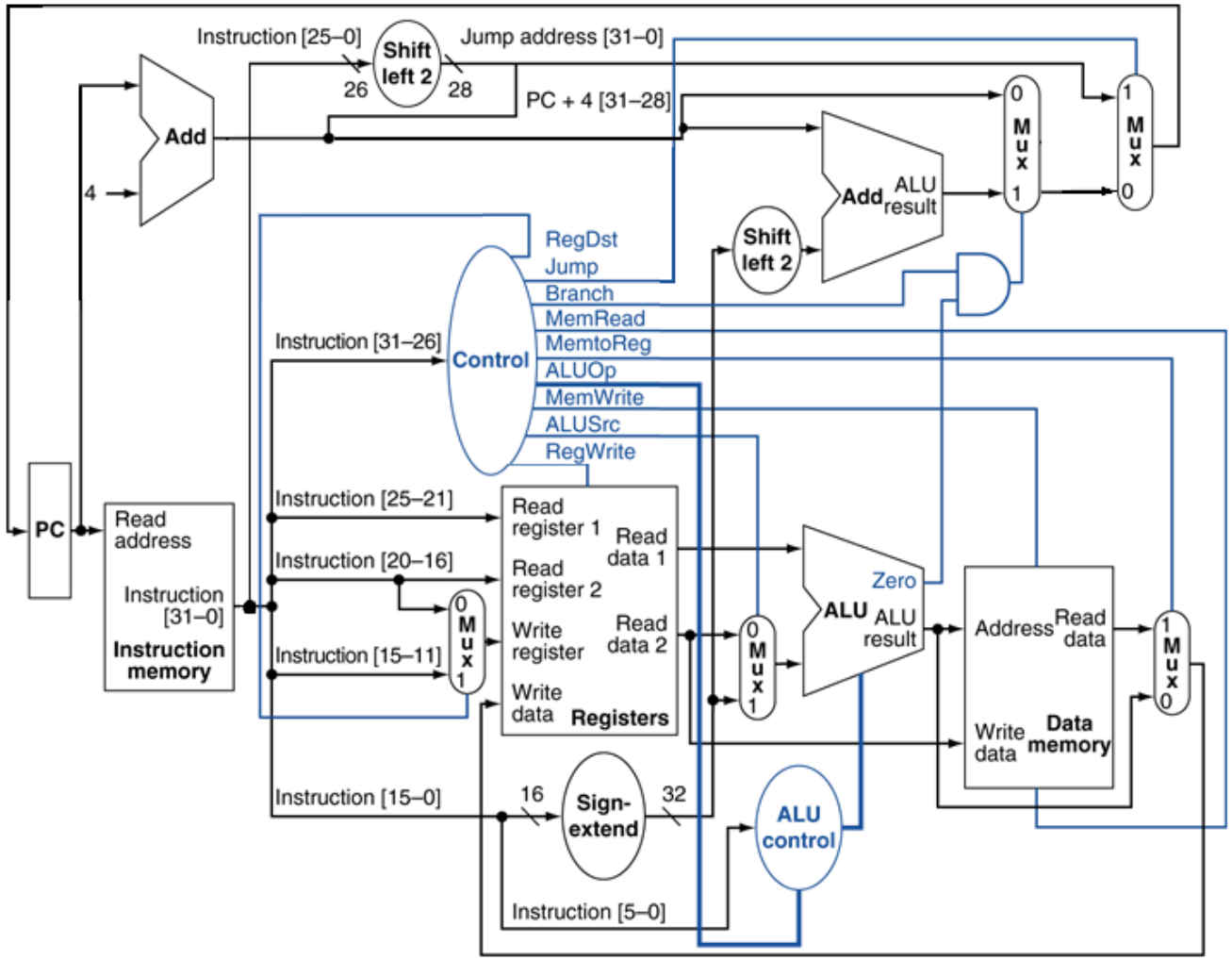
The last control signal is RegWrite. Since our instruction – addi – seeks to store the result of the add in the register file, we should assert this signal. This allows the processor to write to the register file.

Partial Credit 9:

RegWrite = 1.

Solution 9:

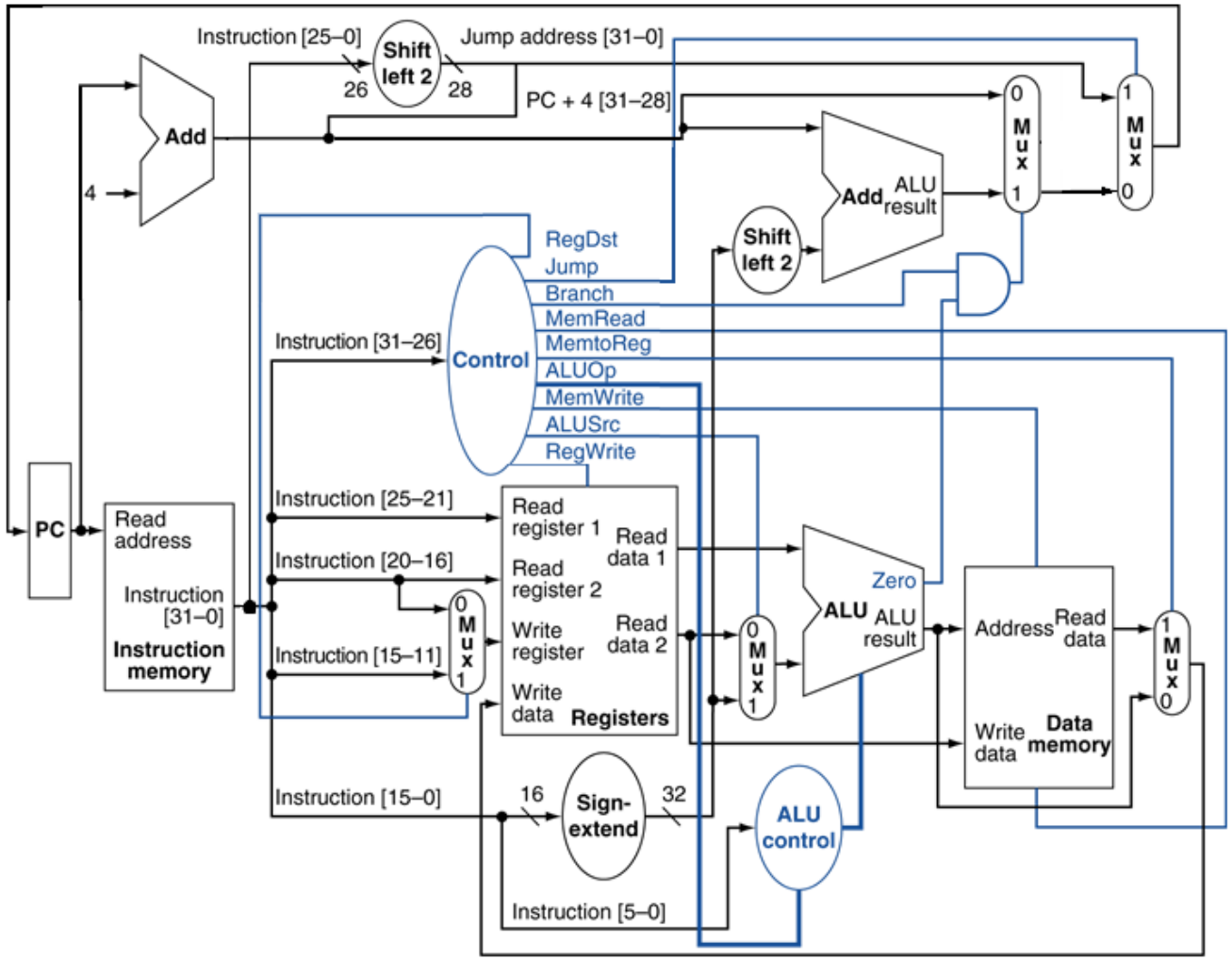
Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

The first control signal is RegDst. Our instruction – lw – is an I-Type instruction with the following format:

Partial Credit 1:

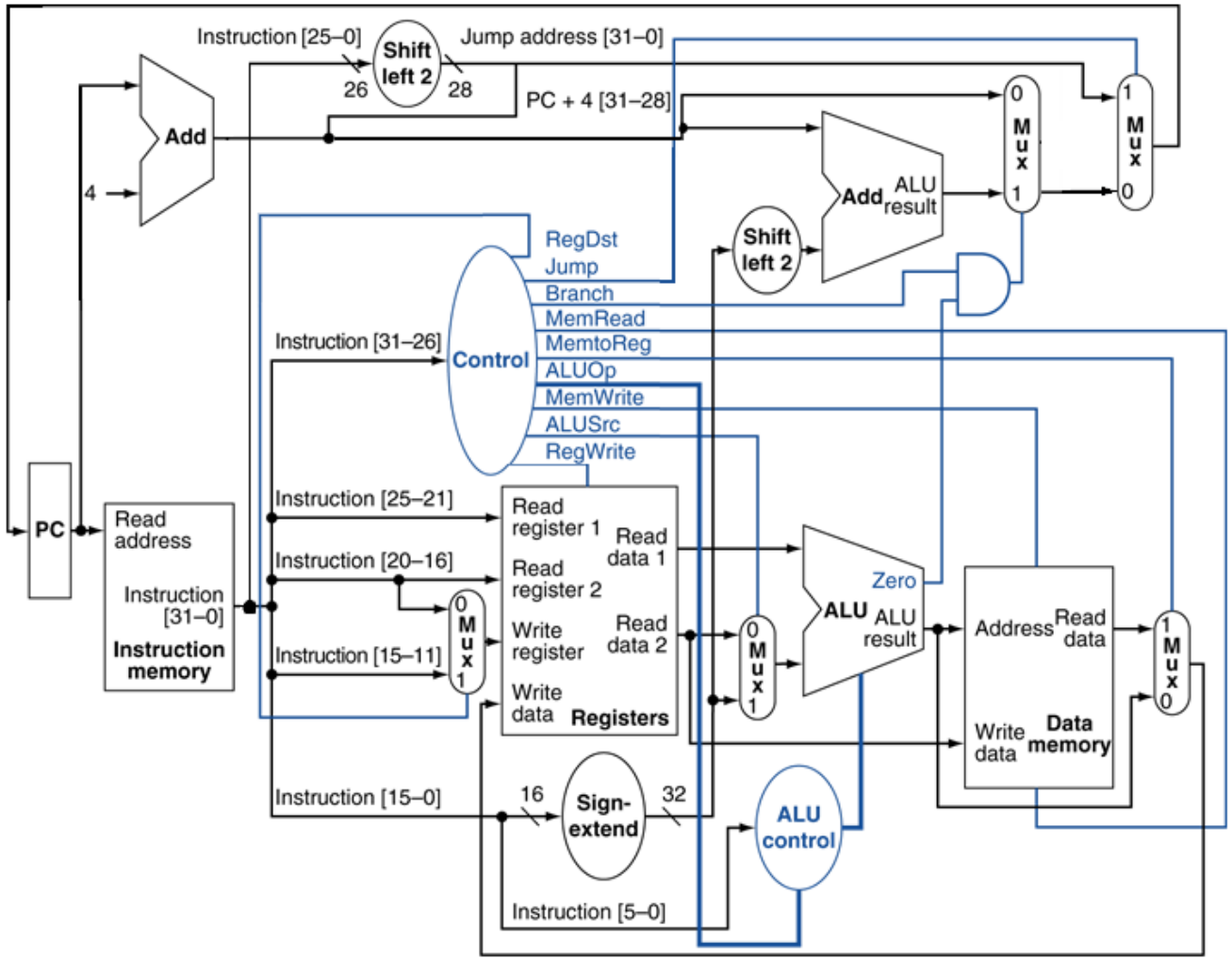
op	rs	rt	constant
6 bits	5 bits	5 bits	16 bits

This tells us the destination register is specified by bits 20-16. So we should set RegDst to 0 to let bits 20-16 specify the write register.

RegDst = 0.

Solution 1:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

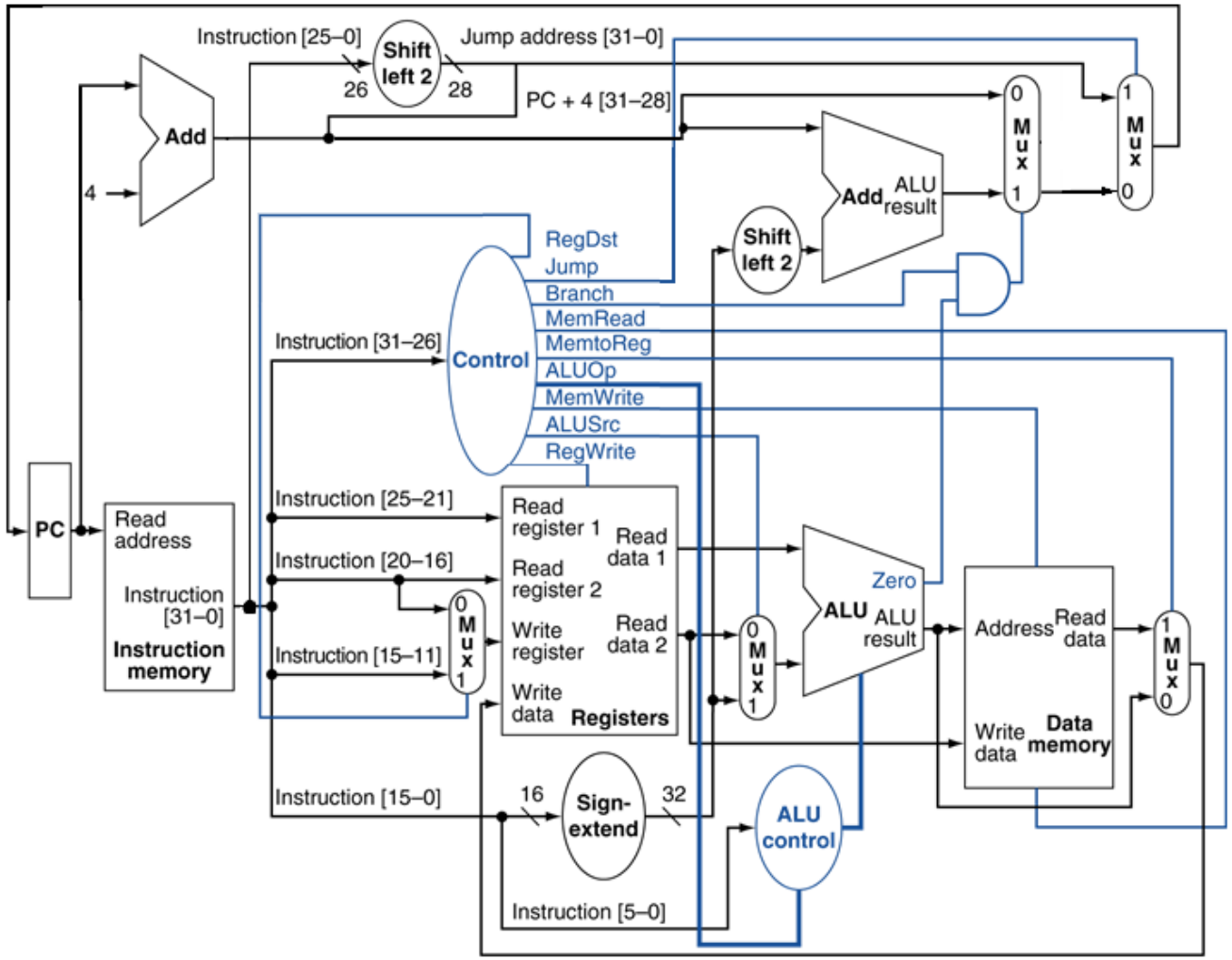
The next control signal is Jump. Since our instruction – lw – is not a jump instruction, this signal should be set to 0.

Partial Credit 2:

Jump = 0.

Solution 2:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

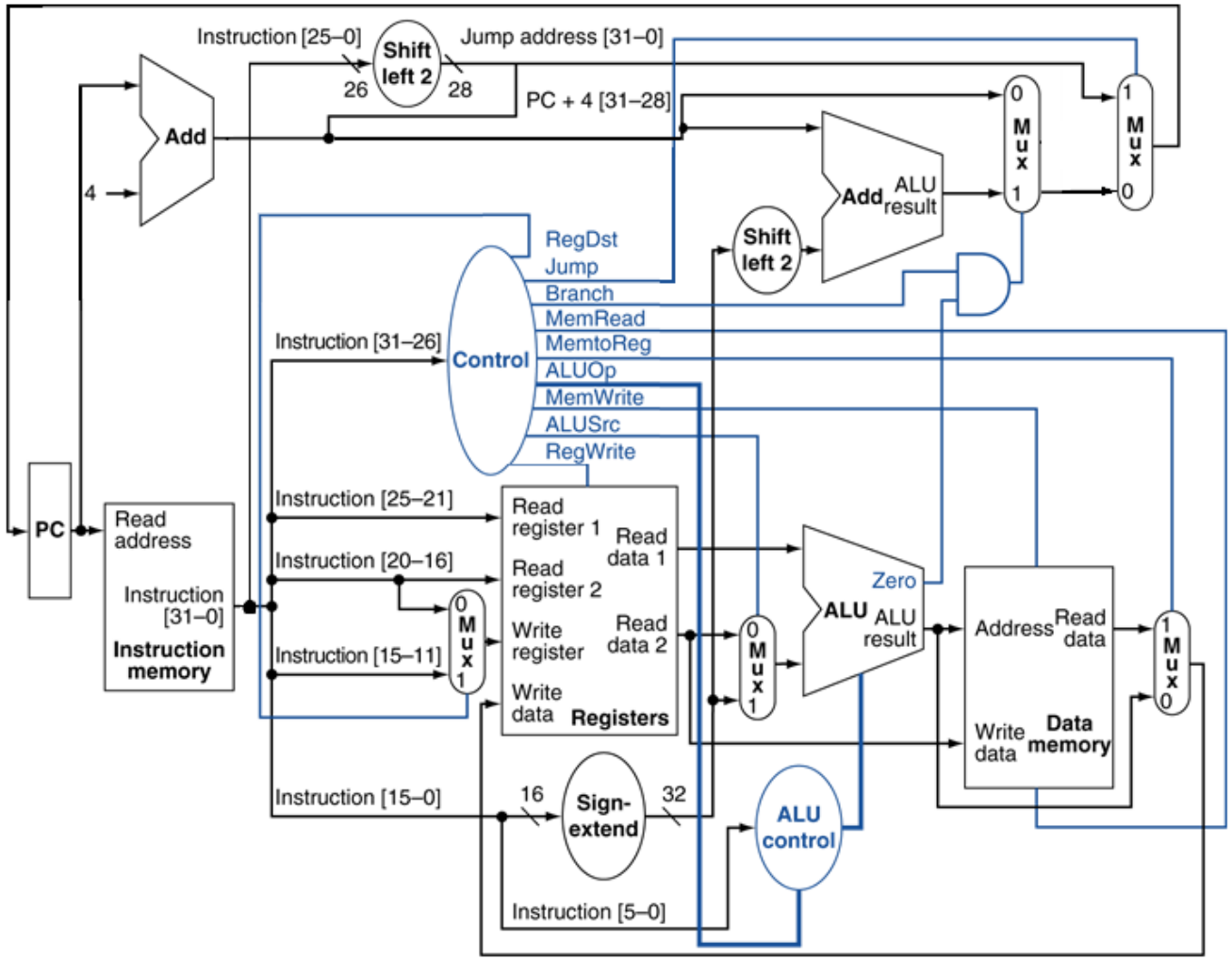
The next control signal is Branch. Since our instruction – lw – is not a branch instruction, this signal should be set to 0.

Partial Credit 3:

Branch = 0.

Solution 3:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

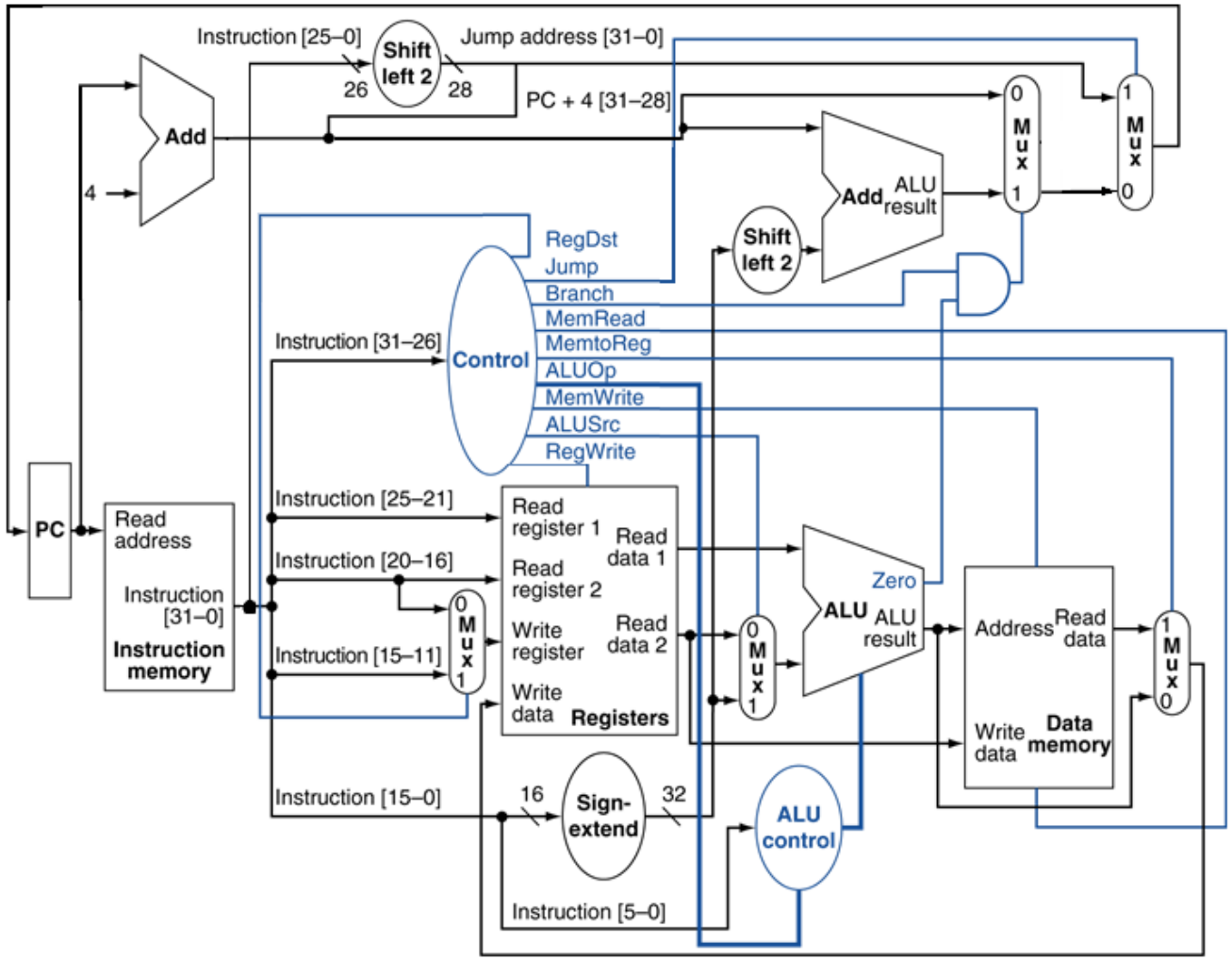
The next control signal is MemRead. Since this is a load word instruction, this signal should be set to 1.

Partial Credit 4:

MemRead = 1.

Solution 4:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

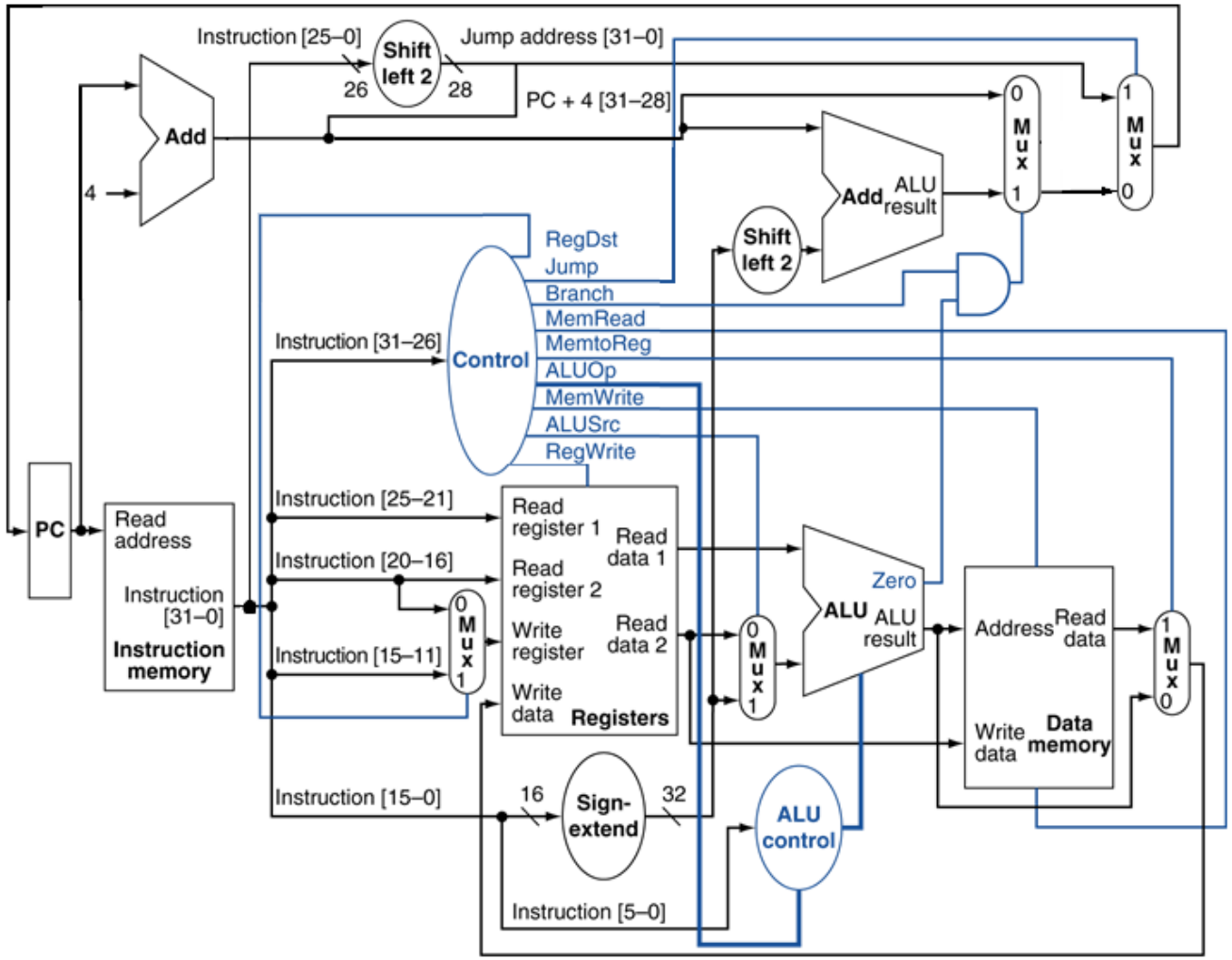
The next control signal is MemtoReg. Since our instruction is a load word instruction, this signal should be set to 1. This tells the processor to send the value obtained from memory back to the register file.

Partial Credit 5:

MemtoReg = 1.

Solution 5:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

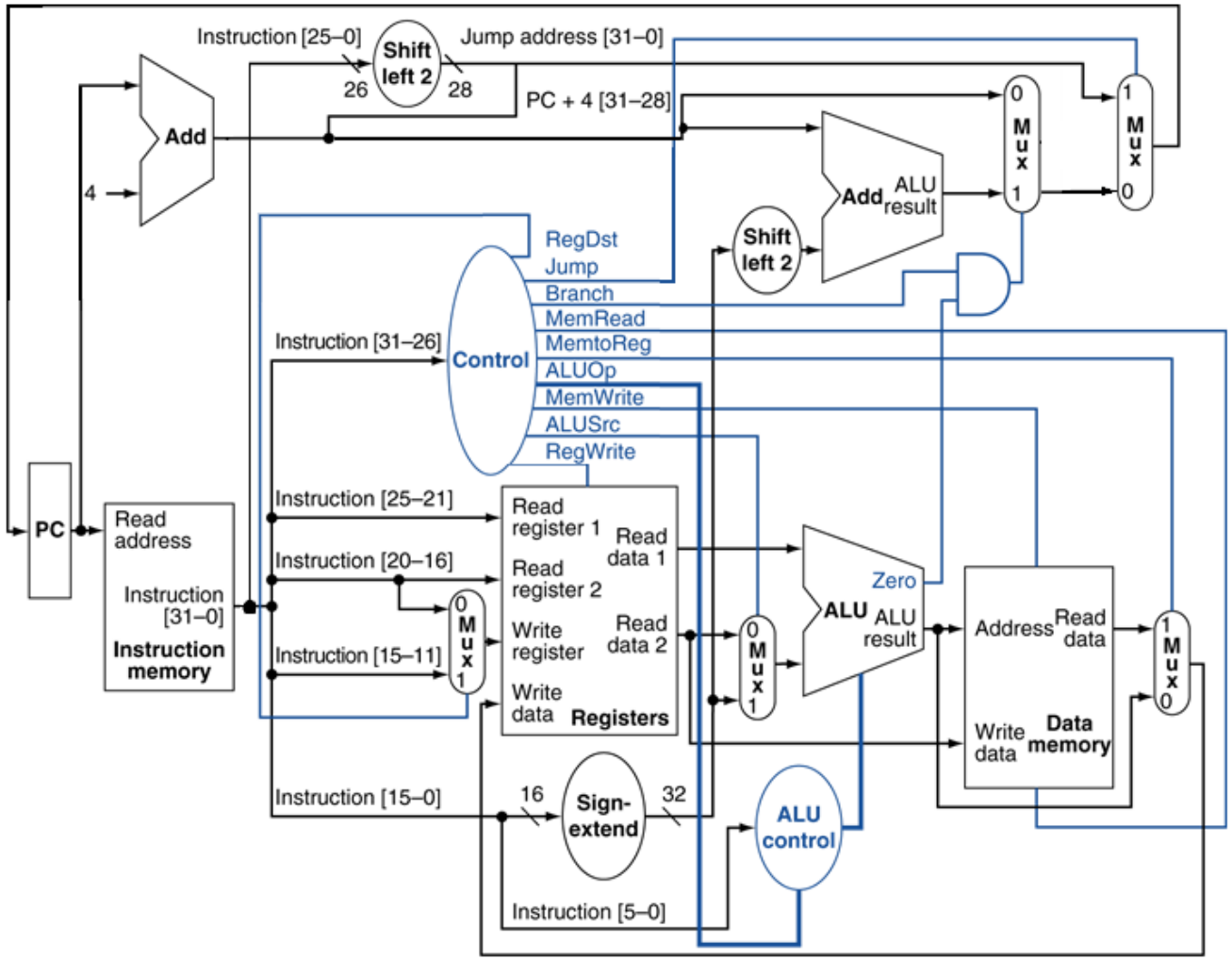
The next control signal is ALUOp. For this signal we need to consider what action the ALU should perform. With a lw instruction, we expect the ALU to perform an addition operation in order to add the base address and the offset together to get our final memory address.

Partial Credit 6:

Solution 6:

ALUOp = add.

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

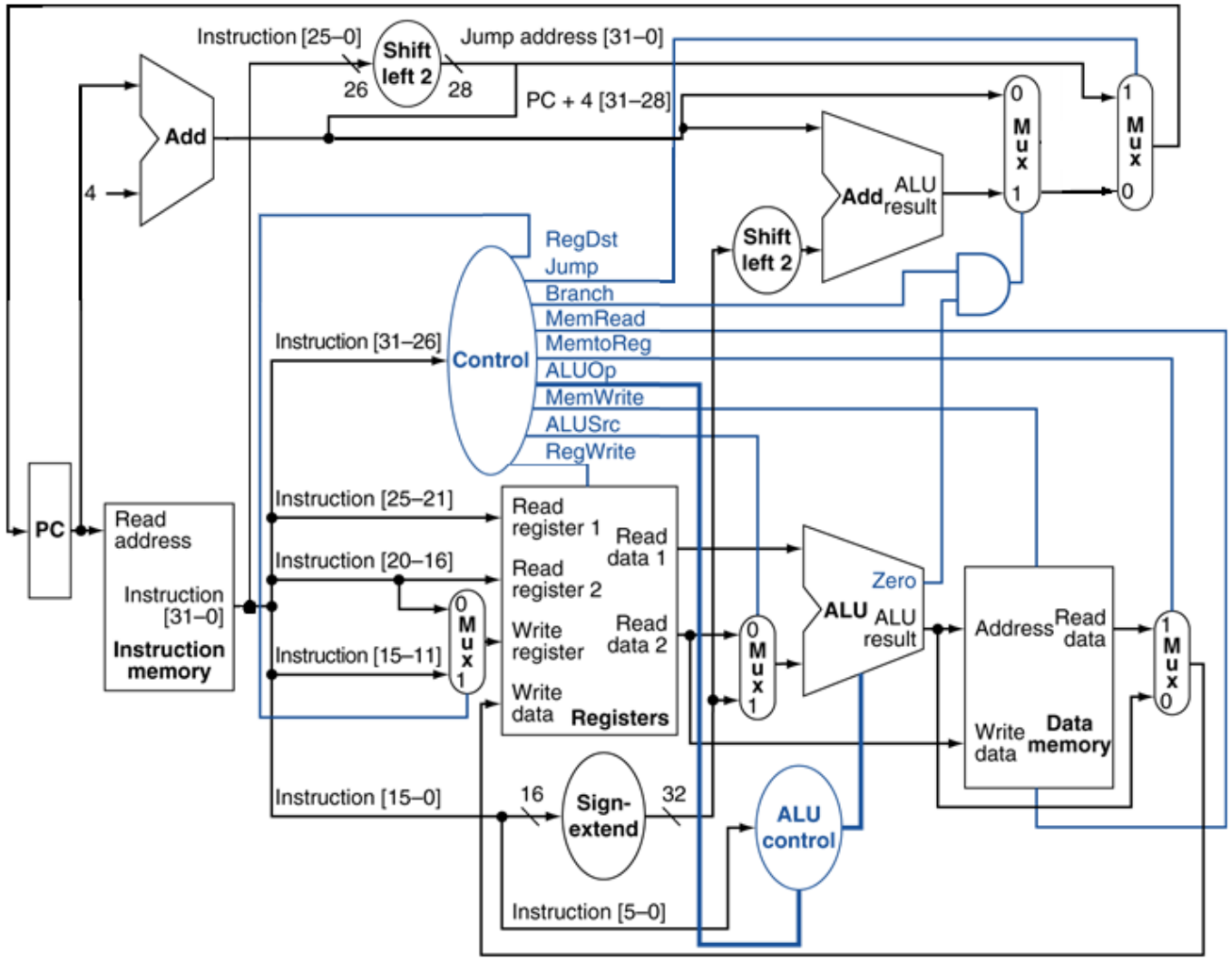
The next control signal is MemWrite. Since our instruction – lw – is not a store word instruction, this signal should be set to 0.

Partial Credit 7:

MemWrite = 0.

Solution 7:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

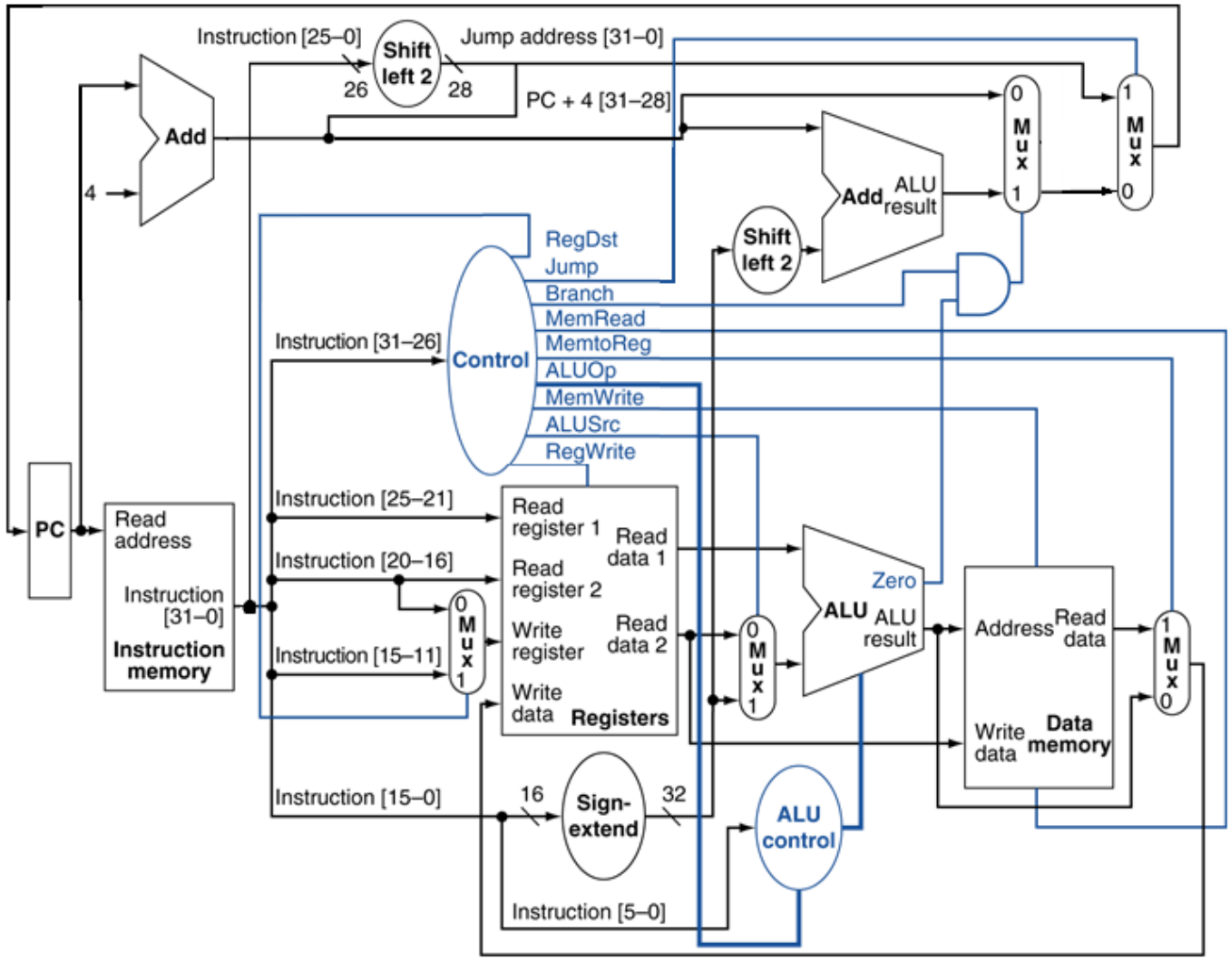
The next control signal is ALUSrc. Since our instruction – lw – seeks to add a register and a constant value together to form a memory address we should send the sign-extended immediate as the second input to the ALU. To do this we need to set ALUSrc to 1.

Partial Credit 8:

ALUSrc = 1.

Solution 8:

Example: Control Signals



How should the control signals be set for a lw instruction?

Given:

The last control signal is RegWrite. Since our instruction – lw – seeks to store the value obtained from memory in the register file, we should assert this signal. This allows the processor to write to the register file.

Partial Credit 9:

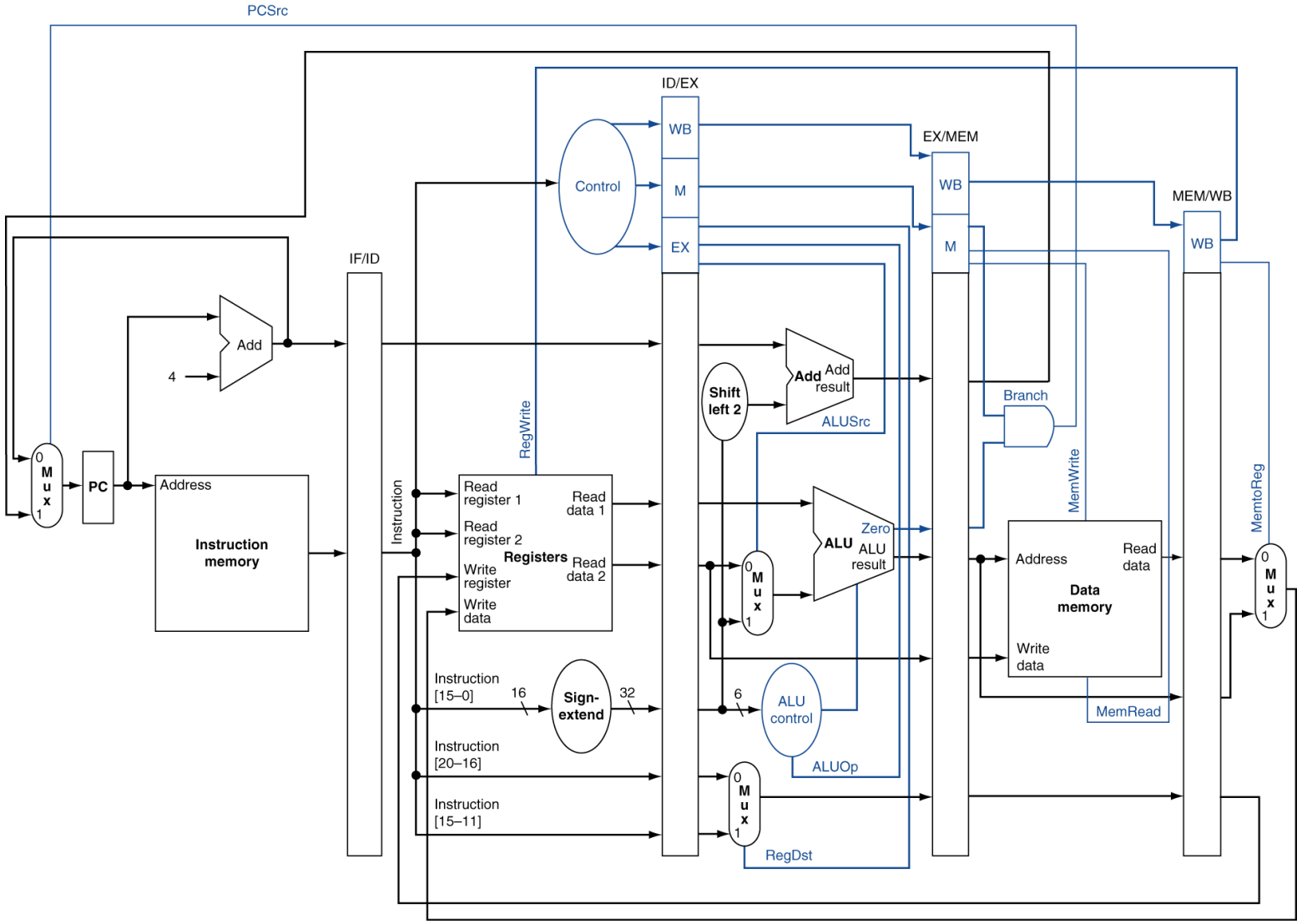
RegWrite = 1.

Solution 9:

Review: Pipelining

Pipelining allows us to overlap instructions in execution. When an instruction finishes the first stage and moves on to the second stage, the next instruction can start the first stage.

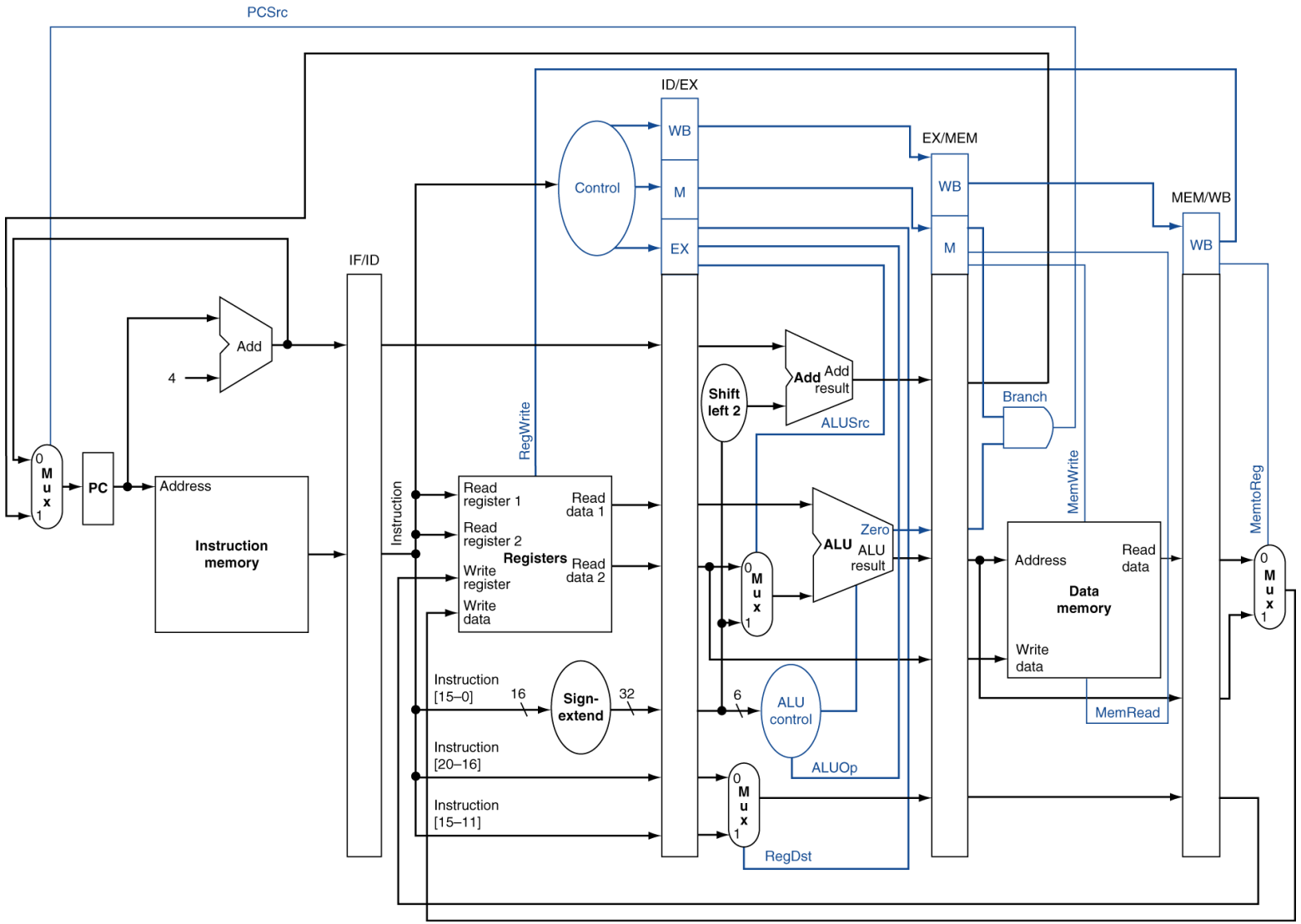
This increases our throughput: the number of instructions we can complete in a certain amount of time.



Review: Pipelining

A MIPS RISC has 5 stages in the pipeline:

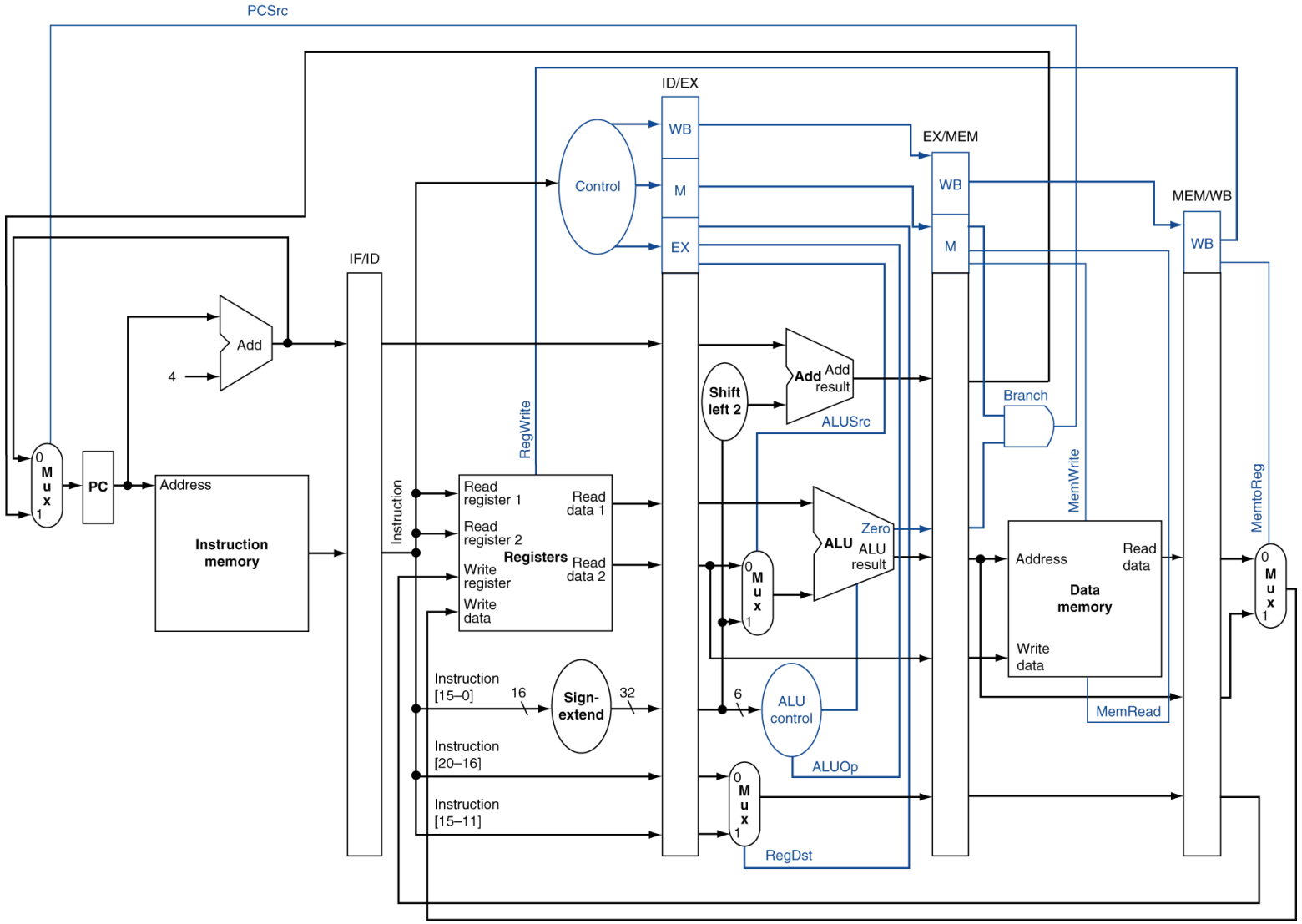
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (Ex)
- Memory Access (Mem)
- Write Back (WB)



Review: Pipelining

A MIPS RISC has 5 stages in the pipeline:

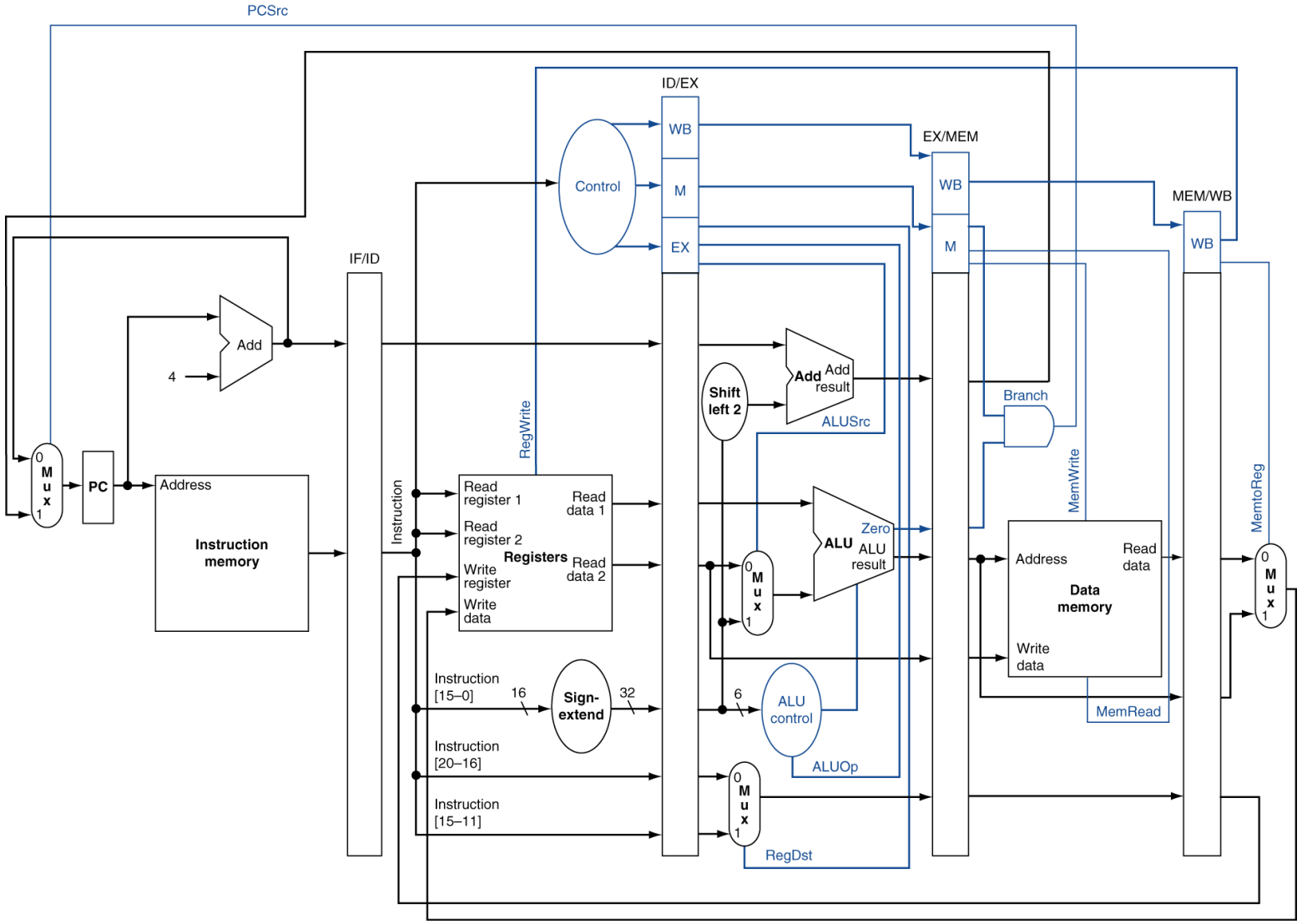
- Instruction Fetch (IF)
  - During this stage, we fetch the next instruction located in Instruction Memory at PC. We send this along to the pipeline register to wait for the next stage.
  - Also in this stage, we update PC by adding 4 to the current PC value.
- Instruction Decode (ID)
- Execution (Ex)
- Memory Access (Mem)
- Write Back (WB)



Review: Pipelining

A MIPS RISC has 5 stages in the pipeline:

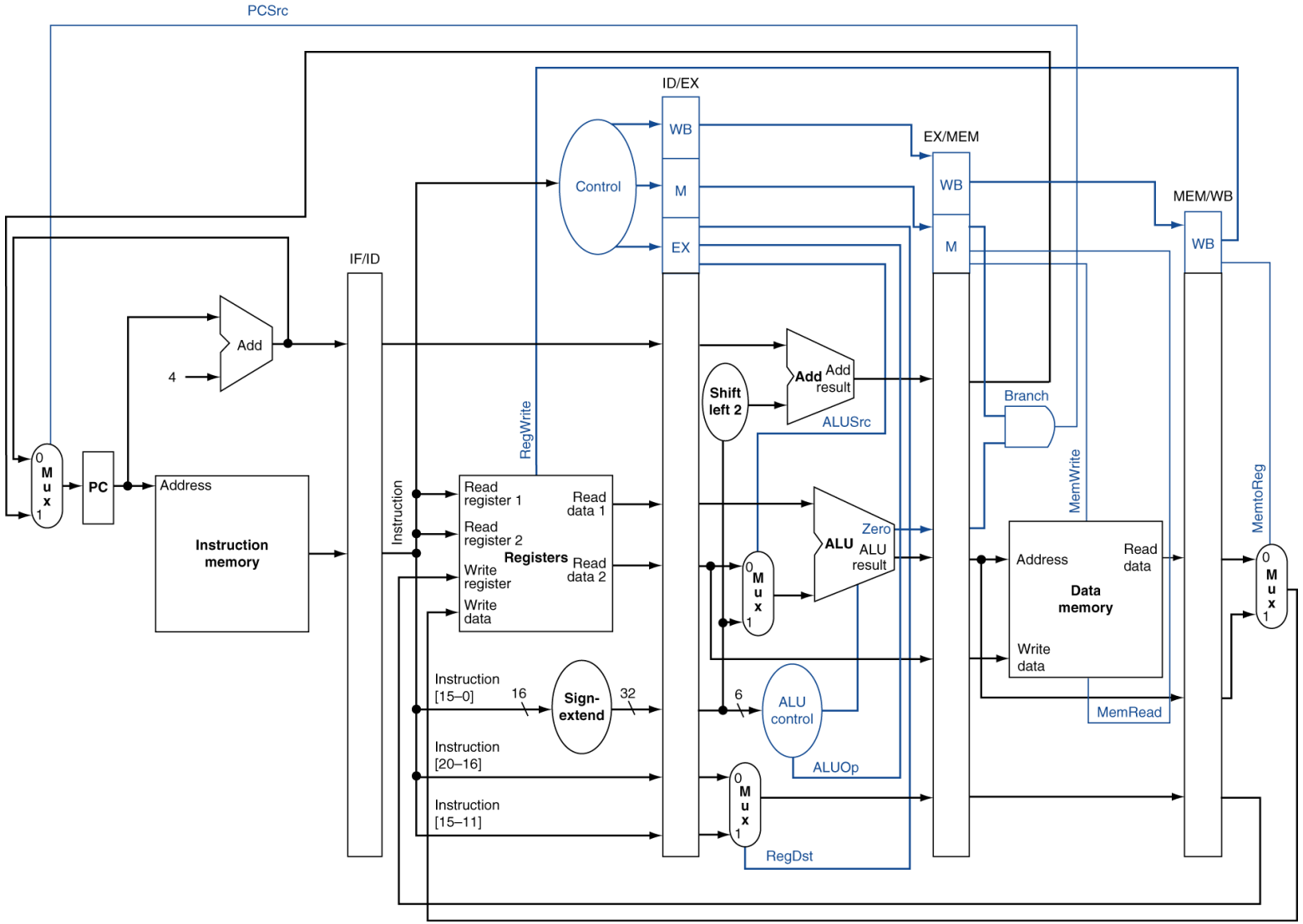
- Instruction Fetch (IF)
- Instruction Decode (ID)
  - In this stage, we send the opcode to the Control Unit and determine the control signals for the rest of the datapath.
  - We also read from the register file in this stage: setting up read data 1 and read data 2 for the next stage.
  - We also send our 16 least significant bits to the sign-extension unit.
- Execution (Ex)
- Memory Access (Mem)
- Write Back (WB)



Review: Pipelining

A MIPS RISC has 5 stages in the pipeline:

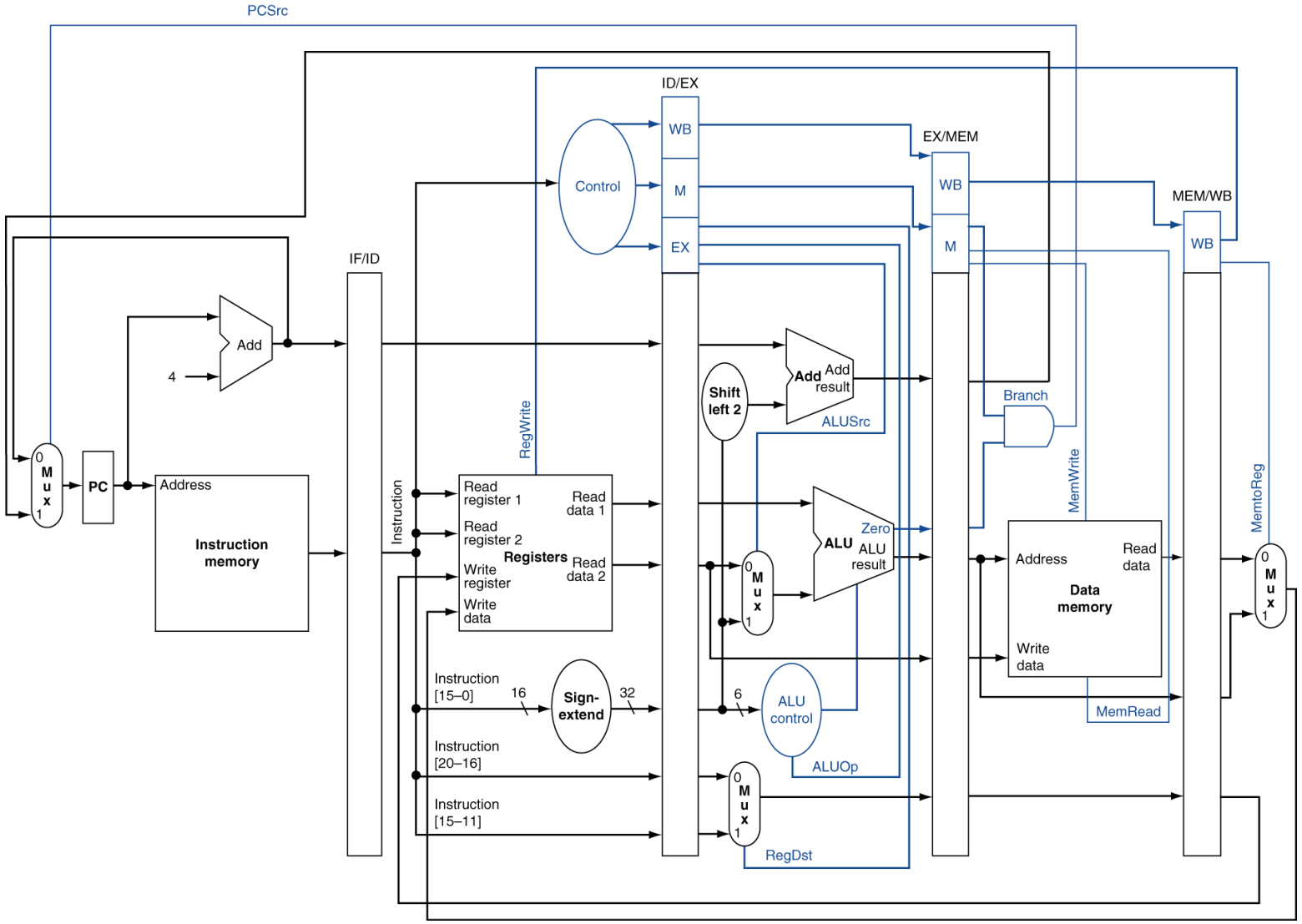
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (Ex)
  - In this stage we do several computations. The arithmetic logic unit will perform an action based on the opcode: add, subtract, and, or, slt.
  - We also calculate a branch target address and determine the location for the ALU result.
- Memory Access (Mem)
- Write Back (WB)



Review: Pipelining

A MIPS RISC has 5 stages in the pipeline:

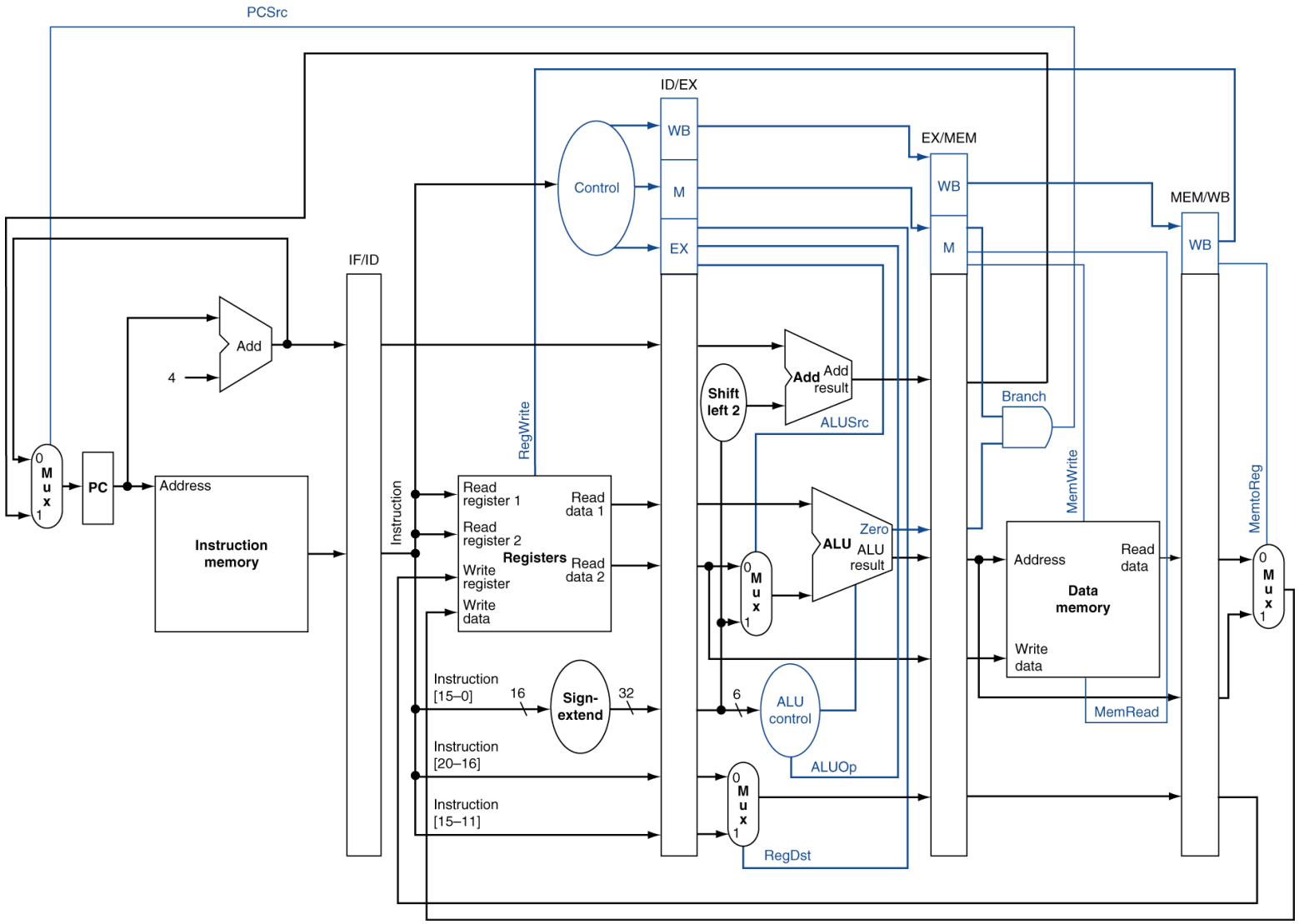
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (Ex)
- Memory Access (Mem)
  - In this stage we can access the Data Memory. We will do this if the current instruction is either a load word or store word.
  - We also resolve any branch decisions in this stage.
- Write Back (WB)



Review: Pipelining

A MIPS RISC has 5 stages in the pipeline:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (Ex)
- Memory Access (Mem)
- Write Back (WB)
  - In the last stage, we update the register file, if needed. We need to carry the write destination register along with the instruction to each pipeline stage to make sure we write the result to correct location.



Review: Hazards

Pipelining introduces Hazards to the datapath. These occur whenever the next instruction cannot continue as expected. There are three situation where that might happen:

Structure hazard

- A required resource is busy

Data hazard

- Need to wait for previous instruction to complete its data read/write

Control hazard

- Deciding on control action depends on previous instruction

Review: Hazards

Structural Hazards have been dealt with in the MIPS datapath by duplicating several structural units. We have separate instruction and data memories so that stage 1 and stage 4 do not have to compete. There are several shifters and adders to manipulate different target addresses, so that our next Program Counter will be ready when we need it.

Review: Hazards

Data Hazards occur whenever our current instruction has to wait for a previous instruction to complete. Consider the following example:

```
add     $t0, $s1, $s2
add     $s0, $t0, $zero
```

The second add instruction has $t0 as one of the sources, but the first add instruction has $t0 as it's result. The first add instruction will not be writing the result to $t0 until it reaches stage 5 (write back). But the second add instruction would normally be reading when the first instruction is in stage 3 (execution).

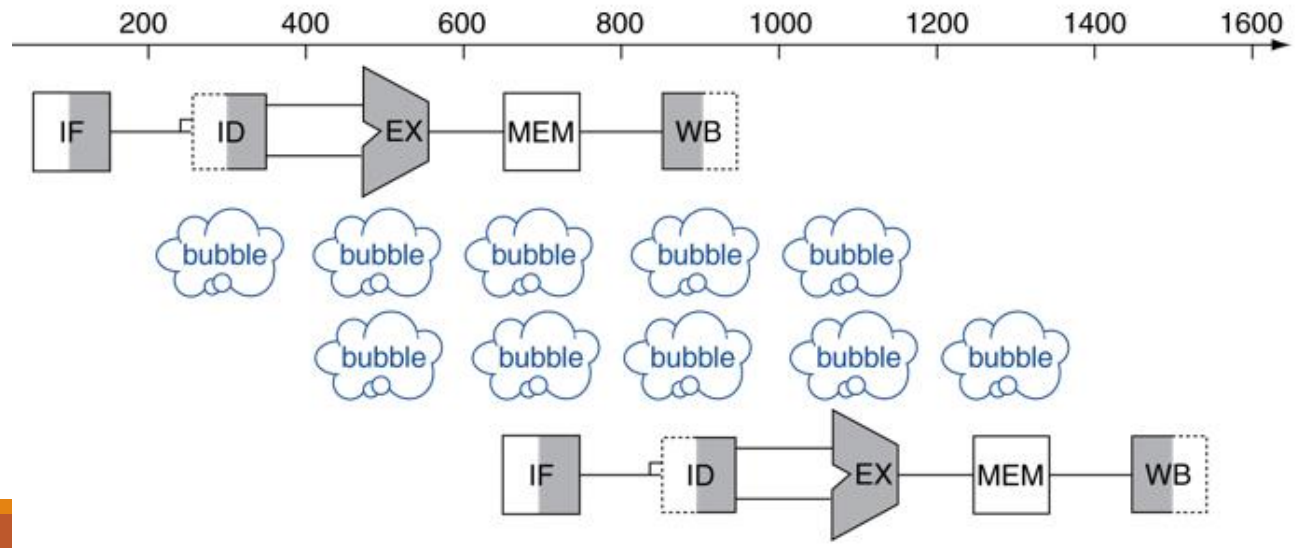
Review: Hazards

Data Hazard:

add $t0, $s1, $s2

add $s0, $t0, $zero

There are a couple ways to address this. The easiest method is to stall the pipeline until the first instruction has completed enough for the second instruction to begin. The write back stage of the first instruction can overlap with the instruction decode stage of the second instruction.

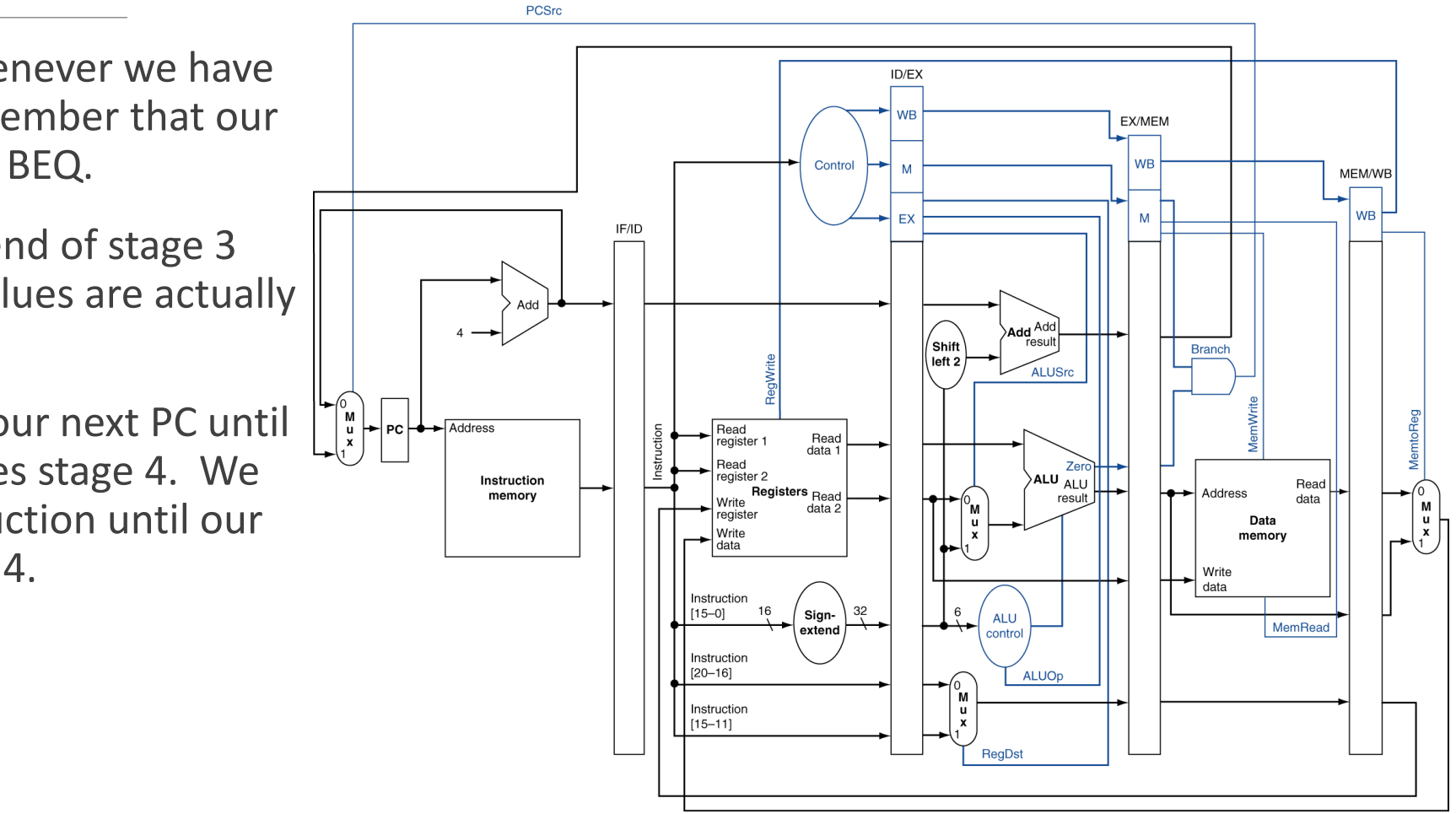


Review: Hazards

Control Hazards occur whenever we have a branch instruction. Remember that our RISC system only supports BEQ.

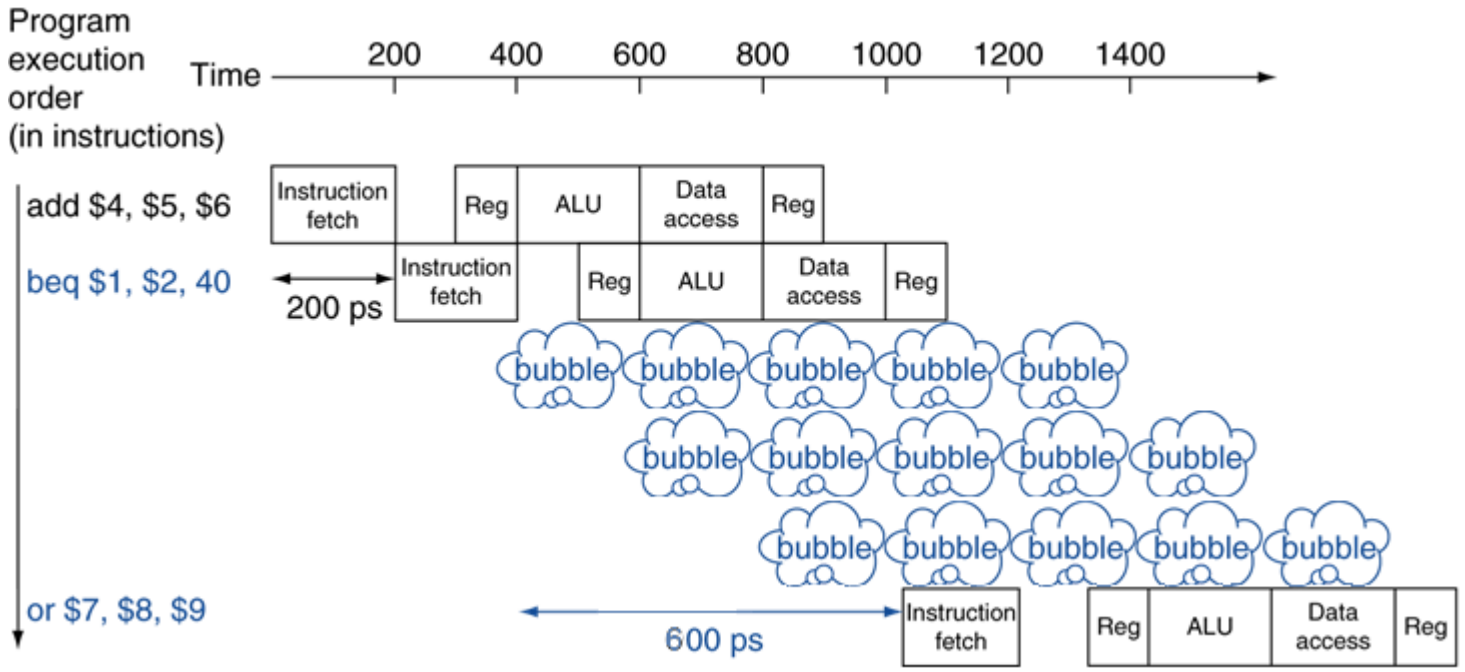
We won't know until the end of stage 3 whether or not the two values are actually equal to each other.

Therefore, we cannot set our next PC until the BEQ instruction reaches stage 4. We won't fetch the next instruction until our BEQ instruction is in stage 4.



Review: Hazards

Similar to data hazards, the easiest method is to stall until the branch has been resolved.



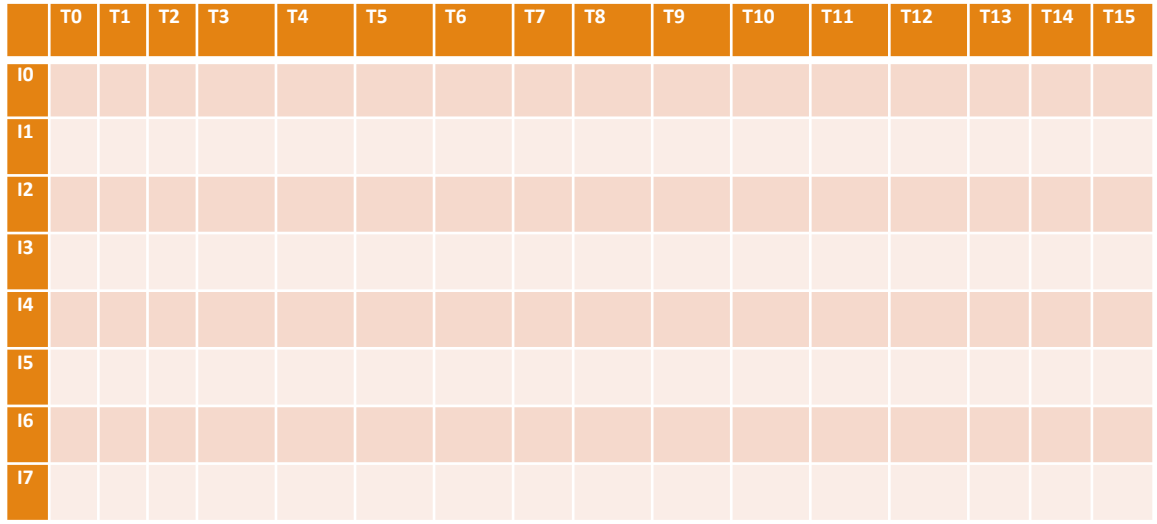
Example: Hazards

Given: Consider the following assembly language code:

```
I0:     sub     $t4, $t1, $s2
I1:     add     $s3, $s2, $t4
I2:     add     $t1, $t0, $s0
I3:     add     $t1, $t0, $s0
I4:     sub     $t4, $t1, $s2
I5:     lw      $t5, 0($t1)
I6:     slt     $t0, $s0, $s1
I7:     sw      $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

I0: _____
I1: _____
I2: _____
I3: _____
I4: _____
I5: _____
I6: _____
I7: _____



Example: Hazards

Given: Consider the following assembly language code:

```
I0:   sub   $t4, $t1, $s2
I1:   add   $s3, $s2, $t4
I2:   add   $t1, $t0, $s0
I3:   add   $t1, $t0, $s0
I4:   sub   $t4, $t1, $s2
I5:   lw    $t5, 0($t1)
I6:   slt   $t0, $s0, $s1
I7:   sw    $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 1:

I0: __no hazard_____
I1: _____
I2: _____
I3: _____
I4: _____
I5: _____
I6: _____
I7: _____

Partial Credit 1:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1																
I2																
I3																
I4																
I5																
I6																
I7																

Since hazards are situations that prevent an instruction from running, we assume the first instruction cannot experience a hazard as there is nothing prior to it.

We show the stages of instruction zero in the chart to track when $t4 – the destination register – will be updated.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:   sub   $t4, $t1, $s2
I1:   add   $s3, $s2, $t4
I2:   add   $t1, $t0, $s0
I3:   add   $t1, $t0, $s0
I4:   sub   $t4, $t1, $s2
I5:   lw    $t5, 0($t1)
I6:   slt   $t0, $s0, $s1
I7:   sw    $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 2:

I0: __no hazard_____
I1: __data hazard_____
I2: _____
I3: _____
I4: _____
I5: _____
I6: _____
I7: _____

Partial Credit 2:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2																
I3																
I4																
I5																
I6																
I7																

In instruction one, we are trying to add $t4 and $s2 together. However, $t4 is being written by the previous instruction – instruction zero. In the chart we can see that this value does not reach the register file until T4. Since we write in the first half of the clock cycle and read in the second, we can overlap the WB stage of I0 and the ID stage of I1.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:   sub   $t4, $t1, $s2
I1:   add   $s3, $s2, $t4
I2:   add   $t1, $t0, $s0
I3:   add   $t1, $t0, $s0
I4:   sub   $t4, $t1, $s2
I5:   lw    $t5, 0($t1)
I6:   slt   $t0, $s0, $s1
I7:   sw    $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 3:

I0: __no hazard_____
I1: __data hazard_____
I2: __no hazard_____
I3: _____
I4: _____
I5: _____
I6: _____
I7: _____

Partial Credit 3:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2					IF	ID	EX	MEM	WB							
I3																
I4																
I5																
I6																
I7																

In instruction two, we are trying to add $t0 and $s0 together. Neither of these registers have been modified by our segment of assembly code, so this instruction should be able to proceed without a hazard.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:   sub   $t4, $t1, $s2
I1:   add   $s3, $s2, $t4
I2:   add   $t1, $t0, $s0
I3:   add   $t1, $t0, $s0
I4:   sub   $t4, $t1, $s2
I5:   lw    $t5, 0($t1)
I6:   slt   $t0, $s0, $s1
I7:   sw    $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 4:

I0: __no hazard_____
I1: __data hazard_____
I2: __no hazard_____
I3: __no hazard_____
I4: _____
I5: _____
I6: _____
I7: _____

Partial Credit 4:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2					IF	ID	EX	MEM	WB							
I3						IF	ID	EX	MEM	WB						
I4																
I5																
I6																
I7																

In instruction three, we are trying to add $t0 and $s0 together. Even though we read these in our previous instruction these registers have not been modified. So this instruction should be able to proceed without a hazard.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:   sub   $t4, $t1, $s2
I1:   add   $s3, $s2, $t4
I2:   add   $t1, $t0, $s0
I3:   add   $t1, $t0, $s0
I4:   sub   $t4, $t1, $s2
I5:   lw    $t5, 0($t1)
I6:   slt   $t0, $s0, $s1
I7:   sw    $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 5:

I0: __no hazard_____
I1: __data hazard_____
I2: __no hazard_____
I3: __no hazard_____
I4: __data hazard_____
I5: _____
I6: _____
I7: _____

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2					IF	ID	EX	MEM	WB							
I3						IF	ID	EX	MEM	WB						
I4									IF	ID	EX	MEM	WB			
I5																
I6																
I7																

Partial Credit 5:

In instruction four, we are trying to subtract $s2 from $t1. $t1 was most recently modified by instruction three, which will not write the value of $t1 to the register file until T9. I4 must wait until the same clock cycle to read the correct value of $t1. This is a data hazard.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:   sub   $t4, $t1, $s2
I1:   add   $s3, $s2, $t4
I2:   add   $t1, $t0, $s0
I3:   add   $t1, $t0, $s0
I4:   sub   $t4, $t1, $s2
I5:   lw    $t5, 0($t1)
I6:   slt   $t0, $s0, $s1
I7:   sw    $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 6:

I0: __no hazard_____
I1: __data hazard_____
I2: __no hazard_____
I3: __no hazard_____
I4: __data hazard_____
I5: __no hazard_____
I6: _____
I7: _____

Partial Credit 6:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2					IF	ID	EX	MEM	WB							
I3						IF	ID	EX	MEM	WB						
I4									IF	ID	EX	MEM	WB			
I5										IF	ID	EX	MEM	WB		
I6																
I7																

In instruction five, we load a value from memory at the location $t1 + 0 and store the result into $t5. $t1 has had time to become stable since being written by I3.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:    sub    $t4, $t1, $s2
I1:    add    $s3, $s2, $t4
I2:    add    $t1, $t0, $s0
I3:    add    $t1, $t0, $s0
I4:    sub    $t4, $t1, $s2
I5:    lw     $t5, 0($t1)
I6:    slt    $t0, $s0, $s1
I7:    sw     $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 7:

I0: __no hazard_____
I1: __data hazard_____
I2: __no hazard_____
I3: __no hazard_____
I4: __data hazard_____
I5: __no hazard_____
I6: __no hazard_____
I7: _____

Partial Credit 7:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2					IF	ID	EX	MEM	WB							
I3						IF	ID	EX	MEM	WB						
I4									IF	ID	EX	MEM	WB			
I5										IF	ID	EX	MEM	WB		
I6											IF	ID	EX	MEM	WB	
I7																

In instruction six, we check to see if $s0 is less than $s1. Neither of these registers have been modified by our segment of assembly code, so this instruction can proceed without hazards.

Example: Hazards

Given: Consider the following assembly language code:

```
I0:    sub    $t4, $t1, $s2
I1:    add    $s3, $s2, $t4
I2:    add    $t1, $t0, $s0
I3:    add    $t1, $t0, $s0
I4:    sub    $t4, $t1, $s2
I5:    lw     $t5, 0($t1)
I6:    slt    $t0, $s0, $s1
I7:    sw     $s2, 0($s0)
```

For each instruction, identify whether or not a hazard should be detected. If so, identify the type of hazard as structure, data, or control. Assume the instructions are being processed on a MIPS pipelined datapath without forwarding.

Solution 8:

I0: __no hazard_____
I1: __data hazard_____
I2: __no hazard_____
I3: __no hazard_____
I4: __data hazard_____
I5: __no hazard_____
I6: __no hazard_____
I7: __no hazard_____

Partial Credit 8:

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
I0	IF	ID	EX	MEM	WB											
I1				IF	ID	EX	MEM	WB								
I2					IF	ID	EX	MEM	WB							
I3						IF	ID	EX	MEM	WB						
I4									IF	ID	EX	MEM	WB			
I5										IF	ID	EX	MEM	WB		
I6											IF	ID	EX	MEM	WB	
I7												IF	ID	EX	MEM	WB

In instruction seven, we are trying to store the value of $s2 into memory at the location $s0 + 0. Neither of these registers have been modified by our segment of assembly code, so this instruction can proceed without hazards.