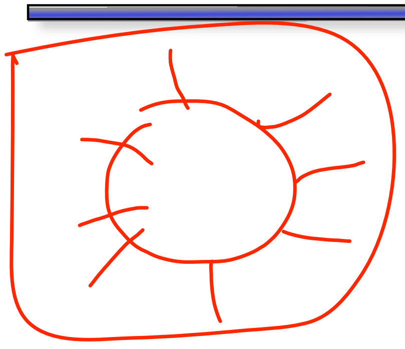
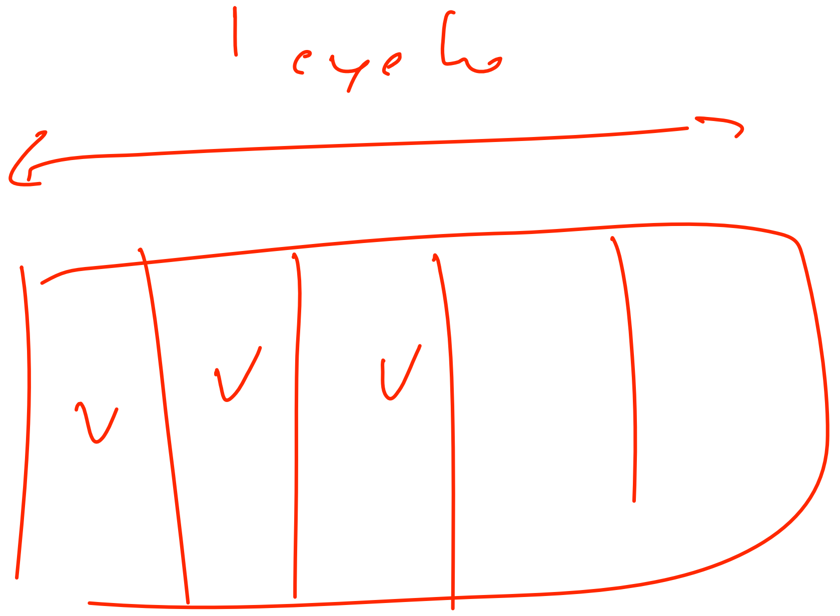



1 cycle

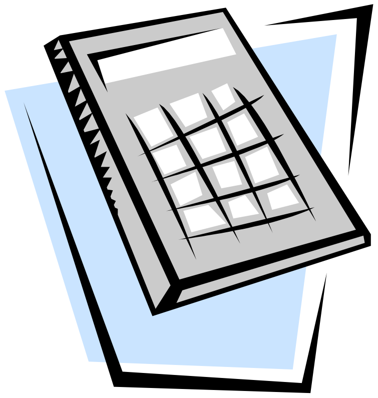
Multicycle Datapath

- As an added bonus, we can eliminate some of the extra hardware from the single-cycle datapath.
  - We will restrict ourselves to using each functional unit once per cycle, just like before.
  - But since instructions require multiple cycles, we could reuse some units in a *different* cycle during the execution of a single instruction.
- For example, we could use the same ALU:
  - to increment the PC (first clock cycle), and
  - for arithmetic operations (third clock cycle).

Proposed execution stages

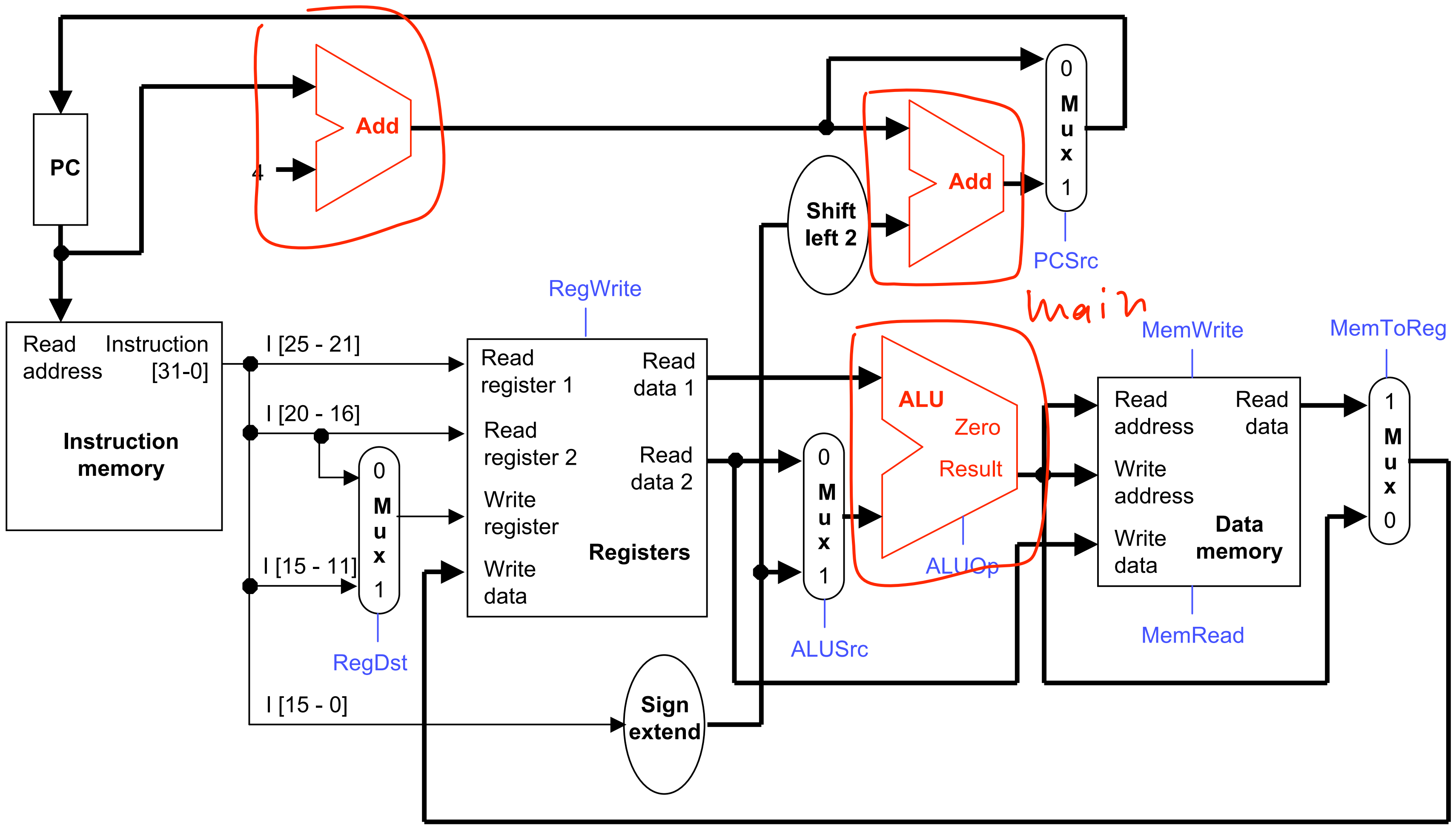
1. Instruction fetch and PC increment ALU
2. Reading sources from the register file
3. Performing an ALU computation ALU
4. Reading or writing (data) memory
5. Storing data back to the register file

Two extra adders



- Our original single-cycle datapath had an ALU and two adders.
- The arithmetic-logic unit had two responsibilities.
  - Doing an operation on two registers for arithmetic instructions. ✓ branch
  - Adding a register to a sign-extended constant, to compute effective addresses for lw and sw instructions. mem inst
- One of the extra adders incremented the PC by computing PC + 4.
- The other adder computed branch targets, by adding a sign-extended, shifted offset to (PC + 4).

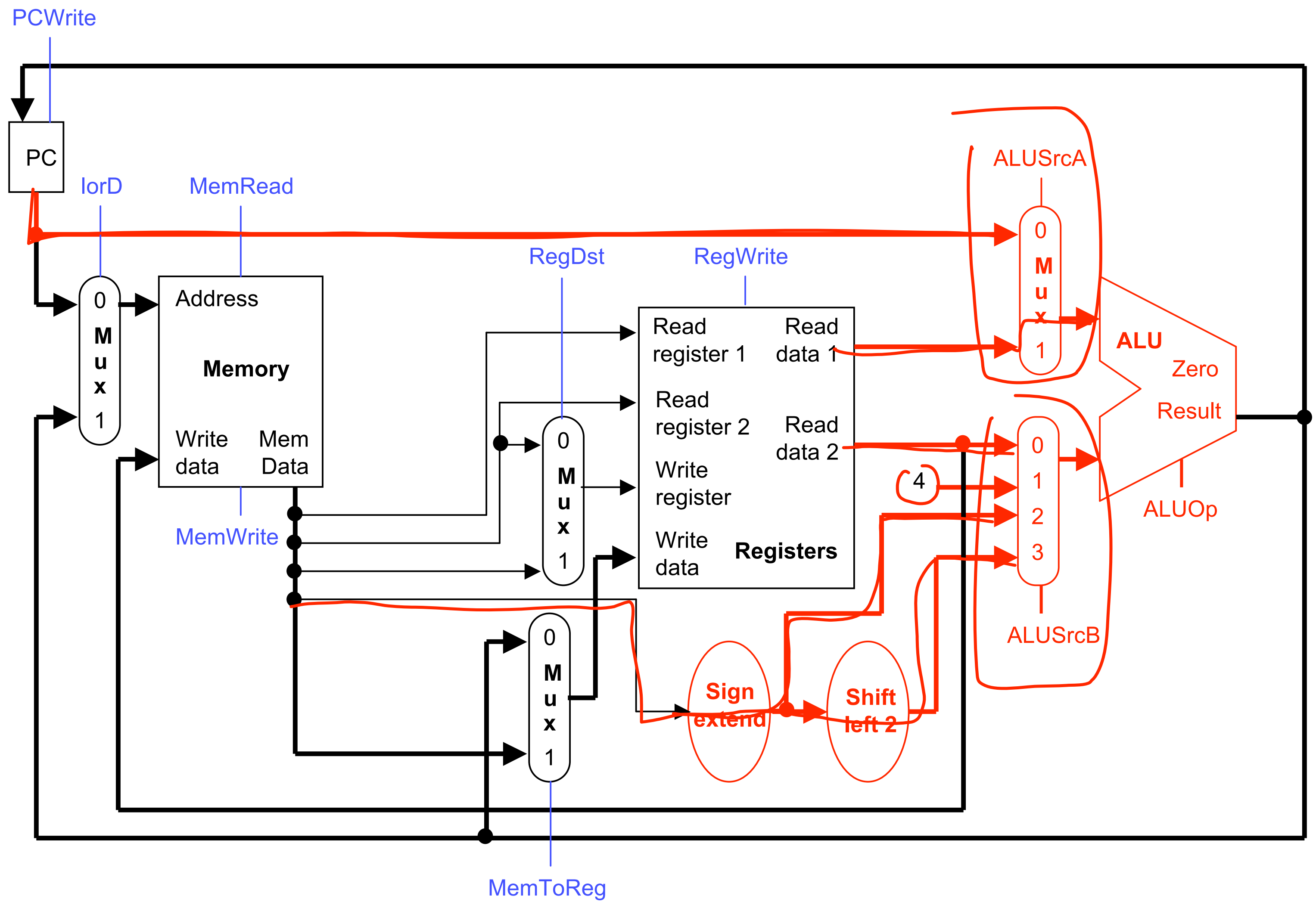
The extra single-cycle adders



Our new adder setup

- We can eliminate *both* extra adders in a multicycle datapath, and instead use just one ALU, with multiplexers to select the proper inputs.
- A 2-to-1 mux ALUSrcA sets the first ALU input to be the PC or a register.
- A 4-to-1 mux ALUSrcB selects the second ALU input from among:
  - the register file (for arithmetic operations),
  - a constant 4 (to increment the PC),
  - a sign-extended constant (for effective addresses), and
  - a sign-extended and shifted constant (for branch targets).
- This permits a single ALU to perform all of the necessary functions.
  - Arithmetic operations on two register operands.
  - Incrementing the PC.
  - Computing effective addresses for lw and sw.
  - Adding a sign-extended, shifted offset to (PC + 4) for branches.

The multicycle adder setup highlighted



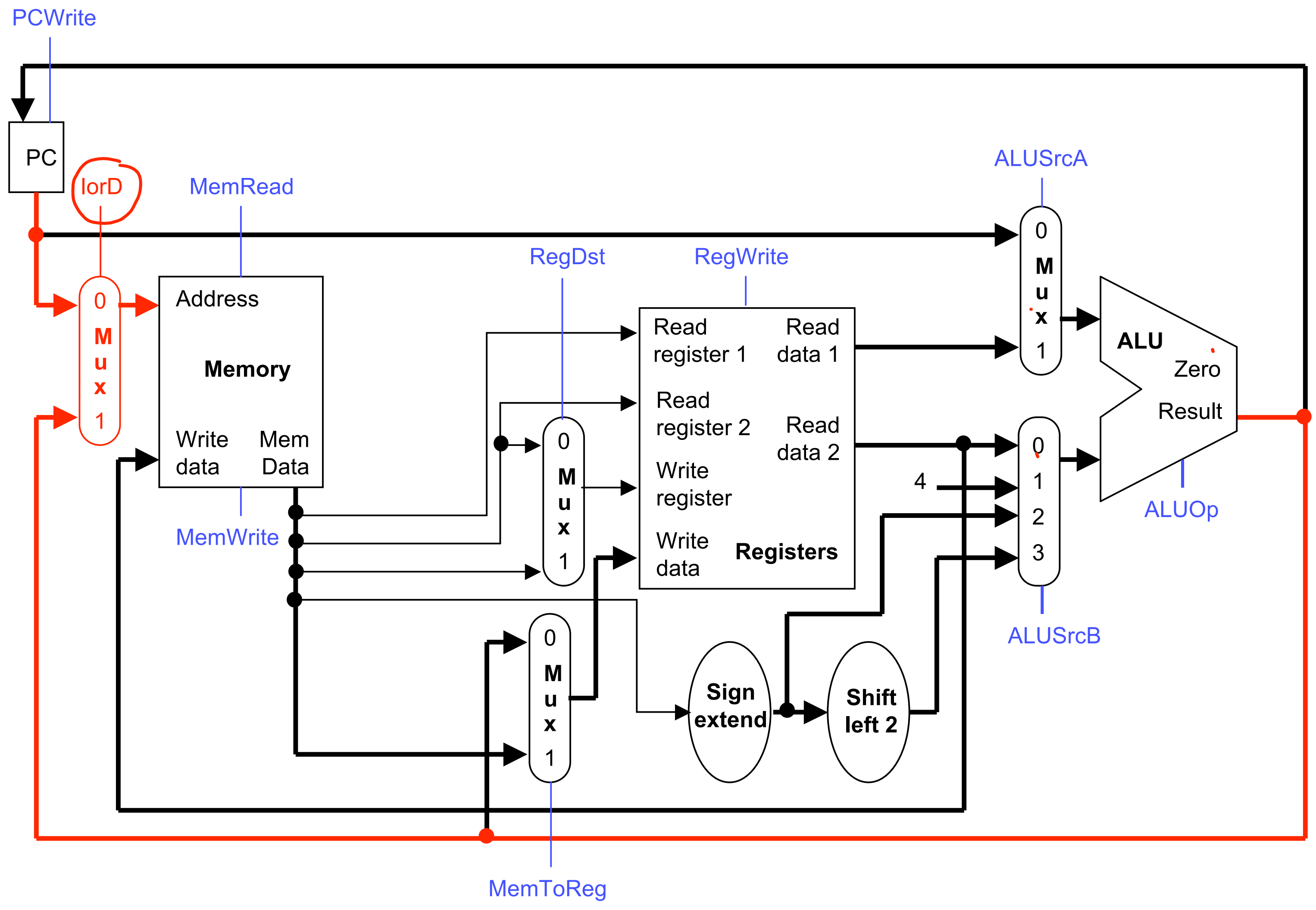
Eliminating a memory

- Similarly, we can get by with one unified memory, which will store *both* program instructions *and* data. (a Princeton architecture)
- This memory is used in both the instruction fetch and data access stages, and the address could come from either:
  - the PC register (when we're fetching an instruction), or
  - the ALU output (for the effective address of a lw or sw).
- We add another 2-to-1 mux, IorD, to decide whether the memory is being accessed for instructions or for data.

Proposed execution stages

1. Instruction fetch and PC increment ✓
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory ✓
5. Storing data back to the register file

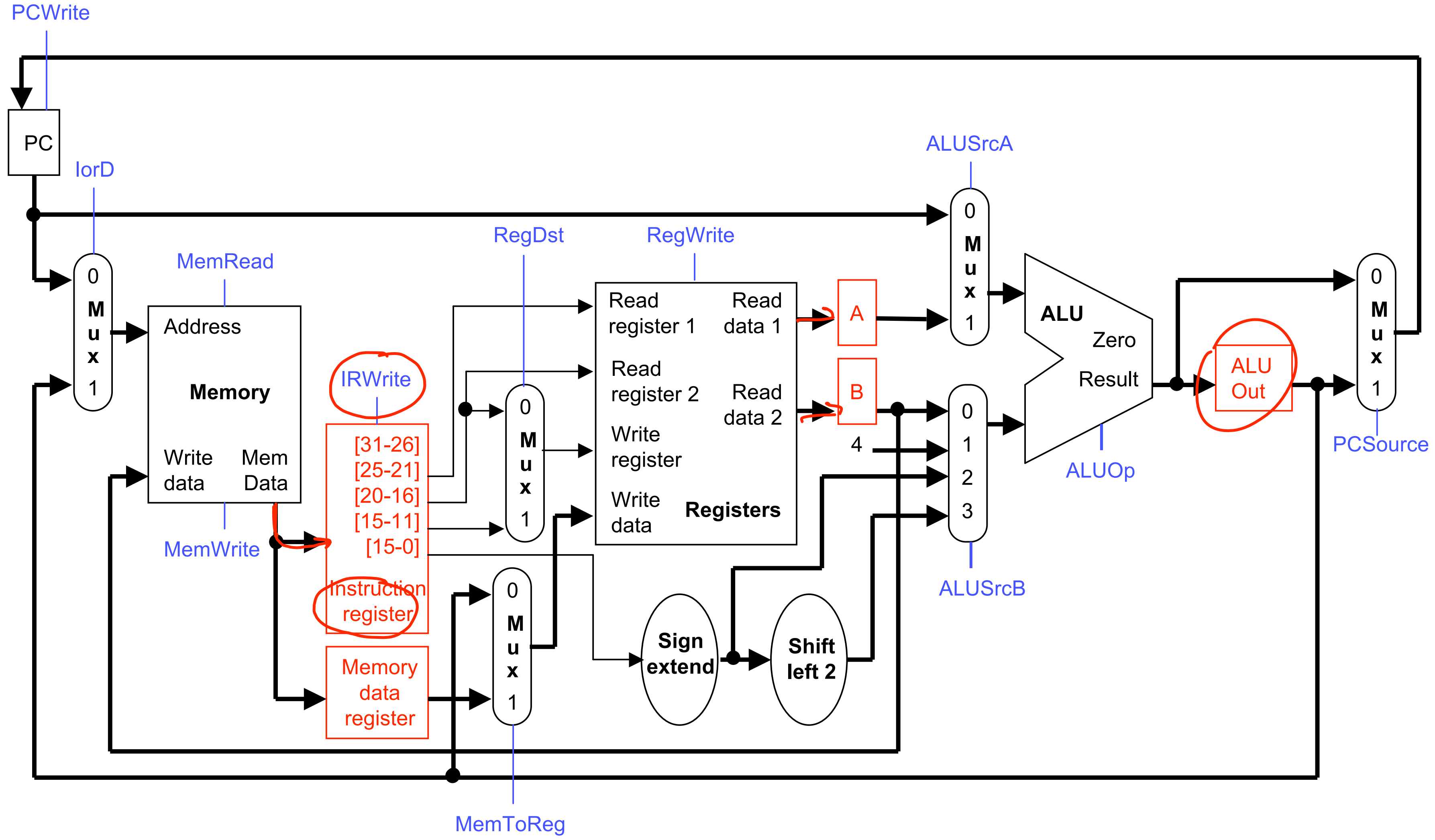
The new memory setup highlighted



Intermediate registers

- Sometimes we need the output of a functional unit in a later clock cycle during the execution of one instruction.
  - The instruction word fetched in stage 1 determines the destination of the register write in stage 5.
  - The ALU result for an address computation in stage 3 is needed as the memory address for lw or sw in stage 4.
- These outputs will have to be stored in intermediate registers for future use. Otherwise they would probably be lost by the next clock cycle.
  - The instruction read in stage 1 is saved in Instruction register.
  - Register file outputs from stage 2 are saved in registers A and B.
  - The ALU output will be stored in a register ALUOut.
  - Any data fetched from memory in stage 4 is kept in the Memory data register, also called MDR.

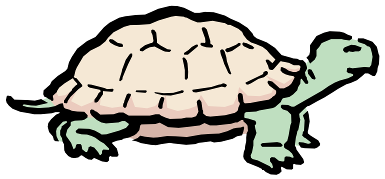
The final multicycle datapath



Register write control signals

- We have to add a few more control signals to the datapath.
- Since instructions now take a variable number of cycles to execute, we cannot update the PC on each cycle.
  - Instead, a PCWrite signal controls the loading of the PC.
  - The instruction register also has a write signal, IRWrite. We need to keep the instruction word for the duration of its execution, and must explicitly re-load the instruction register when needed.
- The other intermediate registers, MDR, A, B and ALUOut, will store data for only one clock cycle at most, and do not need write control signals.

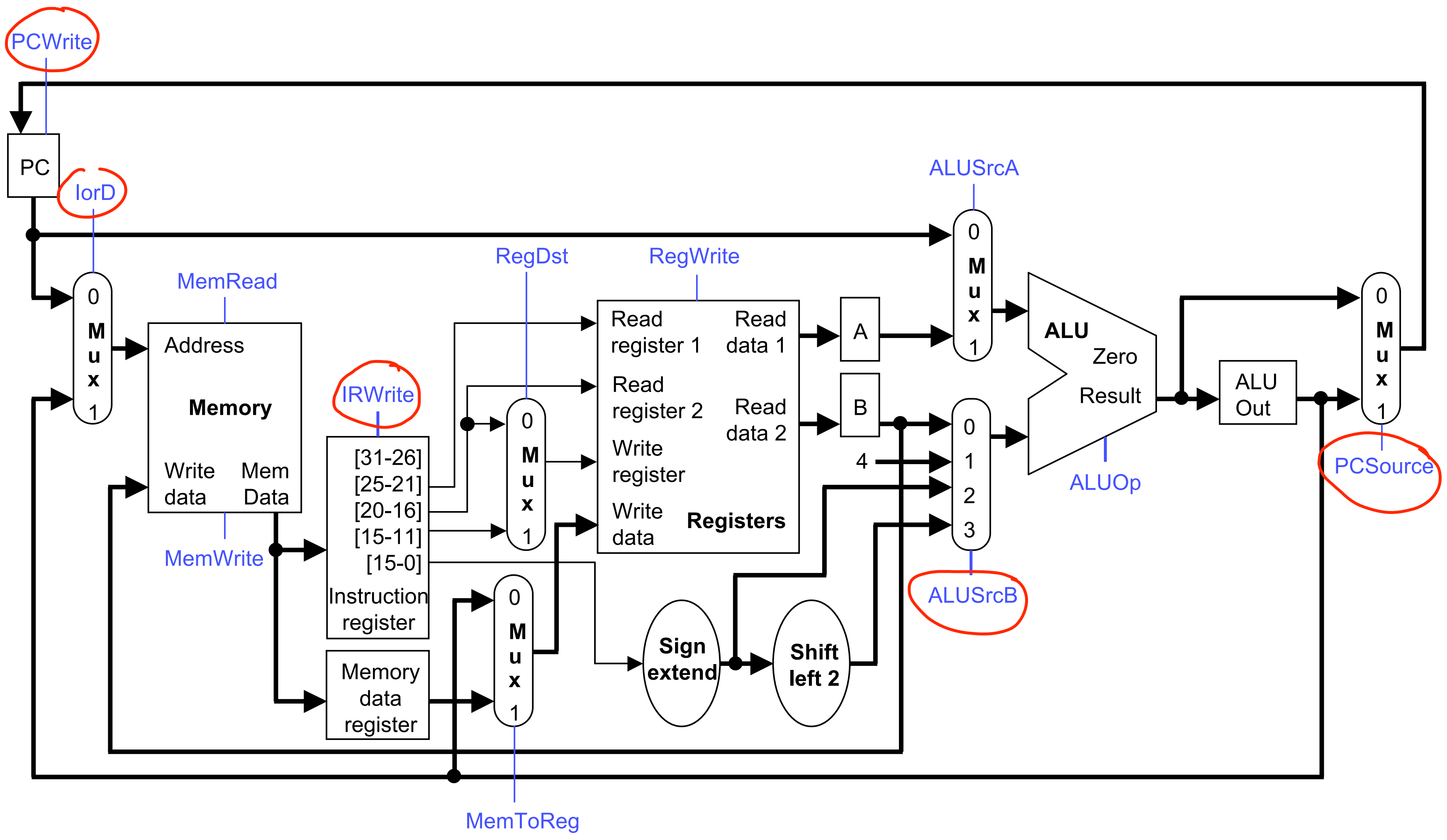
Summary of Multicycle Datapath



- A single-cycle CPU has two main disadvantages.
  - The cycle time is limited by the worst case latency.
  - It requires more hardware than necessary.
- A multicycle processor splits instruction execution into several stages.
  - Instructions only execute as many stages as required. ✓
  - Each stage is relatively simple, so the clock cycle time is reduced. ✓
  - Functional units can be reused on different cycles. ✓
- We made several modifications to the single-cycle datapath.
  - The two extra adders and one memory were removed.
  - Multiplexers were inserted so the ALU and memory can be used for different purposes in different execution stages.
  - New registers are needed to store intermediate results.
- Next time, we'll look at controlling this datapath.

Controlling the multicycle datapath

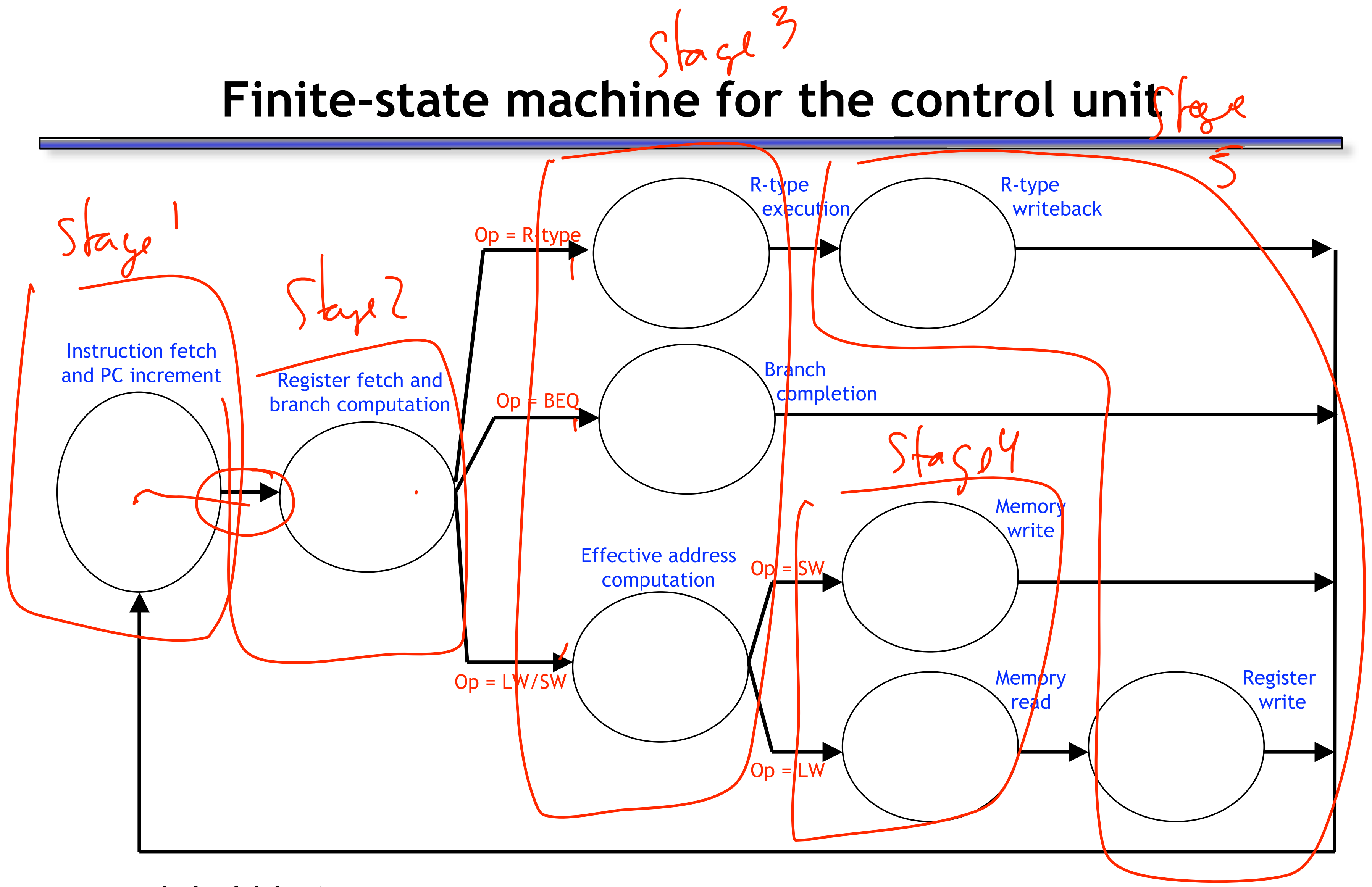
- Now we talk about how to control this datapath.



Multicycle control unit

- The control unit is responsible for producing all of the control signals.
- Each instruction requires a *sequence* of control signals, generated over multiple clock cycles.
  - This implies that we need a state machine.
  - The datapath control signals will be outputs of the state machine.
- Different instructions require different sequences of steps.
  - This implies the instruction word is an input to the state machine.
  - The next state depends upon the exact instruction being executed.
- After we finish executing one instruction, we'll have to repeat the entire process again to execute the next instruction.

Finite-state machine for the control unit



- Each bubble is a state
  - Holds the control signals for a single cycle
  - Note: All instructions do the same things during the first two cycles

Stage 1: Instruction Fetch

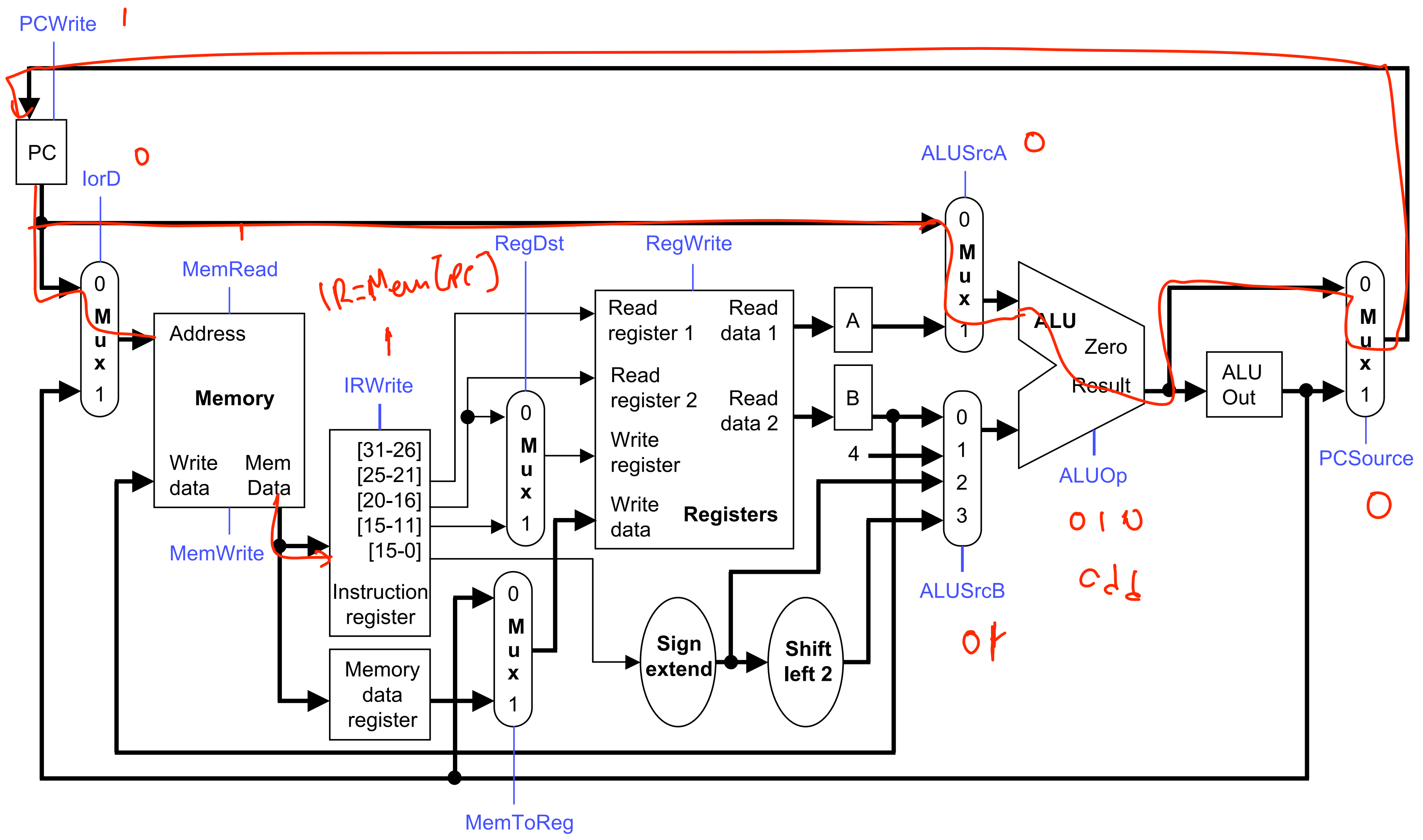
- Stage 1 includes two actions which use two separate functional units: the memory and the ALU.
  - Fetch the instruction from memory and store it in IR.

$$\underline{\text{IR} = \text{Mem[PC]}} \checkmark$$

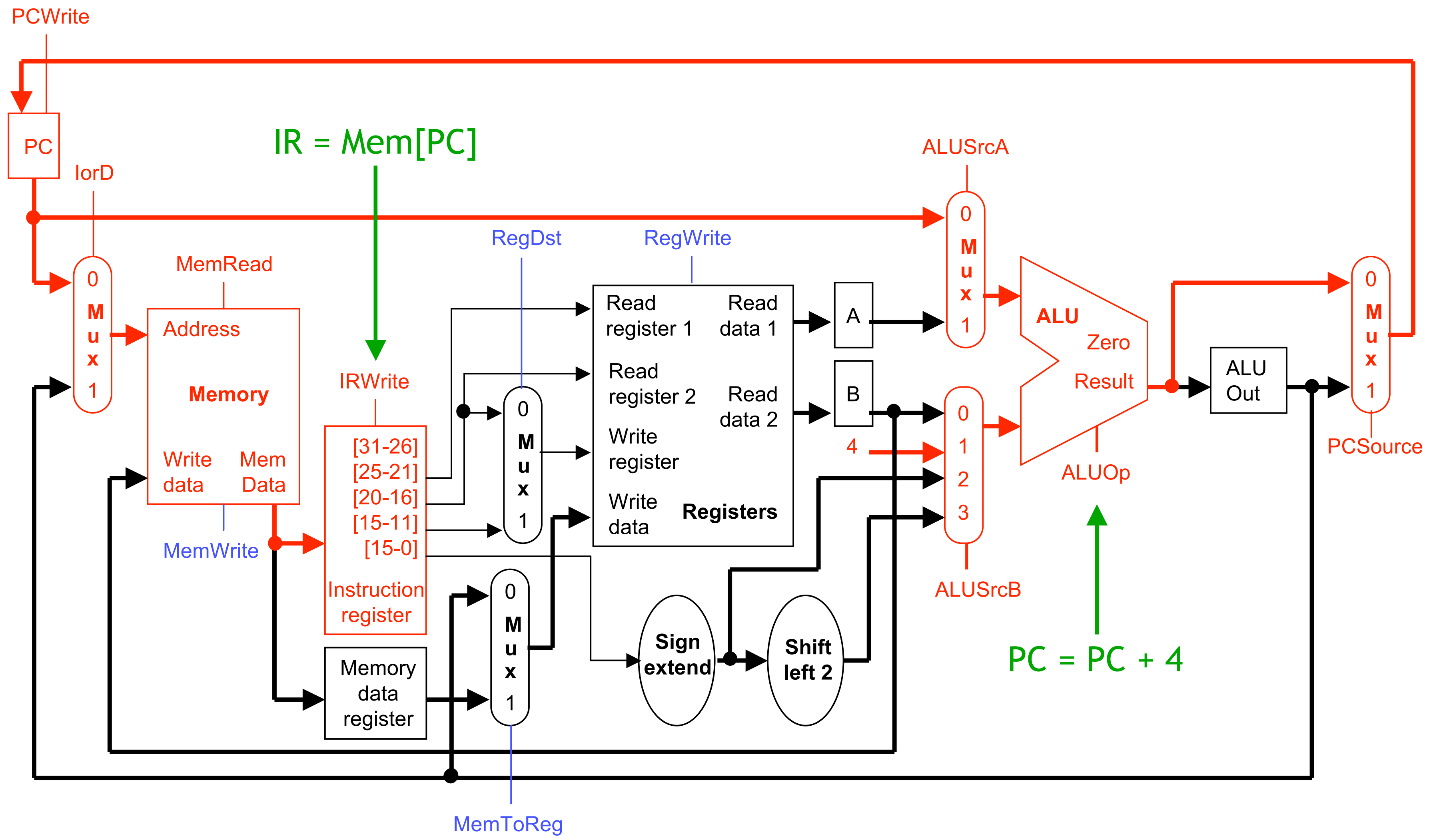
  - Use the ALU to increment the PC by 4.

$$\text{PC} = \text{PC} + 4 \quad \checkmark$$

Stage 1: Instruction Fetch



Stage 1: Instruction fetch and PC increment



Stage 1 control signals

- Instruction fetch: IR = Mem[PC]

Signal	Value	Description
MemRead	1	Read from memory
IorD	0	Use PC as the memory read address
IRWrite	1	Save memory contents to instruction register

- Increment the PC: PC = PC + 4

Signal	Value	Description
ALUSrcA	0	Use PC as the first ALU operand
ALUSrcB	01	Use constant 4 as the second ALU operand
ALUOp	ADD	Perform addition
PCWrite	1	Change PC
PCSource	0	Update PC from the ALU output

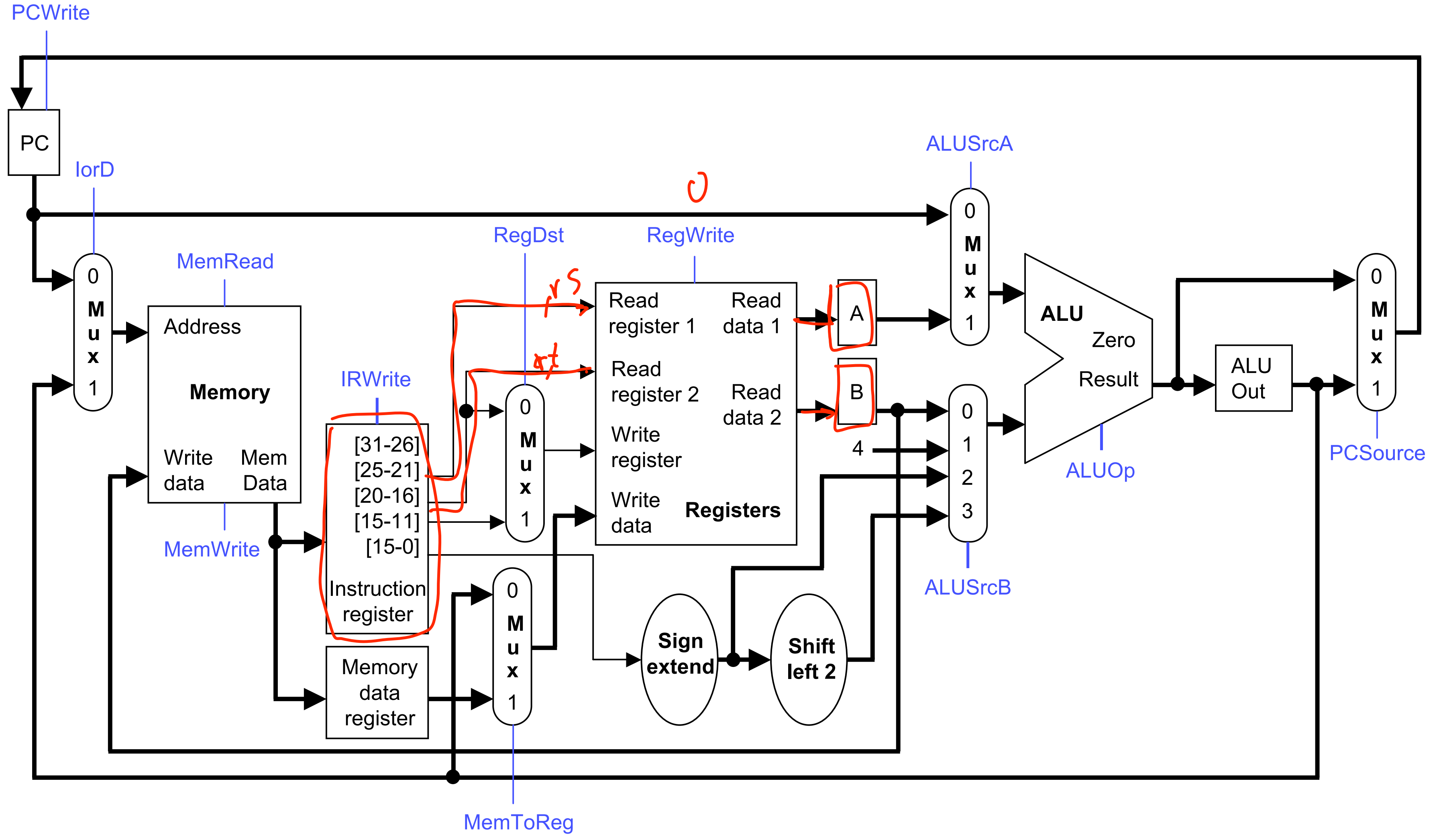
- We'll assume that all control signals not listed are implicitly set to 0.

Stage 2: Read registers

- Stage 2 is much simpler.
  - Read the contents of source registers rs and rt, and store them in the intermediate registers A and B. (Remember the rs and rt fields come from the instruction register IR.)

A = Reg[IR[25-21]]
B = Reg[IR[20-16]]

Stage 2: Register File Read



Stage 2 control signals

- No control signals need to be set for the register reading operations A = Reg[IR[25-21]] and B = Reg[IR[20-16]].
  - IR[25-21] and IR[20-16] are already applied to the register file.
  - Registers A and B are already written on every clock cycle.

Executing Arithmetic Instructions: Stages 3 & 4

- We'll start with R-type instructions like add $t1, $t1, $t2.
- Stage 3 for an arithmetic instruction is simply ALU computation.

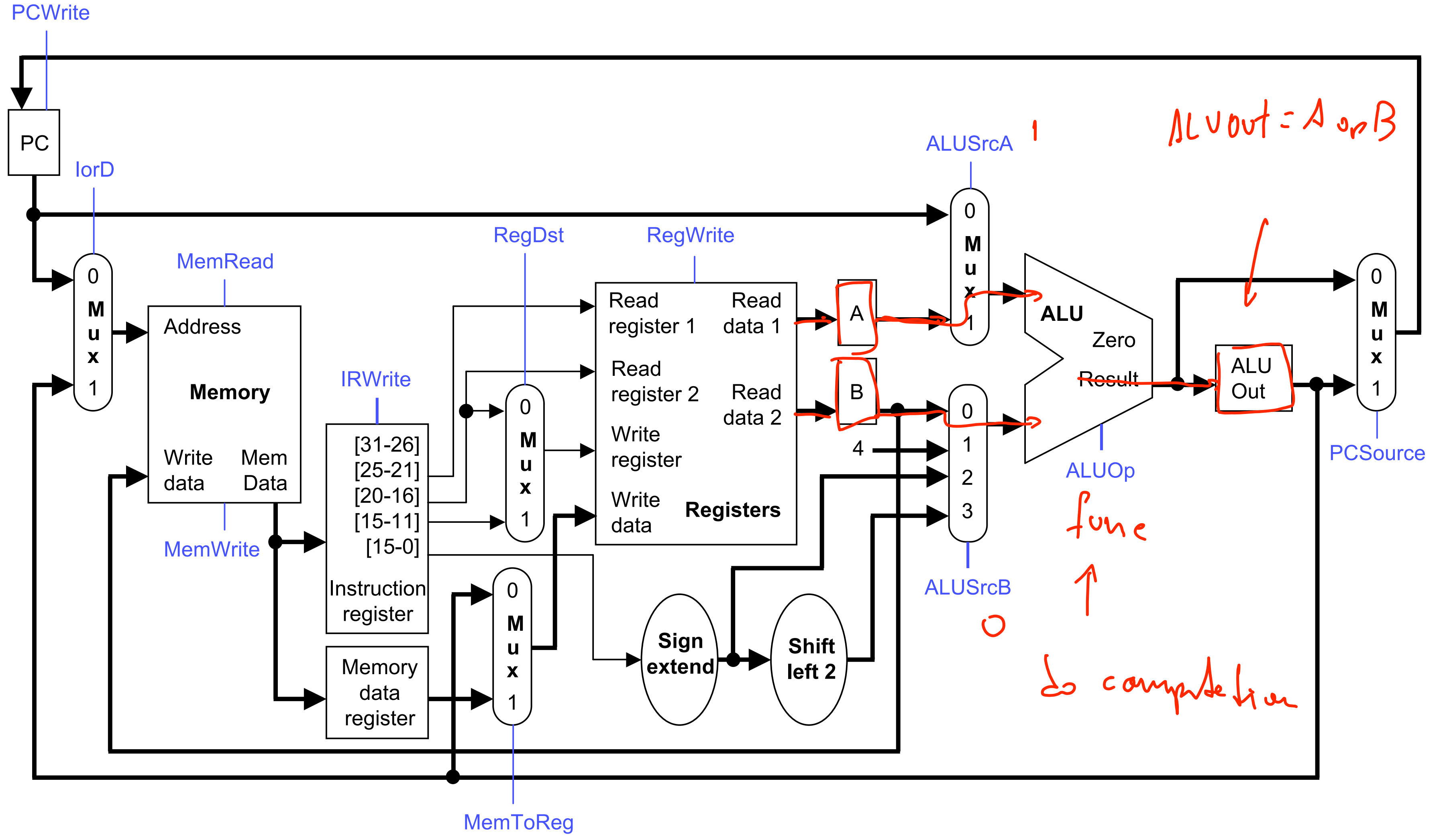
ALUOut = A op B

  - A and B are the intermediate registers holding the source operands.
  - The ALU operation is determined by the instruction's "func" field and could be one of add, sub, and, or, slt.

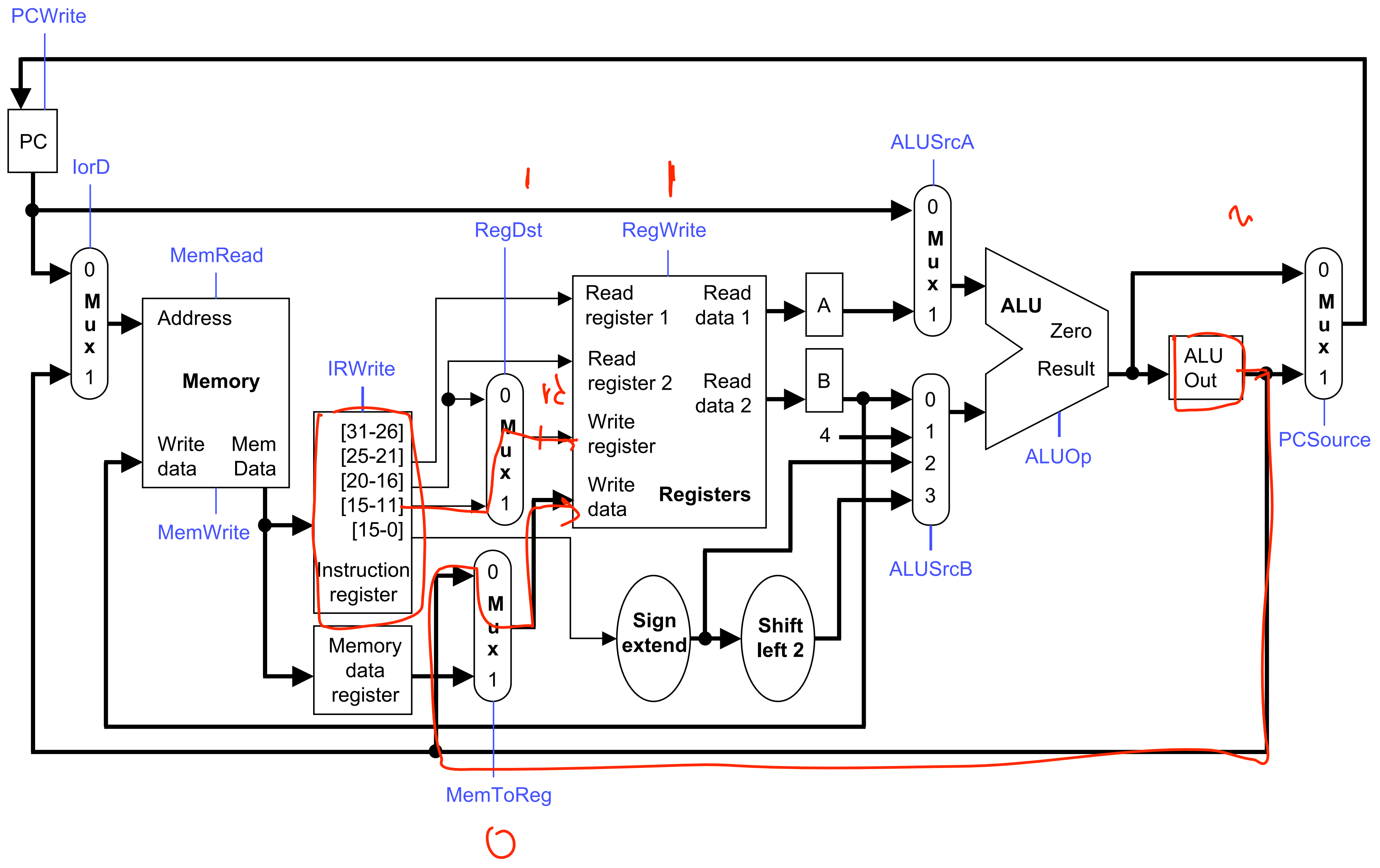
- Stage 4, the final R-type stage, is to store the ALU result generated in the *previous* cycle into the destination register rd.

Reg[IR[15-11]] = ALUOut

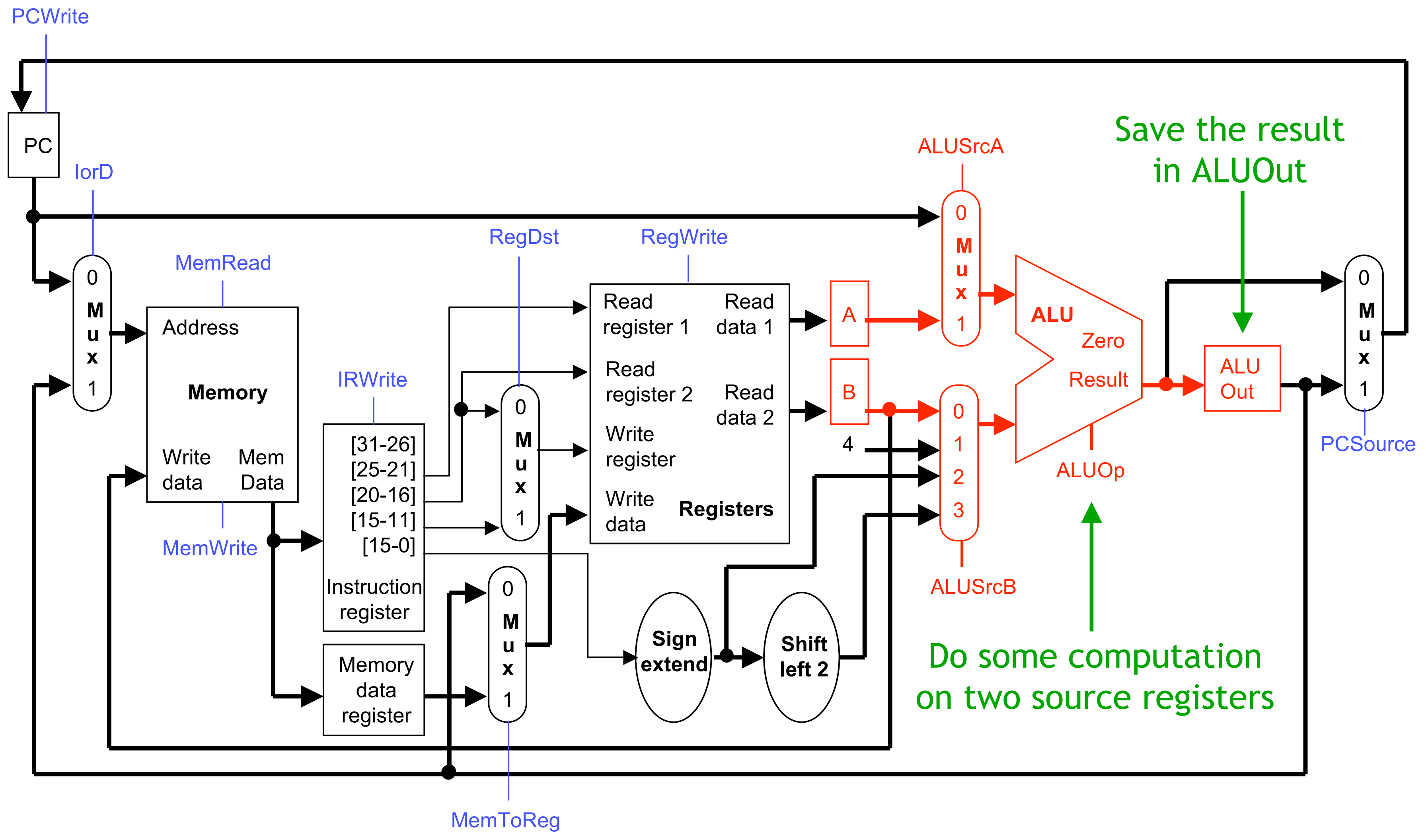
Stage 3 (R-Type): ALU operation



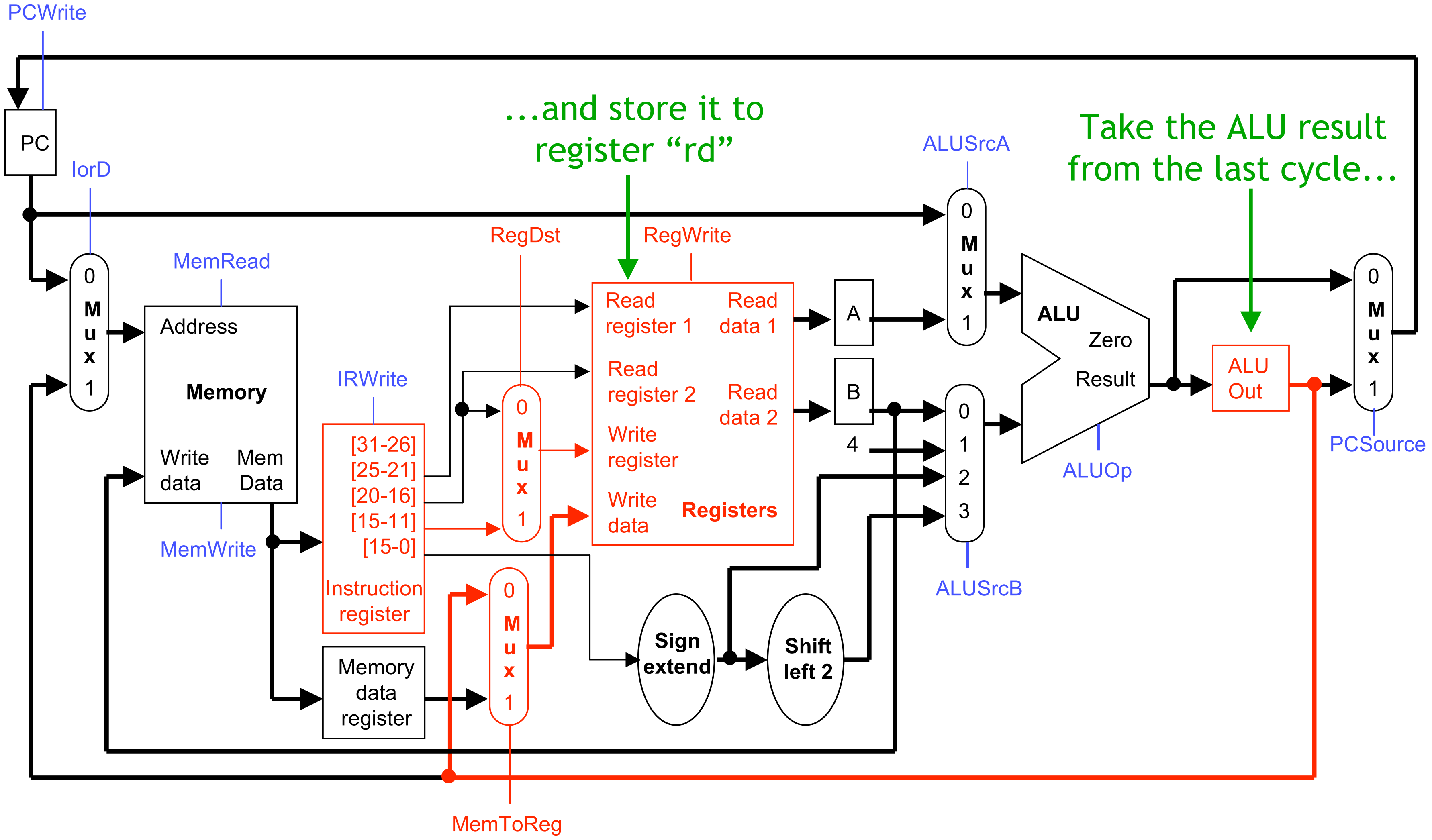
Stage 4 (R-Type): Register Writeback



Stage 3 (R-type): instruction execution



Stage 4 (R-type): write back



Stages 3-4 (R-type) control signals

- Stage 3 (execution): ALUOut = A op B

Signal	Value	Description
ALUSrcA	1	Use A as the first ALU operand
ALUSrcB	00	Use B as the second ALU operand
ALUOp	func	Do the operation specified in the “func” field

- Stage 4 (writeback): Reg[IR[15-11]] = ALUOut

Signal	Value	Description
RegWrite	1	Write to the register file
RegDst	1	Use field rd as the destination register
MemToReg	0	ALUOut contains the data to write

Executing a beq instruction

- We can execute a branch instruction in three stages or clock cycles.
  - But it requires a little cleverness...
  - Stage 1 involves instruction fetch and PC increment.

```
IR = Mem[PC]
PC = PC + 4
```

✓

  - Stage 2 is register fetch and branch target computation.

```
A = Reg[IR[25-21]]
B = Reg[IR[20-16]]
```

✓ A - B = 0 ?

  - Stage 3 is the final cycle needed for executing a branch instruction.
    - Assuming we have the branch target available

```
if (A == B) then
  PC = branch_target
```

✓

When should we compute the branch target?

- We need the ALU to do the computation.
  - When is the ALU not busy?

Cycle	ALU
1	PC + 4
2	Sinso !
3	A - B = 0 ?

Ehm

When should we compute the branch target?

- We need the ALU to do the computation.
  - When is the ALU not busy?

Cycle	ALU
1	PC = PC + 4
2	Here
3	Comparing A & B

Optimistic execution

- But, we don't know whether or not the branch is taken in cycle 2!!
- That's okay…. we can still go ahead and compute the branch target first. The book calls this optimistic execution.
  - The ALU is otherwise free during this clock cycle.
  - Nothing is harmed by doing the computation early. If the branch is not taken, we can just ignore the ALU result.
- This idea is also used in more advanced CPU design techniques.
  - Modern CPUs perform branch prediction, which we'll discuss in a few lectures in the context of pipelining.

Stage 2 Revisited: Compute the branch target

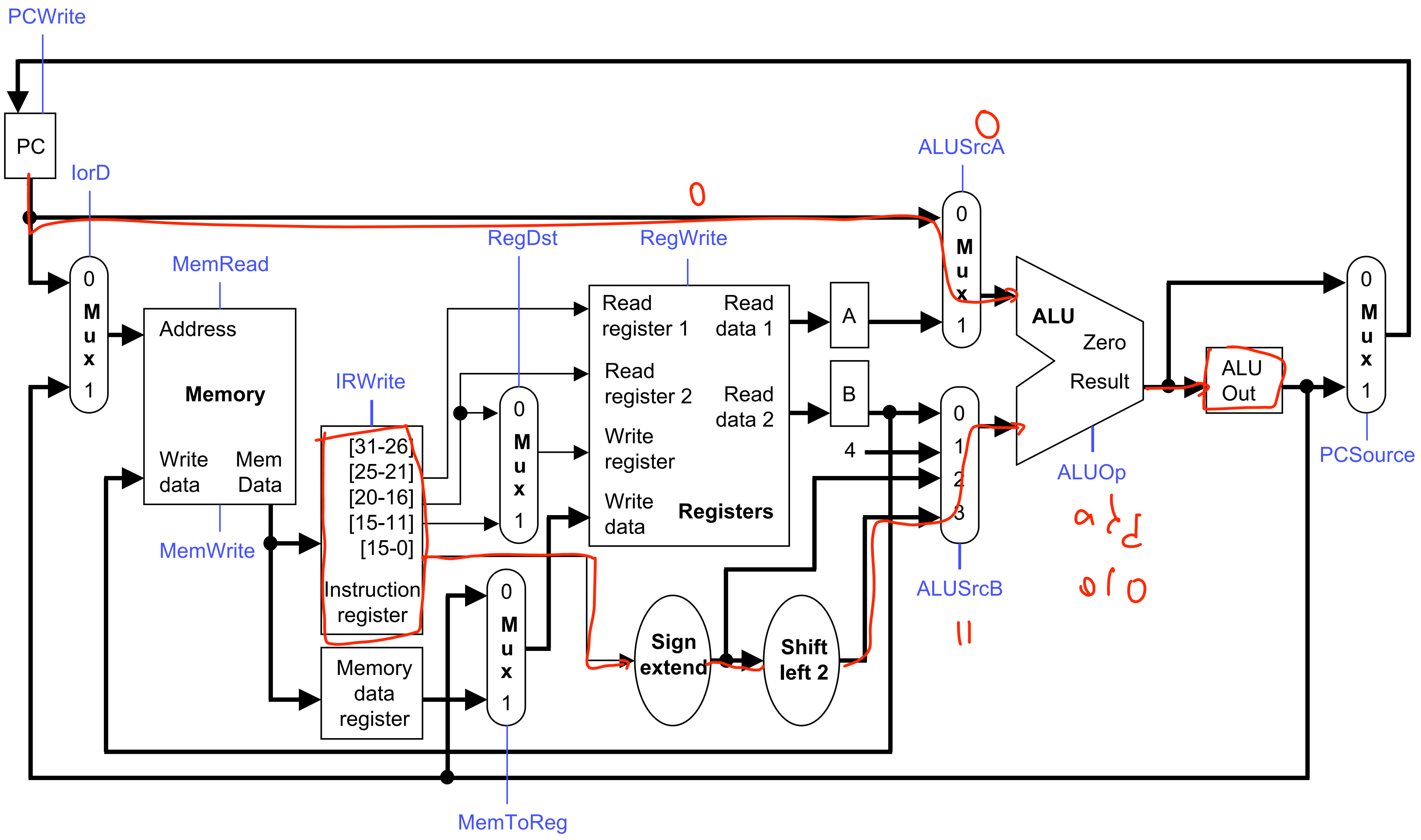
- To Stage 2, we'll add the computation of the branch target.
  - Compute the branch target address by adding the new PC (the original PC + 4) to the sign-extended, shifted constant from IR.

    ALUOut = PC + (sign-extend(IR[15-0]) << 2)

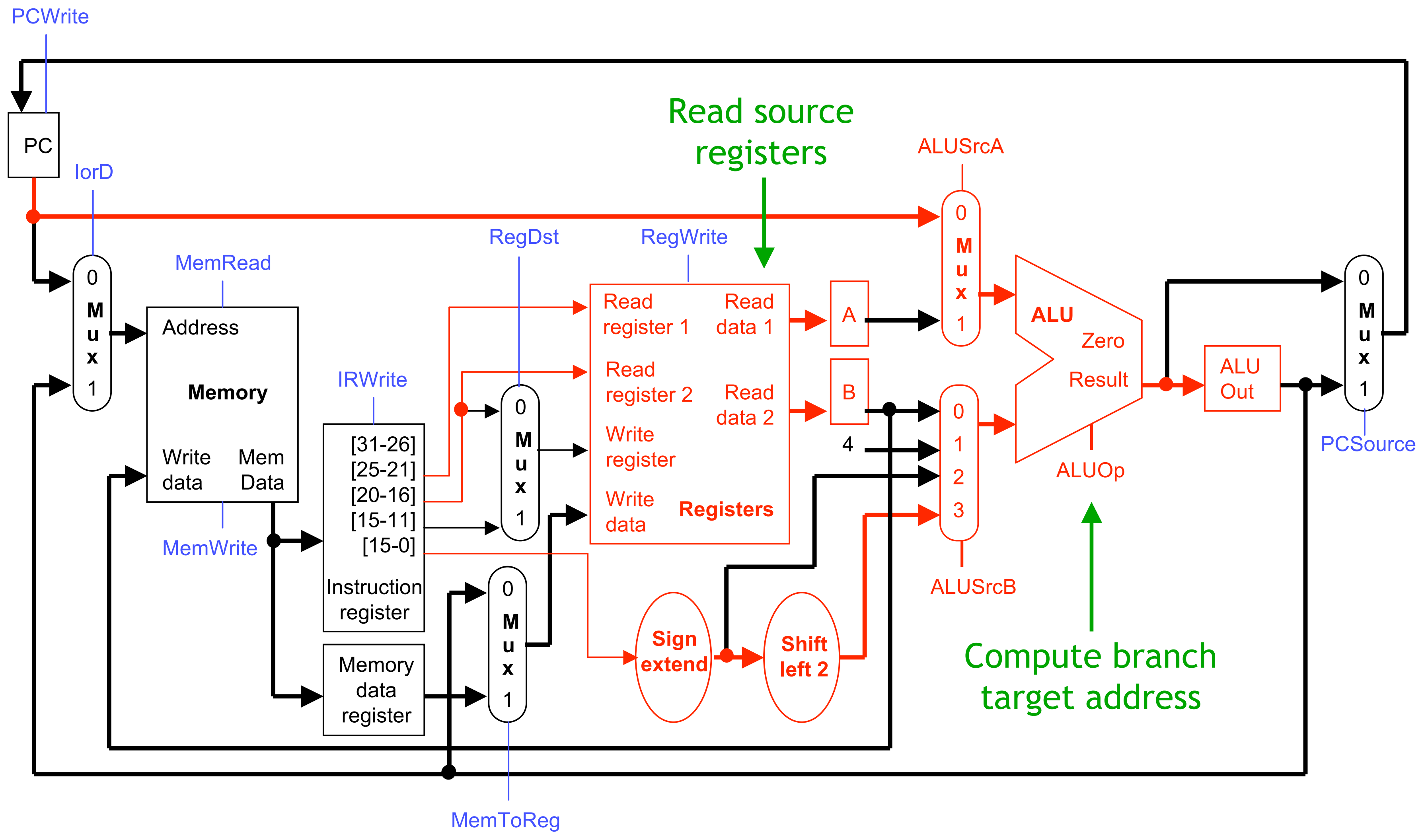
    We save the target address in ALUOut for now, since we don't know yet if the branch should be taken.

  - What about R-type instructions that always go to PC+4 ?

Stage 2 (Revisited): Branch Target Computation



Stage 2: Register fetch & branch target computation



Stage 2 control signals

- No control signals need to be set for the register reading operations A = Reg[IR[25-21]] and B = Reg[IR[20-16]].
  - IR[25-21] and IR[20-16] are already applied to the register file.
  - Registers A and B are already written on every clock cycle.

- Branch target computation: ALUOut = PC + (sign-extend(IR[15-0]) << 2)

Signal	Value	Description
ALUSrcA	0	Use PC as the first ALU operand
ALUSrcB	11	Use (sign-extend(IR[15-0]) << 2) as second operand
ALUOp	ADD	Add and save the result in ALUOut

ALUOut is also written automatically on each clock cycle.

Branch completion

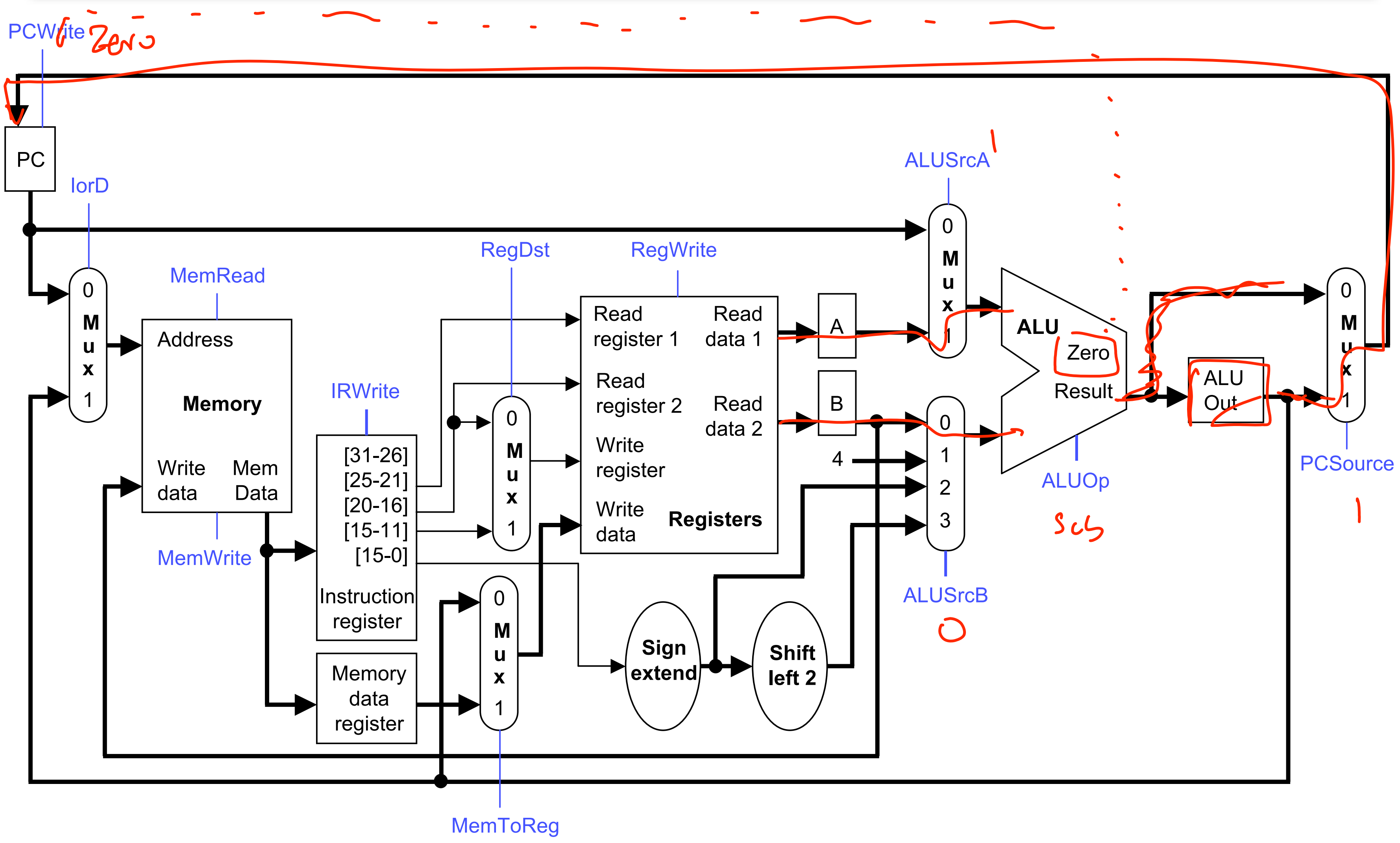
- Stage 3 is the final cycle needed for executing a branch instruction.

```
if (A == B) then
     PC = ALUOut
```

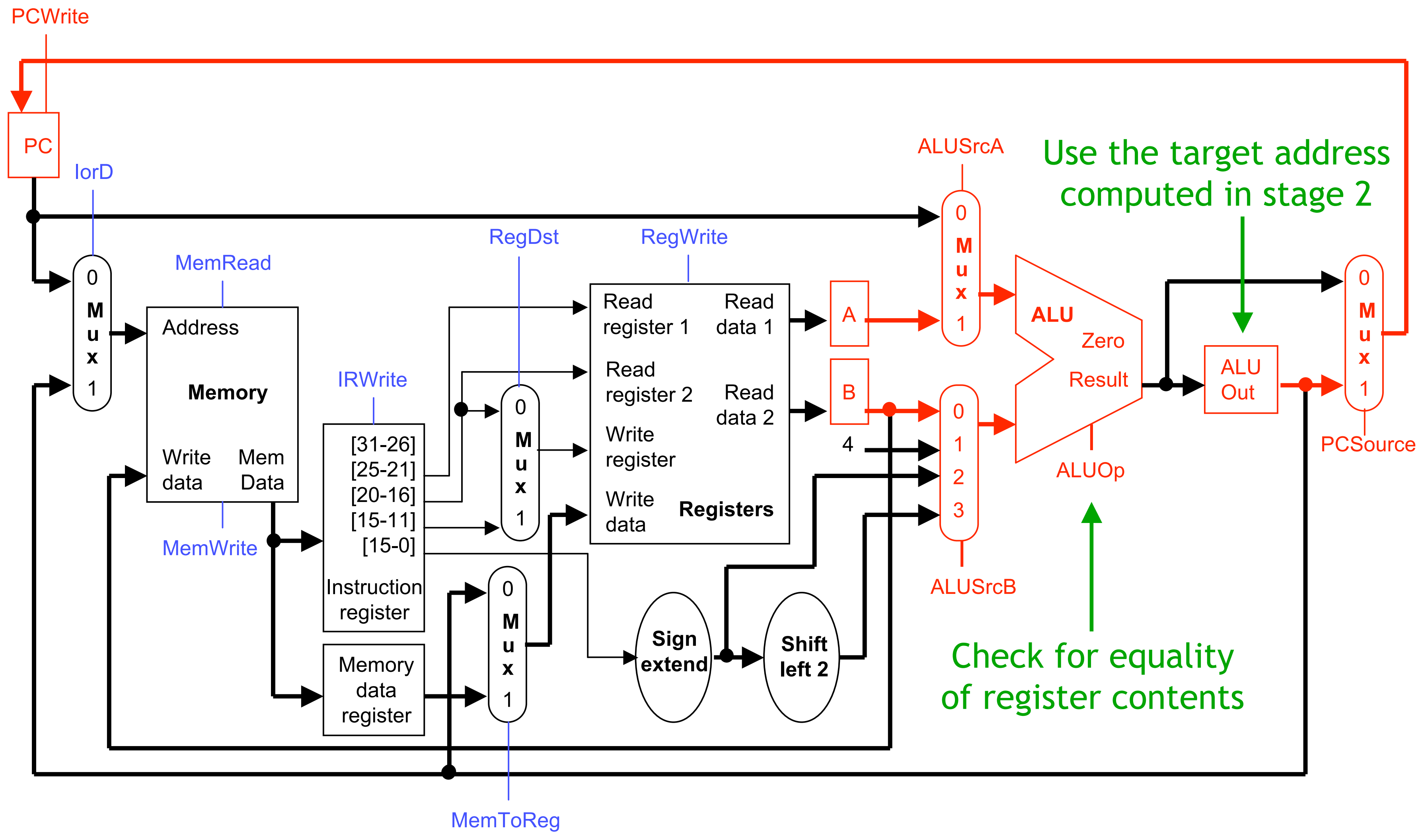
- Remember that A and B are compared by subtracting and testing for a result of 0, so we must use the ALU again in this stage.

(not d!)

Stage 3 (BEQ): Branch Completion



Stage 3 (beq): Branch completion



Stage 3 (beq) control signals

- Comparison: if (A == B) ...

Signal	Value	Description
ALUSrcA	1	Use A as the first ALU operand
ALUSrcB	00	Use B as the second ALU operand
ALUOp	SUB	Subtract, so Zero will be set if A = B

- Branch: ...then PC = ALUOut

Signal	Value	Description
PCWrite	Zero	Change PC only if Zero is true (i.e., A = B)
PCSource	1	Update PC from the ALUOut register

- ALUOut contains the ALU result from the *previous* cycle, which would be the branch target. We can write that to the PC, even though the ALU is doing something different (comparing A and B) during the *current* cycle.

Executing a sw instruction

- A store instruction, like sw $a0, 16($sp), also shares the same first two stages as the other instructions.
  - ✓Stage 1: instruction fetch and PC increment.
  - ✓Stage 2: register fetch and branch target computation.

- Stage 3 computes the effective memory address using the ALU.

  ALUOut = A + sign-extend(IR[15-0])

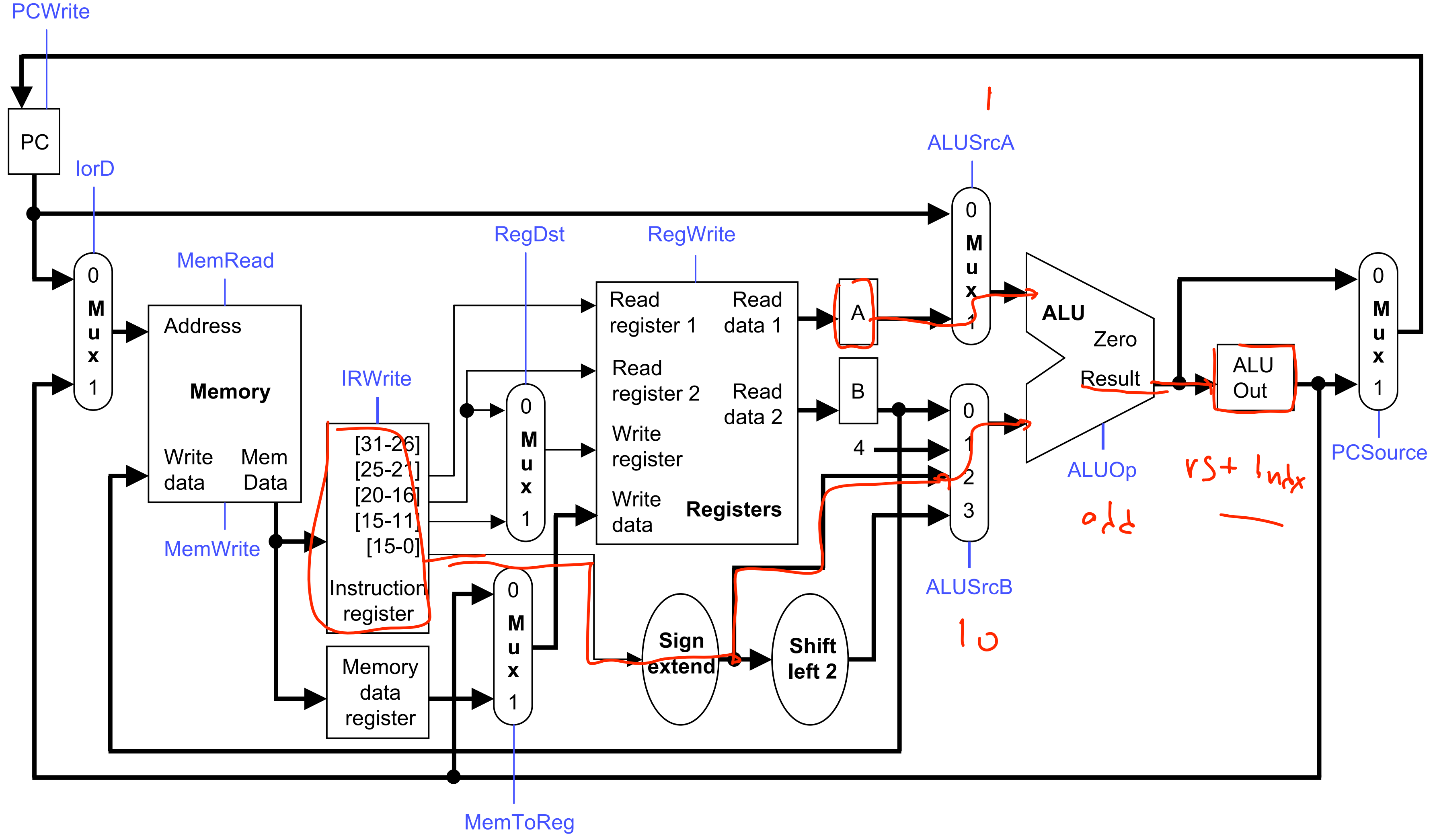
  A contains the base register (like $sp), and IR[15-0] is the 16-bit constant offset from the instruction word, which is *not* shifted.

- Stage 4 saves the register contents (here, $a0) into memory.

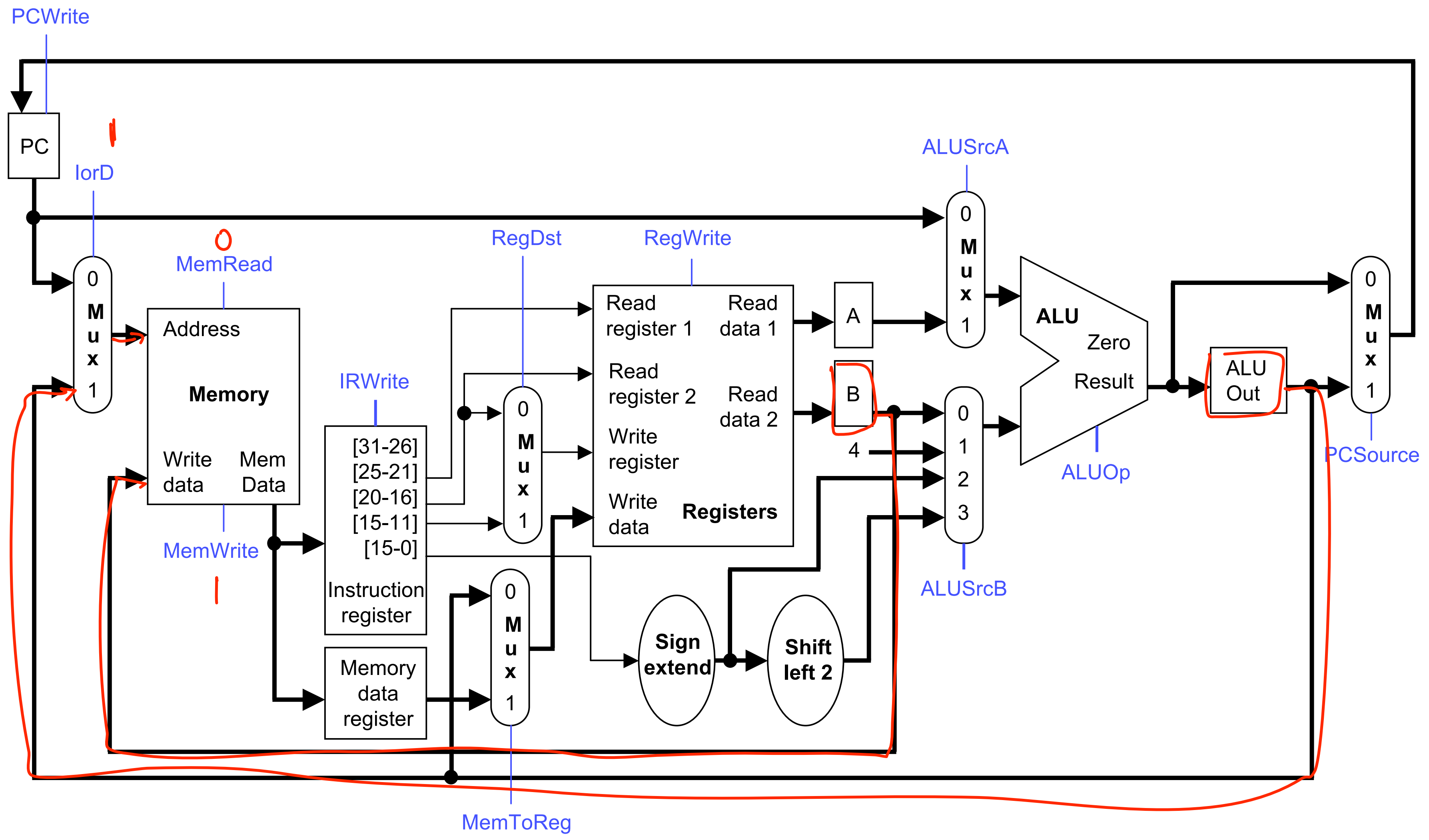
  Mem[ALUOut] = B

  Remember that the second source register rt was already read in Stage 2 (and again in Stage 3), and its contents are in intermediate register B.

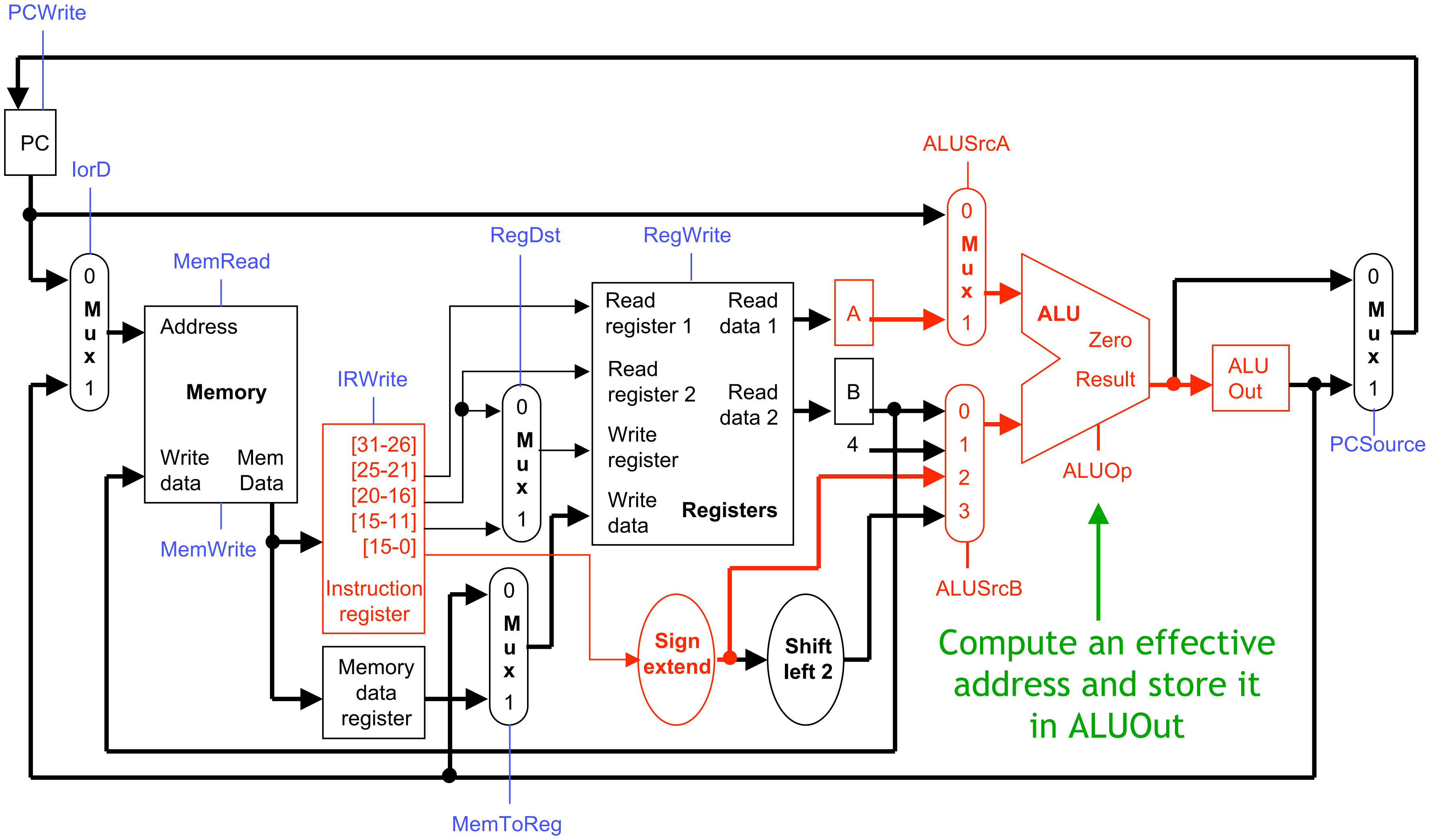
Stage 3 (SW): Effective Address Computation



Stage 4 (SW): Memory Write

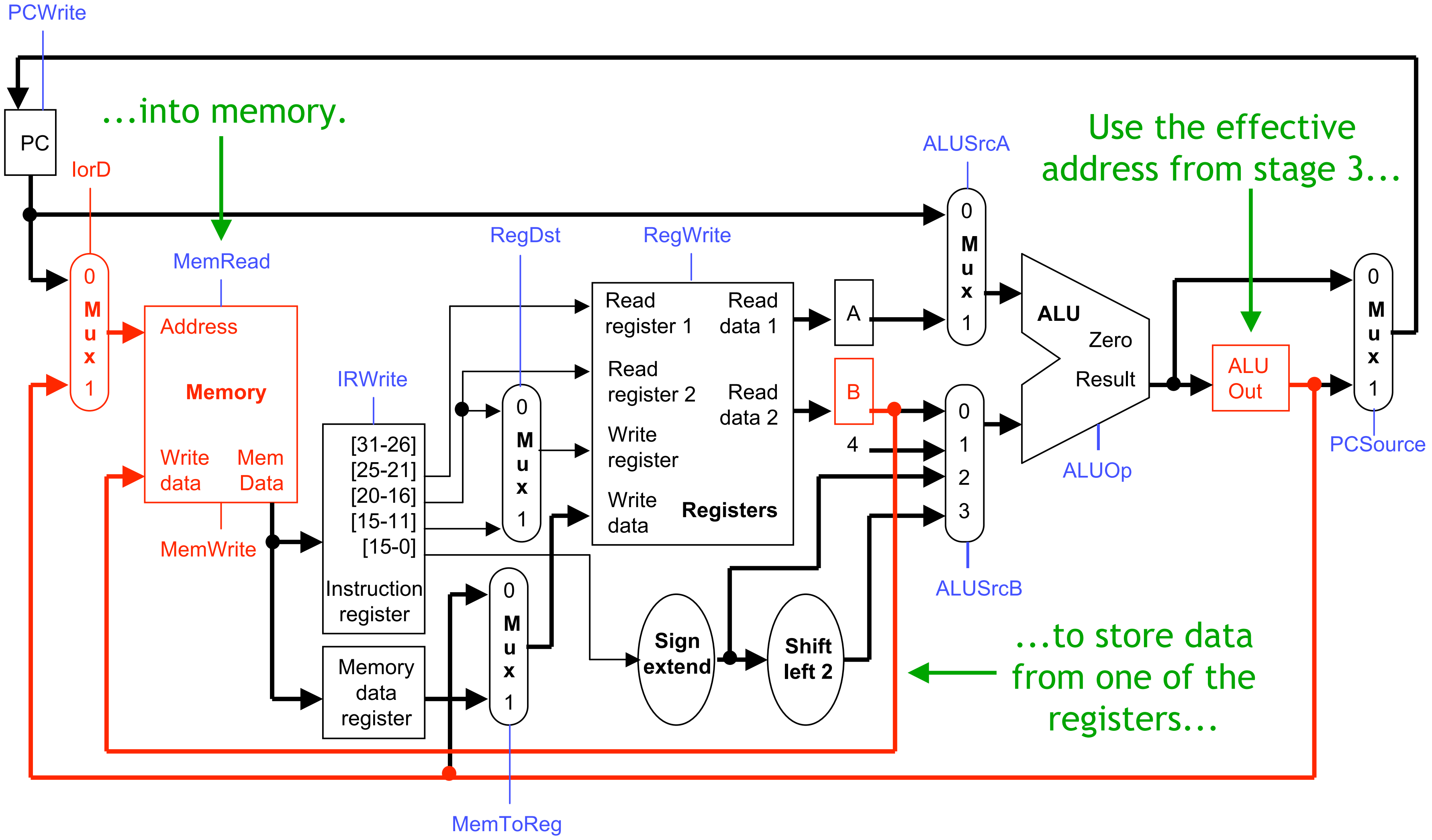


Stage 3 (sw): effective address computation



Compute an effective address and store it in ALUOut

Stage 4 (sw): memory write



Stages 3-4 (sw) control signals

- Stage 3 (address computation): ALUOut = A + sign-extend(IR[15-0])

Signal	Value	Description
ALUSrcA	1	Use A as the first ALU operand
ALUSrcB	10	Use sign-extend(IR[15-0]) as the second operand
ALUOp	010	Add and store the resulting address in ALUOut

- Stage 4 (memory write): Mem[ALUOut] = B

Signal	Value	Description
MemWrite	1	Write to the memory
IorD	1	Use ALUOut as the memory address

The memory's "Write data" input *always* comes from the B intermediate register, so no selection is needed.

Executing a lw instruction

- Finally, lw is the most complex instruction, requiring five stages.
- The first two are like all the other instructions.
  - Stage 1: instruction fetch and PC increment. ✓
  - Stage 2: register fetch and branch target computation. ✓
- The third stage is the same as for sw, since we have to compute an effective memory address in both cases.
  - Stage 3: compute the effective memory address.

Stages 4-5 (lw): memory read and register write

- Stage 4 is to read from the effective memory address, and to store the value in the intermediate register MDR (memory data register).

  MDR = Mem[ALUOut]

- Stage 5 stores the contents of MDR into the destination register.

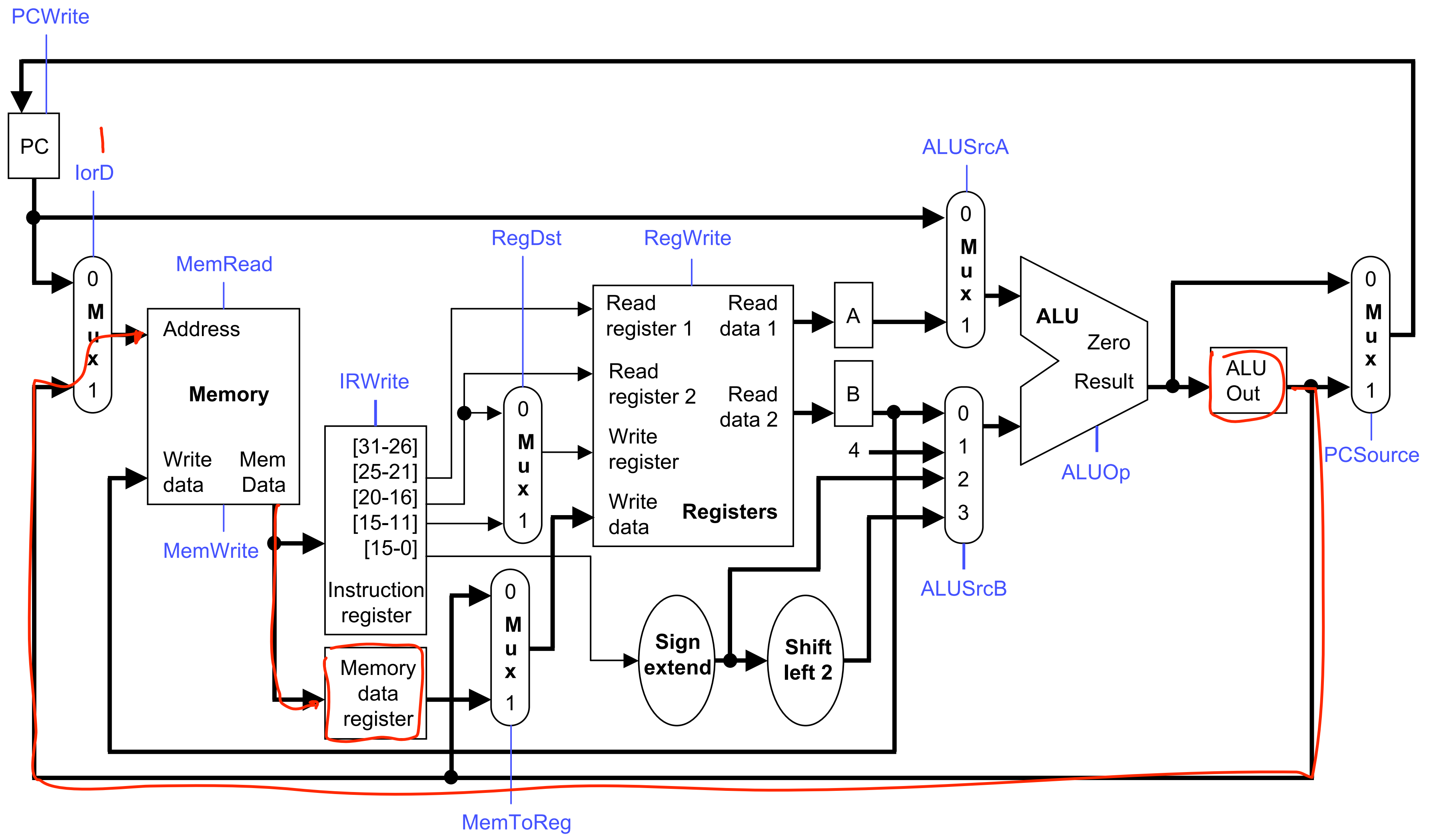
  Reg[IR[20-16]] = MDR

Remember that the destination register for lw is field rt (bits 20-16) and *not* field rd (bits 15-11).

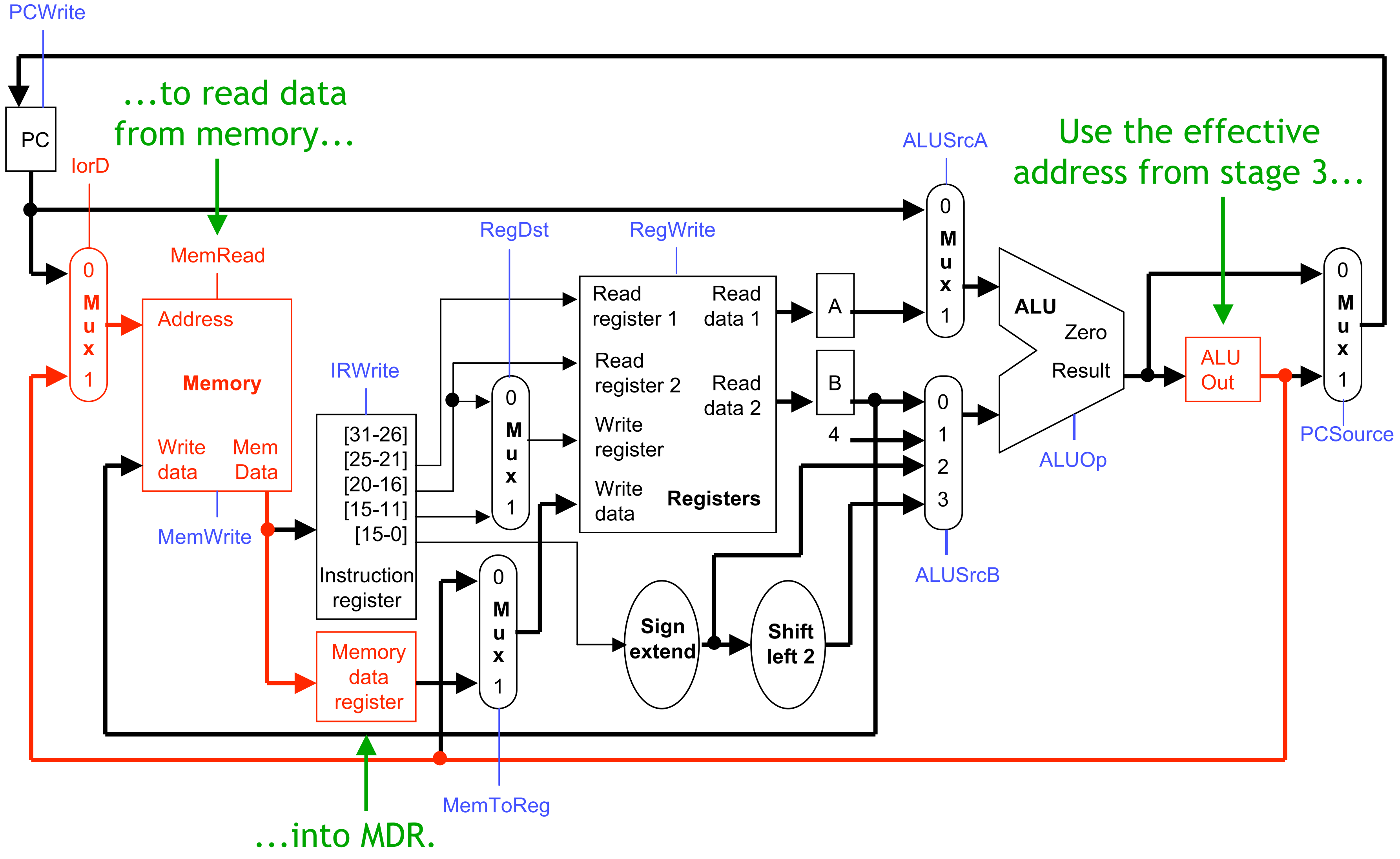
$\nearrow$

annoying

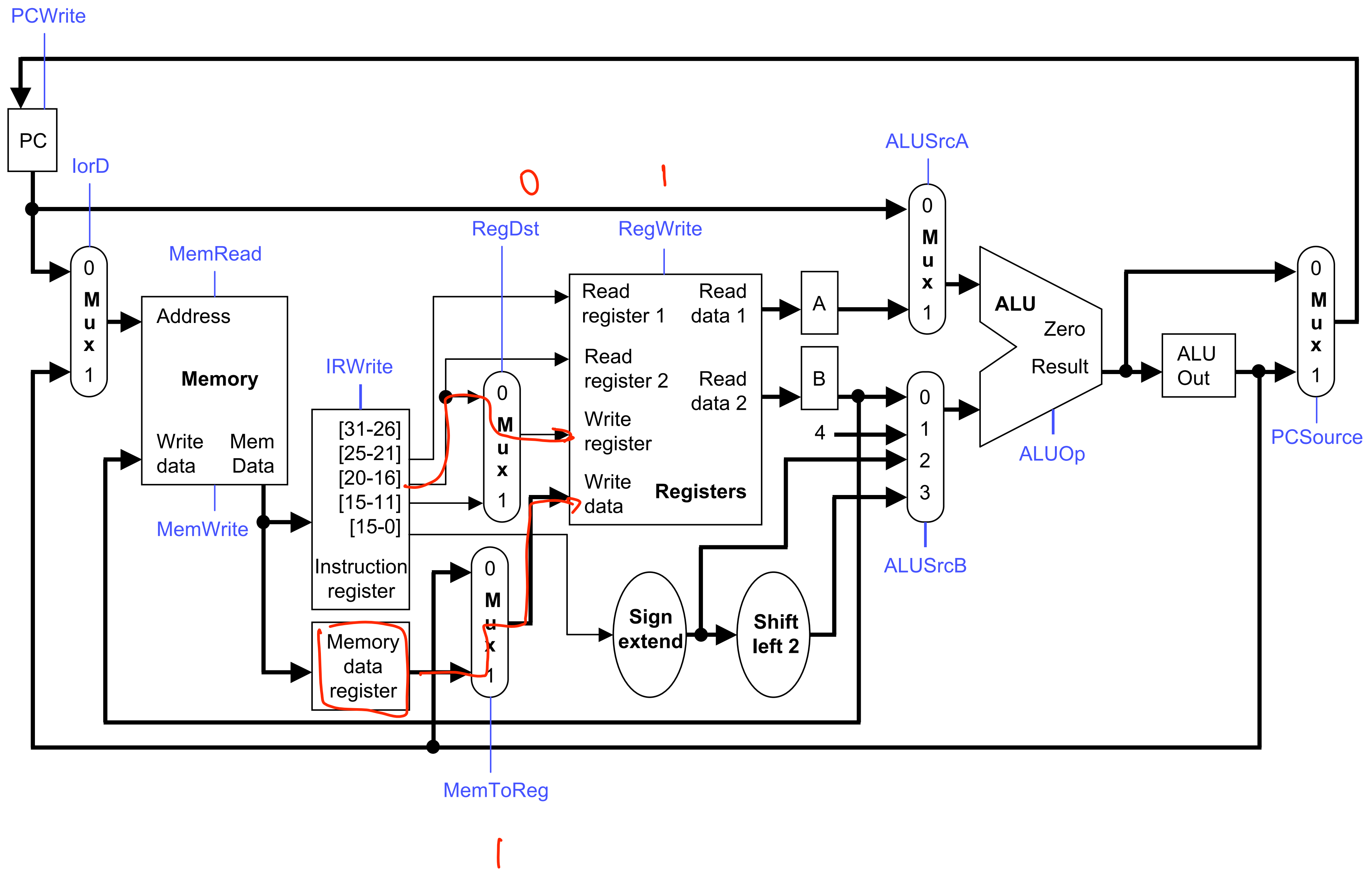
Stage 4 (LW): Memory Read



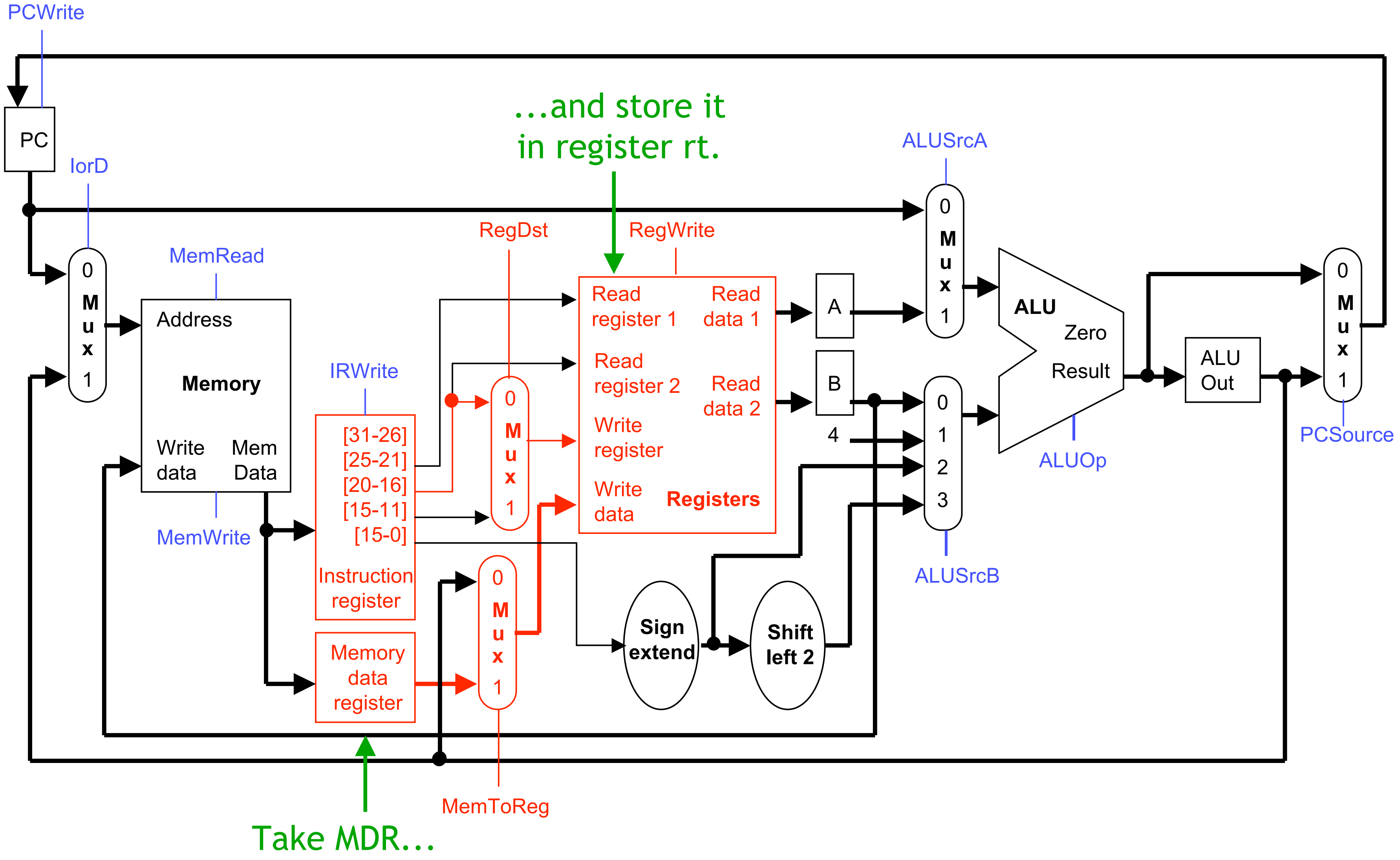
Stage 4 (lw): memory read



Stage 5 (LW): Register Writeback



Stage 5 (lw): register write



Stages 4-5 (lw) control signals

- Stage 4 (memory read): MDR = Mem[ALUOut]

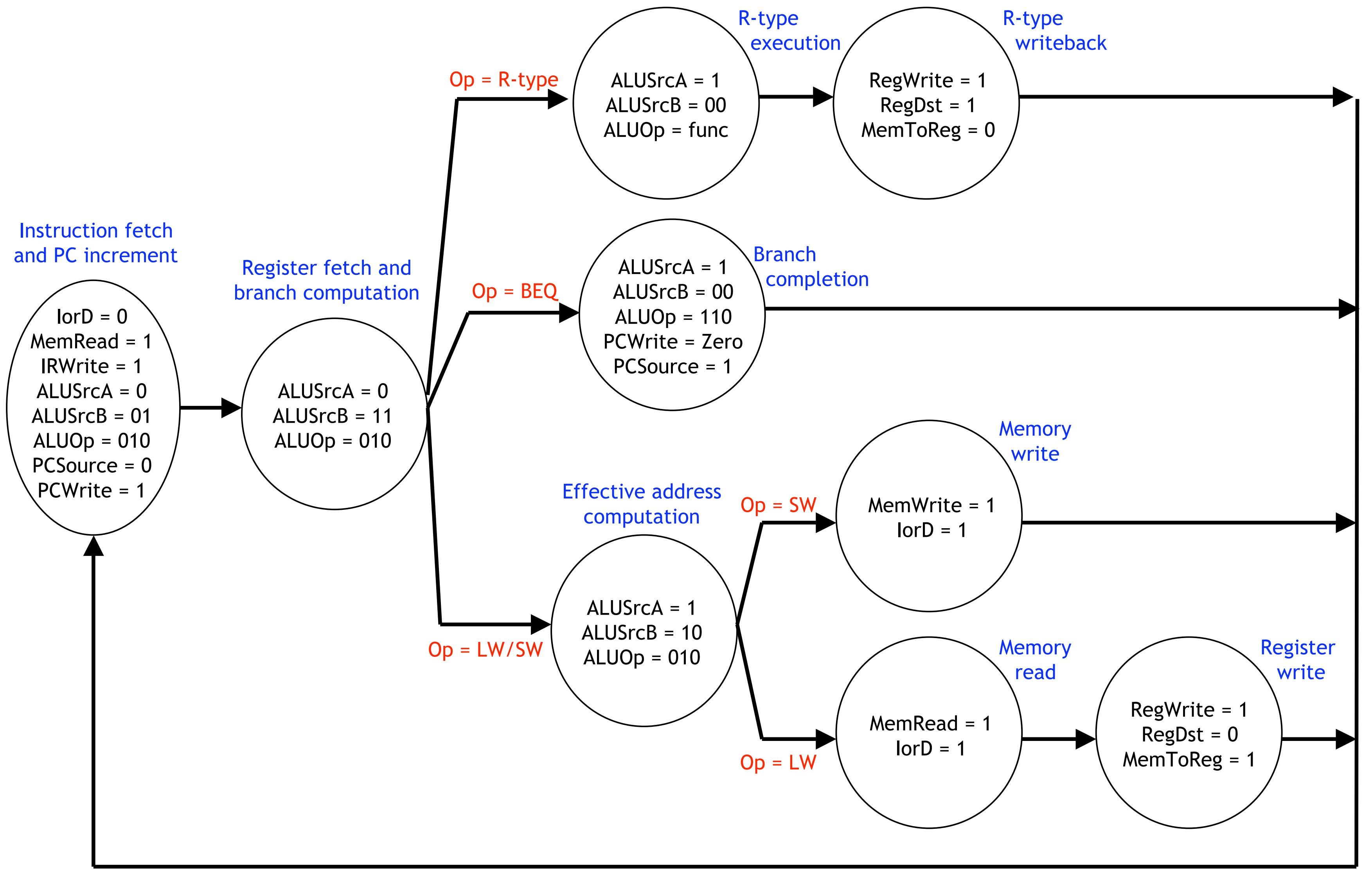
Signal	Value	Description
MemRead	1	Read from memory
IorD	1	Use ALUOut as the memory address

The memory contents will be automatically written to MDR.

- Stage 5 (writeback): Reg[IR[20-16]] = MDR

Signal	Value	Description
RegWrite	1	Store new data in the register file
RegDst	0	Use field rt as the destination register
MemToReg	1	Write data from MDR (from memory)

Finite-state machine for the control unit



Implementing the FSM

- This can be translated into a state table; here are the first two states.

Current State	Input (Op)	Next State	Output (Control signals)											
			PC Write	IorD	Mem Read	Mem Write	IR Write	Reg Dst	MemTo Reg	Reg Write	ALU SrcA	ALU SrcB	ALU Op	PC Source
Instr Fetch	X	Reg Fetch	1	0	1	0	1	X	X	0	0	01	010	0
Reg Fetch	BEQ	Branch compl	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	R-type	R-type execute	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	LW/SW	Compute eff addr	0	X	0	0	0	X	X	0	0	11	010	X

- You can implement this the hard way.
  - Represent the current state using flip-flops or a register.
  - Find equations for the next state and (control signal) outputs in terms of the current state and input (instruction word).
- Or you can use the easy way.
  - Stick the whole state table into a memory, like a ROM.
  - This would be much easier, since you don’t have to derive equations.

Summary

- Now you know how to build a multicycle controller!
  - Each instruction takes several cycles to execute.
  - Different instructions require different control signals and a different number of cycles.
  - We have to provide the control signals in the right sequence.