Processor: Multicycle Implementation

Dr. Tao Xie

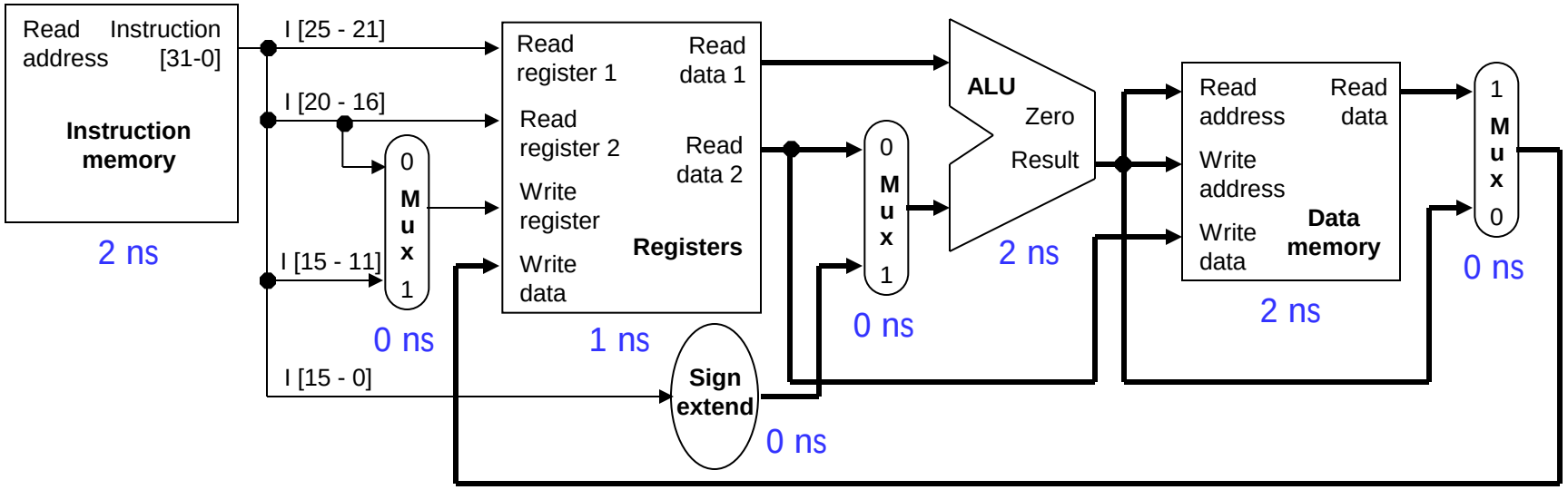
Fall, 2017

These slides are adapted from notes by Dr. David Patterson (UCB)

The slowest instruction...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- Assuming the delays shown here:

Instruction	Time
Arithmetic	2+1+2+1 = 6
Loads	2+1+2+2+1 = 8
Stores	2+1+2+2 = 7
Branches	2+1+2 = 5



How bad is this?

- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns.
- Let's consider the *gcc* benchmark.

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be:

(48% x 6ns) + (22% x 8ns) + (11% x 7ns) + (19% x 5ns) = 6.36ns

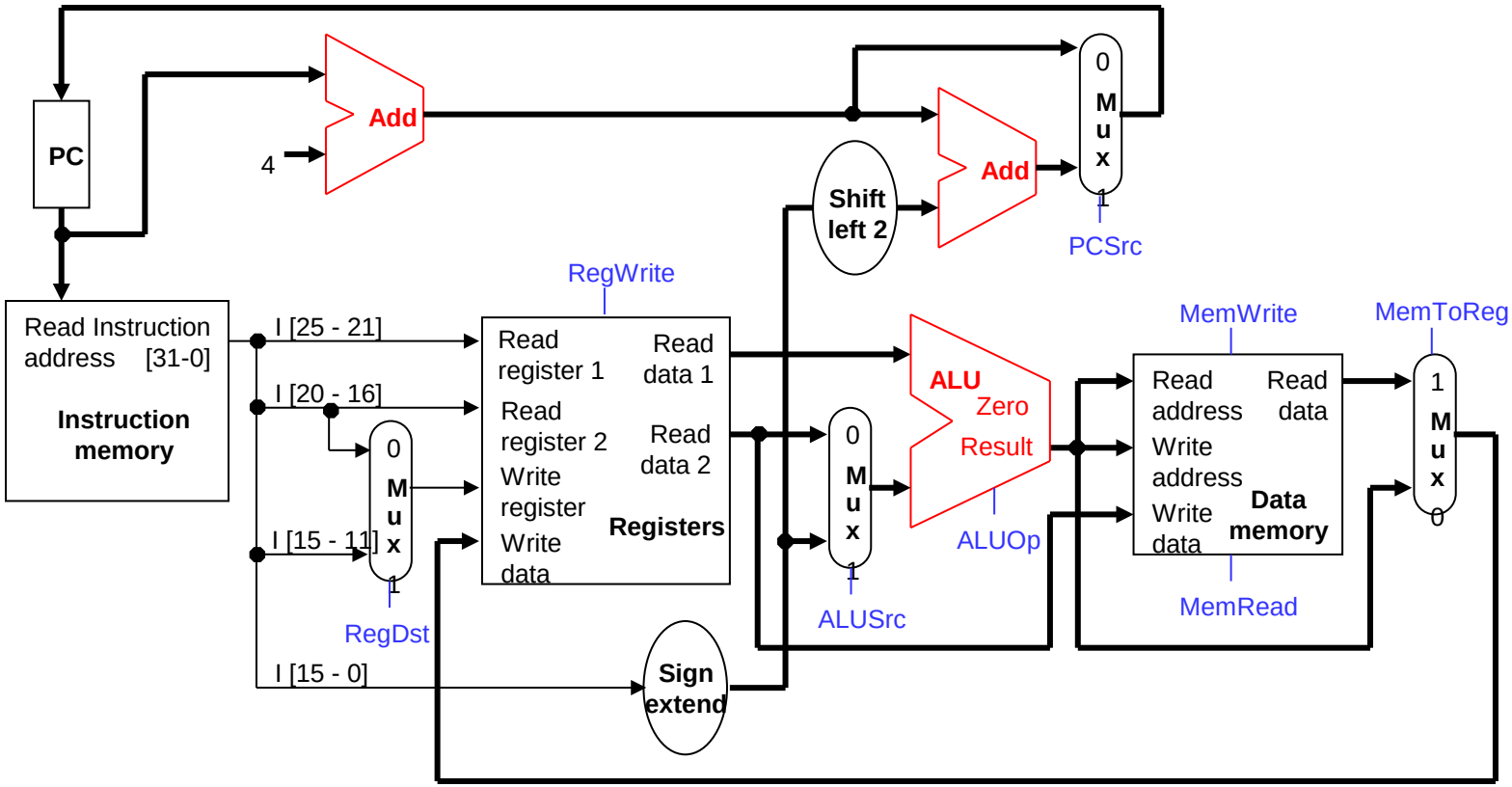
- The single-cycle datapath is about 1.26 times slower!

It gets worse...

- We've made very *optimistic* assumptions about memory latency:
  - Main memory accesses on modern machines is >50ns.
    - For comparison, an ALU on the Pentium4 takes ~0.3ns.
- Our worst case cycle (loads/stores) includes 2 memory accesses
  - A modern single cycle implementation would be stuck at < 10Mhz.
  - Caches will improve common case access time, not worst case

It isn't particularly hardware efficient, either…

- A single-cycle datapath also uses extra hardware—one ALU is not enough, since we must do up to three calculations in one clock cycle for a beq.
- *This used to be a big deal, but now transistors are cheap. Heat issue is more important.*



Goto pp.8

A multistage approach to instruction execution: Key Idea

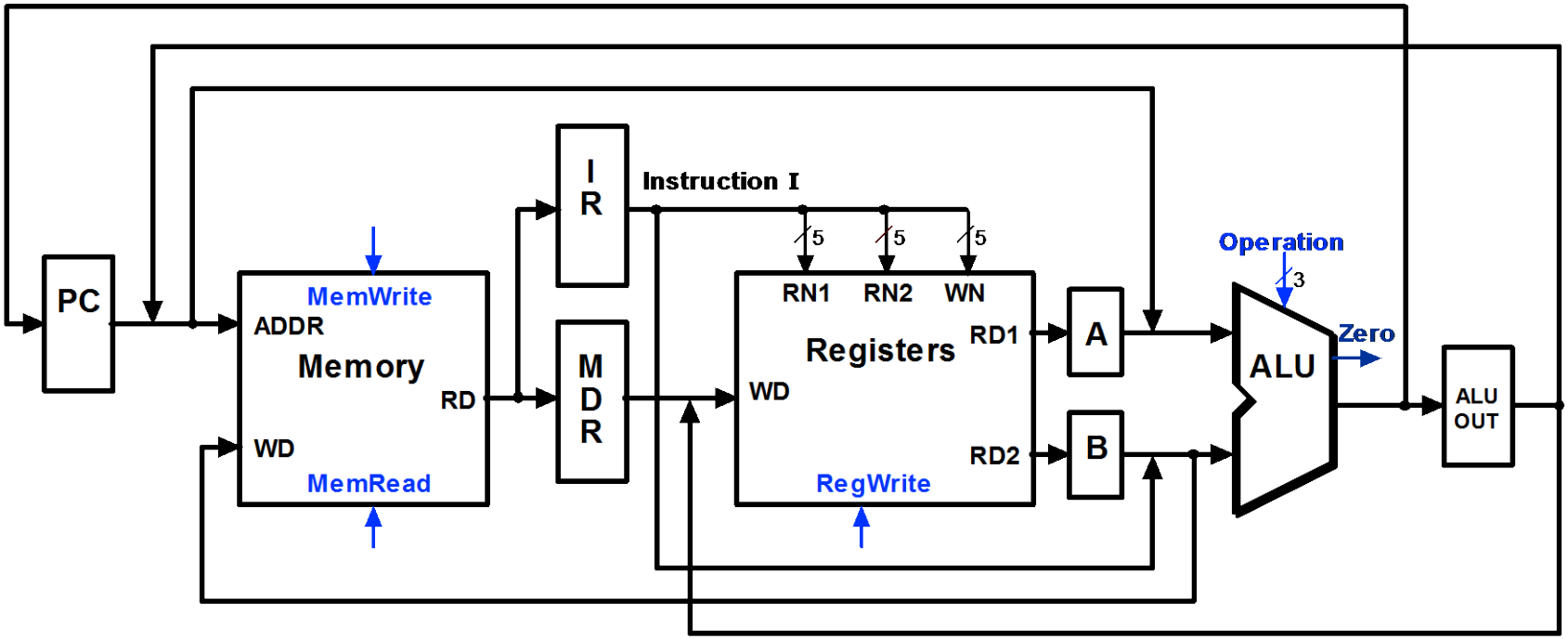
- Break instruction execution into multiple cycles
- One clock cycle for each major task
  1. Instruction Fetch (IF)
  2. Instruction Decode and Register Fetch (ID)
  3. Execution, memory address computation, or branch computation (EX)
  4. Memory access / R-type instruction completion (MEM)
  5. Memory read completion (WB)
- Share hardware to simplify datapath

This would mean that instructions complete as soon as possible, instead of being limited by the slowest instruction.

Characteristics of Multicycle Design

- Instructions take more than one cycle
  - Some instructions take more cycles than others
  - Clock cycle is shorter than single-cycle clock
- Reuse of major components simplifies datapath
  - Single ALU for all calculations
  - Single memory for instructions and data
  - But, added registers needed to store values across cycles
- Control Unit Implemented State Machine
  - Control signals no longer a function of just the instruction

Multicycle Datapath - High-Level View



Goto pp.5

Review: Register Transfers

- Instruction Fetch

  Instruction <- MEM[PC]; PC = PC + 4;

- Instruction Execution

Instr.	Register Transfers
`add`	R[rd] <- R[rs] + R[rt];
`sub`	R[rd] <- R[rs] - R[rt];
`and`	R[rd] <- R[rs] & R[rt];
`or`	R[rd] <- R[rs] | R[rt];
`lw`	R[rt] <- MEM[R[rs] + s_extend(offset)];
`sw`	MEM[R[rs] + sign_extend(offset)] <- R[rt];
`beq`	if (R[rs] == R[rt]) then PC <- PC+4 + s_extend(offset<<2)
	else PC <- PC + 4
`j`	PC <- upper(PC)@(address << 2)

Key idea: break into multiple cycles!

Multicycle Execution

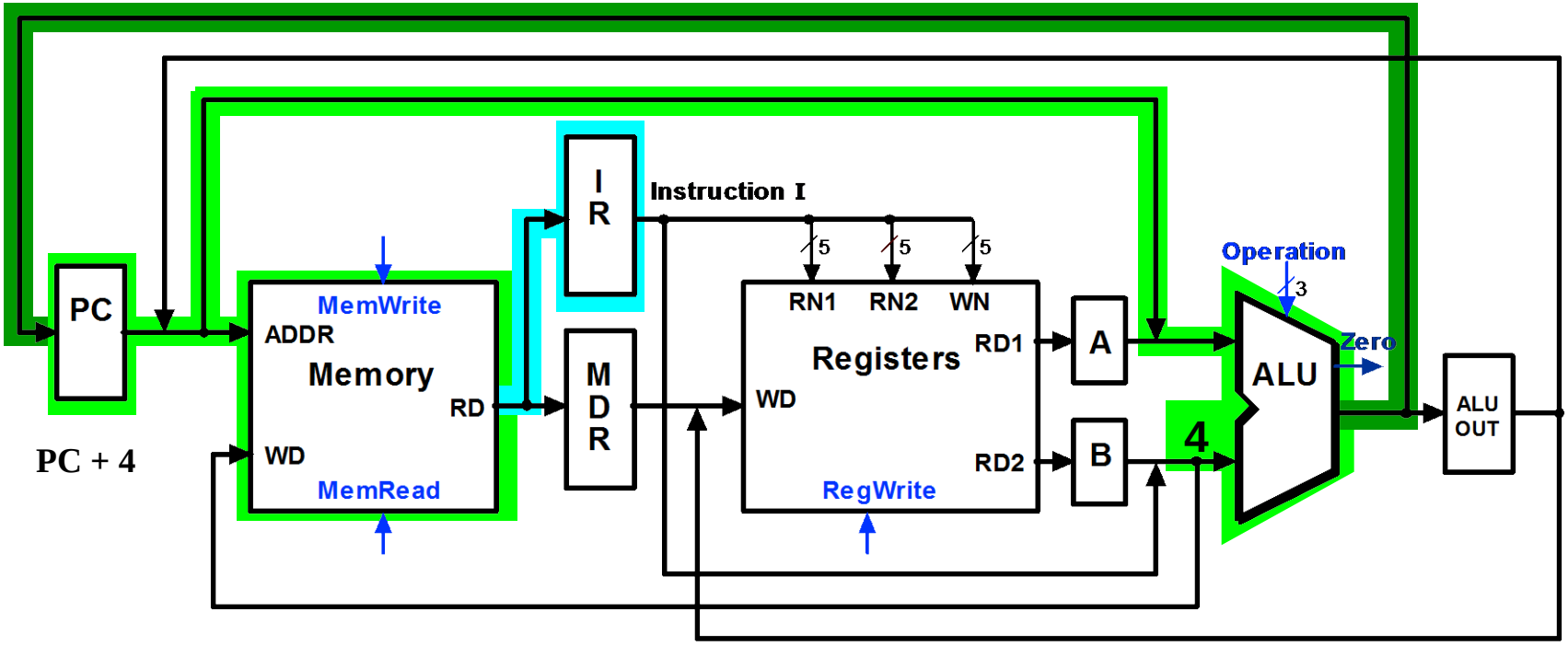
- Instructions take between 3 and 5 clock cycles

	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
(1)	Instruction fetch	IR = Memory[PC]			
		PC = PC + 4			
(2)	Instruction decode/register fetch	A = Reg [IR[25-21]]			
		B = Reg [IR[20-16]]			
		ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
(3)	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
(4)	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
(5)	Memory read completion		Load: Reg[IR[20-16]] = MDR		

New registers needed to store values across clock steps!

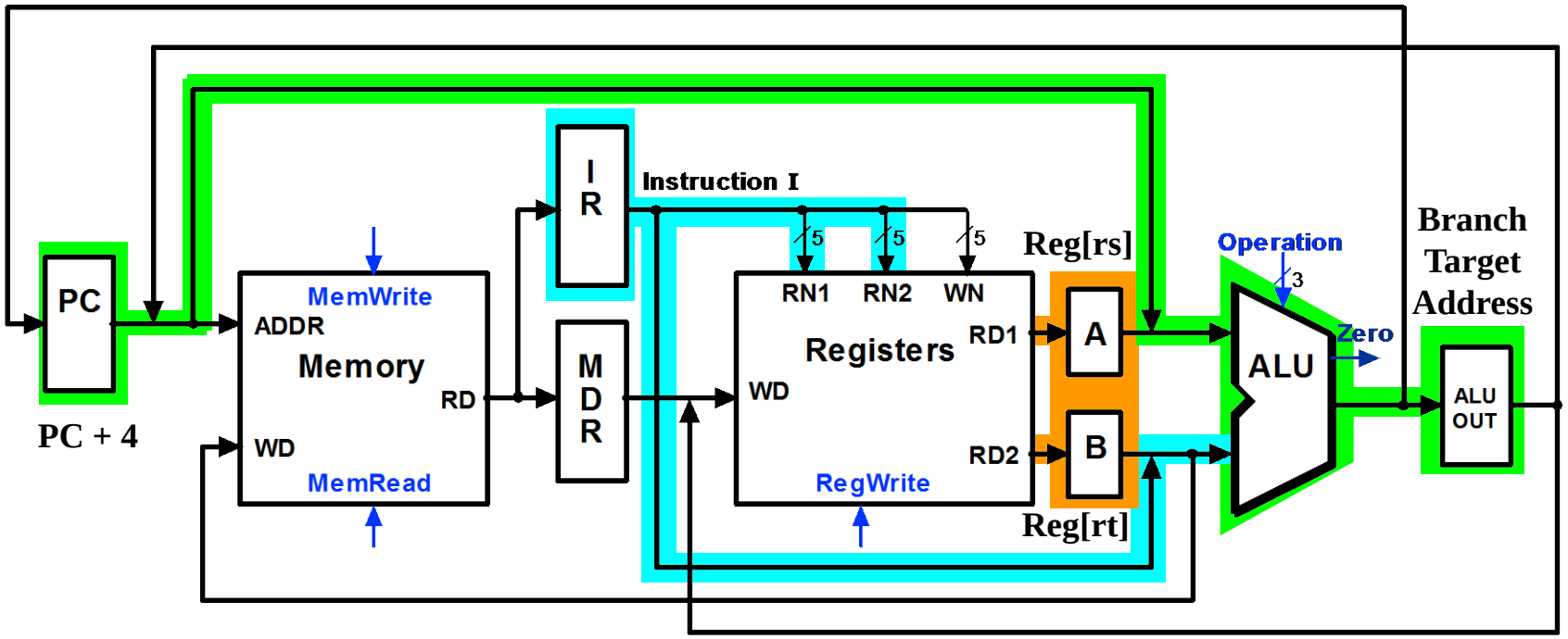
Multicycle Execution Step (1)
Instruction Fetch

```
IR = Memory[PC];
PC = PC + 4;
```



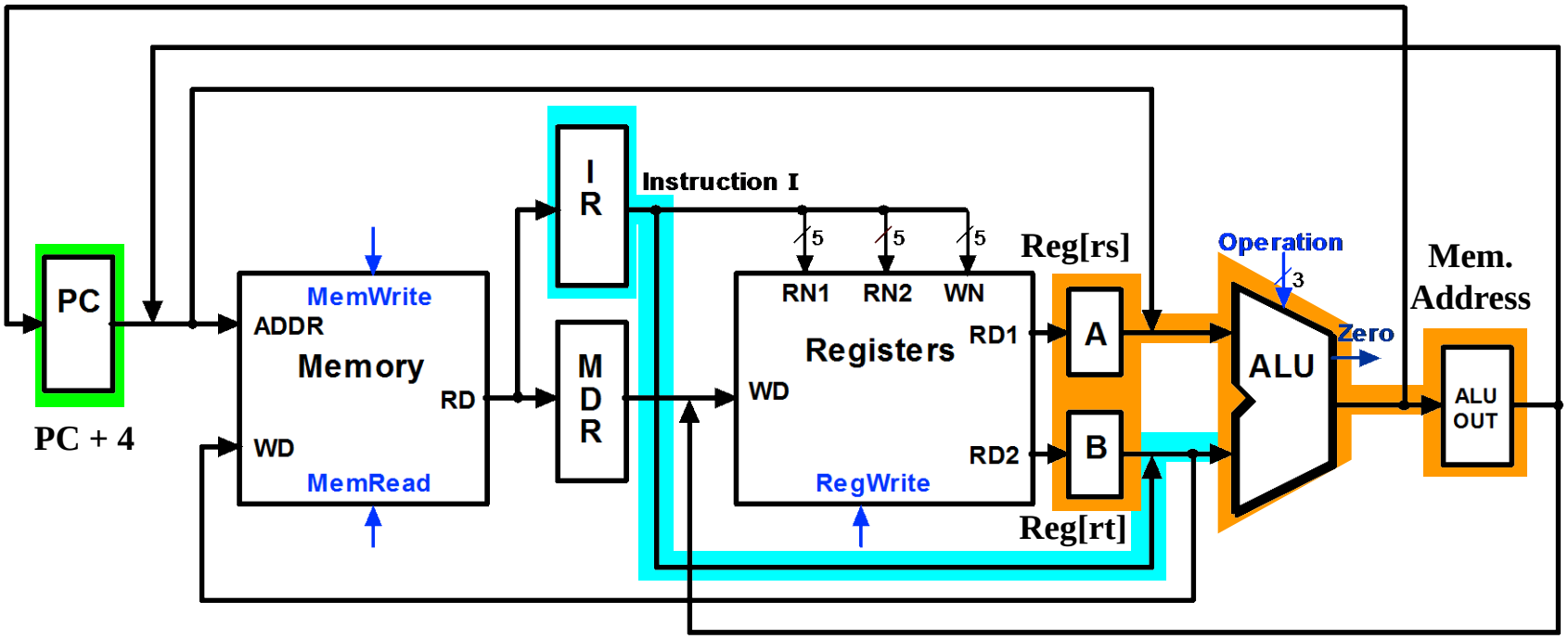
Multicycle Execution Step (2)
Instruction Decode and Register Fetch

```
A = Reg[IR[25-21]];                (A = Reg[rs])
B = Reg[IR[20-15]];                (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2)
```



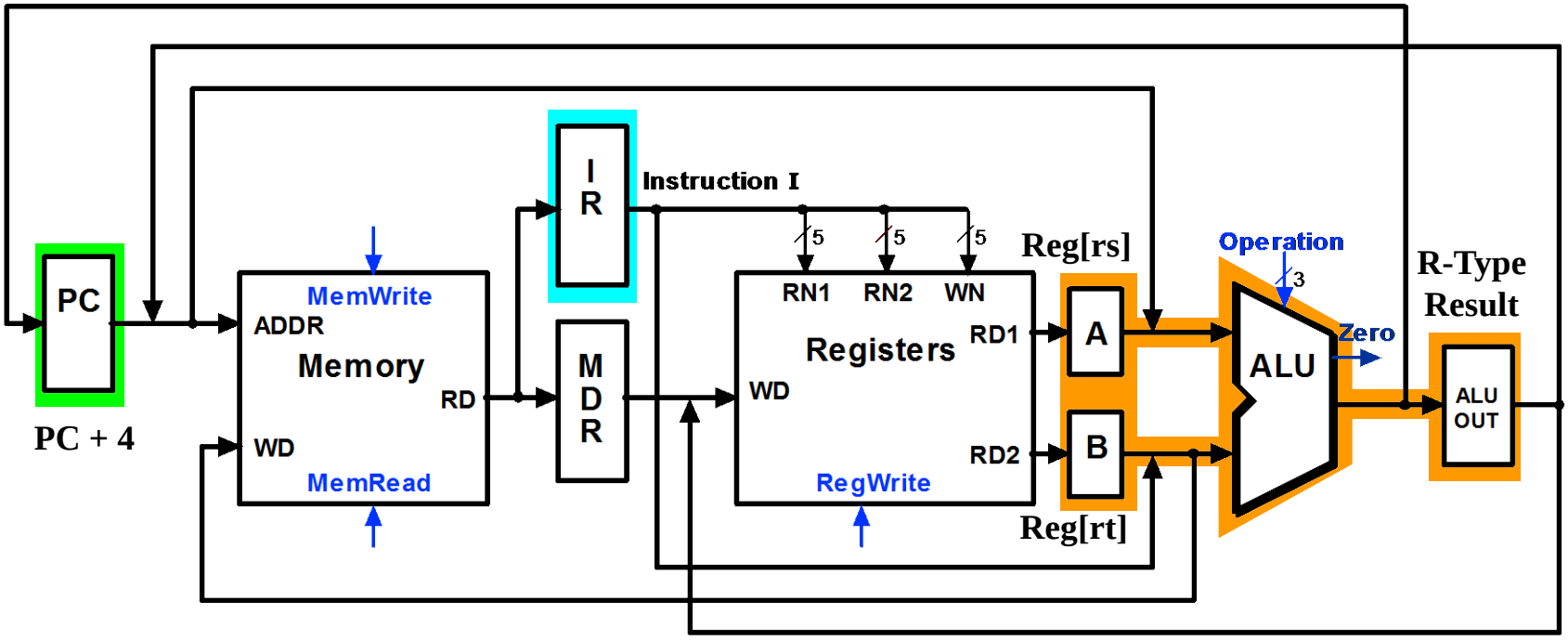
Multicycle Execution Steps (3)
Memory Reference Instructions

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



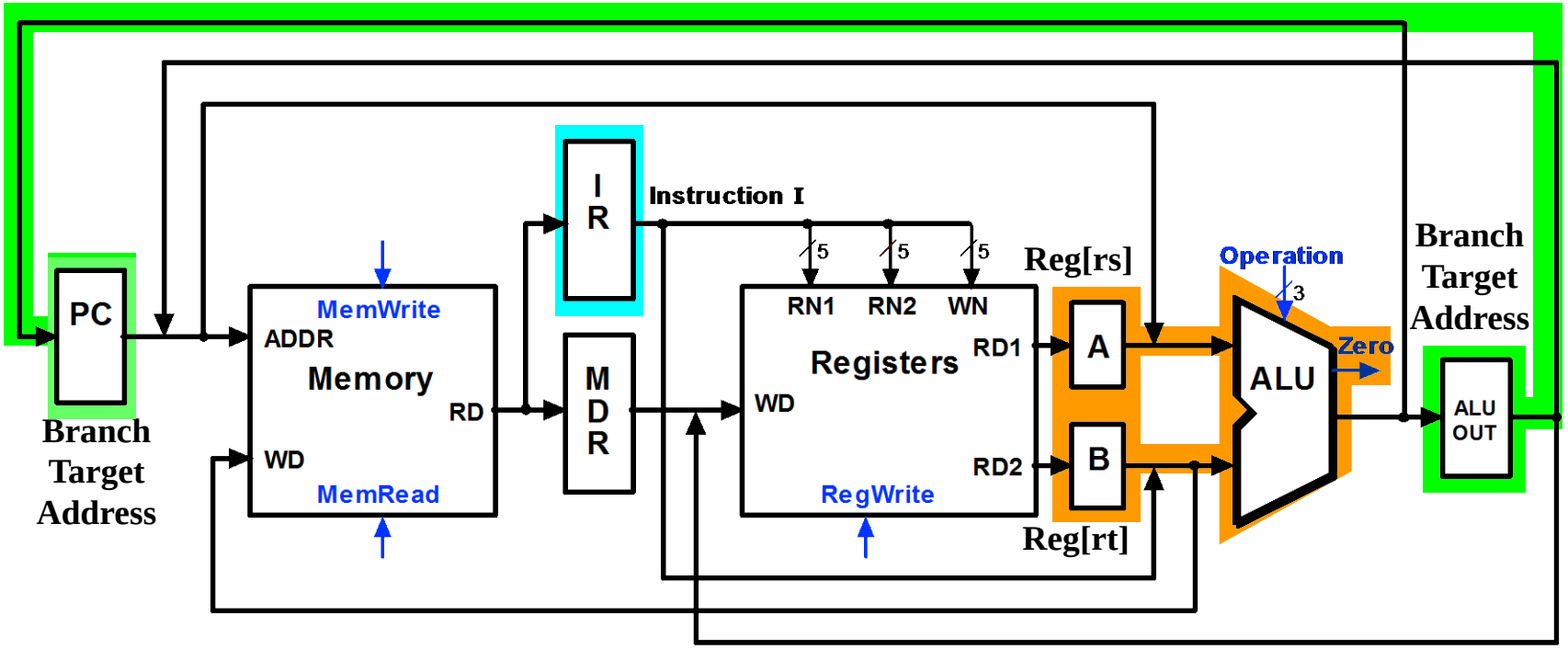
Multicycle Execution Steps (3)
ALU Instruction (R-Type)

```
ALUOut = A op B
```



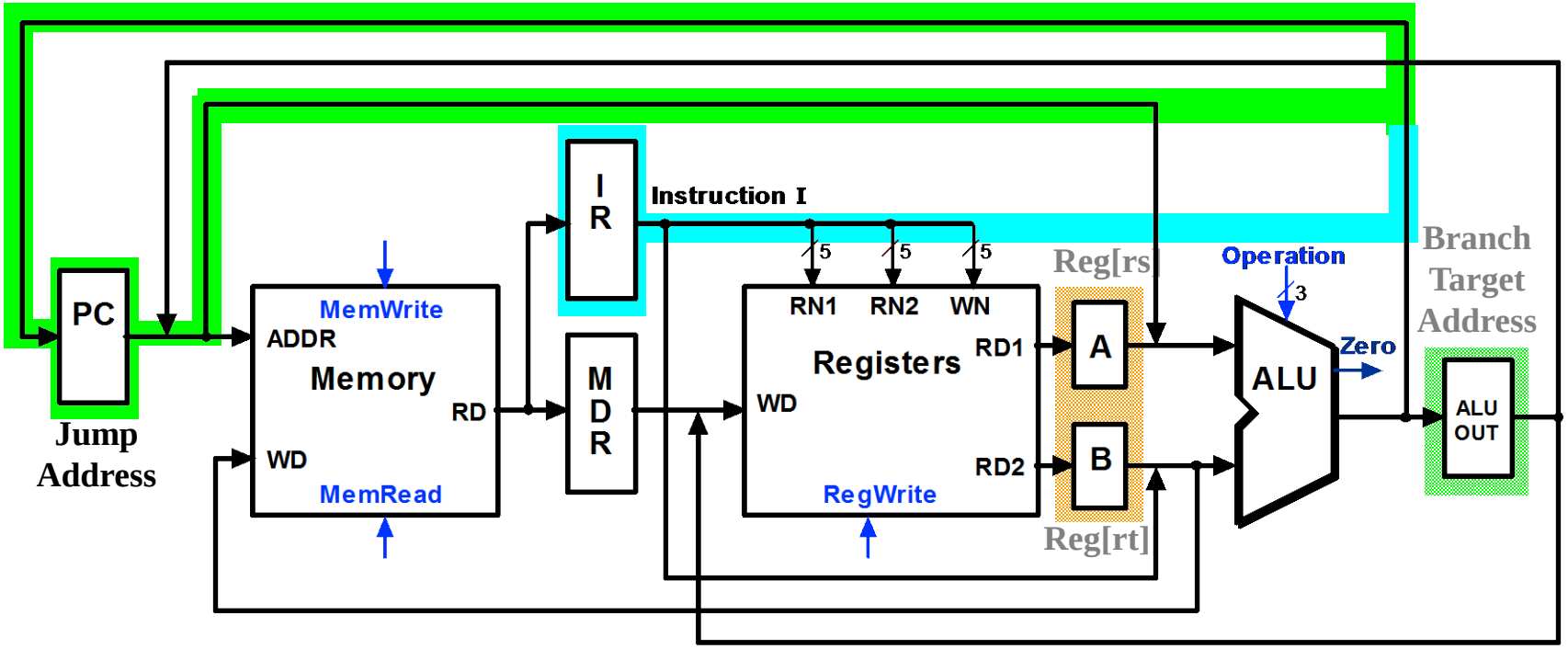
Multicycle Execution Steps (3)
Branch Instructions

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



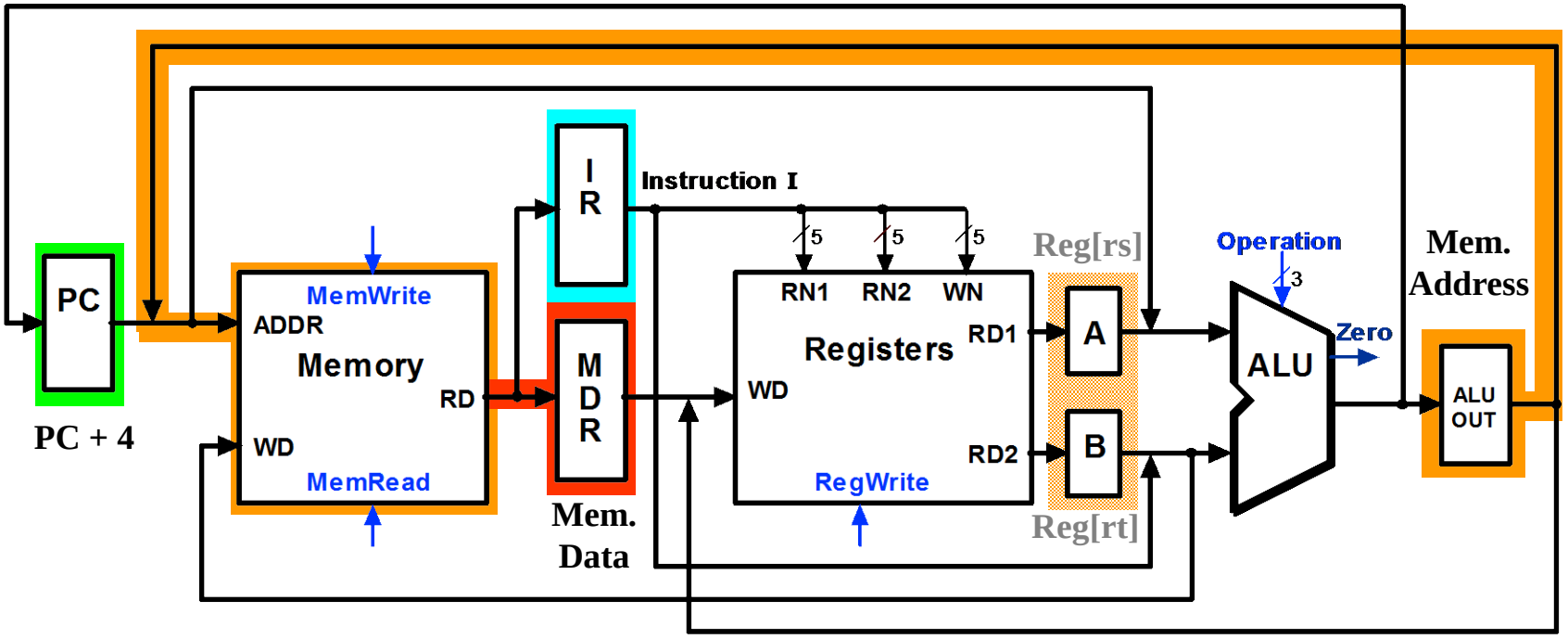
Multicycle Execution Step (3) Jump Instruction

```
PC = PC[31-28] concat (IR[25-0] << 2)
```



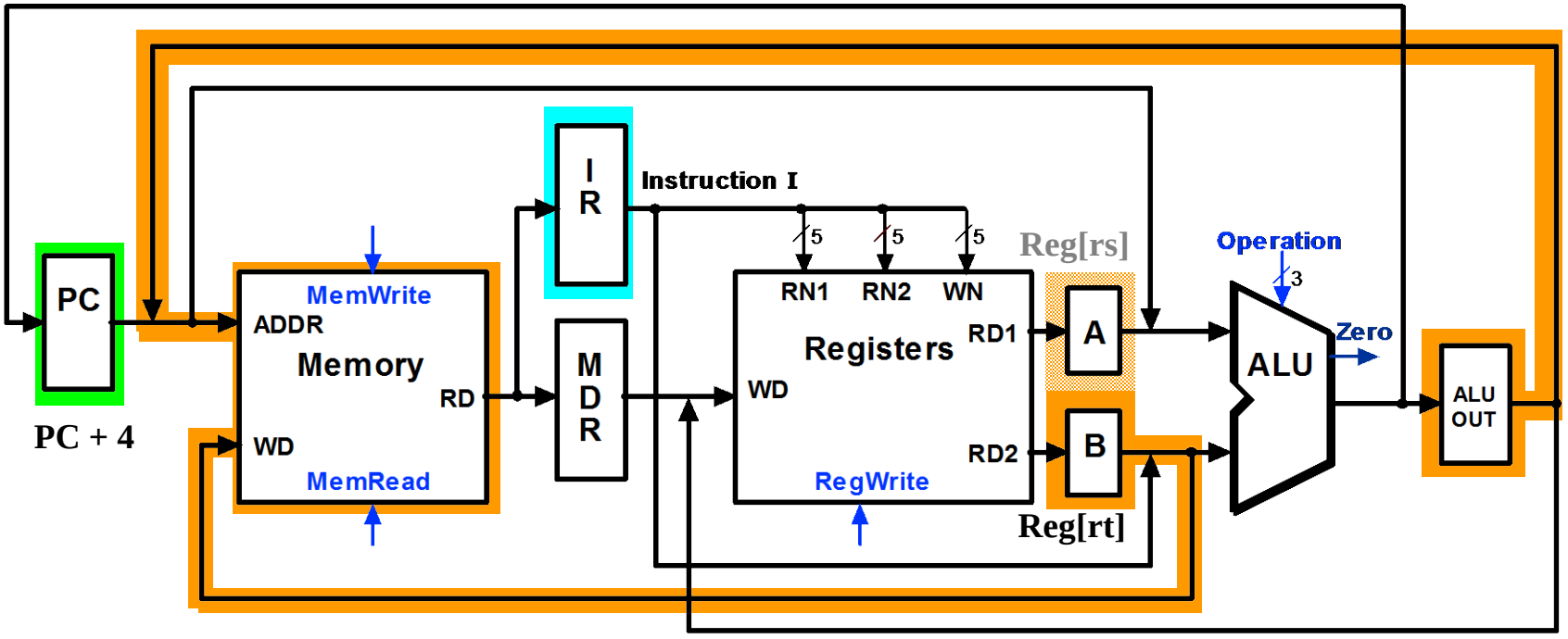
Multicycle Execution Steps (4) Memory Access - Read (lw)

```
MDR = Memory[ALUOut];
```



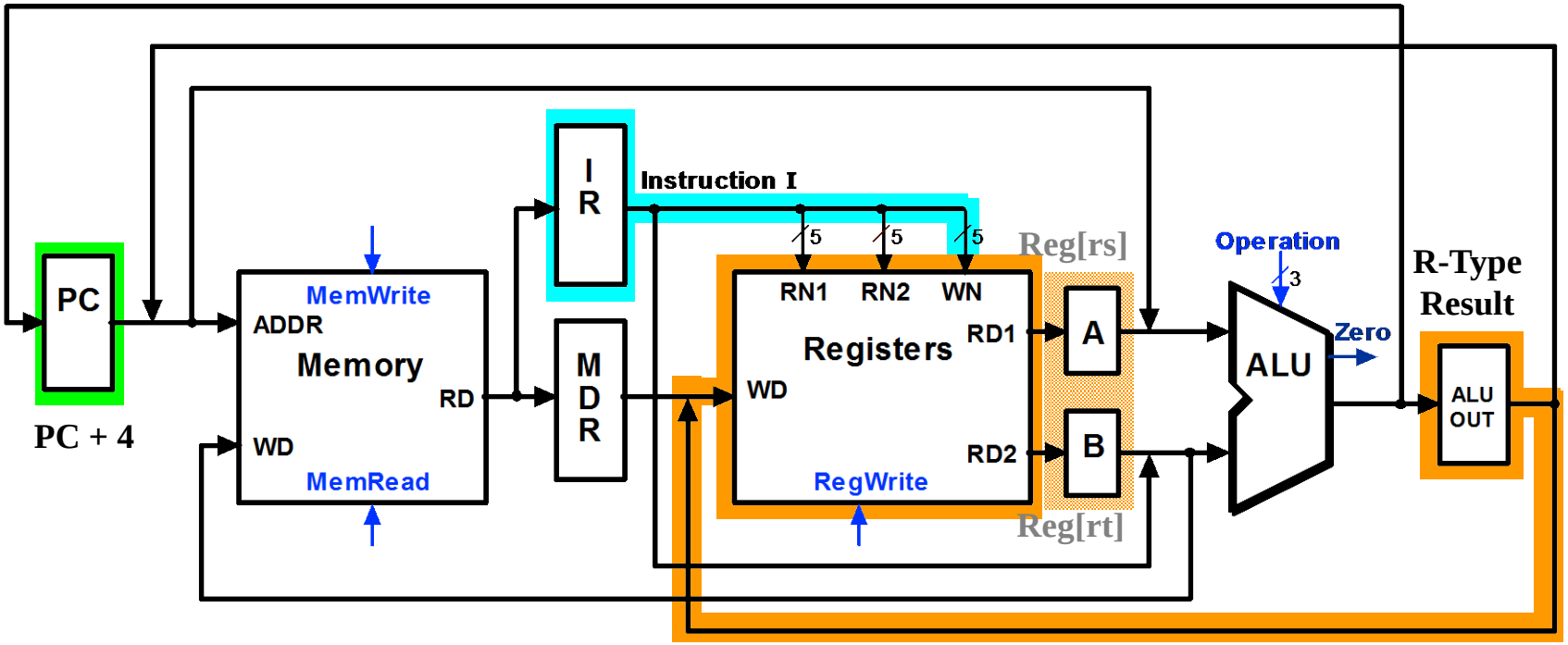
Multicycle Execution Steps (4) Memory Access - Write (sw)

```
Memory[ALUOut] = B;
```



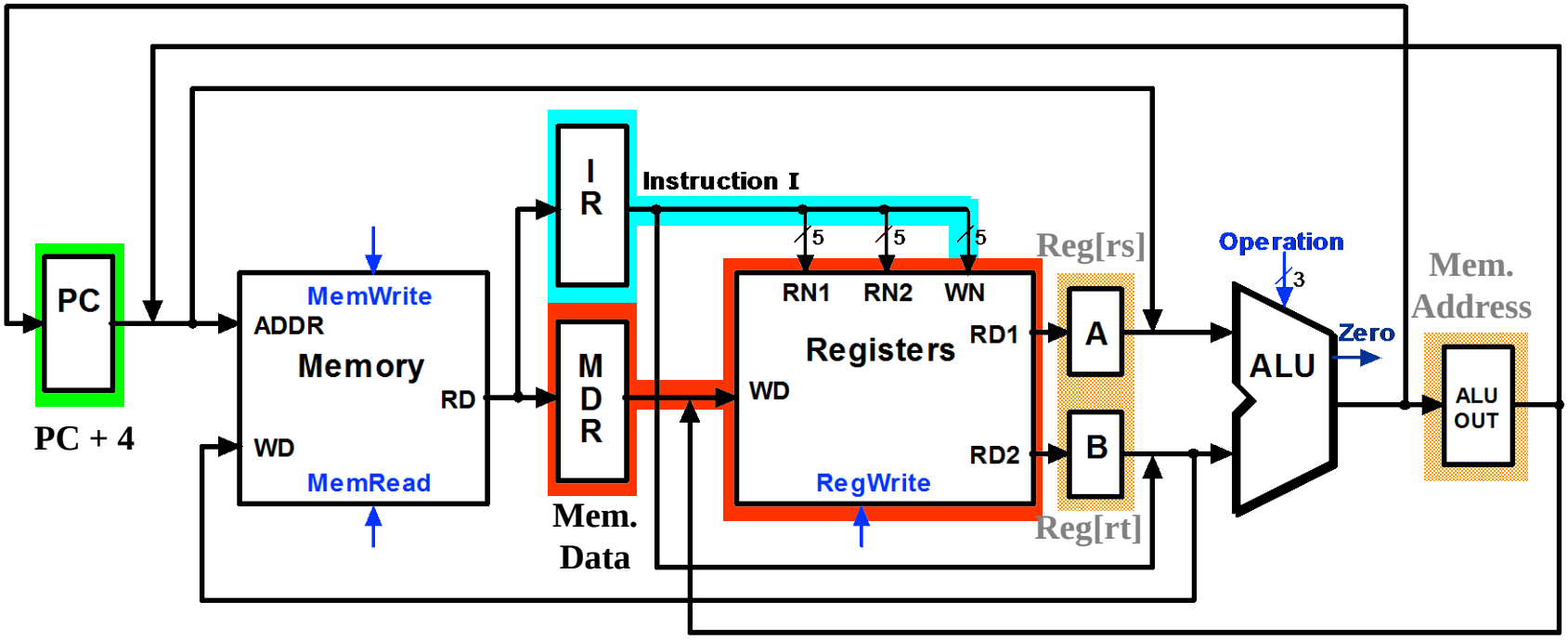
Multicycle Execution Steps (4)
ALU Instruction (R-Type)

```
Reg[IR[15:11]] = ALUOUT
```



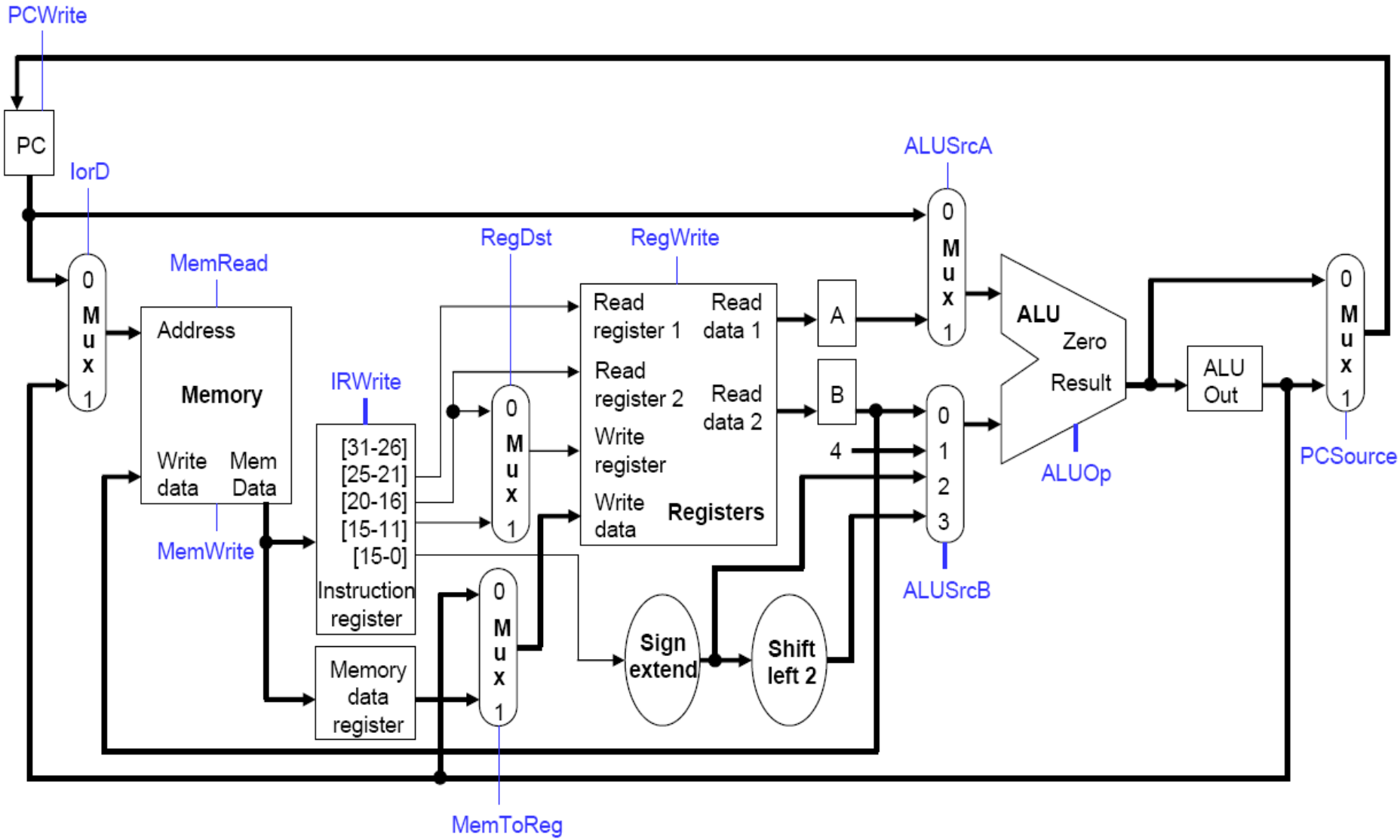
Multicycle Execution Steps (5)
Memory Read Completion (lw)

```
Reg[IR[20-16]] = MDR;
```



Controlling the multicycle datapath

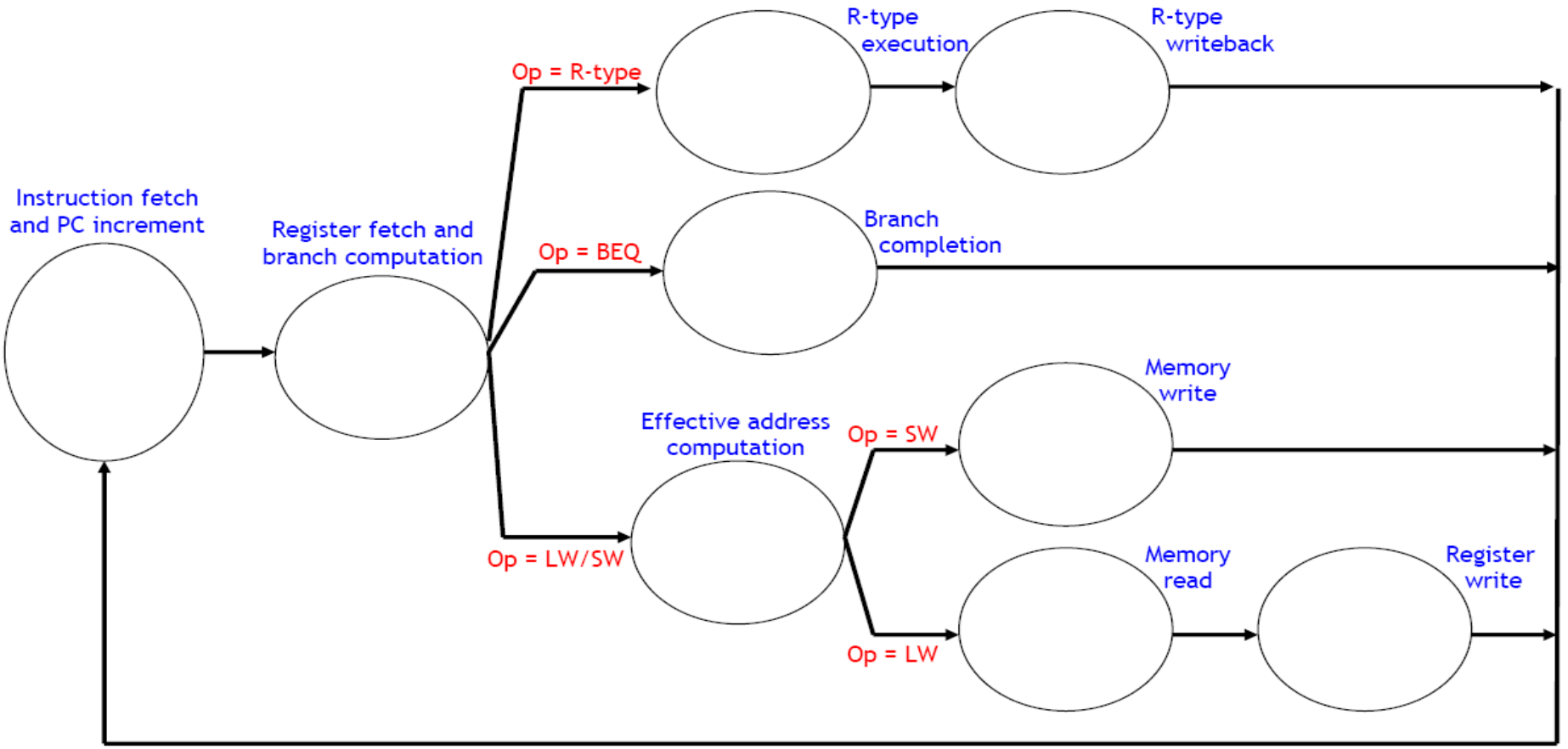
- Today, we talk about how to control this datapath.



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 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 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.

- 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

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 weeks in the context of pipelining.

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.

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.

$$Reg[IR[20\text{-}16]] = MDR$$

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

Stages 4-5 (lw) control signals

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

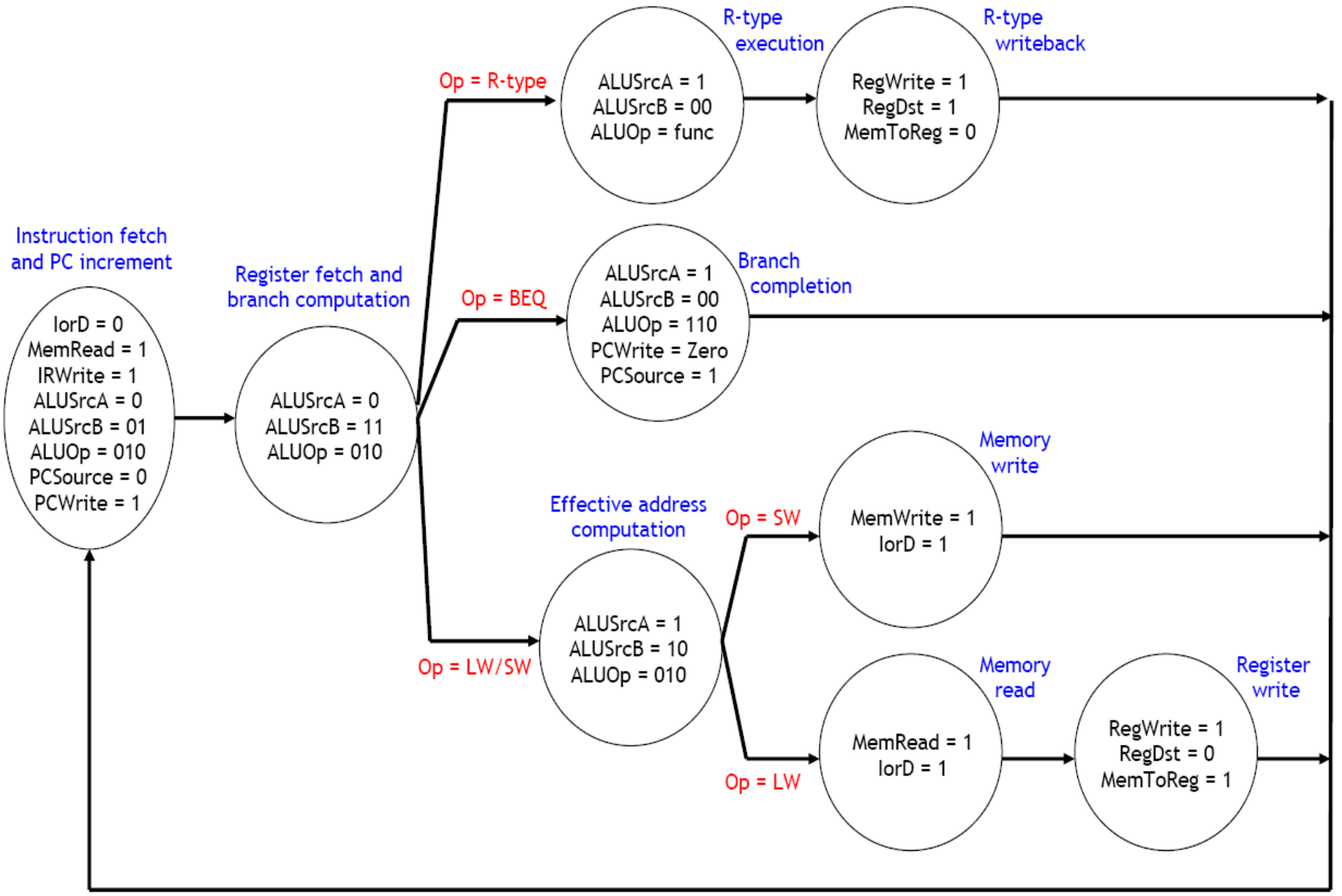
Signal	Value	Description
MemRead	1	Read from memory
IorD	1	Use ALUOut as the memory address

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.