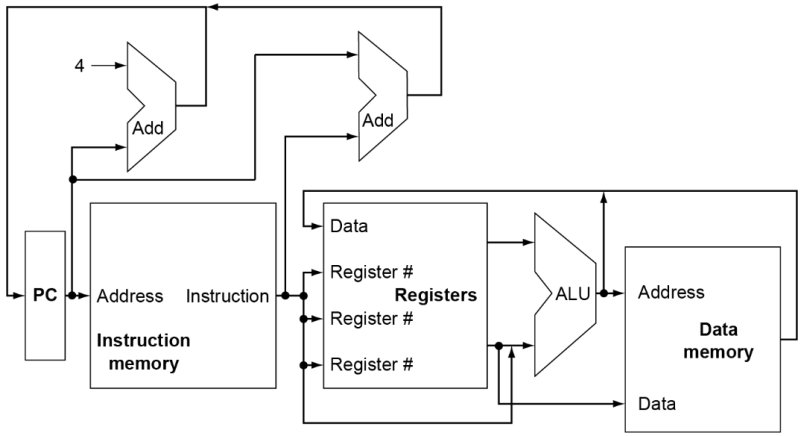
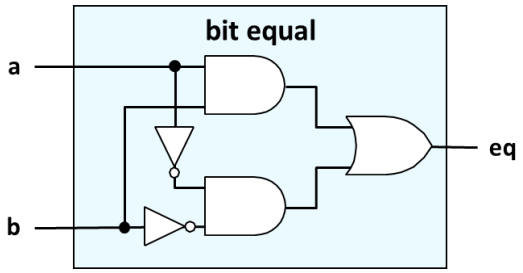
Processor Architecture

Sequential RISC-V Implementation



Module Outline

- Recap – The Computer in a Nutshell
- A Basic RISC-V Implementation
- Building the Datapath
- Instruction Execution Phases
- Building the Control Path
- Module Summary

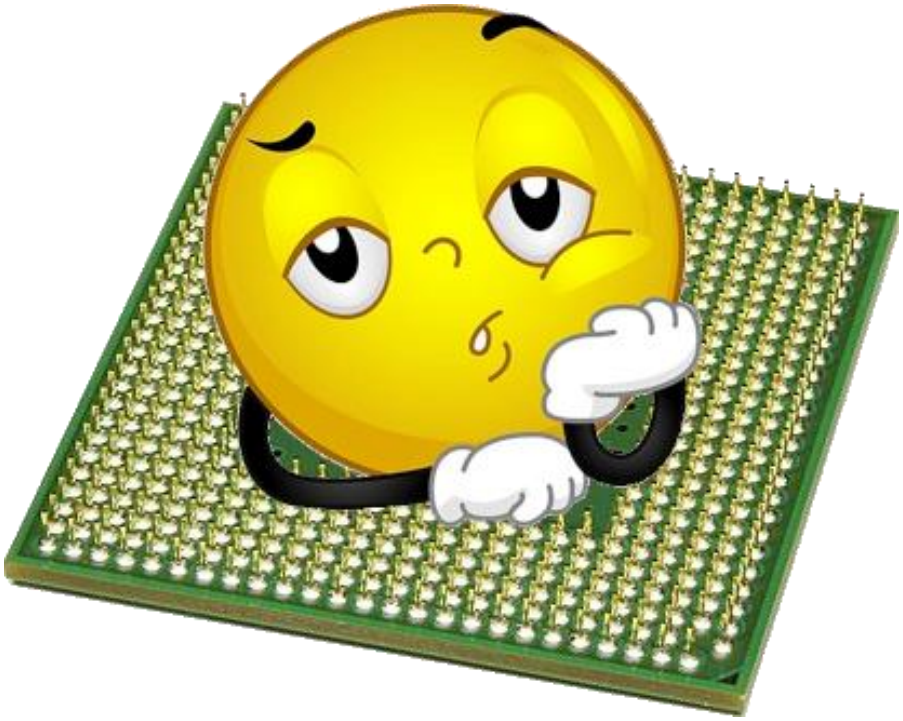


Recap

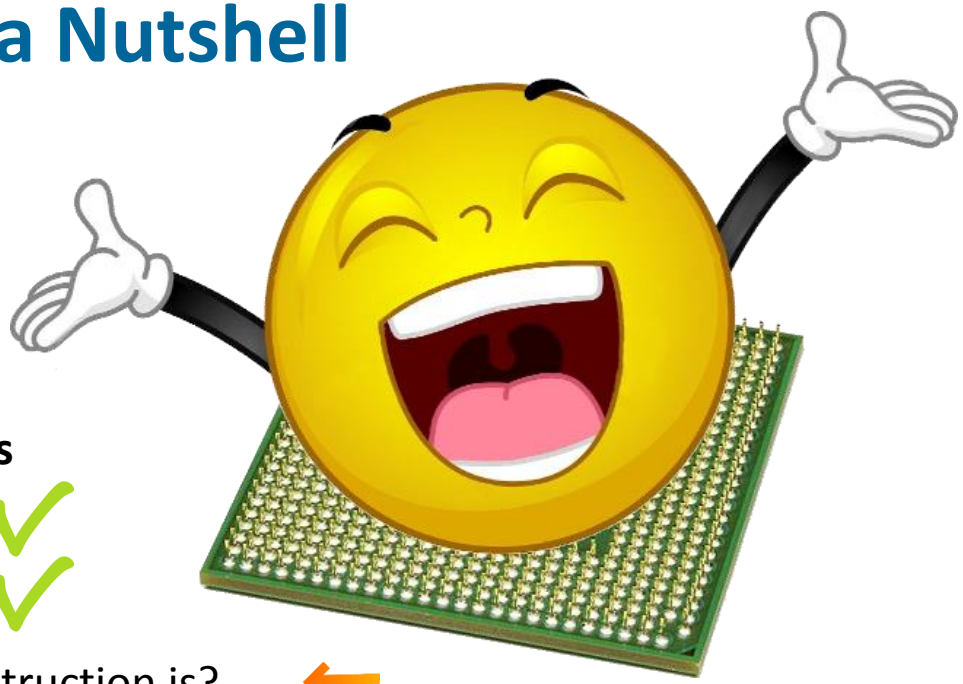
The Computer in a Nutshell

Recap: The Computer in a Nutshell

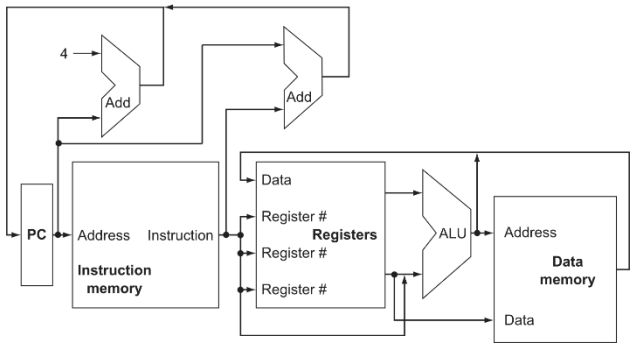
- **Life as a computer**
  1. read next instruction
  2. execute next instruction
  3. goto step 1



Recap: The Computer in a Nutshell



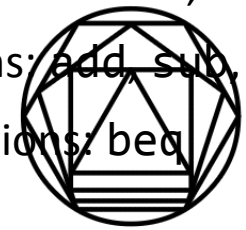
- **Lots of interesting engineering problems**
  - where are the instructions? ✓
  - how does an instruction look like? ✓
  - how do we know where the next instruction is? ←
  - how do we know how to execute an instruction? ←
  - where is the data?
  - how do we bring the data to the processor? ←
  - what happens after powering on?
  - what happens after powering off?
  - how do we make the computer as fast as possible? ←



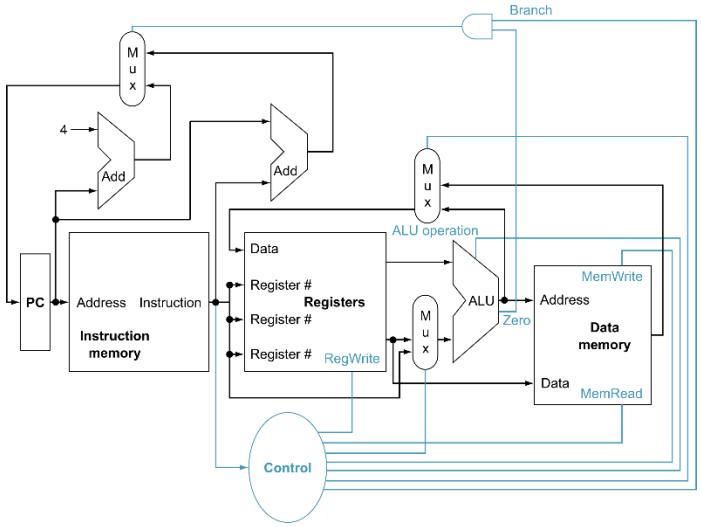
A Basic RISC-V Implementation

Let's Build Our Own RISC-V Processor!

- Slightly simplified RISC-V processor
  - memory operations: `lw`, `sw`
  - ALU operations: `add`, `sub`, and, or
  - branch operations: `beq`



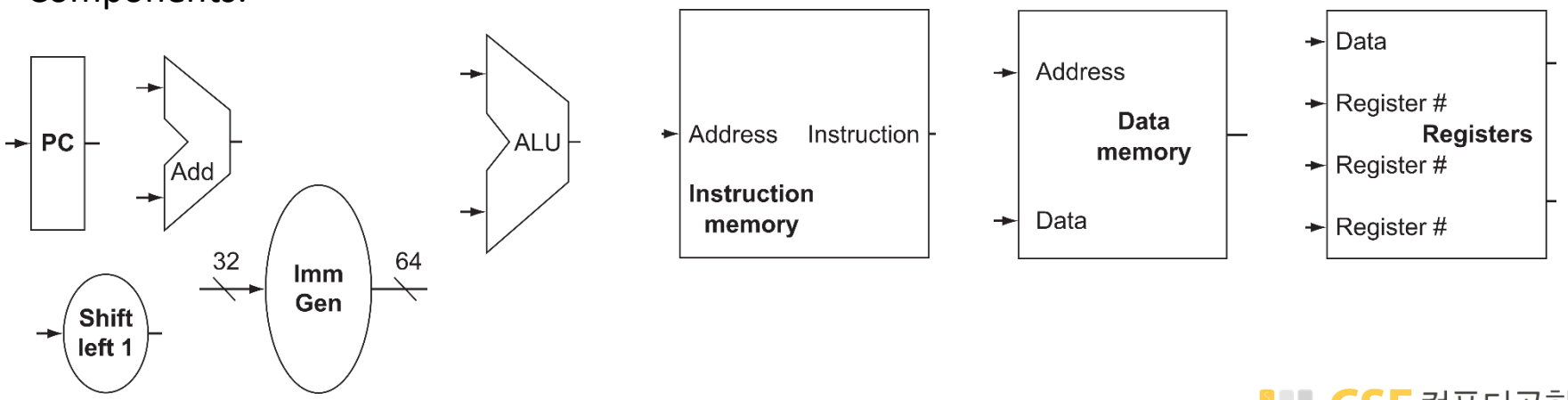
- We start with a sequential implementation that executes one full instruction per cycle



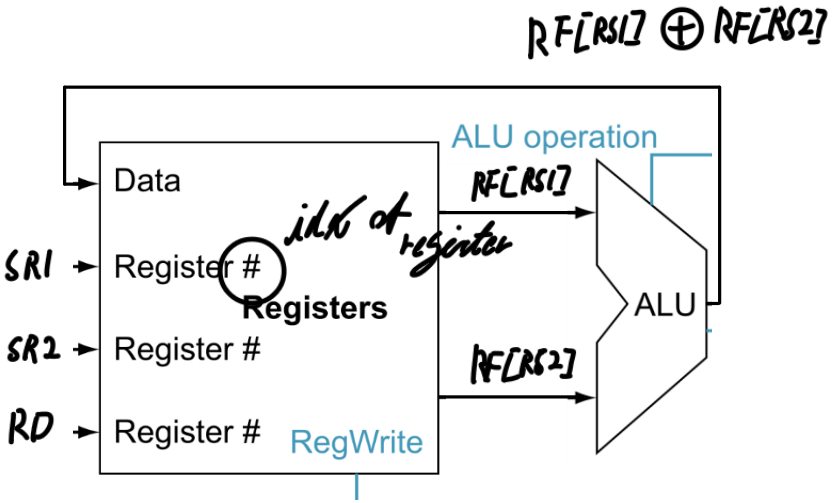
Building the Datapath

Building a Datapath

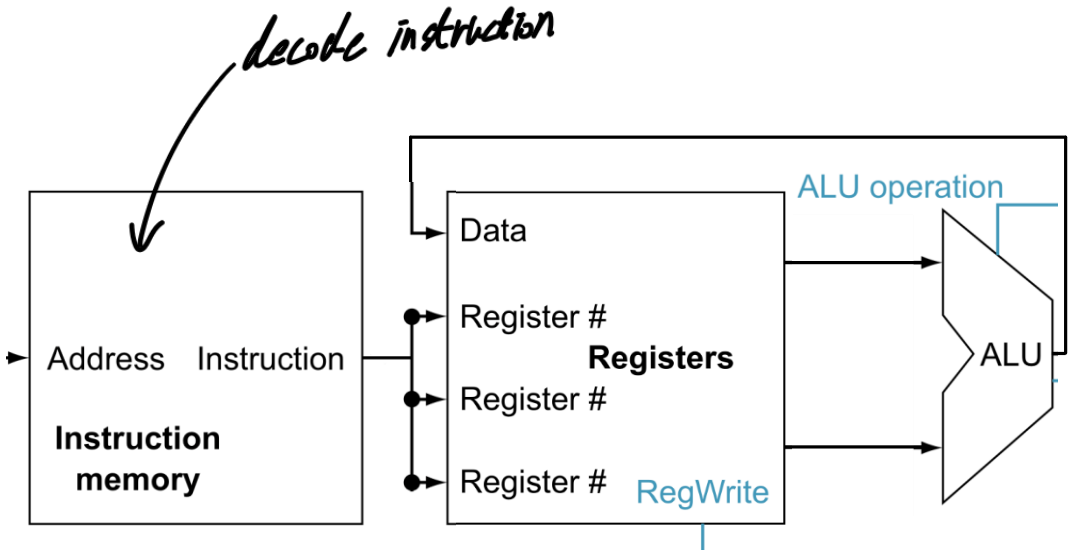
- Datapath
  - Elements that process data and addresses in the CPU
  - Registers, ALUs, MUX's, Memories, …
- Datapath executes an instruction in one clock cycle
  - Each datapath element can only perform one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions
- Components:



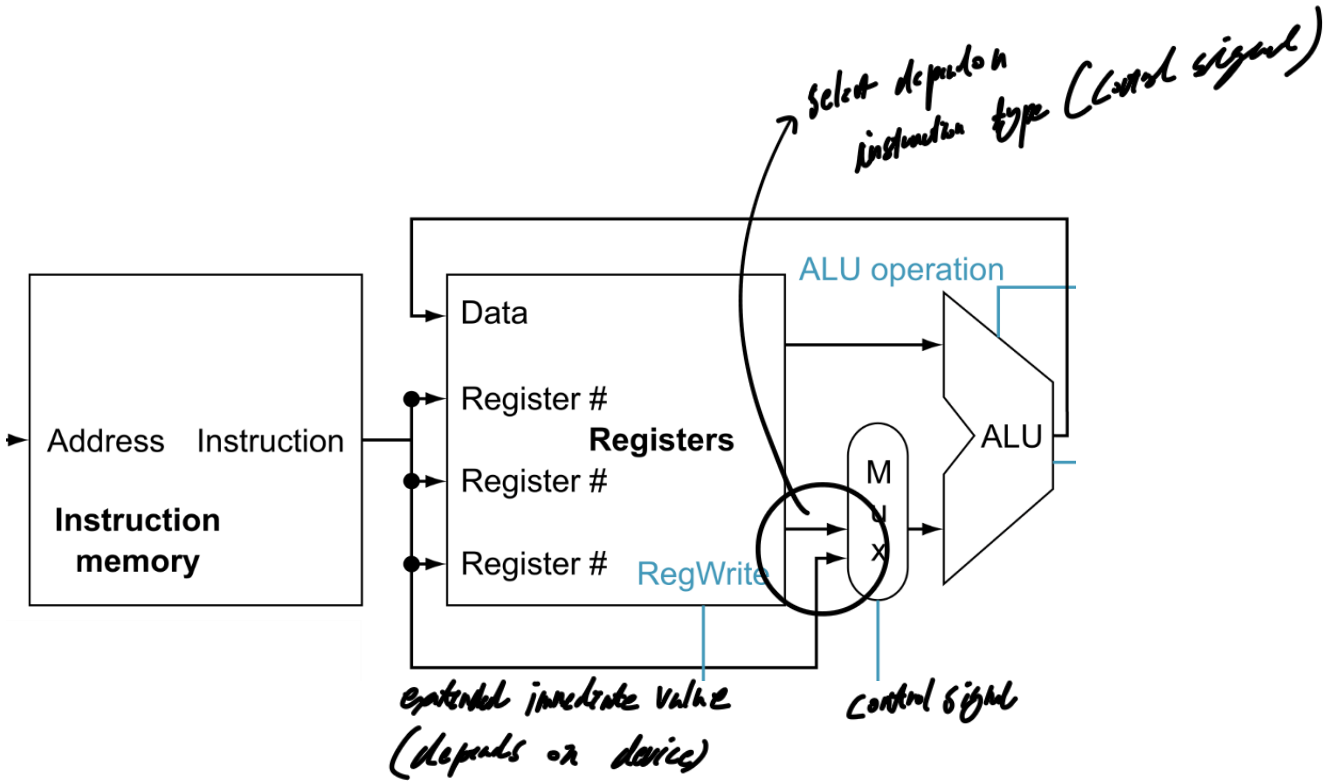
R-Type ALU Operations



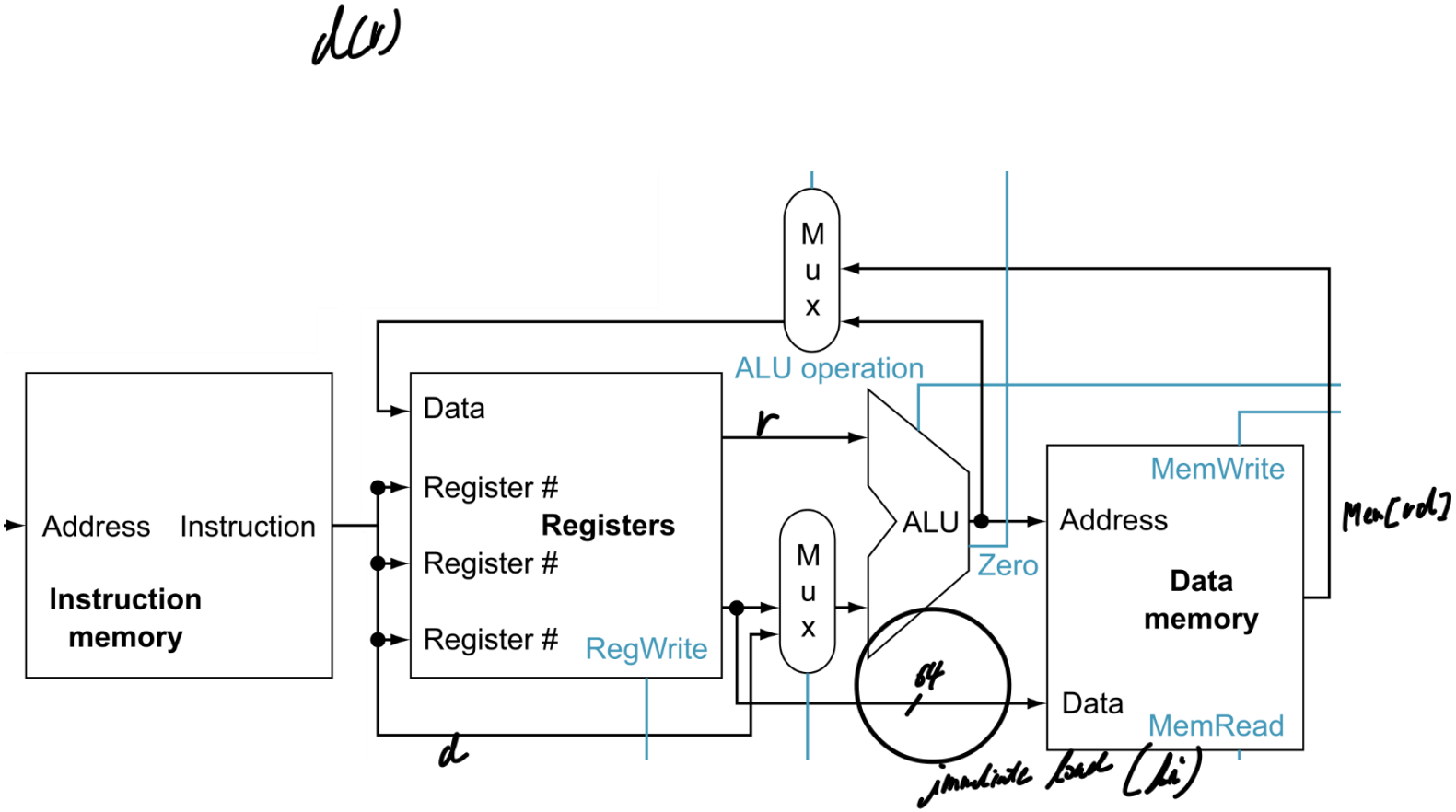
Adding Instruction Memory



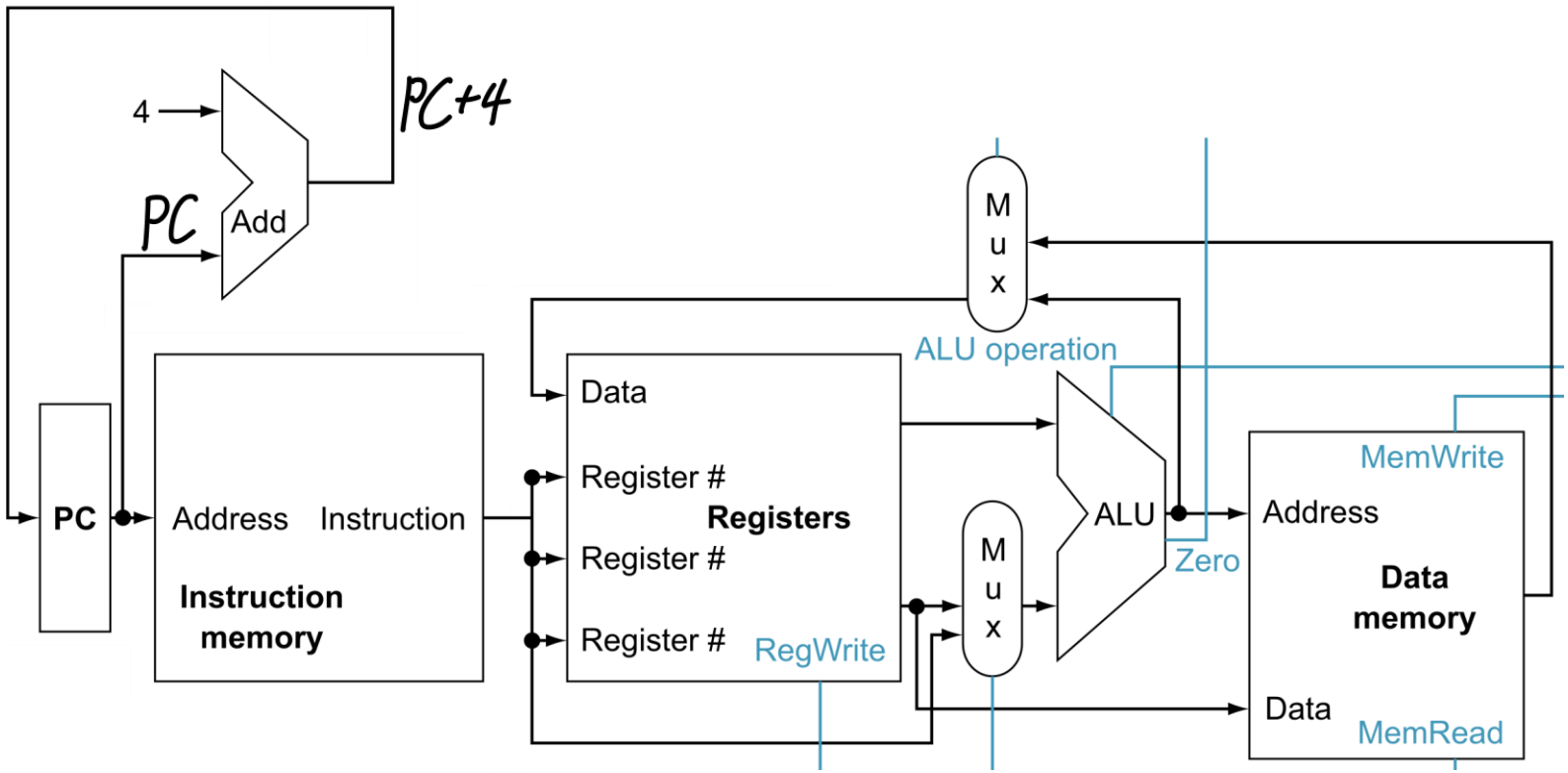
Supporting Immediate Operands



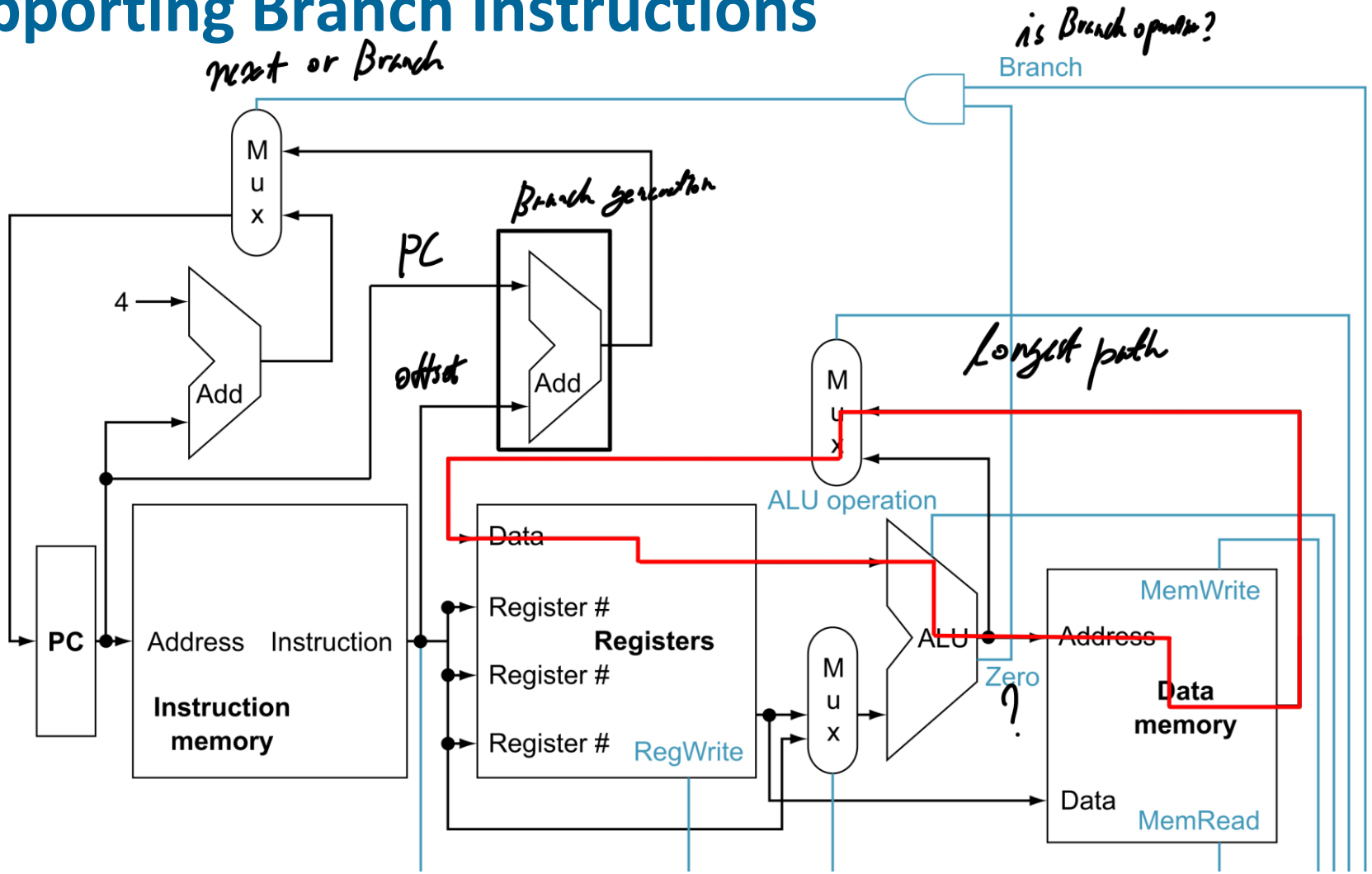
Adding Support for Load and Store Operations

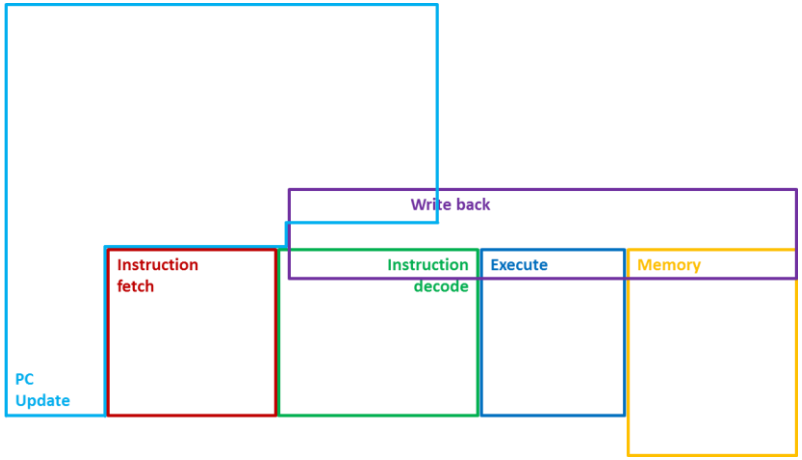


Updating the PC



Supporting Branch Instructions





Instruction Execution Phases

Instruction Execution Phases

1. **Instruction fetch (IF)**

   fetch instruction from instruction memory

2. **Instruction decode (ID)**

   generate control signals for components, read register file

3. **Execute (E)**

   ALU executes selected operation

4. **Memory (M)**

   access memory

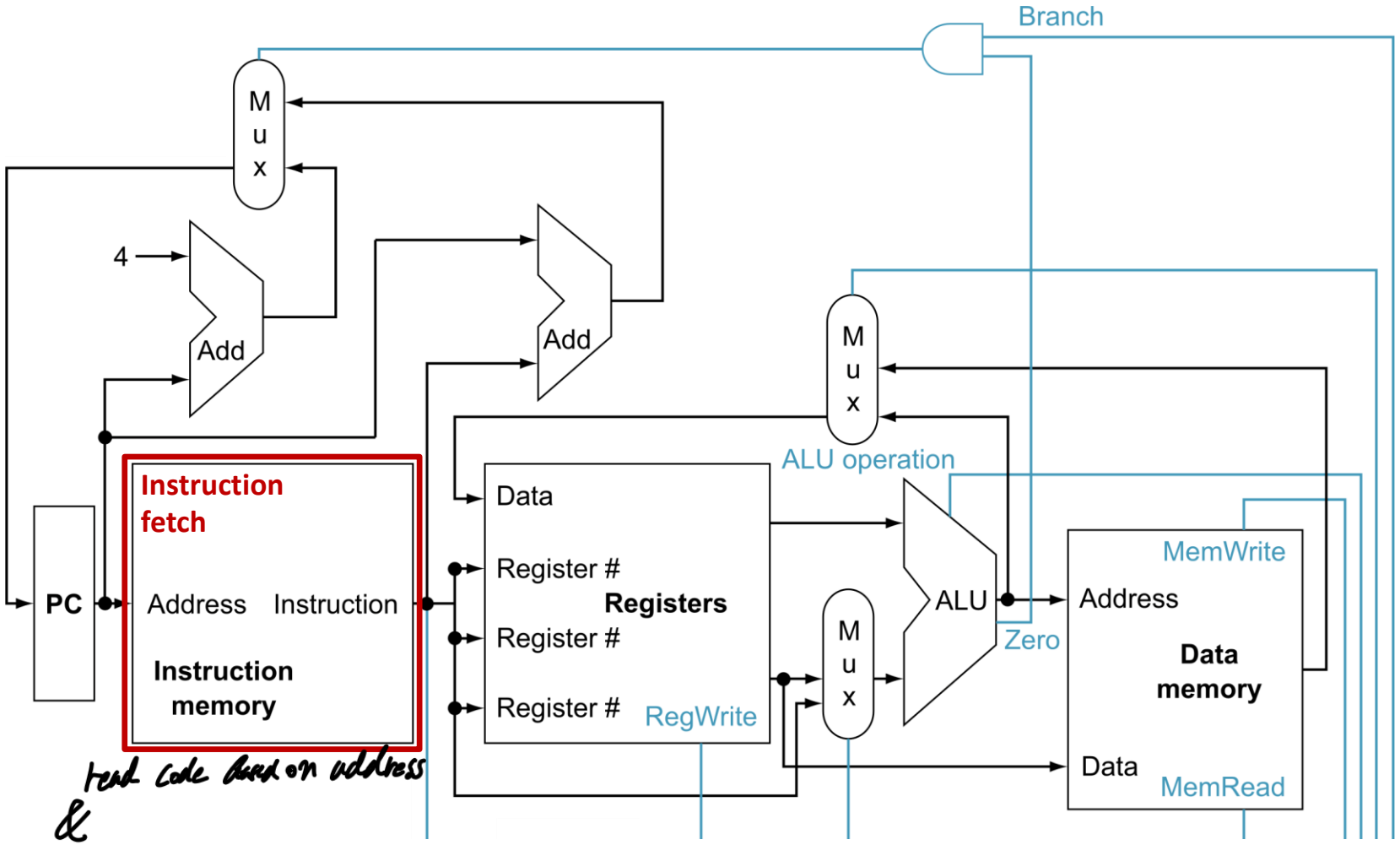
5. **Write-back (WB)**

   write result to register file

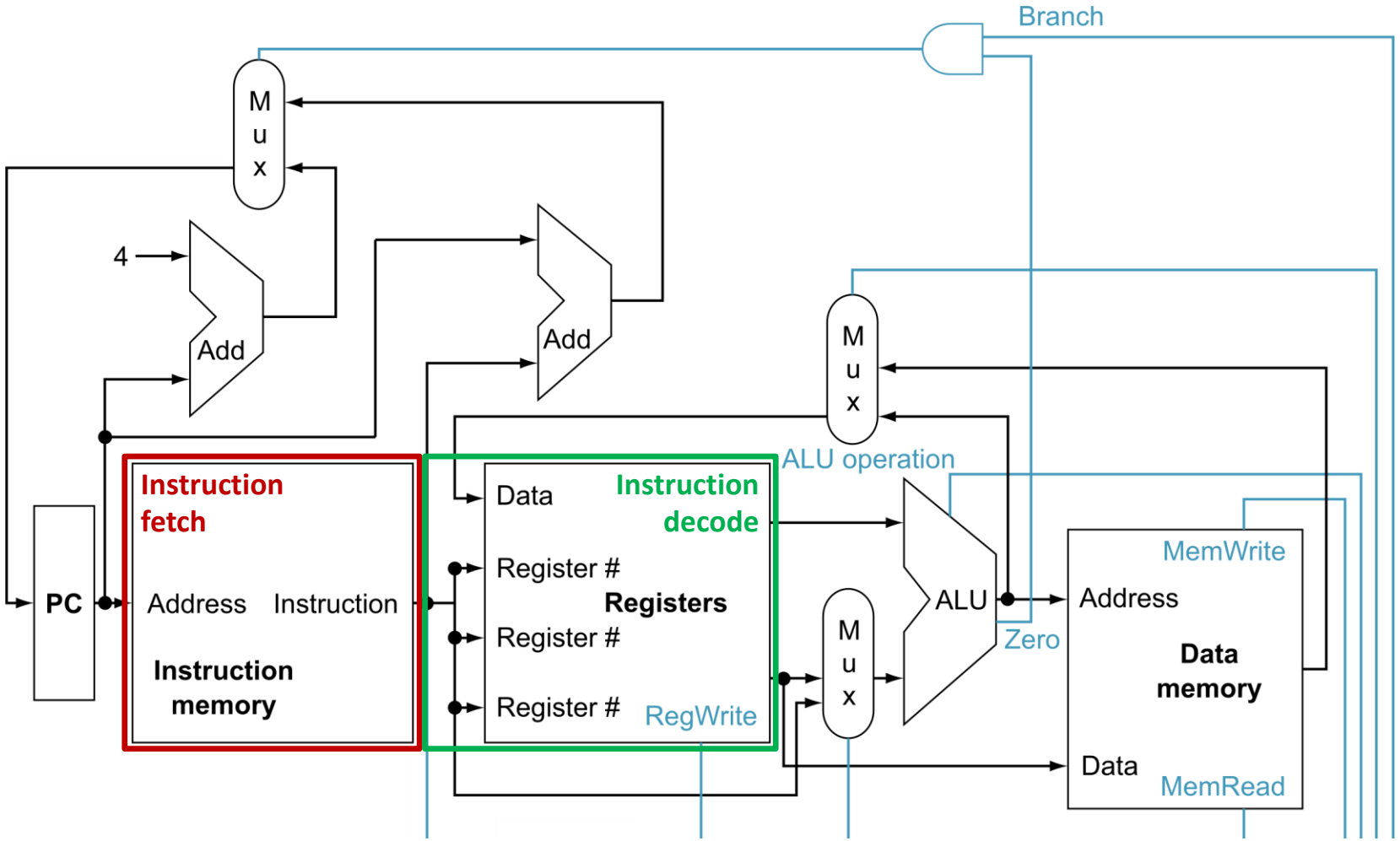
6. **Update PC (PC)**

   point PC to next instruction

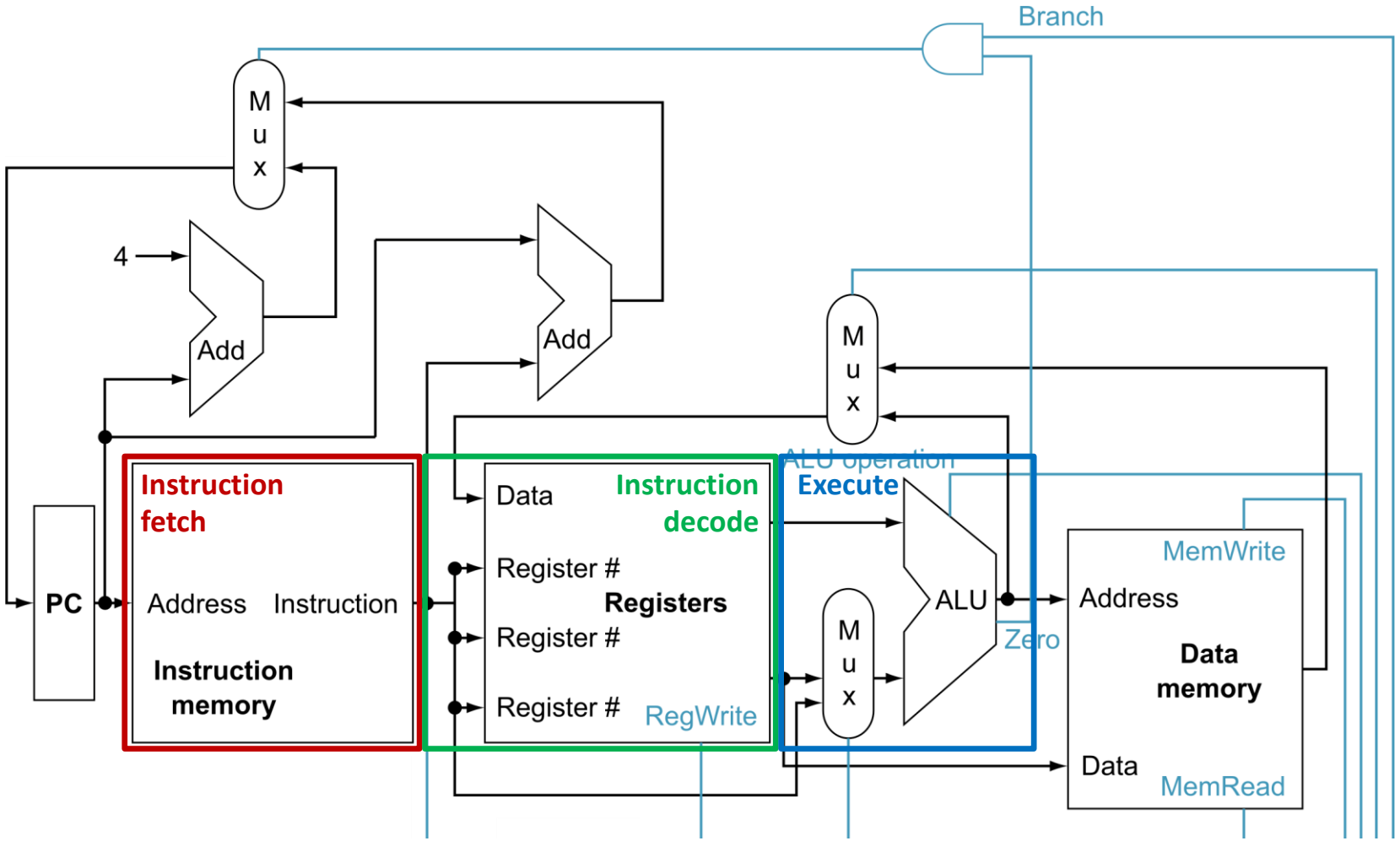
Instruction Execution Phases



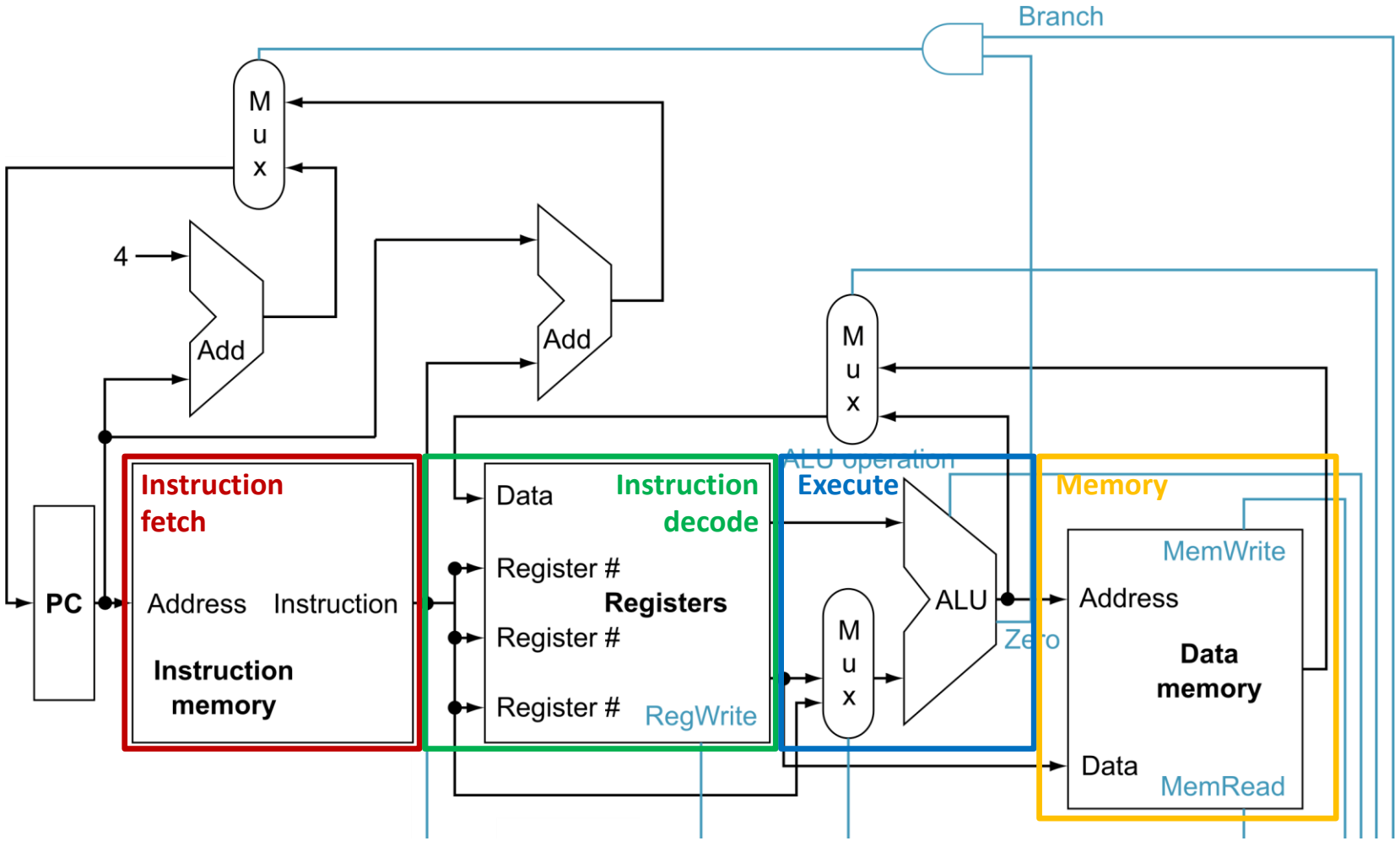
Instruction Execution Phases



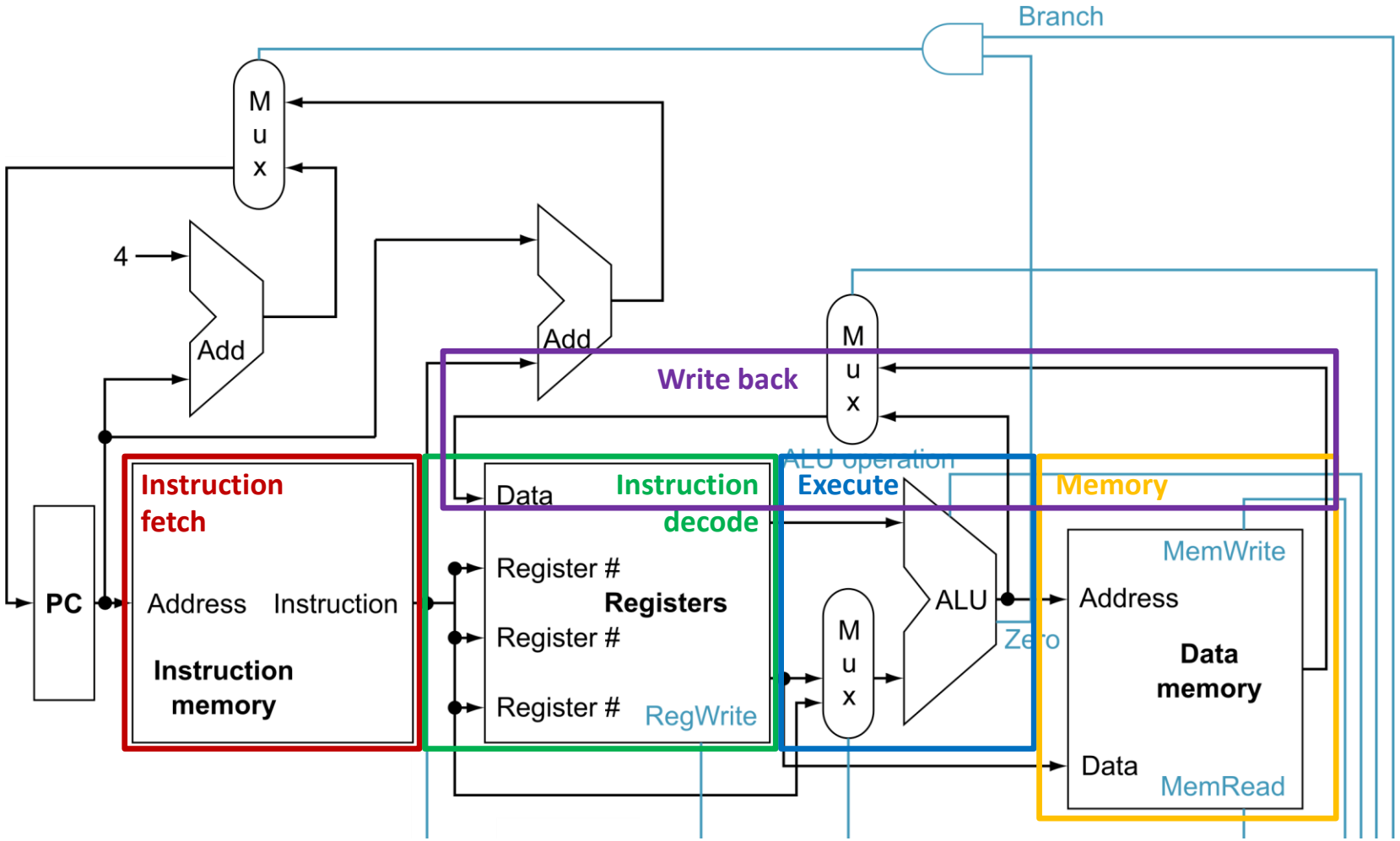
Instruction Execution Phases



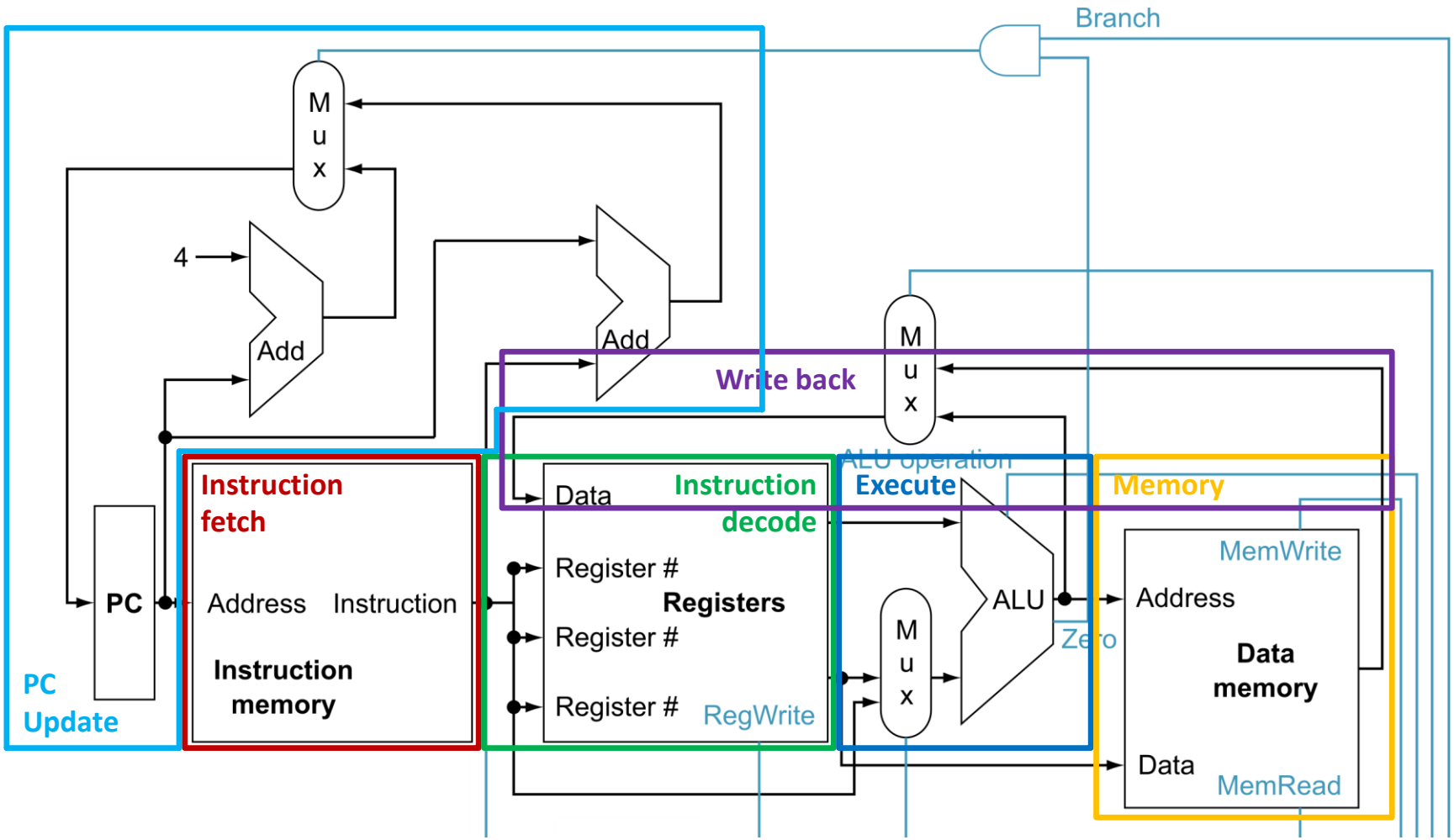
Instruction Execution Phases



Instruction Execution Phases



Instruction Execution Phases



Instruction Execution Phases

	R-type	ld	sd	beq
IF				
ID				
E				
M				
WB				
PC				

Instruction Execution Phases

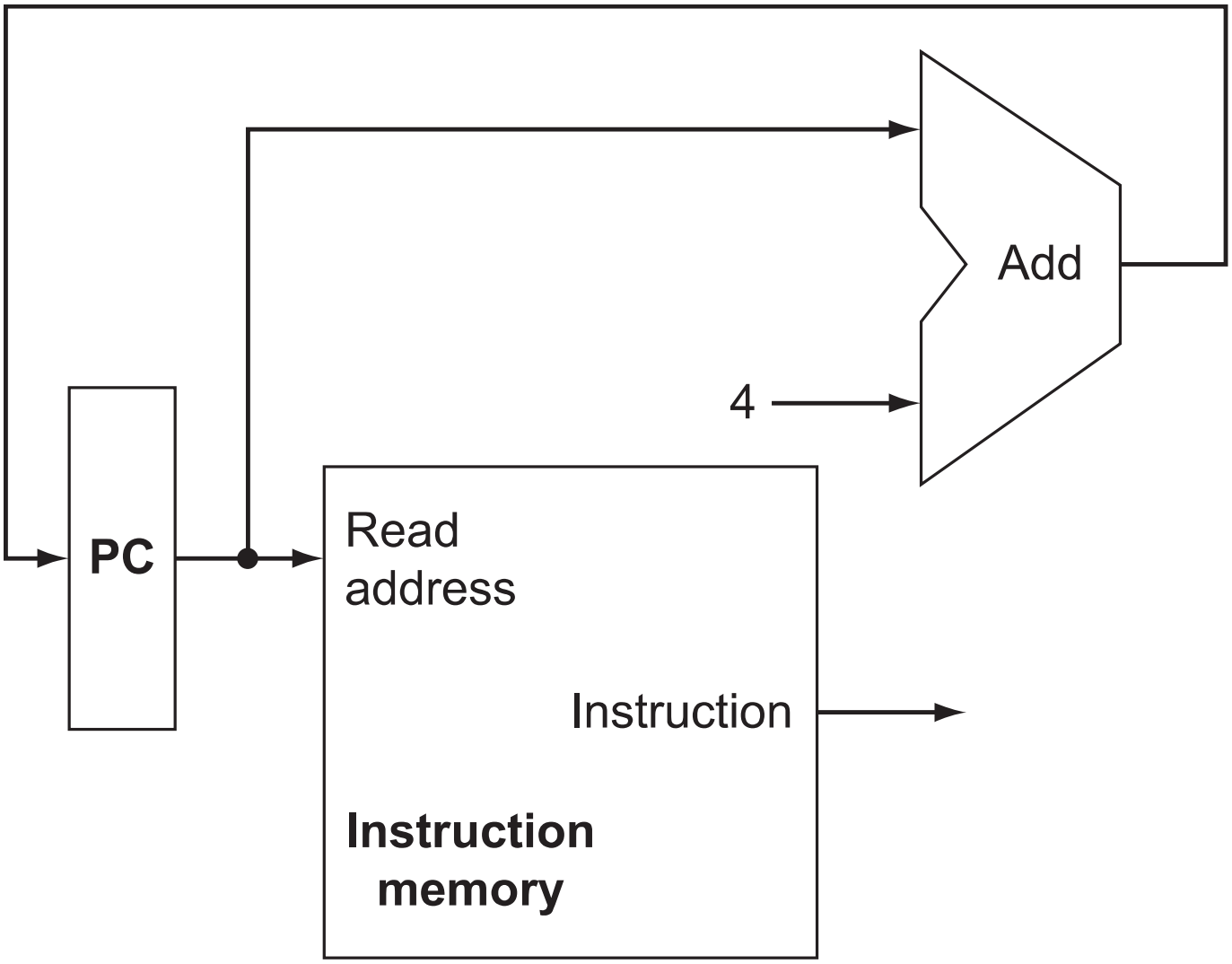
	R-type	ld	sd	beq
IF	inst ← IM_4[PC] valP ← PC + 4	inst ← IM_4[PC] valP ← PC + 4	inst ← IM_4[PC] valP ← PC + 4	inst ← IM_4[PC] valP ← PC + 4
ID	valA ← RF[rs1] valB ← RF[rs2] (register value)	valA ← RF[rs1] valB ← ImmGen(imm) (offset)	valA ← RF[rs1] valB ← ImmGen(imm) valR ← RF[rs2] (saved register)	valA ← RF[rs1] valB ← RF[rs2] valO ← ImmGen(imm) (offset)
E	valE ← valA OP valB (operation (add..))	valE ← valA + valB (Set address, offset + Address)	valE ← valA + valB	valZ ← valA-valB == 0 (check zero) valB ← PC + valO<<1 (branch address)
M		valM ← DM_8[valE] (loading data)	DM_8[valE] ← valR (save data)	
WB	RF[rd]← valE	RF[rd]← valM (save in rd register)		PC ← valZ ? valB:valP (Branching)
PC	PC ← valP	PC ← valP	PC ← valP	

Instruction Fetch

	R-type	ld	sd	beq
IF	inst ← IM_4[PC] valP ← PC + 4	inst ← IM_4[PC] valP ← PC + 4	inst ← IM_4[PC] valP ← PC + 4	inst ← IM_4[PC] valP ← PC + 4

- Send PC to instruction memory
- Fetch instruction: IMEM[PC]
- Compute address of next instruction

Instruction Fetch

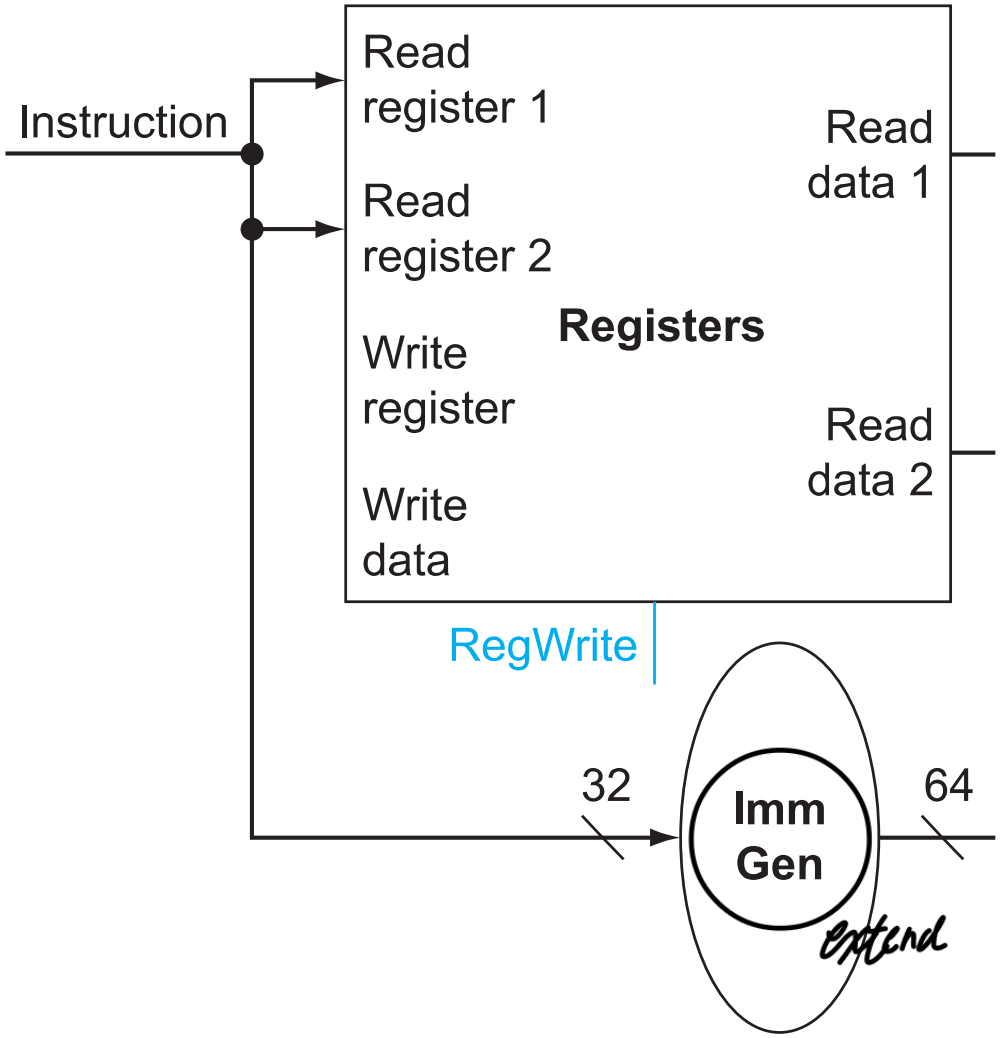


Instruction Decode

	R-type	ld	sd	beq
ID	valA ← RF[rs1] valB ← RF[rs2]	valA ← RF[rs1] valB ← ImmGen(imm)	valA ← RF[rs1] valB ← ImmGen(imm) valR ← RF[rs2]	valA ← RF[rs1] valB ← RF[rs2] valO ← ImmGen(imm)

- Read registers
- Generate constant

Instruction Decode

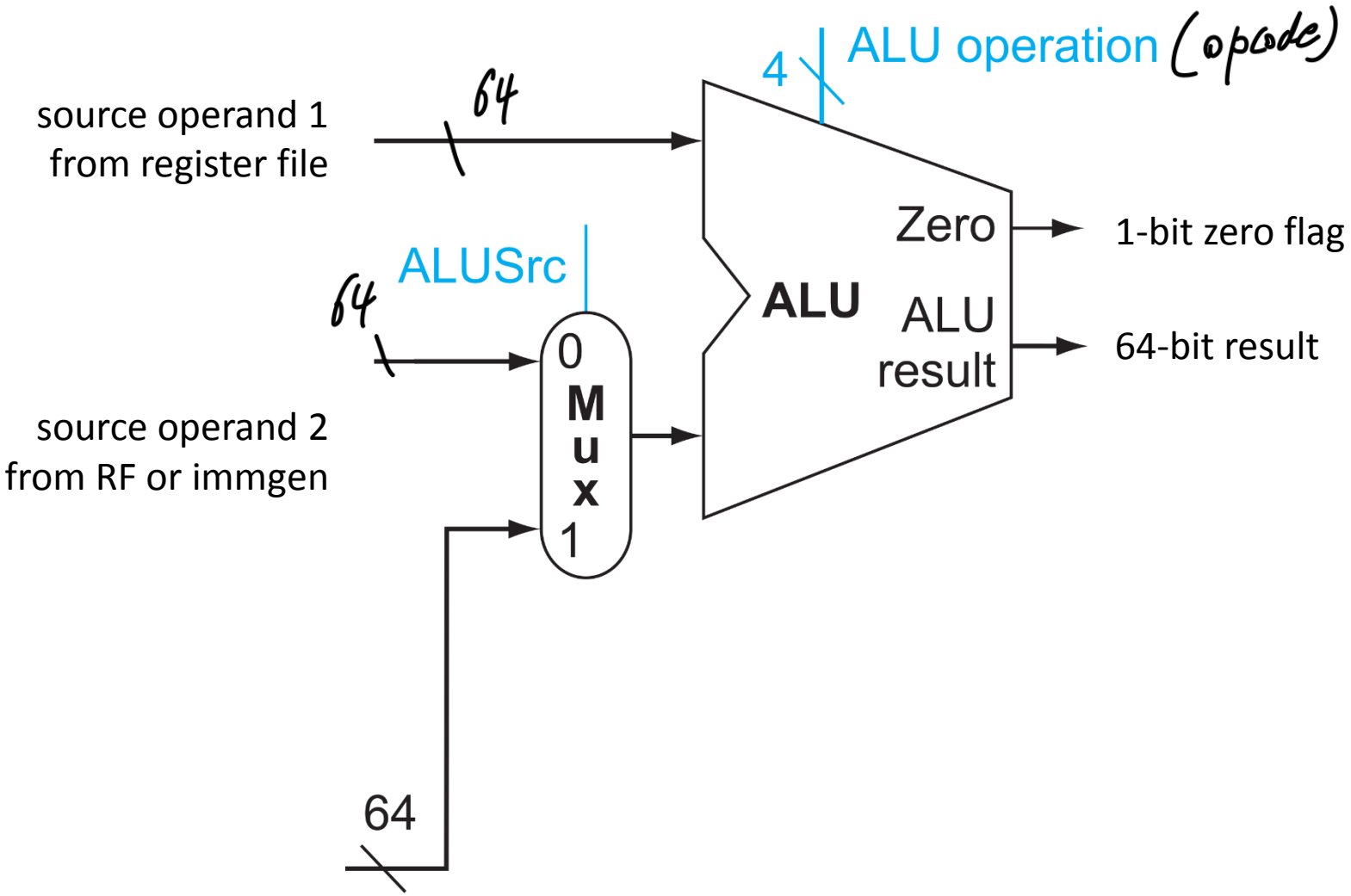


Execute

	R-type	ld	sd	beq
E	valE ← valA OP valB	valE ← valA + valB	valE ← valA + valB	valZ ← valA-valB == 0 valB ← PC + valO<<1

- All instructions
  - ALU computes result of two operands
  - R-type, B-type: generate desired result
  - load/store: compute memory address

Execute

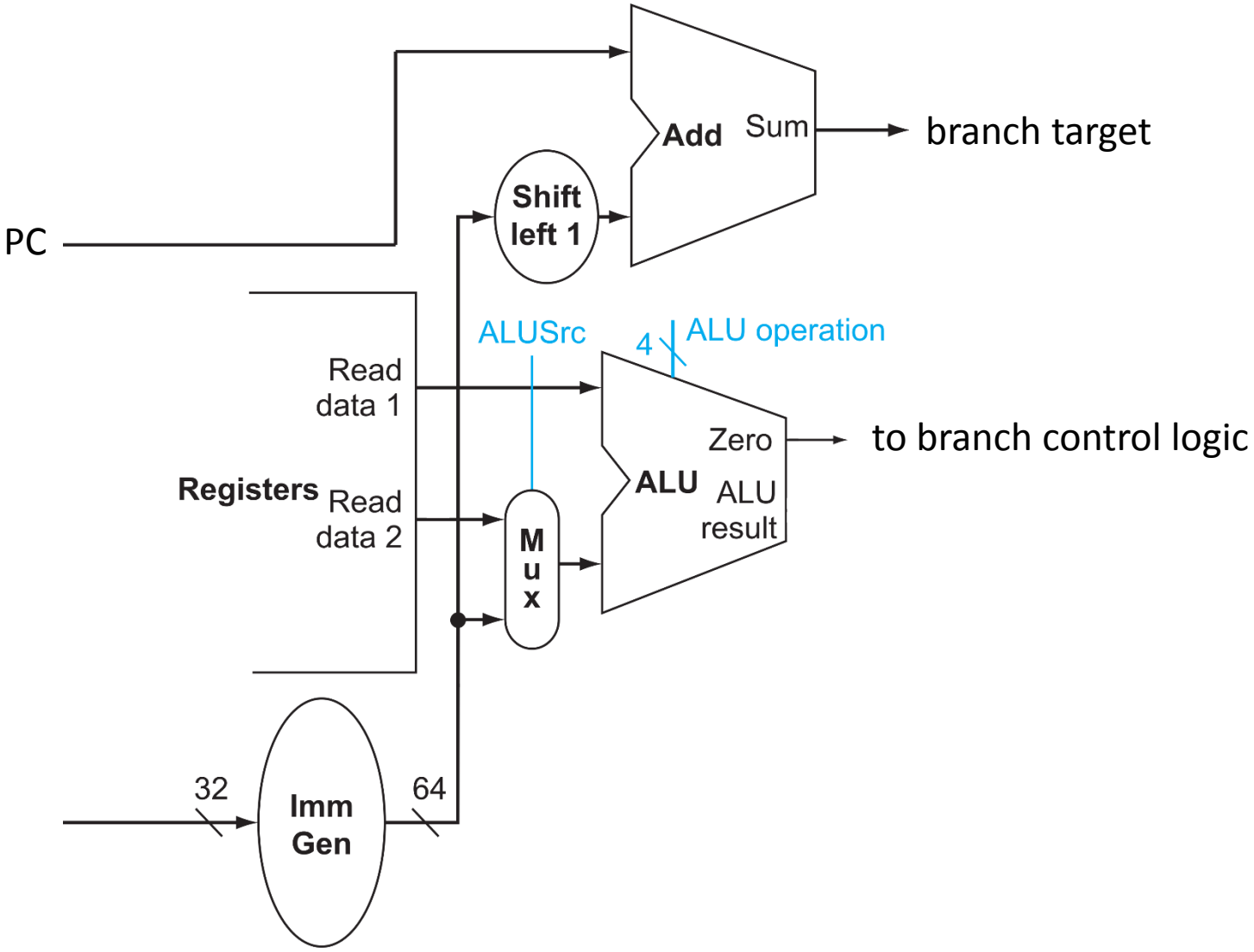


Execute – Compute New PC

	beq
E	valZ ← valA-valB == 0 valB ← PC + valO<<1

- Branch instruction
  - compute PC + offset

Execute – Compute New PC

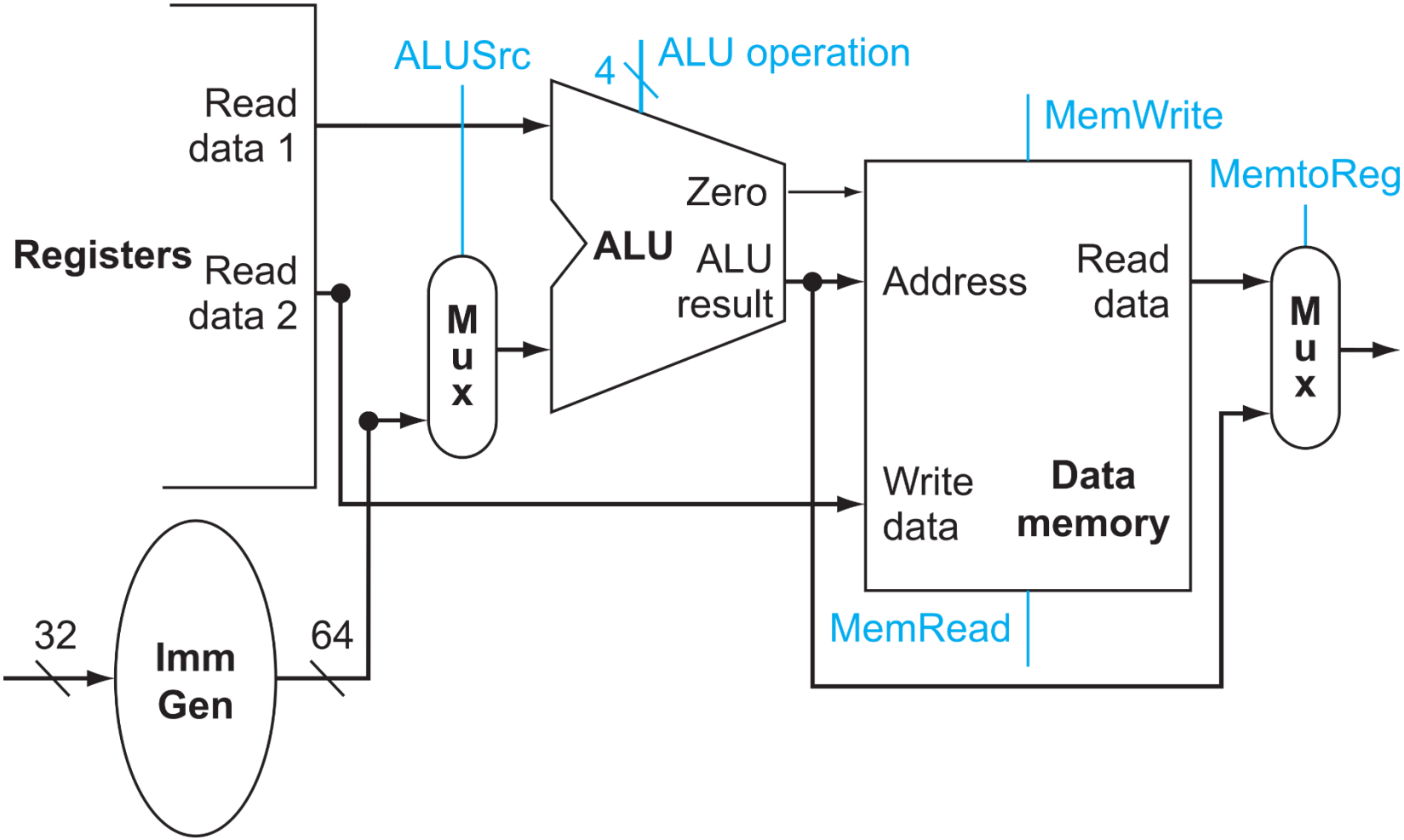


Memory

	R-type	ld	sd	beq
M		valM ← DM_8[valE]	DM_8[valE] ← valR	

- Only active for memory operations
  - load (I-type): read value at computed address
  - store (S-type): write value to computed address

Memory

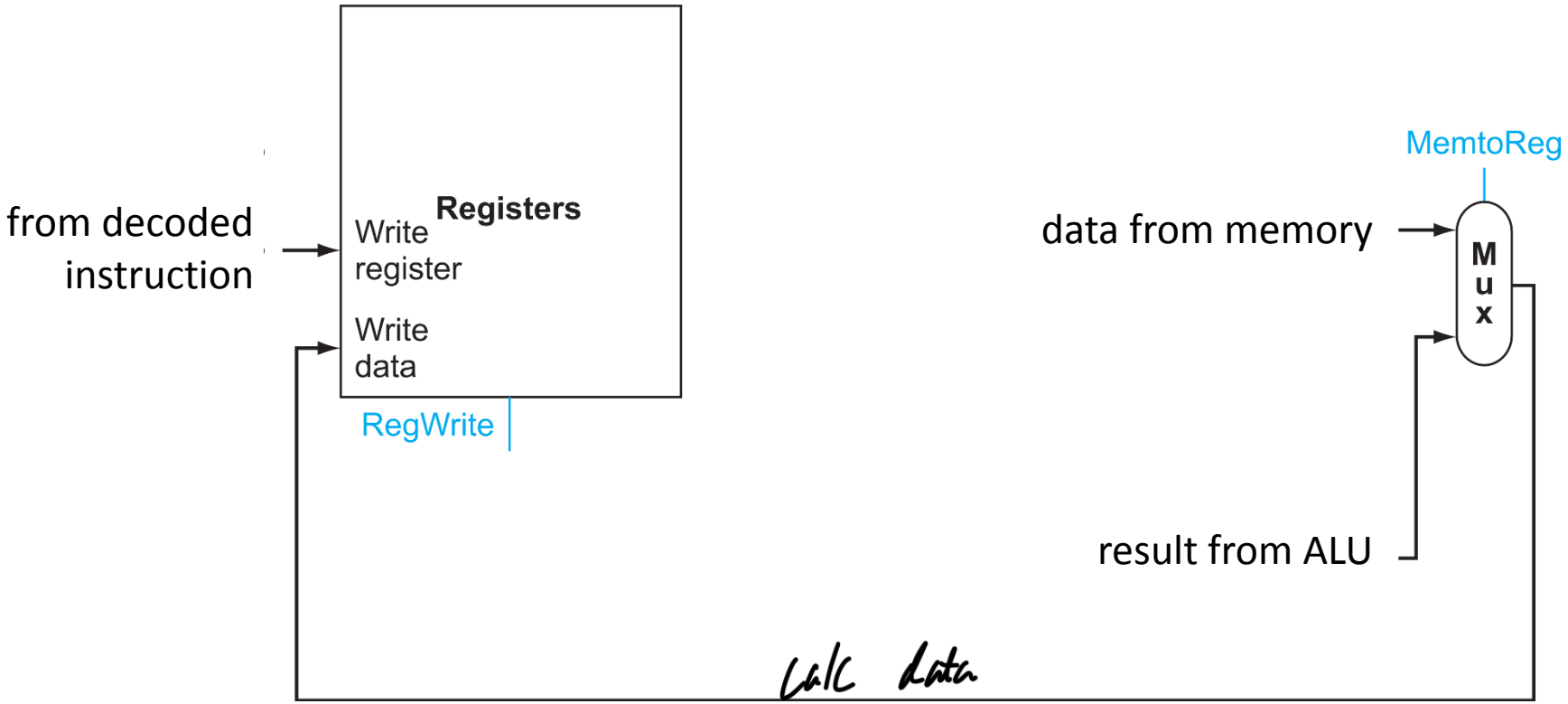


Write-Back

	R-type	ld	sd	beq
WB	RF[rd]← valE	RF[rd]← valM		

- Write value back to register file
  - R-type: computed value
  - load: value obtained from memory

Write-Back

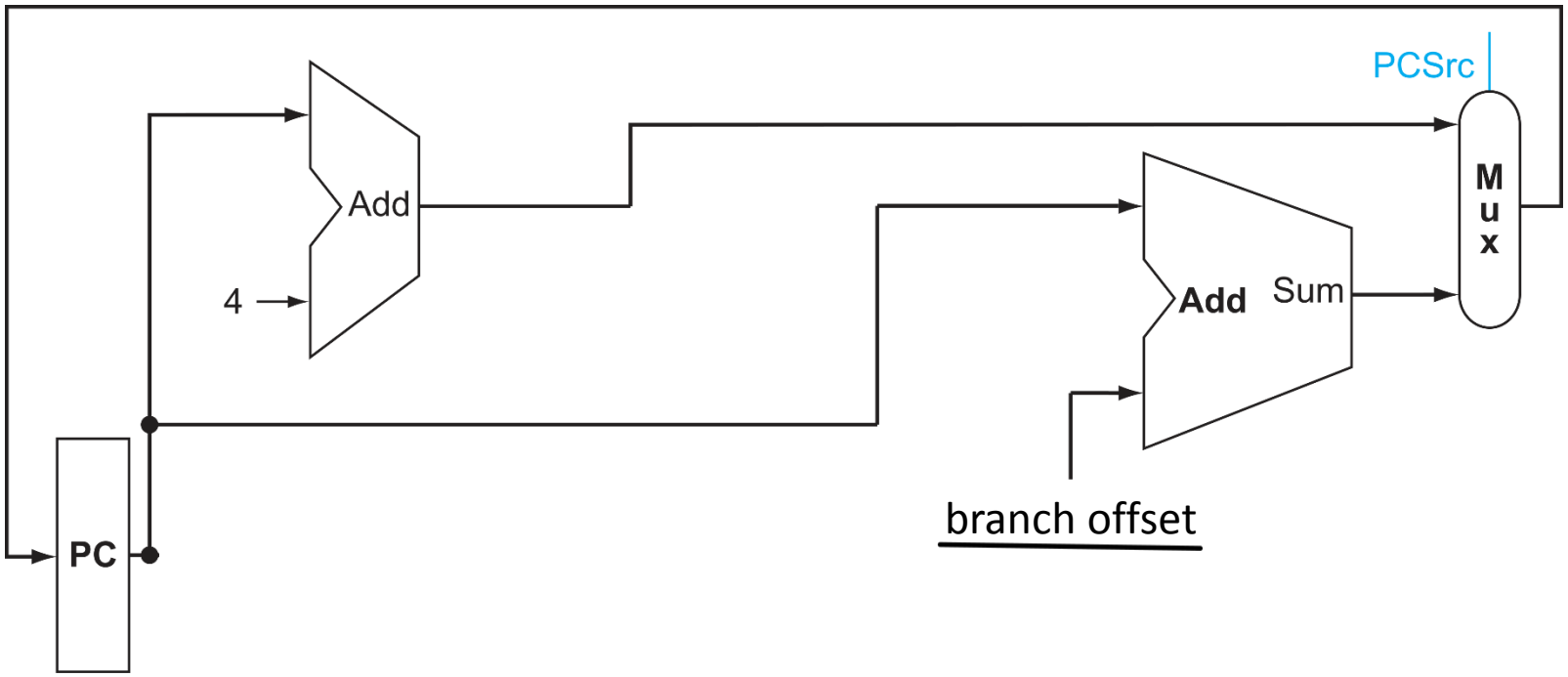


Update PC

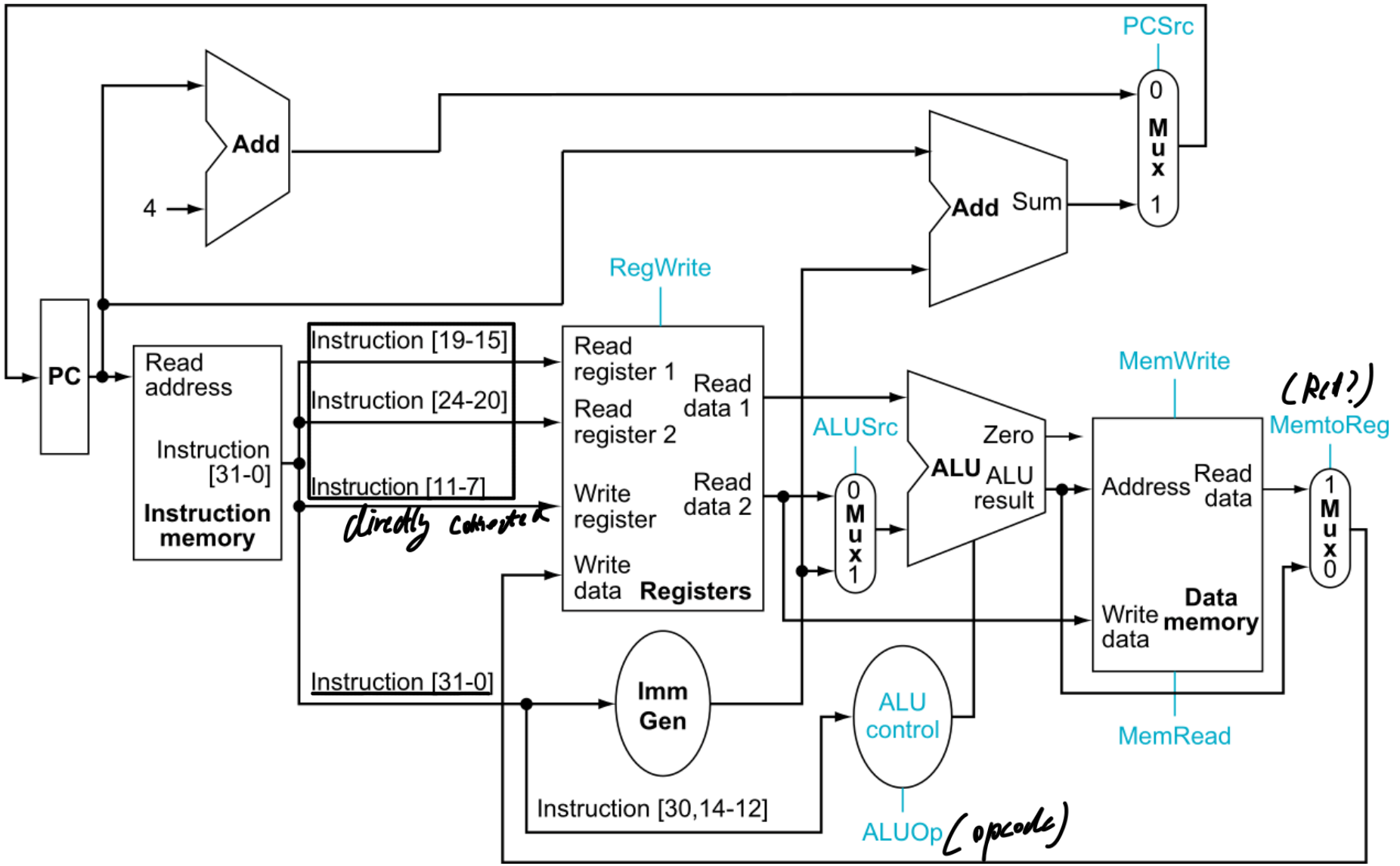
	R-type	ld	sd	beq
PC	PC ← valP	PC ← valP	PC ← valP	PC ← valZ ? valB:valP

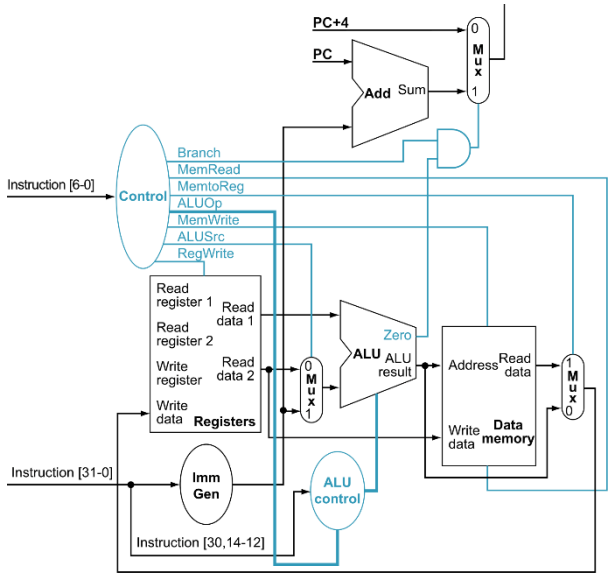
- Set new PC
  - computed branch target for taken beq
  - otherwise PC+4

Update PC



Full Datapath

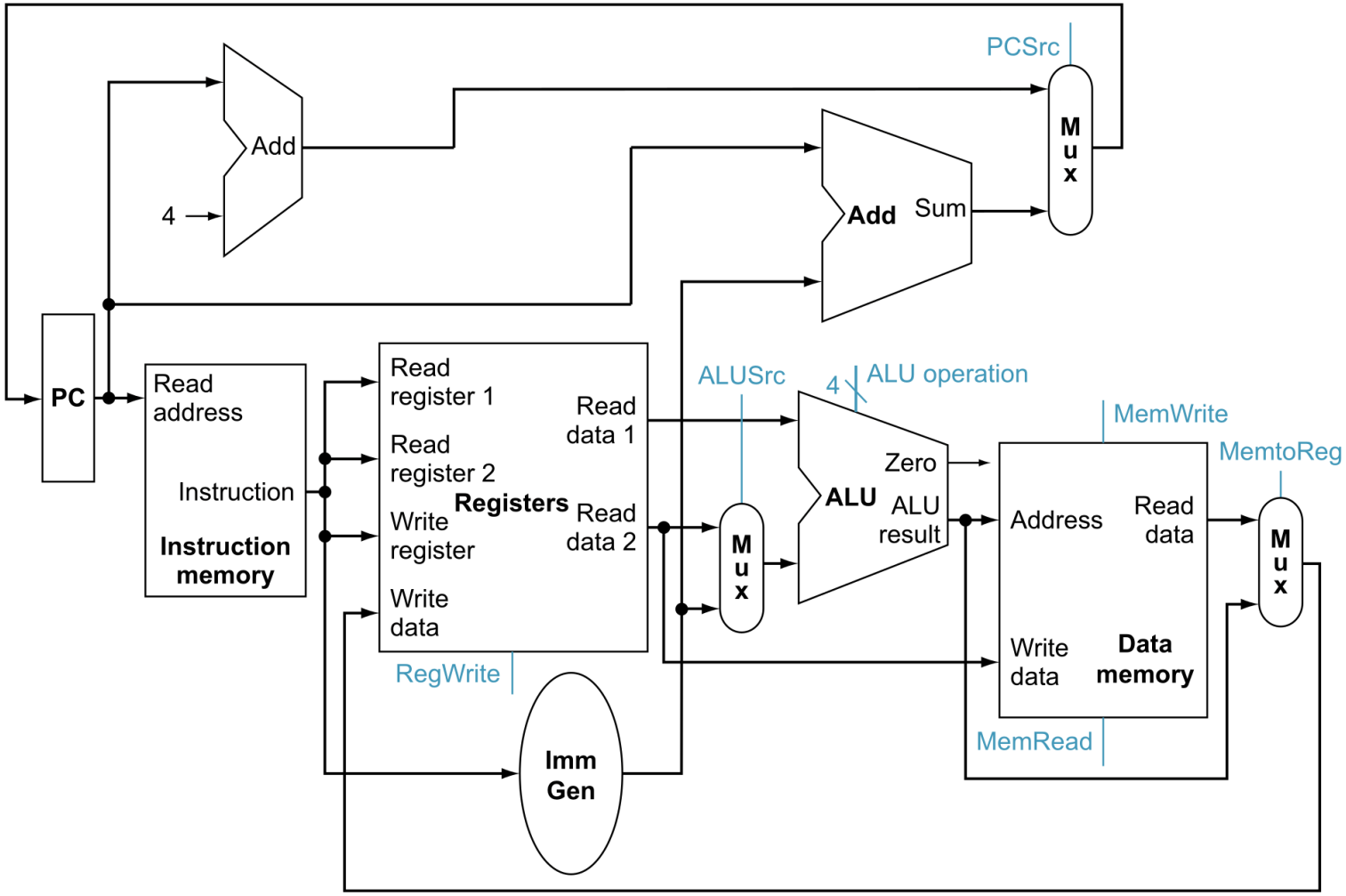




Building the Control Path

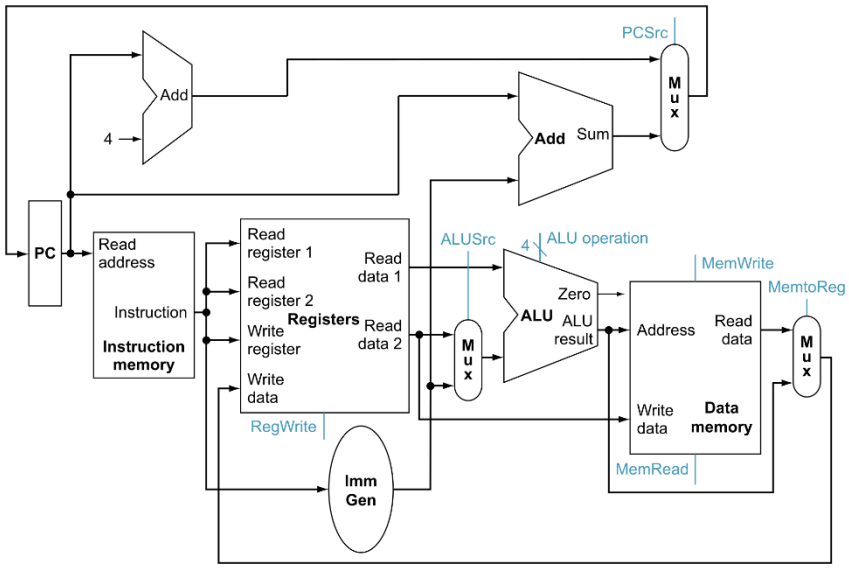
Control Path

- Our processor contains several yet-to-be-defined control signals



Control Signals

- Six 1-bit signals



Signal	Low	High
RegWrite	Nothing (do not update register file)	Write value at *Write data*$_{RF}$ port to register[*Write register*]
ALUSrc	Second ALU operand = *Read data 2*	Second ALU operand = sign-extended, 12 bits of instruction
PCSrc	new PC = old PC + 4	new PC = result of branch target adder
MemRead	Nothing (do not read memory)	*Read data* = mem[*Address*]
Mem Write	Nothing (do not write memory)	mem[*Address*] = *Write data*$_{DM}$
MemtoReg	*Write data*$_{RF}$ = output of ALU	*Write data*$_{RF}$ = output of data memory

ALU Control Signals

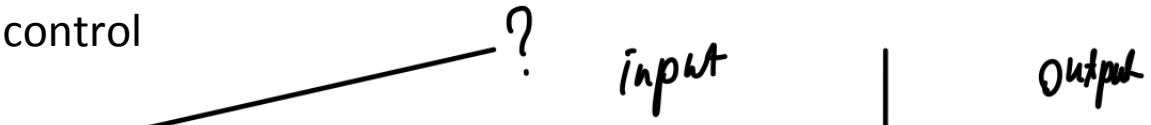
- The RISC-V ALU (textbook Appendix A) defines the following control signals

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

- The ALU will be used for
  - R-type: Function depends on opcode
  - Load/Store: Function = add
  - Branch: Function = subtract

ALU Control Signals (cont'd)

- Combinational logic derives ALU control
  - Input
    - 2-bit ALUOp derived from opcode field
    - funct7 and funct3 from instruction
  - Output: ALU control



opcode	ALUOp	Operation	funct7	funct3	ALU function	ALU control
ld	00	load register	XXXXXXX	XXX	add	0010
sd	00	store register	XXXXXXX	XXX	add	0010
beq	01	branch on equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
		subtract	0100000	000	subtract	0110
		AND	0000000	111	AND	0000
		OR	0000000	110	OR	0001

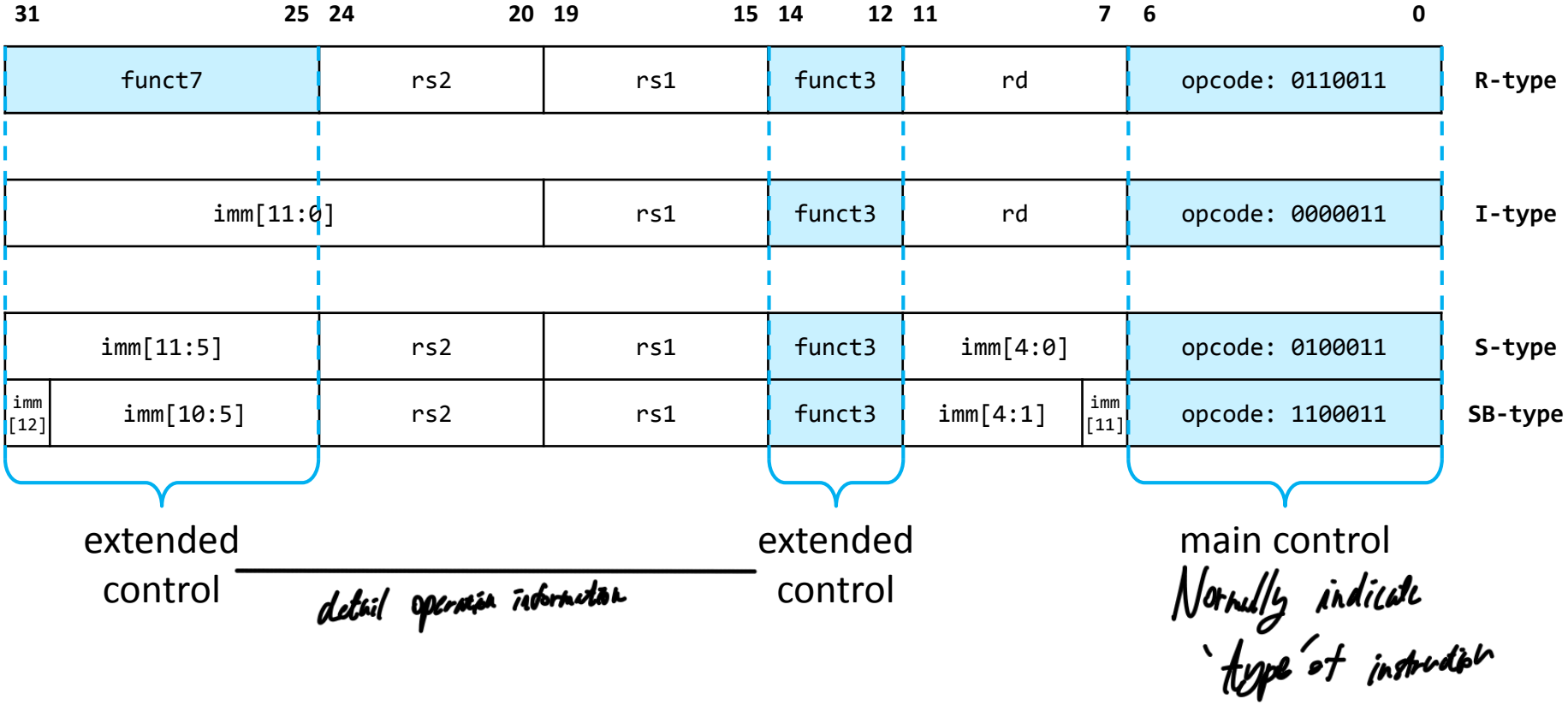
ALU Control Signals (cont'd)

- ALU control truth table
  (only showing entries for which the ALU control must have a specific value)

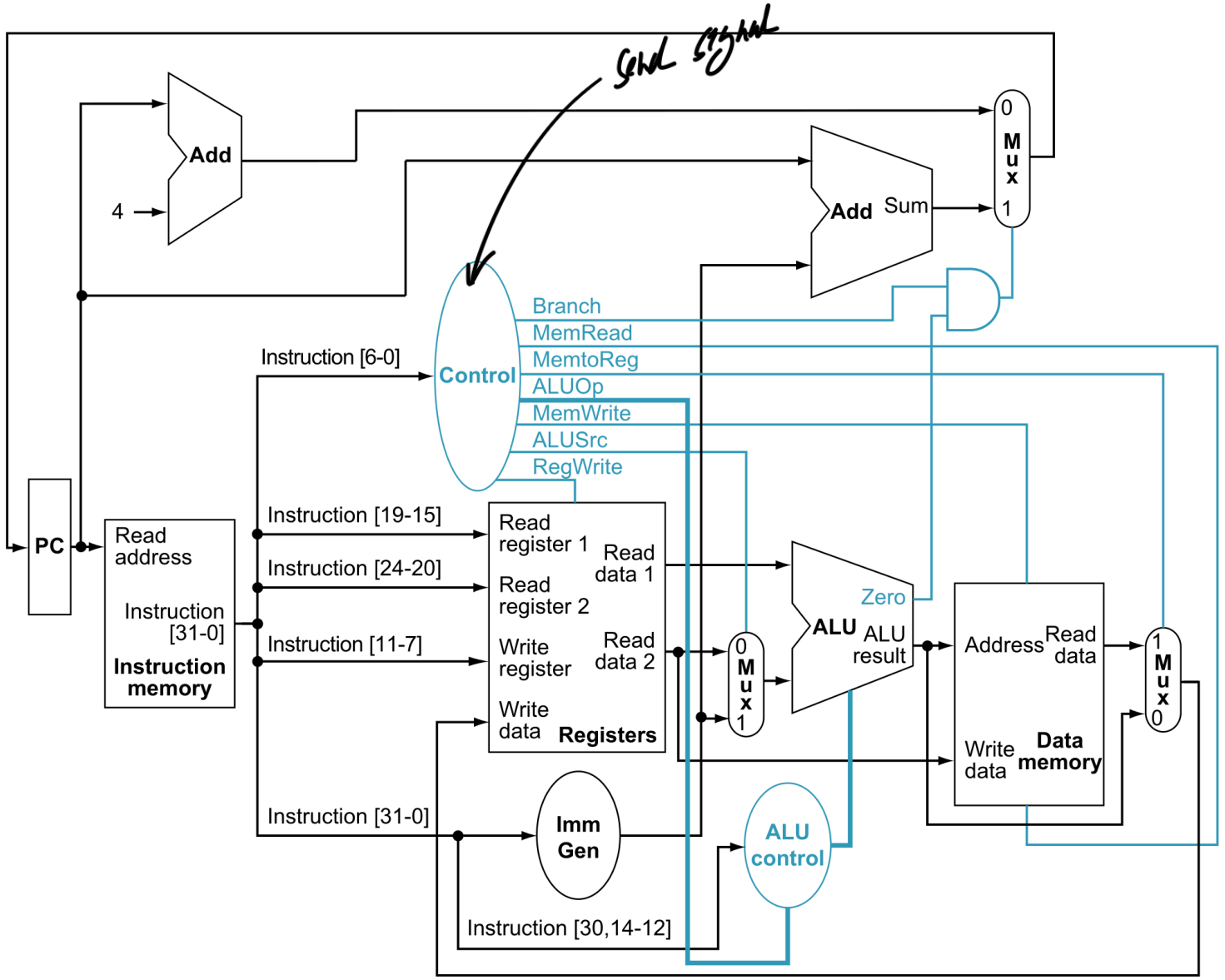
ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

The Main Control Unit

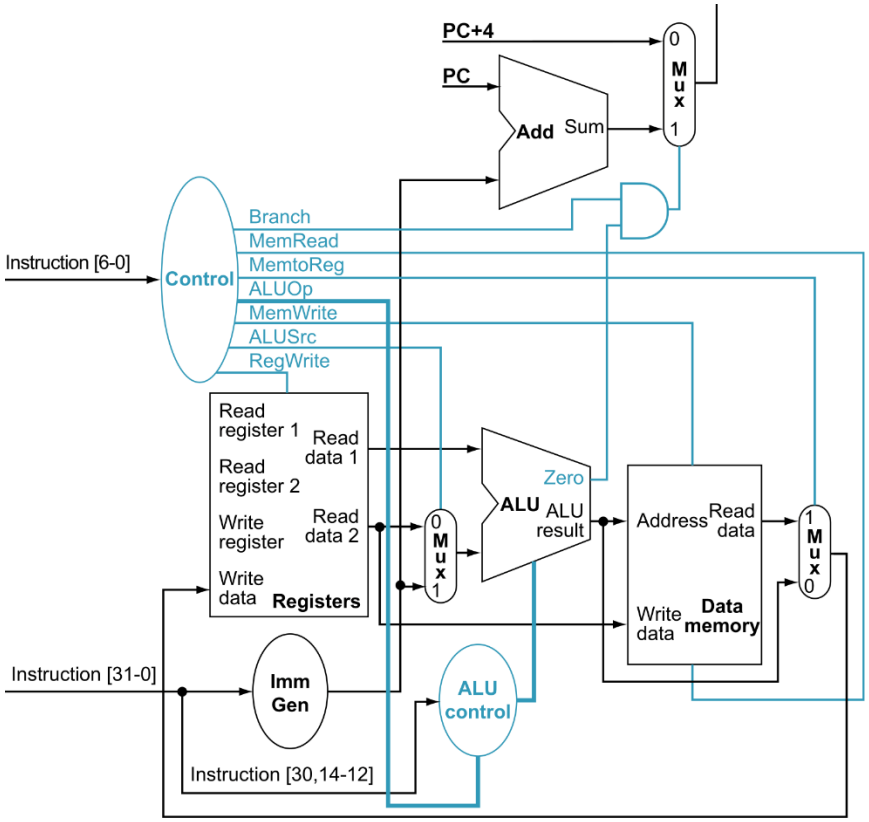
- Derive necessary control signals from instruction



Datapath with Control



Datapath with Control



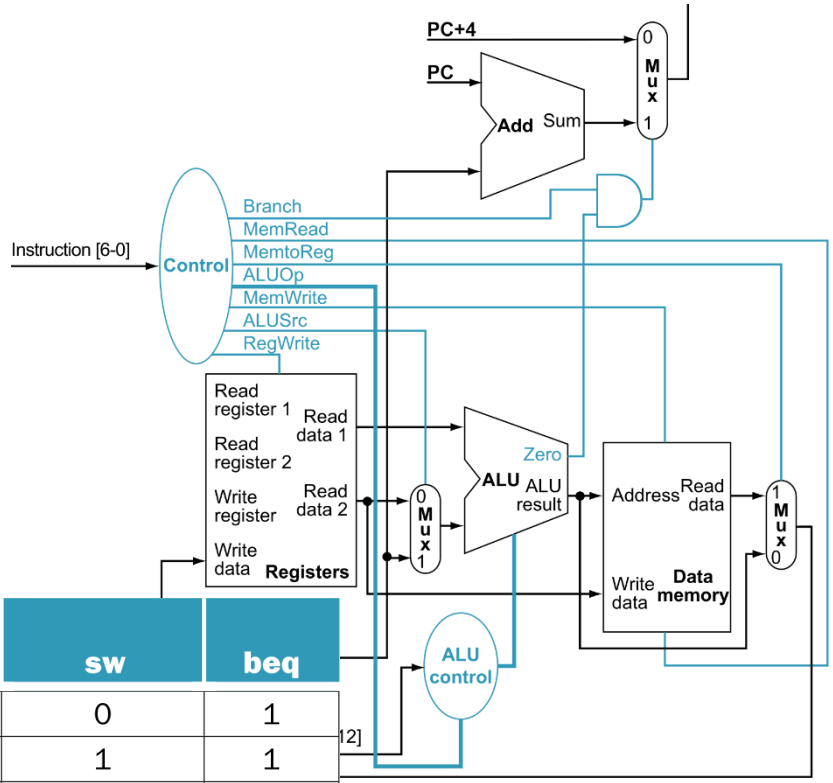
Imm or SR2

need memtoReg?

Write back?

Instruction	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
R-type	0	0	1	0	0	0	10
ld	1	1	1	1	0	0	00
sd	1	X	0	0	1	0	00
beq	0	X	0	0	0	1	01

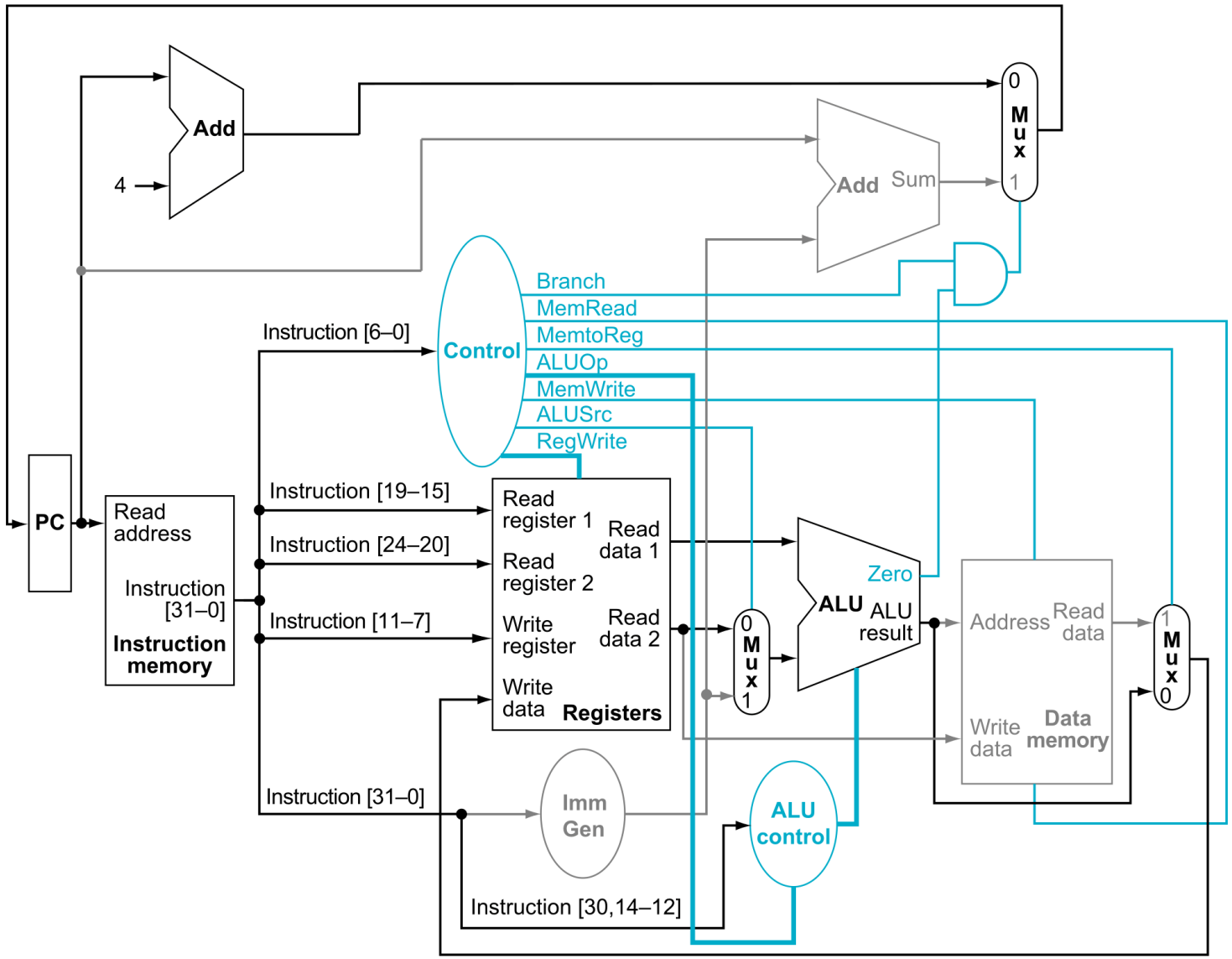
Datapath with Control



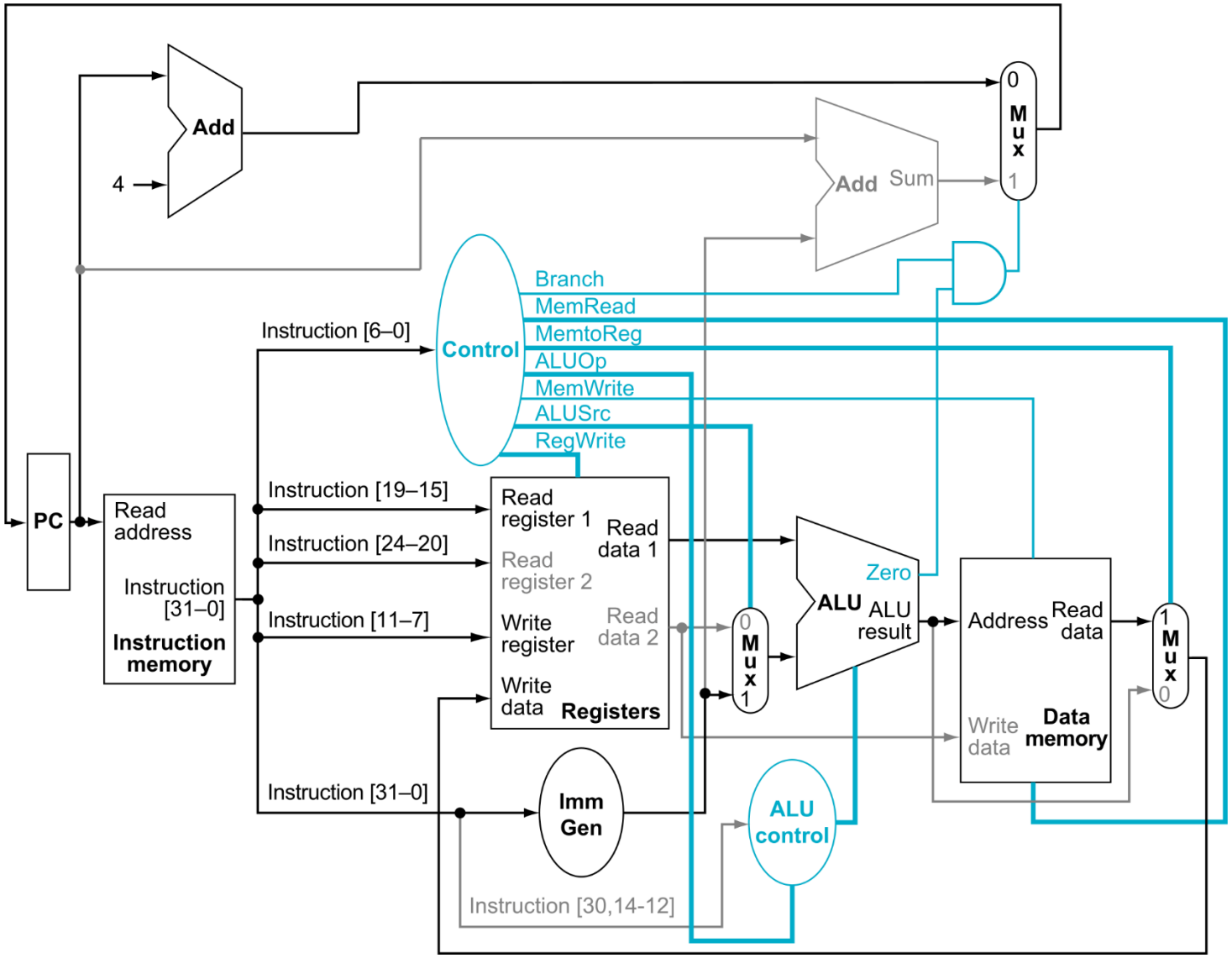
opcode

Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

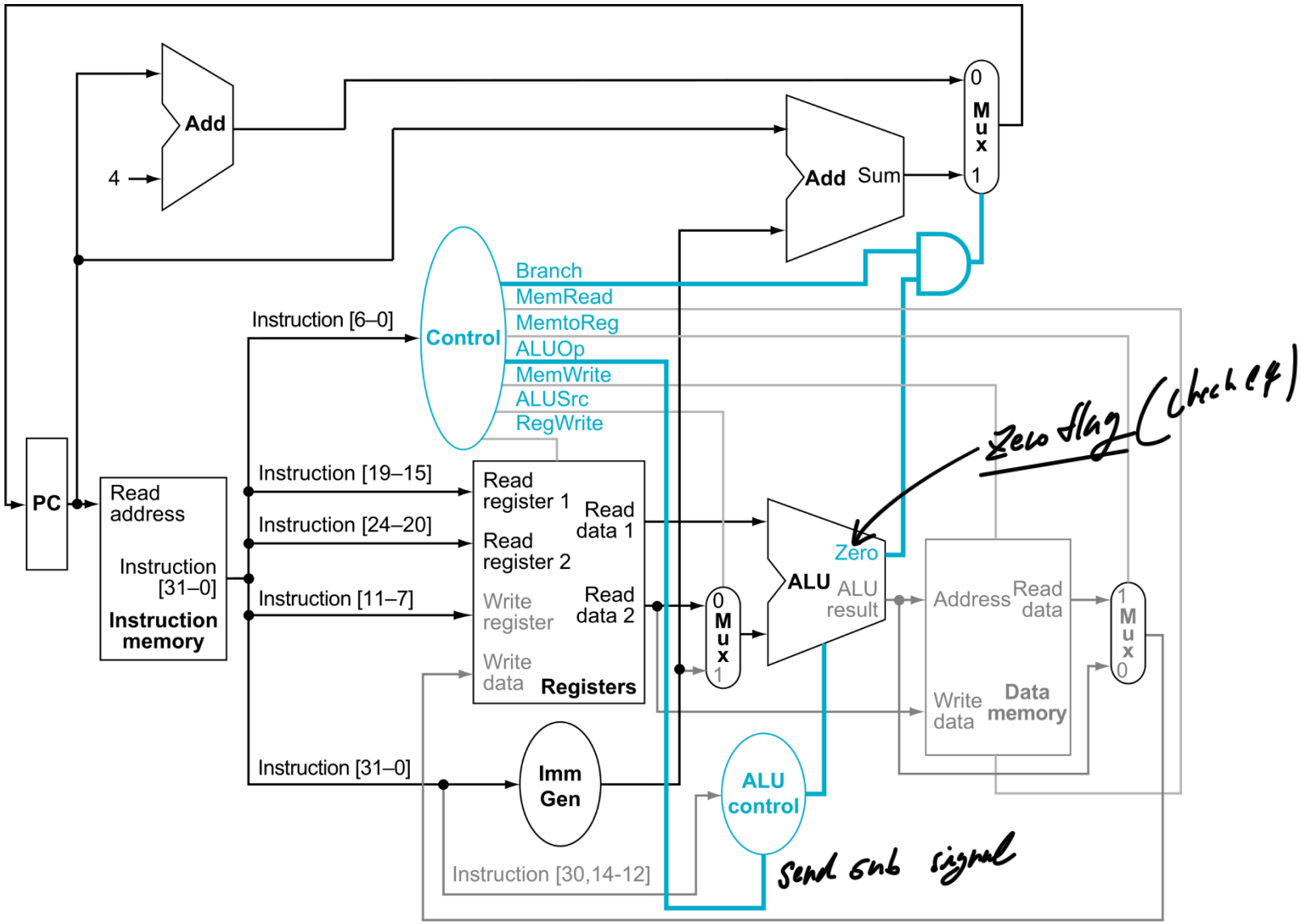
Example: R-Type Instruction

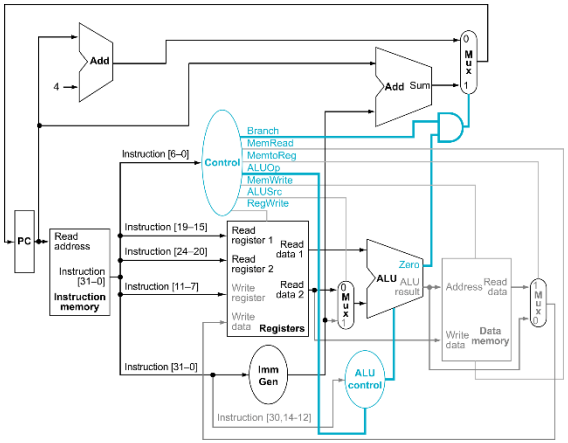


Example: Load Instruction



Example: BEQ Instruction





Module Summary

Sequential Implementation

- **Built from combinational and sequential circuits**
- **Datapath connects circuits**
- **Control path**
  - Generate control signals in dependence of instruction encoding
- **Executes one instruction per cycle**
  - Not used anymore today – too slow