KU-The Future

COSE222: Computer Architecture

Ch. 4 The Processor
(The Single-Cycle Processor)

Gunjae Koo

Department of Computer Science and Engineering

Korea University

KOREA UNIVERSITY

* Based on the lecture slides from “Computer Organization and Design”

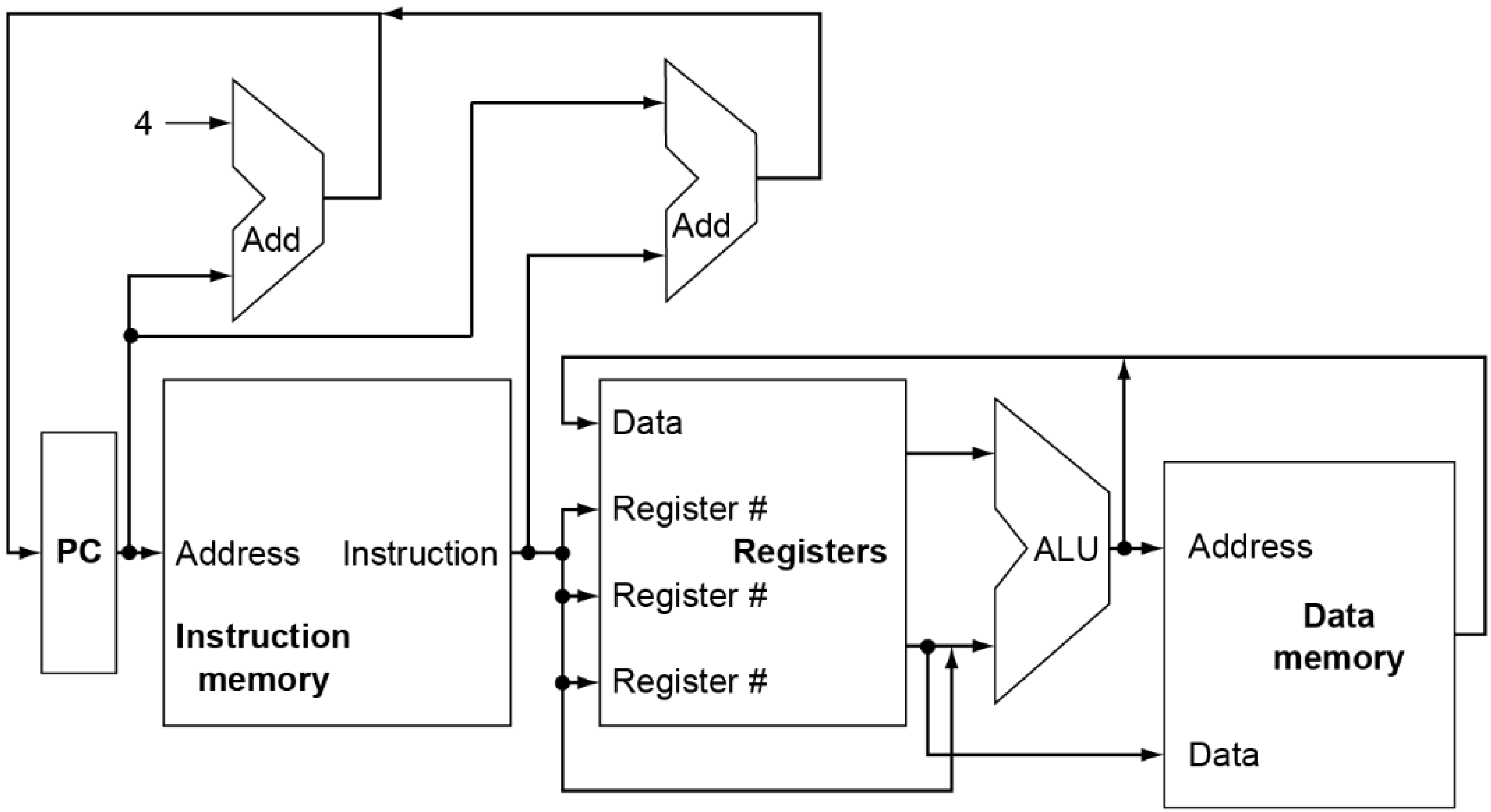
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two RISC-V implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `ld`, `sd`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`
  - Control transfer: `beq`

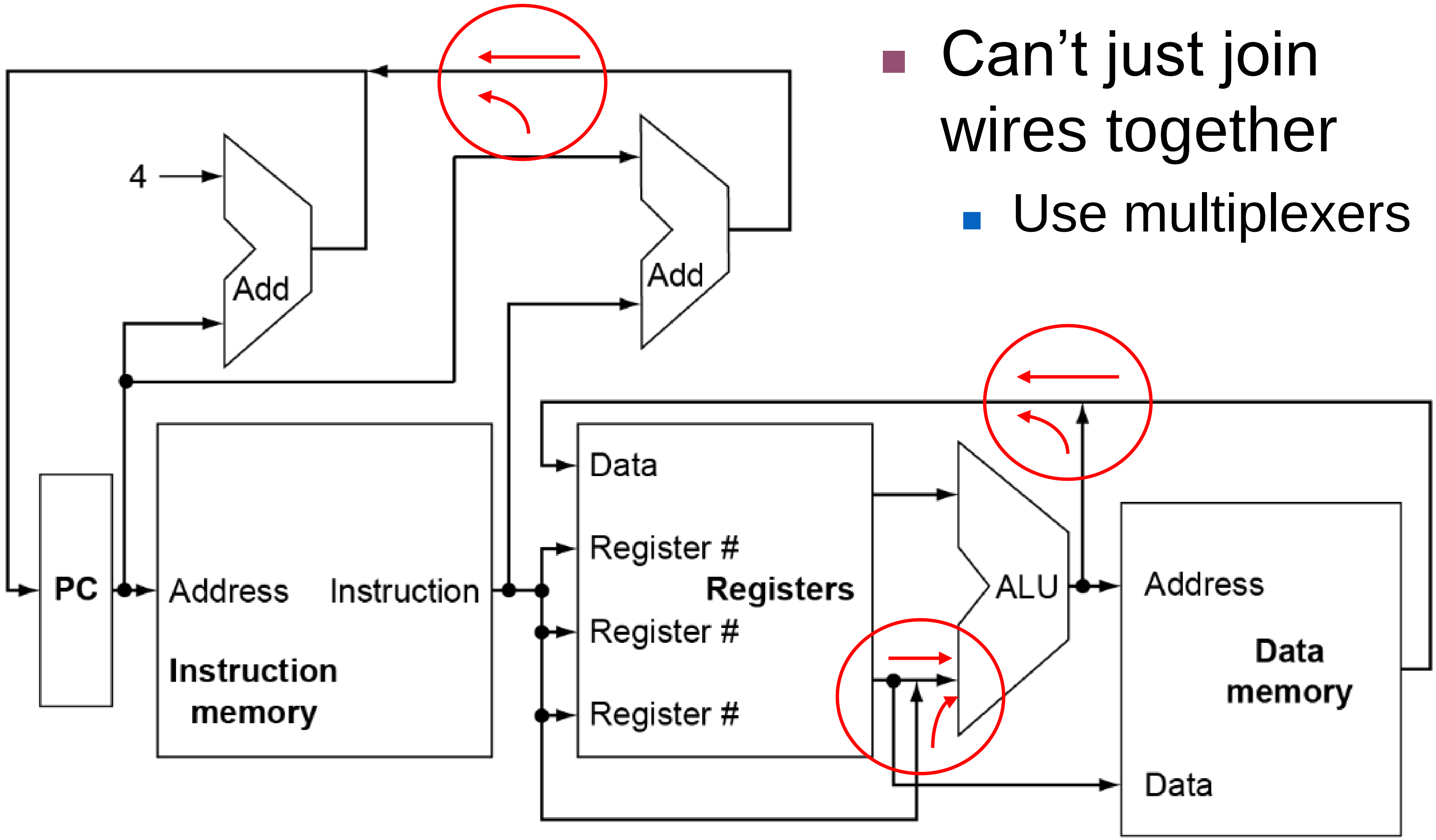
Instruction Execution

- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC $\leftarrow$ target address or PC + 4

CPU Overview

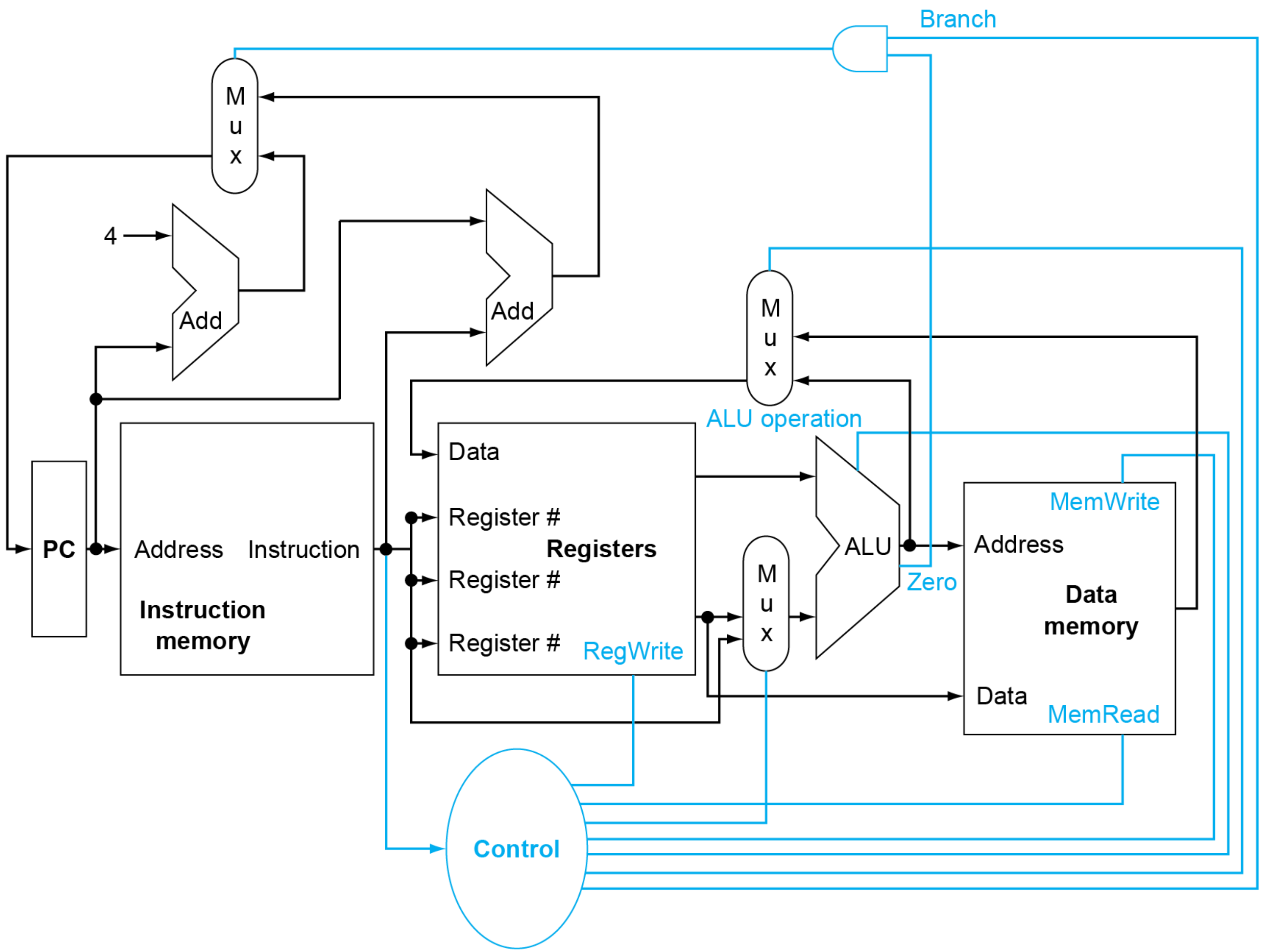


Multiplexers (MUXes)



- Can't just join wires together
  - Use multiplexers

Control



Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

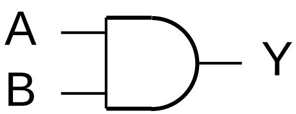
Logic Design Conventions

Logic Design Basics

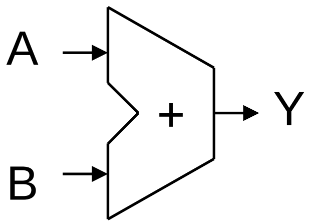
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

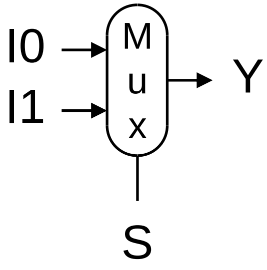
- AND-gate
  - Y = A & B



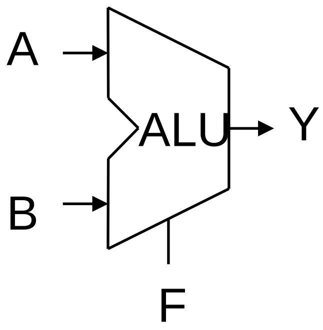
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

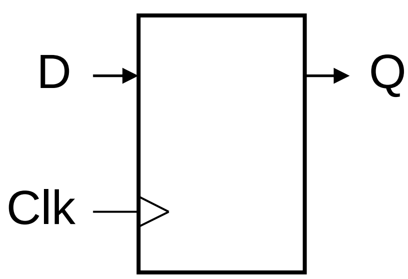


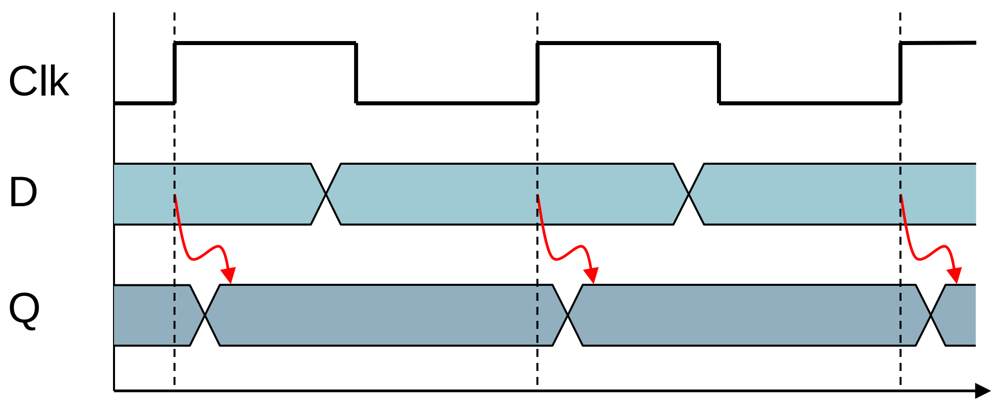
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

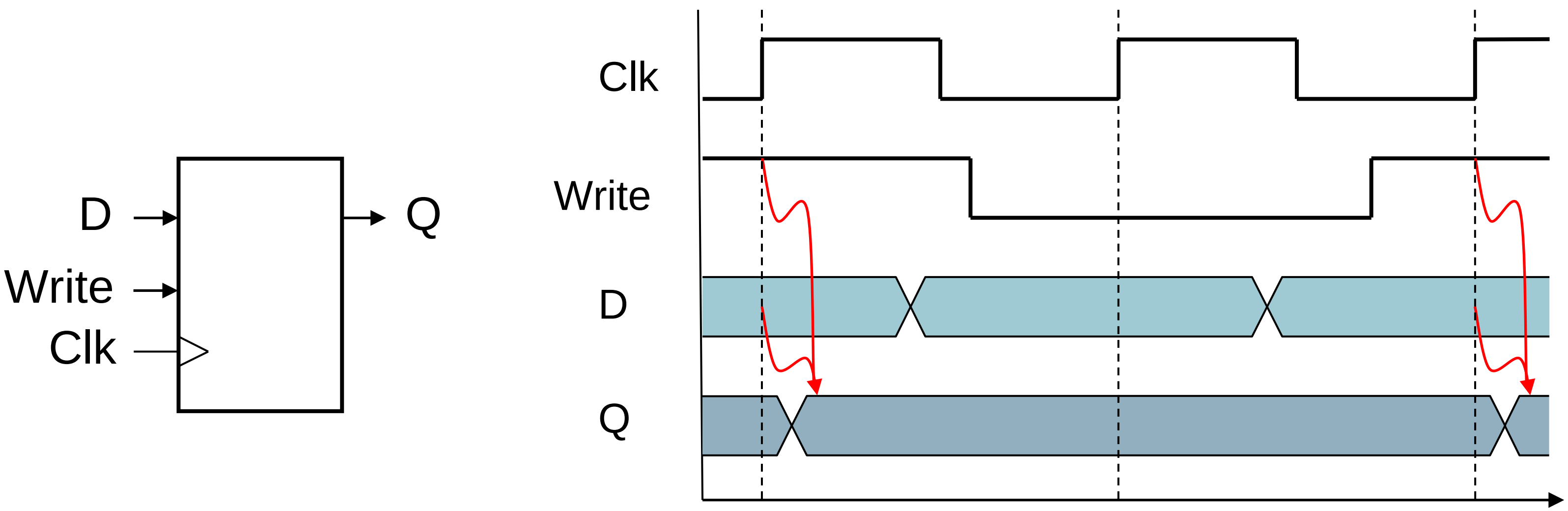
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





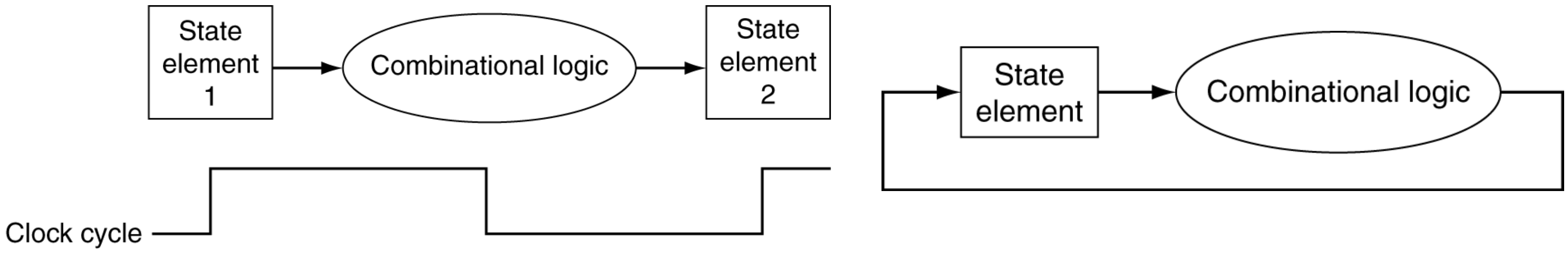
Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

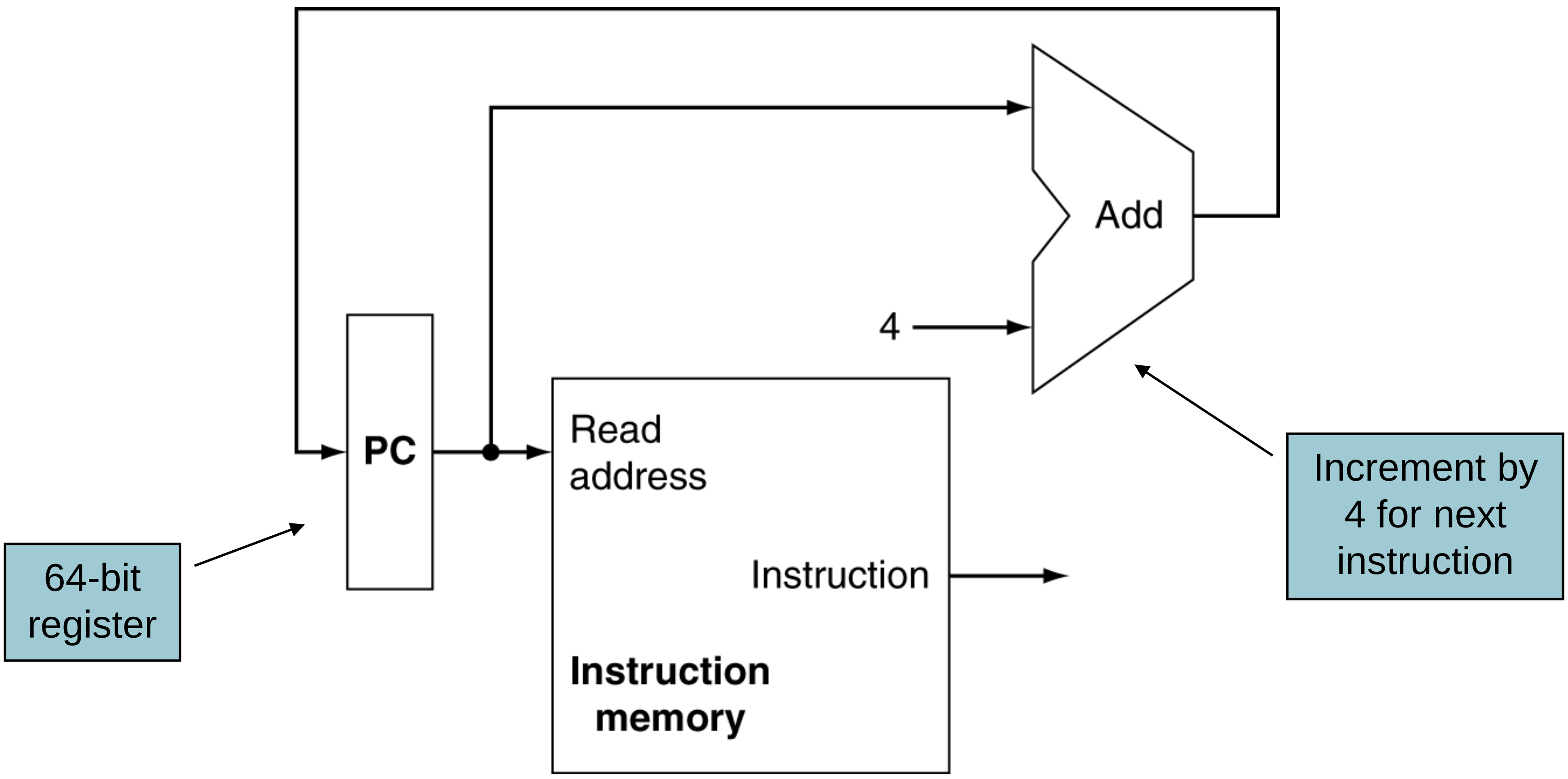


Building a Datapath

Building a Datapath

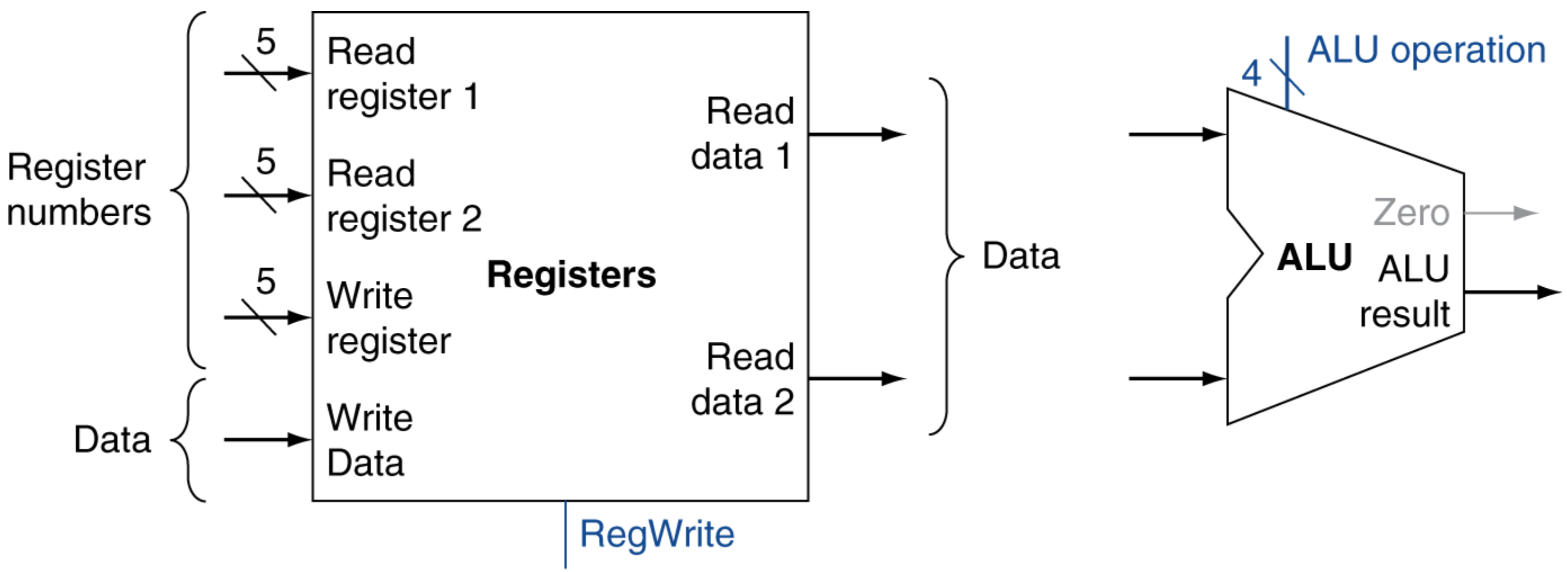
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a RISC-V datapath incrementally
  - Refining the overview design

Instruction Fetch



R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



a. Registers

b. ALU

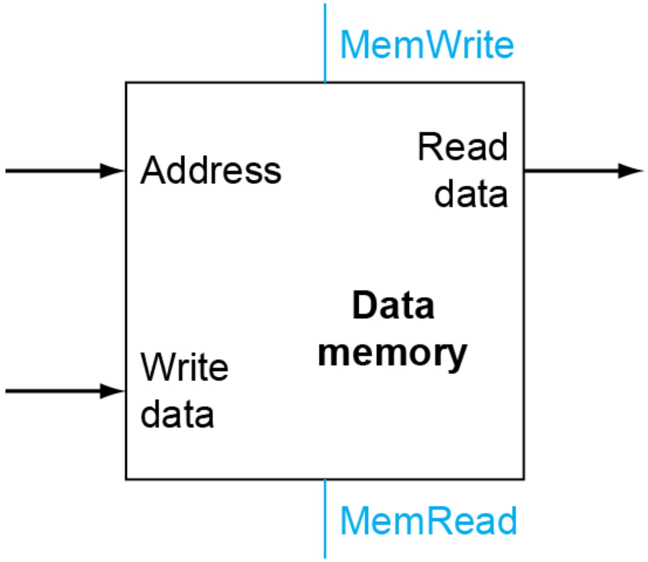
RISC-V R-format Instructions *(review)*

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

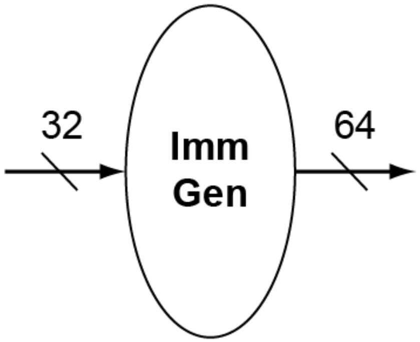
- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit



b. Immediate generation unit

RISC-V I-format Instructions *(review)*

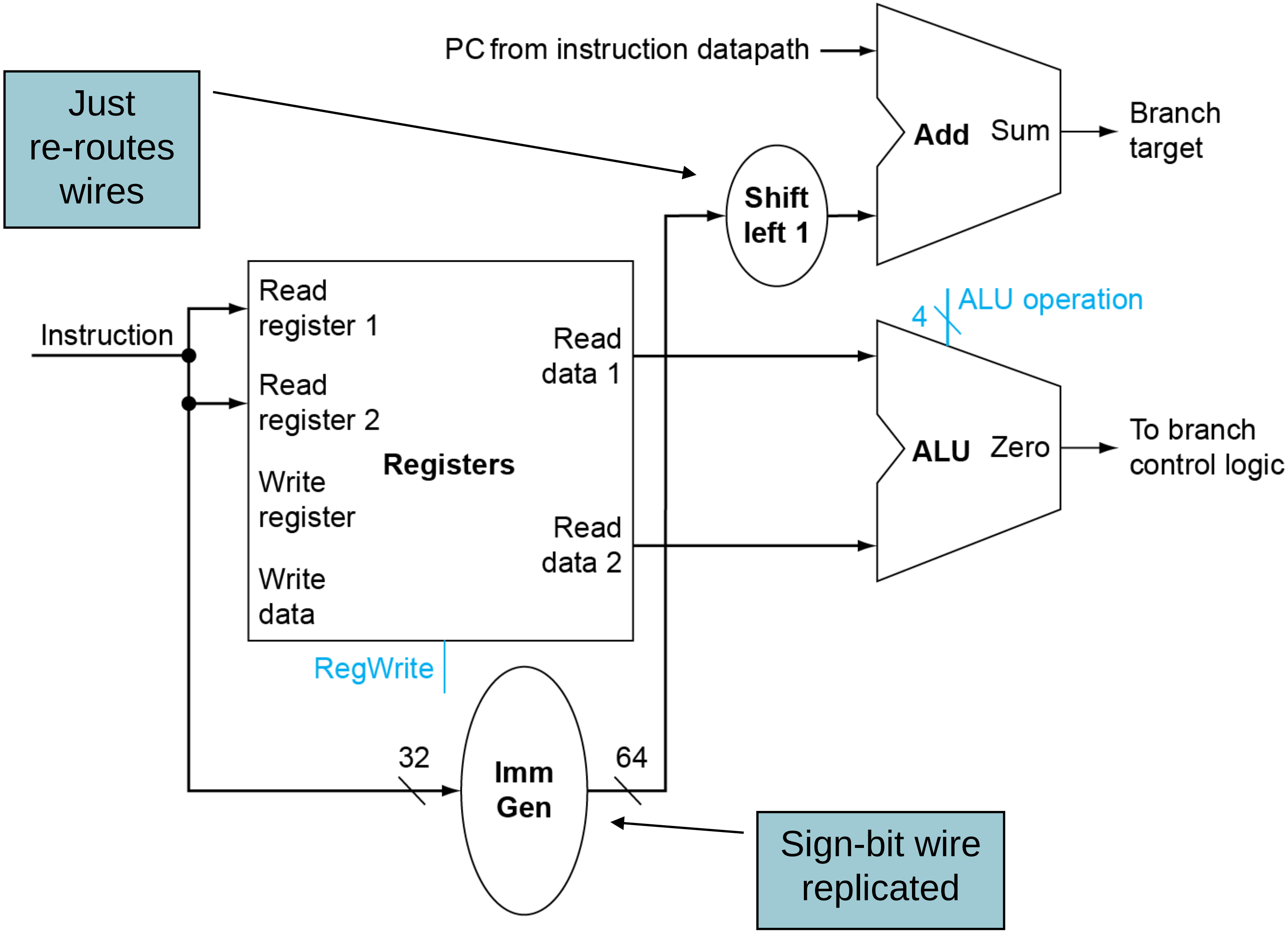
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

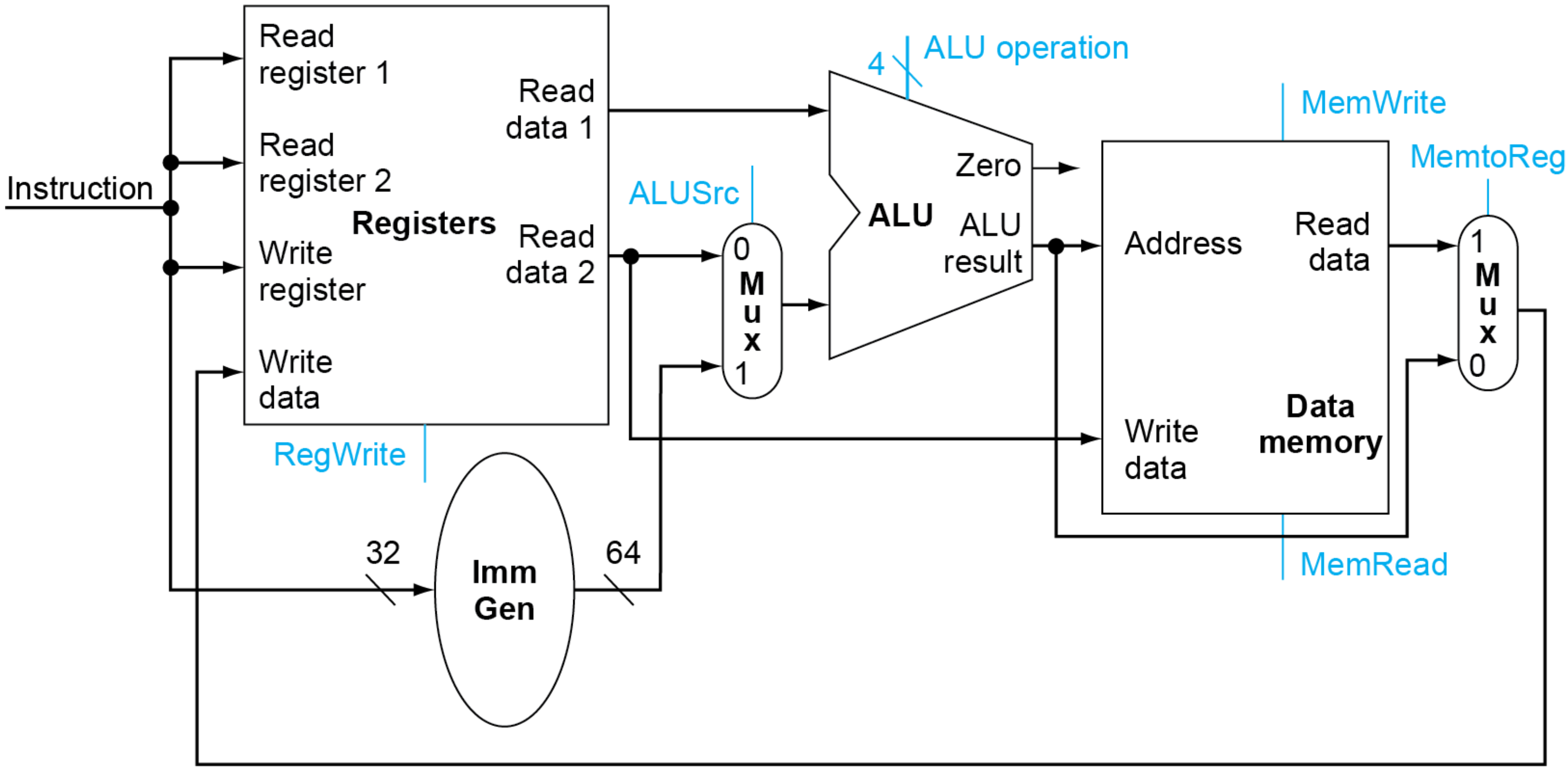
Branch Instructions



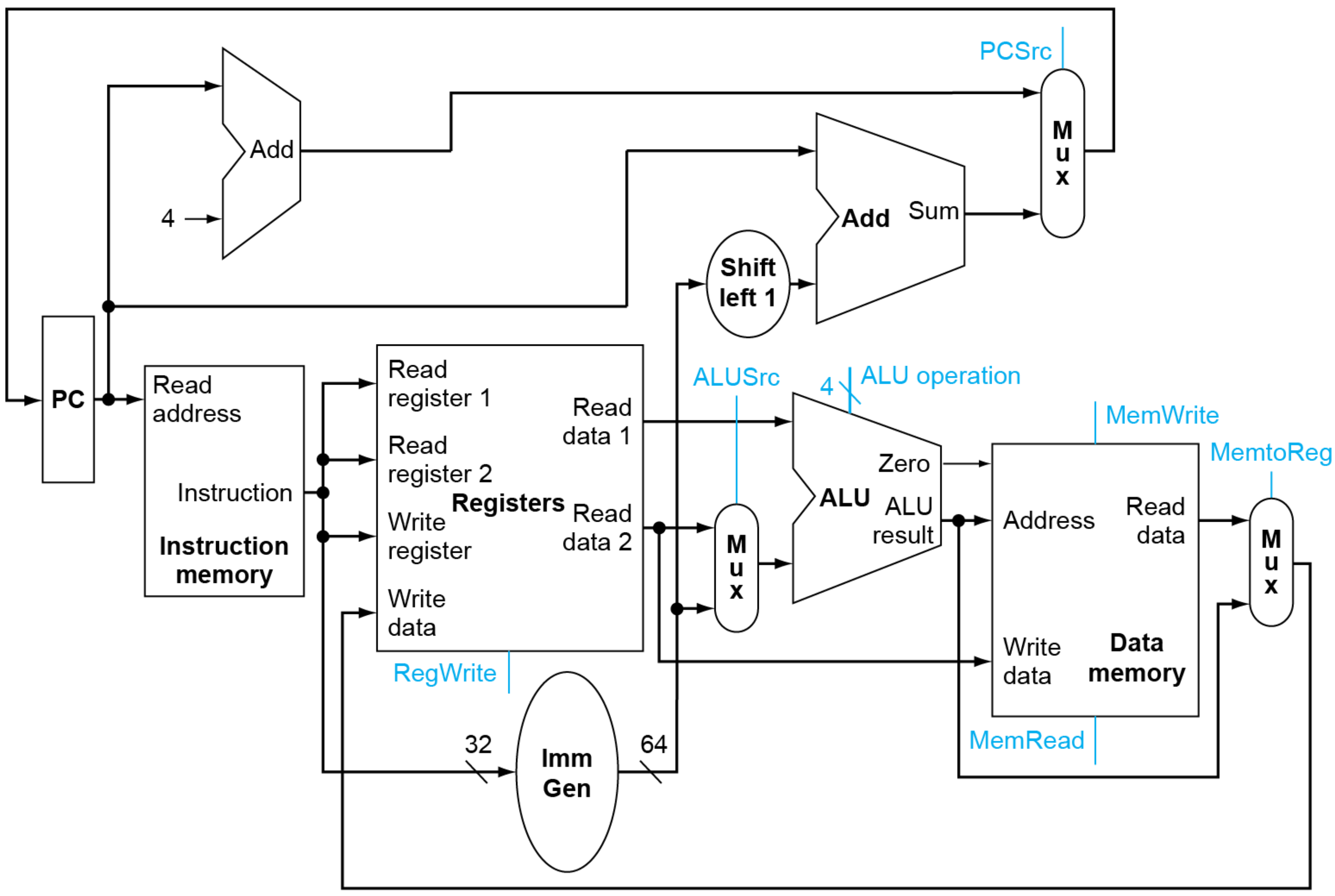
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



A Simple Implementation Scheme

ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	0000011	add	0010
sd	00	store register	0100011	add	0010
beq	01	branch on equal	1100011	subtract	0110
R-type	10	add	0110011	add	0010
		subtract		subtract	0110
		AND		AND	0000
		OR		OR	0001

The Main Control Unit

- Control signals derived from instruction

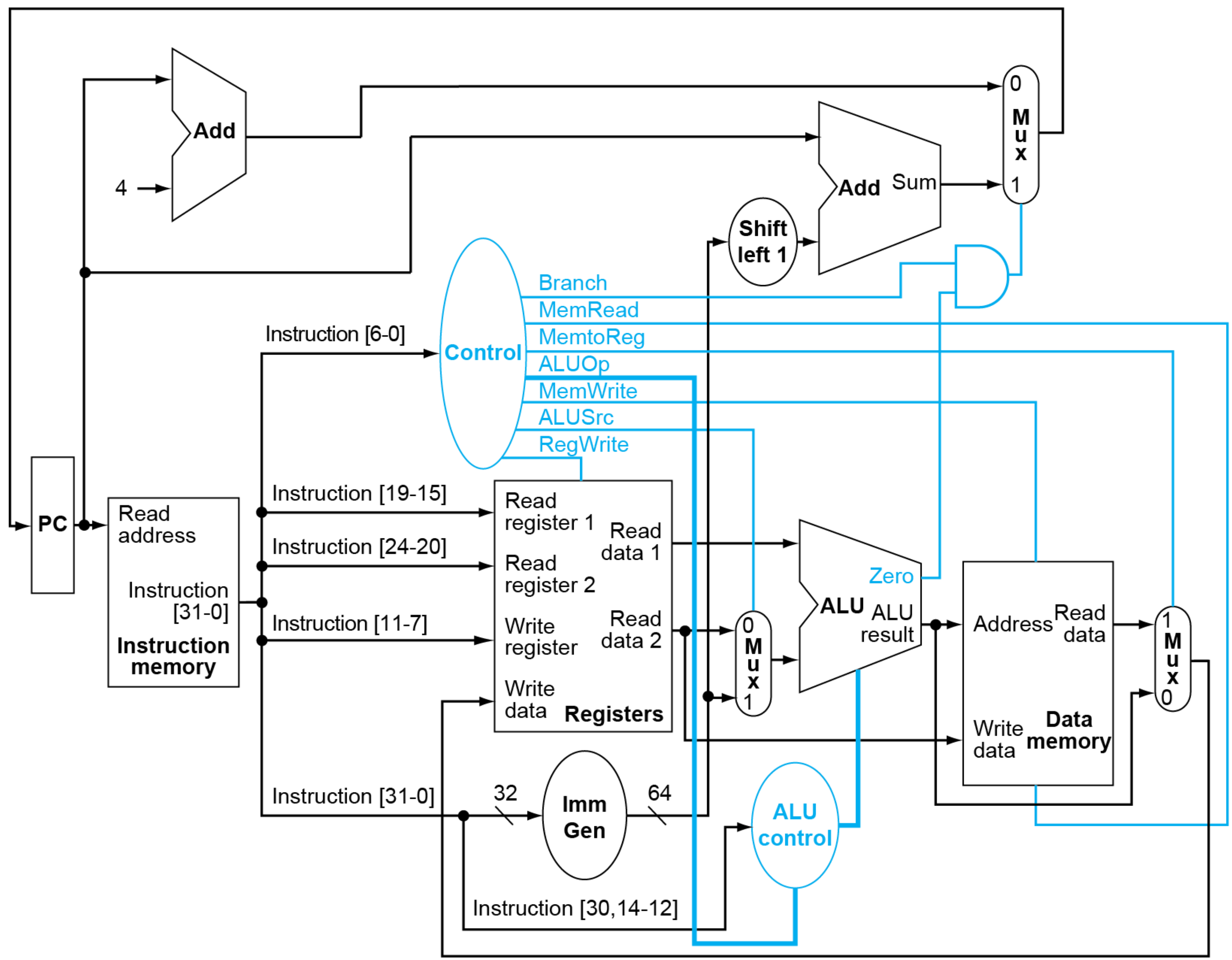
Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

ALUOp		Funct7 field							Funct3 field			
ALUOpl	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

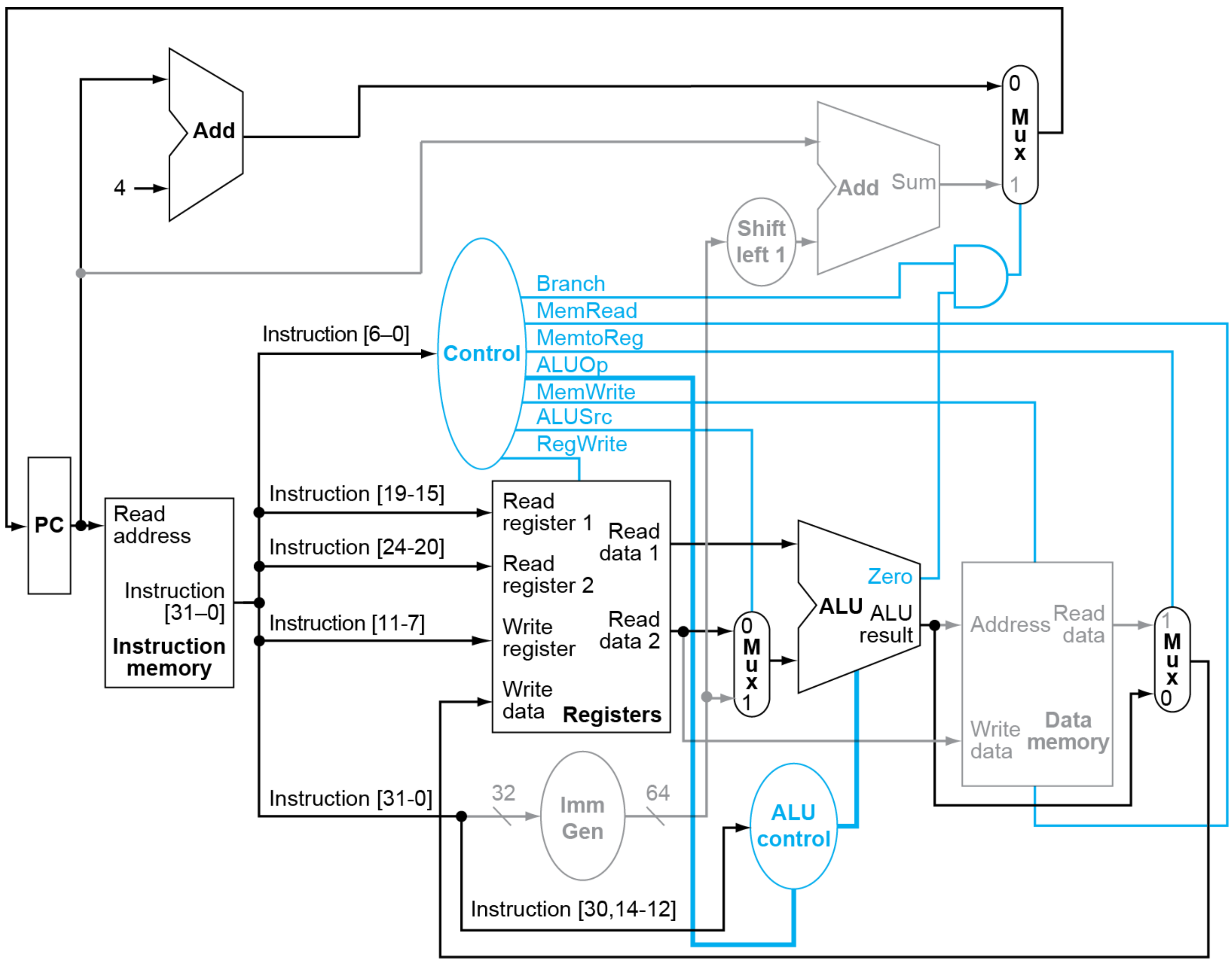
Control Unit (RV32I)

Base interger instrucitons: RV32I, register width = 32								
Category	Mnemonic	Instrunction	Format	Name	Description (in Verilog)	opcode	func3	func7
Loads	LB	LB rd, rs1, imm	I	Load Byte	R[rd] = {24'bM[](7), M[R[rs1]+imm](7:0)}	0000011	000	
	LH	LH rd, rs1, imm	I	Load Halfword	R[rd] = {16'bM[](15), M[R[rs1]+imm](15:0)}	0000011	001	
	LW	LW rd, rs1, imm	I	Load Word	R[rd] = M[R[rs1]+imm]	0000011	010	
	LBU	LBU rd, rs1, imm	I	Load Byte Unsigned	R[rd] = {24'b0, M[R[rs1]+imm](7:0)}	0000011	100	
	LHU	LHU rd, rs1, imm	I	Load Halfwaod Unsigned	R[rd] = {16'b0, M[R[rs1]+imm](15:0)}	0000011	101	
Stores	SB	SB rs1, rs2, imm	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	0100011	000	
	SH	SH rs1, rs2, imm	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	0100011	001	
	SW	SW rs1, rs2, imm	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	0100011	010	
Shifts	SLL	SLL rd, rs1, rs2	R	Shift Left	R[rd] = R[rs1] << R[rs2]	0110011	001	0000000
	SLLI	SLLI rd, rs1, shamt	I	Shift Left Immediate	R[rd] = R[rs1] << imm	0010011	001	0000000
	SRL	SRL rd, rs1, rs2	R	Shift Right	R[rd] = R[rs1] >> R[rs2]	0110011	101	0000000
	SRLI	SRLI rd, rs1, shamt	I	Shift Right Immediate	R[rd] = R[rs1] >> imm	0010011	101	0000000
	SRA	SRA rd, rs1, rs2	R	Shift Right Arithmetic	R[rd] = R[rs1] >> R[rs2] (arithmetic)	0110011	101	0100000
	SRAI	SRAI rd, rs1, shamt	I	Shift Right Arith Imm	R[rd] = R[rs1] >> imm (arithmetic)	0010011	101	0100000
Arithmetic	ADD	ADD rd, rs1, rs2	R	Add	R[rd] = R[rs1] + R[rs2]	0110011	000	0000000
	ADDI	ADDI rd, rs1, imm	I	Add Immediate	R[rd] = R[rs1] + imm	0010011	000	
	SUB	SUB rd, rs1, rs2	R	Subtract	R[rd] = R[rs1] - R[rs2]	0110011	000	0100000
	LUI	LUI rd, imm	U	Load Upper Immediate	R[rd] = {imm, 12'b0}	0110111		
	AUIPC	AUIPC rd, imm	U	Add Upper Imm to PC	R[rd] = PC + {imm, 12'b0}	0010111		
Logical	XOR	XOR rd, rs1, rs2	R	XOR	R[rd] = R[rs1] ^ R[rs2]	0110011	100	0000000
	XORI	XORI rd, rs1, imm	I	XOR Immediate	R[rd] = R[rs1] ^ imm	0010011	100	
	OR	OR rd, rs1, rs2	R	OR	R[rd] = R[rs1] | R[rs2]	0110011	110	0000000
	ORI	ORI rd, rs1, imm	I	OR Immediate	R[rd] = R[rs1] | imm	0010011	110	
	AND	AND rd, rs1, rs2	R	AND	R[rd] = R[rs1] & R[rs2]	0110011	111	0000000
	ANDI	ANDI rd, rs1, imm	I	AND Immediate	R[rd] = R[rs1] & imm	0010011	111	
Compare	SLT	SLT rd, rs1, rs2	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1: 0	0110011	010	0000000
	SLTI	SLTI rd, rs1, imm	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1: 0	0010011	010	
	SLTU	SLTU rd, rs1, rs2	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1: 0 (unsigned)	0110011	011	0000000
	SLTIU	SLTIU rd, rs1, imm	I	Set Less Than Imm Unsigned	R[rd] = (R[rs1] < imm) ? 1: 0 (unsigned)	0010011	011	
Branches	BEQ	BEQ rs1, rs2, imm	SB	Branch if Equal	if (R[rs1] == R[rs2]) PC = PC + {imm, 1'b0}	1100011	000	
	BNE	BNE rs1, rs2, imm	SB	Branch if Not Equal	if (R[rs1] != R[rs2]) PC = PC + {imm, 1'b0}	1100011	001	
	BLT	BLT rs1, rs2, imm	SB	Branch if Less Than	if (R[rs1] < R[rs2]) PC = PC + {imm, 1'b0}	1100011	100	
	BGE	BGE rs1, rs2, imm	SB	Branch if Greater or Equal	if (R[rs1] >= R[rs2]) PC = PC + {imm, 1'b0}	1100011	101	
	BLTU	BLTU rs1, rs2, imm	SB	Branch if Less Than Imm	if (R[rs1] < R[rs2]) PC = PC + {imm, 1'b0} (unsigned)	1100011	110	
	BGEU	BGEU rs1, rs2, imm	SB	Branch if GE Imm	if (R[rs1] >= R[rs2]) PC = PC + {imm, 1'b0} (unsigned)	1100011	111	
Jump	JAL	JAL rd, imm	UJ	Jump & Link	R[rd] = PC + 4; PC = PC + {imm, 1'b0}	1101111		
	JALR	JALR rd, rs1, imm	UJ	Jump & Link Register	R[rd] = PC + 4; PC = R[rs1] + imm	1100111	000	

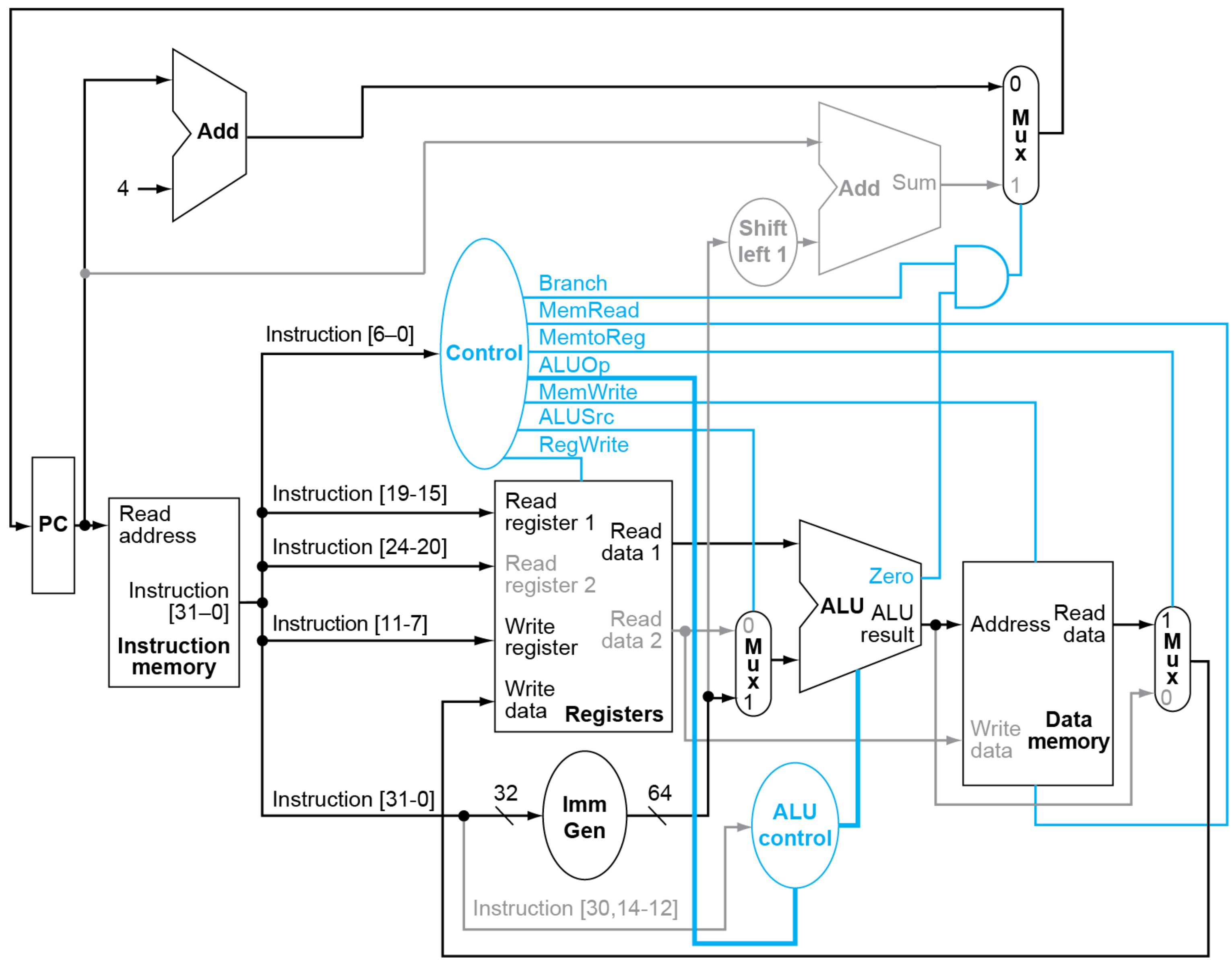
Datapath with Control



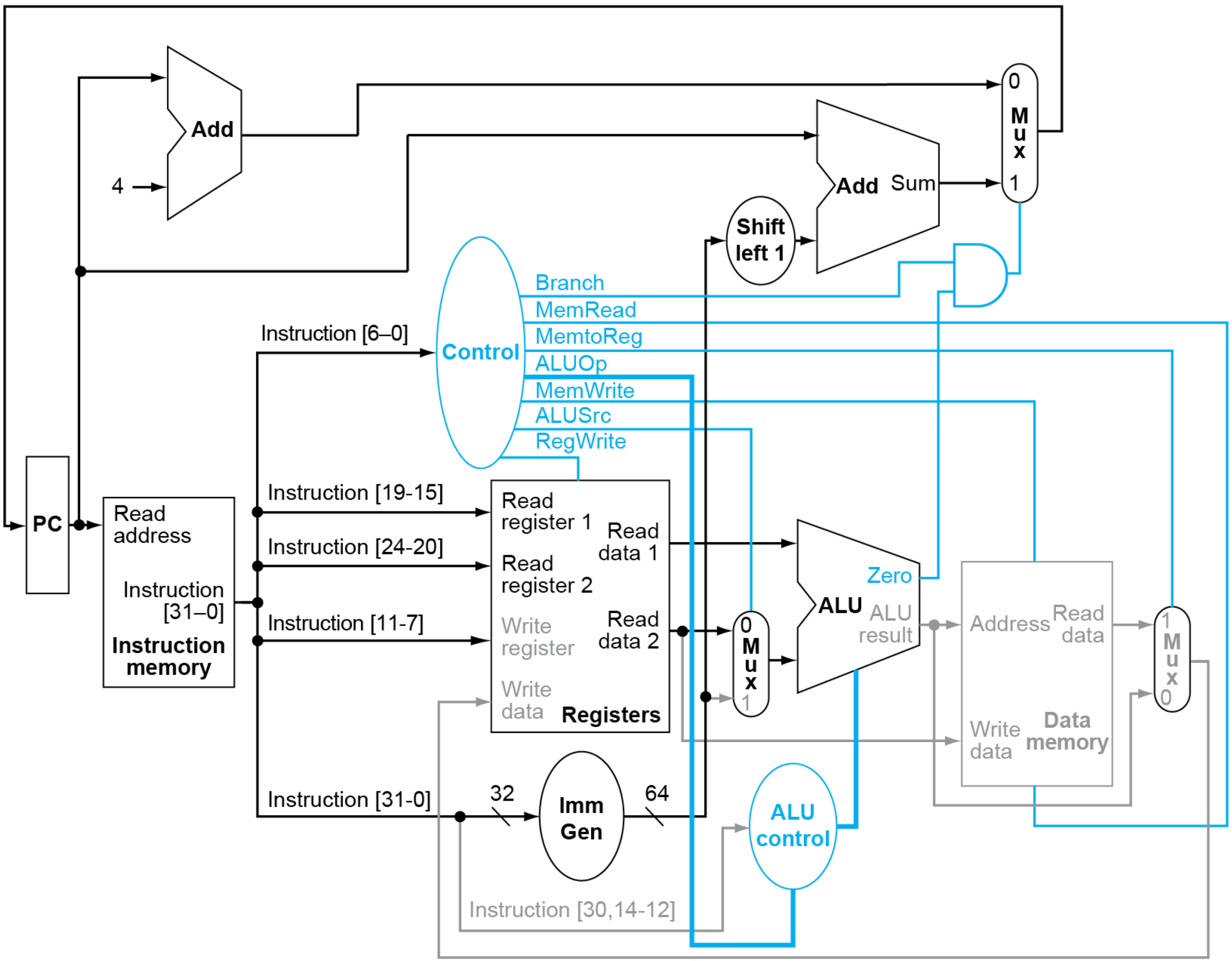
R-Type Instruction



Load Instruction



BEQ Instrution



Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Thank you

Questions?

Back-up Slides