

Topic 5

Single Cycle Processor

Introduction

- Starting from this topic, we will examine two RISC-V implementations
 - A simplified single-cycle implementation
 - A more realistic pipelined implementation
- Supporting a simple subset, showing most aspects of RISC-V design
 - For memory reference: lw, sw
 - For arithmetic/logical: add, sub, and, or
 - For conditional branch: beq
- You are expected to be able to change the hardware to support more instructions

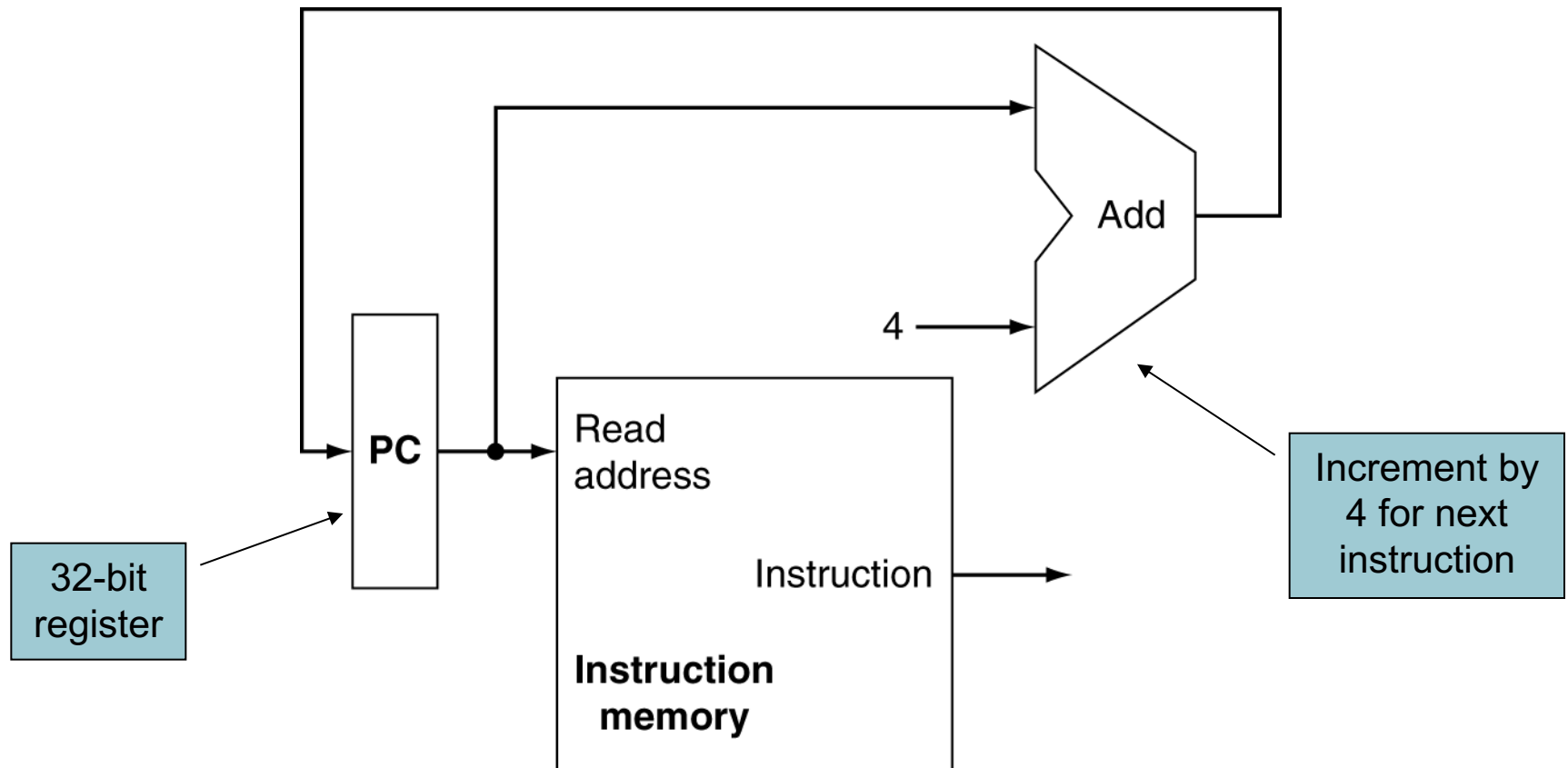
Instruction Execution

- ① PC \rightarrow program memory, fetch instruction
- ② In a 32-bit instruction, use register addresses to access Register File, read registers
- ③ Depending on instructions
 - Use ALU/Adder to calculate
 - Arithmetic/logic operations
 - Memory address for load/store
 - Branch target address
- ④ Access data memory for load or store, or access register file to save calculation results
- ⑤ Update PC \leftarrow target address or PC + 4

Building a Datapath

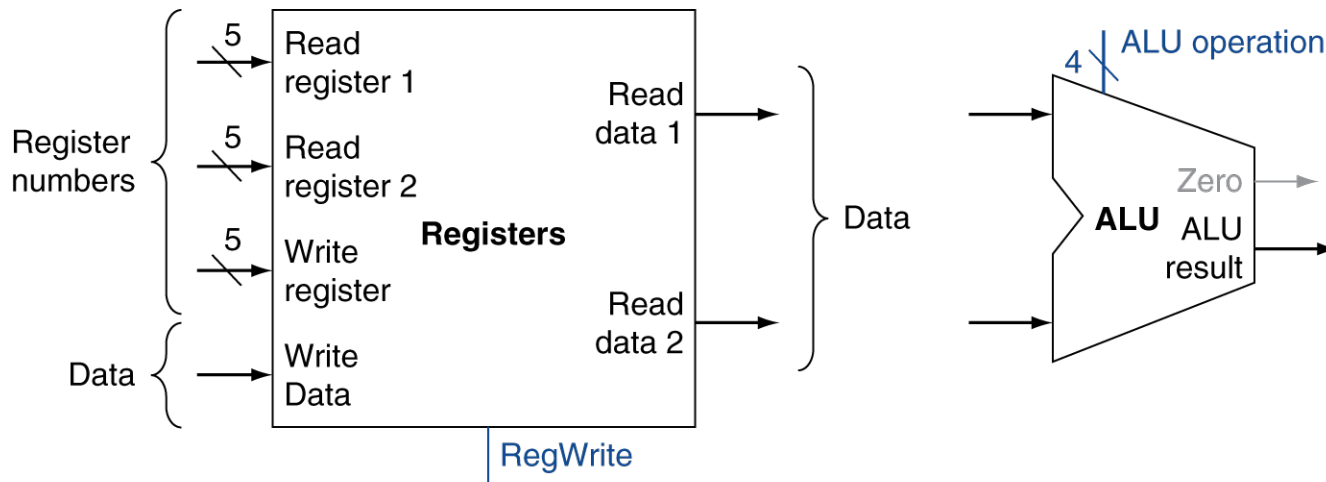
- Datapath
 - Elements that process, store, or route data in the CPU
 - E.g. registers, ALUs, mux's, memories, ...
- We will build a RISC-V datapath incrementally
- We will add a control unit to provide control signals to the datapath

Instruction Fetch



R-Format Instructions

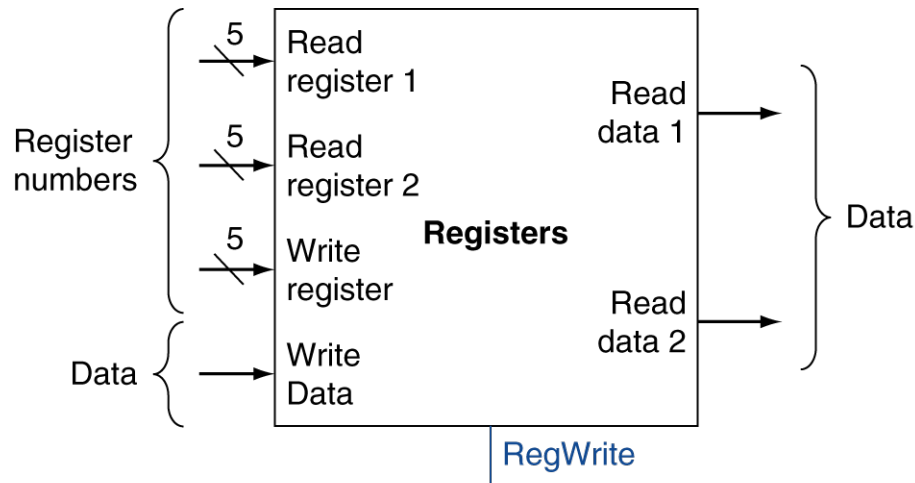
- Operations: add rd, rs1, rs2
 - Read two register operands
 - Perform arithmetic/logical operation
 - Write register result
- We need:



a. Registers

b. ALU

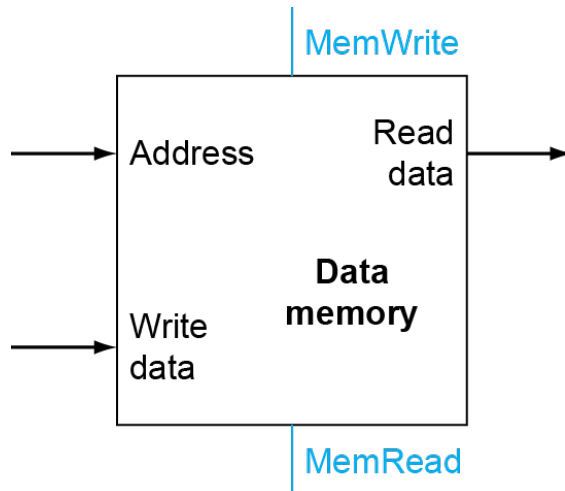
Register Files



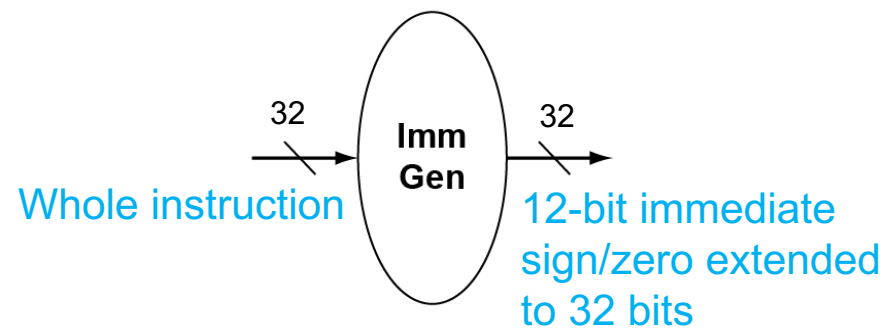
- Data is read out whenever register numbers are provided and $\text{RegWrite} = 0$
- Writes
 - Edge-triggered
 - All the write inputs (Write Data, Write Register, RegWrite) must be valid before the clock edge – setup time
- Can read and write the same register within a clock cycle: write first, followed by read.

Load/Store Instructions

- Operations: `lw rd, imm12(rs1)`
`sw rs2, imm12(rs1)`
 - Read register operands
 - Calculate address using 12-bit immediate
 - Load: Read memory and update register
 - Store: Write register value to memory
- So, we need:



a. Data memory unit



b. Immediate generation unit

Beq Instructions

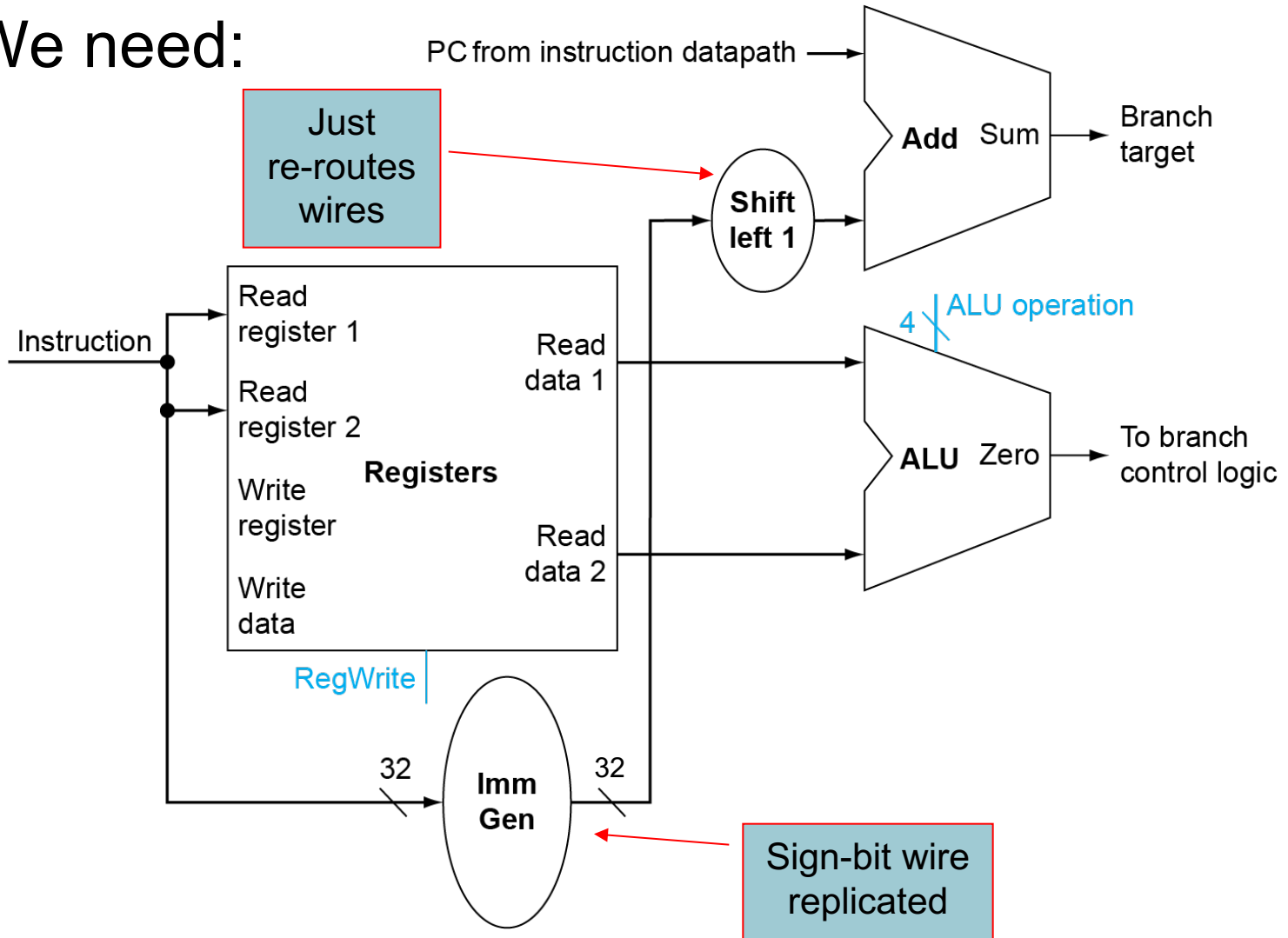
- Operations: beq rs1, rs2, target
 - Read register operands
 - Compare operands
 - $rs1 == rs2? \Rightarrow rs1 - rs2 == 0?$
 - Use ALU, subtract then check the Zero output of the ALU
 - Calculate target address

Target PC = Current PC + immediate \times 2

 - Sign-extend the immediate
 - Shift left 1 bit (multiply by 2)
 - Add to PC value

Branch Instructions

- We need:



Performance Considerations

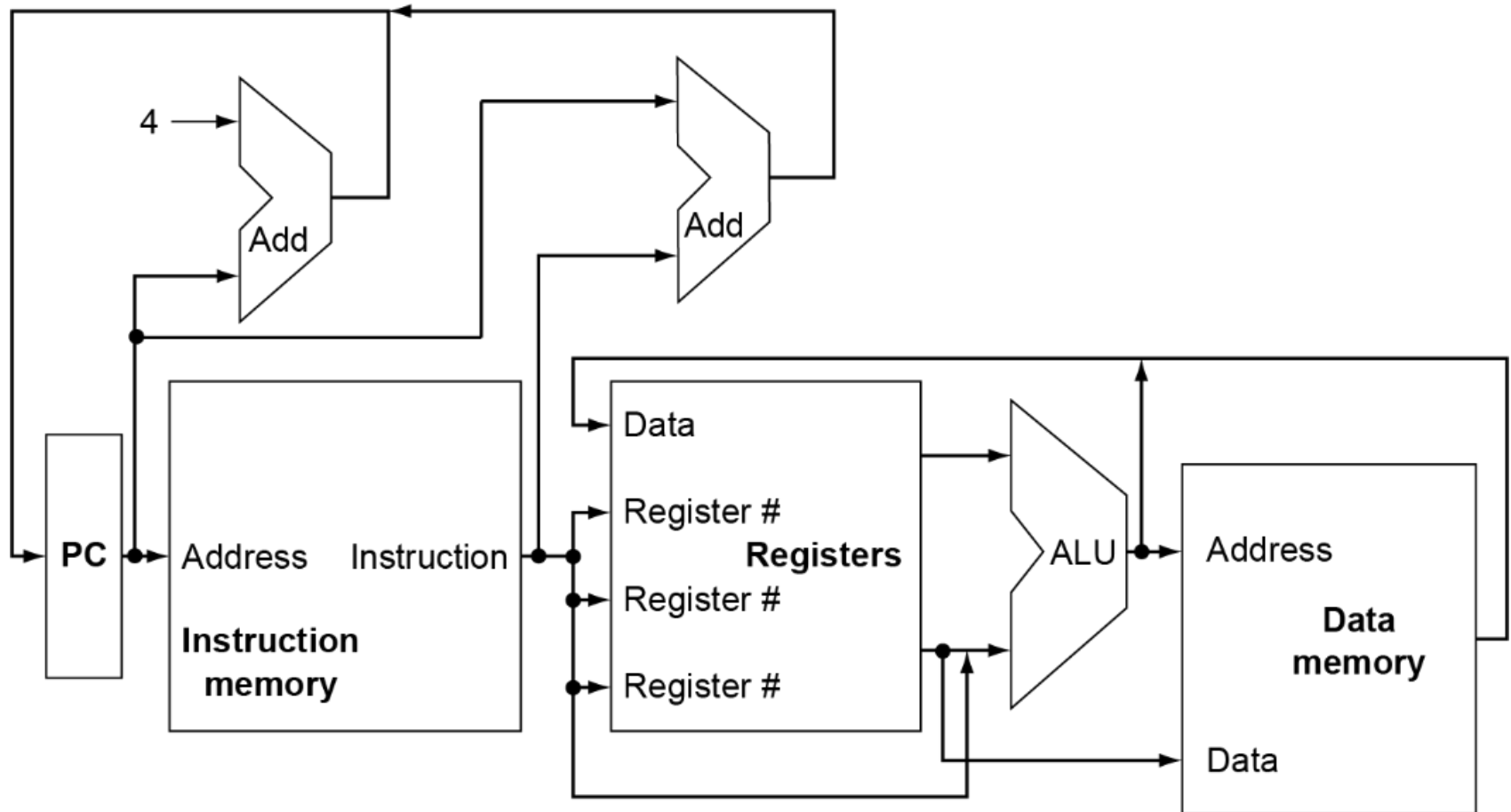
Type	Field					
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
I-type	immediate[11:0]		rs1	funct3	rd	opcode
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
B-type	immed[11,9:4]	rs2	rs1	funct3	immed[3:0,10]	opcode
U-type	immediate[19:0]				rd	opcode
J-type	immediate[19,9:0,10,18:11]				rd	opcode

- In B-type (SB-type) and J-type (UJ-type), immediate bits are swirled around
 - It saves hardware (muxes) in “**Imm Gen**” component which picks different fields in an instruction to assemble the immediate number according to different type of the instruction

Composing the Elements

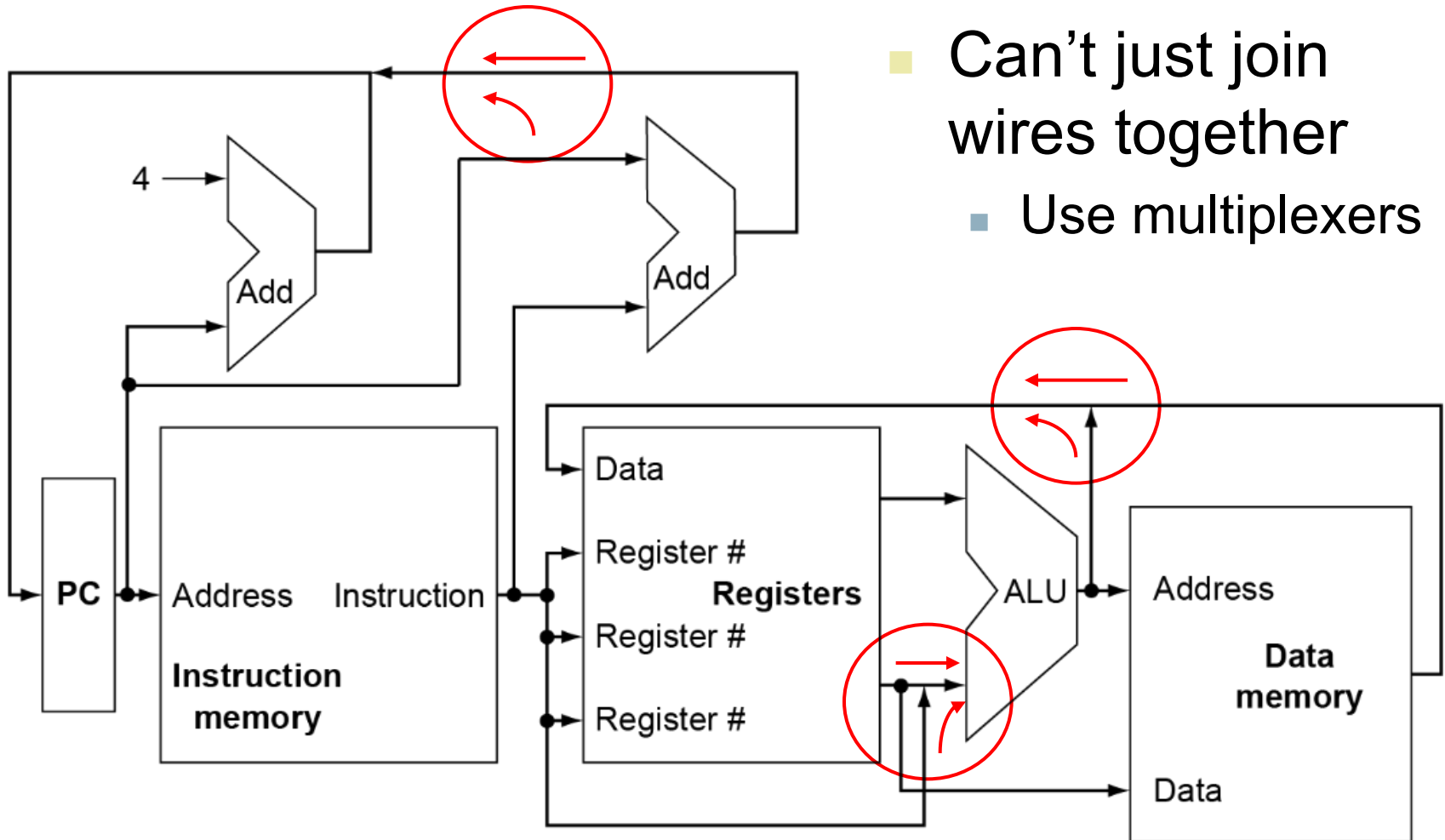
- First version of datapath executes an instruction in one single clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction memory and data memory
- Use multiplexers where alternate data sources are used for different instructions

Combine The Operations

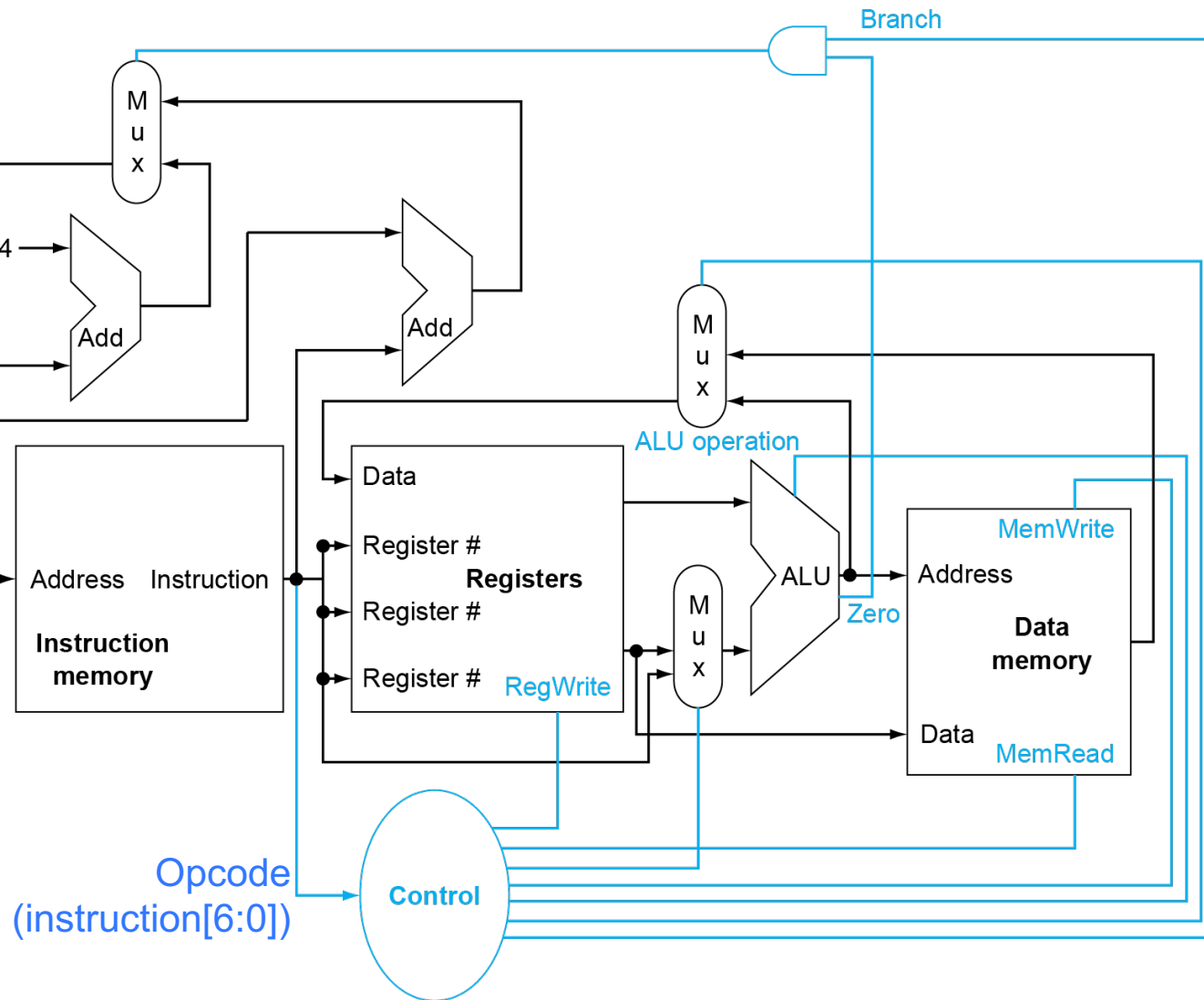


Add Multiplexers

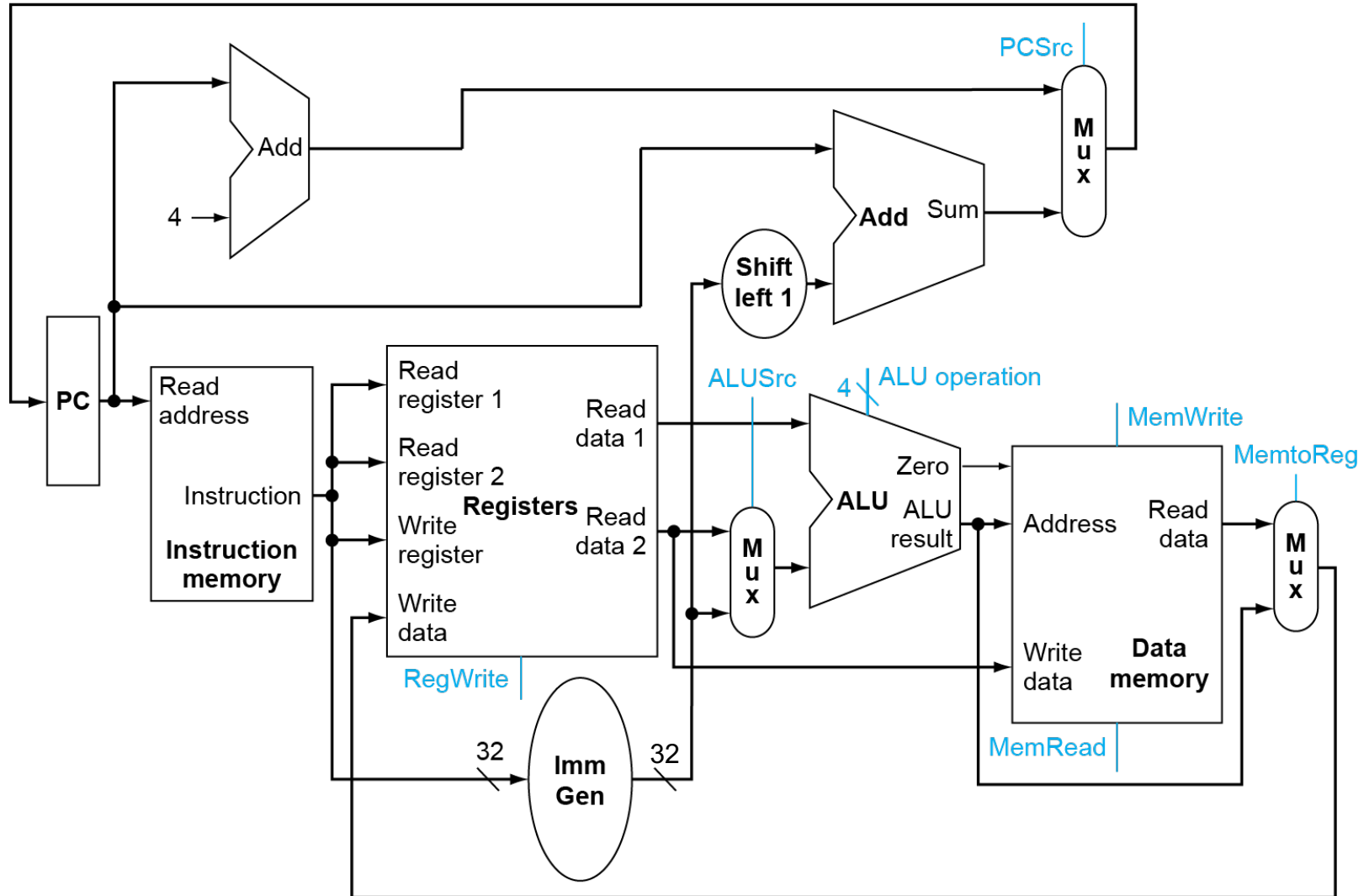
- Can't just join wires together
 - Use multiplexers



Add Control Signals

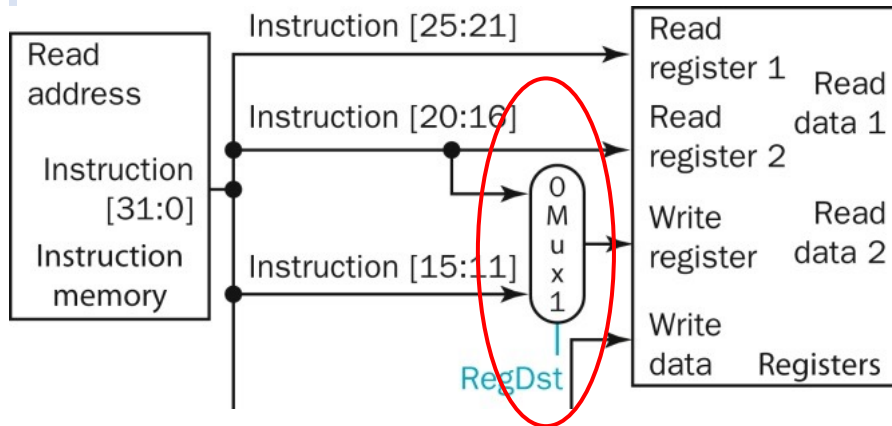


Putting Everything Together

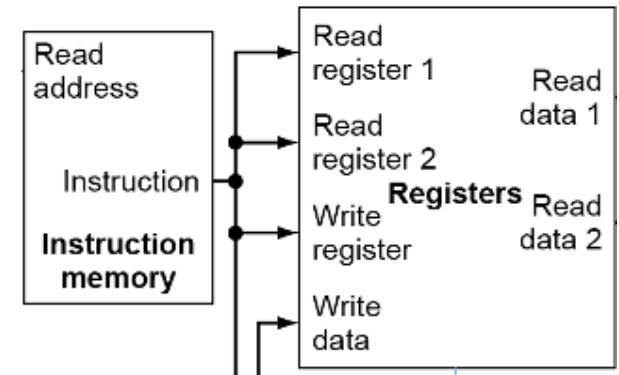


Performance Consideration

- RISC-V has more types (6) than MIPS (4)
 - But it saves hardware on the critical path



V.S.



MIPS Architecture: simpler instruction format, but more complicated hardware, longer delay

RISC-V Architecture: more complicated instruction format, but simpler hardware, thus better performance

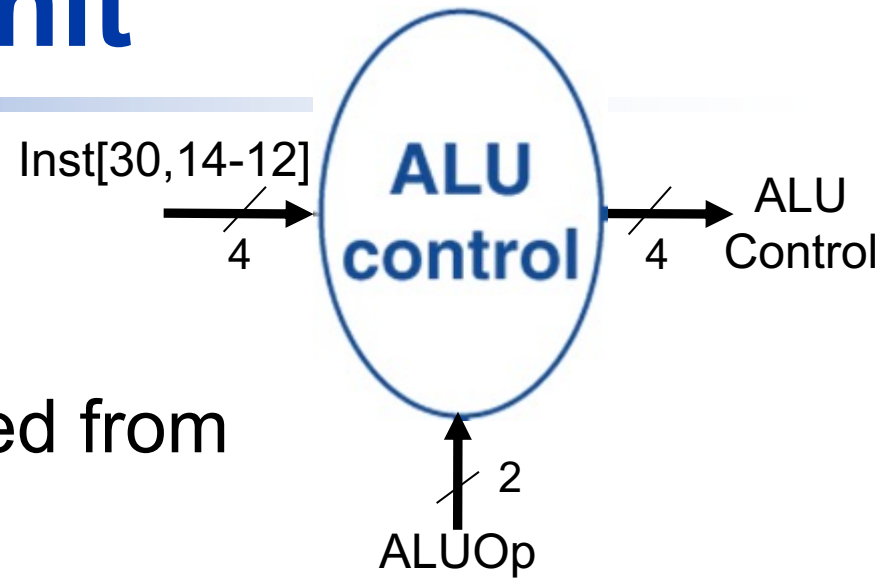
Add ALU Control

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

ALU control Signal	Function
0000	AND
0001	OR
0010	add
0110	subtract

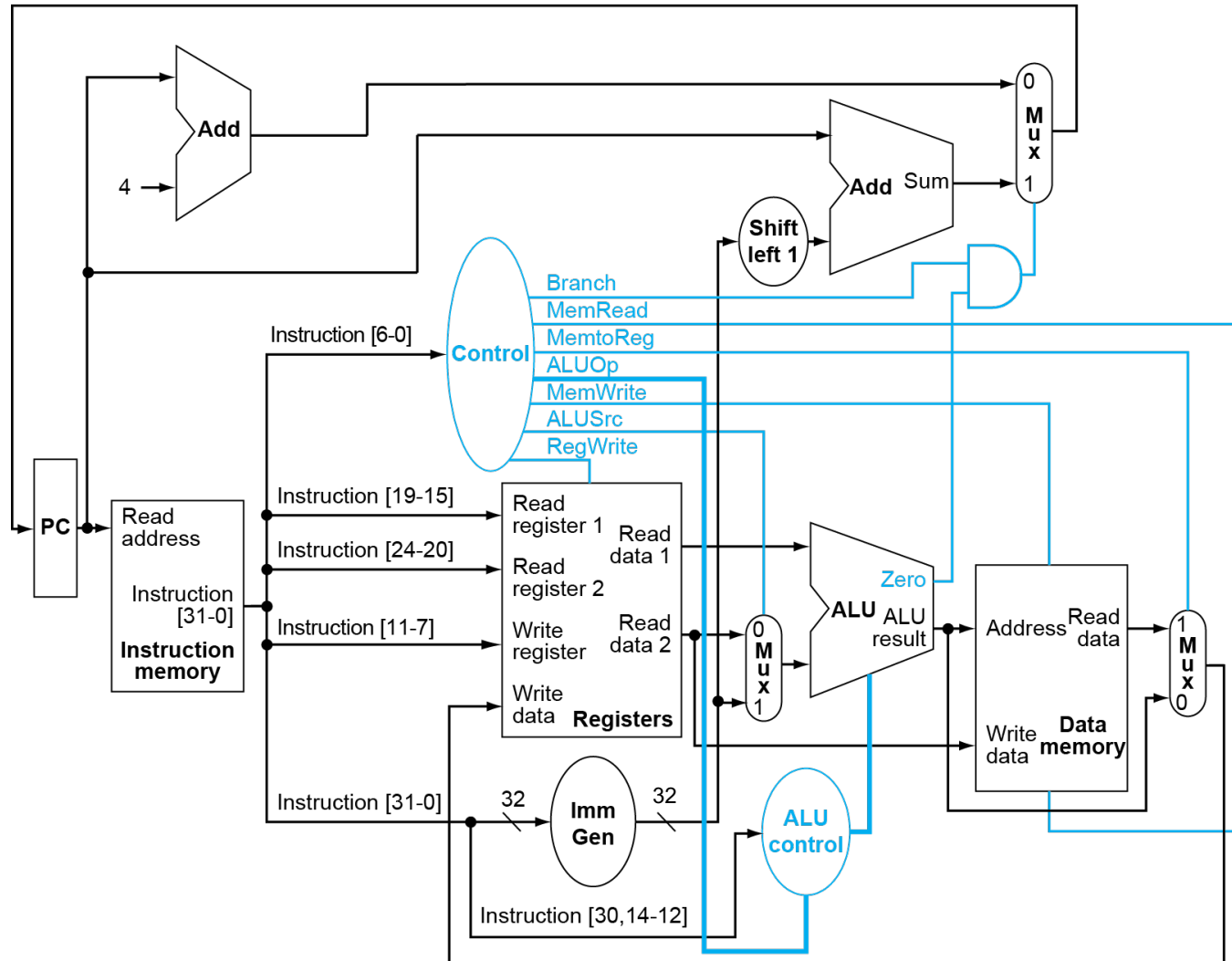
ALU Control Unit

- ALU Control is a combinational logic
 - 2-bit ALUOp is derived from opcode

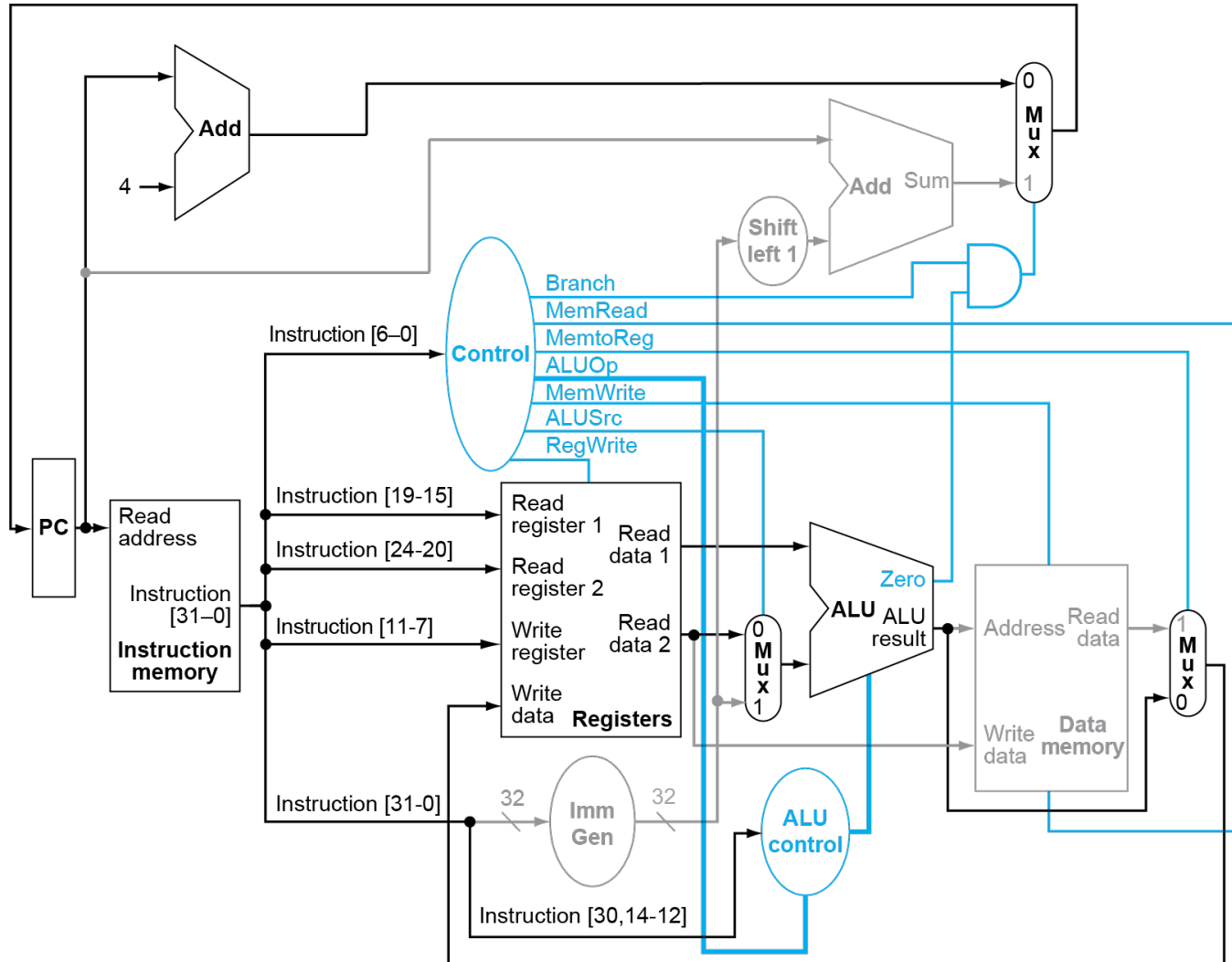


Instr.	Operation	ALU function	Opcode field	ALUOp (input)	i[30] (input)	funct3 (input)	ALU control (output)
lw	load register	add	XXXXXXXX	00	X	010	0010
sw	store register	add	XXXXXXXX	00	X	010	0010
beq	branch on equal	subtract	XXXXXXXX	01	X	000	0110
R-type	add	add	100000	10	0	000	0010
	subtract	subtract	100010		1	000	0110
	AND	AND	100100		0	111	0000
	OR	OR	100101		0	110	0001

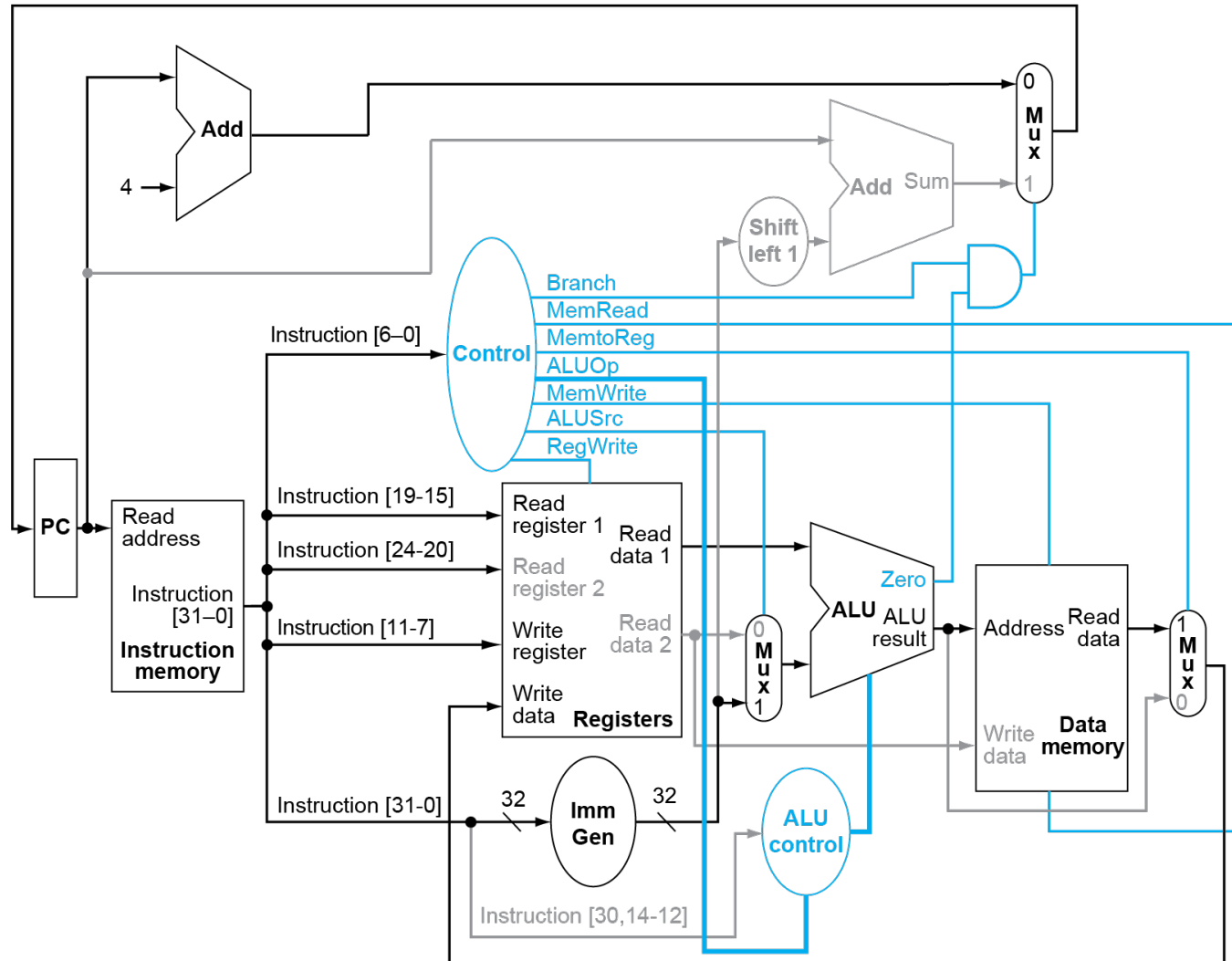
Datapath With Control Unit



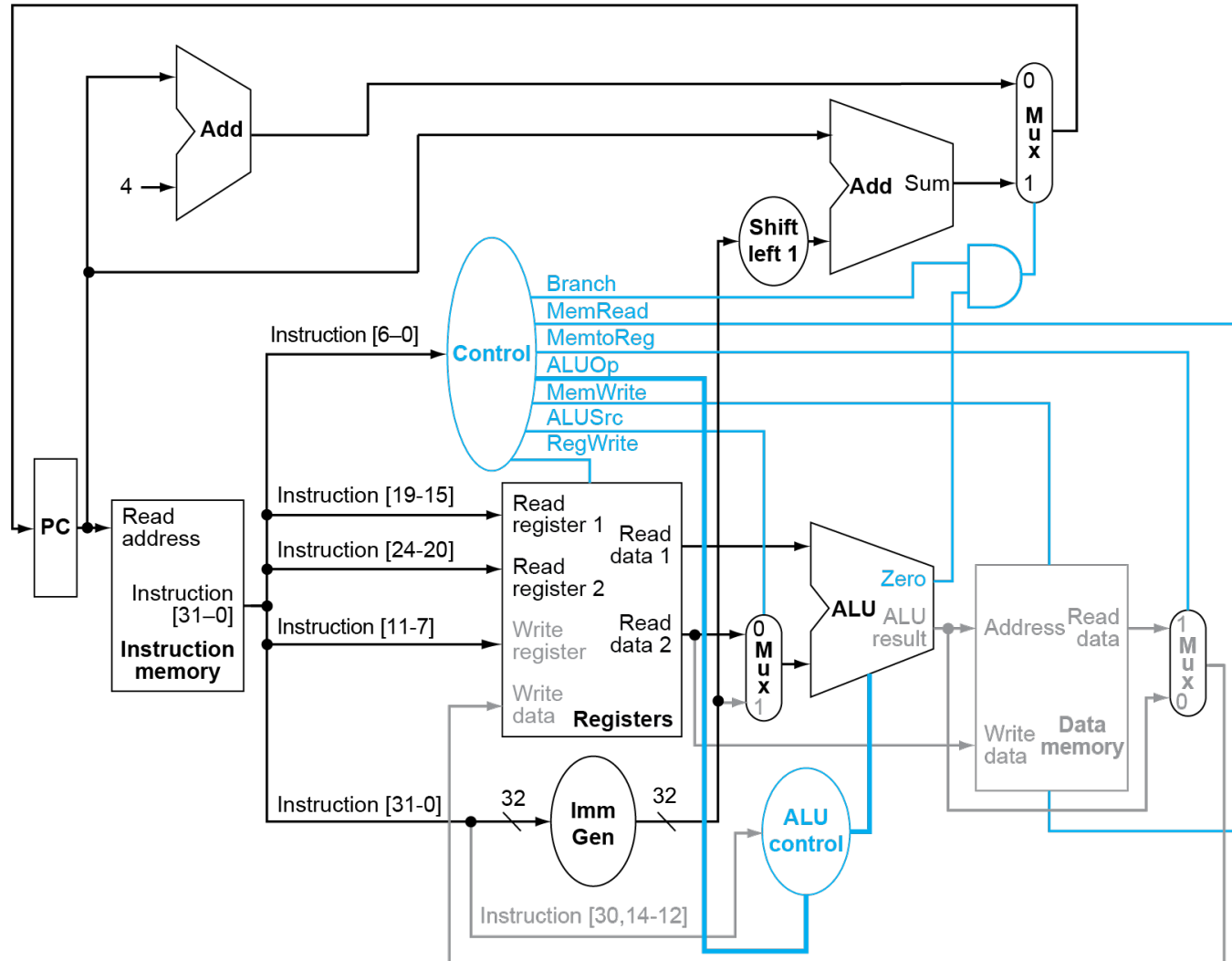
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction



Class Exercise: Control Signals

- Control signals derived from instruction: fill out the following table

Inst.	ALU Src	ALU Op	Mem Write	Mem Read	Branch	Memto Reg	Reg Write
add							
lw							
sw							
beq							
addi							

Performance Related Factors

- Algorithm
 - Determines number of operations executed
- Programming language, compiler, architecture
 - Determine number of machine instructions (lines of source code) executed per operation
- Processor and memory system
 - Determine how fast instructions are executed
- I/O system (including OS)
 - Determines how fast I/O operations are executed

How to Measure Computer Performance?

- Execution (response) time
 - How long it takes to do a task
 - Measure for PCs and embedded computers
- Throughput
 - Total work done per unit time
 - Servers

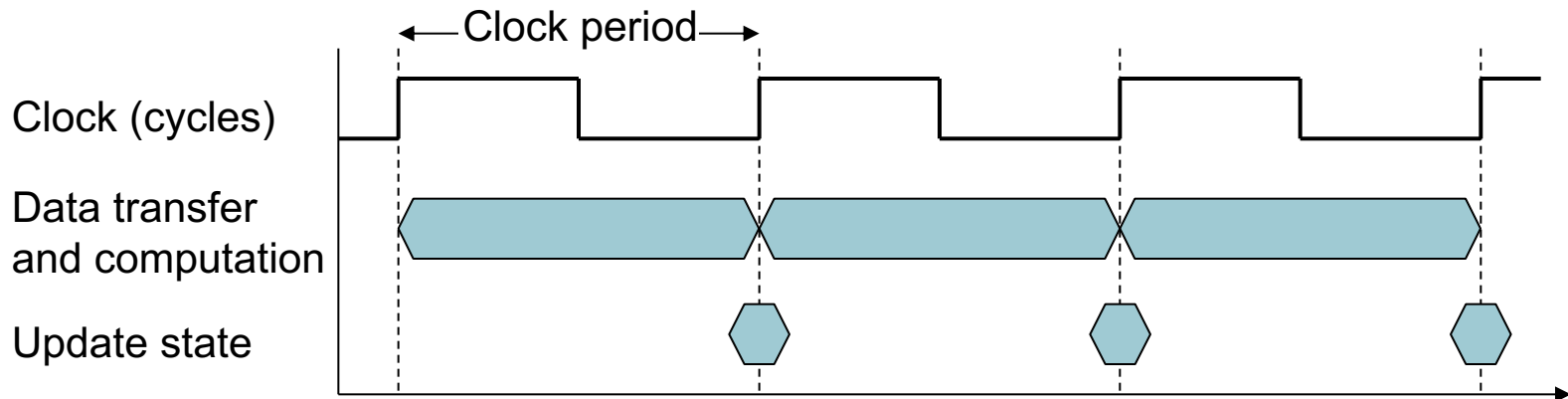
We'll focus on **execution time** for now

Execution Time

- Elapsed time – for System Performance
 - Total execution time to complete a task, including
 - Processing, I/O, OS overhead, idle time, everything
 - But doesn't completely reflect computer's performance if it focuses on better throughput
- CPU time – for CPU Performance
 - CPU execution time processing a task
 - Exclude I/O time, time spent for other tasks
 - Comprises **user CPU time** and **system CPU time**
 - Hard to differentiate
 - Different programs run with different CPU performance and system performance

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
 - e.g., $250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$
- Clock frequency (rate): cycles per second
 - e.g., $4.0\text{GHz} = 4000\text{MHz} = 4.0 \times 10^9\text{Hz}$

CPU Time

$$\begin{aligned}\text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}\end{aligned}$$

- Performance improved by
 - Reducing number of clock cycles
 - Increasing clock rate
 - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
 - Aim for 6s CPU time
 - Can do faster clock, but causes $1.2 \times$ clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

Instruction Count and CPI

Clock Cycles = Instruction Count \times Cycles per Instruction

CPU Time = Instruction Count \times CPI \times Clock Cycle Time

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count (IC) for a program
 - Determined by program, ISA and compiler
- Average cycles per instruction (CPI)
 - Determined by CPU hardware
 - If different instructions have different CPI
 - Average CPI affected by instruction mix

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= 1 \times 2.0 \times 250\text{ps} = 1 \times 500\text{ps} \leftarrow \text{A is faster...}\end{aligned}$$

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= 1 \times 1.2 \times 500\text{ps} = 1 \times 600\text{ps}\end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{1 \times 600\text{ps}}{1 \times 500\text{ps}} = 1.2 \leftarrow \text{...by this much}$$

Performance Summary

The BIG Picture

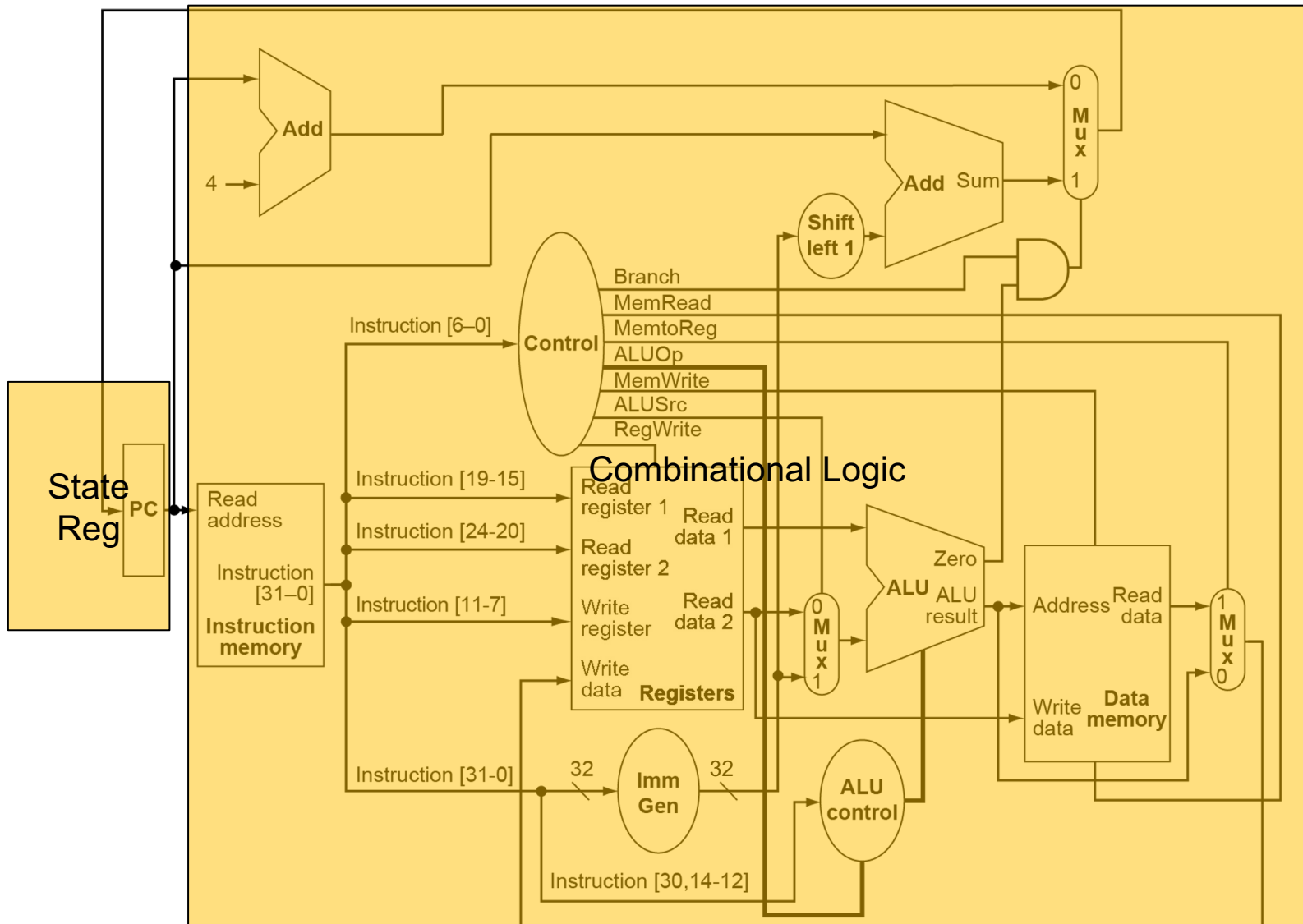
$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on

- Algorithm: affects IC, possibly CPI
- Programming language: affects IC, CPI
- Compiler: affects IC, CPI
- Instruction set architecture: affects IC, CPI, T_c

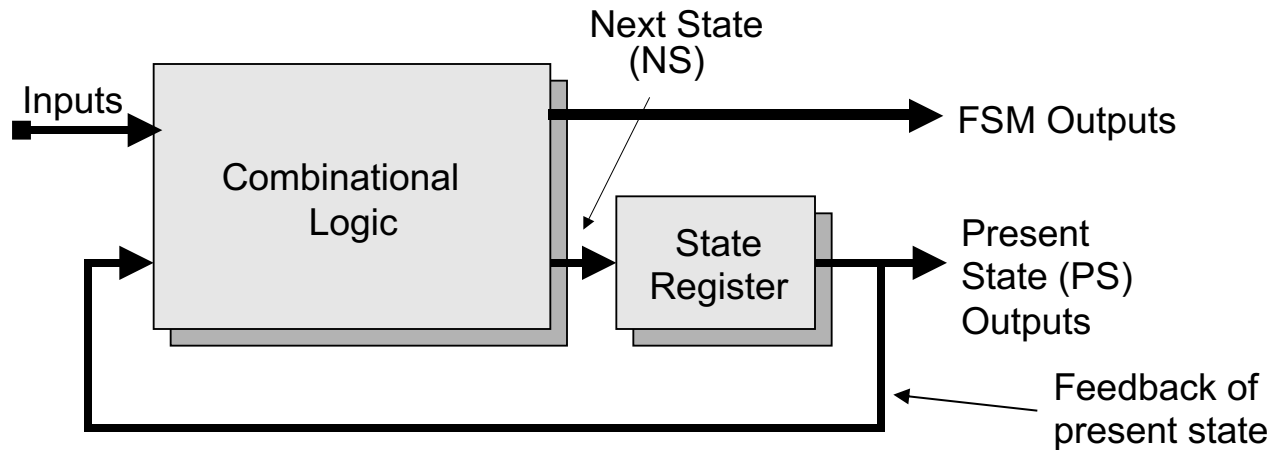
Clock Period

Single Cycle Implementation



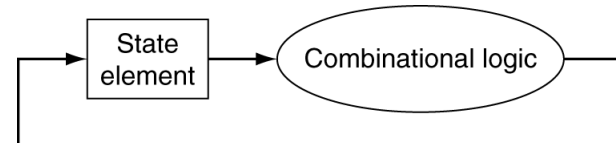
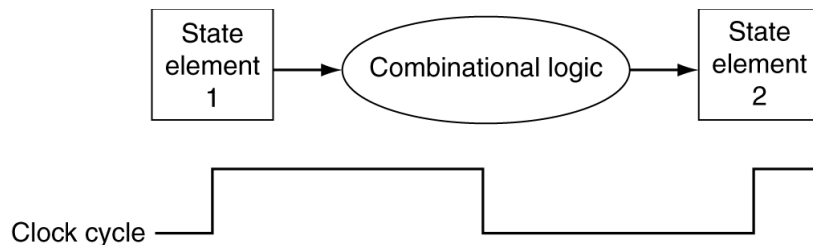
State Machine

- Finite State Machine



Clocking Methodology

- Combinational logic does the computation during clock cycles
 - Between clock edges
 - Input (present state) from state elements, output (next state) to state element
 - Among all kinds of computations, longest delay determines clock period



Performance Consideration

- Critical path of the combination logic determines the clock cycle time
 - Critical path: the path of the instruction that takes the longest time to finish
- Clock cycle time is fixed once determined
 - For all the instructions
 - Waste time on the shorter instructions
- Each instruction takes one clock cycle – single cycle implementation
- How to improve the performance (faster speed)?

$$\text{CPU time} = \text{IC} * \text{CPI} * T_C$$

Performance Consideration

- Assume times for major components are
 - 100ps for register read or write
 - 200ps for accessing memory
 - 200ps for ALU operations

Instruction	Instr fetch	Register read	ALU op	Data Memory	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Performance Consideration

- Assume 100 instructions are executed
 - 15% are loads
 - 15% are stores
 - 40% are R format instructions
 - 30% are branches
- What's the clock cycle time for single-cycle processor?
- Execution time using single-cycle processor?