Chapter 4

The Processor

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]
[Also adapted from *lecture slide* by Mary Jane Irwin, www.cse.psu.edu/~mji]

Introduction

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version □ slow
  - A more realistic pipelined version □ Optimized
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

Instruction Execution

- For every instruction, the first two steps are identical
  - PC → instruction memory, fetch instruction
  - Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic and logical operations execution
    - Memory address calculation for load/store
    - Branch for comparison
  - Access data memory for load/store
  - PC ← target address (for branch) or PC + 4

Fetch Instruction

Decode Instruction

add $8, $17, $18

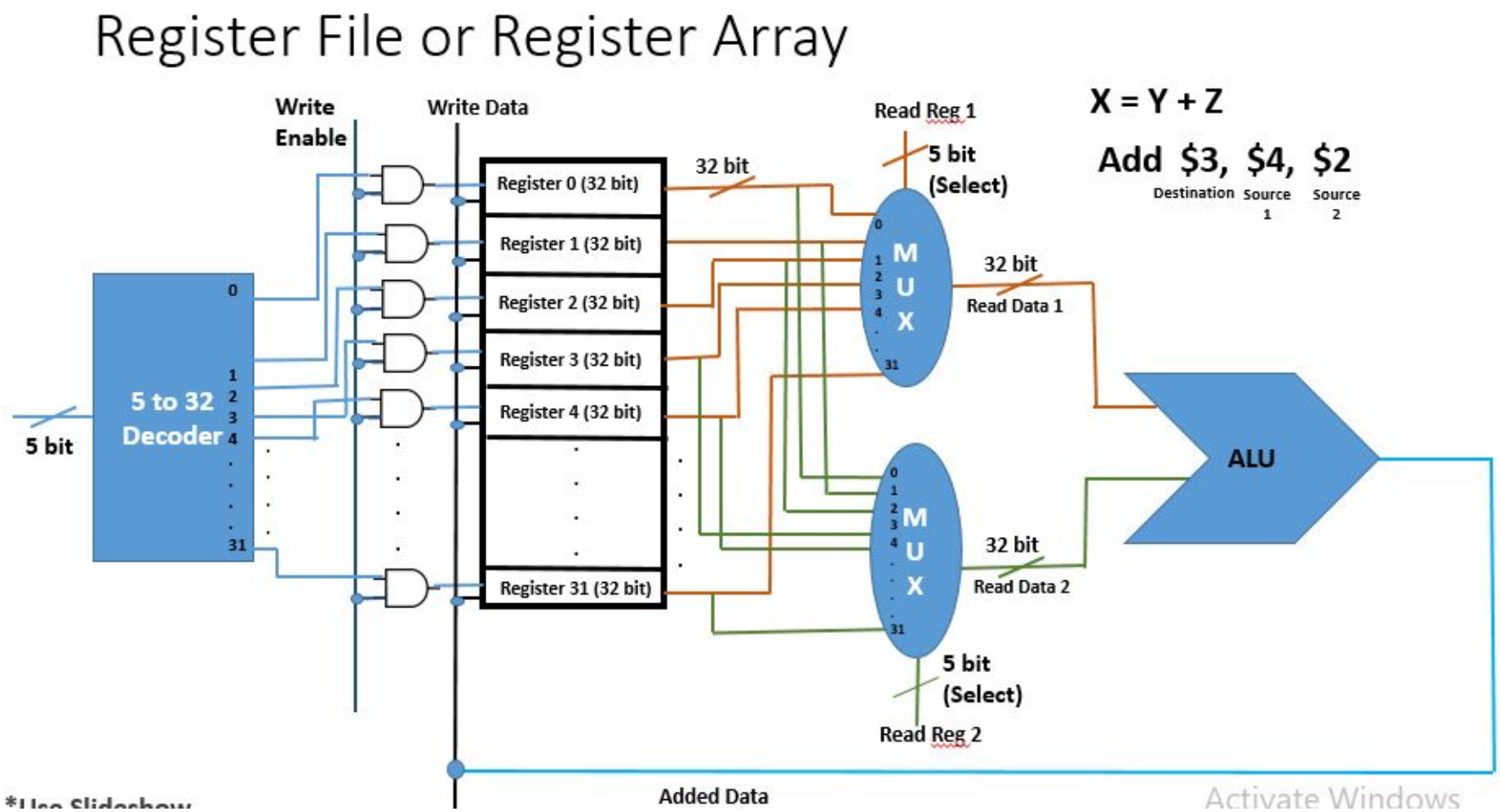
000000	10001	10010	01000	00000	100000
opcode	rs	rt	rd	shamt	funct

rd (5 bit) goes to decoder selects write register

rs (5 bit) goes to MuX reads register

rd (5 bit) goes to MuX reads register

Execute Instruction



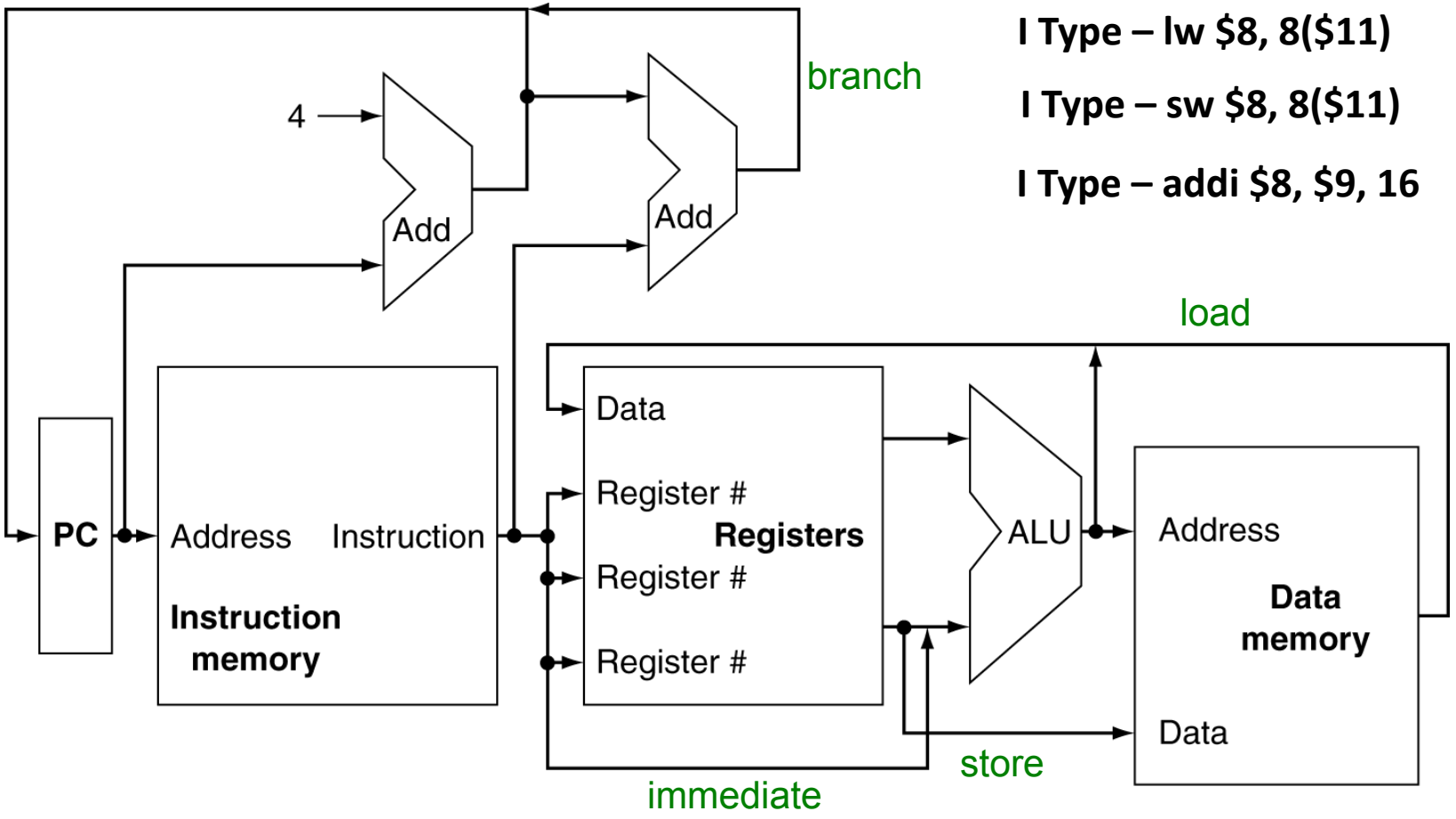
CPU Overview

R Type – add $8, $9, $10

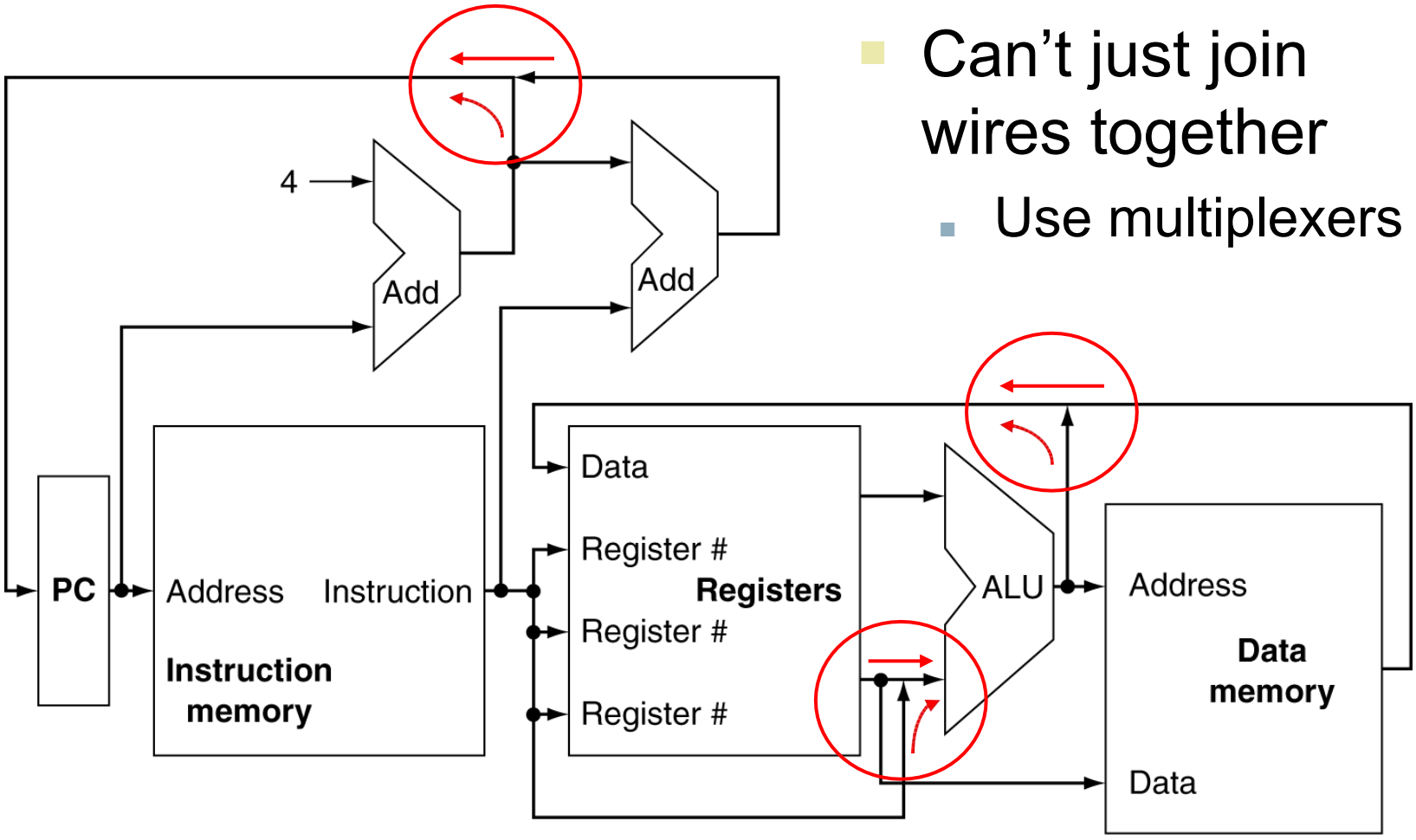
I Type – lw $8, 8($11)

I Type – sw $8, 8($11)

I Type – addi $8, $9, 16

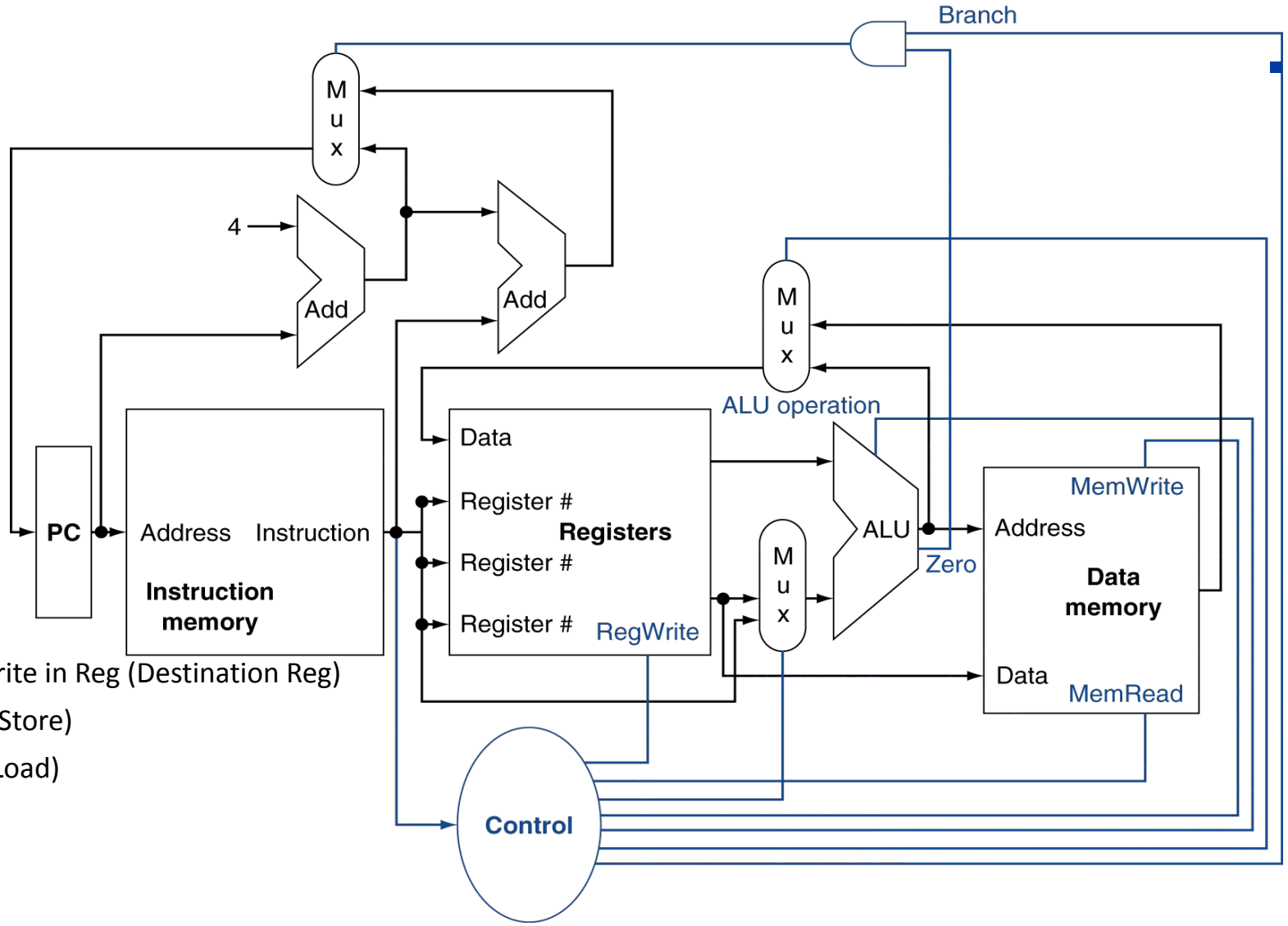


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control



RegWrite – 1 Write in Reg (Destination Reg)

MemWrite – 1 (Store)

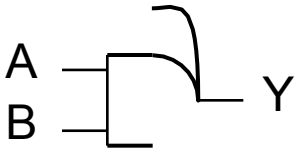
MemRead – 1 (Load)

Logic Design Basics

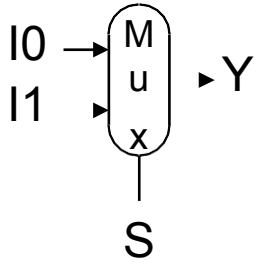
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

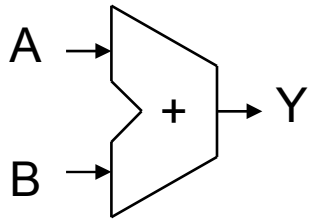
- AND-gate
  - Y = A & B



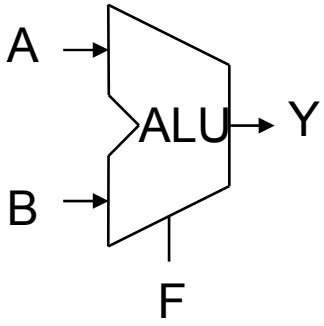
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

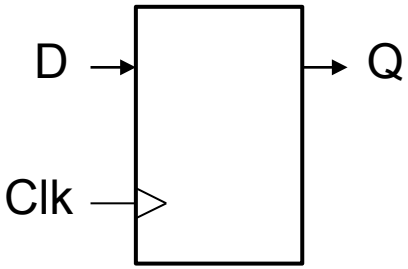


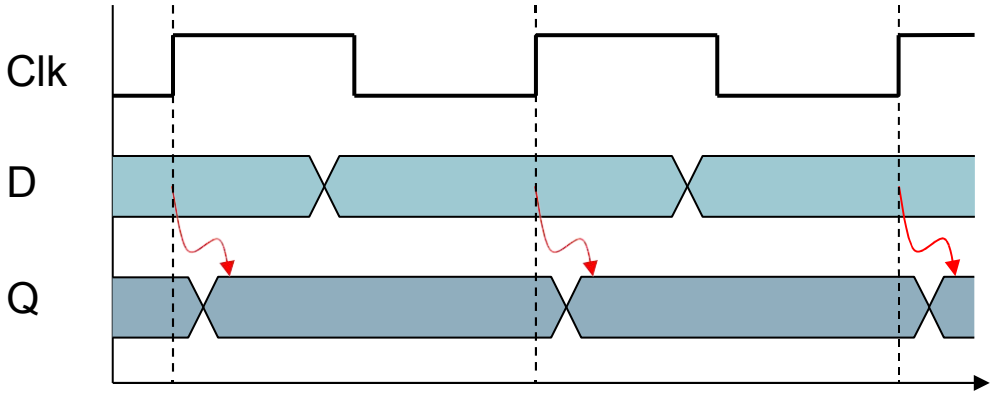
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

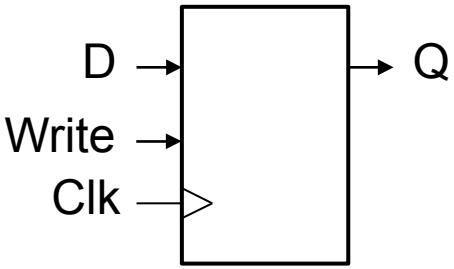
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered
    - rising edge: update when Clk changes from 0 to 1
    - falling edge: update when Clk changes from 1 to 0

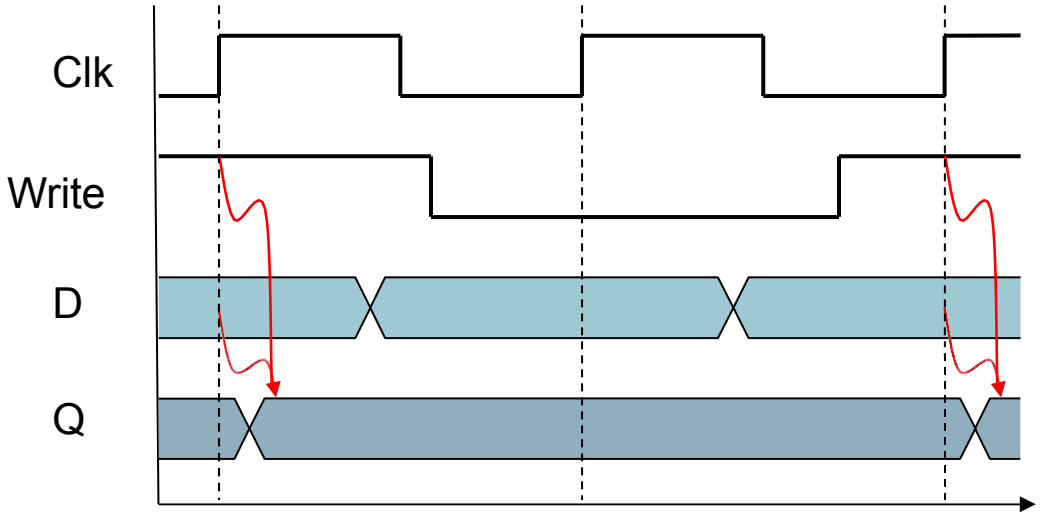




Sequential Elements

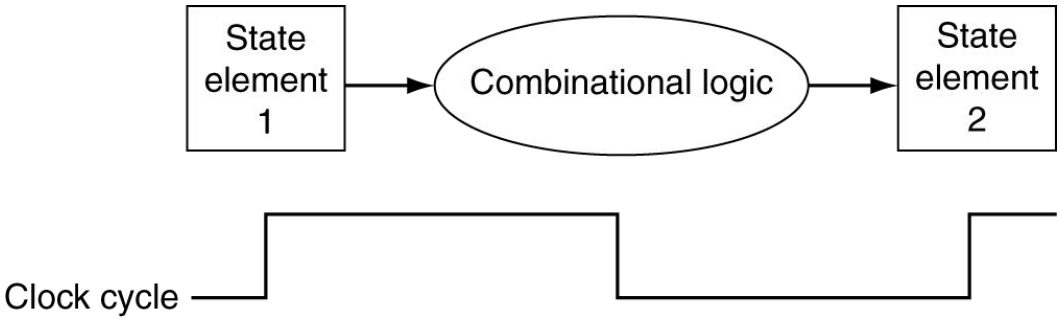
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

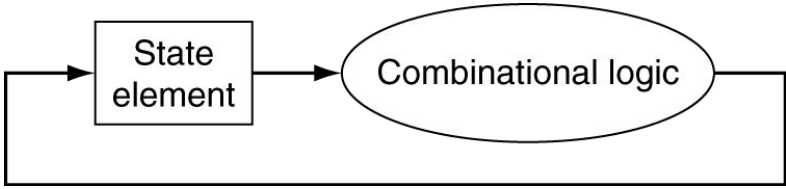




Clocking Methodology

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





add – 5ns

mul – 8ns

sub – 5ns

lw – 10ns

clock period – 10ns

Total time for all Instruction = 10 +10 +10 + 10 = 40ns

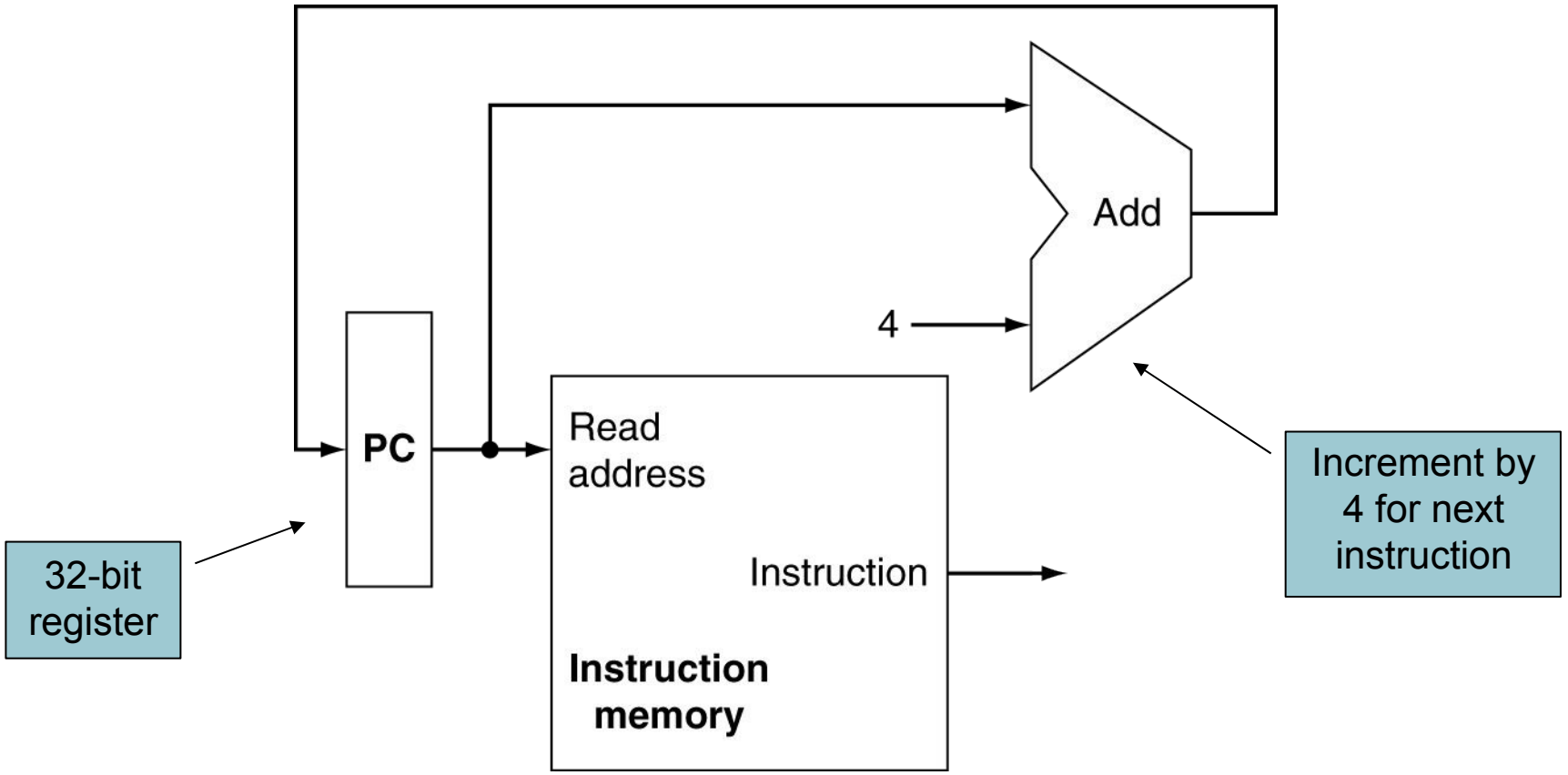
Actual time for all Instruction = 5 +8 +5 + 10 = 28ns

Time Waste= 40 – 28 = 12ns

Building a Datapath

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

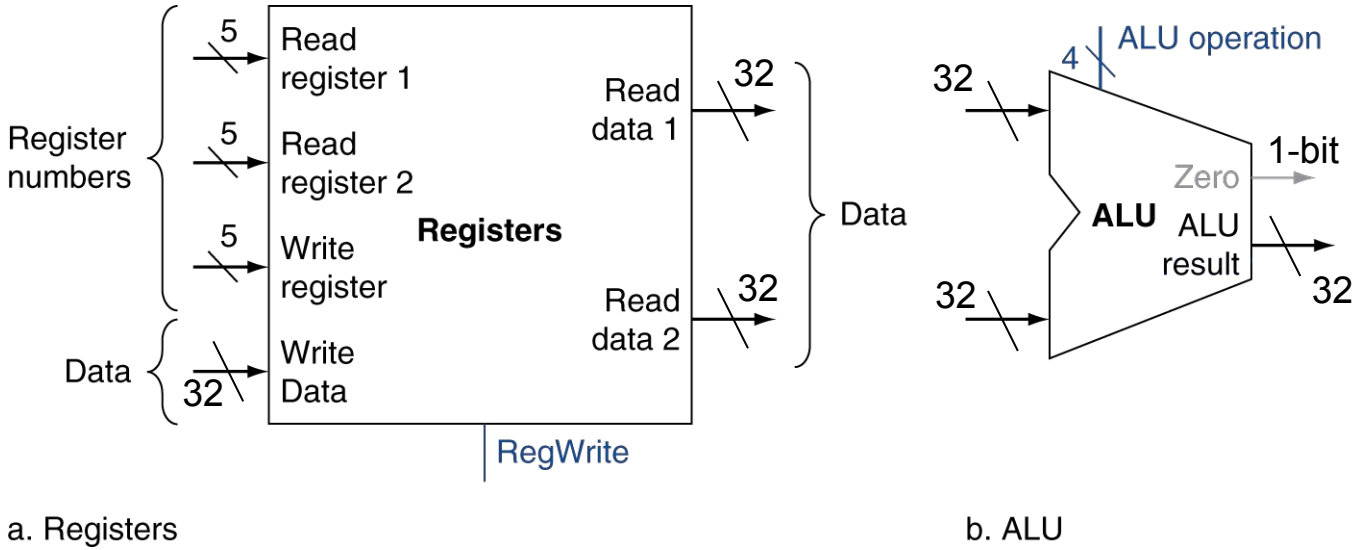
Instruction Fetch



1 Clock Cycle Per Instruction

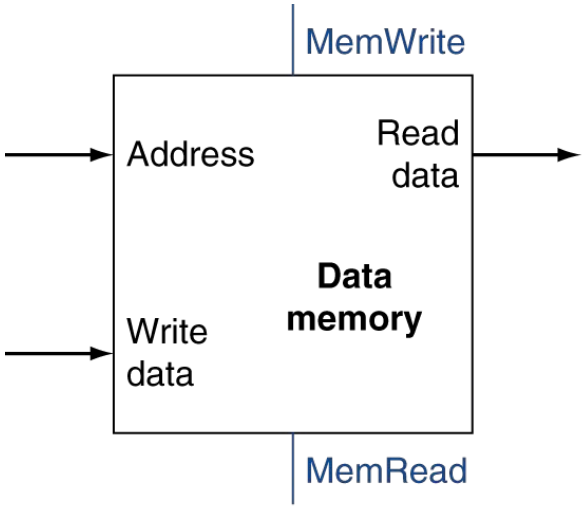
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result
- `Ex. : add $t0, $s2, $t0`

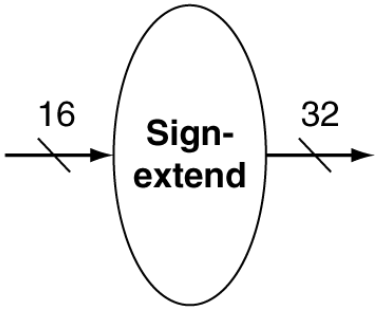


Load/Store Instructions

- lw $t1, offset_value($t2)
  - offset_value: 16-bit signed offset
- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

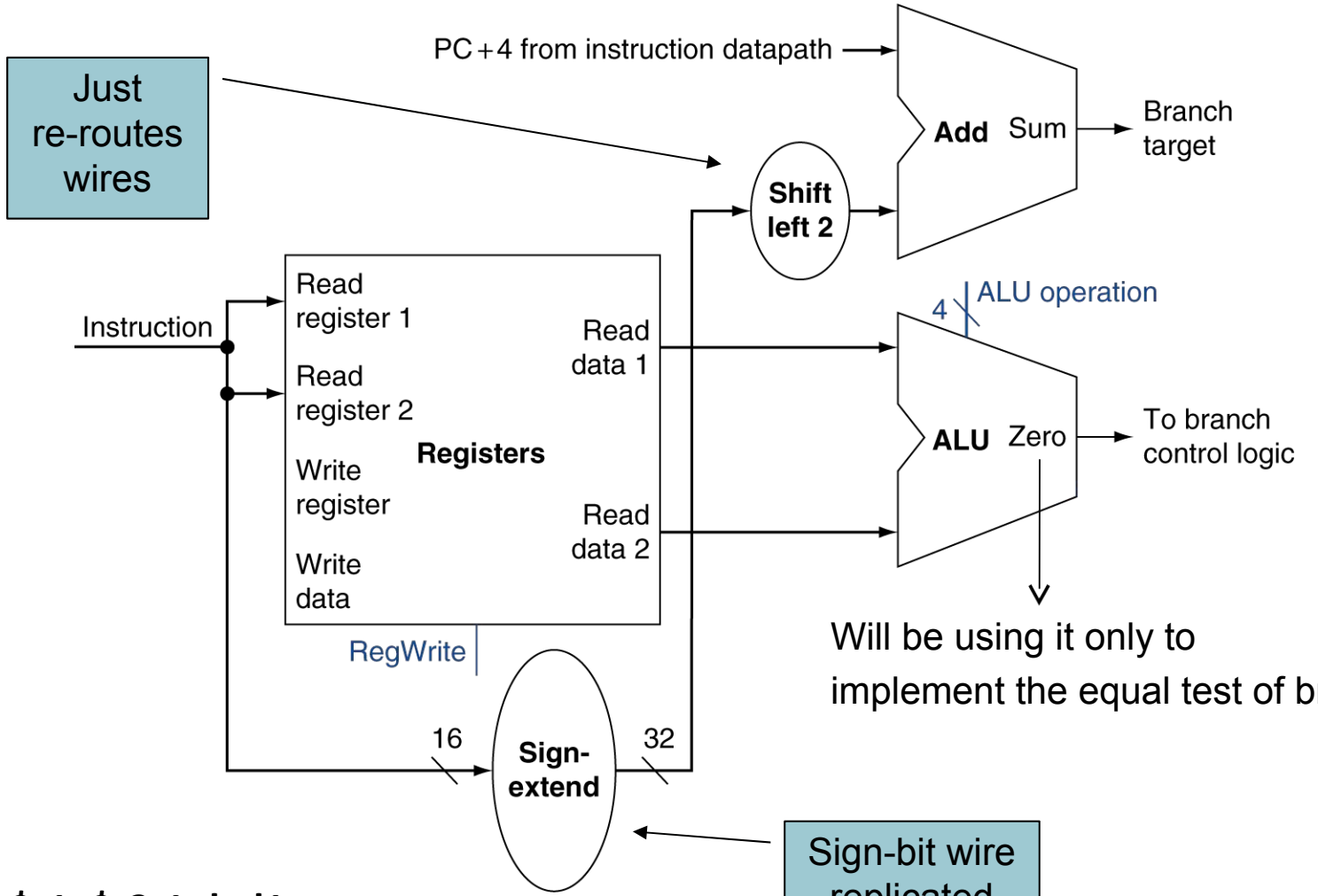


b. Sign extension unit

Branch Instructions

- beq $t1, $t2, offset
  - if (t1 == t2) branch to instruction labeled offset
  - Target address = PC + offset × 4
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
  - Already calculated by instruction fetch

Branch Instructions



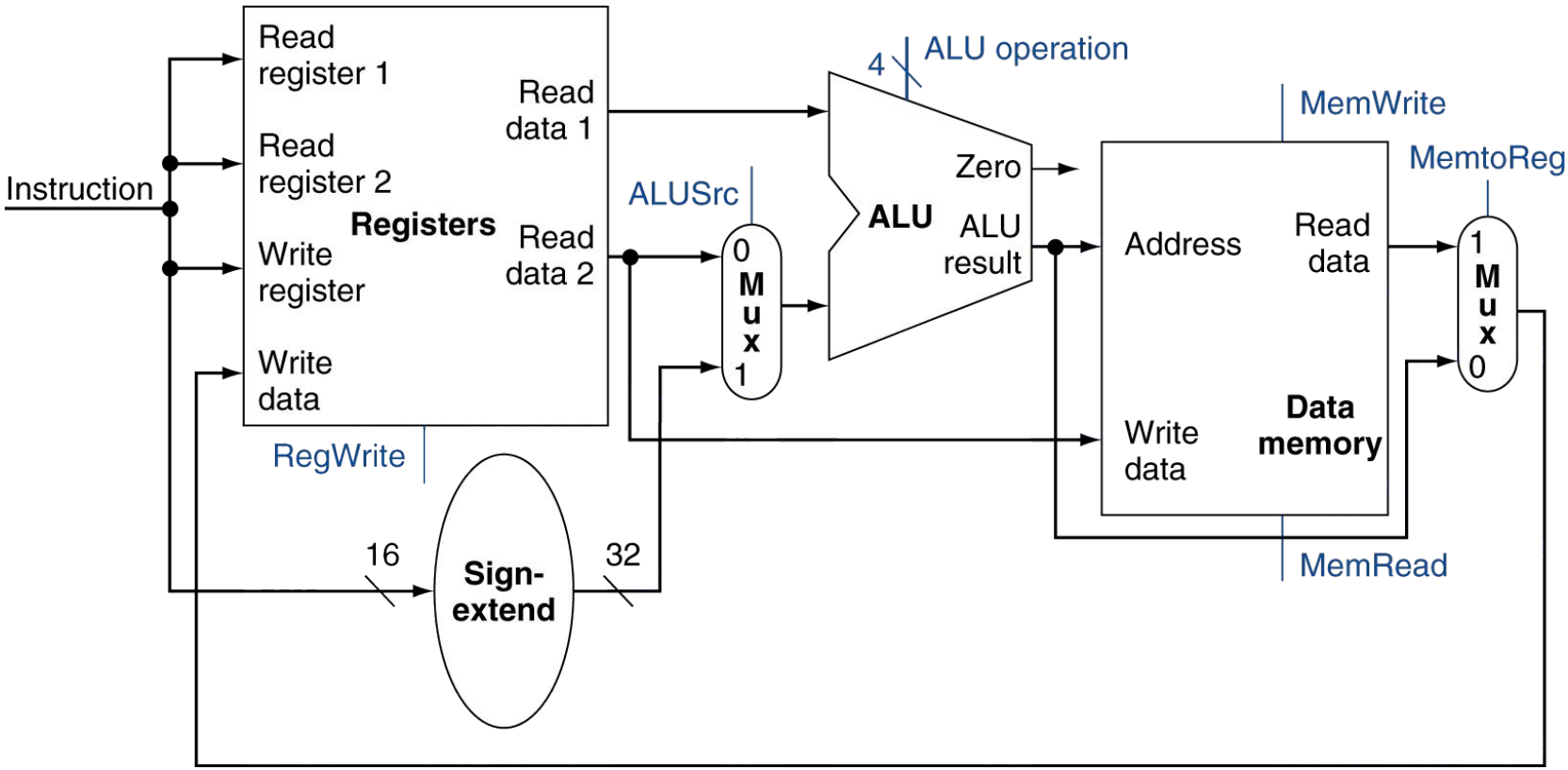
Will be using it only to implement the equal test of branches

beq $s1, $s2, Label1

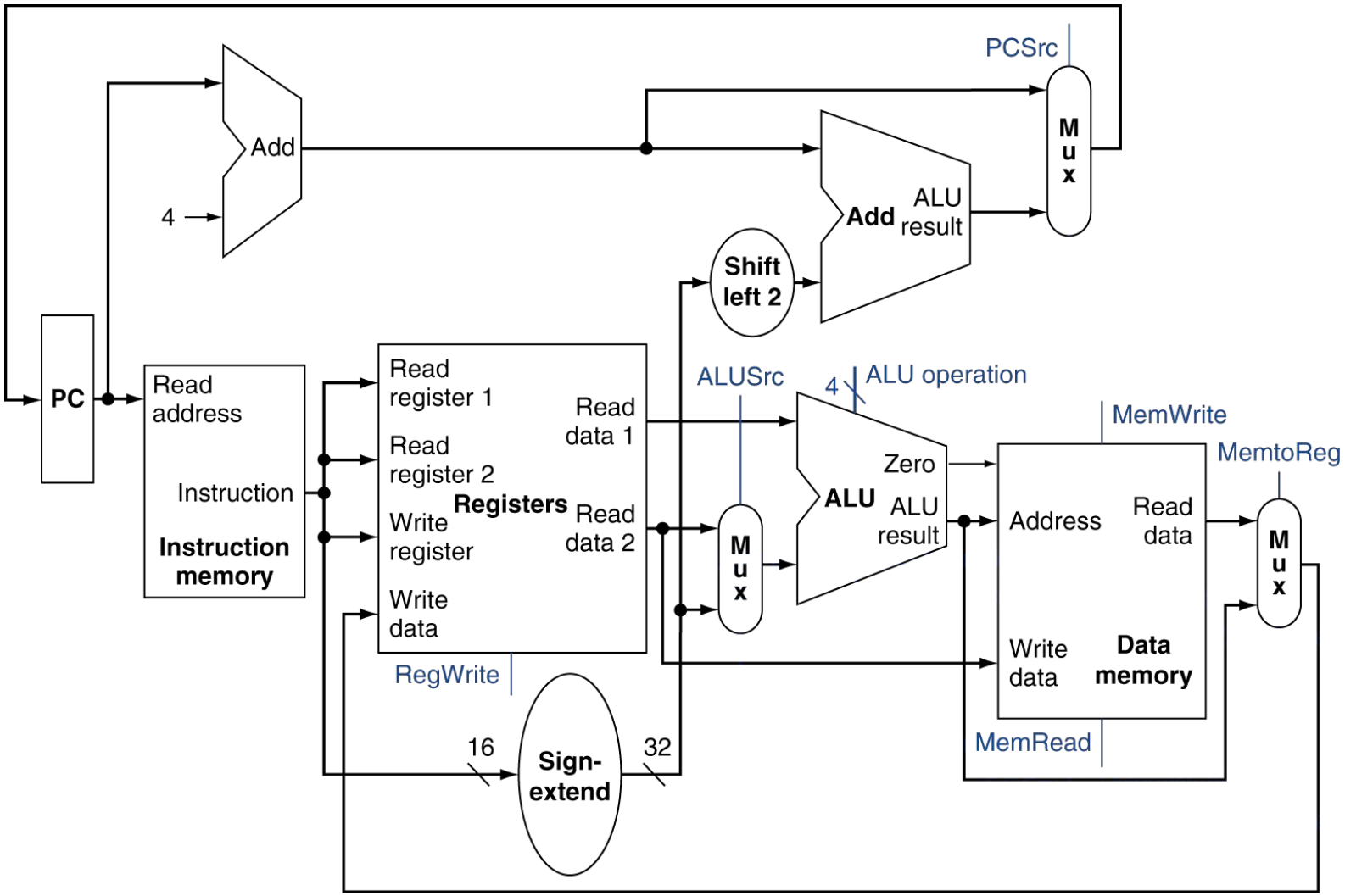
Composing the Elements

- 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
- To share a datapath element between two different instruction classes
  - use multiplexers and control signal

R-Type/Load/Store Datapath



Full Datapath



ALU Control

lw $8, 20($7)

Exact Mem Location = Offset + Base Address

beq $7, $8, L1

Value of $7 – Value of $8 (to check equality)

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

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU Control

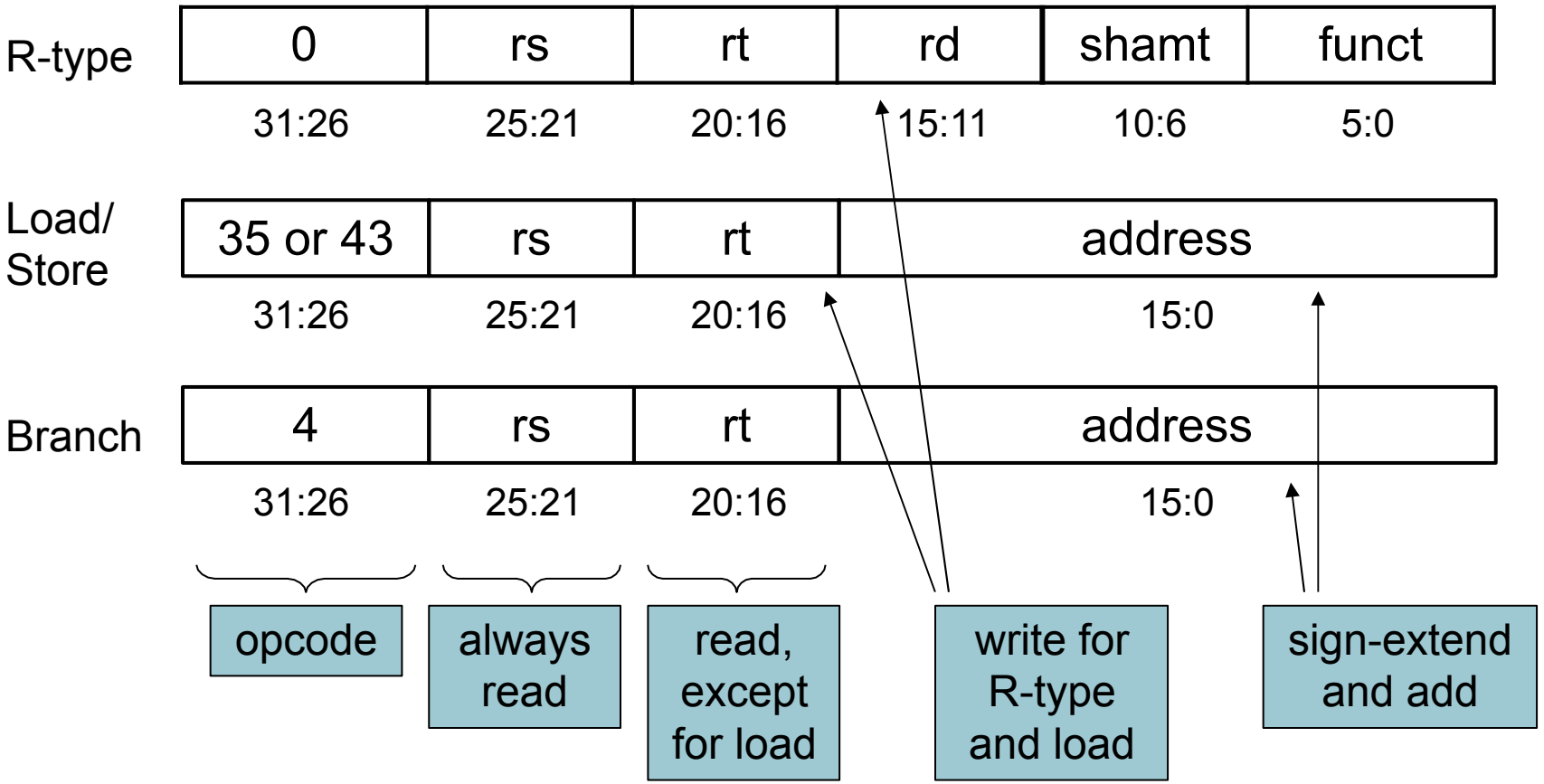
- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

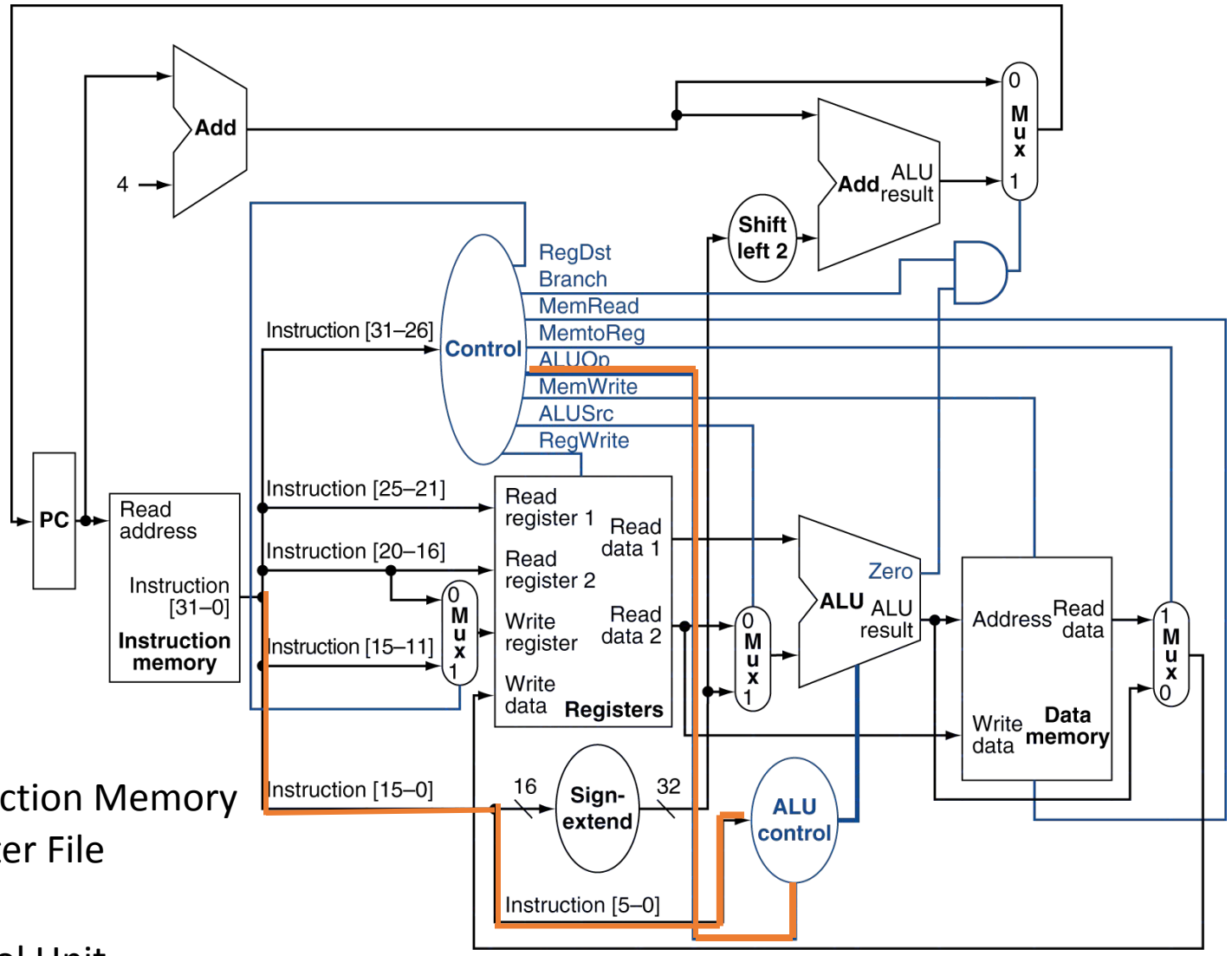
The Main Control Unit

lw $8, 20($7)

- Control signals derived from instruction



Datapath With Control

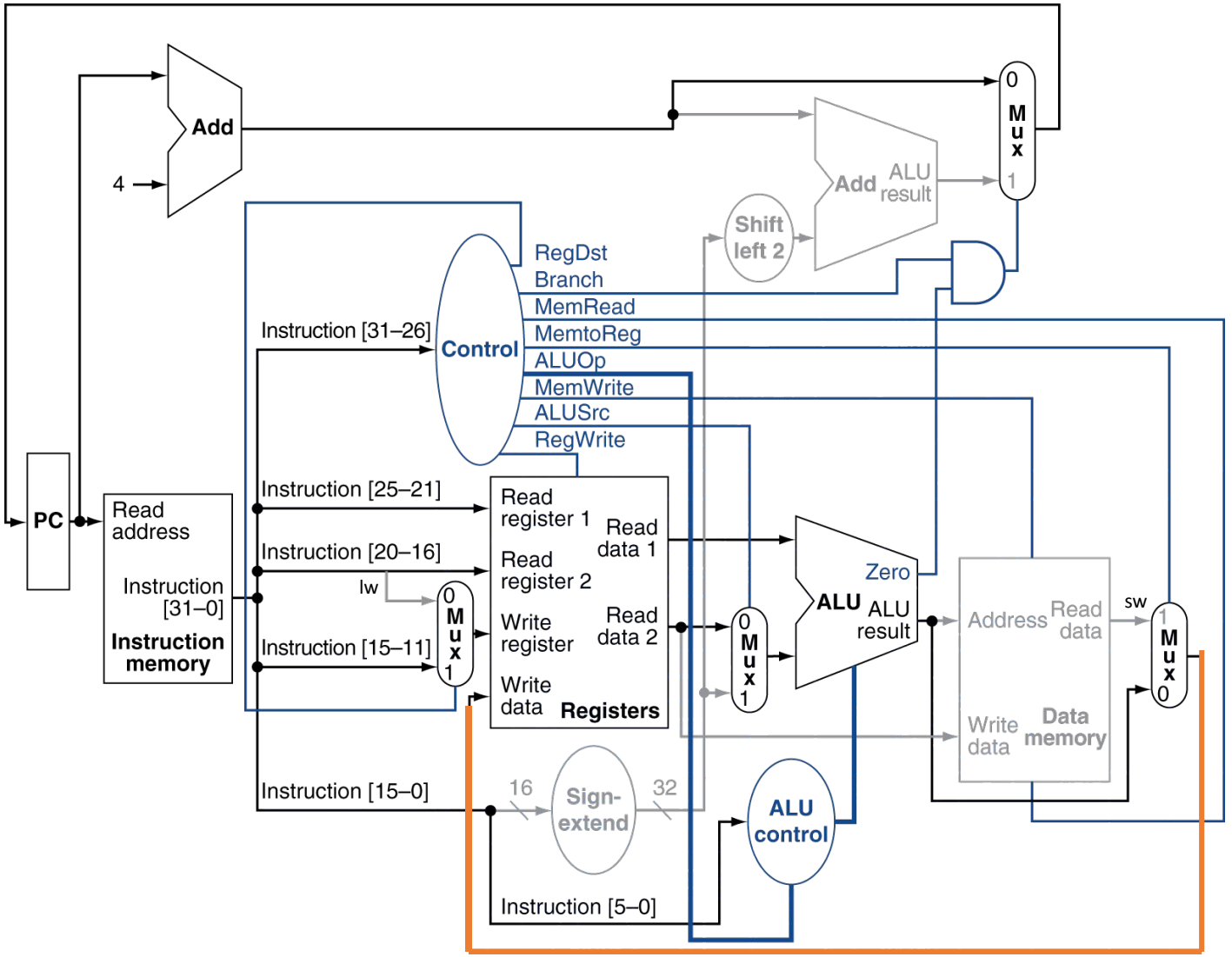


1. PC
2. Instruction Memory
3. Register File
4. ALU
5. Control Unit
6. Data Memory (lw/sw)

R-Type Instruction

add $8, $17, $18

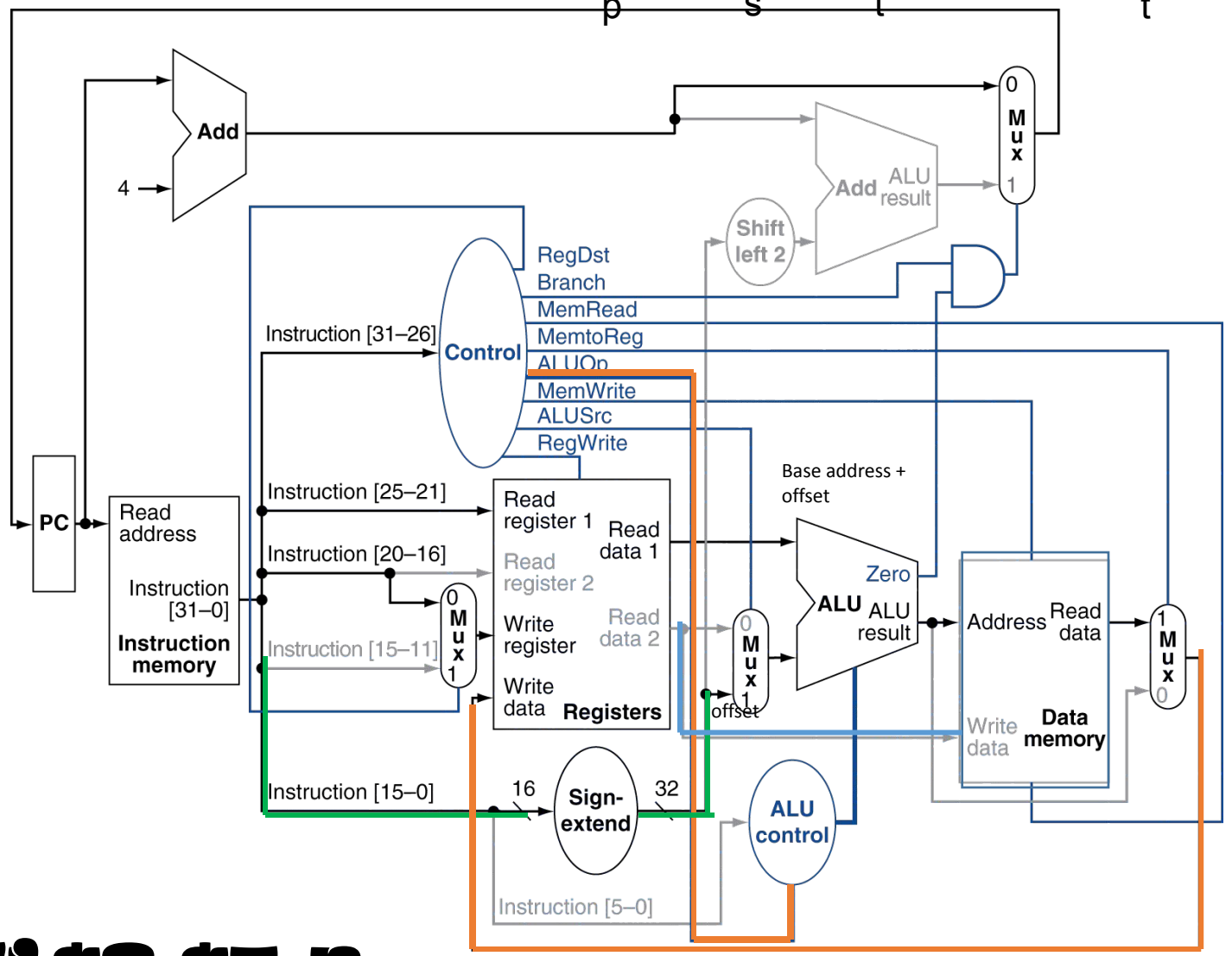
000000	10001	10010	[illegible]	00000	100000
opcode	rs	rt	rd	shamt	funct



Load Instruction

lw $8, 20($7)

op	$7	$8	20
op	rs	rt	offset

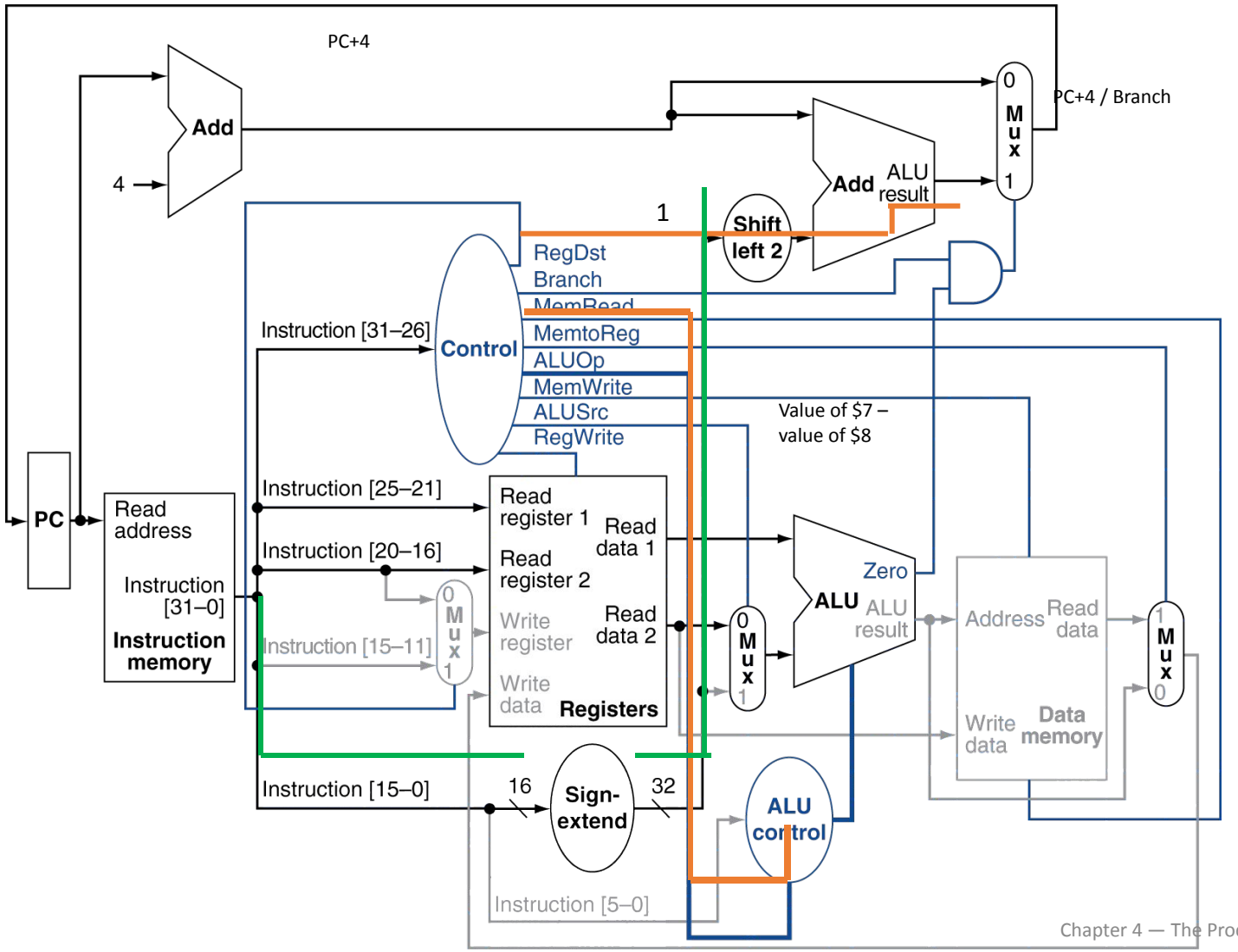


addi $8,$7, 3

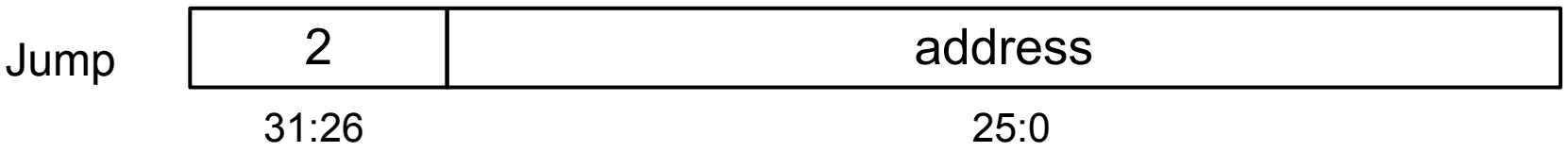
Branch-on-Equal Instruction

beq $7, $8, L1

Op	$7	$8	2
op	rs	rt	offset

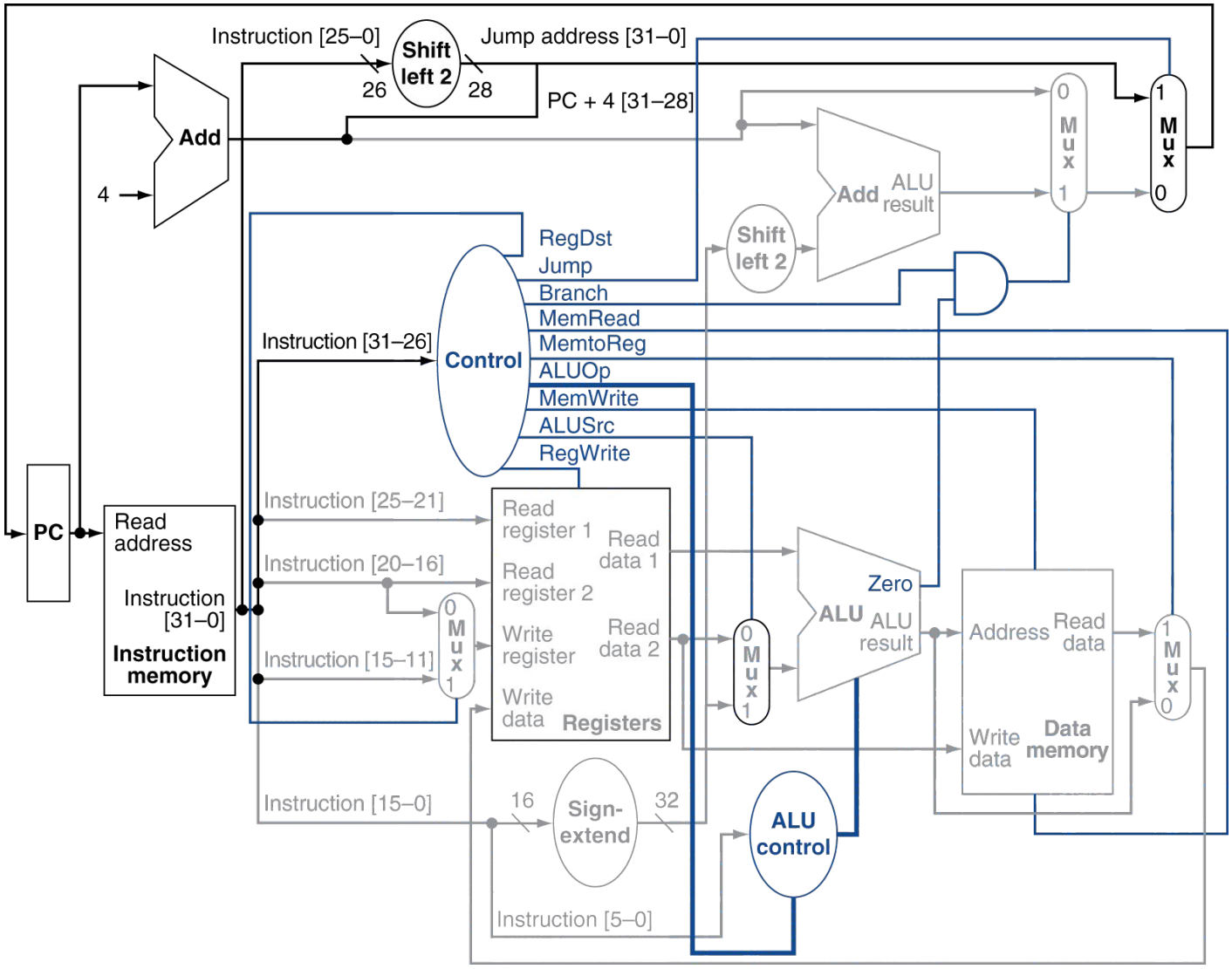


Implementing Jumps

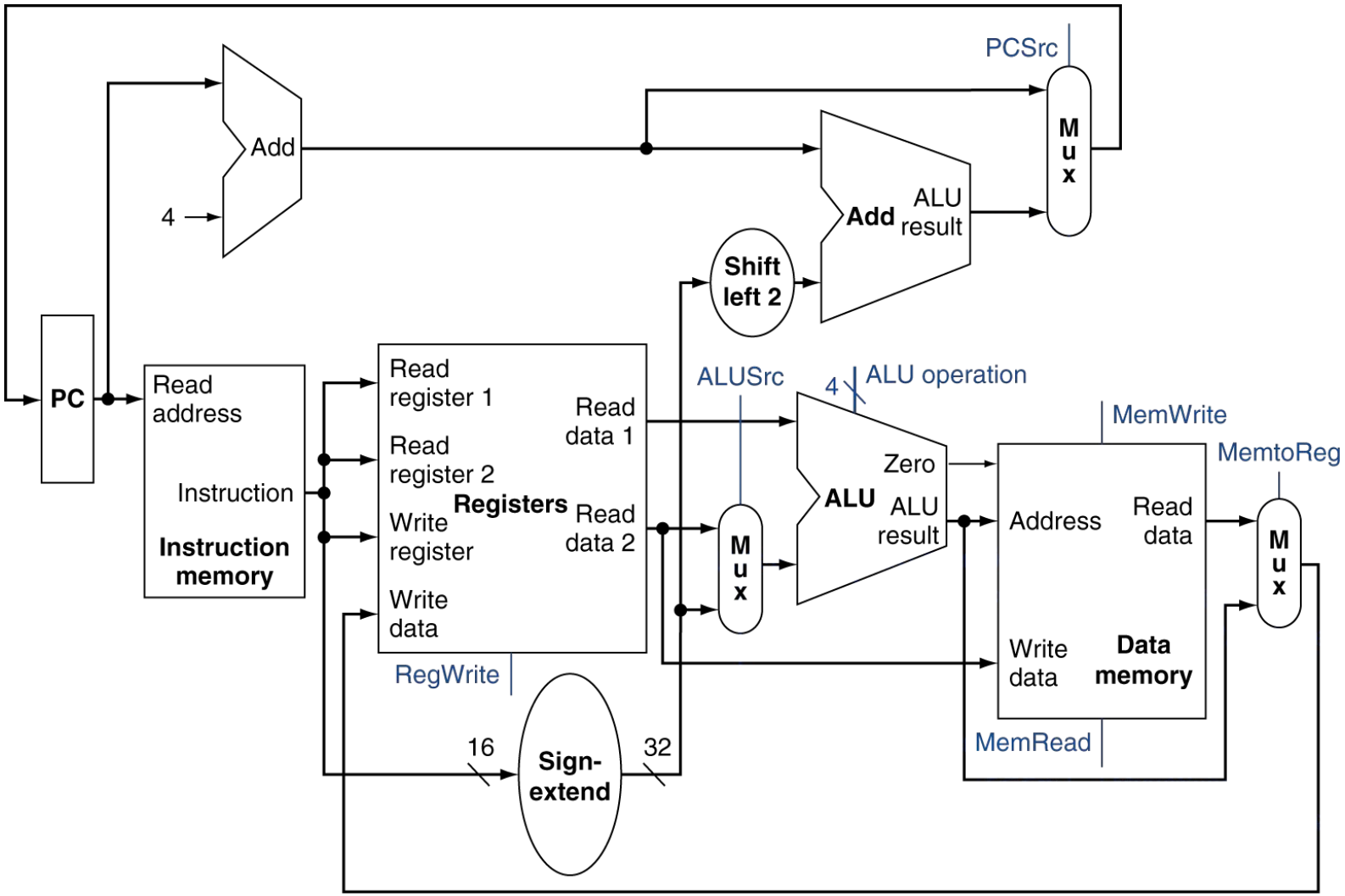


- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



Full Datapath



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

add – 5ns

mul – 8ns

sub – 5ns

lw – 10ns

Total time for all Instruction = 10 +10 +10 + 10 = 40ns

Actual time for all Instruction = 5 +8 +5 + 10 = 28ns

Time Waste= 40 – 28 = 12ns

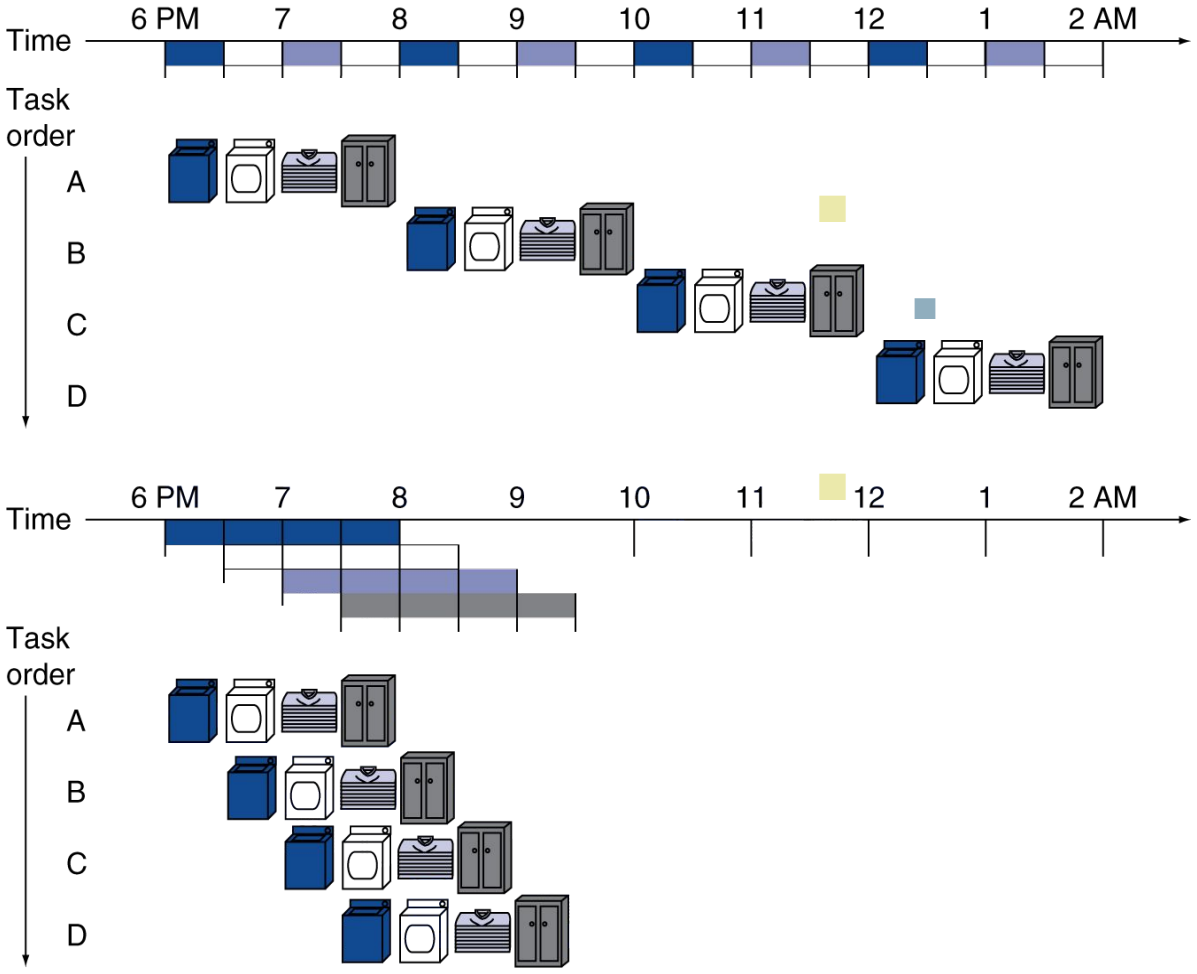
Pipelining Analogy

add $8, $17, $1

lw $t0, 12 ($s0)

sw $t1, 24 ($s1)

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



Instruction: Cloth
1. Wash
2. Dry
3. Fold
4. Store

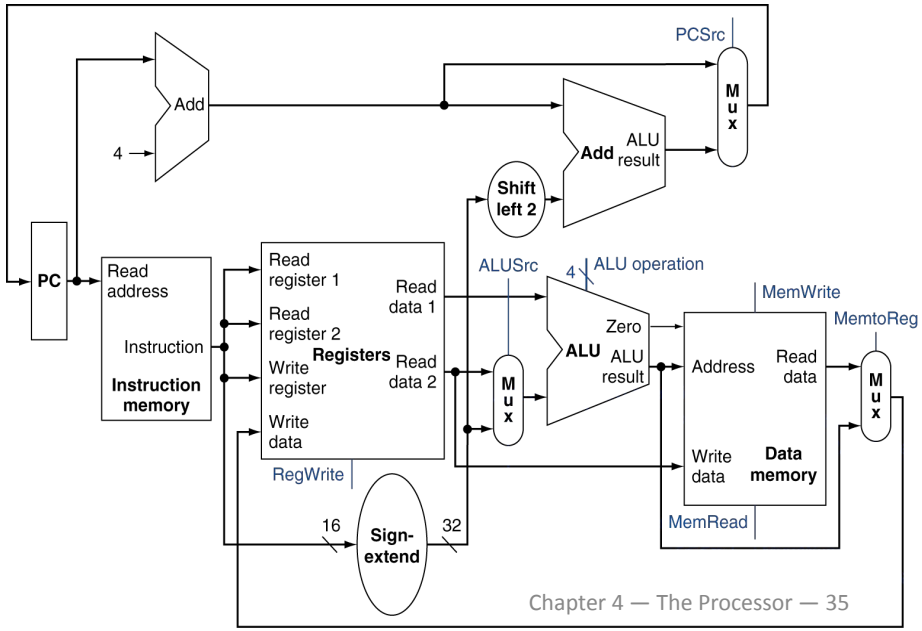
MIPS Pipeline

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

add $8, $17, $18

- lw $t0, 12 ($s0)
- sw $t1, 24 ($s1)



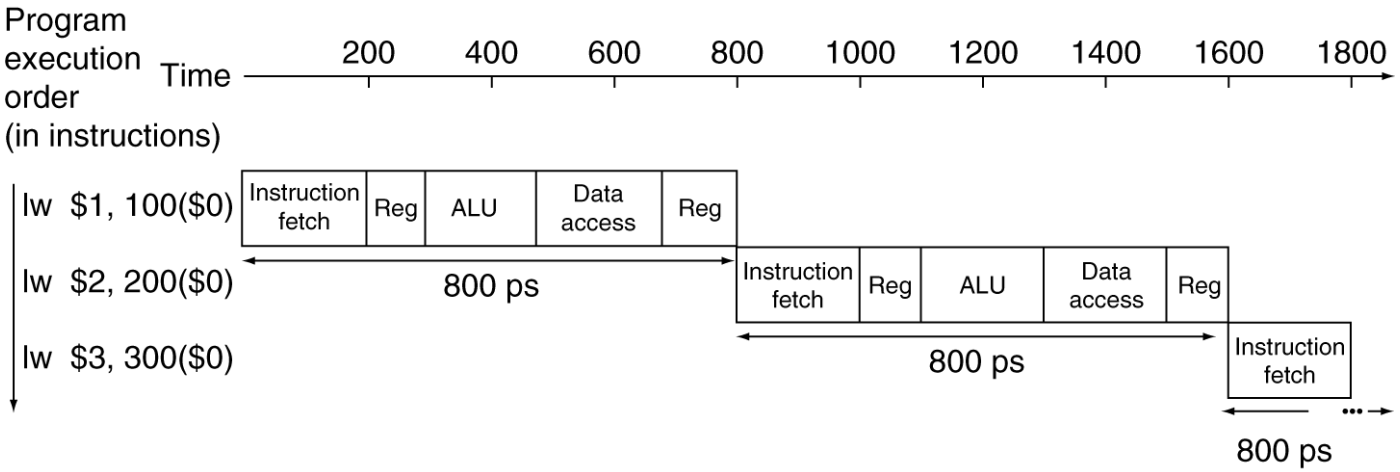
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

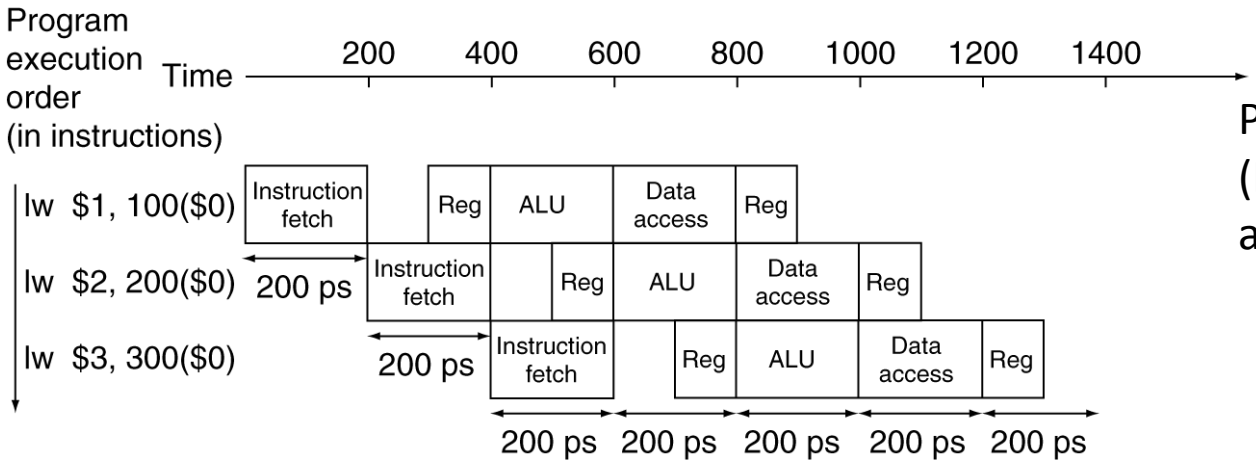
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single Cycle Clock Cycle Tc = (Highest) Duration to complete an instruction



Tc = 800ps



Pipelined Clock Cycle Tc = (Highest) Duration to complete a stage

Tc = 200ps

Pipelining increases throughput

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If not balanced, speedup is less

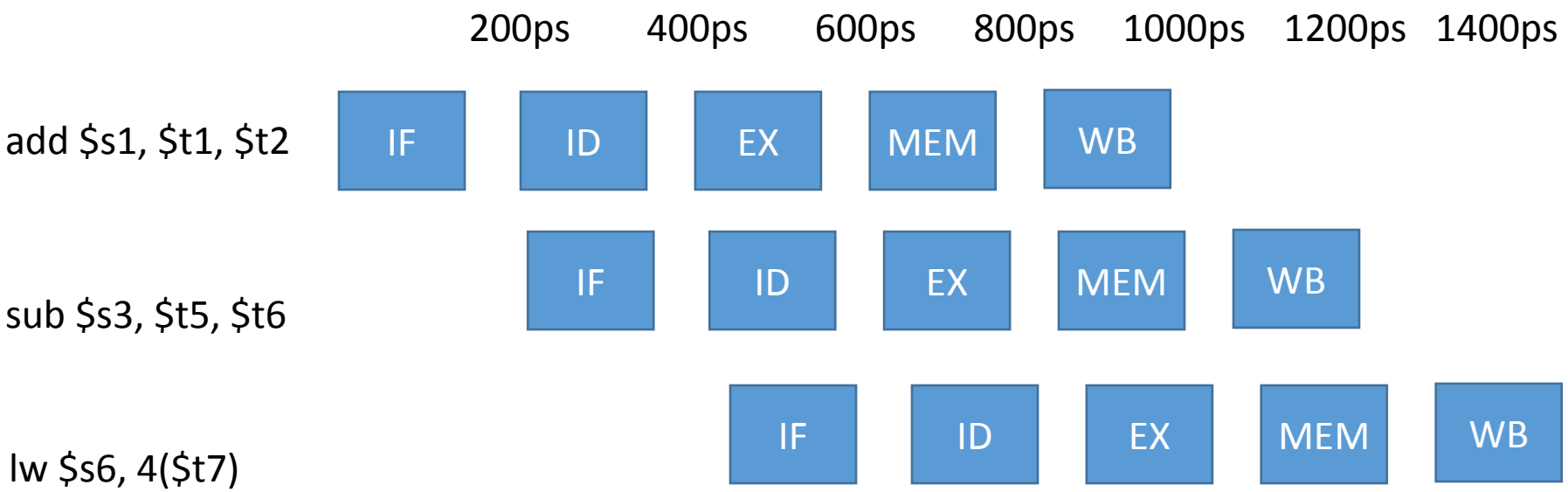
Speedup due to increased throughput

- Latency (time for each instruction) does not decrease

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Pipelining


200ps
400ps
600ps
800ps
1000ps
1200ps
1400ps
add $s1, $t1, $t2
IF
ID
EX
MEM
WB
sub $s3, $t5, $t6
IF
ID
EX
MEM
WB
lw $s6, 4($t7)
IF
ID
EX
MEM
WB

Hazards

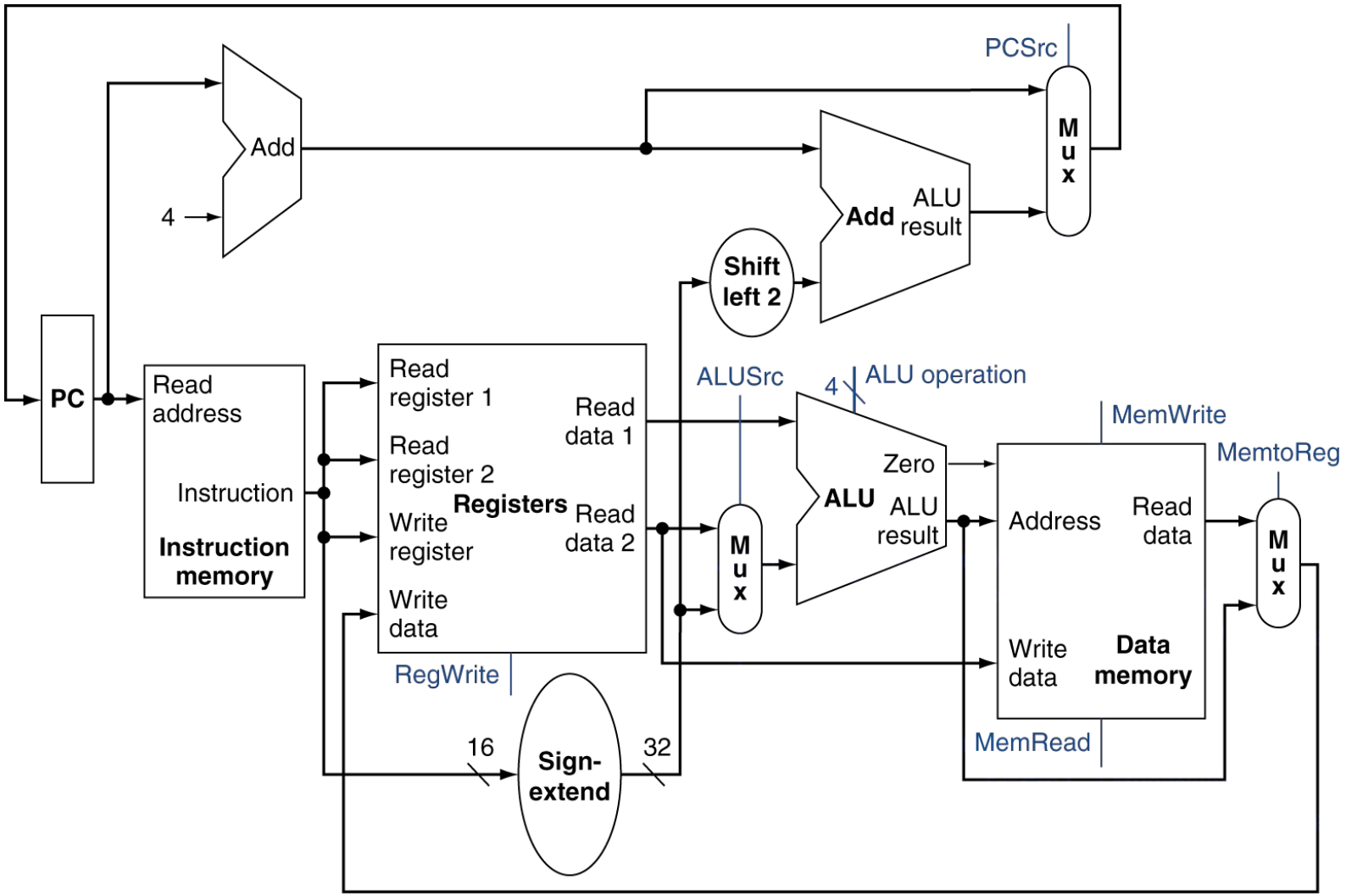
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

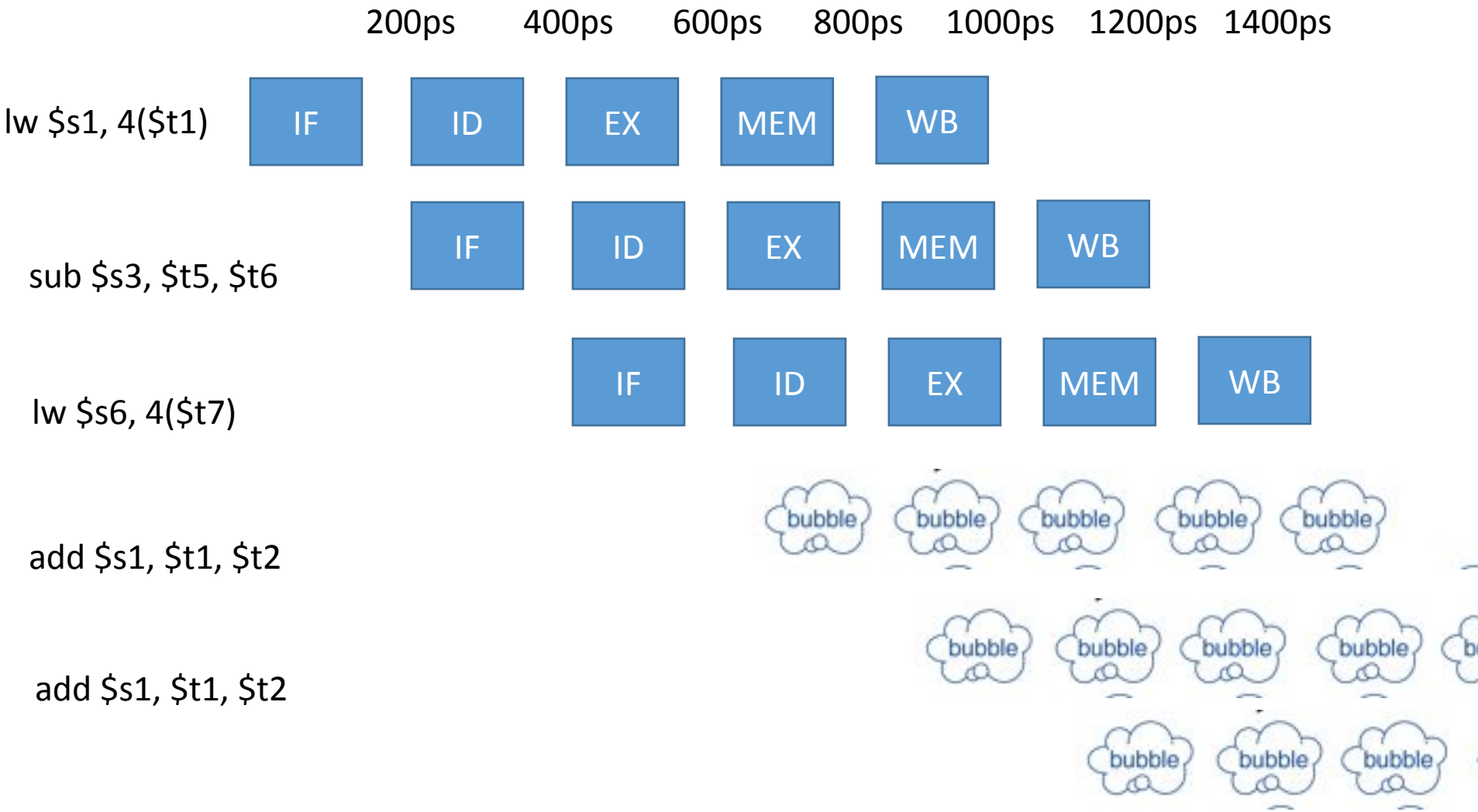
Consider for Von Neuman Architecture

Full Datapath



Pipelining

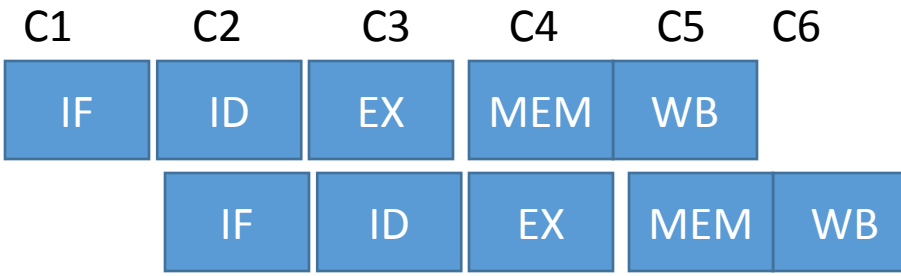
Consider for Von Neuman Architecture

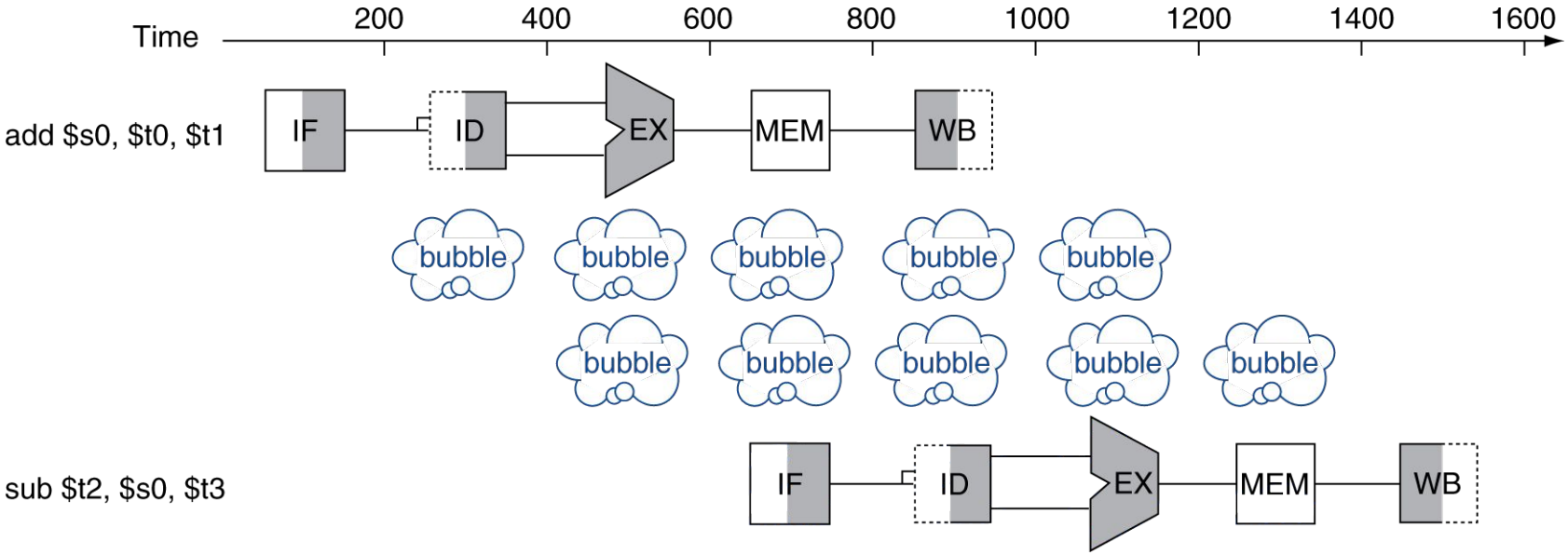


Data Hazards

Stall

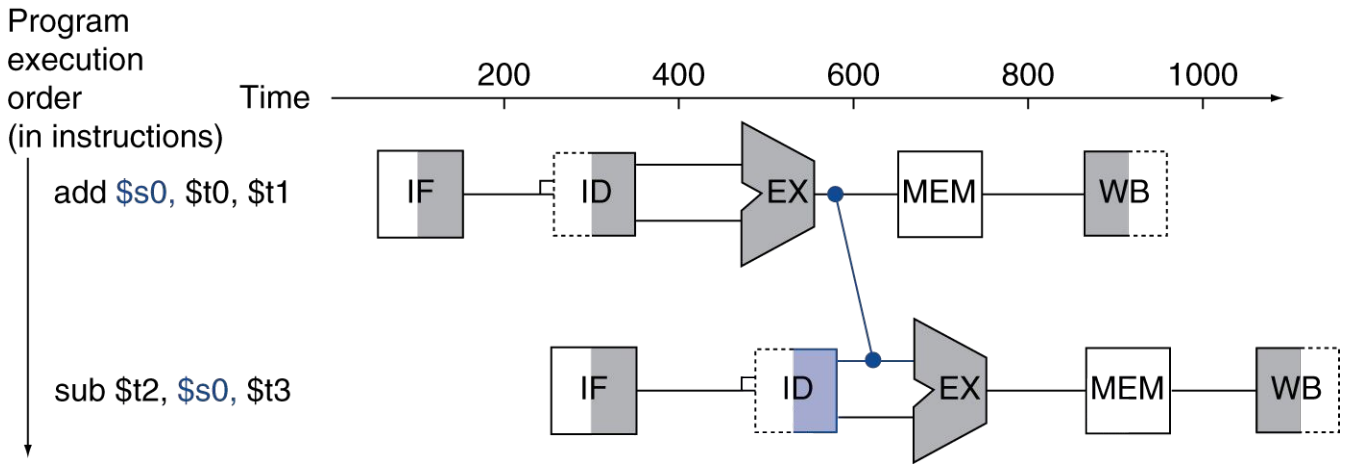
- An instruction depends on completion of data access by a previous instruction
  - `ad  $s0, $t0, $t1`
  - `d`
  - `sub  $t2, $s0, $t3`





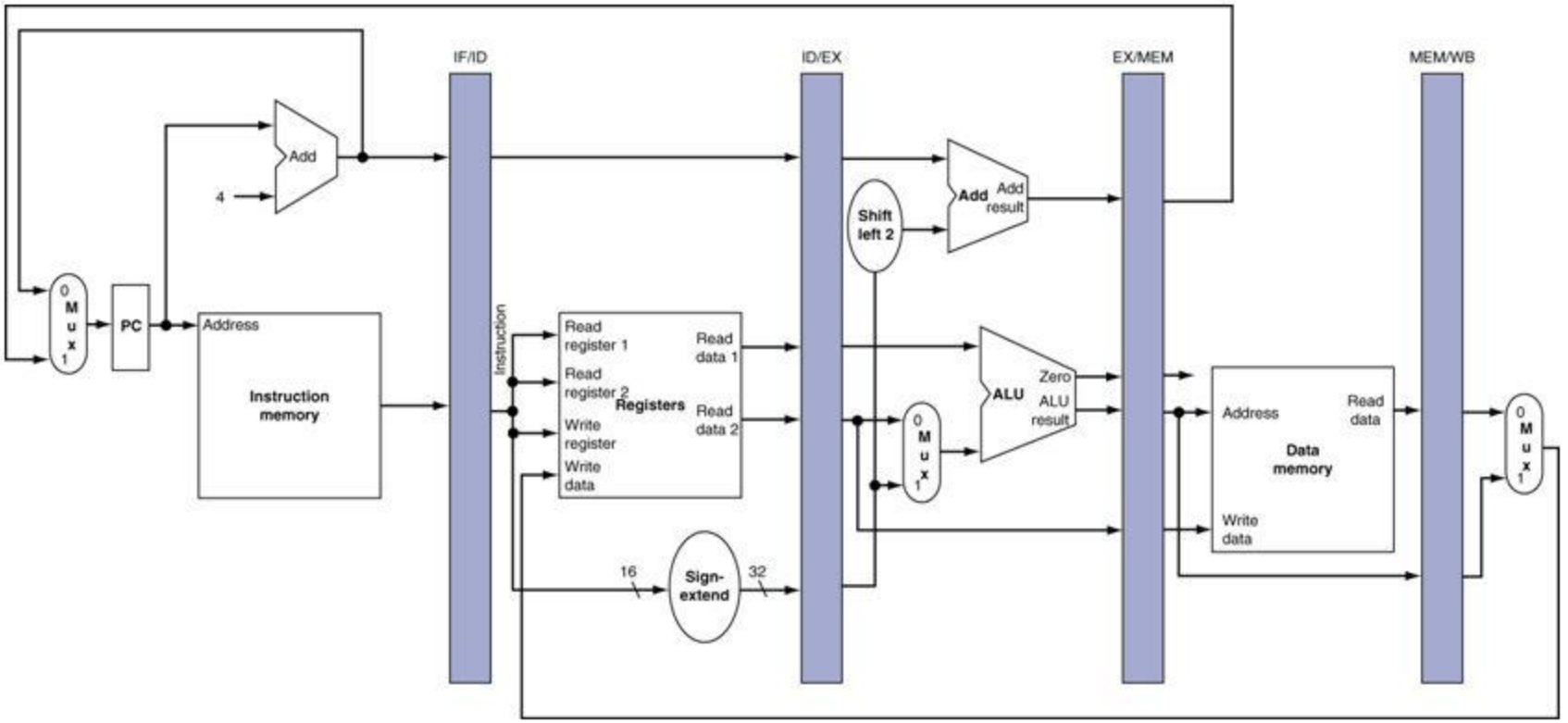
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



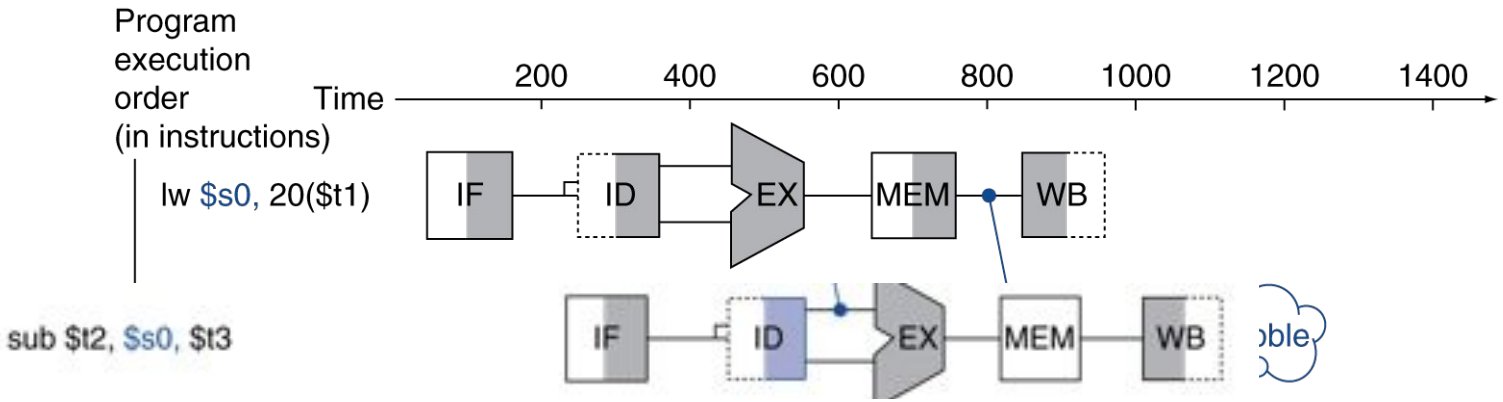
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle



Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!

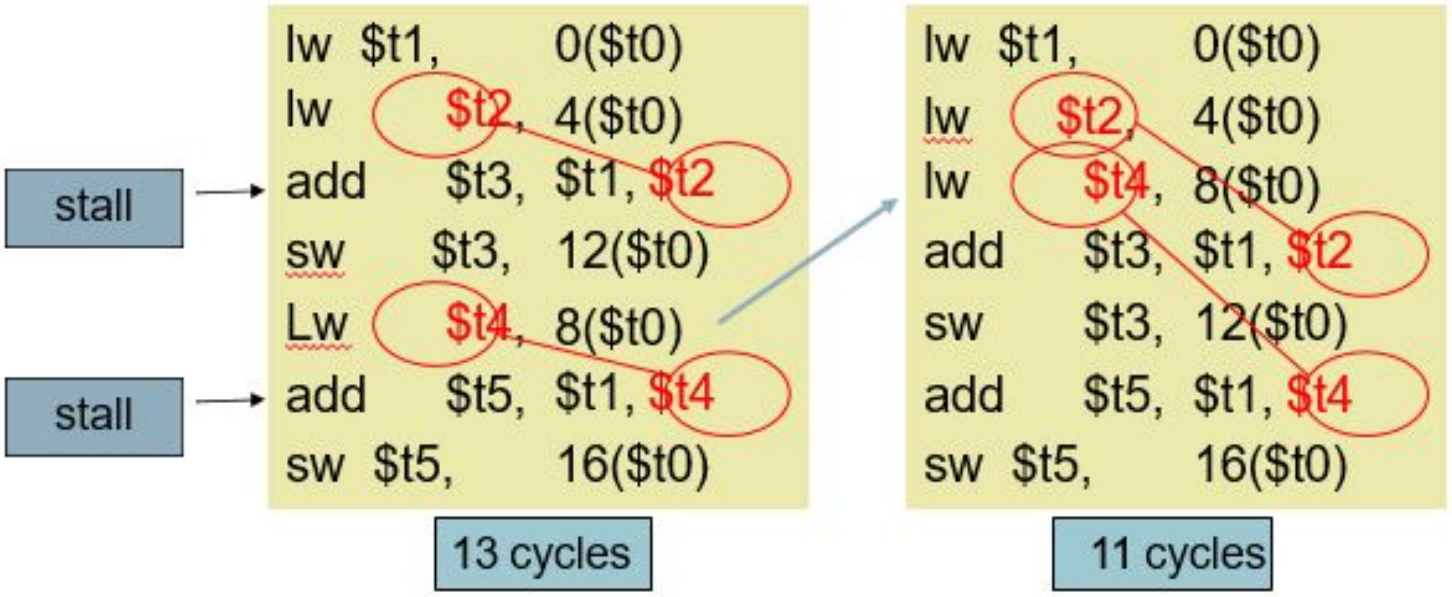


add $s9, $t9, $t8

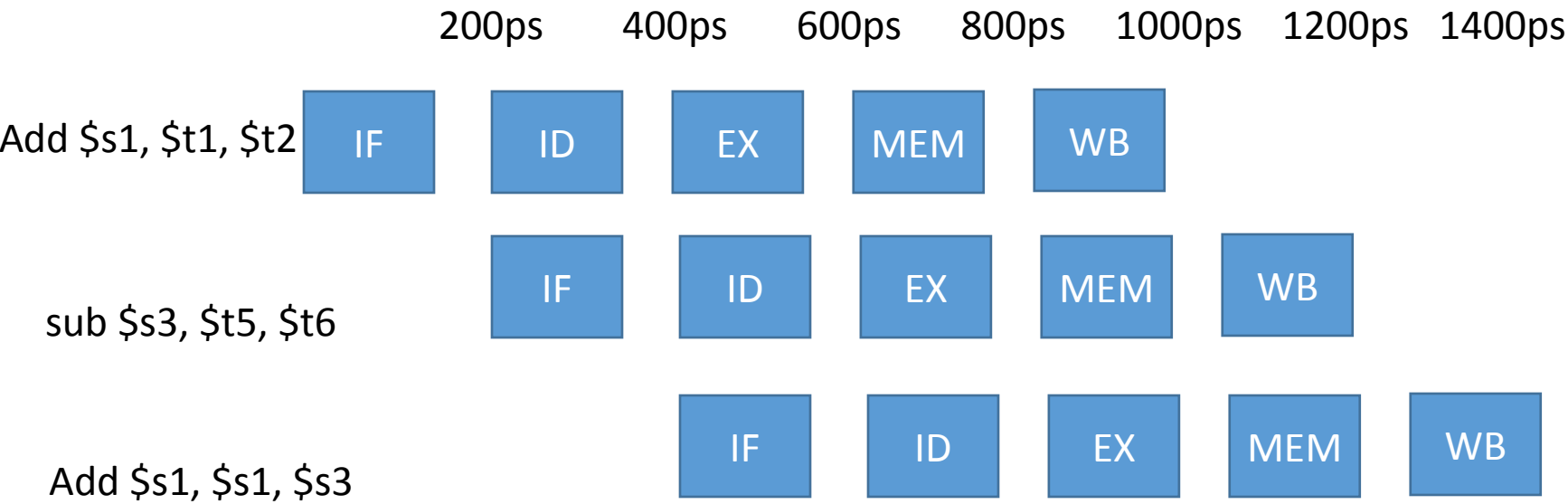
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

C code for `A = B + E; C = B + F;`



Pipelining

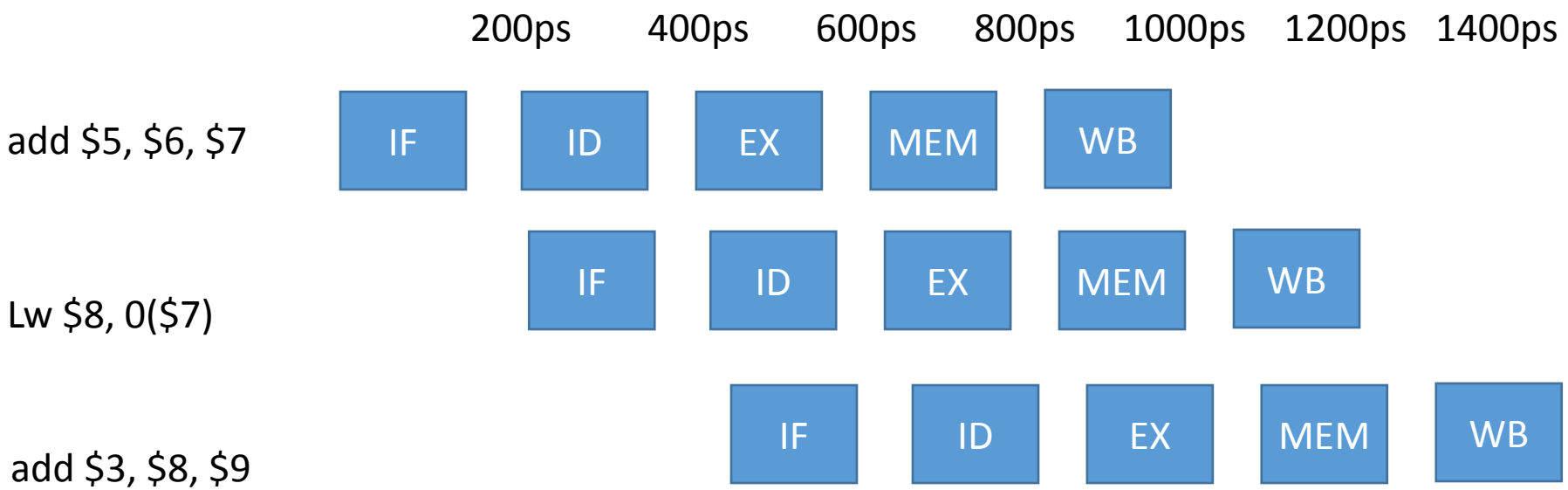

200ps
400ps
600ps
800ps
1000ps
1200ps
1400ps
Add $s1, $t1, $t2
IF
ID
EX
MEM
WB
sub $s3, $t5, $t6
IF
ID
EX
MEM
WB
Add $s1, $s1, $s3
IF
ID
EX
MEM
WB

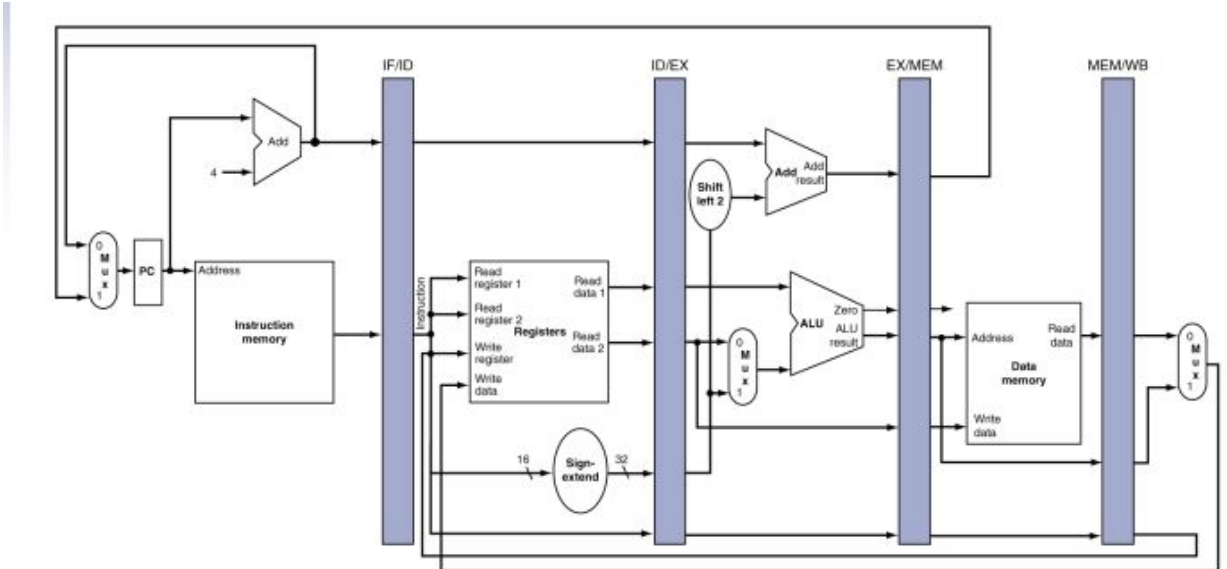
Pipelining

200ps 400ps 600ps 800ps 1000ps 1200ps 1400ps

Sw $s1,20 ($t0)

Pipelining


200ps
400ps
600ps
800ps
1000ps
1200ps
1400ps
add $5, $6, $7
IF
ID
EX
MEM
WB
Lw $8, 0($7)
IF
ID
EX
MEM
WB
add $3, $8, $9
IF
ID
EX
MEM
WB


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign-extend
32

Pipelining Practice

- Consider the following MIPS code:

```
And $1, $2, $3
Lw $3, 40 ($1)
Lw $5, 24 ($3)
Add $7, $5, $6
Sub $2, $5, $3
Sll $1, $3, 5
```

- Total number of Data Hazards? Answer: 4
- If we overcome the data hazard of the above code using Only Stalling, draw the pipelining diagram. What is the CPI? Answer: 16/6
- If we overcome the data hazard of the above code using Stalling and Forwarding, draw the pipelining diagram. What is the CPI? Answer: 12/6
- If we overcome the data hazard of the above code using Stalling, Forwarding and Code Scheduling, draw the pipelining diagram. What is the CPI? Answer: 12/6

Quiz Pipeline Practice (practice with only stalling as well)

2. **Solve the data hazards of the following code using Forwarding + Stalling. (show the forwarding with arrow) (8 Marks)**

```
sub $9, $8, $10
sw $9, 4 ($8)
lw $8, 4 ($9)
add $12, $8, $12
lw $18, 8 ($9)
addi $18, $12, -8
```

2. **Solve the data hazards of the following code using Only Stalling + Forwarding. (show the forwarding with arrow). (8 Marks)**

```
sw $8, 4 ($9)
or $9, $8, $10
lw $9, 20 ($9)
ori $8, $9, 10
and $11, $10, $8
lw $11, 0($9)
```

Quiz Pipeline Practice (practice with only stalling as well)

2. **Solve the data hazards of the following code using Stalling + Forwarding. (show the forwarding with arrow) (8 Marks)**

```
addi $8, $9, 10
sw $9, 4 ($8)
lw $9, 8 ($8)
or $11, $10, $9
sw $9, 0 ($10)
lw $10, 4 ($9)
```

2. **Solve the data hazards of the following code using Forwarding + Stalling. (show the forwarding with arrow) (8 Marks)**

```
lw $8, 20 ($11)
sw $8, 4 ($9)
addi $12, $8, $11
and $14, $12, $8
sw $7, 20 ($14)
or $9, $7, $14
```