# Chapter 4

The Processor

Introduction

- 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
    - A more realistic pipelined version
- Simple subset, shows most aspects
    - Memory reference: `lw`, `sw`
    - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
   - Control transfer: `beq`, `j`

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 target address
  - Access data memory for load/store
  - PC $\leftarrow$ target address or PC + 4

CPU Overview
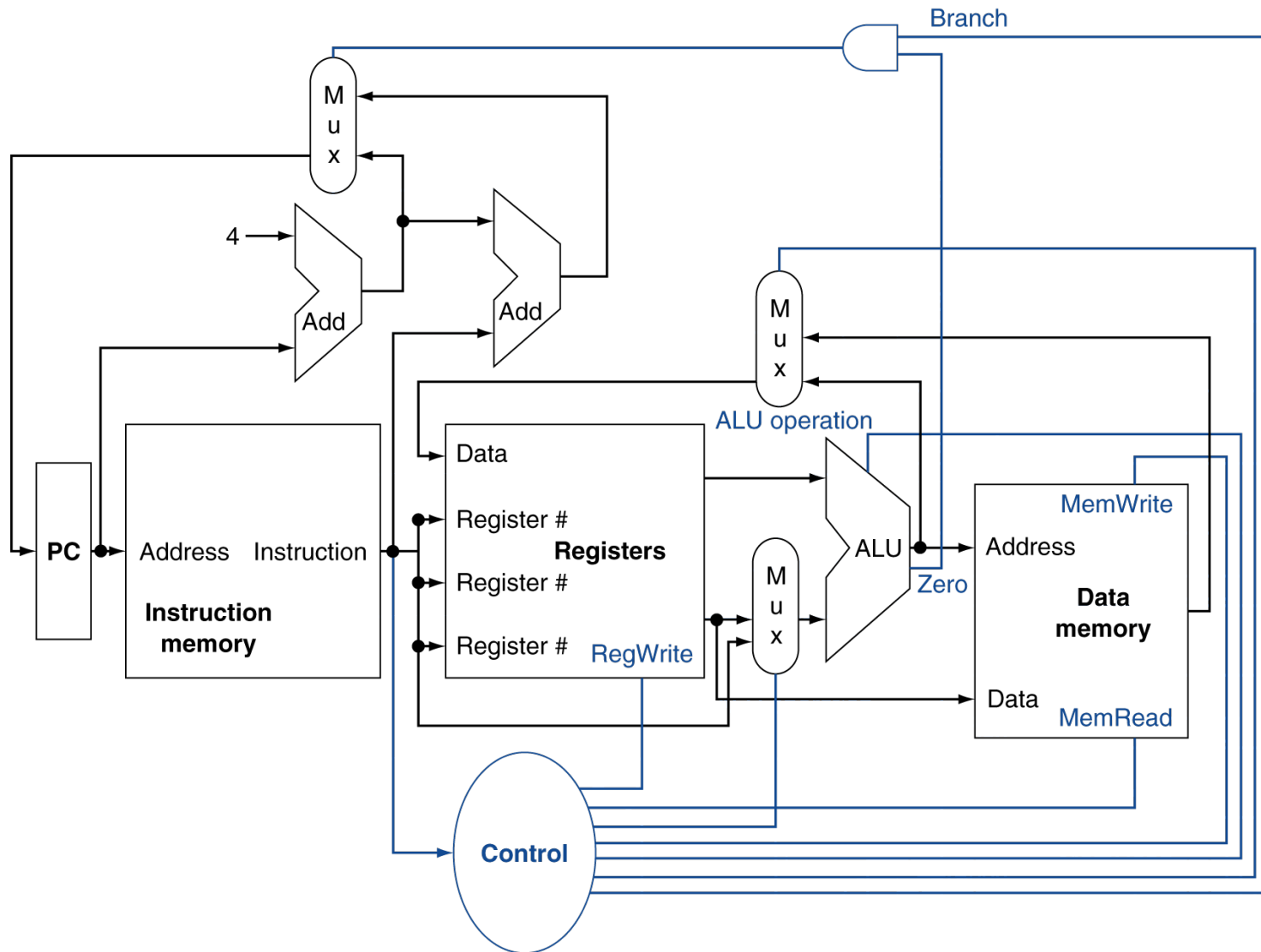
Multiplexers
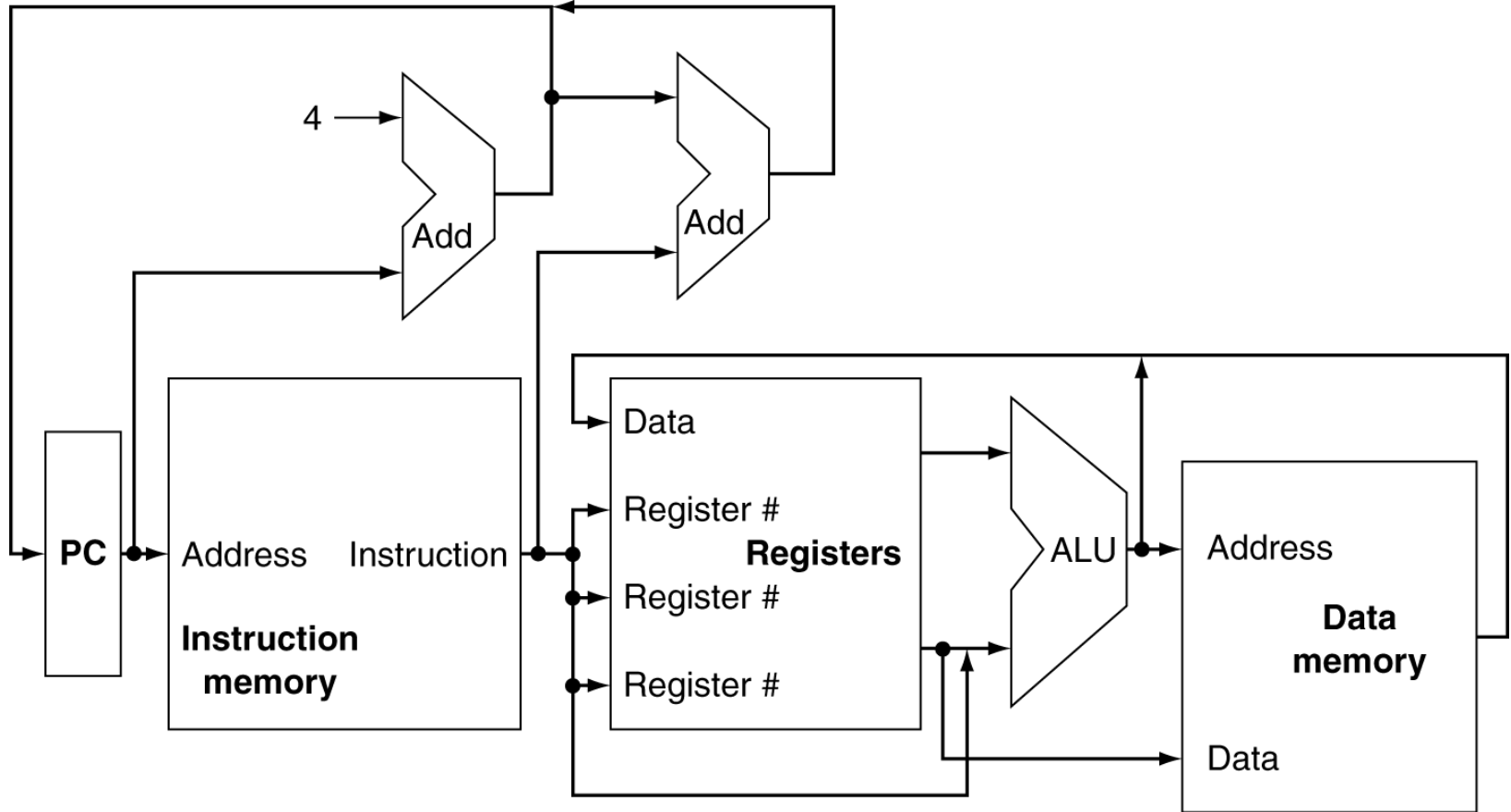
- Can't just join wires together
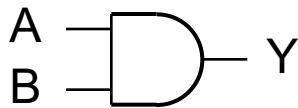 - Use multiplexers
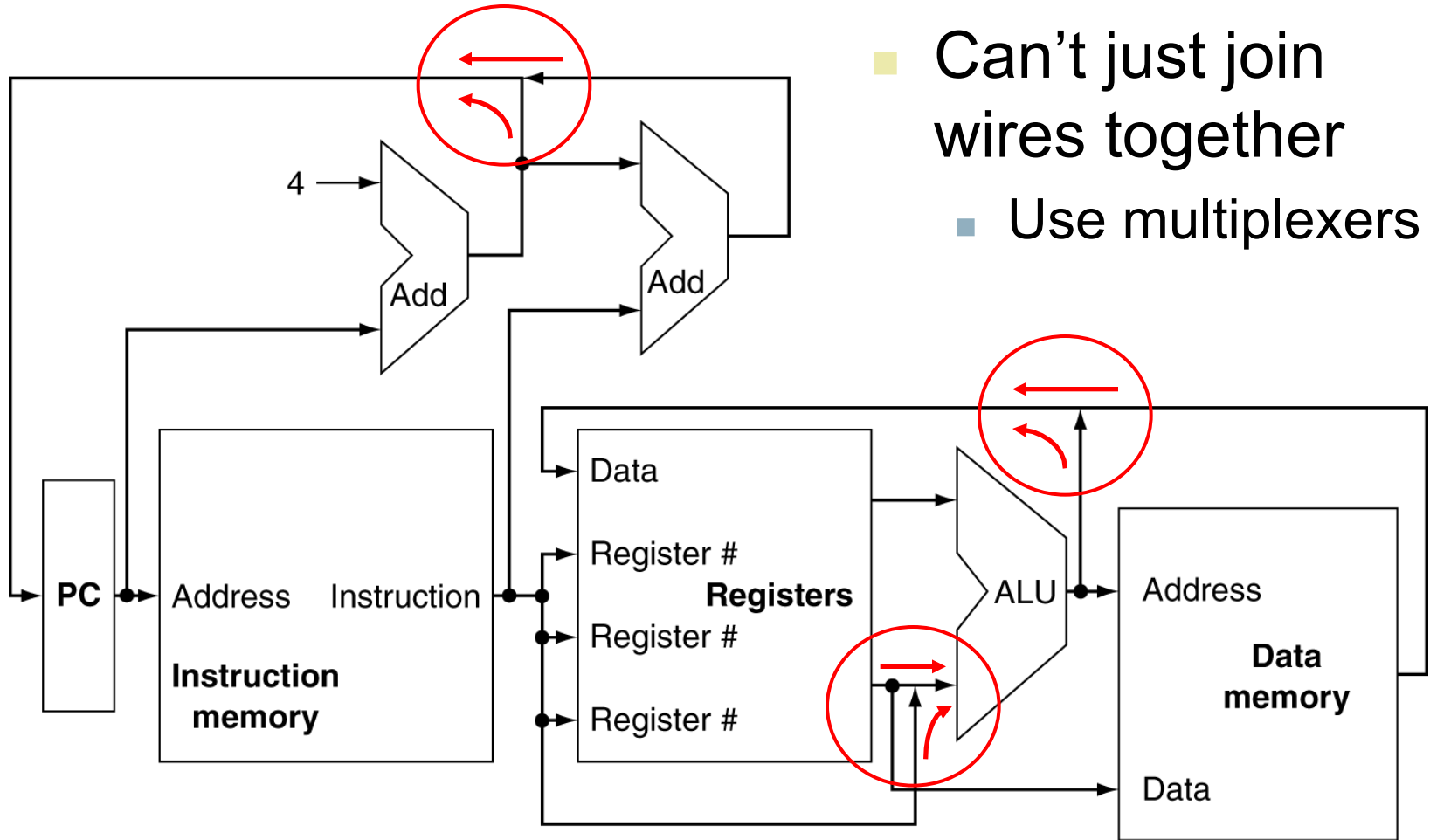
Control

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
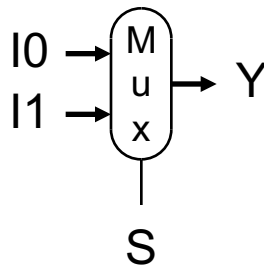
Combinational Elements
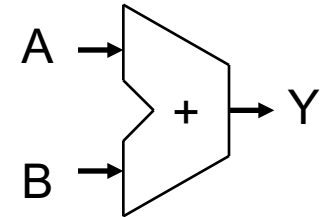
- ## AND-gate
  - Y = A & B

  A ──┐
      │)── Y
  B ──┘
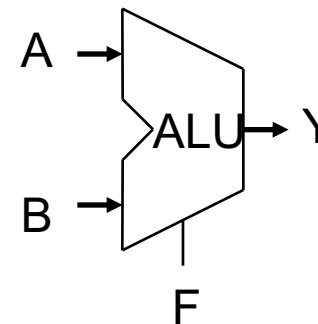
- ## Multiplexer
  - Y = S ? I1 : I0

  I0 →┐
      │ Mux │→ Y
  I1 →┘
      │
      S

- ## Adder
  - Y = A + B

  A →┐
     │ + │→ Y
  B →┘

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

  A →┐
     │ ALU │→ Y
  B →┘
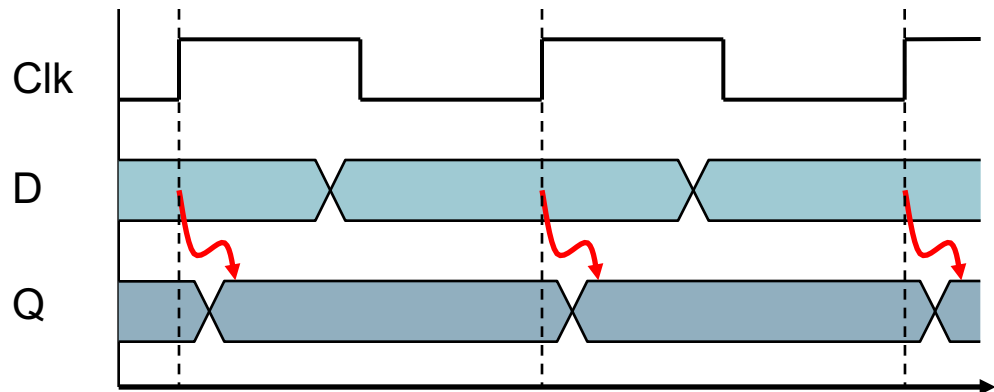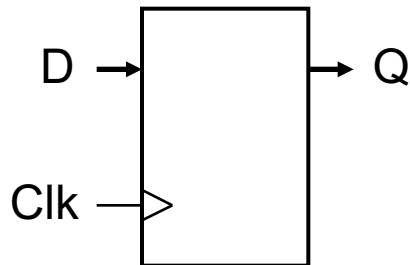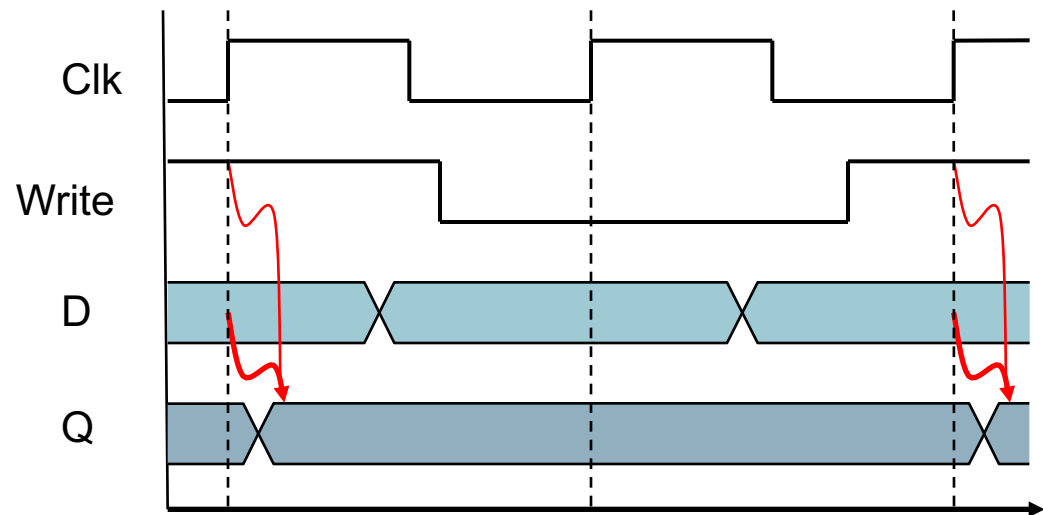   │
     F

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
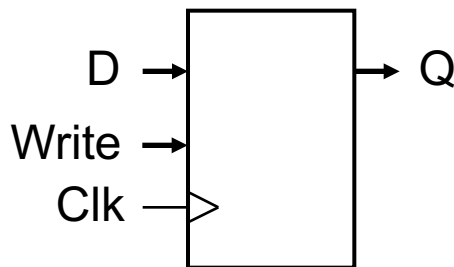
Clocking Methodology

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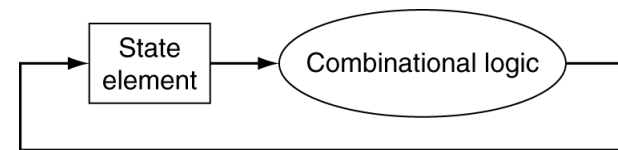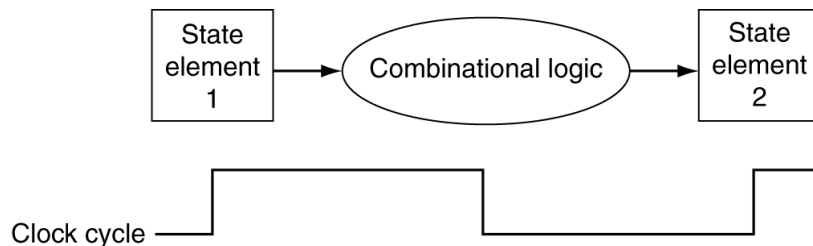
Building a Datapath

- 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
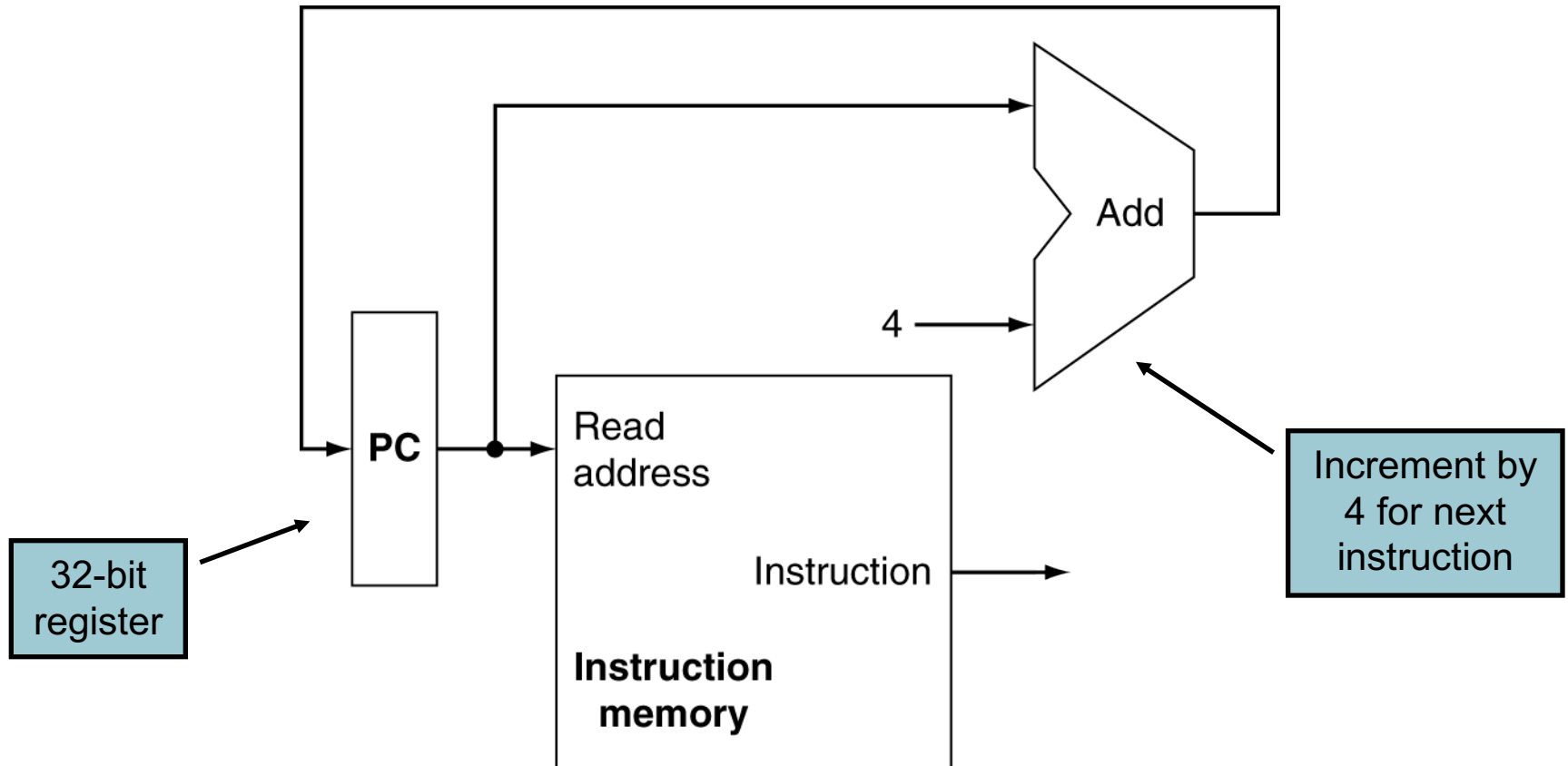
R-Format Instructions

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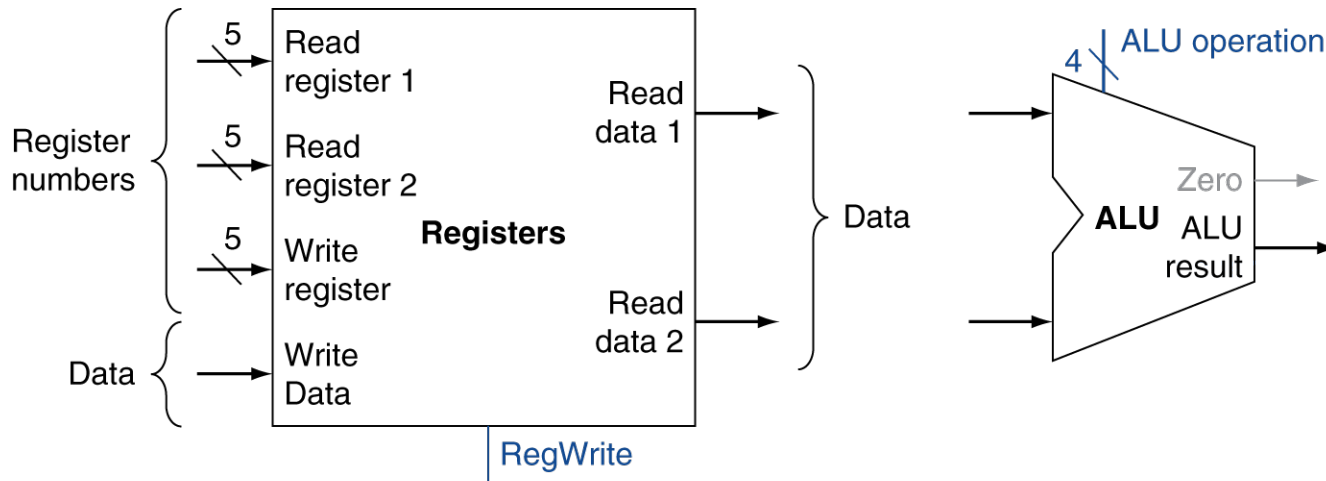
a. Registers

b. ALU

Load/Store Instructions

- 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
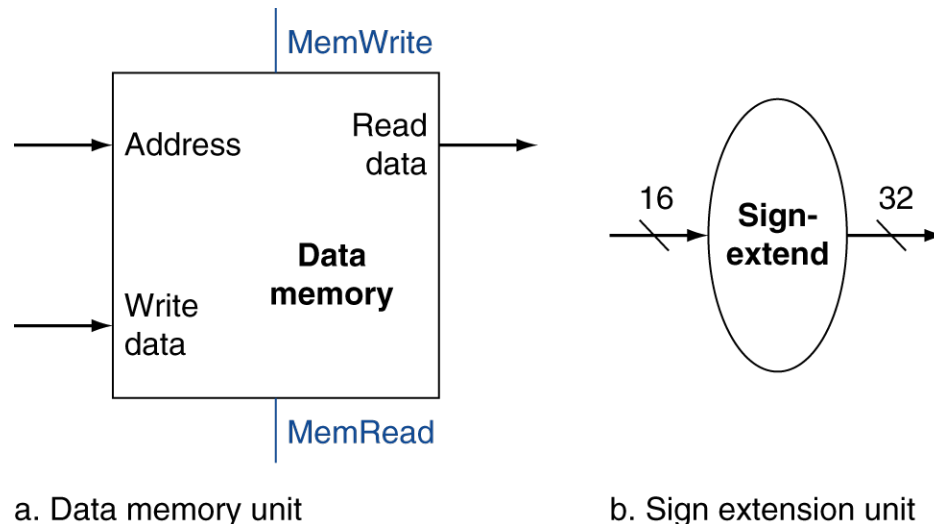
Branch Instructions

- 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
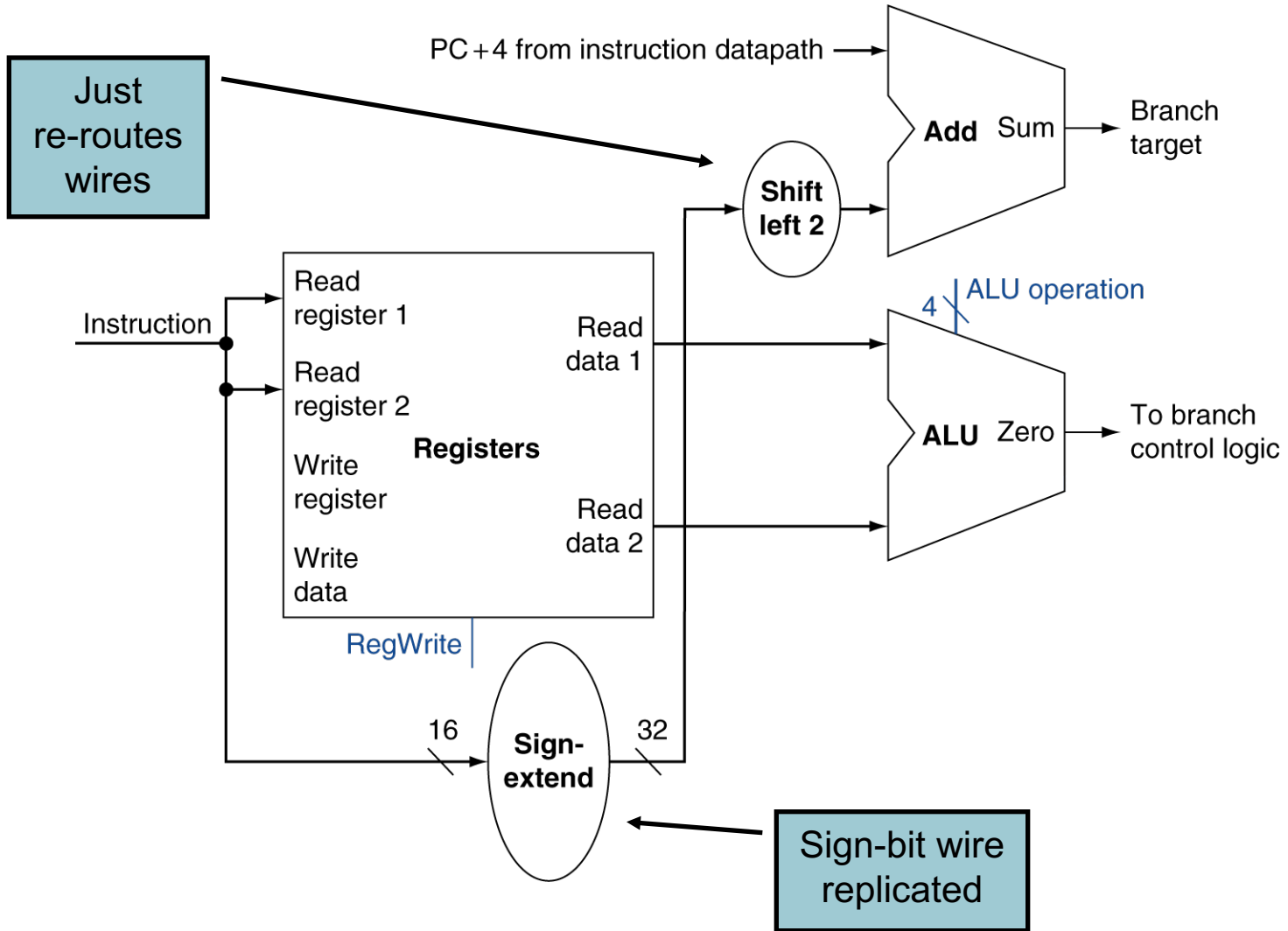
Composing the Elements

- 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
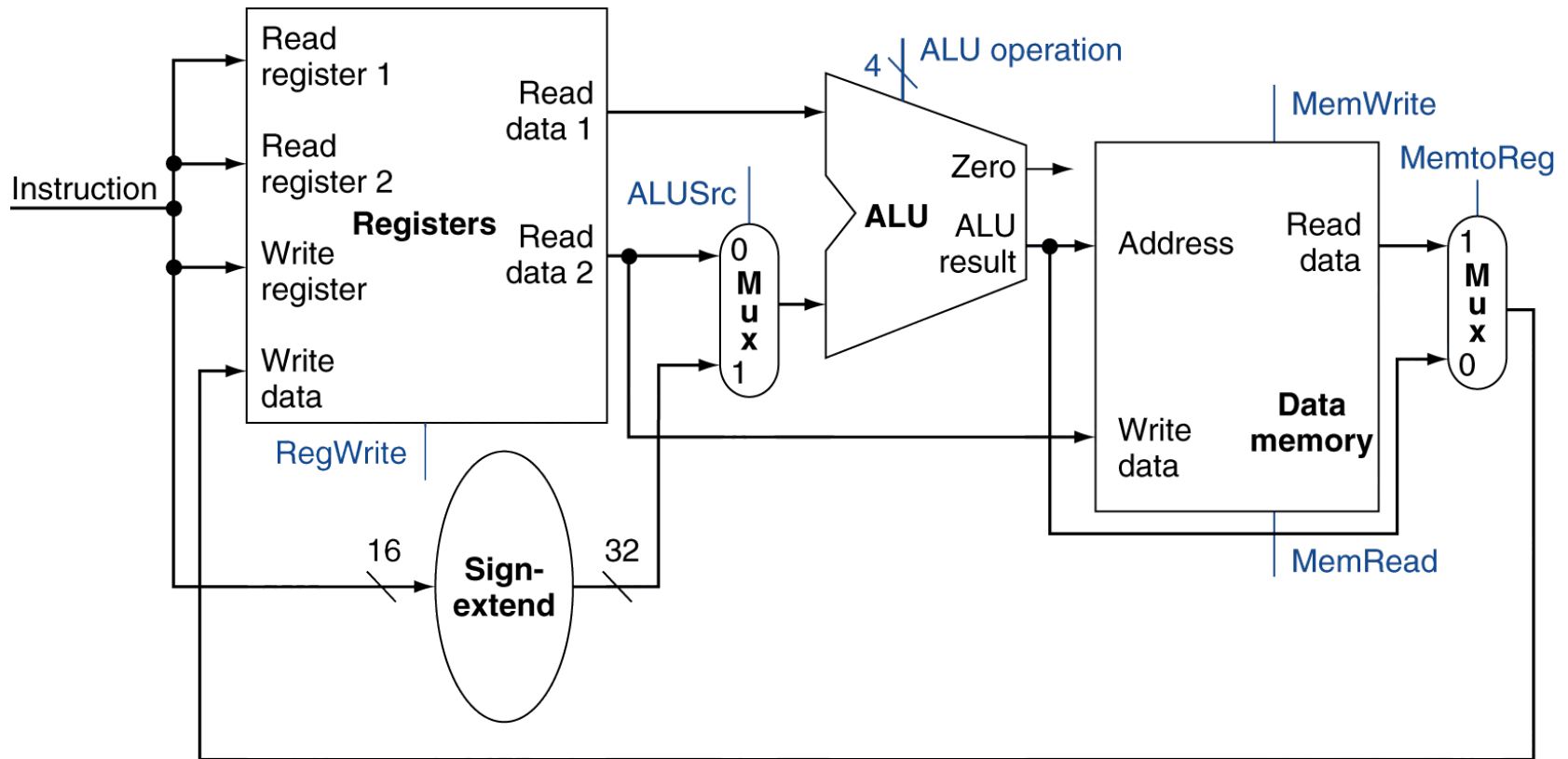
Full Datapath
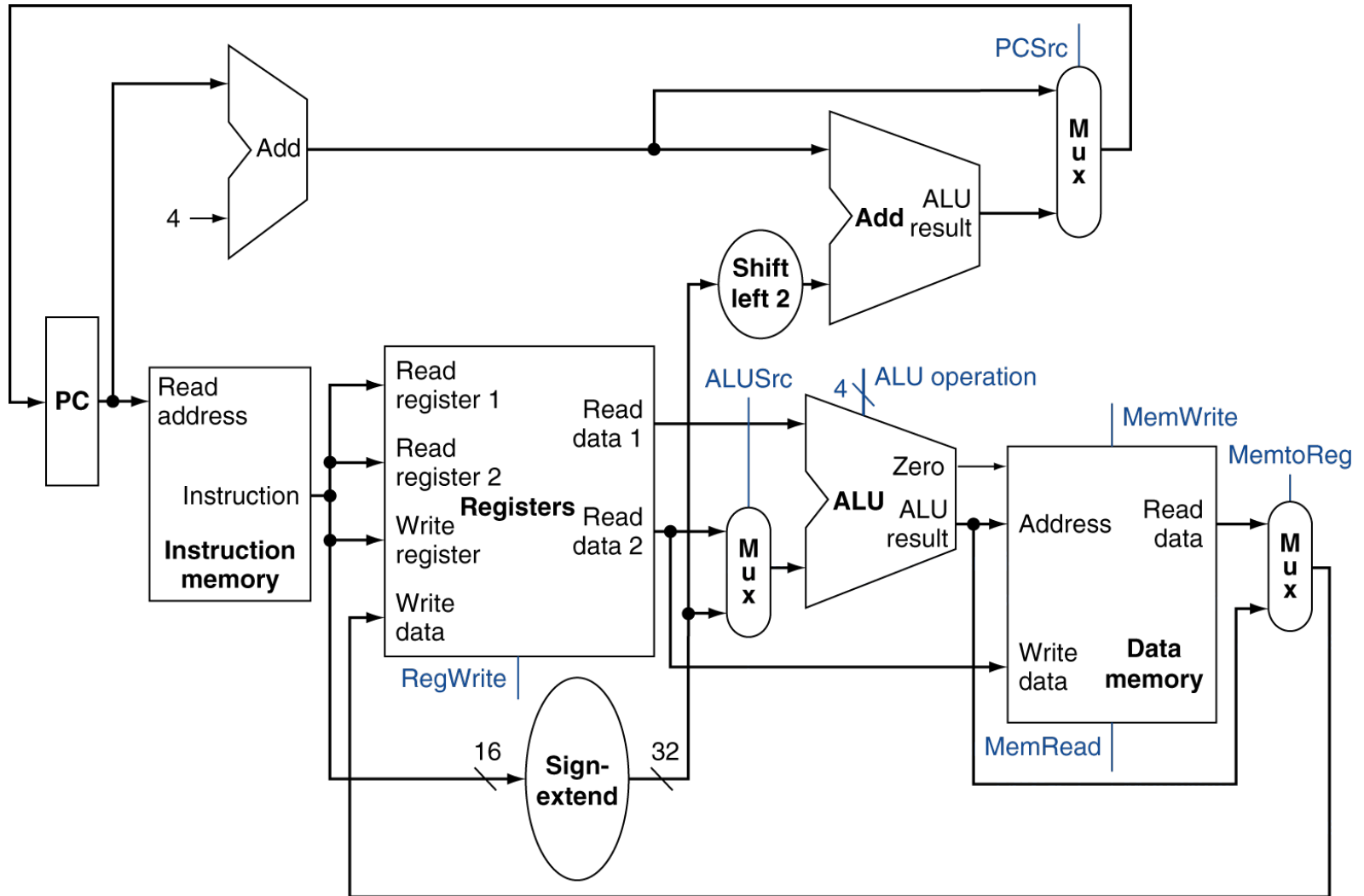
ALU Control

- ## ALU used for

    - ### Load/Store: 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

- Control signals derived from instruction

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

Load/Store	35 or 43	rs	rt	address
	31:26	25:21	20:16	15:0

Branch	4	rs	rt	address
	31:26	25:21	20:16	15:0

opcode

always read

read, except for load

write for R-type and load

sign-extend and add
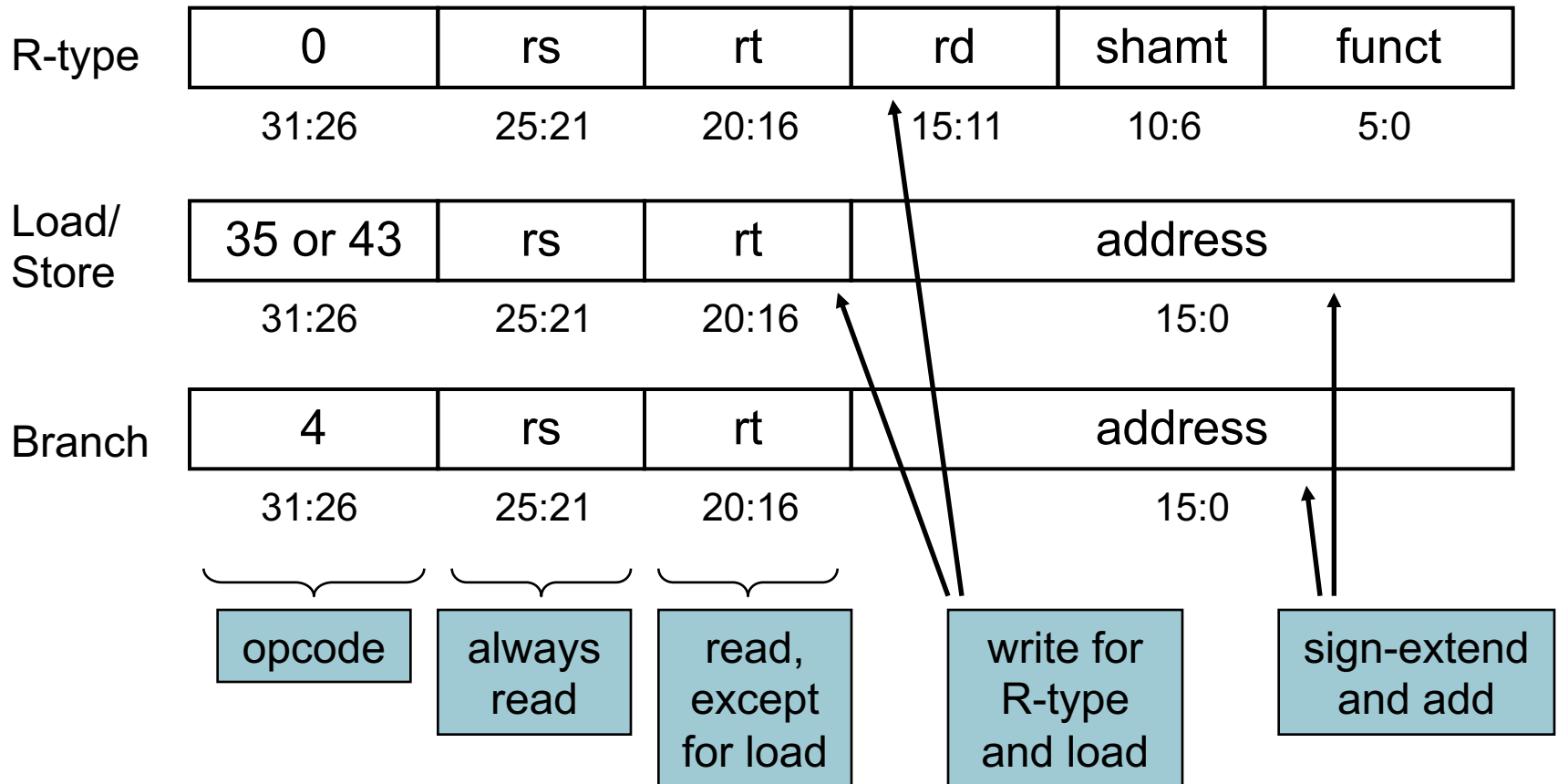
Datapath With Control
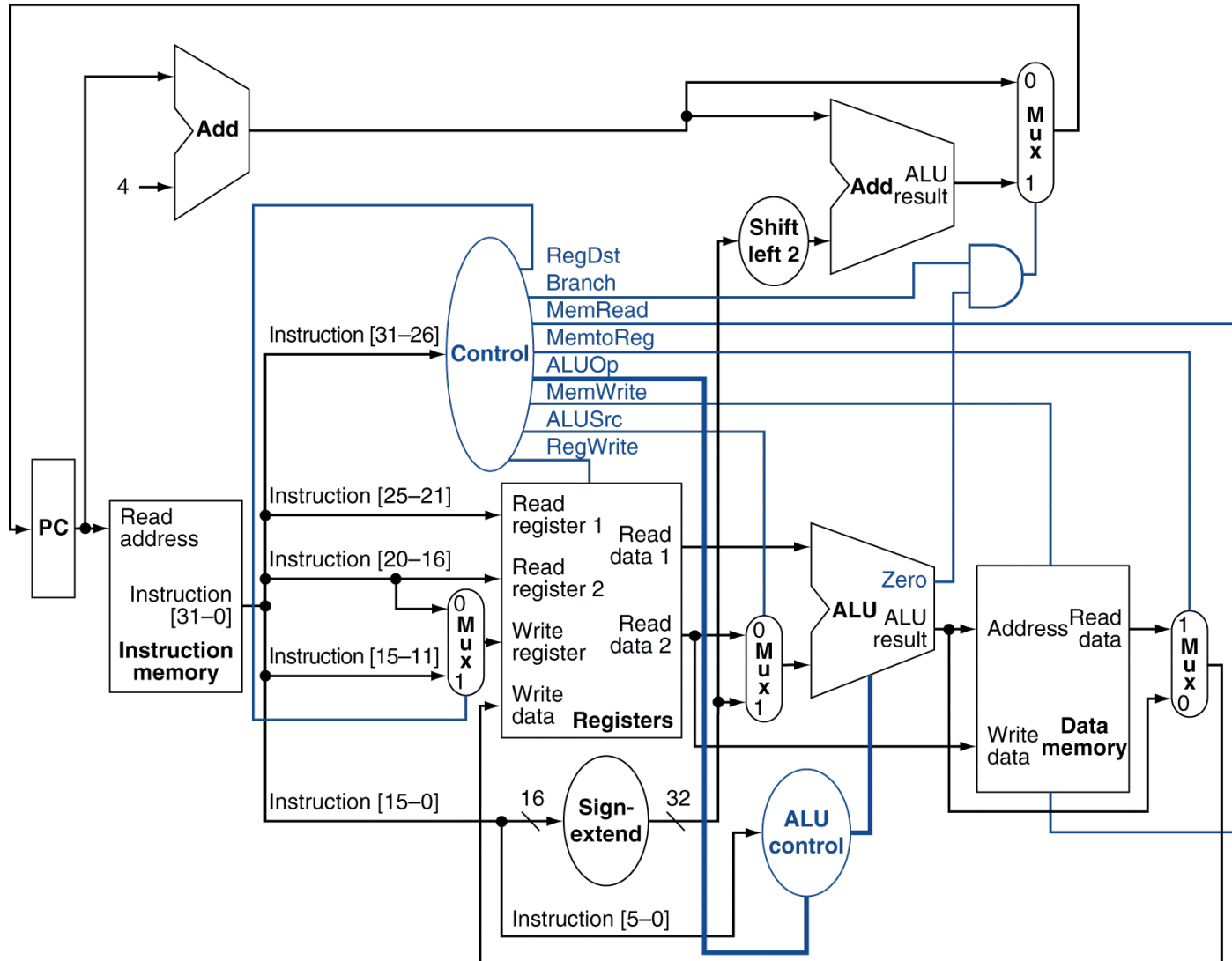
R-Type Instruction

Load Instruction

Branch-on-Equal Instruction
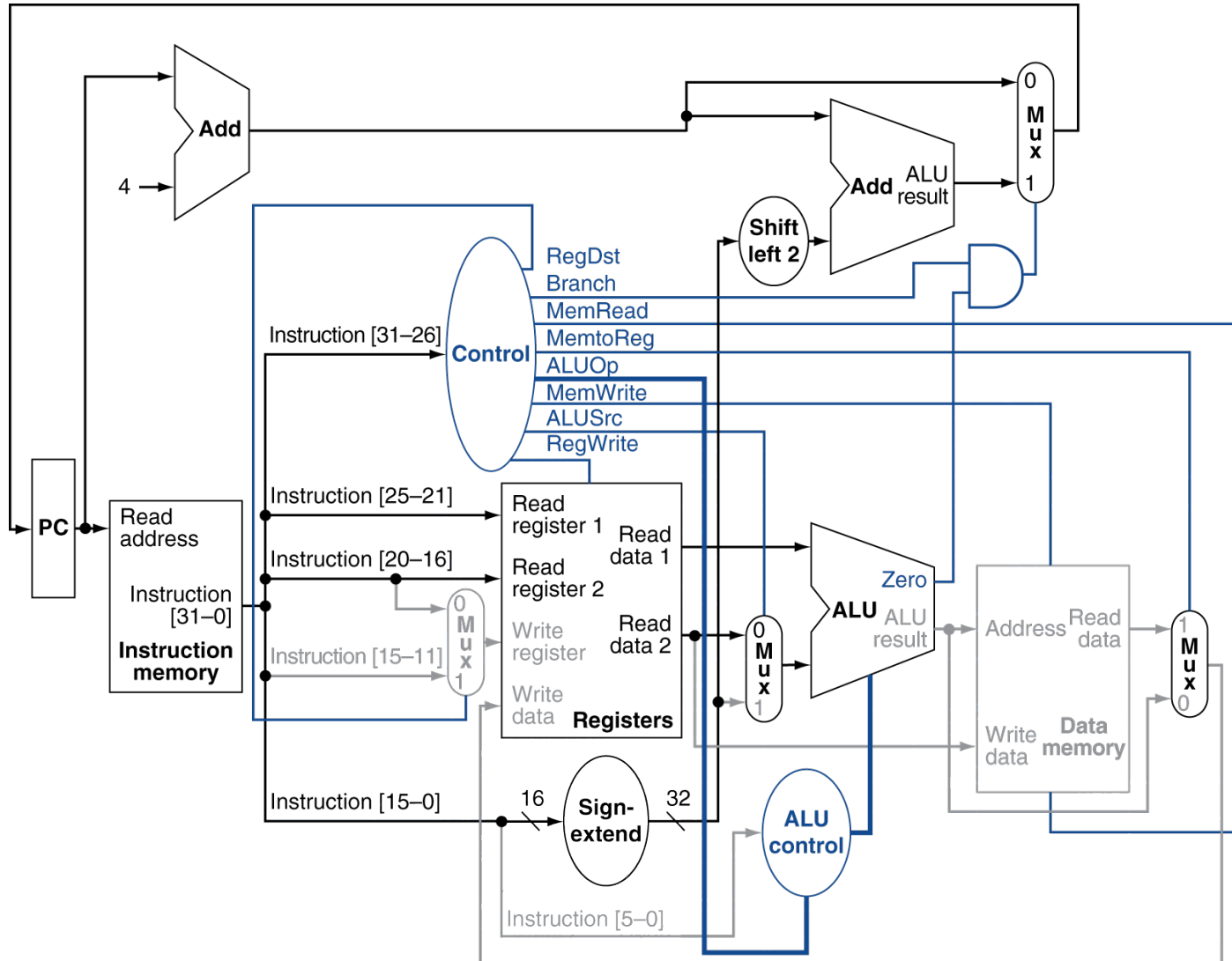
Implementing Jumps

Jump	2	address
	31:26	25:0
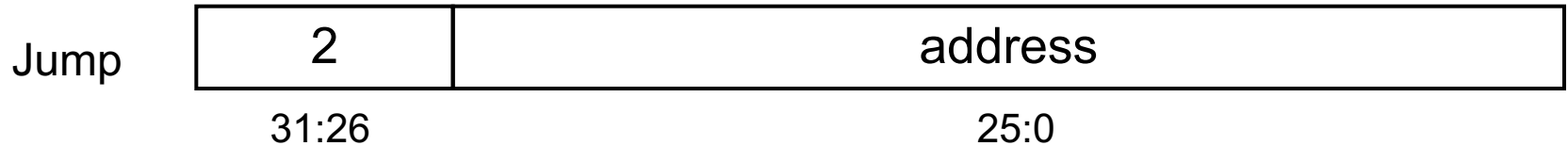
- ## Jump uses word address

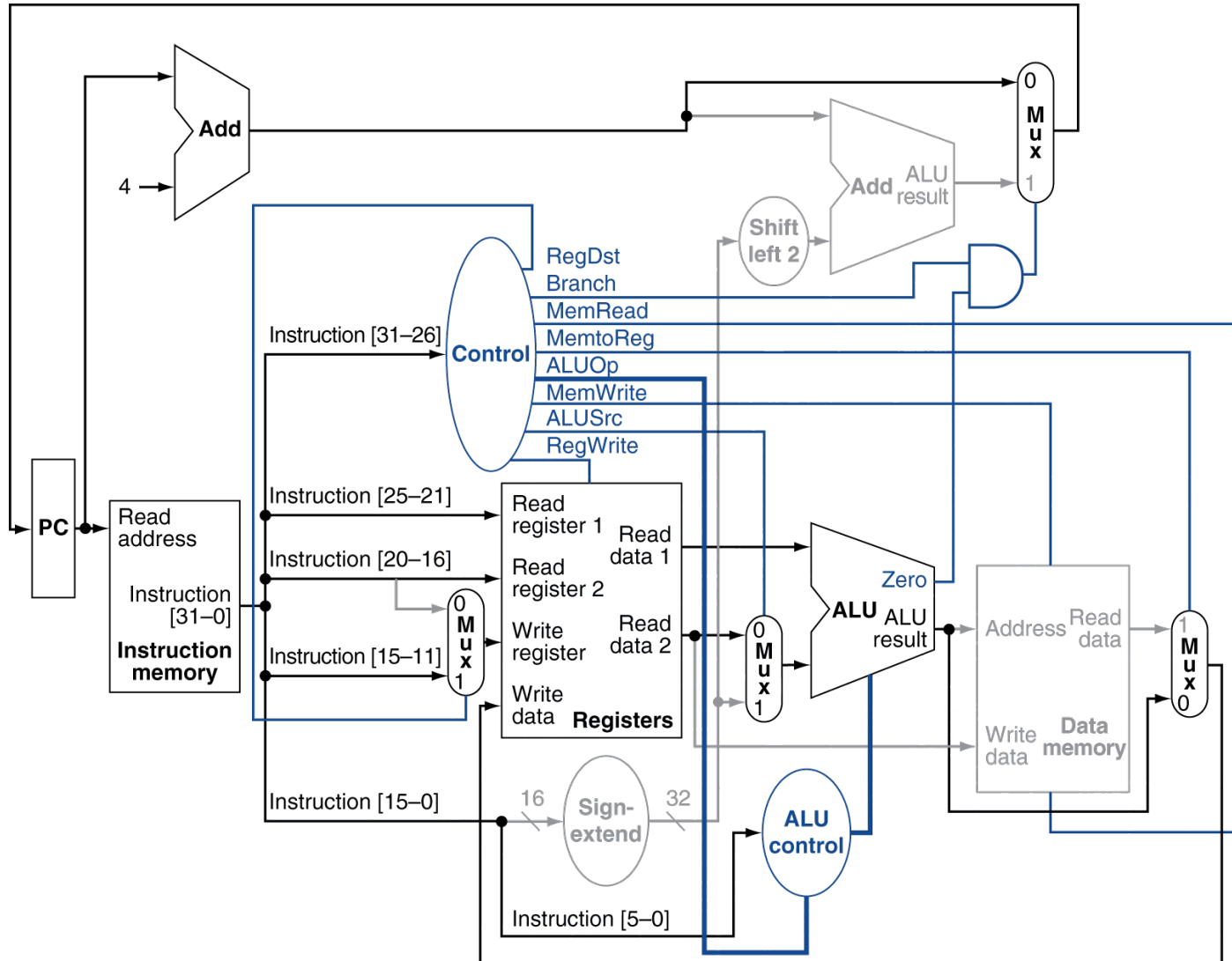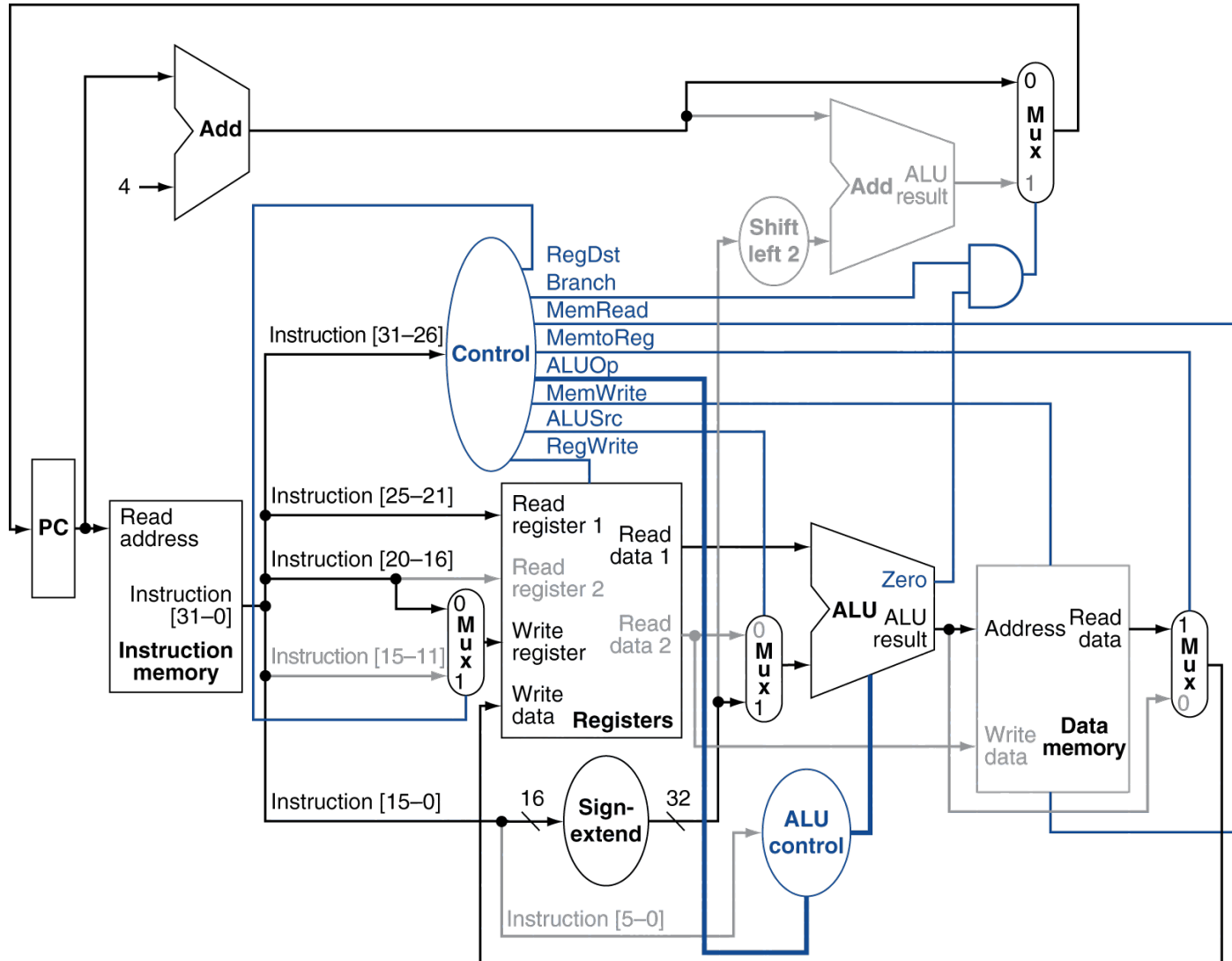- ## Update PC with concatenation of

  - ### Top 4 bits of old PC

  - ### 26-bit jump address

  - ### 00

- ## Need an extra control signal decoded from opcode

Datapath With Jumps Added
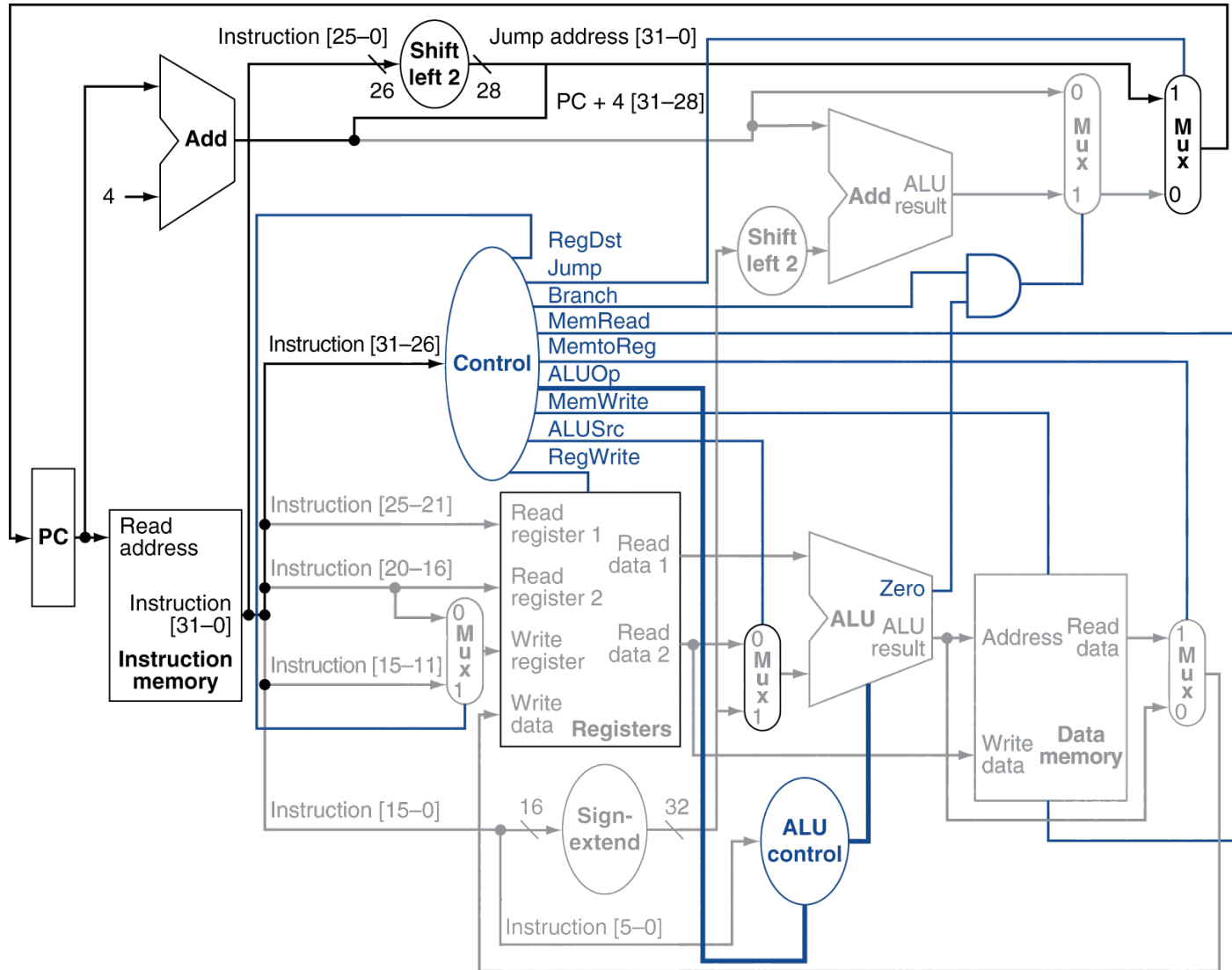
Performance Issues

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

Pipelining Analogy

- ## Pipelined laundry: overlapping execution
  - ### Parallelism improves performance
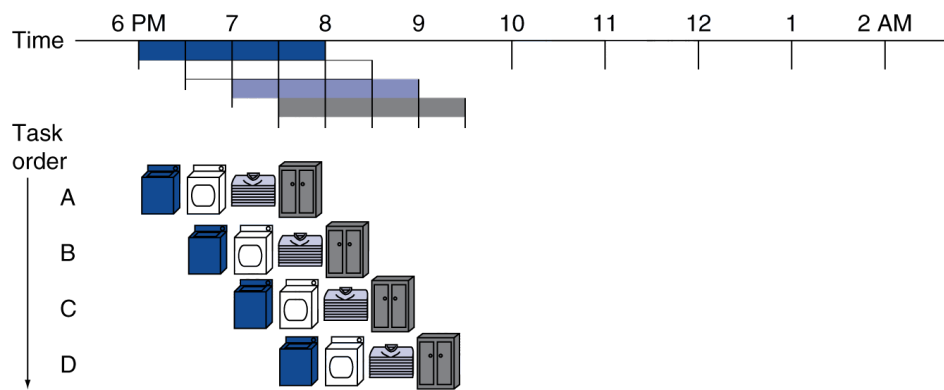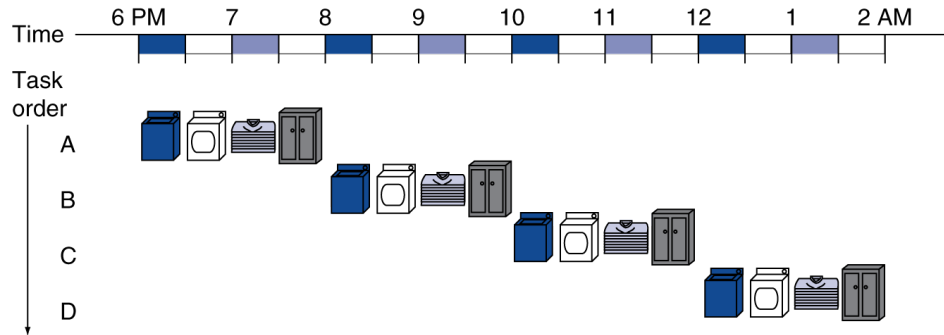
- ## Four loads:
  - ### Speedup = 8/3.5 = 2.3

- ## Non-stop:
  - ### Speedup ≈ number of stages

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

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
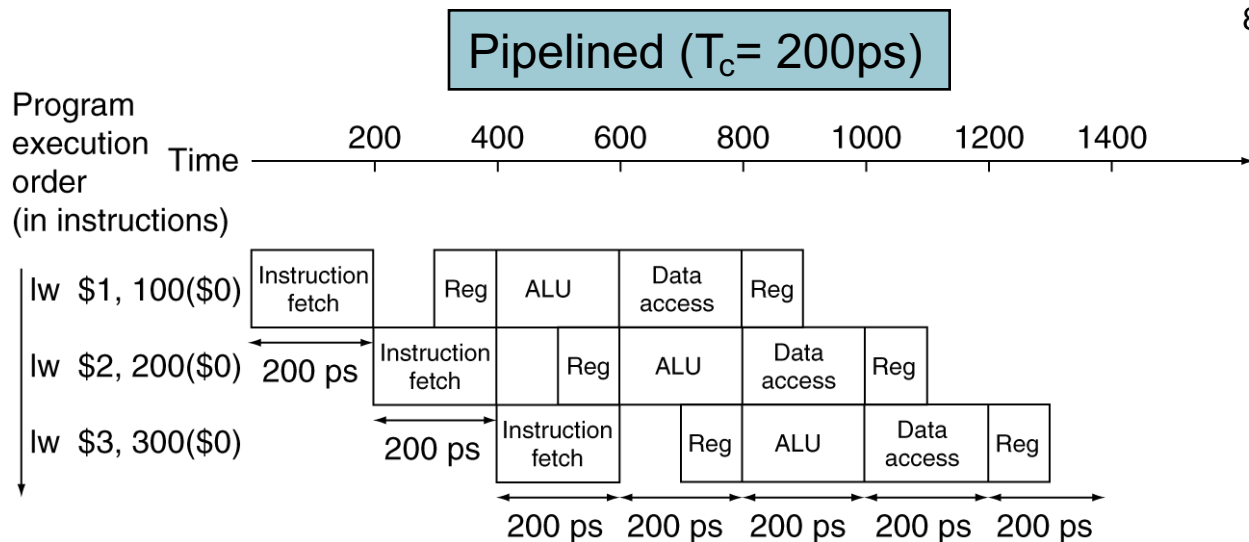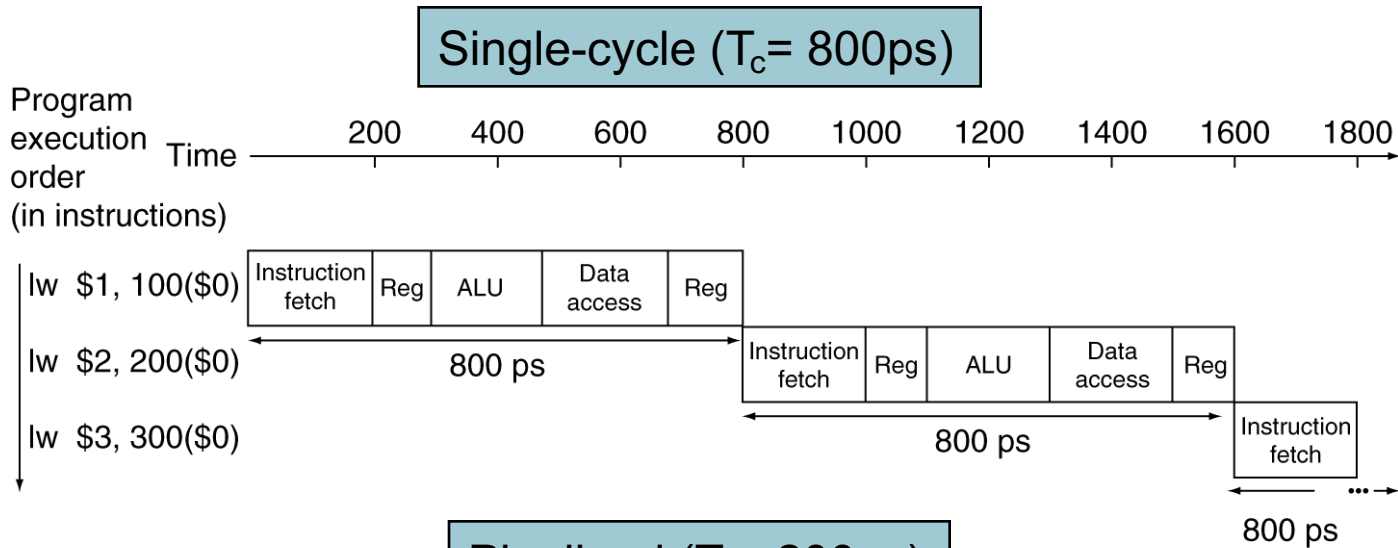
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$

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

  - In our example, 800/5 = 160ps

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipeline Speedup

- Our example with 3 instructions
  - $T_{pipelined}$ = 1400ps
  - $T_{nopipelined}$ = 2400ps
  - Speedup = 2400/1400 = 1.714

- Add 1,000,000 instructions
  - $T_{pipelined}$ = 1,000,000 * 200 + 1400
  - $T_{nopipelined}$ = 1,000,000 * 800 + 2400
  - Speedup = 800,002,400/200,001,400 = 4

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

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

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - ```
    add   $s0, $t0, $t1
    sub   $t2, $s0, $t3
    ```
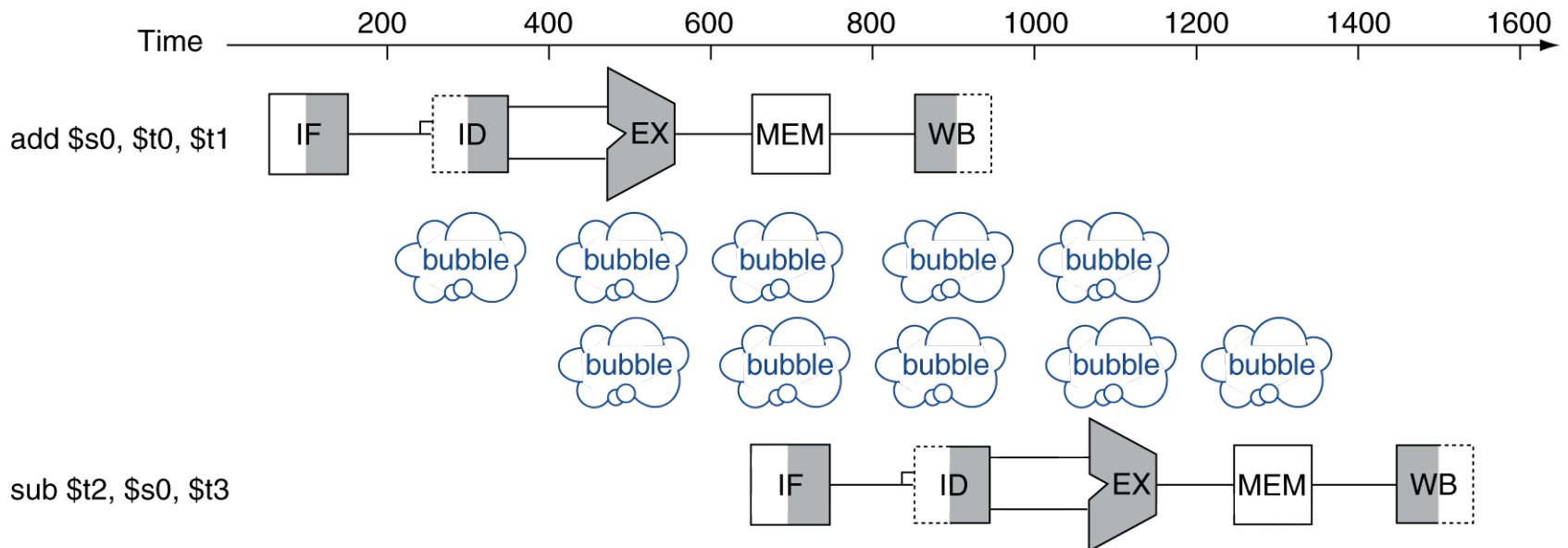
Forwarding (aka Bypassing)

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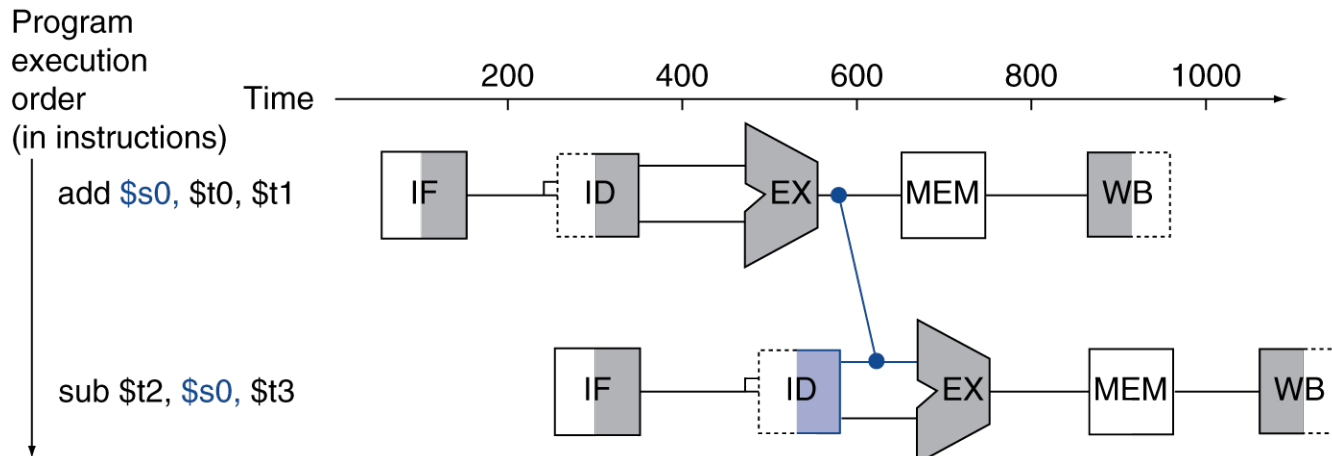
Load-Use Data Hazard

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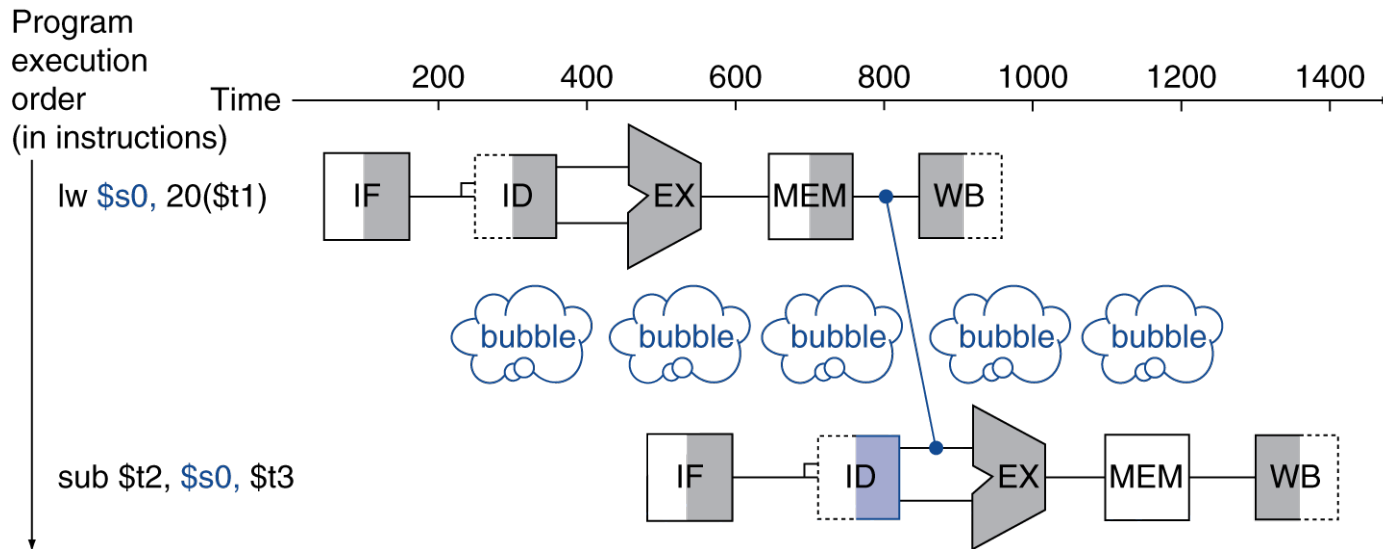
Code Scheduling to Avoid Stalls

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

- C code for `A = B + E; C = B + F;`
  `A(12), B(0), E(4), C(16), F(8)`

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

stall → add $t3, $t1, $t2

stall → add $t5, $t1, $t4

13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles
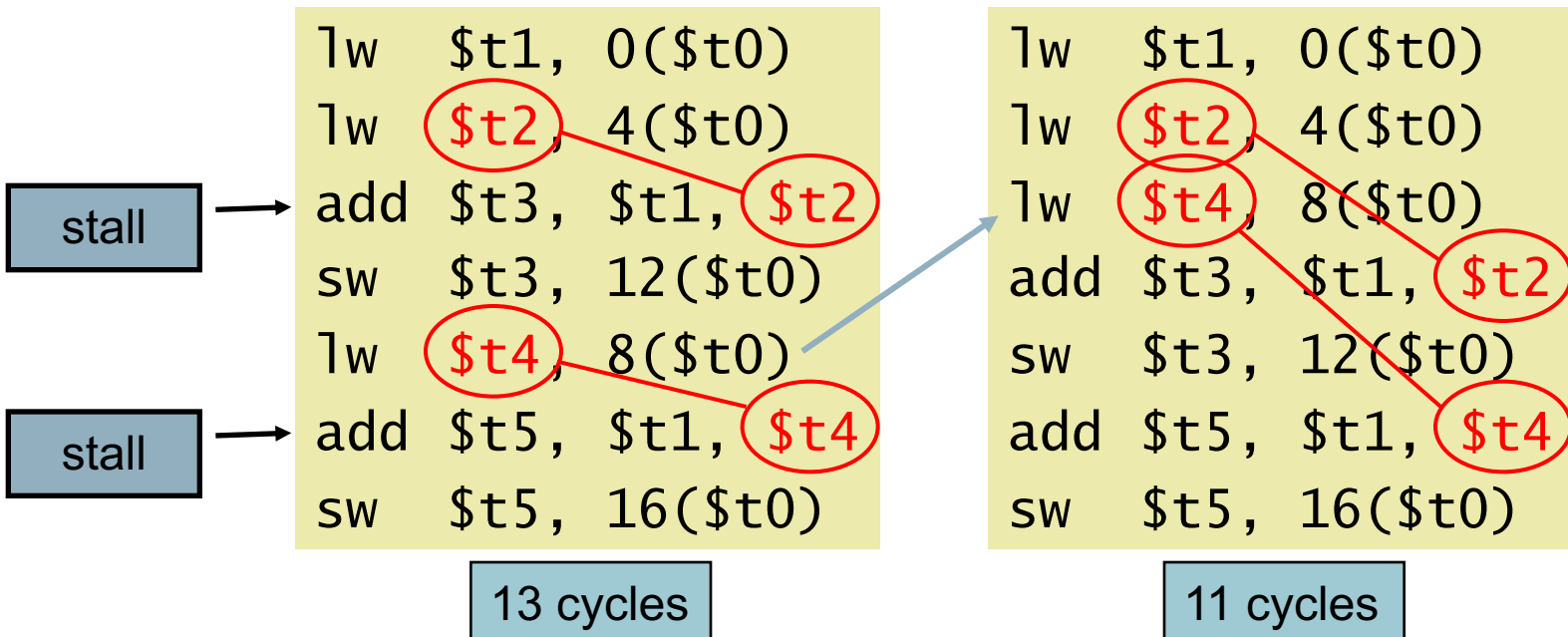
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
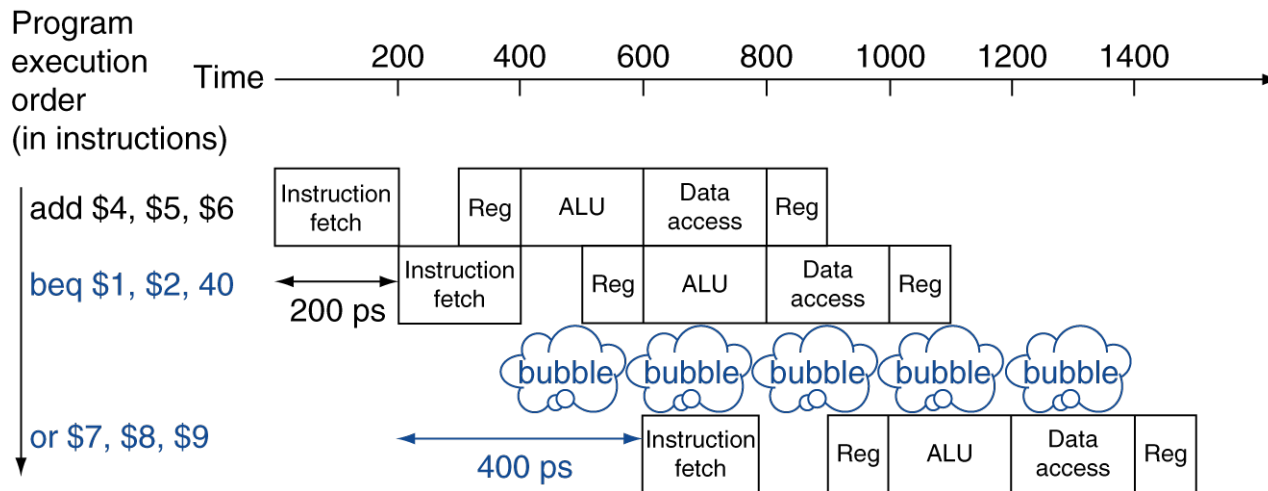
Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct

Program execution order (in instructions)

Time — 200 400 600 800 1000 1200 1400

add $4, $5, $6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0) — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

Prediction incorrect

Program execution order (in instructions)

Time — 200 400 600 800 1000 1200 1400

add $4, $5, $6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

bubble bubble bubble bubble bubble

or $7, $8, $9 — 400 ps — Instruction fetch | Reg | ALU | Data access | Reg

More-Realistic Branch Prediction

- ## Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- ## Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath
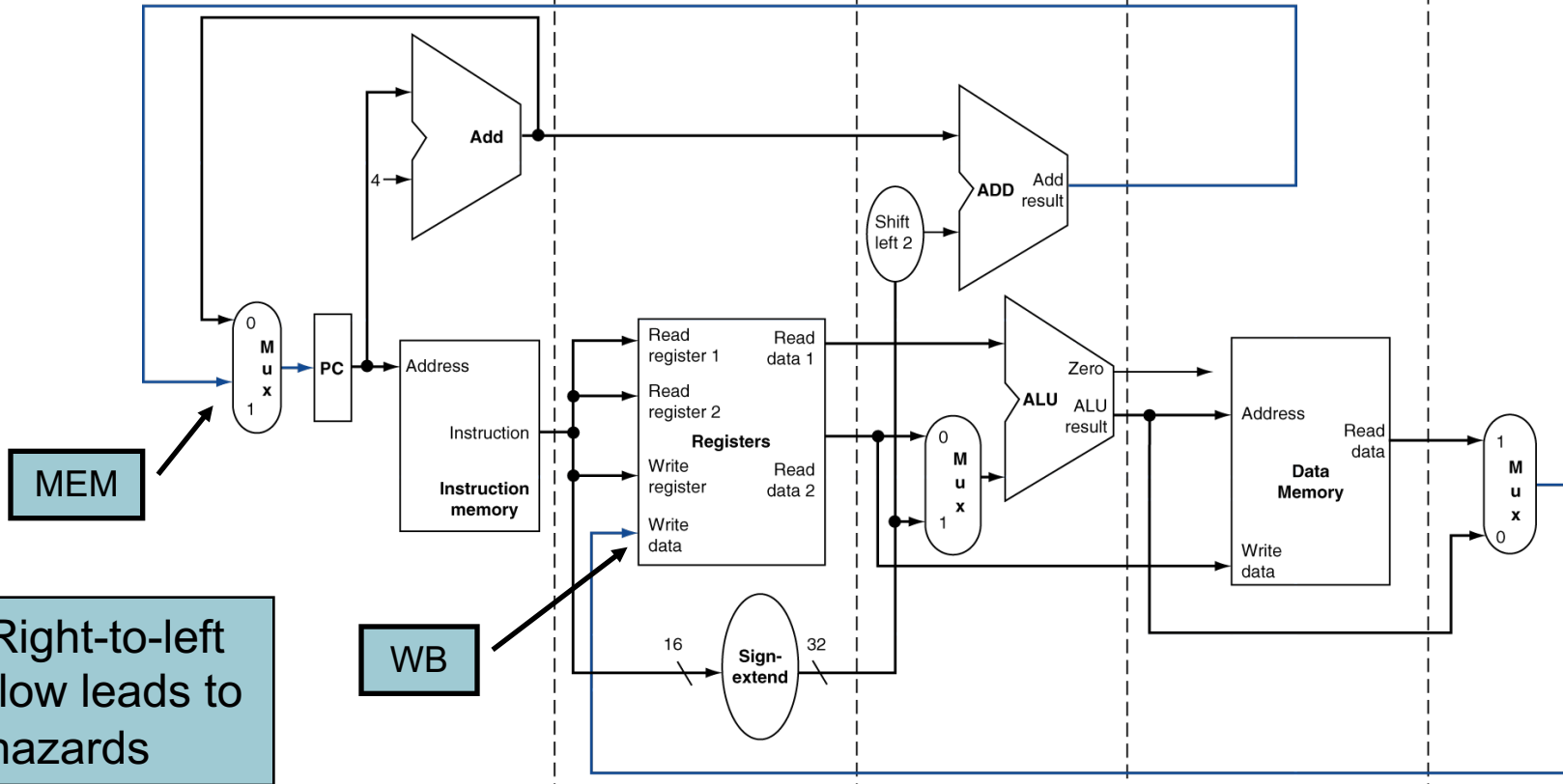
Pipeline registers

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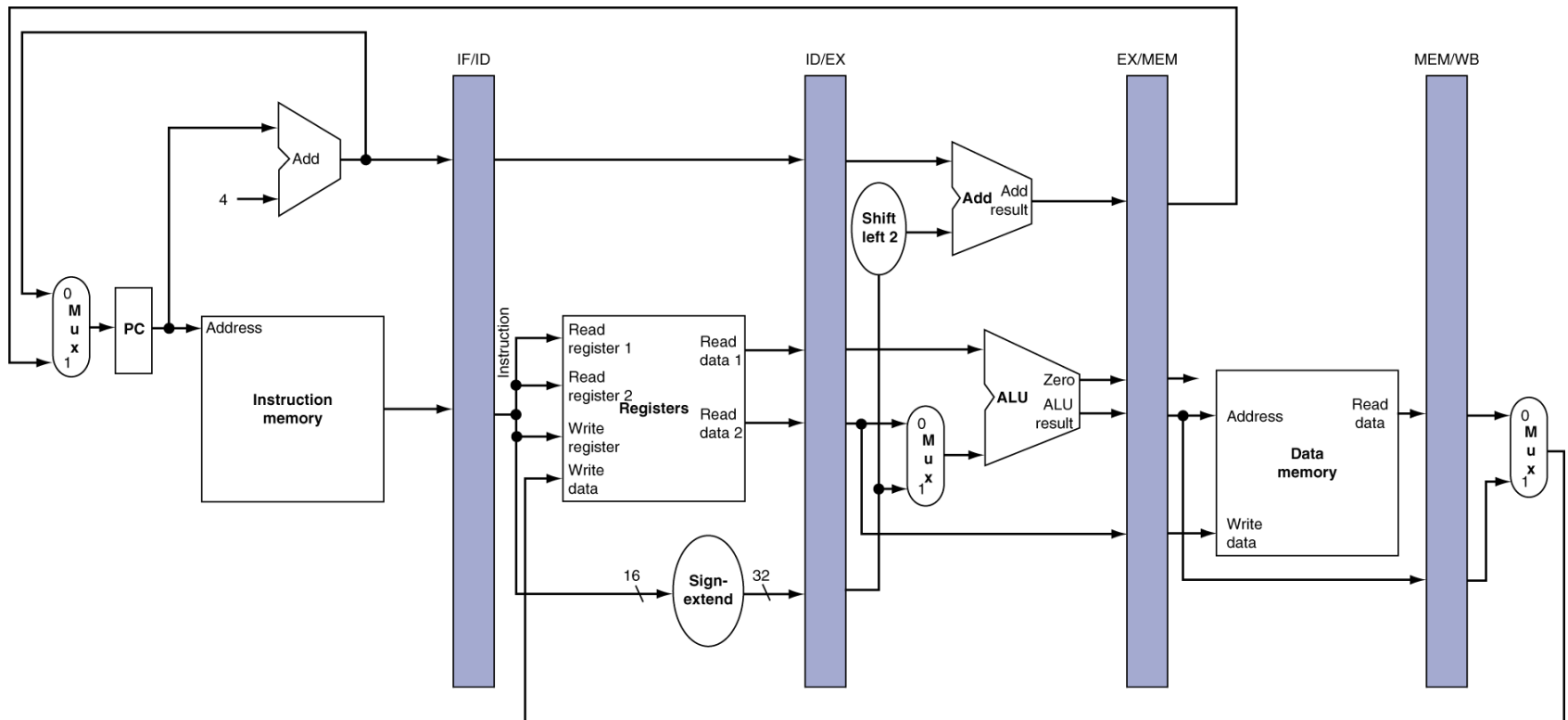
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

IF for Load, Store, …
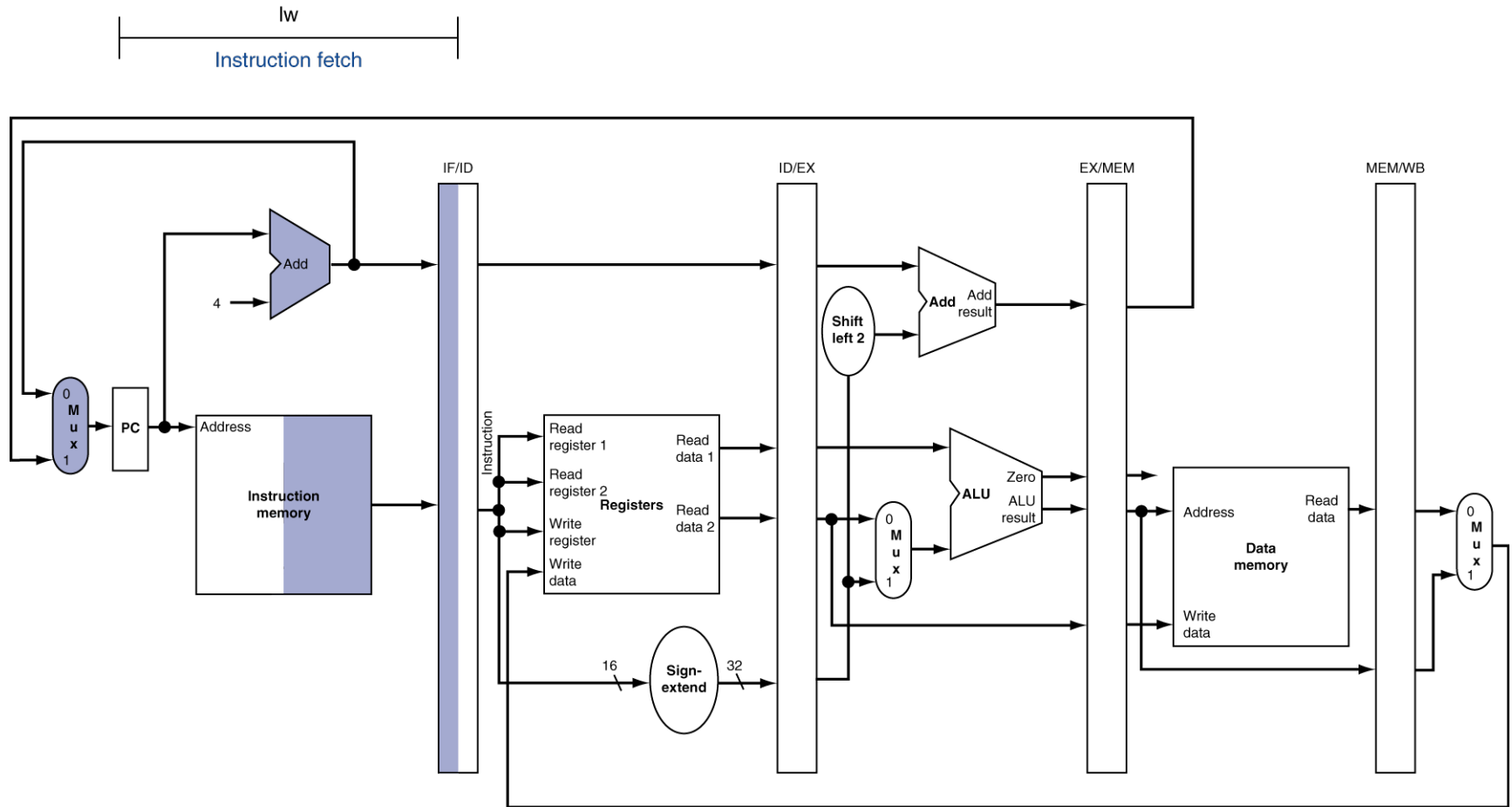
ID for Load, Store, …
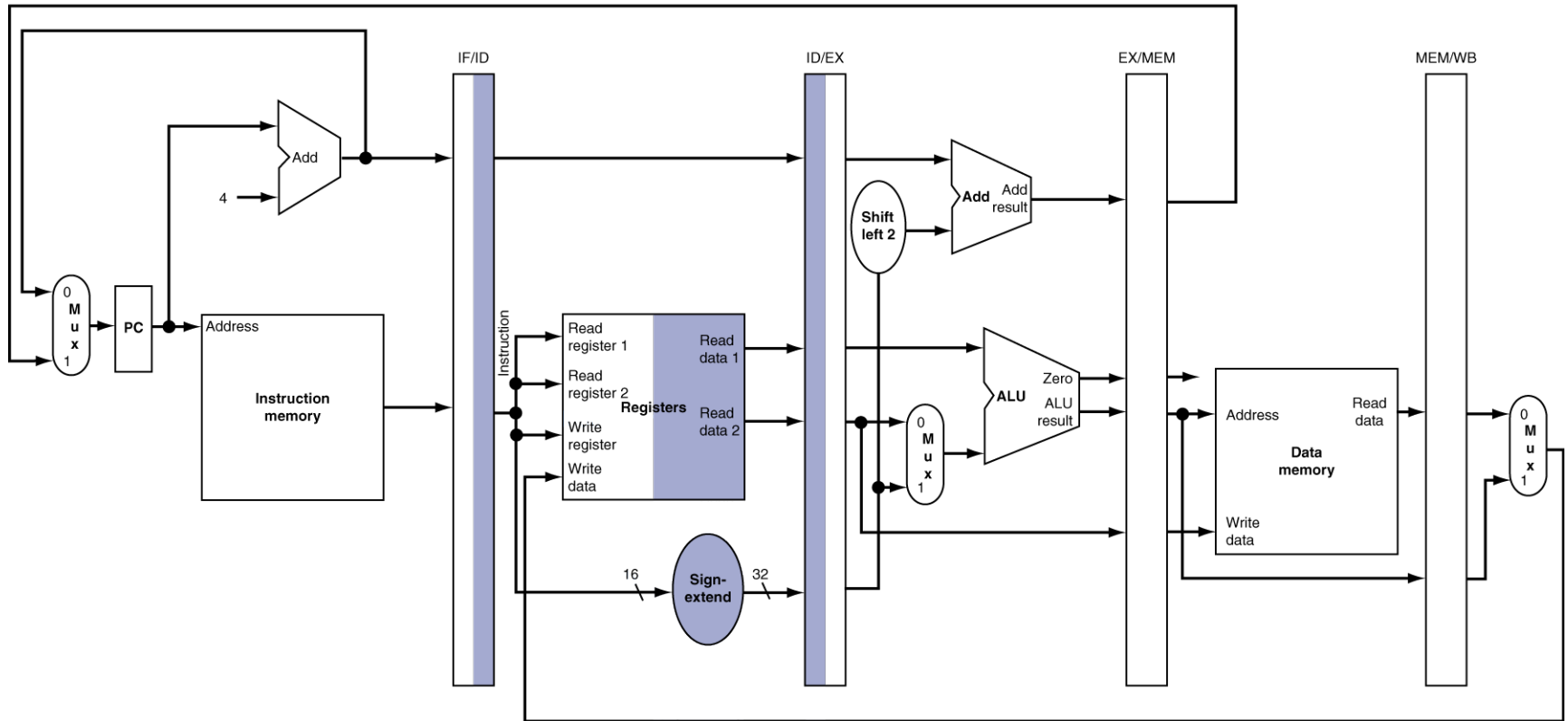
EX for Load

MEM for Load

WB for Load

Corrected Datapath for Load

EX for Store

MEM for Store
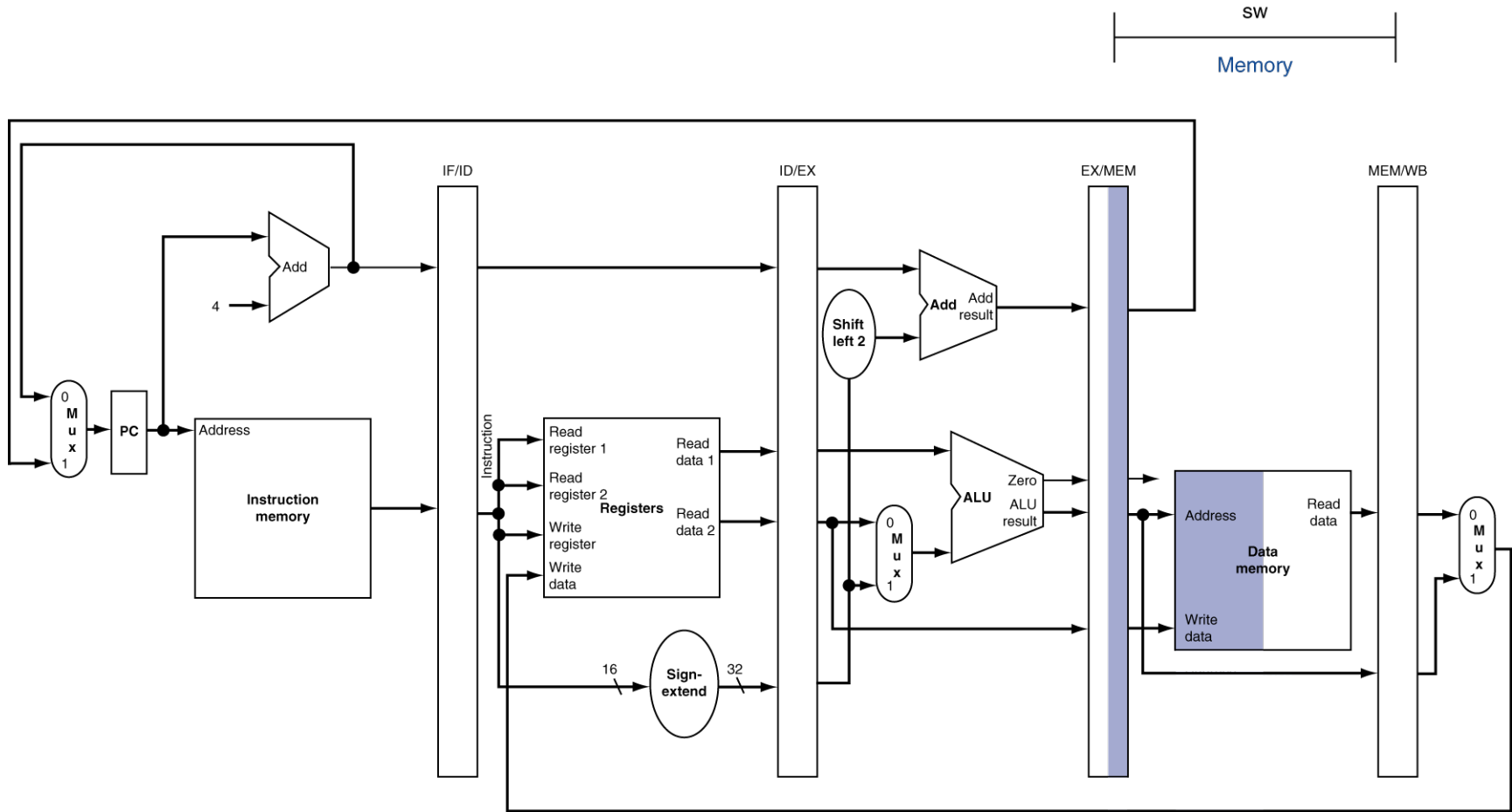
WB for Store
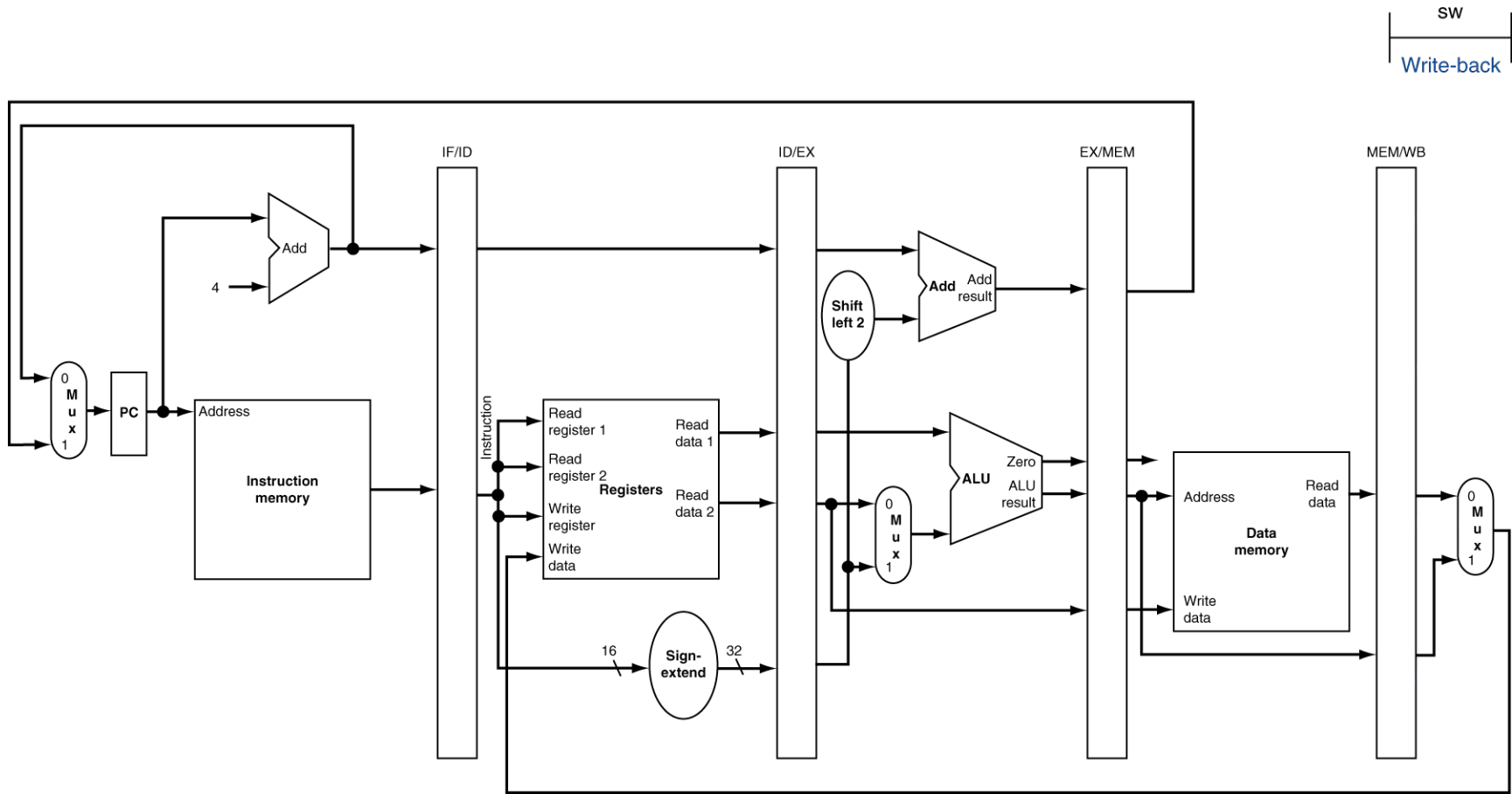
Multi-Cycle Pipeline Diagram

■ Form showing resource usage
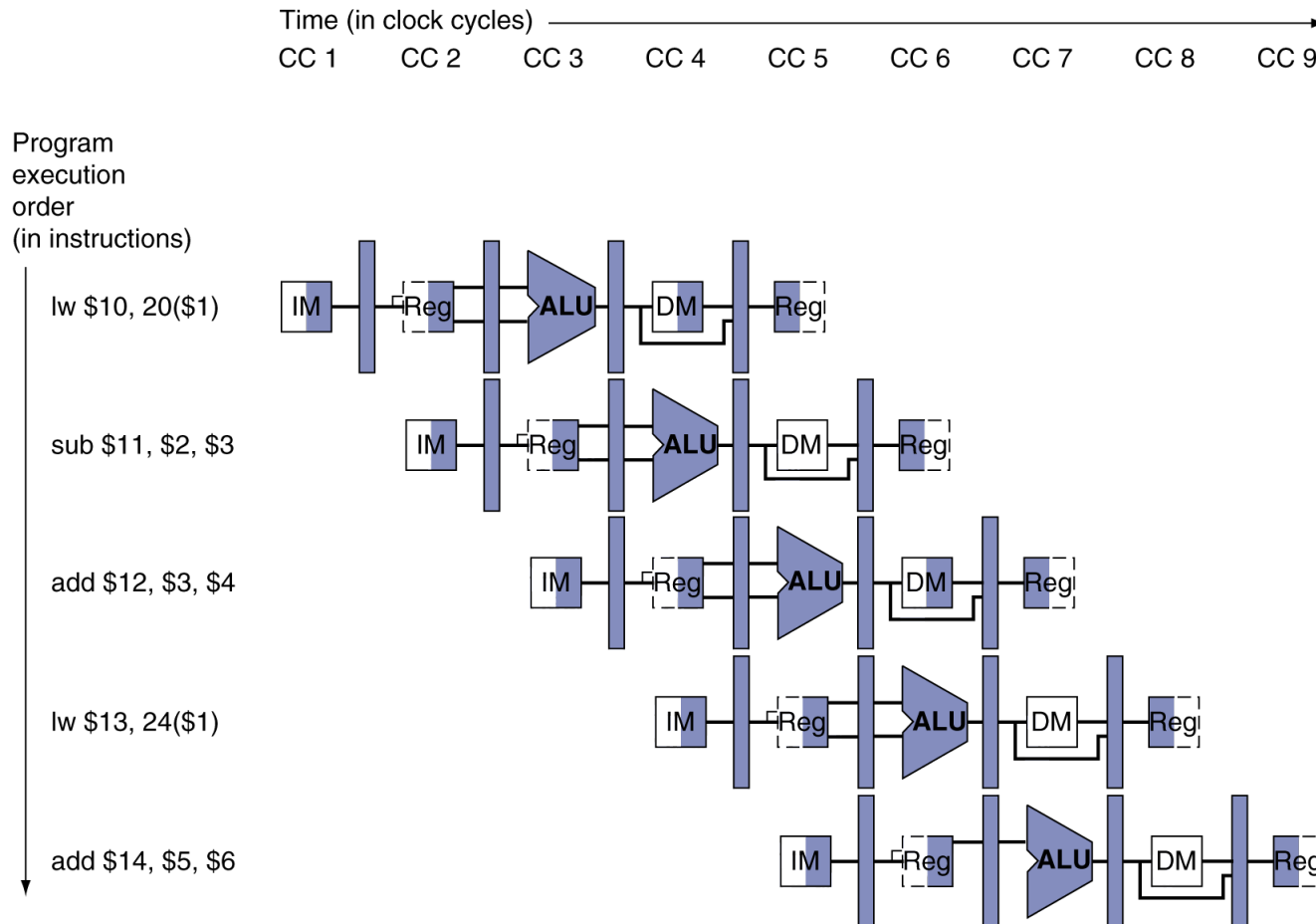
Multi-Cycle Pipeline Diagram

- Traditional form

Time (in clock cycles)

Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Single-Cycle Pipeline Diagram

■ State of pipeline in a given cycle
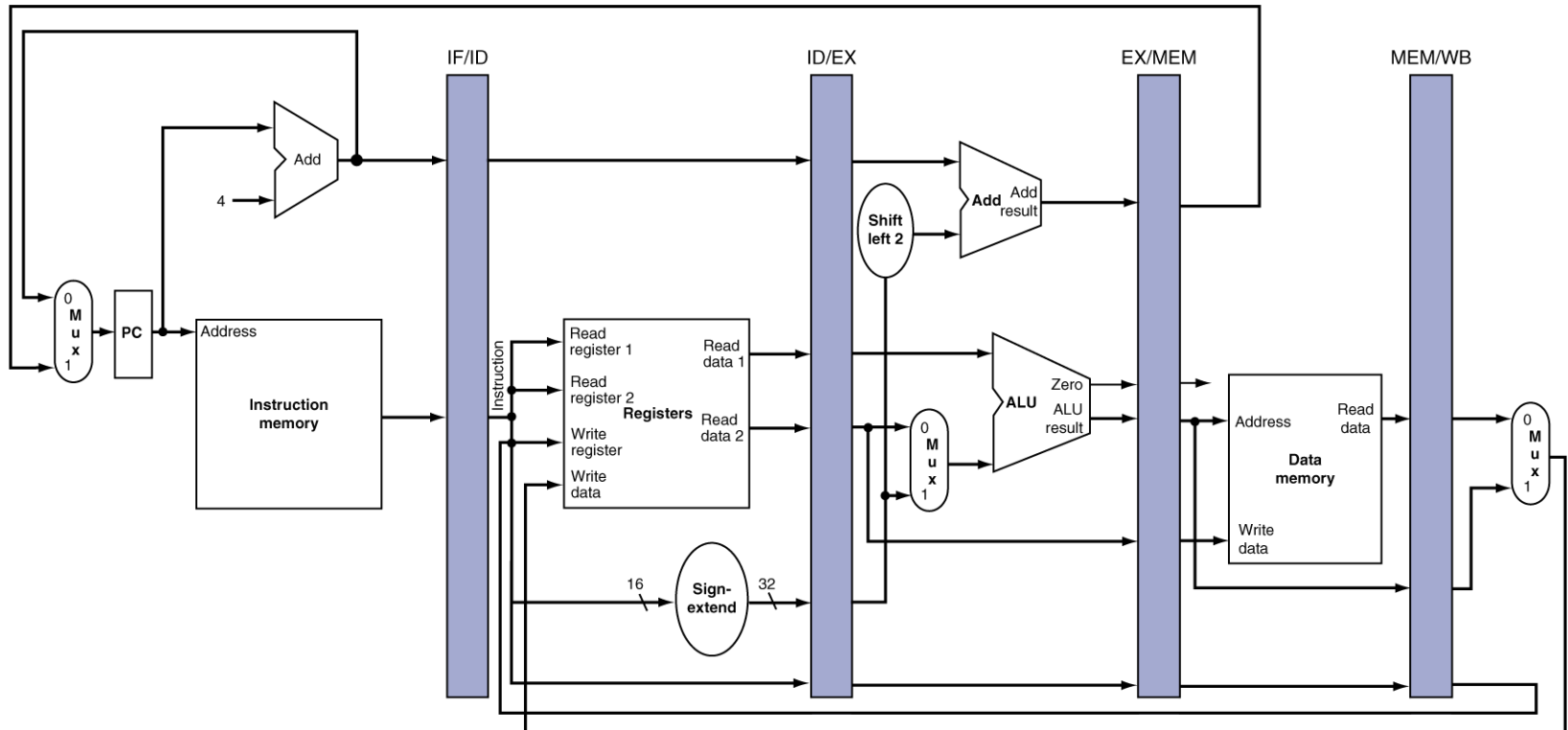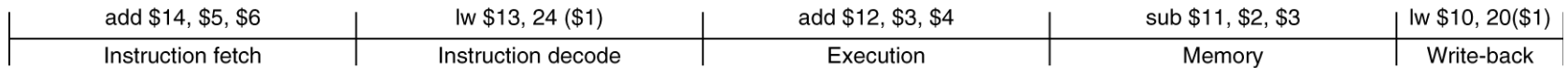
Pipelined Control (Simplified)
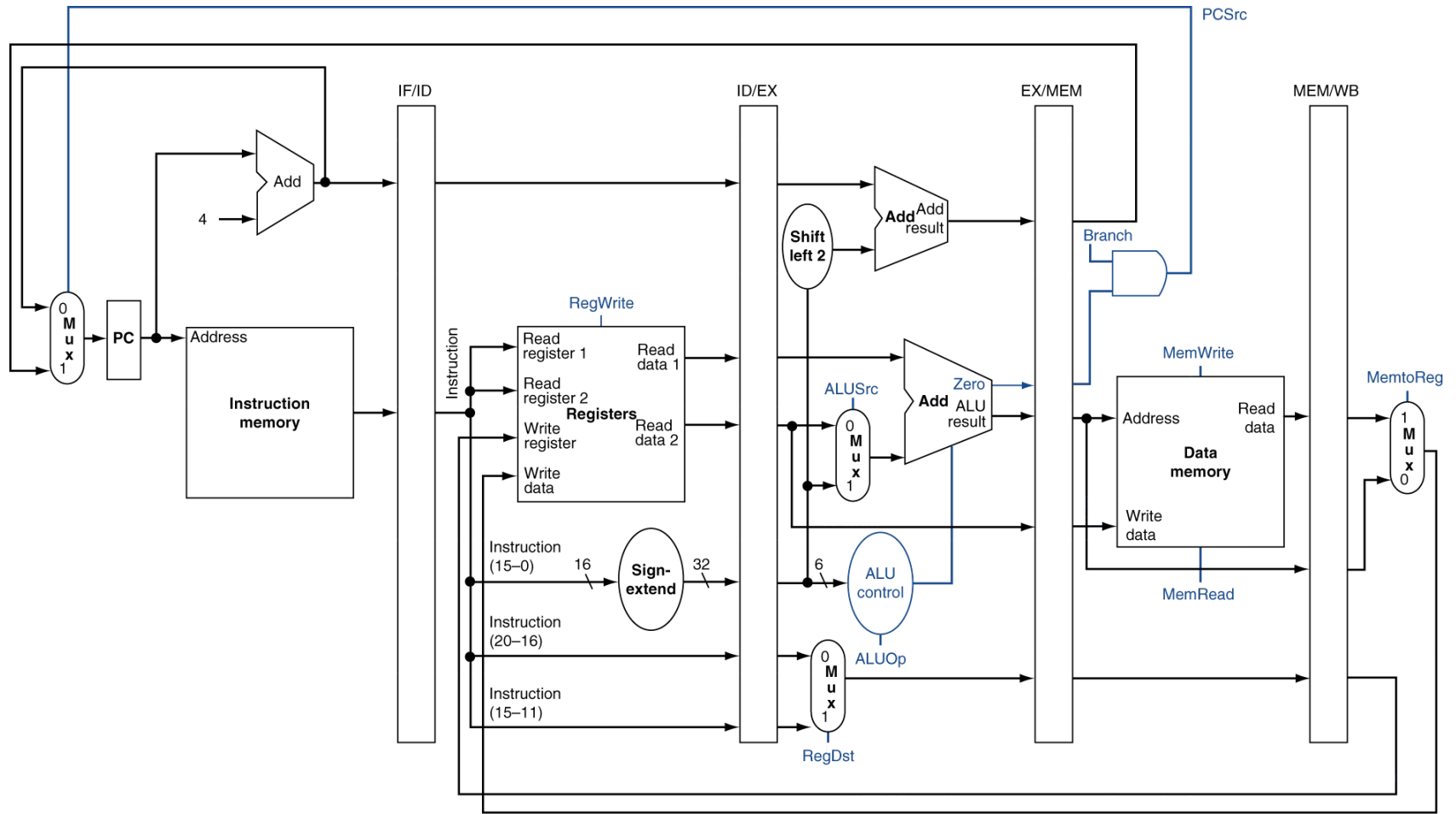
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
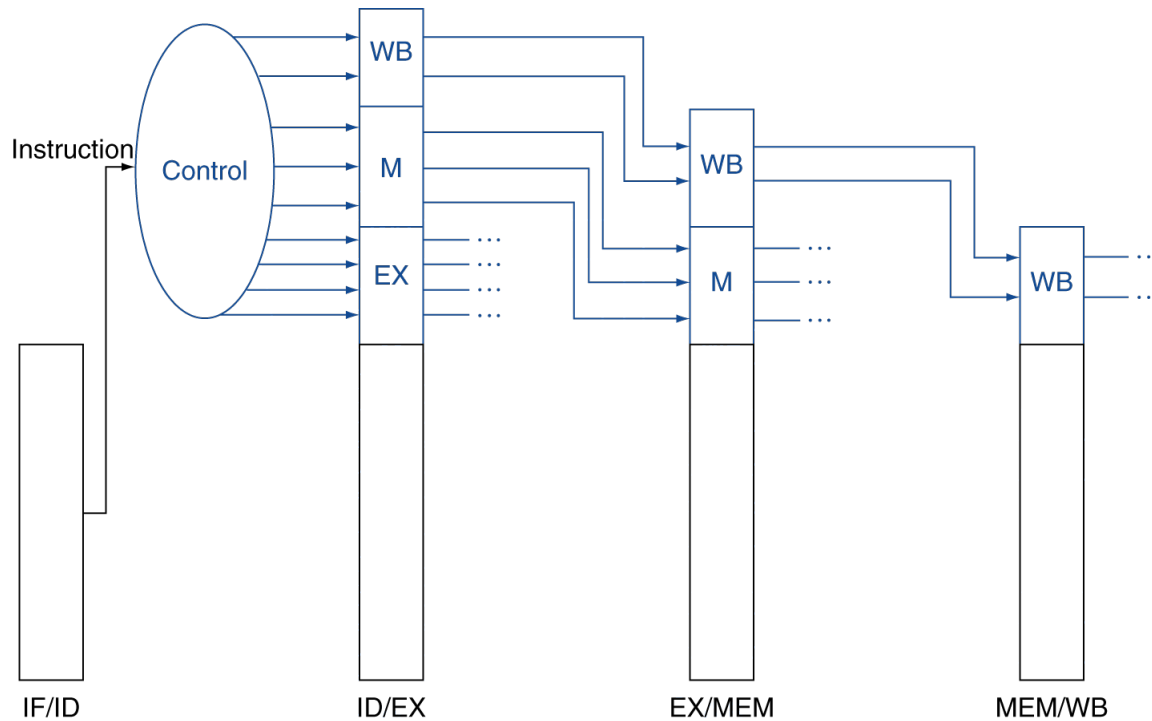
Pipelined Control
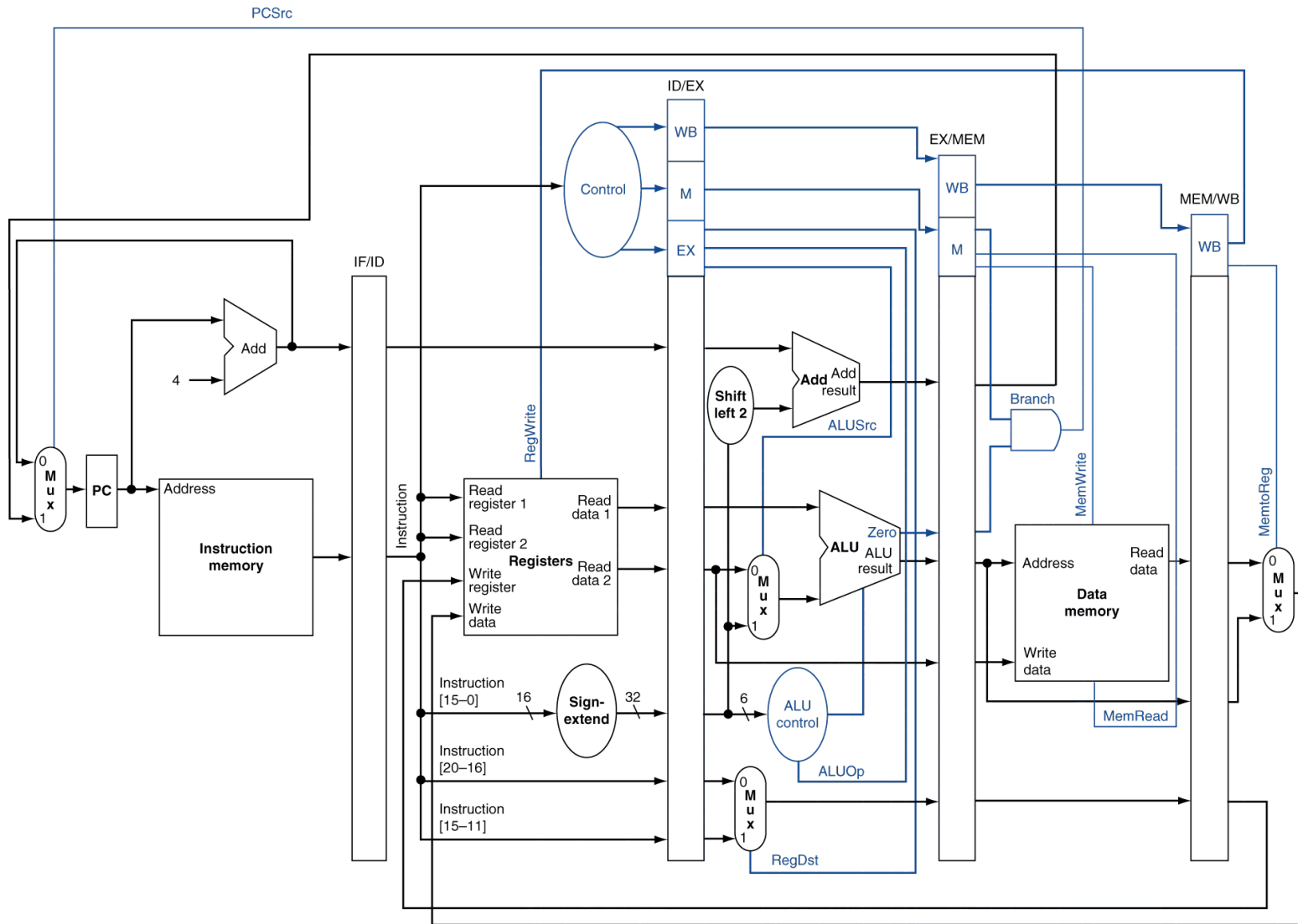
Data Hazards in ALU Instructions

- Consider this sequence:

```
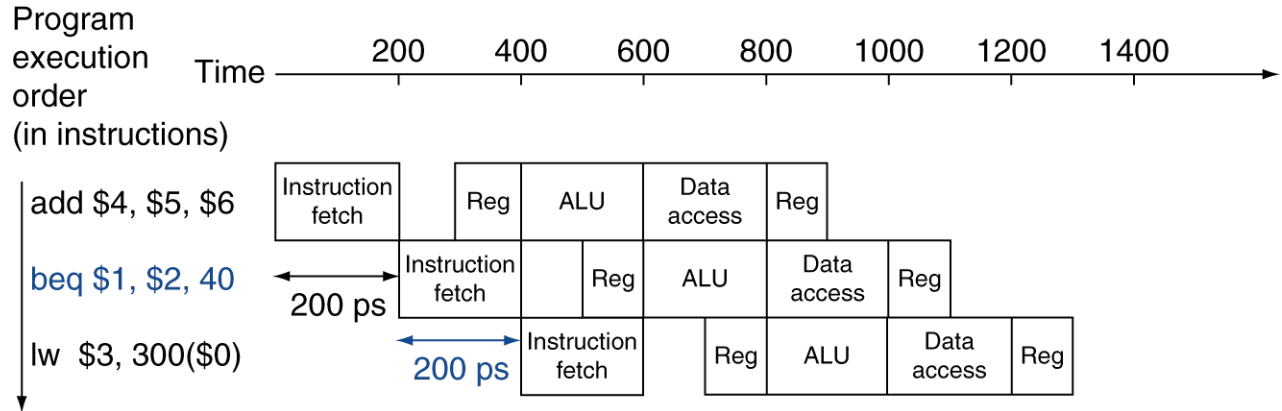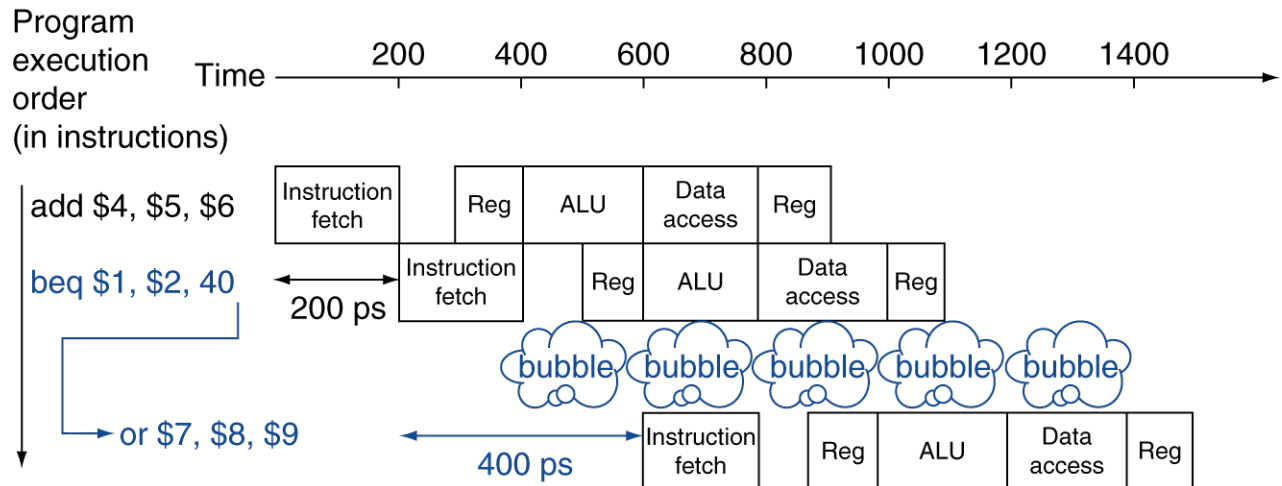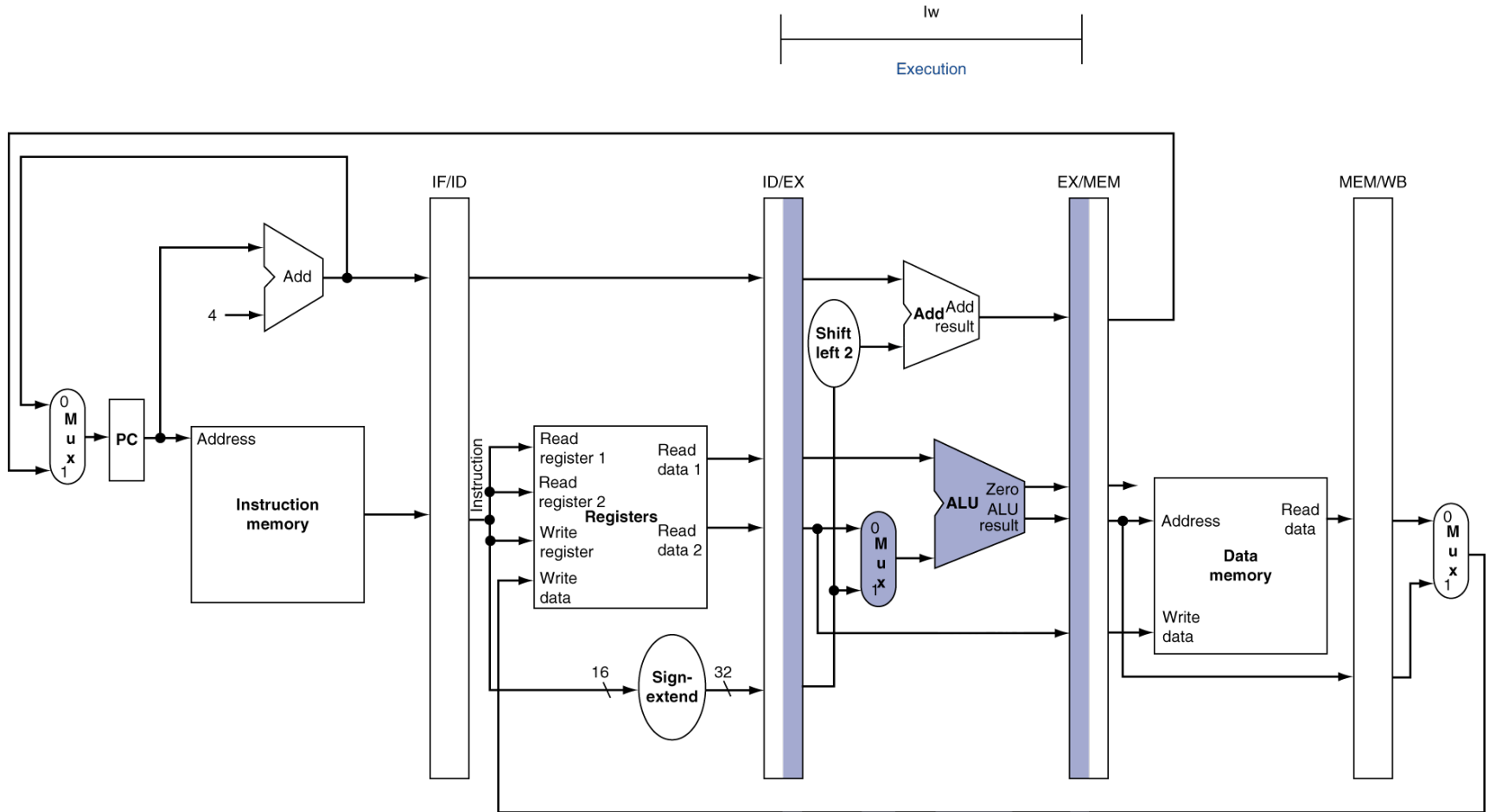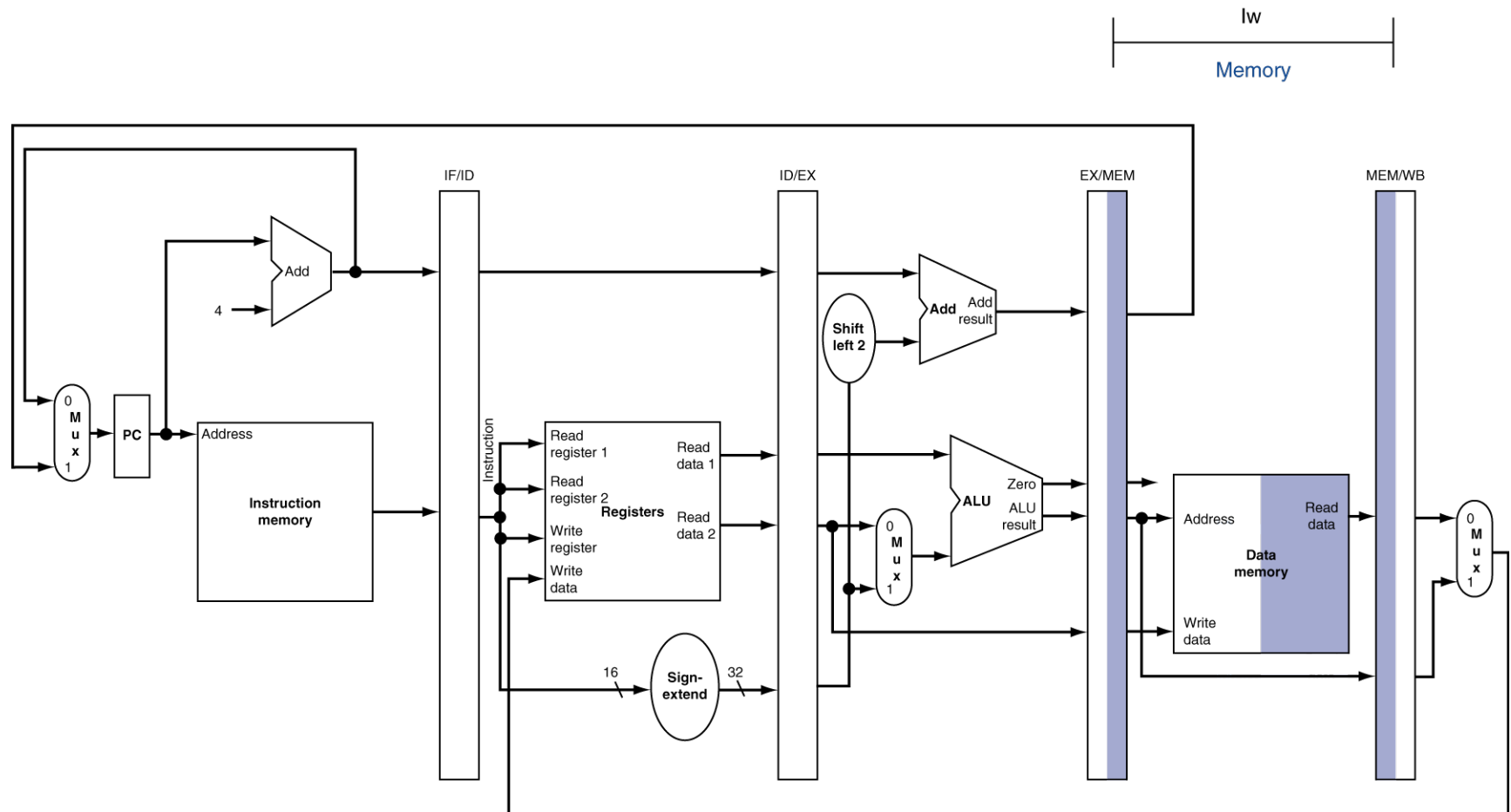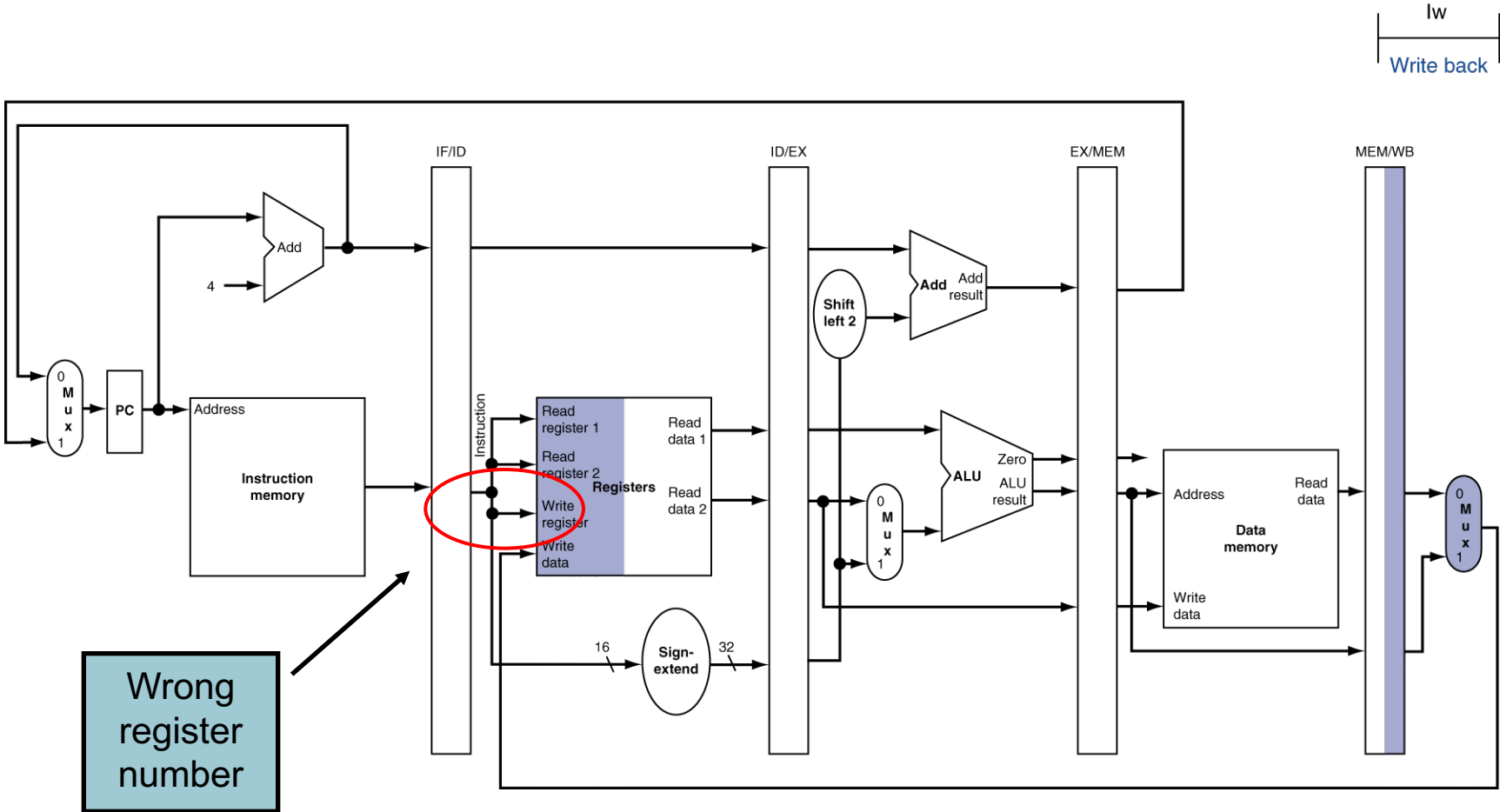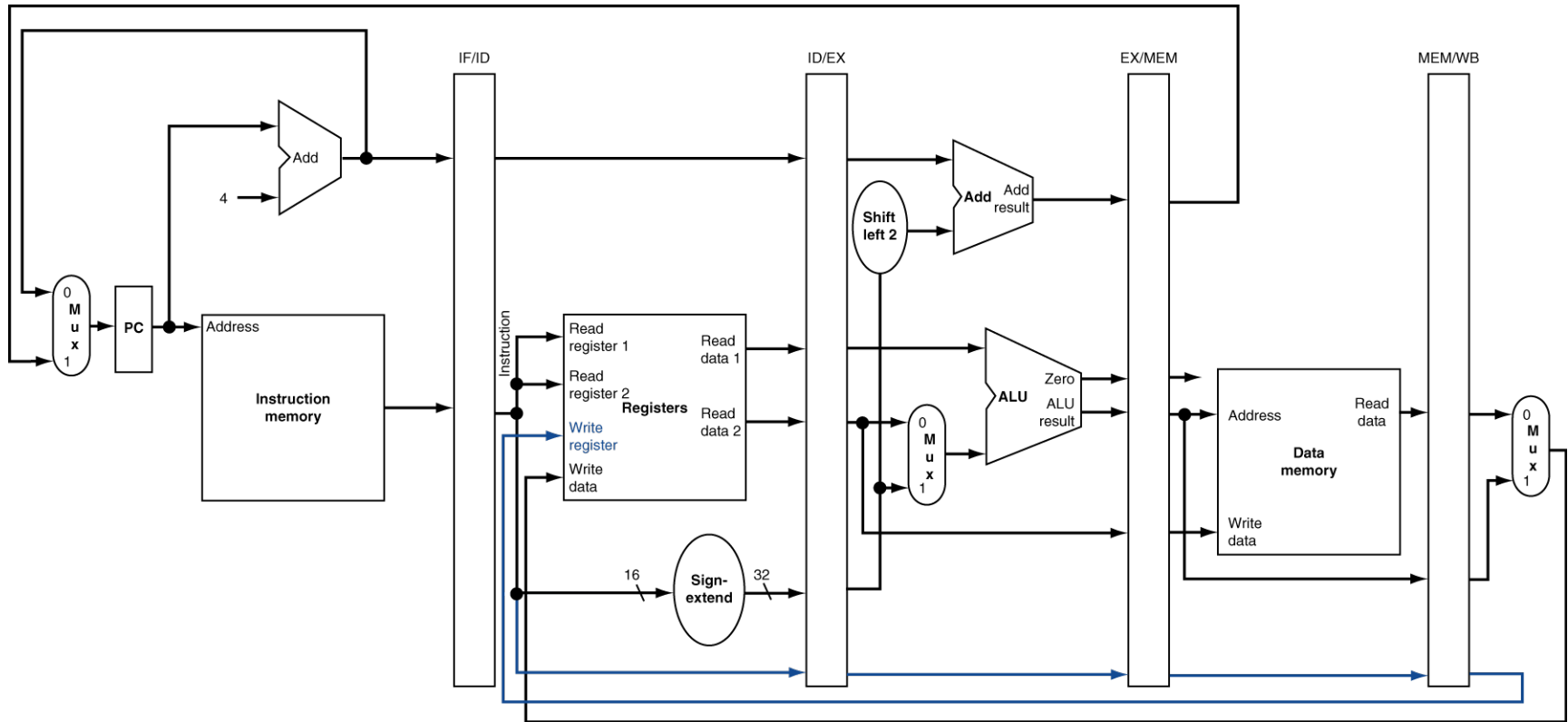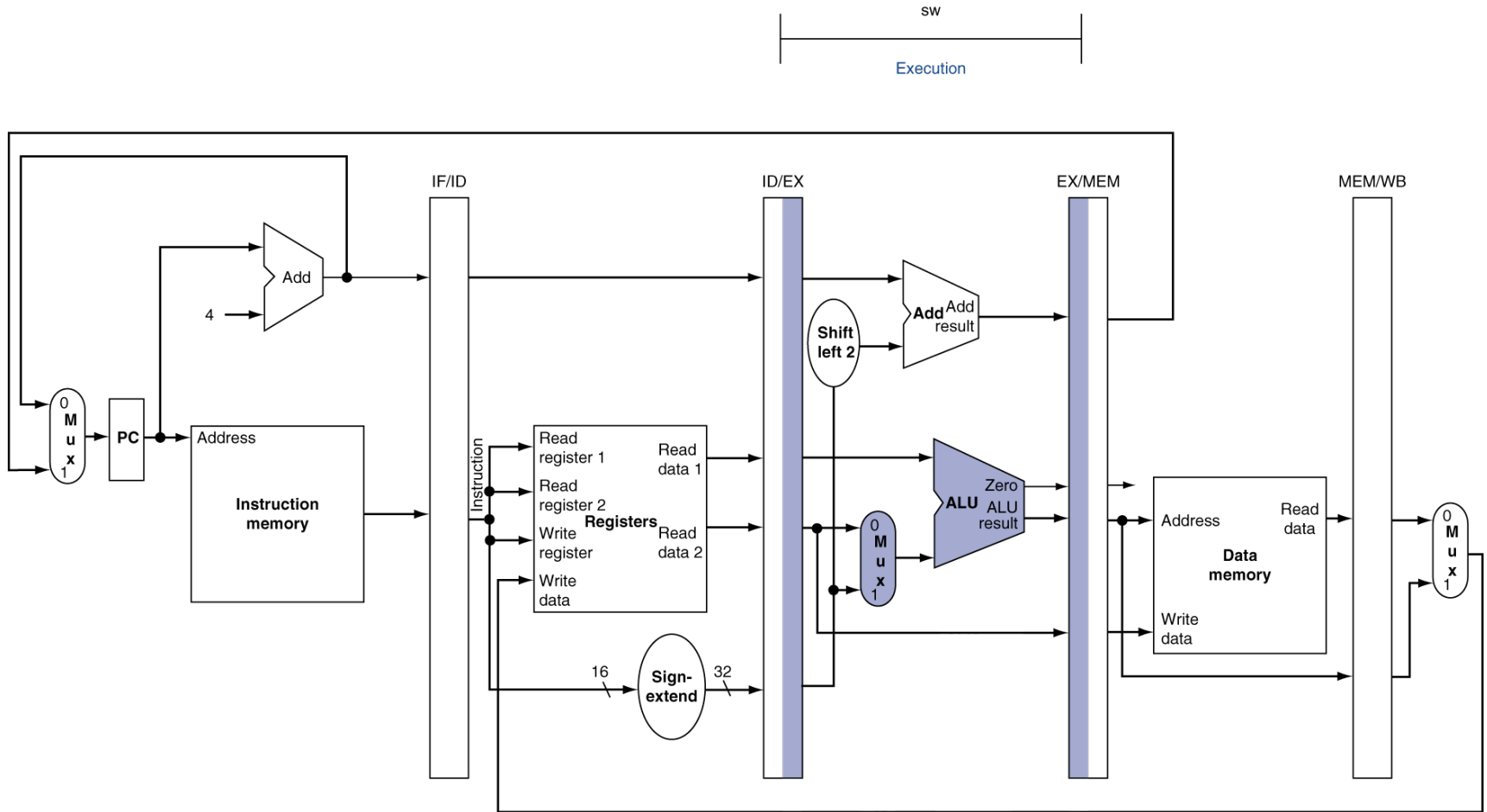sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding
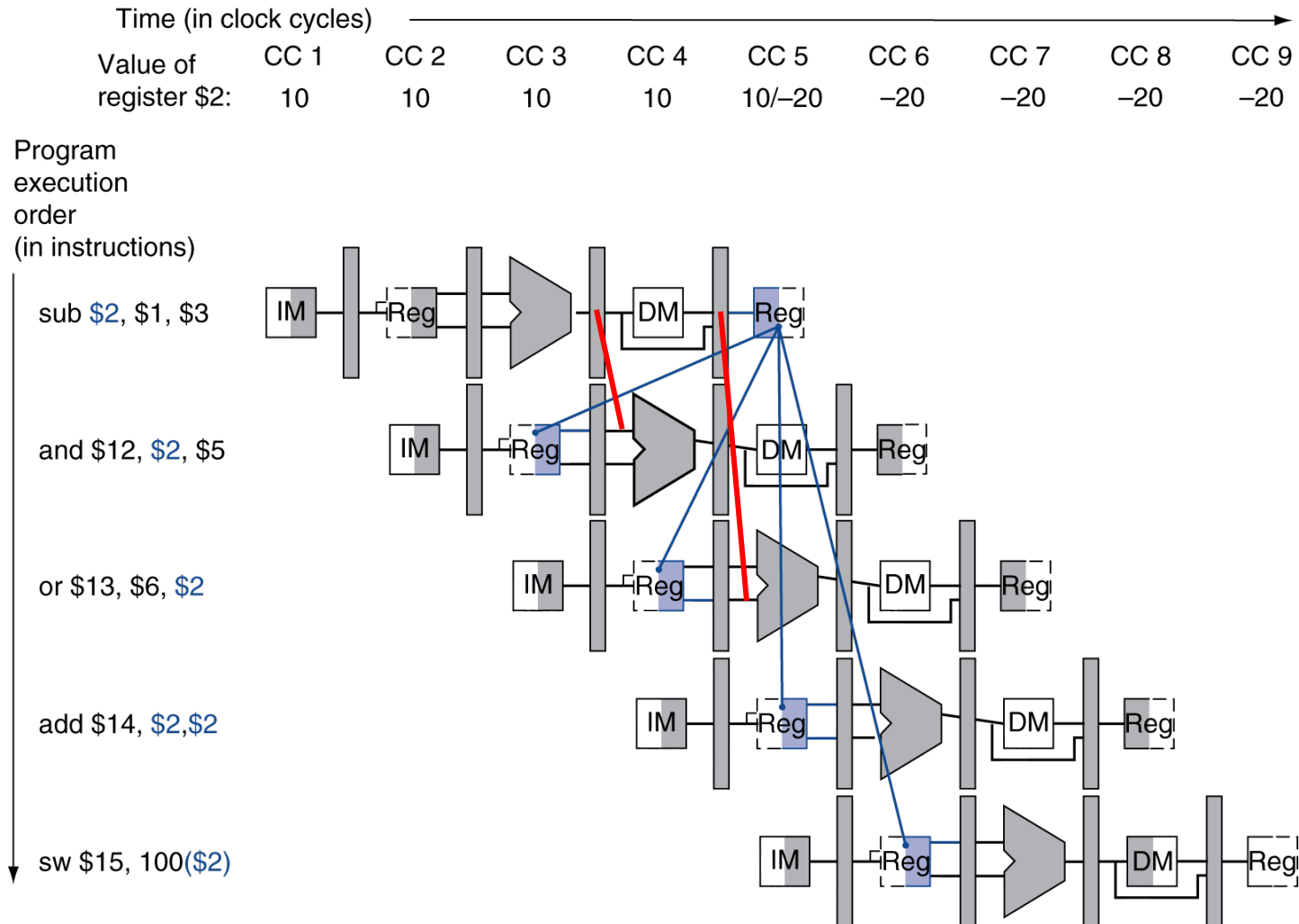
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

    Fwd from EX/MEM pipeline reg

  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

    Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
    - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
    - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Paths
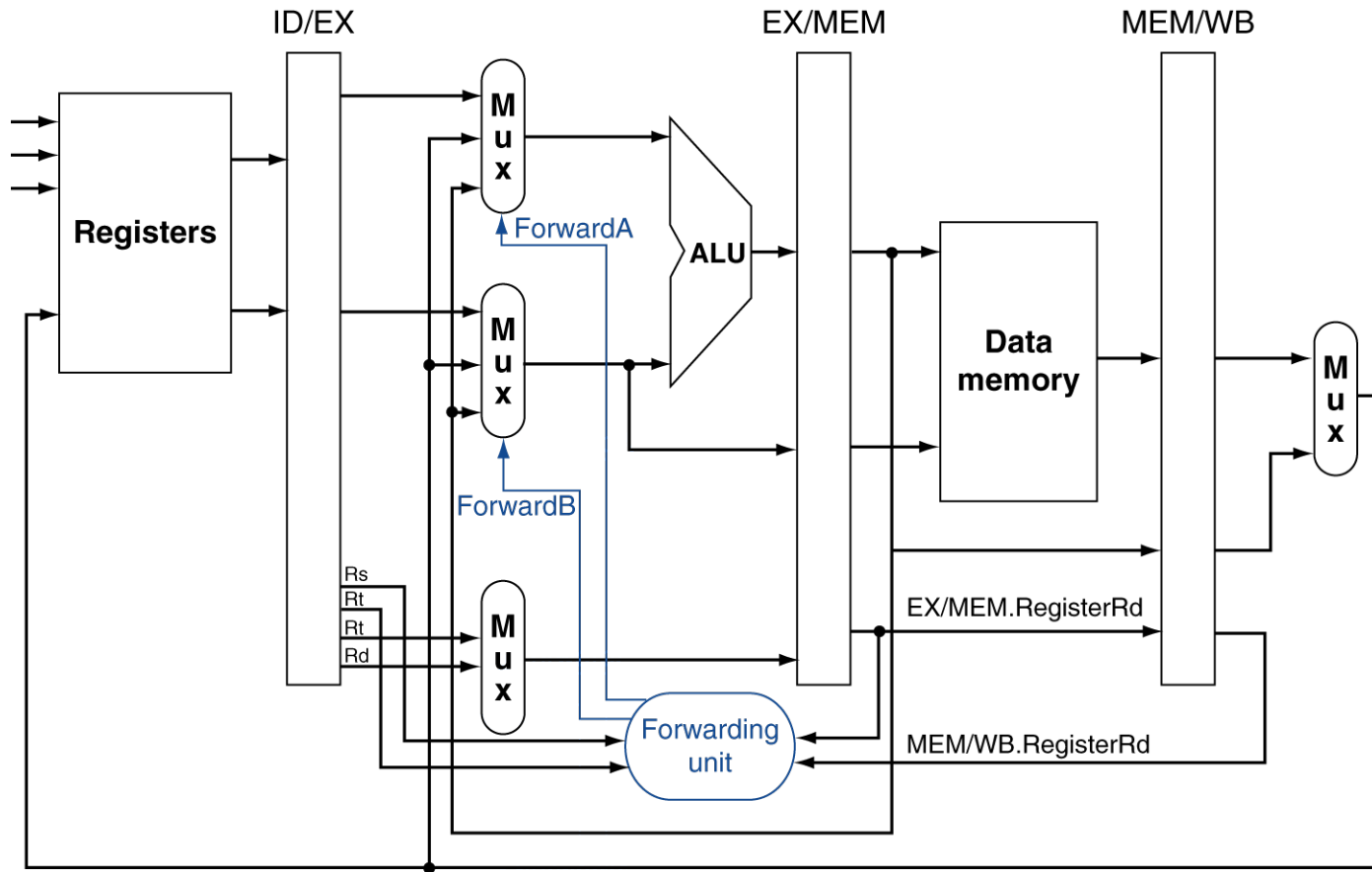
b. With forwarding

Forwarding Conditions

- EX hazard

  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10

  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10

- MEM hazard

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:

  ```
  add  $1,$1,$2
  add  $1,$1,$3
  add  $1,$1,$4
  ```

- Both hazards occur

  - Want to use the most recent

- Revise MEM hazard condition

  - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

- MEM hazard

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

    ForwardA = 01

  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

    ForwardB = 01

Datapath with Forwarding
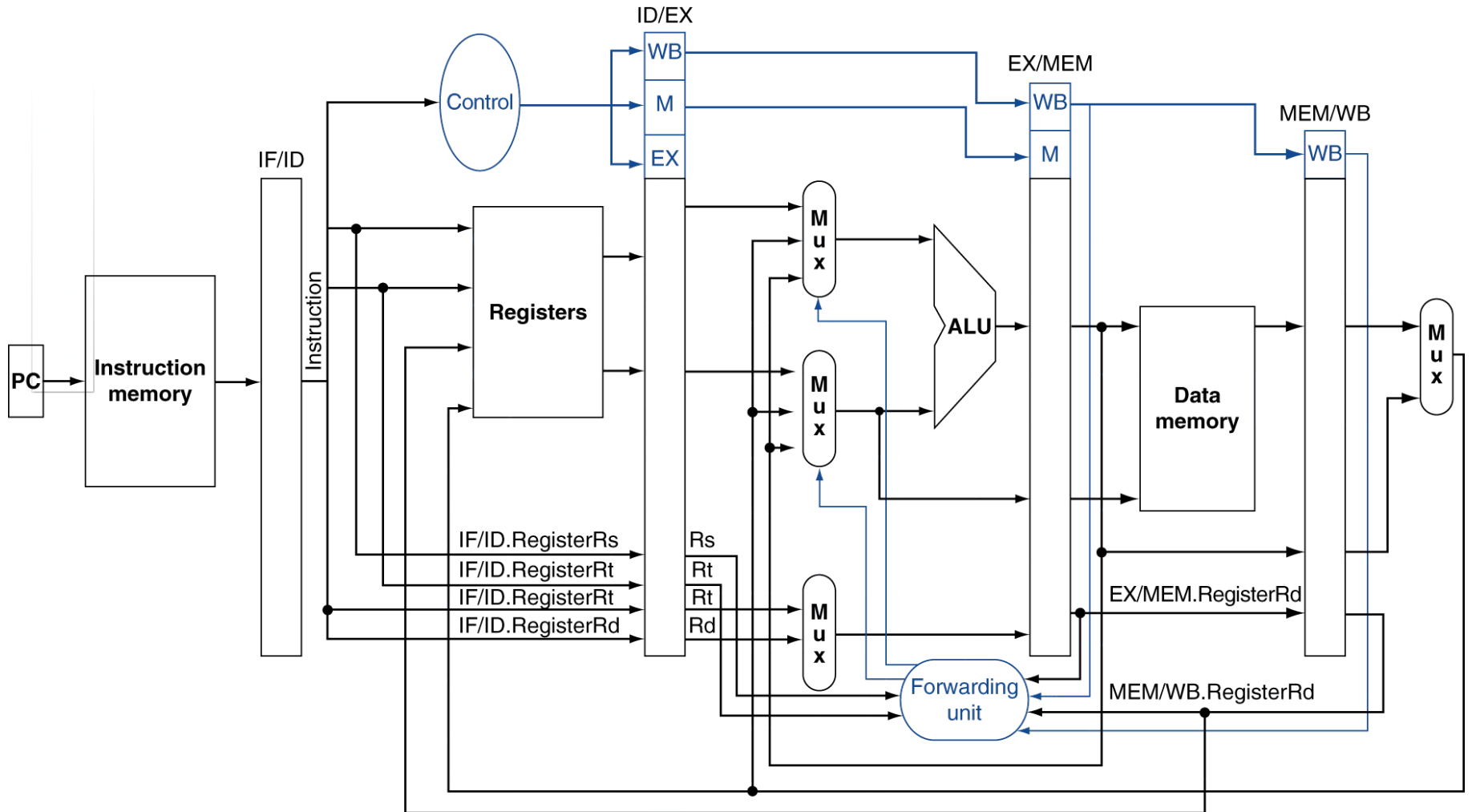
Load-Use Data Hazard
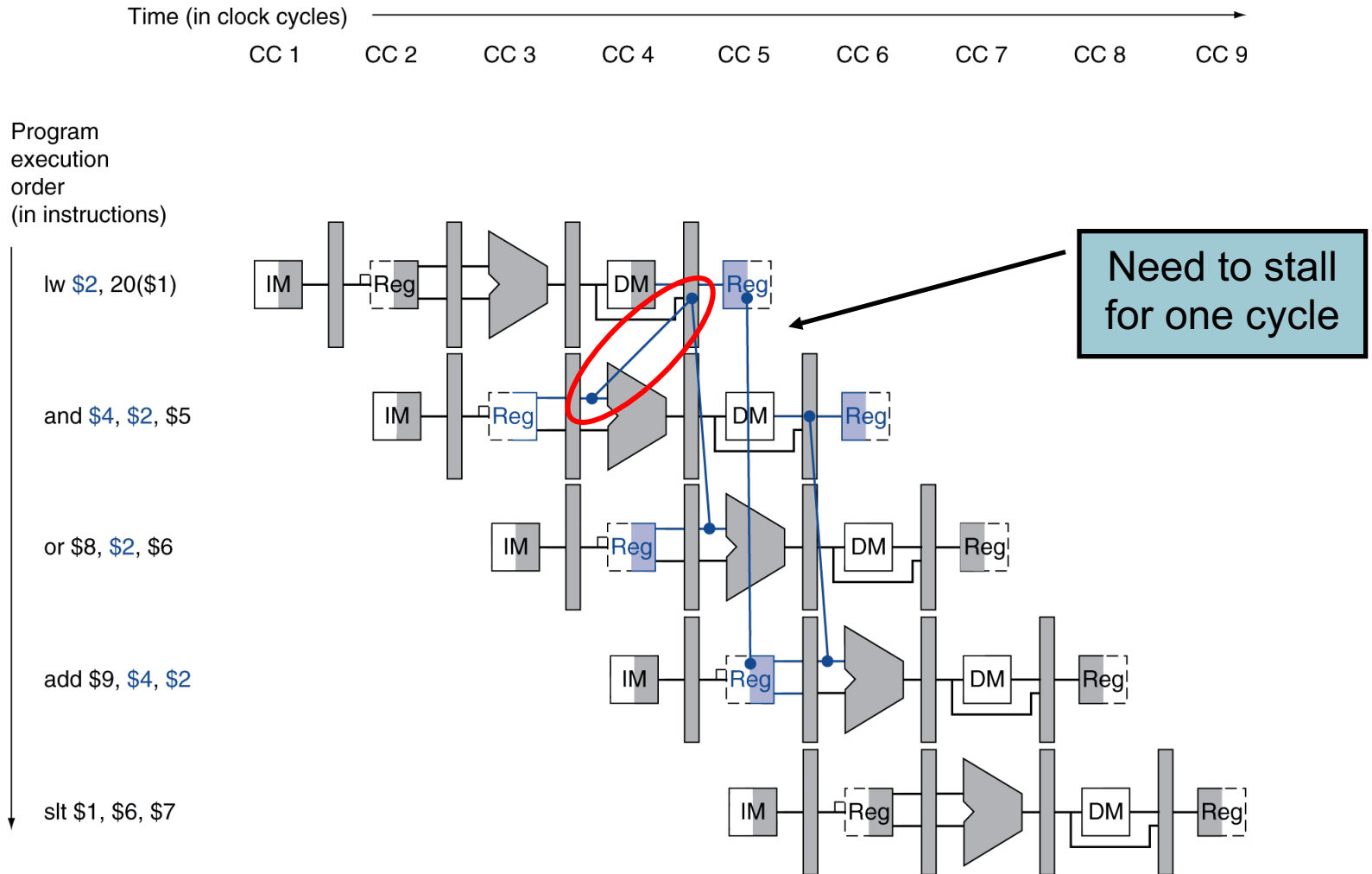
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage

- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt

- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))

- If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

Stall/Bubble in the Pipeline

Or, more accurately…
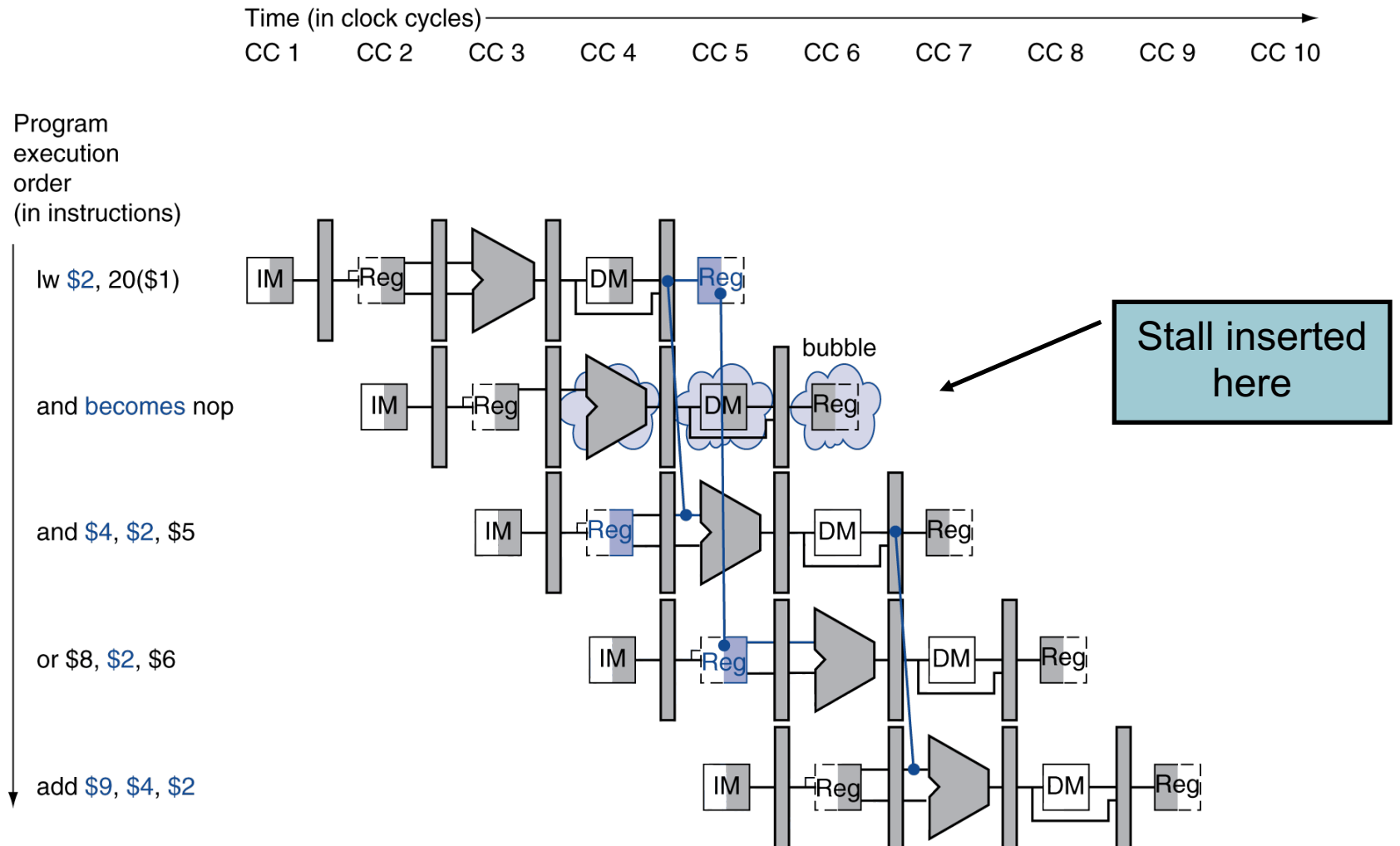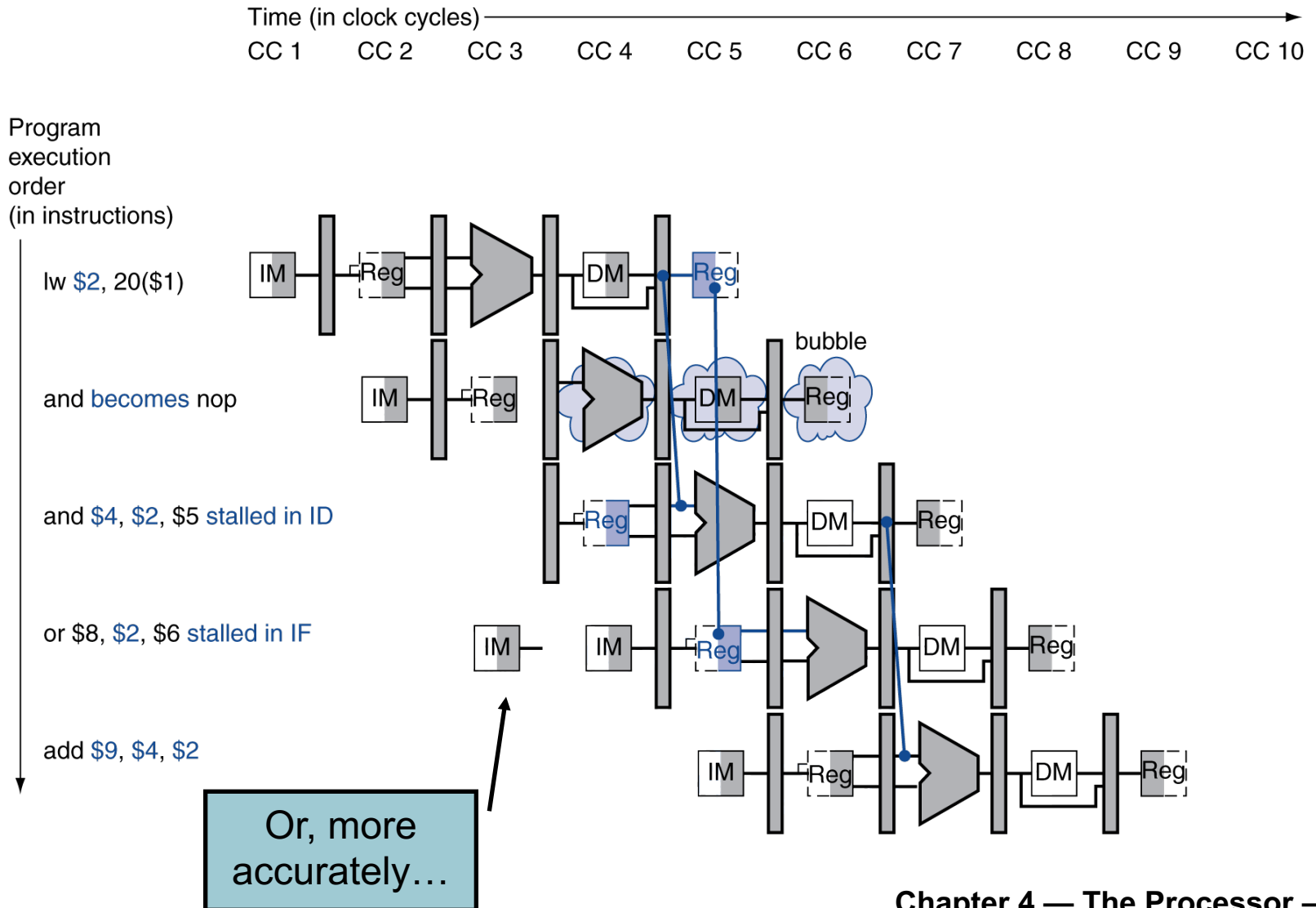
Datapath with Hazard Detection
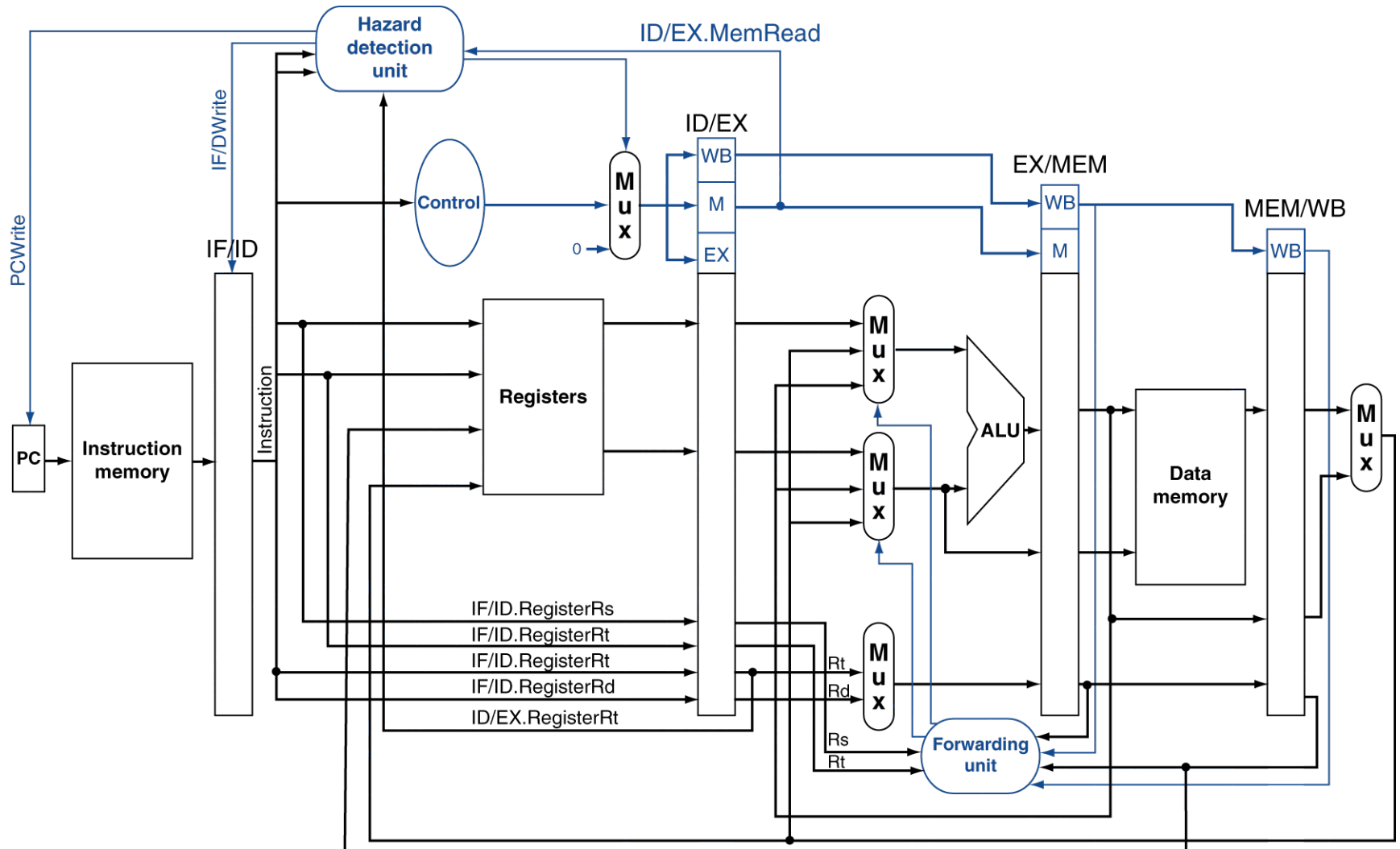
Stalls and Performance

The BIG Picture

- Stalls reduce performance
    - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
    - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM

Time (in clock cycles)

CC 1   CC 2   CC 3   CC 4   CC 5   CC 6   CC 7   CC 8   CC 9

Program
execution
order
(in instructions)

40 beq $1, $3, 28   IM   Reg   DM   Reg

44 and $12, $2, $5   IM   Reg   DM   Reg

48 or $13, $6, $2   IM   Reg   DM   Reg

52 add $14, $2, $2   IM   Reg   DM   Reg

72 lw $4, 50($7)   IM   Reg   DM   Reg

PC

Flush these
instructions
(Set control
values to 0)
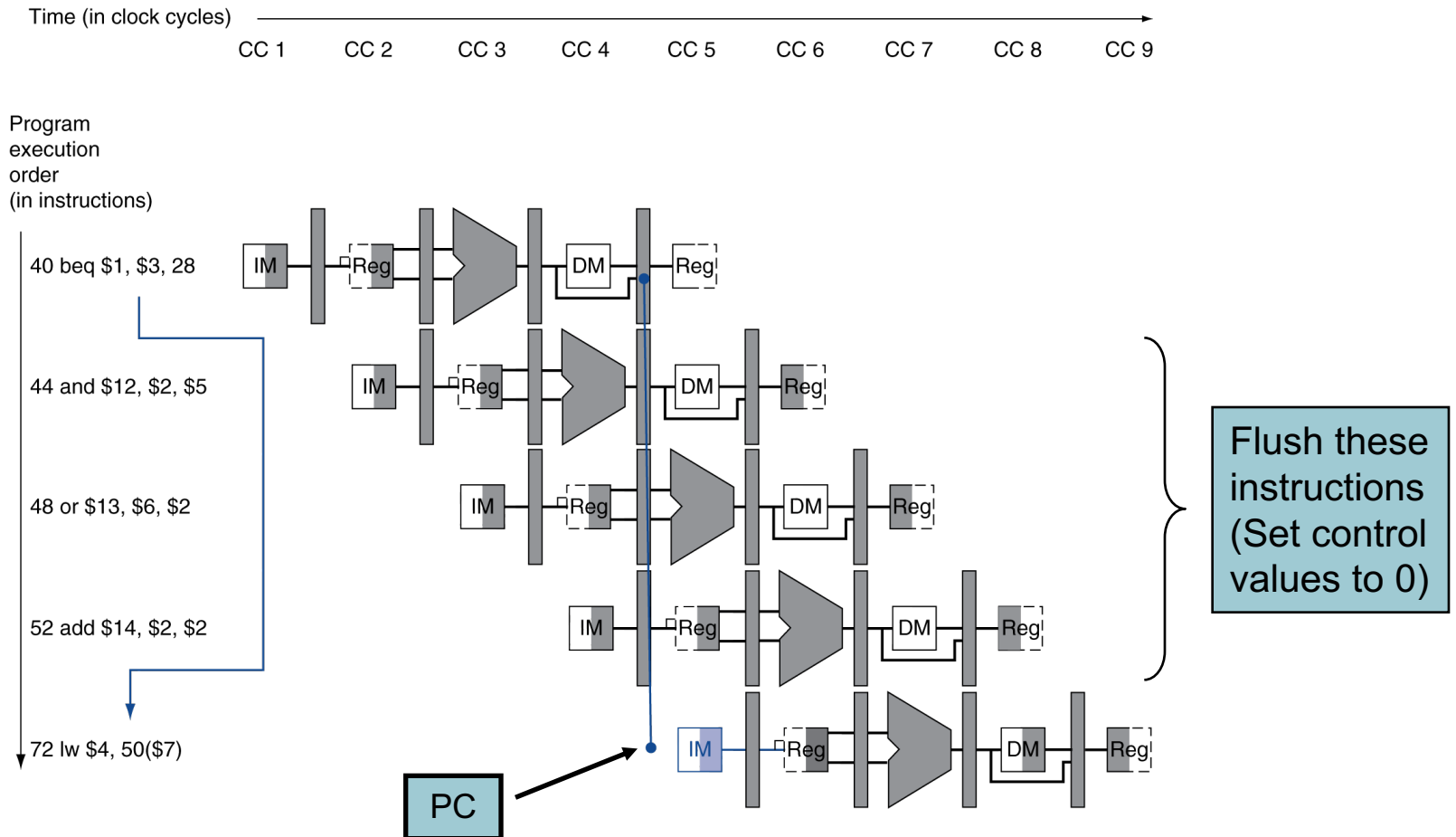
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3      IF | ID | EX | MEM | WB

add $4, $5, $6          IF | ID | EX | MEM | WB

…                           IF | ID | EX | MEM | WB

beq $1, $4, target              IF | ID | EX | MEM | WB
```
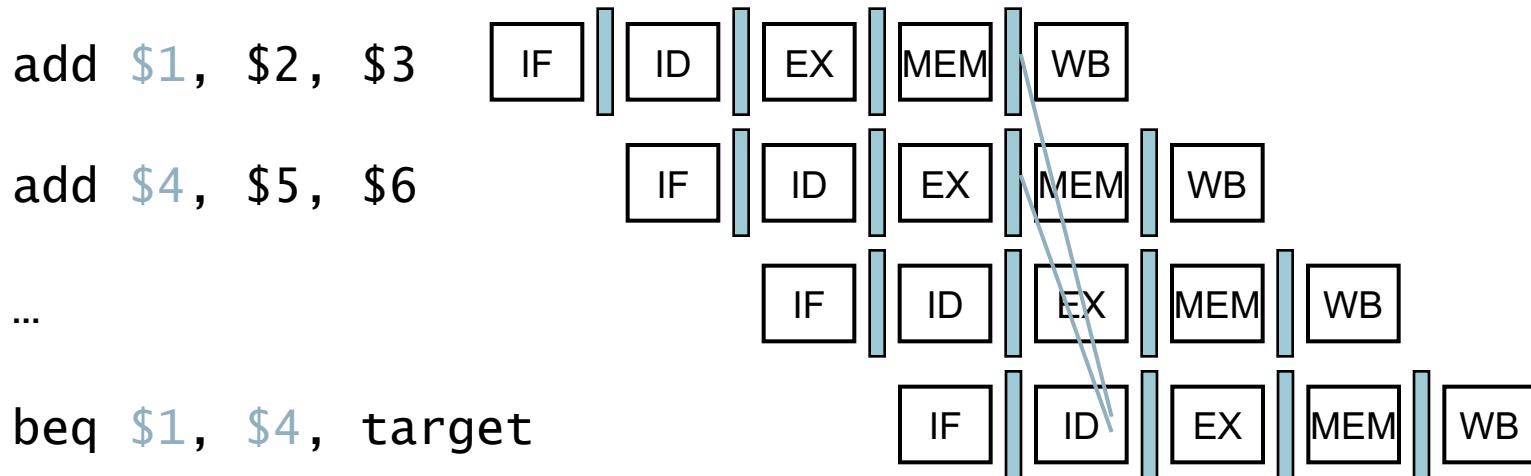
- Can resolve using forwarding

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction

  - Need 1 stall cycle

```
lw   $1, addr         IF | ID | EX | MEM | WB

add  $4, $5, $6            IF | ID | EX | MEM | WB

beq  stalled                   IF | ID | ☁ | ☁ | ☁

beq  $1, $4, target                      ID | EX | MEM | WB
```
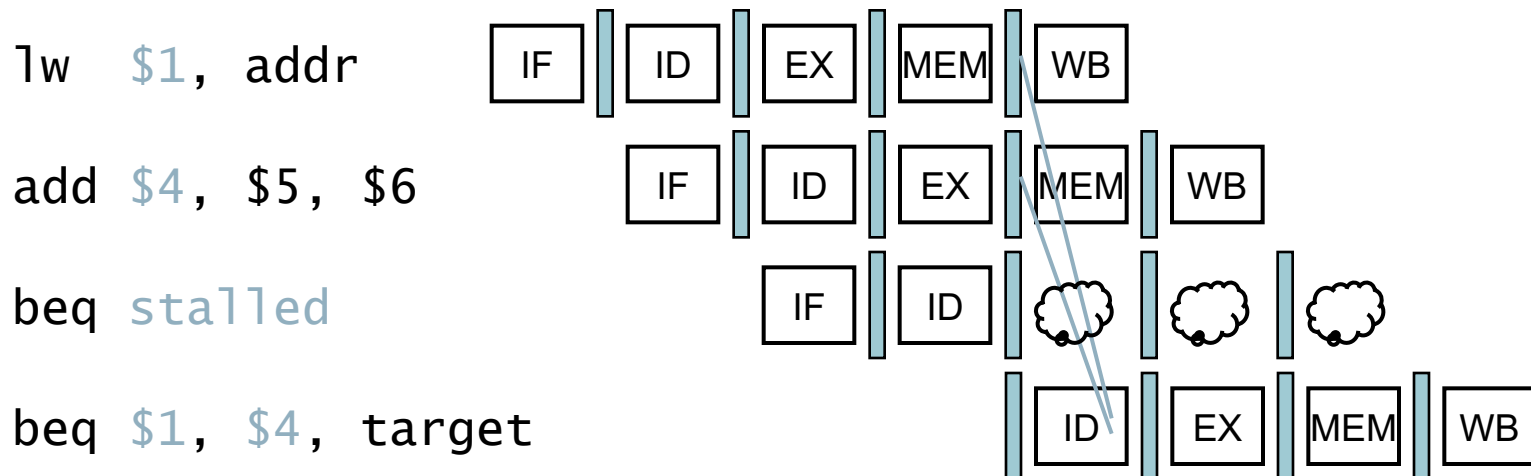
Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw   $1, addr
```
IF | ID | EX | MEM | WB

```
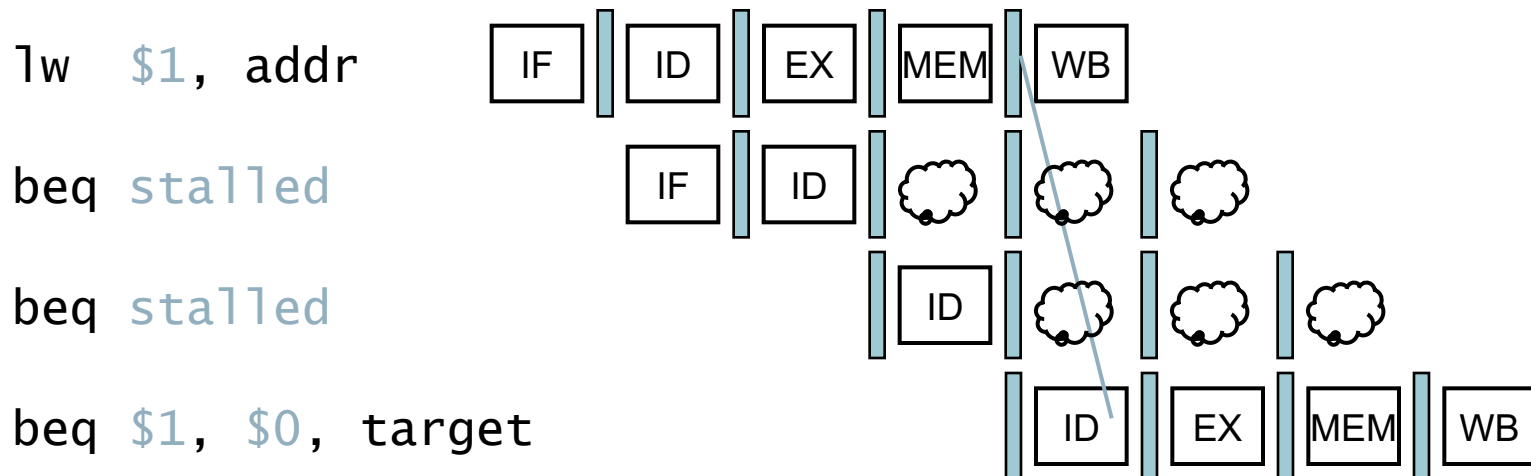beq stalled
```
IF | ID | ☁ | ☁ | ☁

```
beq stalled
```
ID | ☁ | ☁ | ☁

```
beq $1, $0, target
```
ID | EX | MEM | WB

Dynamic Branch Prediction

■ In deeper and superscalar pipelines, branch penalty is more significant

■ Use dynamic prediction

- Branch prediction buffer (aka branch history table)

- Indexed by recent branch instruction addresses

- Stores outcome (taken/not taken)

- To execute a branch

  - Check table, expect the same outcome

  - Start fetching from fall-through or target

  - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

■ Inner loop branches mispredicted twice!

```
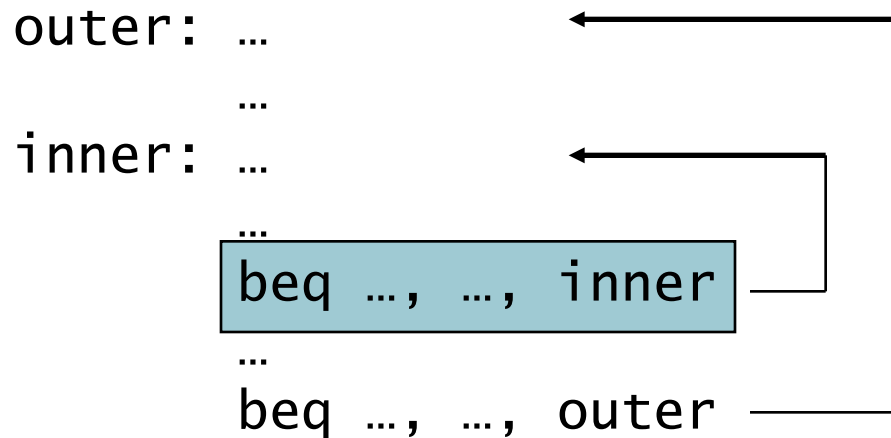outer: …
          …
inner: …
          …
       beq …, …, inner
          …
       beq …, …, outer
```

■ **Mispredict as taken on last iteration of inner loop**

■ **Then mispredict as not taken on first iteration of inner loop next time around**

2-Bit Predictor

- Only change prediction on two successive mispredictions
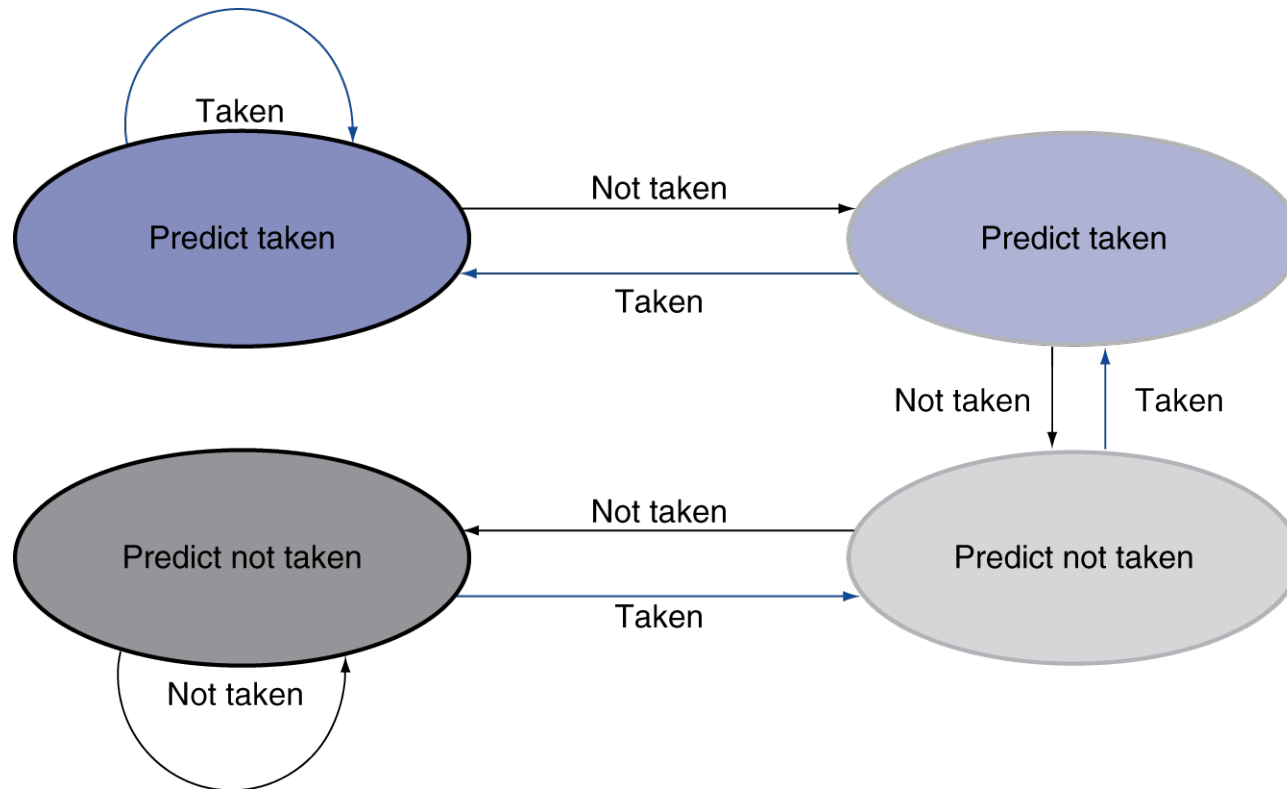
Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder

  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach

  - e.g., complex addressing modes
    - Register update side effects, memory indirection

  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control

- Datapath and control influence design of ISA

- Pipelining improves instruction throughput using parallelism

  - More instructions completed per second

  - Latency for each instruction not reduced

- Hazards: structural, data, control