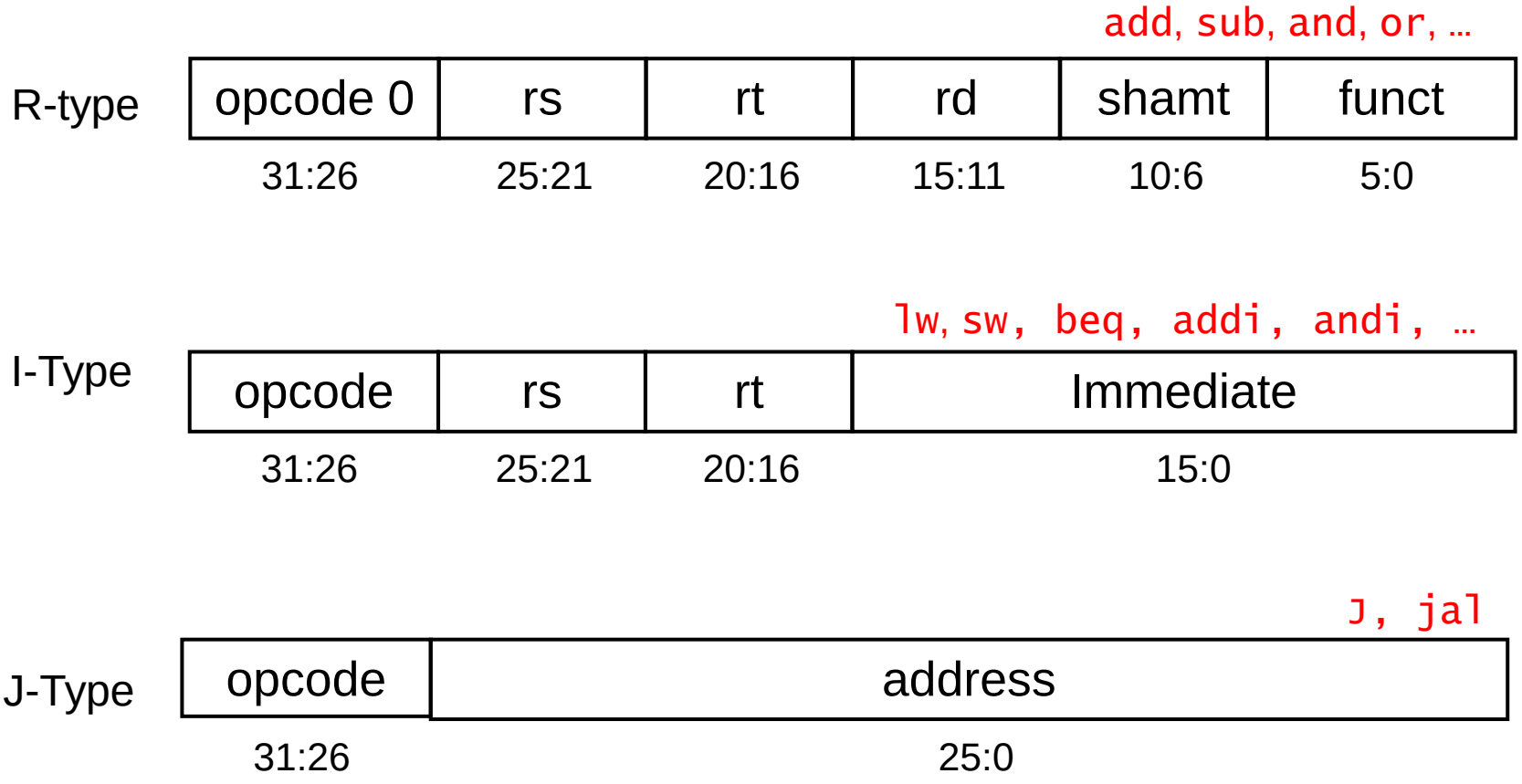
Chapter 4

The Processor

Introduction

- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Instruction types
  - I-Type: lw, sw, beq, addi, andi, …
  - R-Type: add, sub, and, or, …
  - J-Type: j, jal

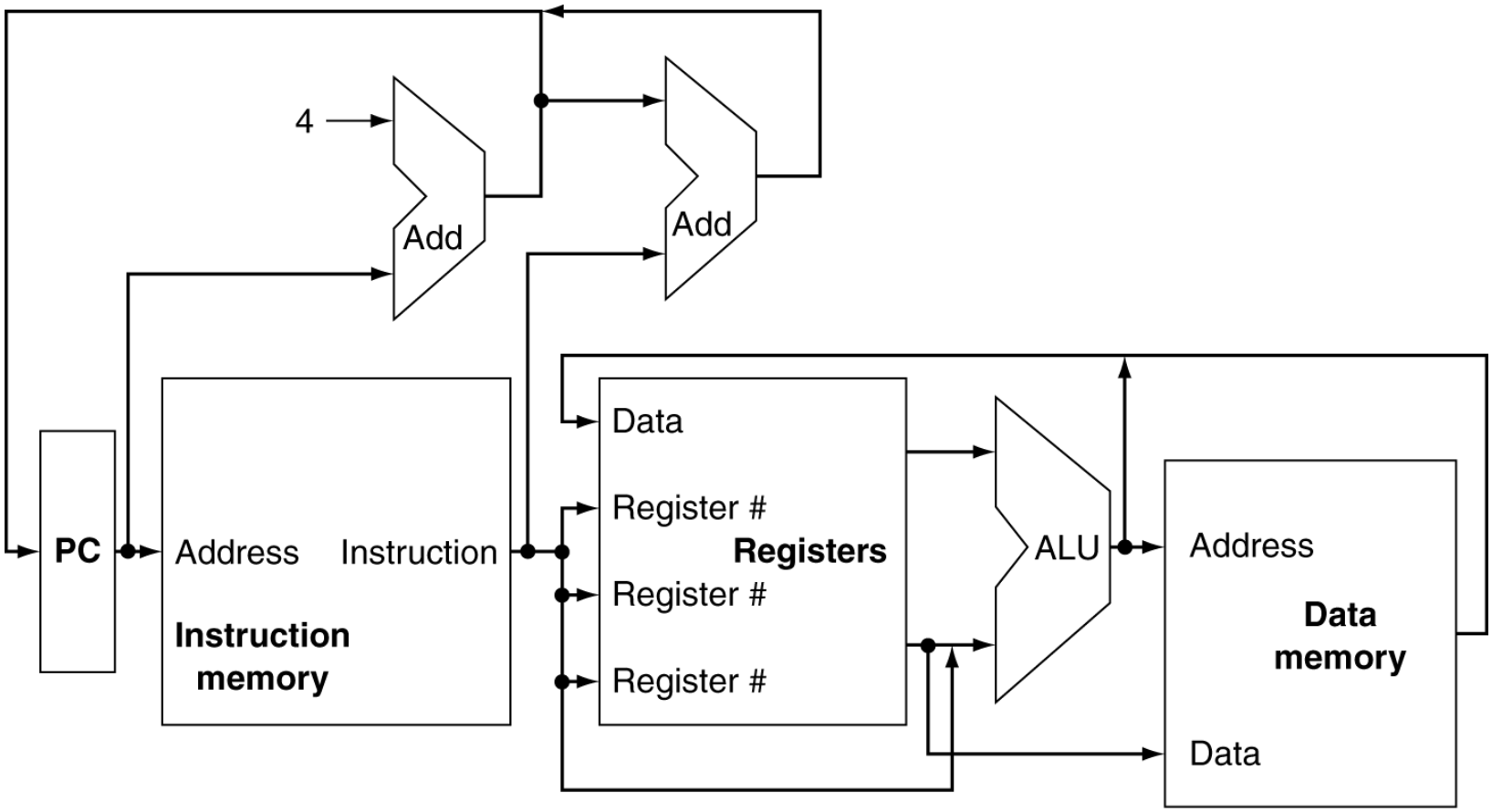
MIPS Instruction Encoding



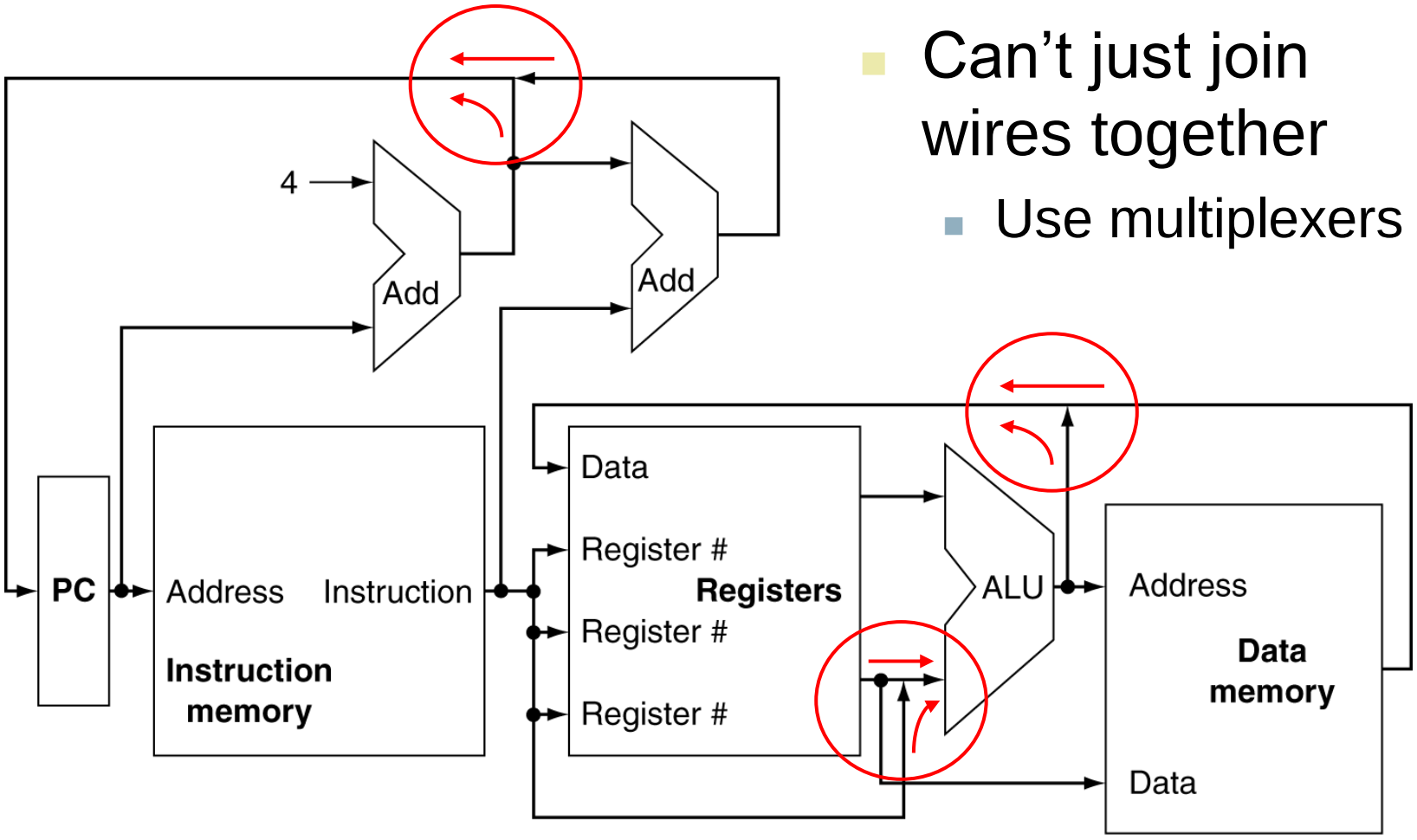
Inside the CPU…

- PC → fetch from instruction memory
- Opcode → identify the format of bits 25:0
  - Register numbers → read from register file (R, I)
  - Immediate → to PC (J, I-branch), to ALU (I-other)
  - Funct & shamt → to ALU (R)
- Depending on the instruction
  - Use ALU to calculate:
    - Arithmetic / logic / shift result
    - Memory addresses for load/store
  - Access data memory for load/store

CPU Overview

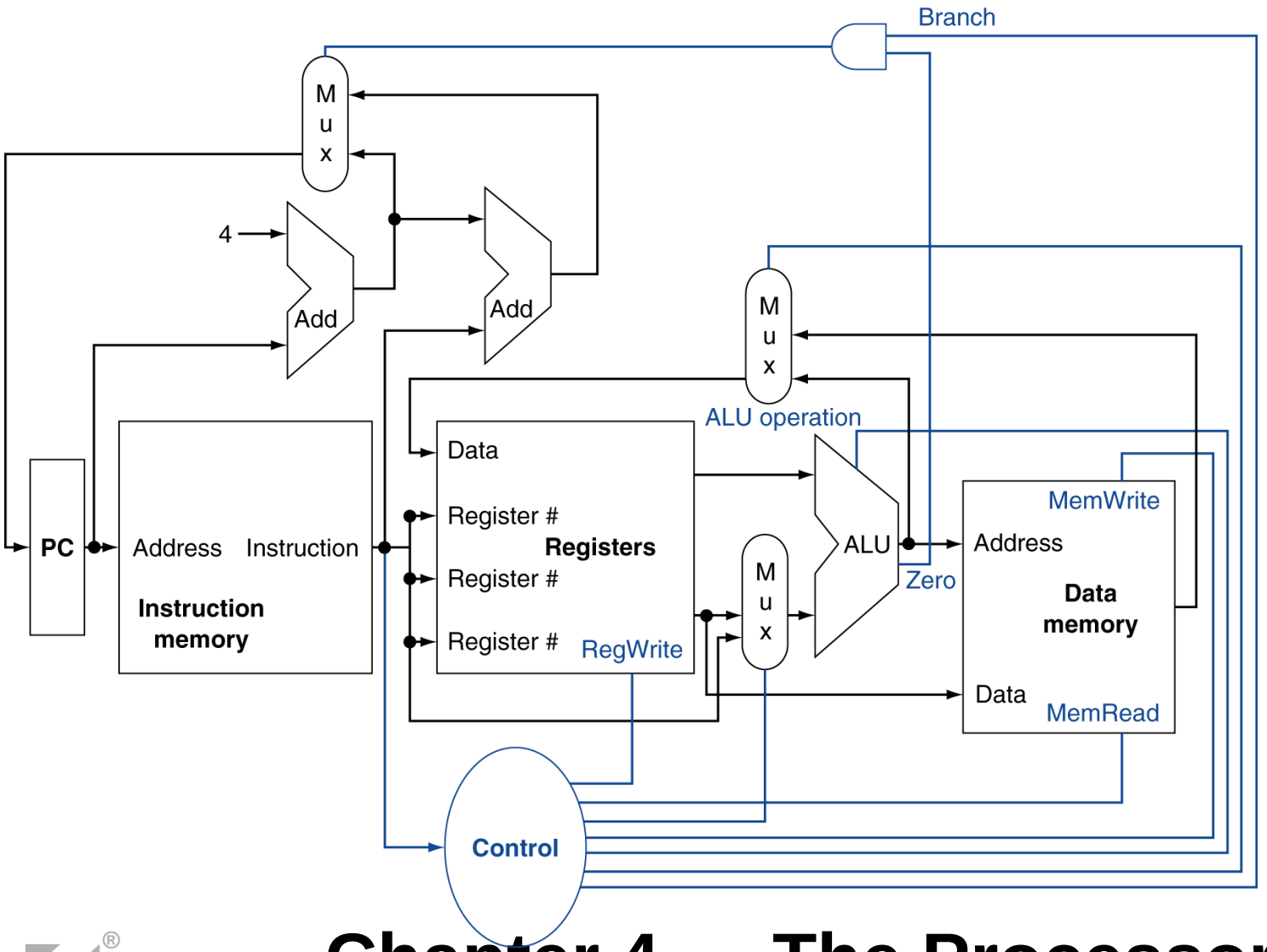


Choices, choices everywhere!



- Can't just join wires together
  - Use multiplexers

Control

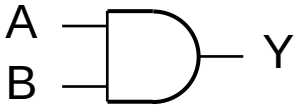

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

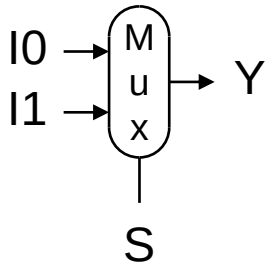
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

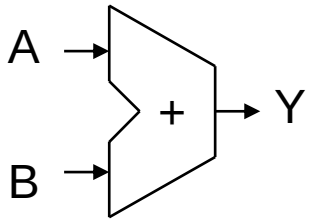
- AND-gate
  - Y = A & B



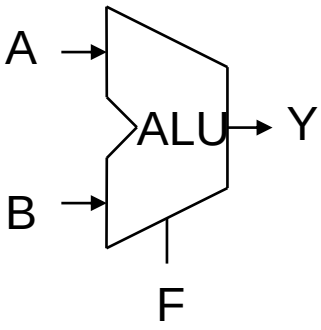
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

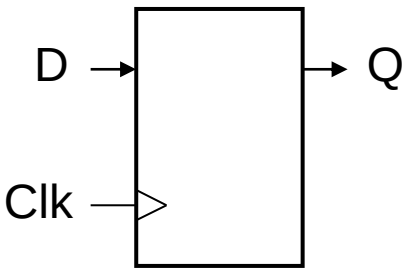


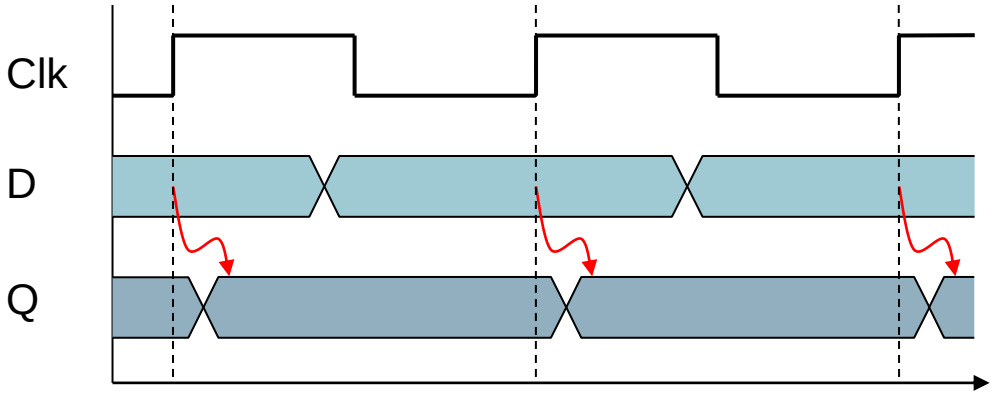
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

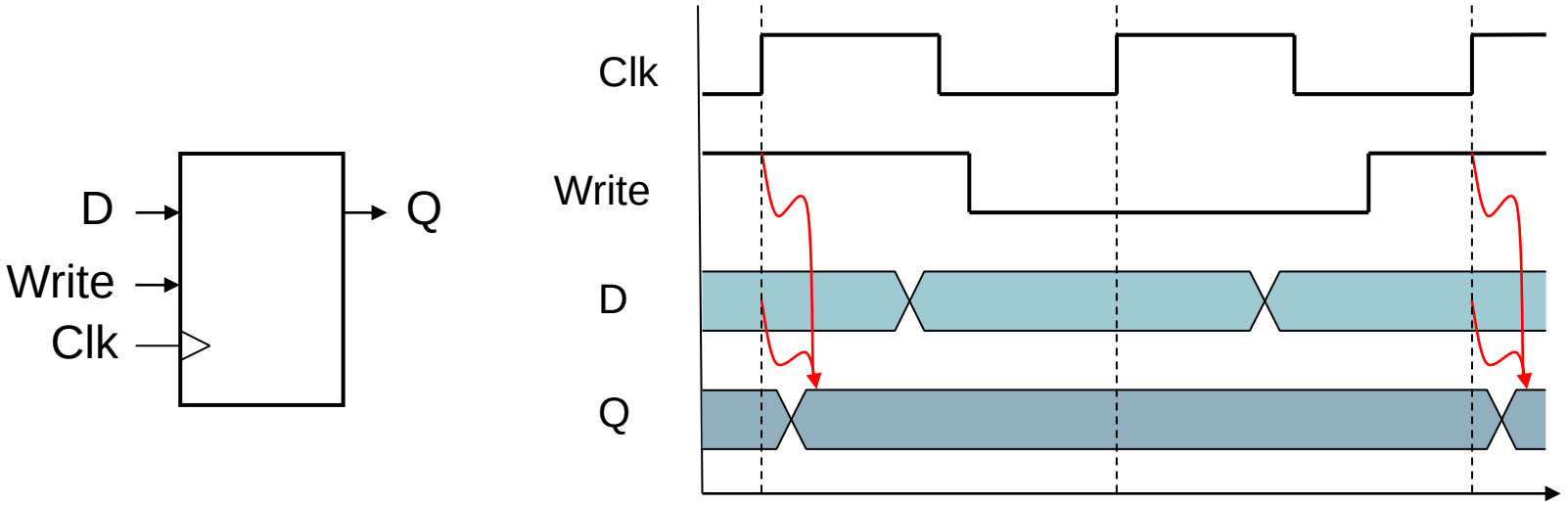
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1 (or from 1 to 0)





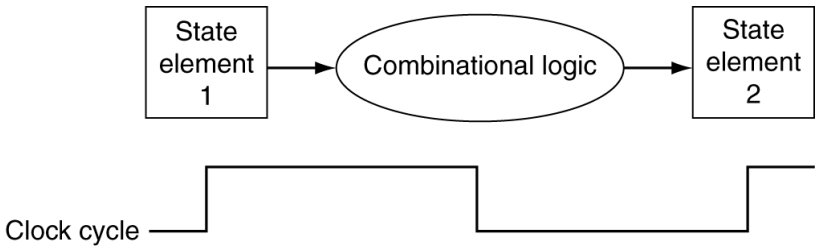
Sequential Elements

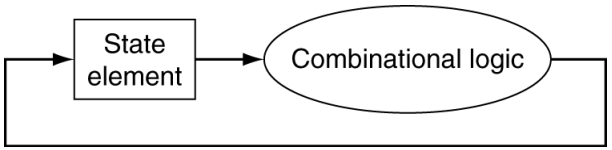
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



Clocking Methodology

- 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

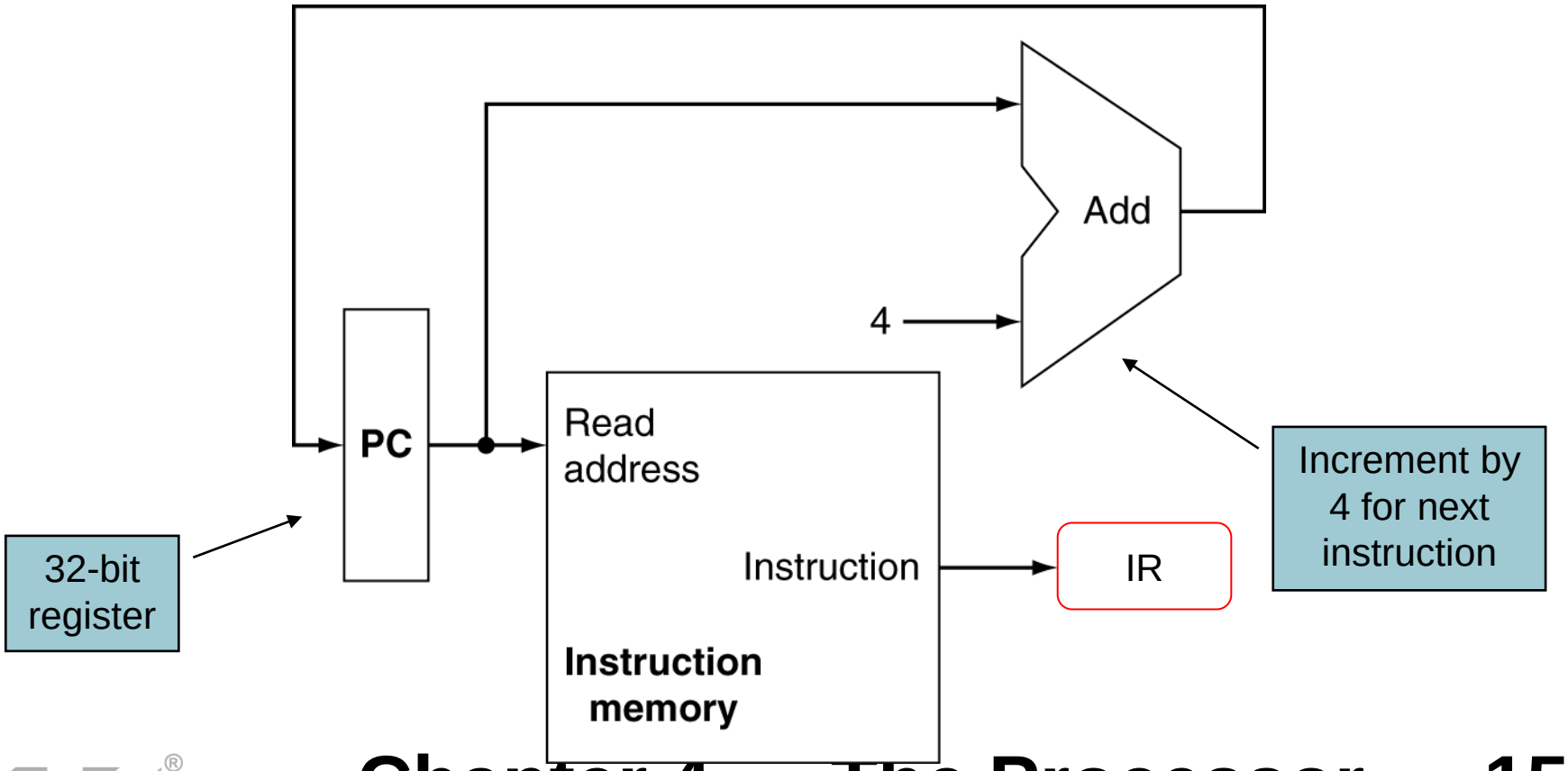
- Datapath
  - Collection of elements that process data and addresses in the CPU
    - Registers, ALU, multiplexers, …
- We will build a MIPS datapath step by step
  - Refining the overview design
  - Adding control logic as required

Instruction Execution

1. PC → fetch from instruction memory
2. Opcode → identify the format of bits 25:0
   - Register numbers → read from register file (R, I)
   - Immediate → to PC (J, I-branch), to ALU (I-other)
3. Funct & shamt → to ALU (R)
4. Depending on the instruction
5. Use ALU to calculate:
6. Arithmetic / logic / shift result
   - Memory address for load/store
   - Access data memory for load/store

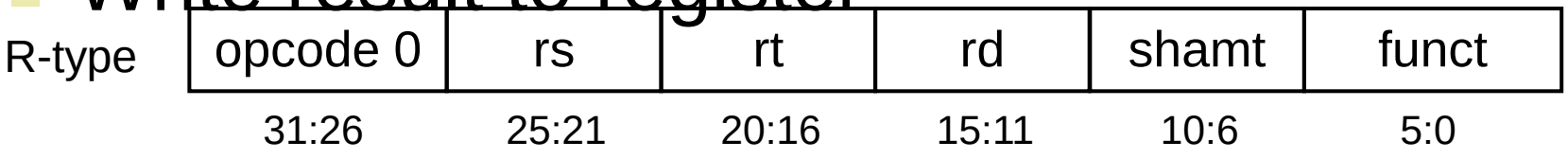
Instruction Fetch

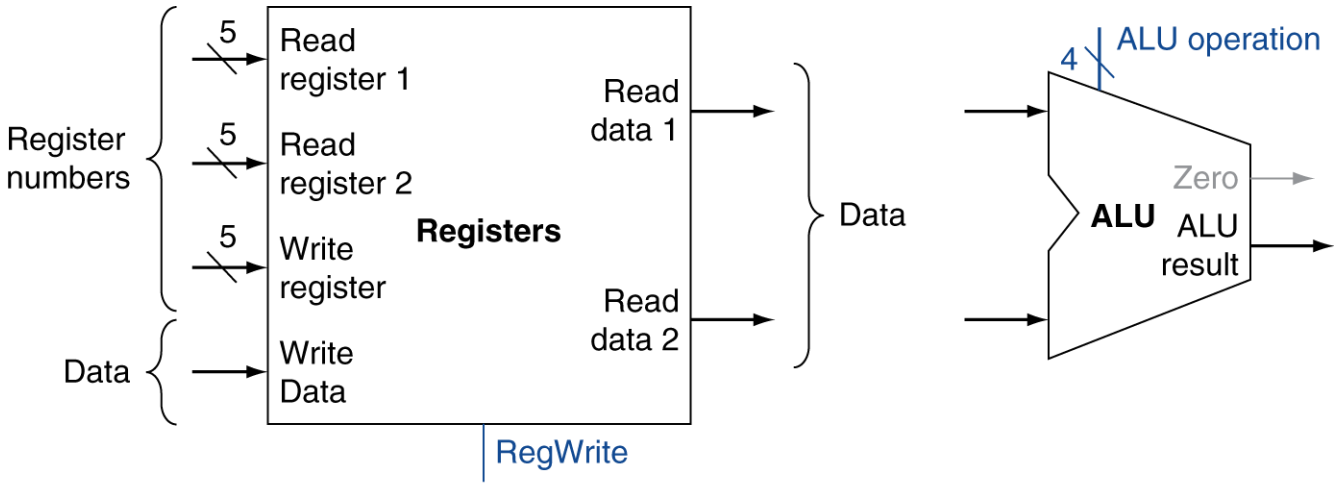
What are the components needed?



R-Type Instructions

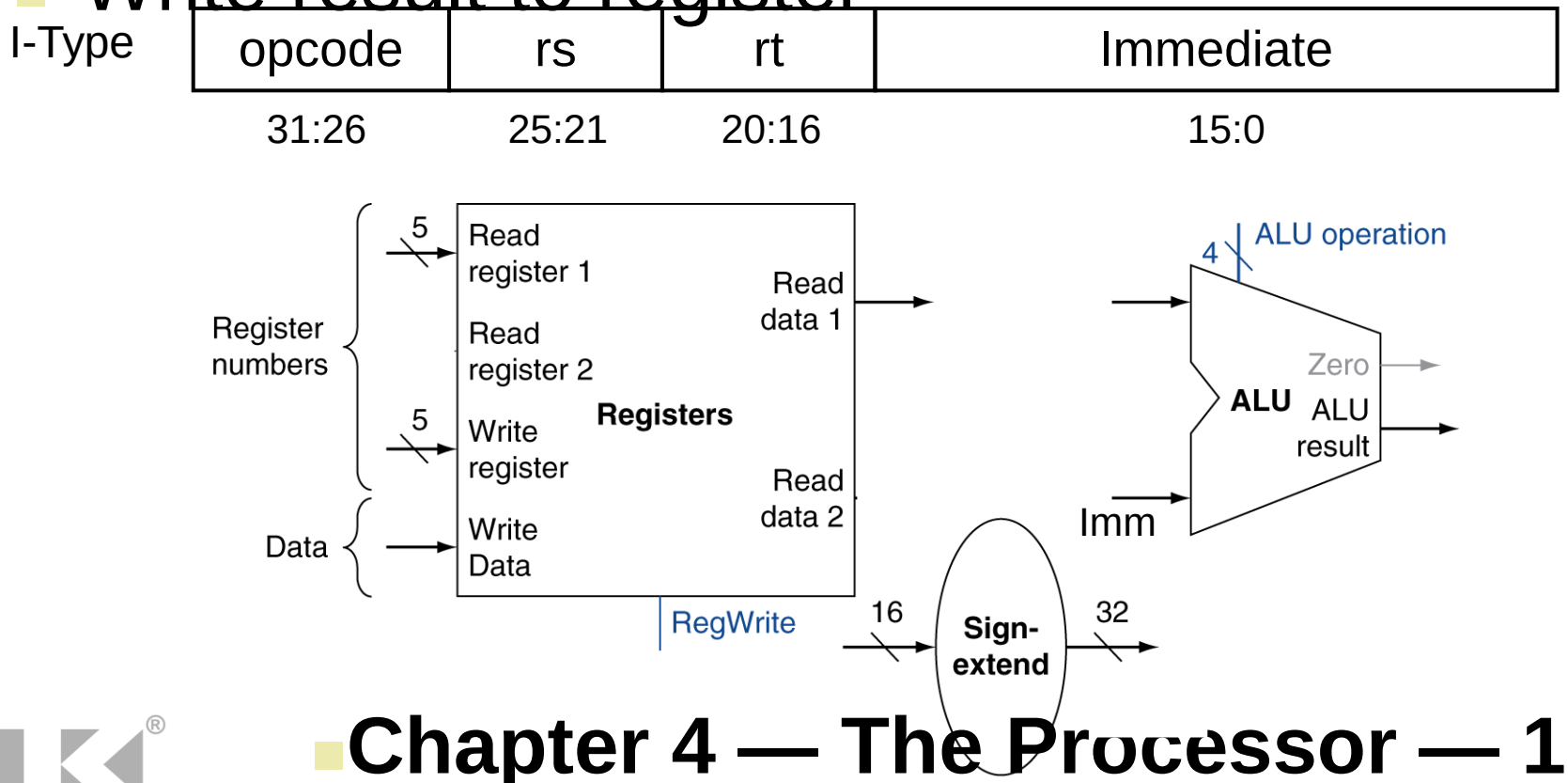
- Read two register operands
- Perform arithmetic/logical/shift operation
- Write result to register





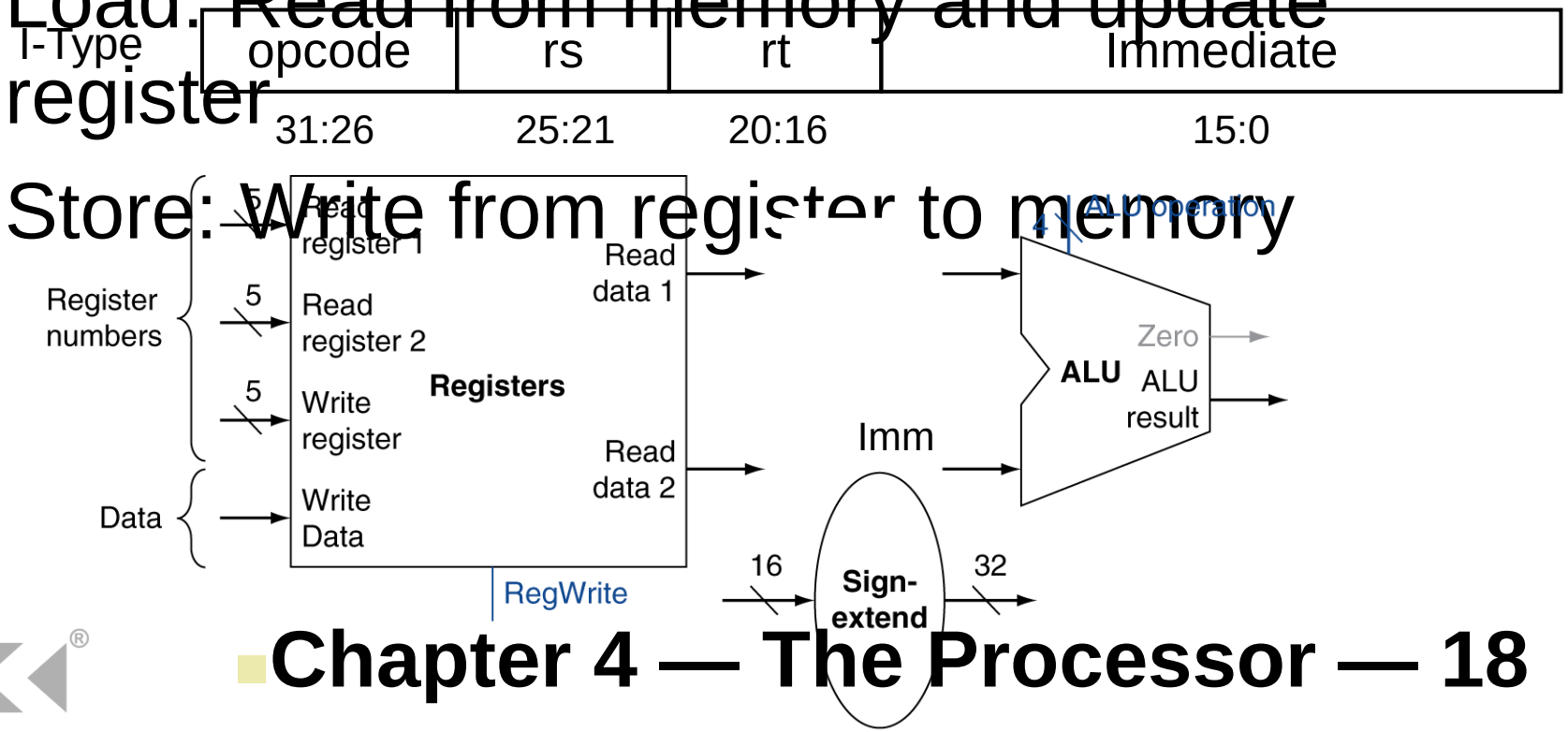
I-Type arith/logic Instructions

- Read one register operand
- Perform arithmetic/logical operation
- Write result to register



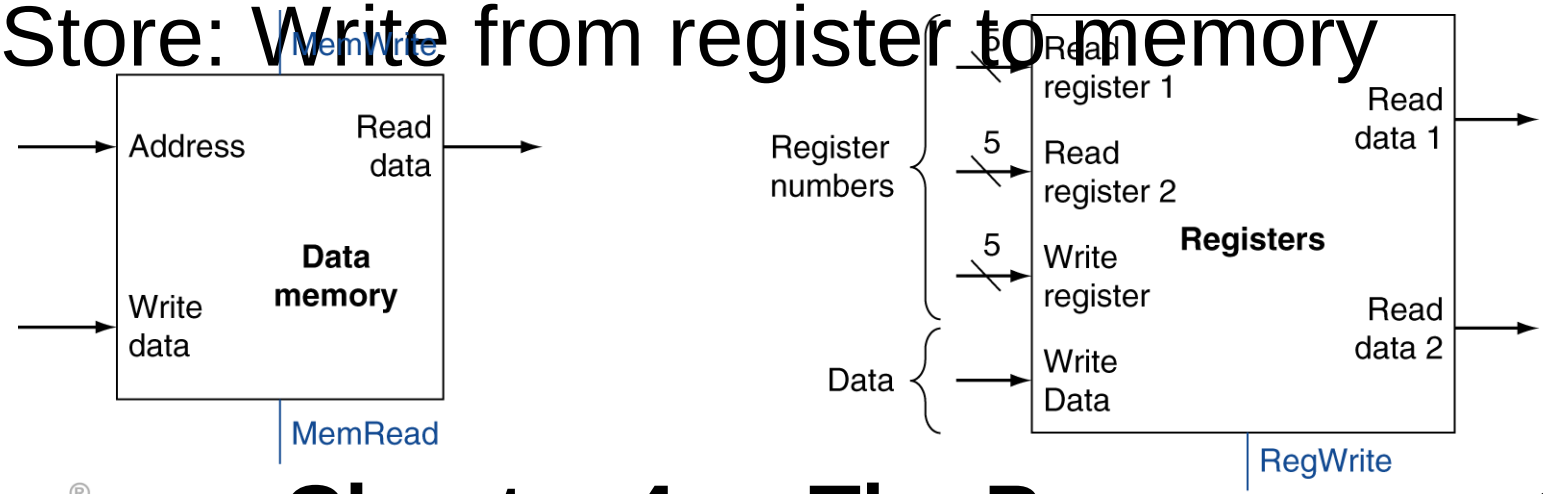
I-Type Load/Store Instructions

- Read register operand (base register)
- Calculate address using 16-bit offset
  - Use ALU, but need to sign-extend offset
- Load: Read from memory and update register
- Store: Write from register to memory



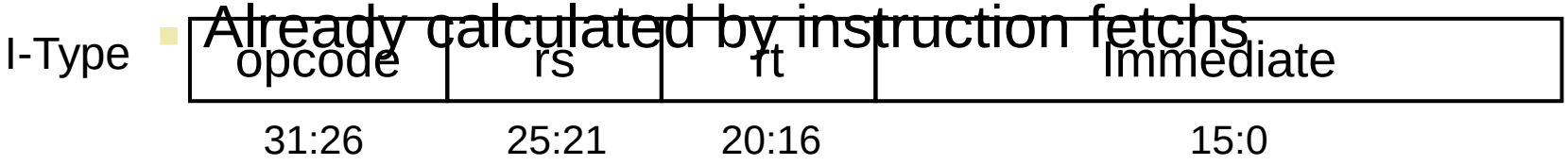
I-Type Load/Store Instructions

- Read register operand (base register)
- Calculate address using 16-bit offset
  - Use ALU, but need to sign-extend offset
- Load: Read from memory and update register
- Store: Write from register to memory

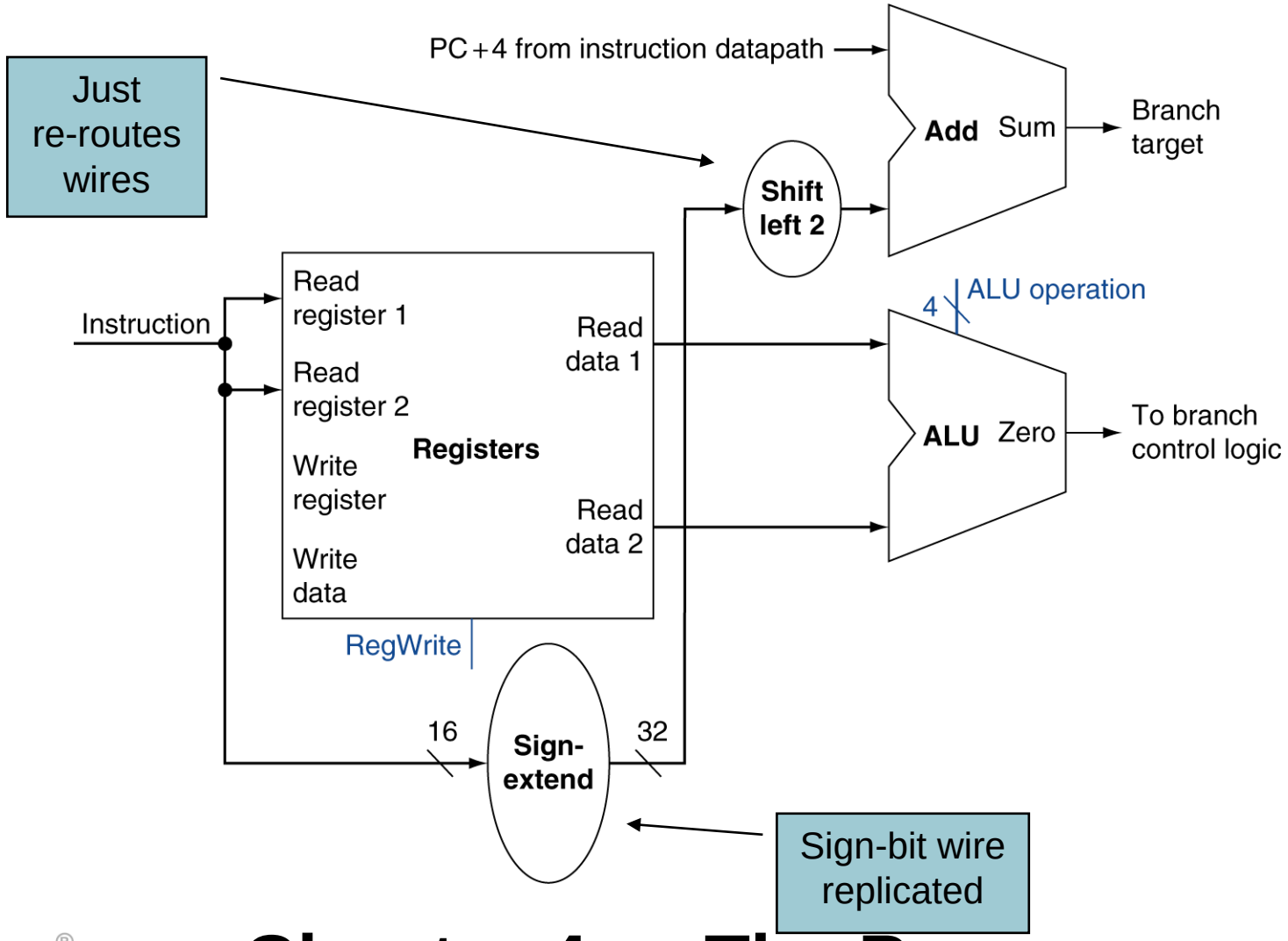


I-Type Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Compute the target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to (PC + 4)
    - Already calculated by instruction fetchs



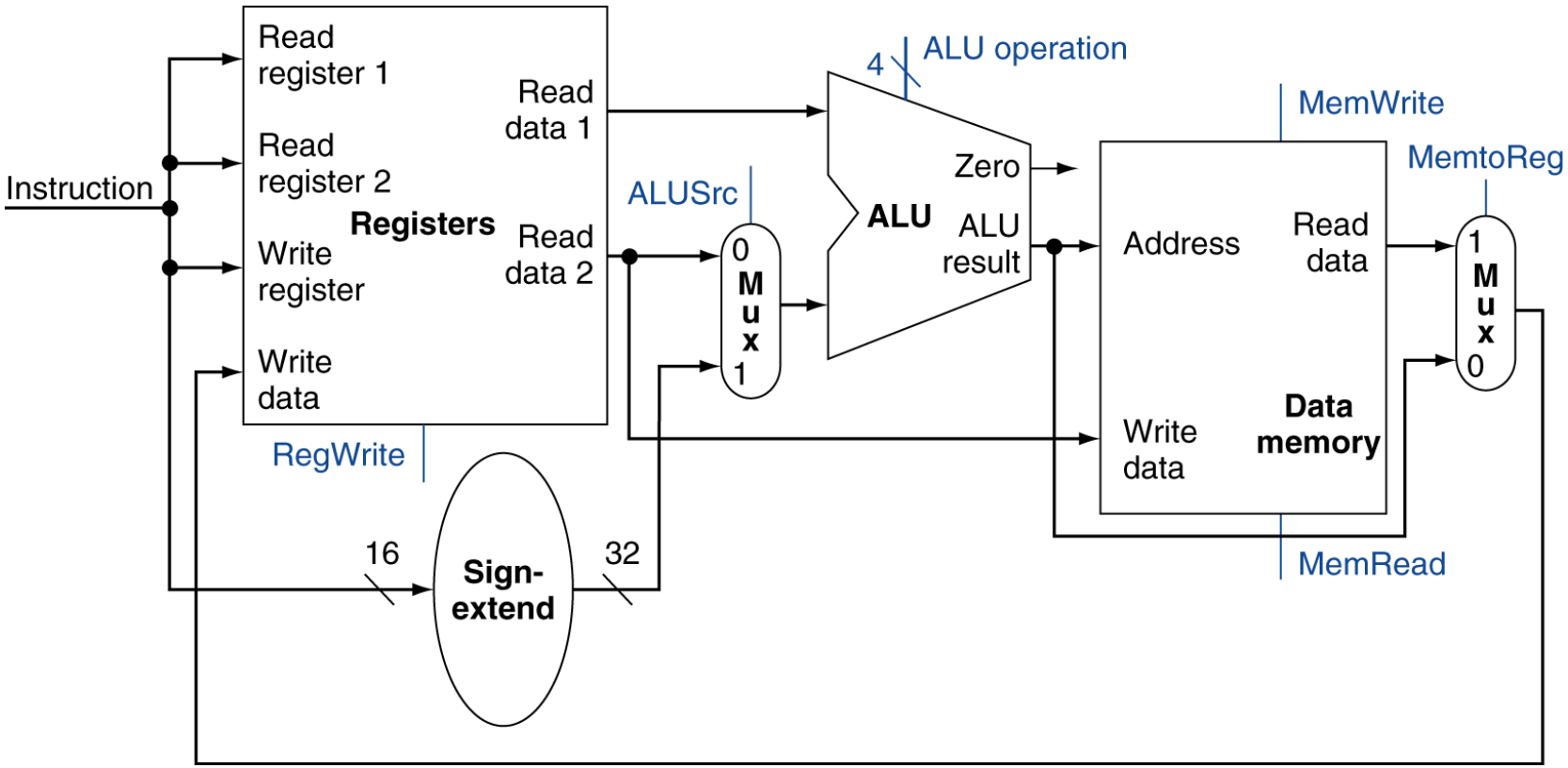
I-Type Branch Instructions



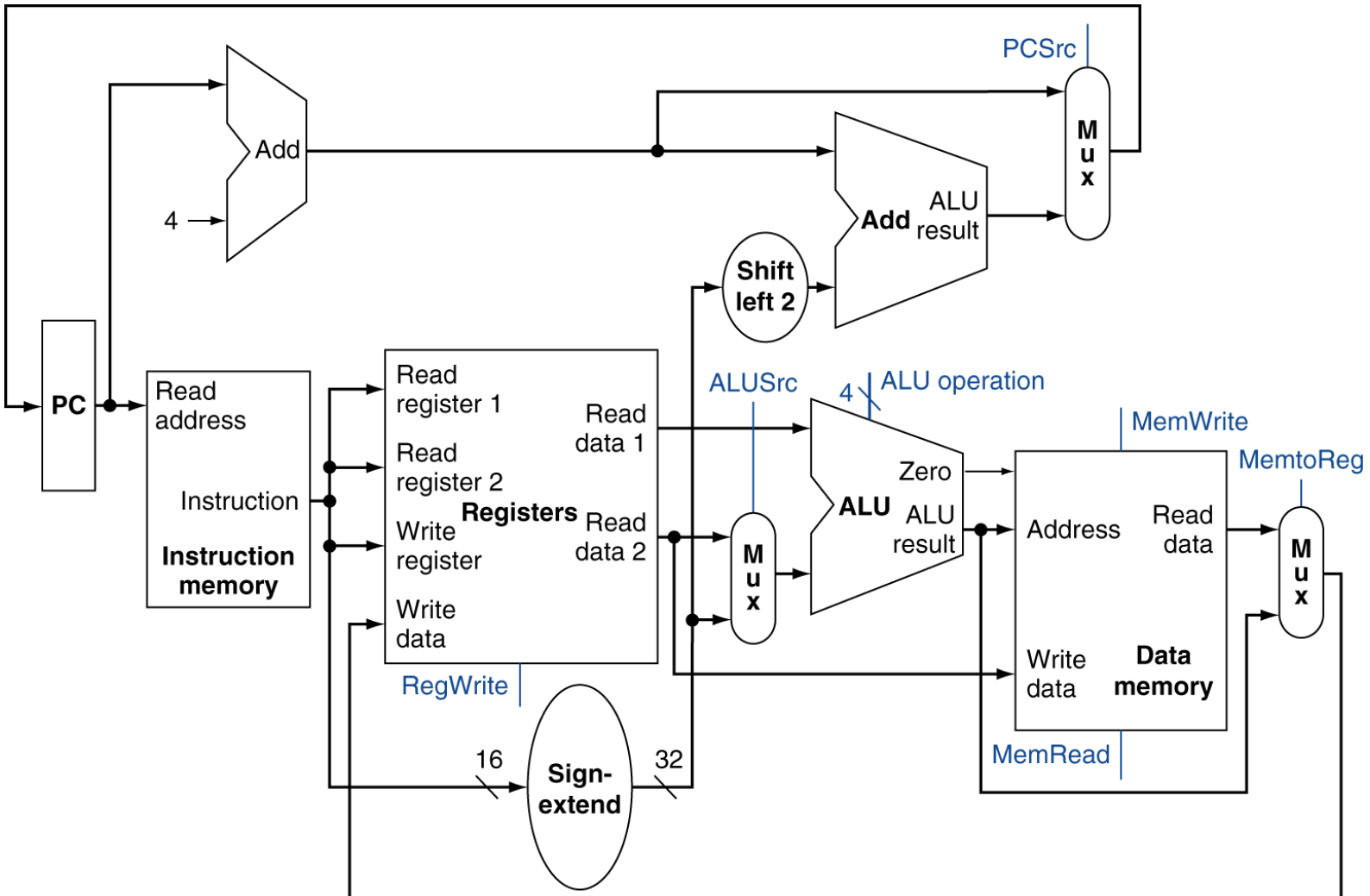
Composing the Elements

- 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

Datapath


Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
ALUSrc
0
M
u
x
1
4
ALU operation
ALU
Zero
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
1
M
u
x
0
16
Sign-extend
32

Full Datapath



ALU Control (lw/sw/beq/R-Type)

- ALU used for
  - lw / sw: Function = add
  - beq: Function = subtract
  - R-type: Function depends on “funct” field

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

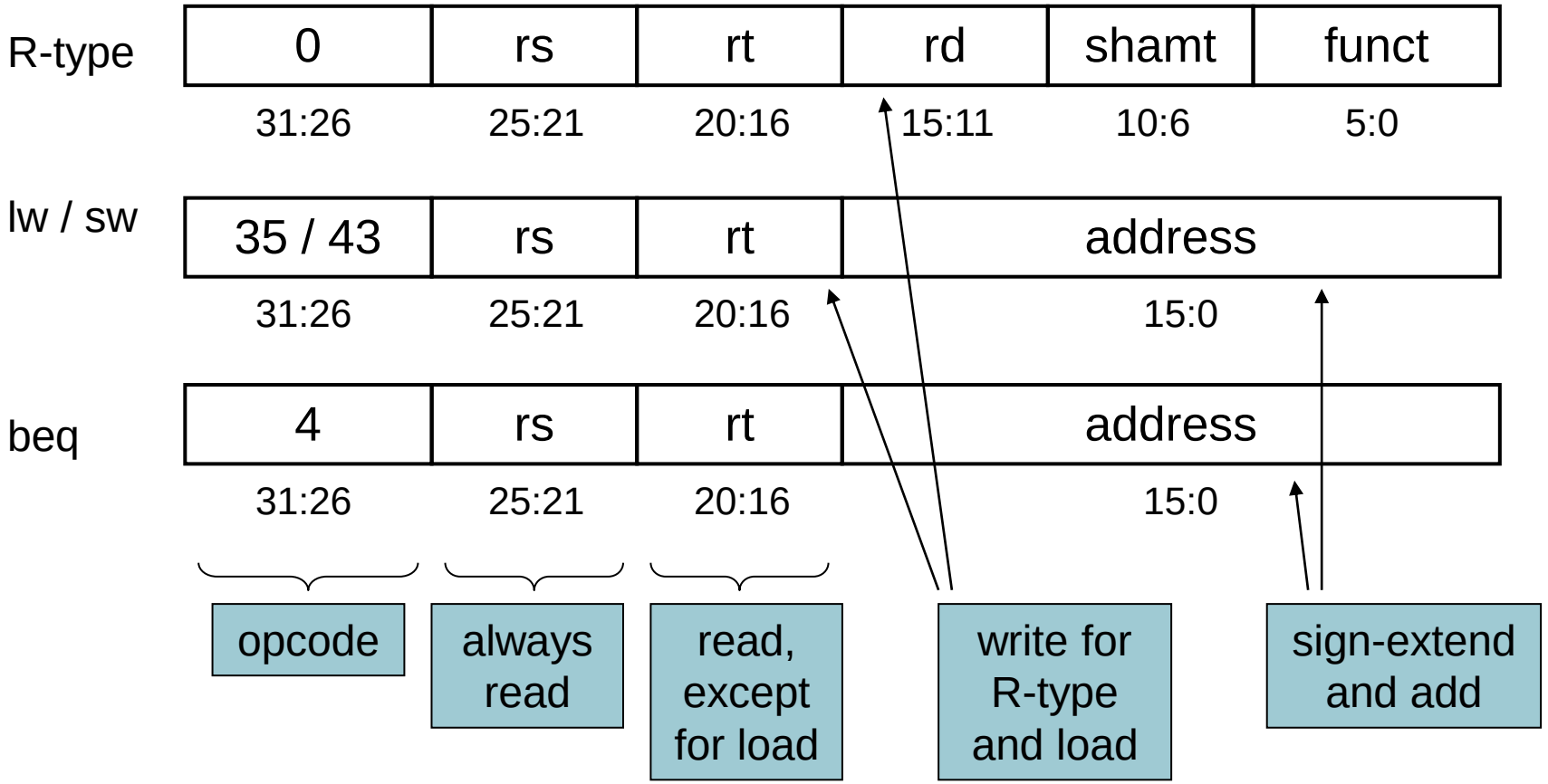
ALU Control

- 2-bit ALUOp signal derived from opcode
  - ALU control derived from ALUOp and funct
  - Try to find combinational logic functions for ALUOp and funct signals, for the instructions given below:

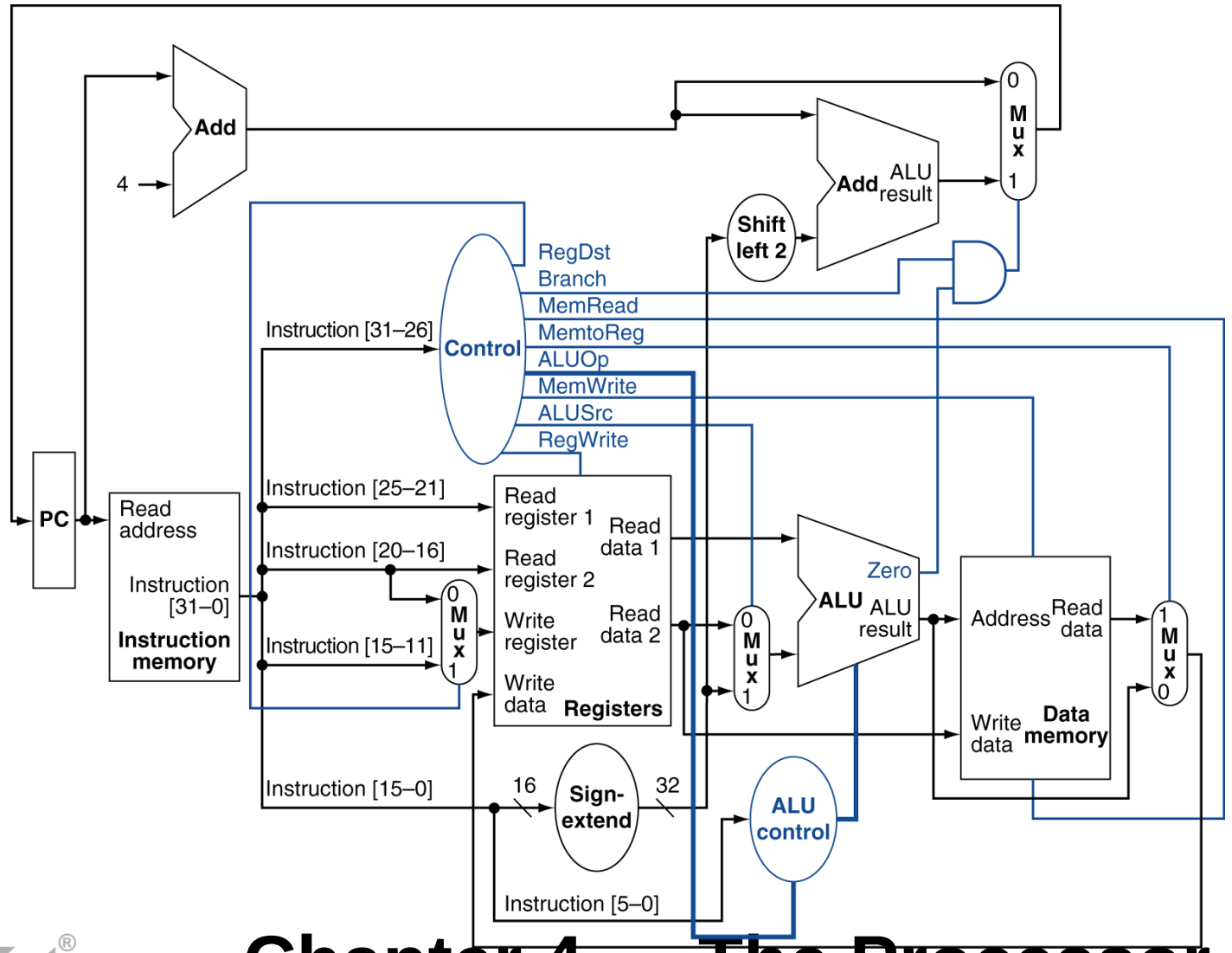
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw 100011	00	load word	XXXXXX	add	0010
sw 101011	00	store word	XXXXXX	add	0010
beq 000100	01	branch equal	XXXXXX	subtract	0110
R-type 000000	10 OR 1x	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

- Control signals derived from instruction

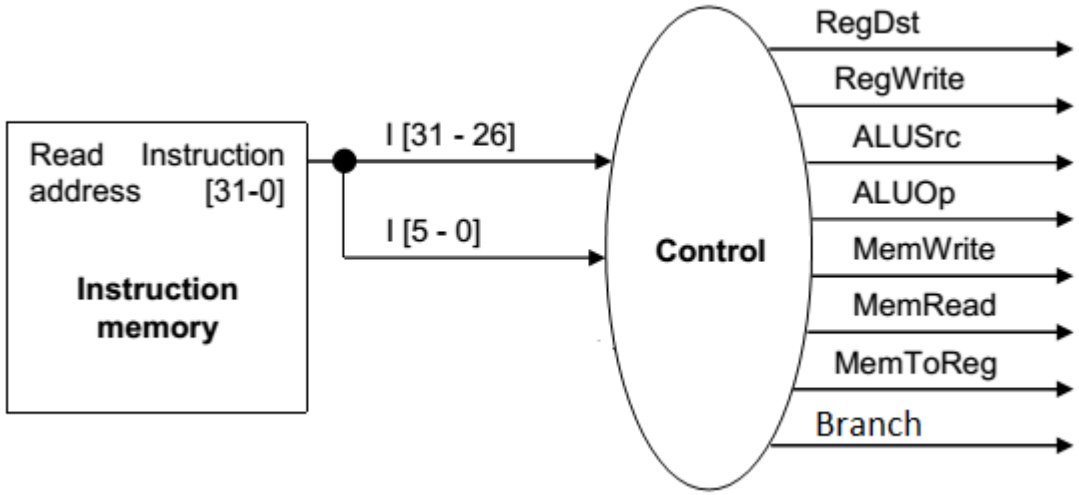


Datapath With Control



Generating Control Signals

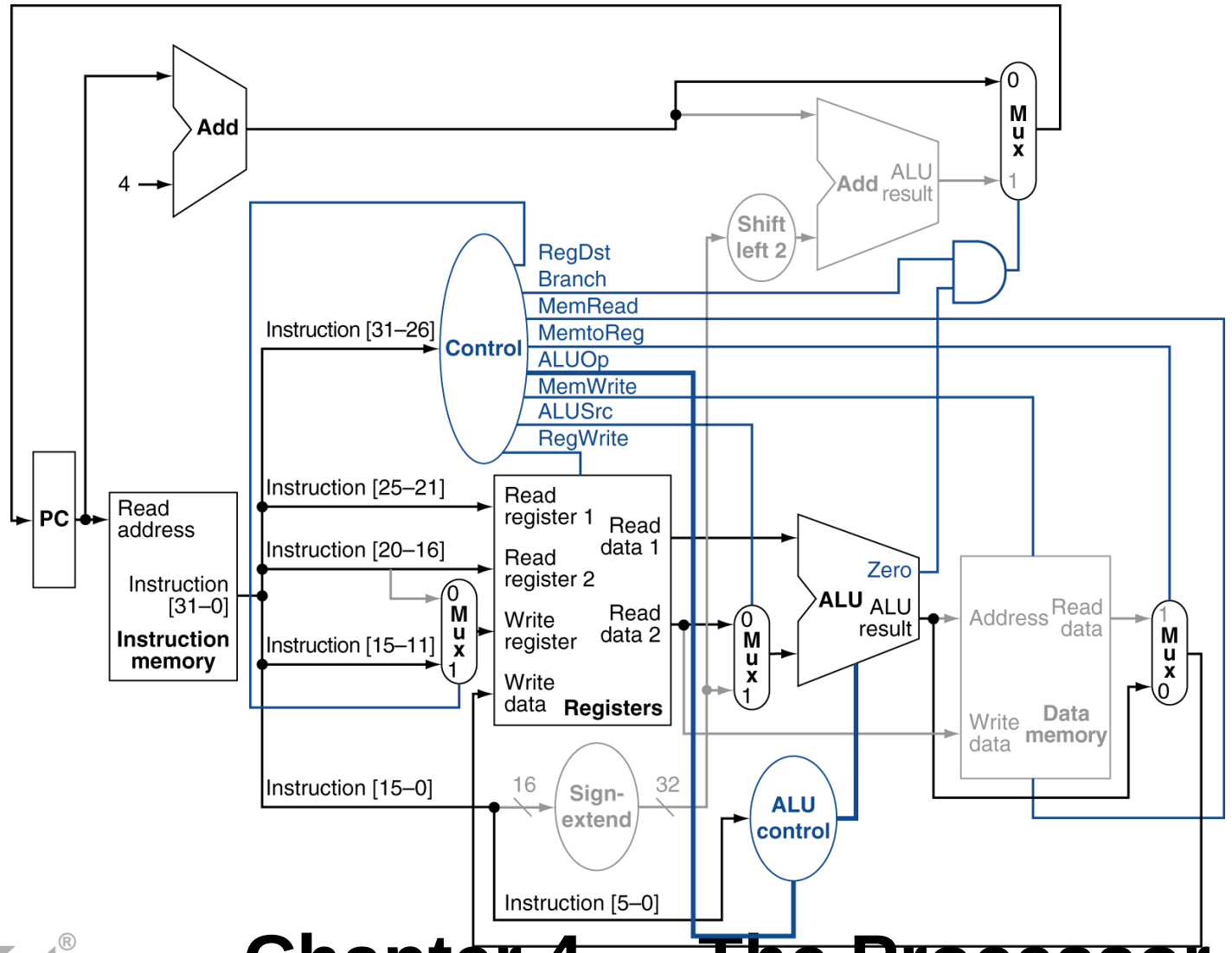
- The control unit needs 12 bits of inputs.
  - Six bits make up the instruction's opcode.
  - Six bits come from the instruction's func field.
- The control unit generates 9 bits of output, corresponding to the blue control signals mentioned on the previous slide
- You can buil
  Boolean alge
  - **Chap**



Generating Control Signals

- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

R-Type Instructions



R-Type Instructions

add, sub, and, or, …

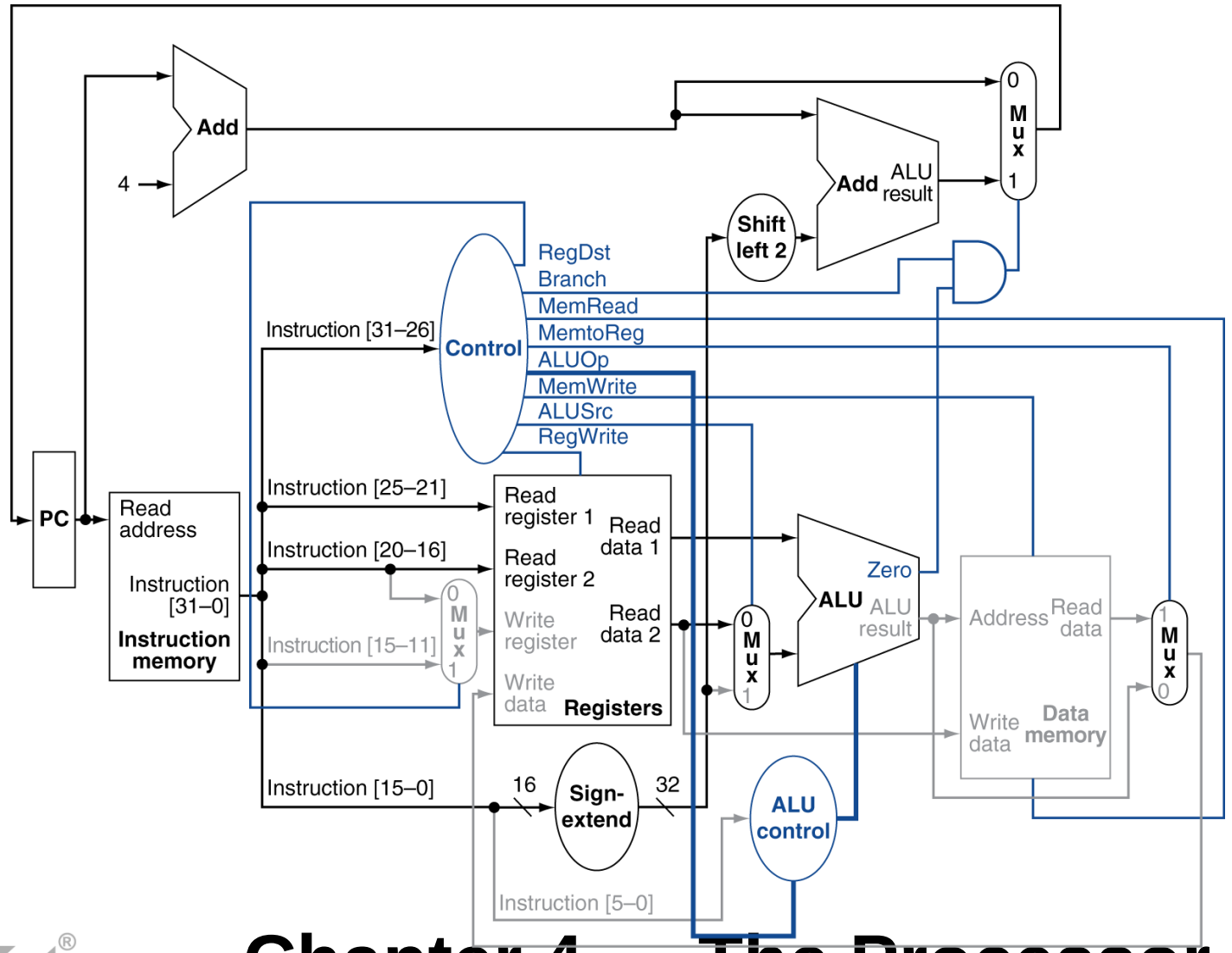
R-type	opcode 0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

`add $1, $2, $3`

add	000000	00010	00011	00001	00000	100000

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Branch-if-Equal Instruction



Branch-if-Equal Instruction

lw, sw, beq, addi, andi, …

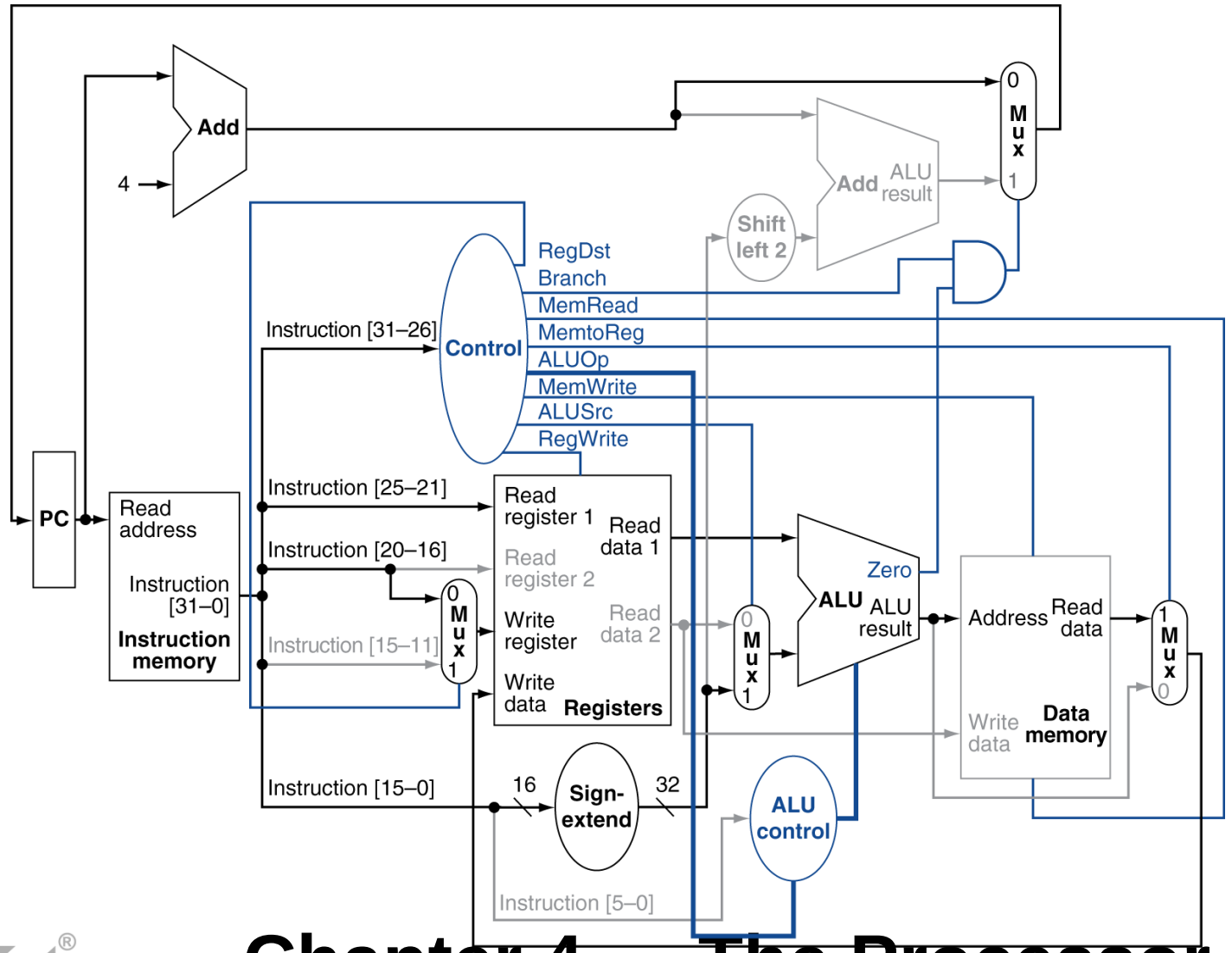
I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

beq $1, $2, 100

beq	000100	00001	00010	0000000001100100

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Load Word Instruction



Load Word Instruction

lw, sw, beq, addi, andi, …

I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

lw $8, 32($9)

lw	100011	01001	01000	0000000000100000

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Store Word Instruction

Draw the relevant data path elements and control signals which are used by a Store Word instruction (sw)

Store Word Instruction

lw, sw, beq, addi, andi, …

I-Type	opcode	rs	rt	Immediate
	31:26	25:21	20:16	15:0

sw $8, 32($9)

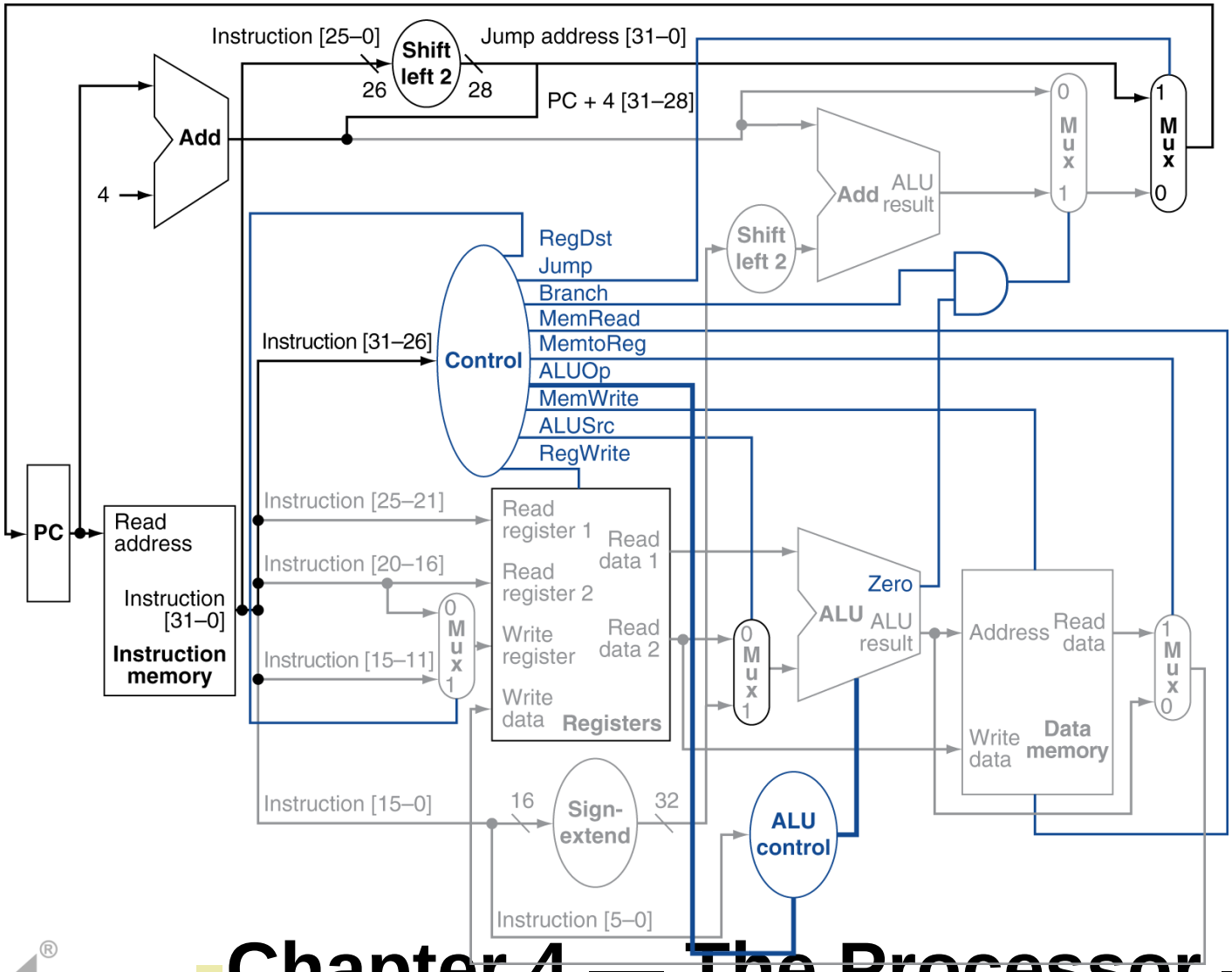
sw	100111	01001	00100	0000000001100100

RegDst	
Branch	
MemRead	
MemtoReg	
ALUOp	
MemWrite	
ALUSrc	
RegWrite	

Implementing Jumps

- Jump uses word address (size 26-bits)
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal generated using opcode

Datapath With Jumps Added



Single-Cycle Implementation

- A datapath contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our single-cycle implementation, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The control unit tells the datapath what to do, based on the instruction that's currently being executed.
  - Our processor has 9 control signals that

The Datapath & the Clock (1)

- On a positive clock edge, the PC is updated with a new address.
- A new instruction can then be loaded from memory. The control unit sets the datapath signals appropriately so that
  - registers are read,
  - ALU output is generated,
  - data memory is read, and
  - branch target addresses are computed.

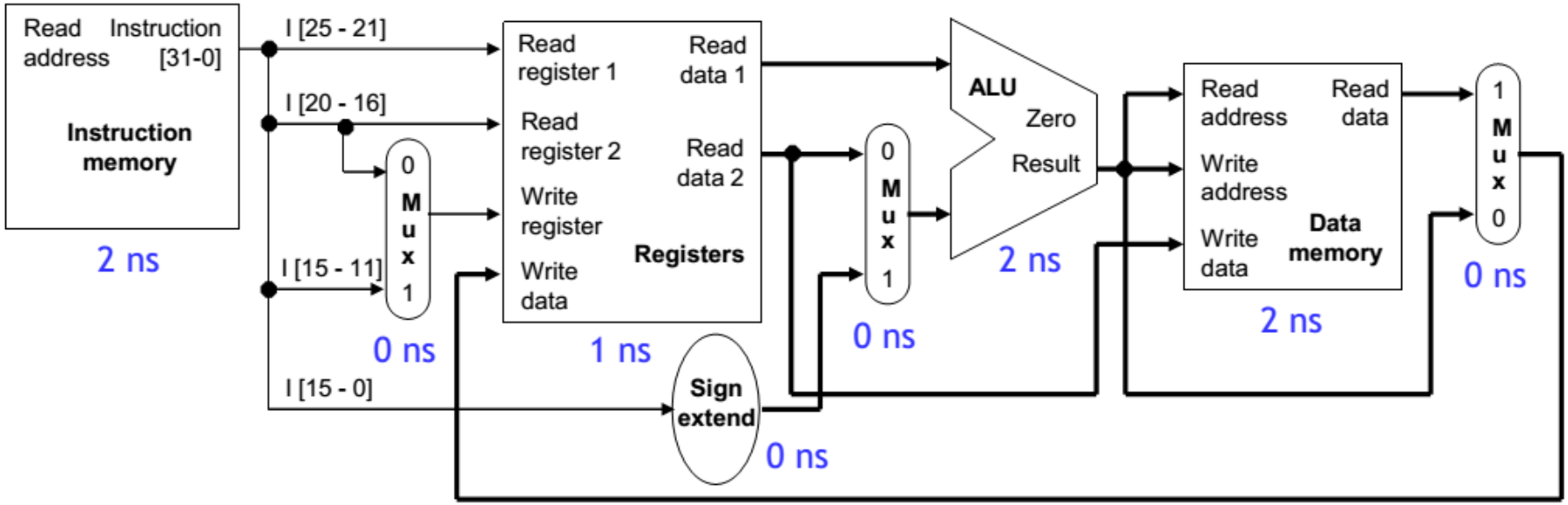
The Datapath & the Clock (2)

- Several things happen on the next positive clock edge.
  - The register file is updated for arithmetic or lw instructions.
  - Data memory is written for a sw instruction.
  - The PC is updated to point to the next instruction.

- In a single-cycle datapath everything in Step 2 must complete within one clock cycle, because the next positive clock edge

Computing the longest (critical) path

- Calculate the instruction latencies for all the instructions we have, assuming the



lw	sw	add	sub	and	or	beq	j

Average Instruction Latency

- Let's consider the gcc instruction mix as in the following table. Calculate the average instruction latency.

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

Performance Issues

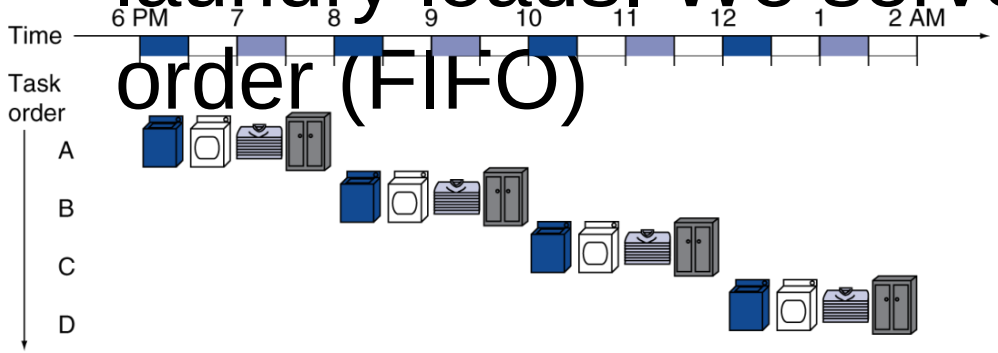
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions, in a single-cycle CPU
- Violates design principle
  - Making the common case fast
- We can improve performance with a multi-cycle implementation

Multi-Cycle CPU

- Different instructions consume different number of clock cycles
  - Arrange the datapath into “stages”
  - Fetch -> Reg Read -> ALU -> Mem Access -> Reg Write
  - Each stage to complete within one (smaller) clock cycle
  - Not all instructions will use all stages
- How to decide clock period?
  - Slowest stage determines the clock period
  - Define stages such that the latencies are equal (roughly)

Pipelining: Laundry Shop

- One employee
- Four customers (A,B,C,D) bring their laundry loads. We serve them on arrival order (FIFO)

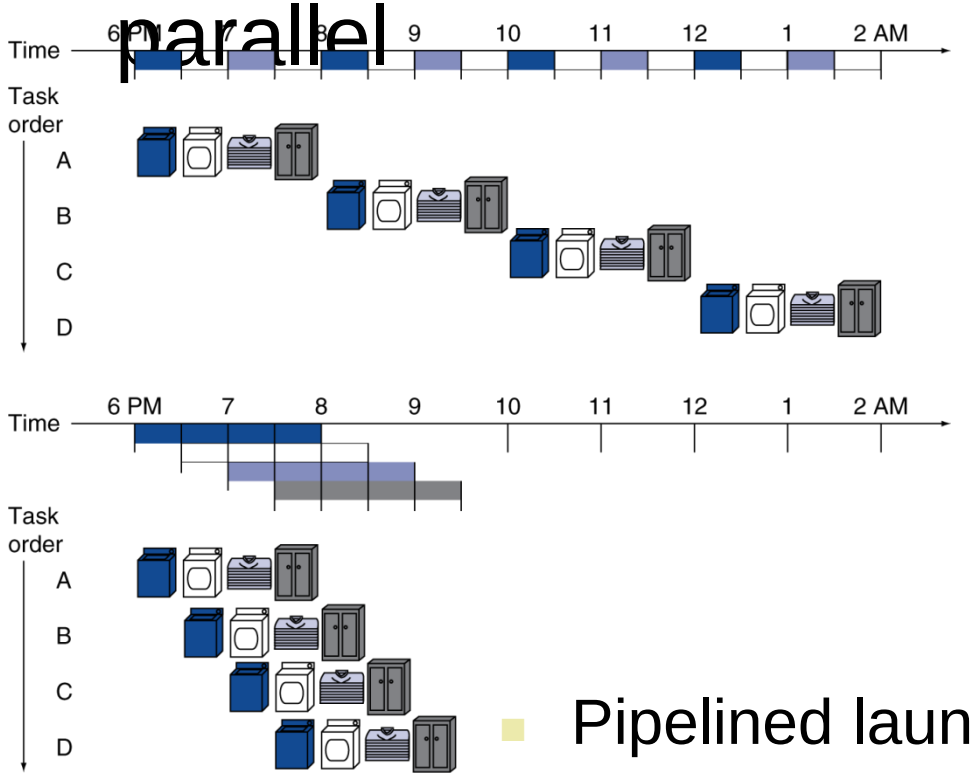


- 2 hours per customer
- What if some customer doesn't need folding and/or packaging?

- One washing machine (30 mins)
- One dryer (30 mins)
- One folding station (30 mins)
- One packaging station (30 mins)

Pipelining: Laundry Shop

- We hire more employees, do work in parallel



- Four loads:
  - Speedup = 8/3.5 = 2.3
- Steady-state:
  - One job finished every 30 mins
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

- Pipelined laundry: overlapping execution
  - Parallelism improves performance
  - Instruction Level Parallelism (ILP)

MIPS Pipeline

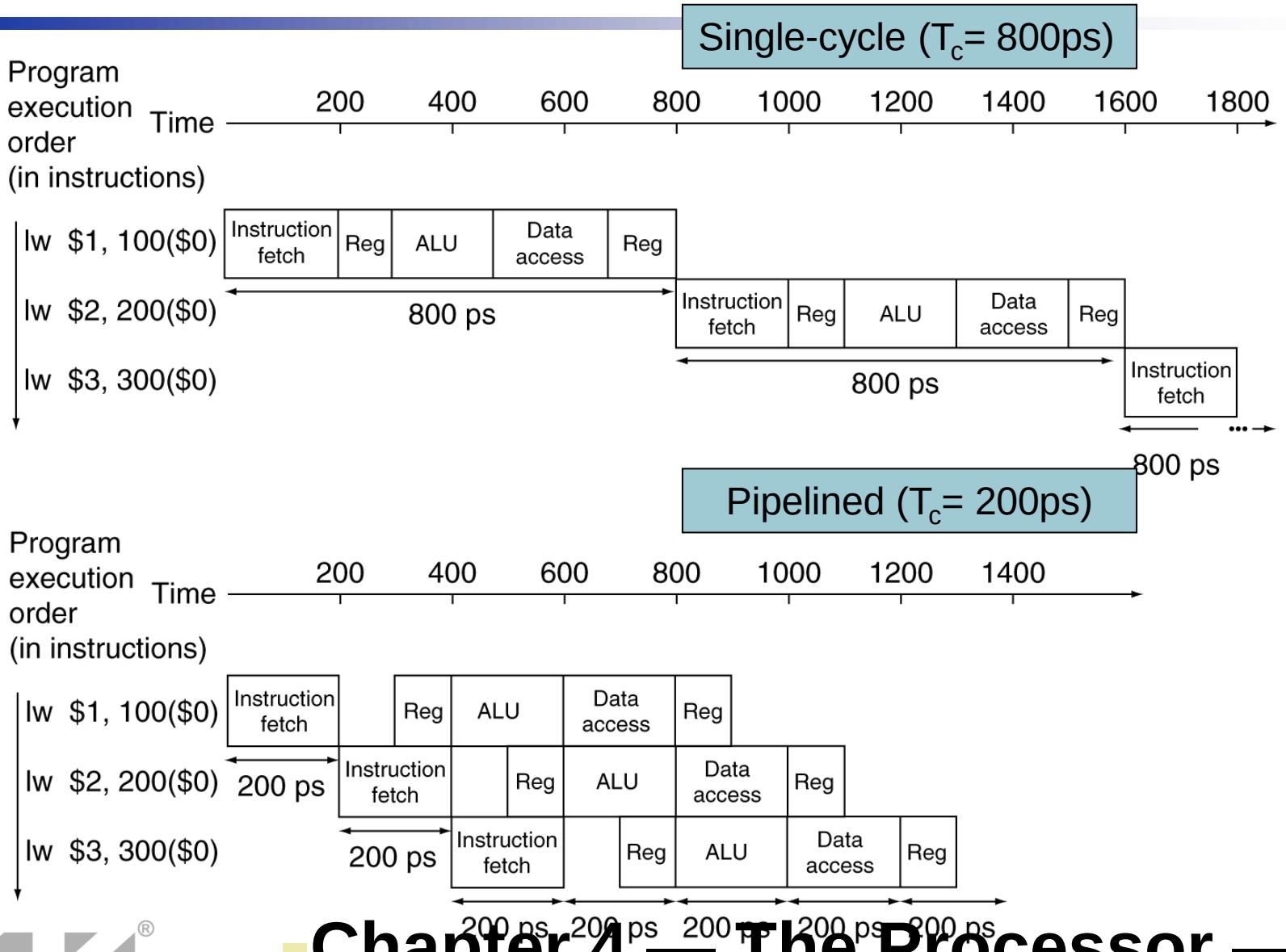
- Five stages, one step per stage
  - IF: Instruction fetch from memory using PC
  - ID: Instruction decode & register read
  - EX: Execute operation or calculate address
  - MEM: Access memory
  - WB: Write result back to register

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$_{pipelined}$ = Time between instructions$_{nonpipelined}$ / Number of stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (absolute time for each instruction) does not decrease

Pipelining and ISA Design

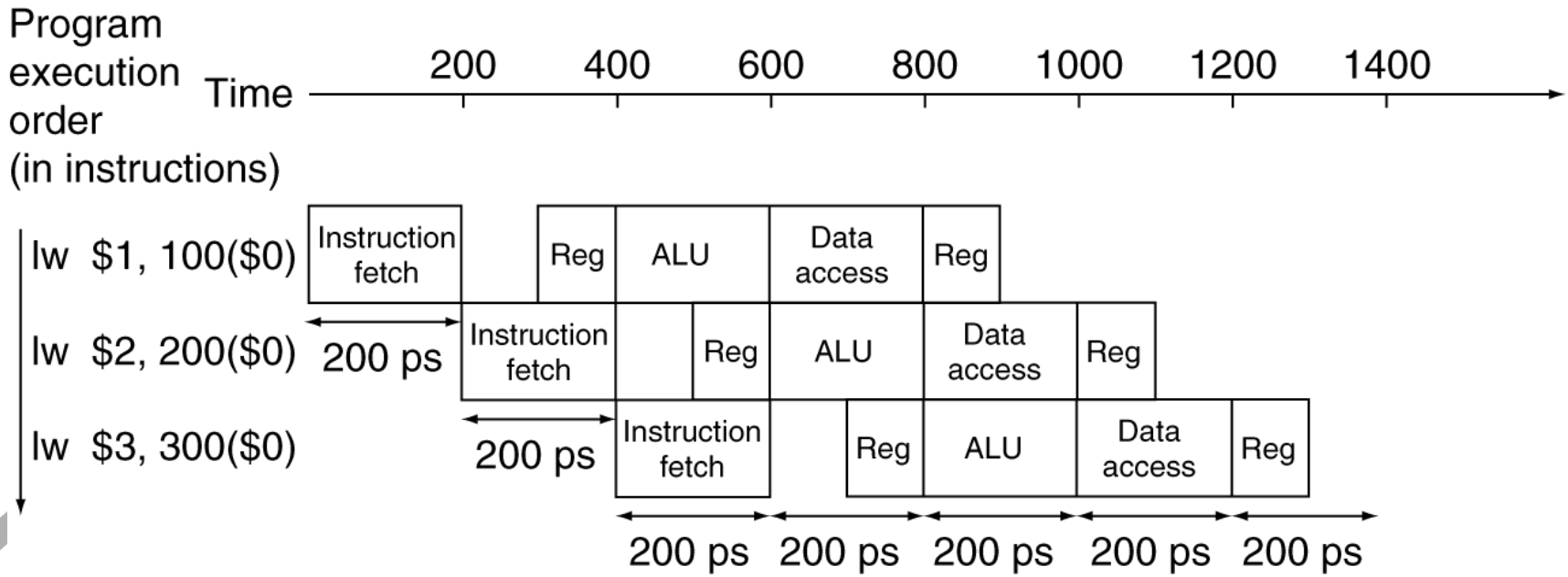
- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - ALU operation decided separately
    - Arithmetic, Logic and Shifting
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage

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 result of previous instruction

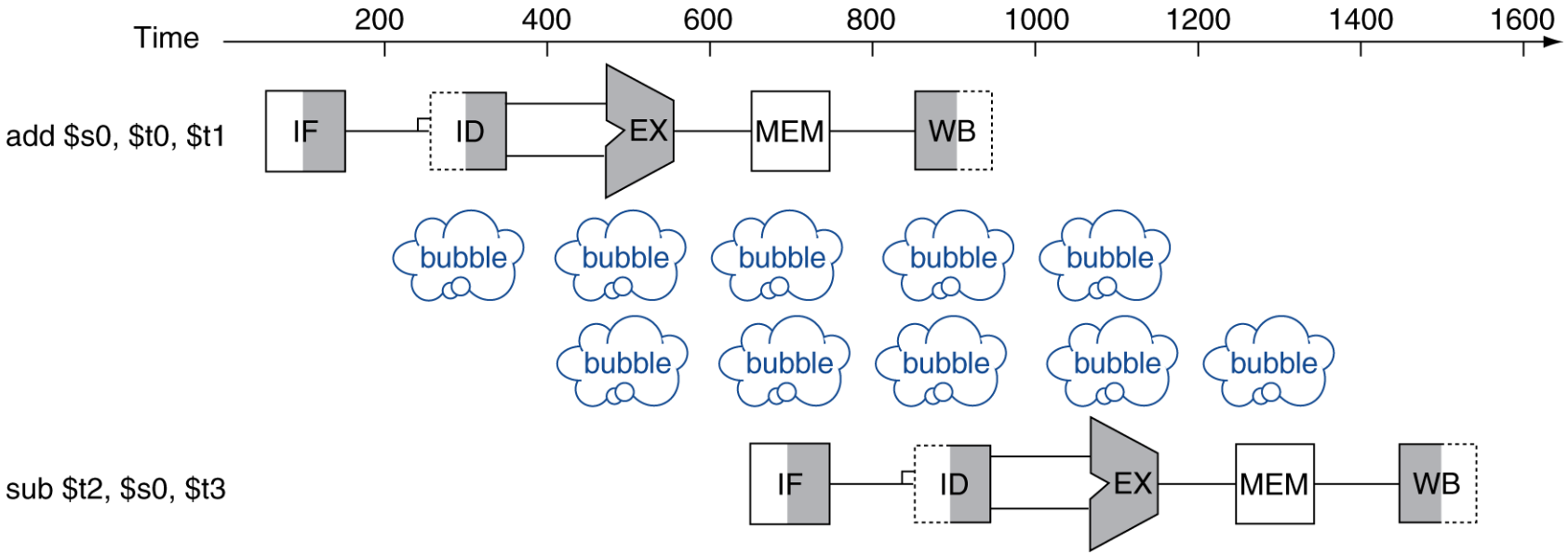
Structure Hazards

- 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



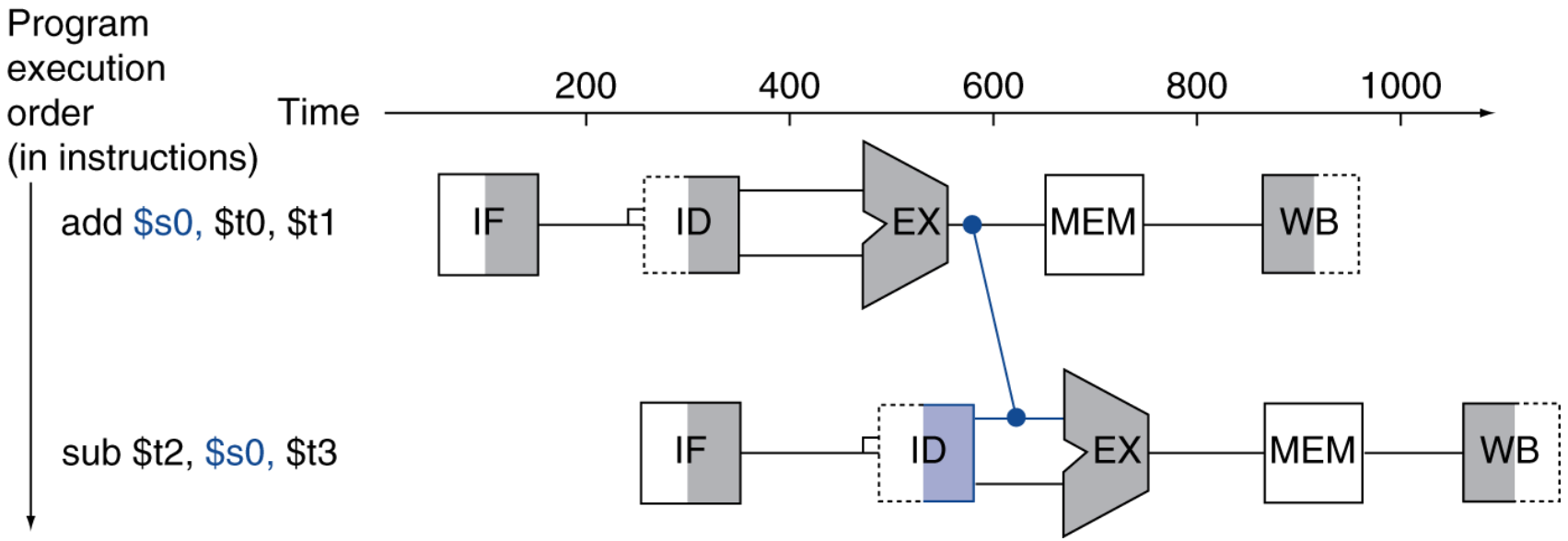
Data Hazards

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



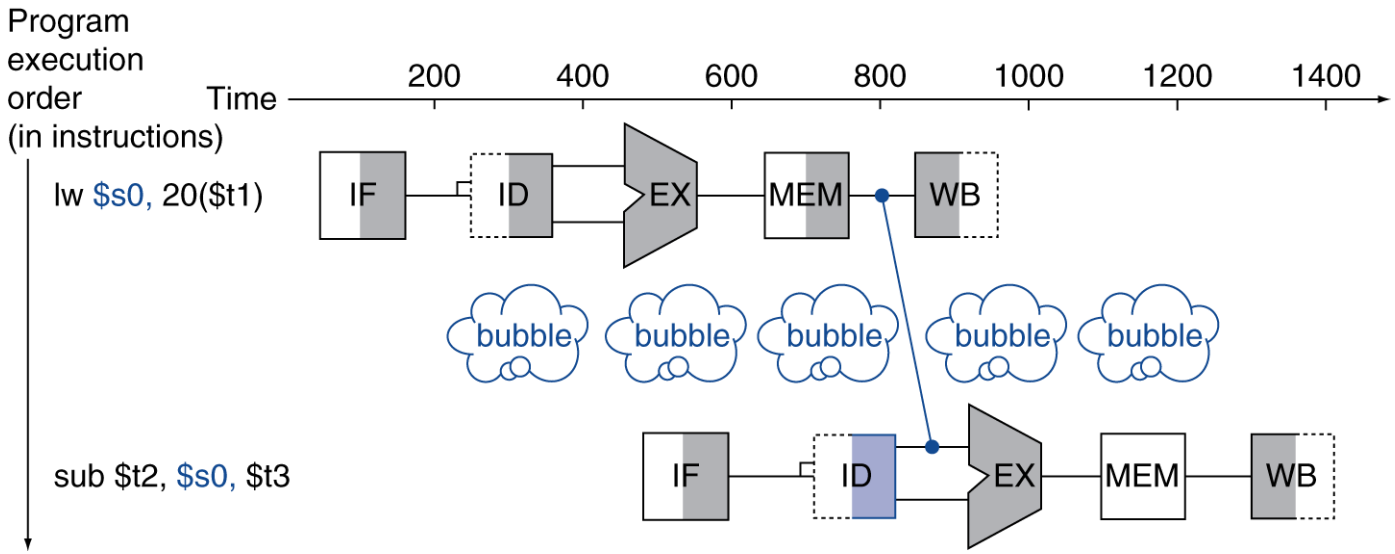
Forwarding (aka Bypassing)

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



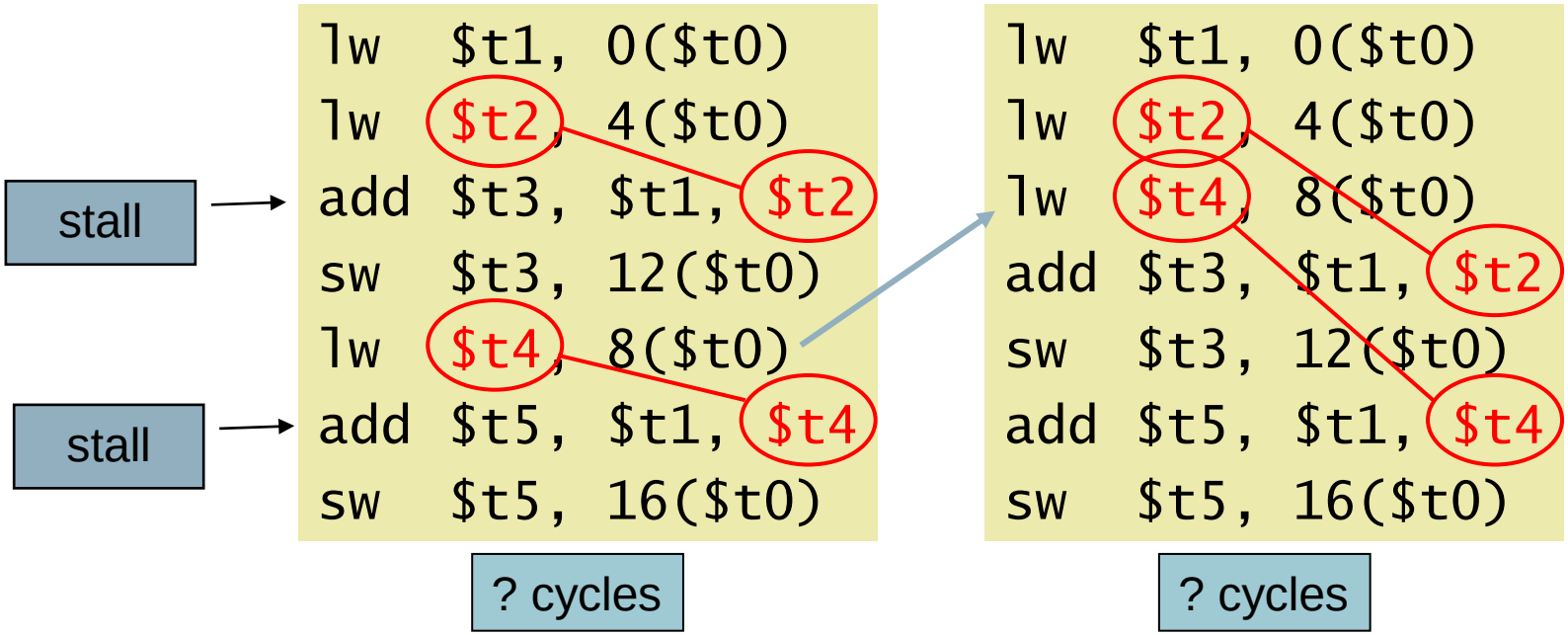
Load-Use Data Hazard

- 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;

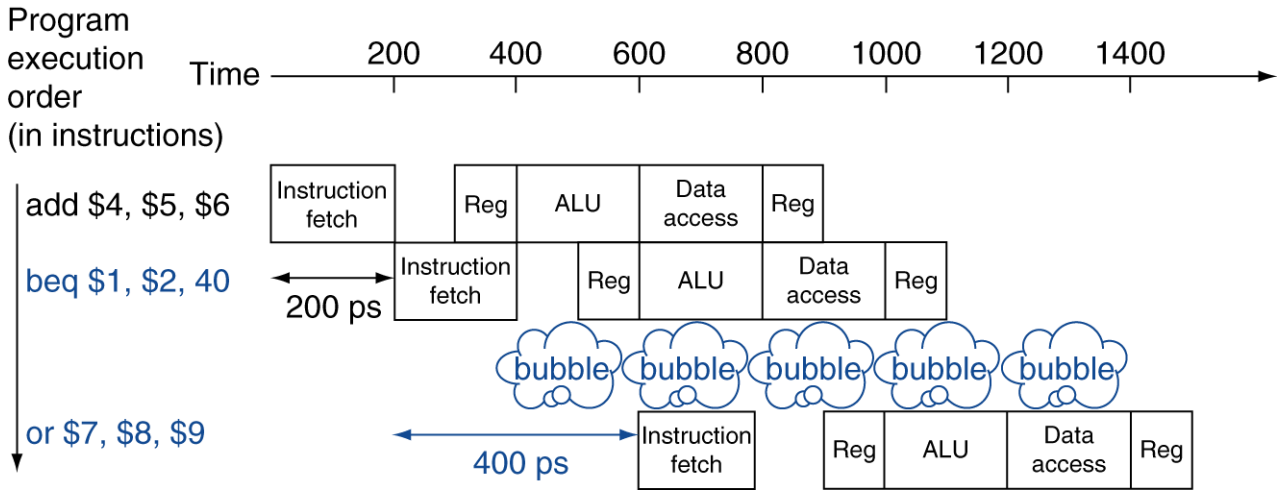


Control Hazards

- 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

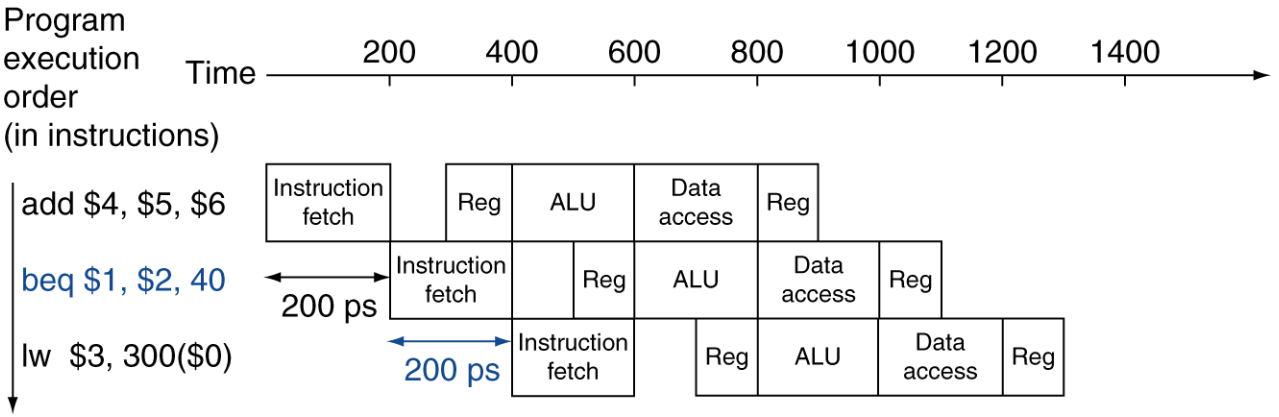


Branch Prediction

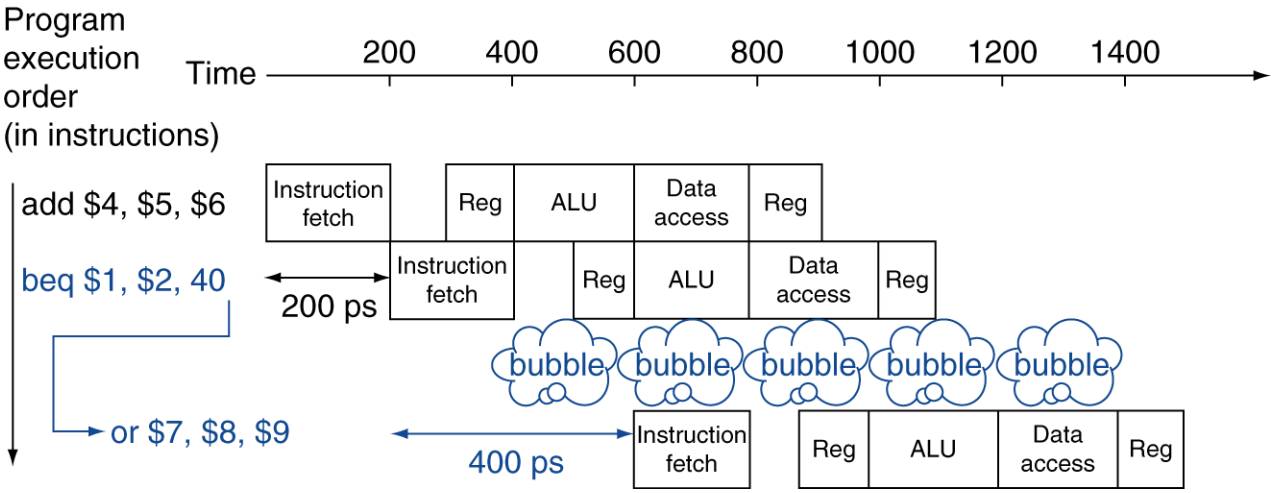
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict that branches are usually not taken
  - Fetch instruction after branch, without stalling

MIPS with Predict Not Taken

Prediction correct



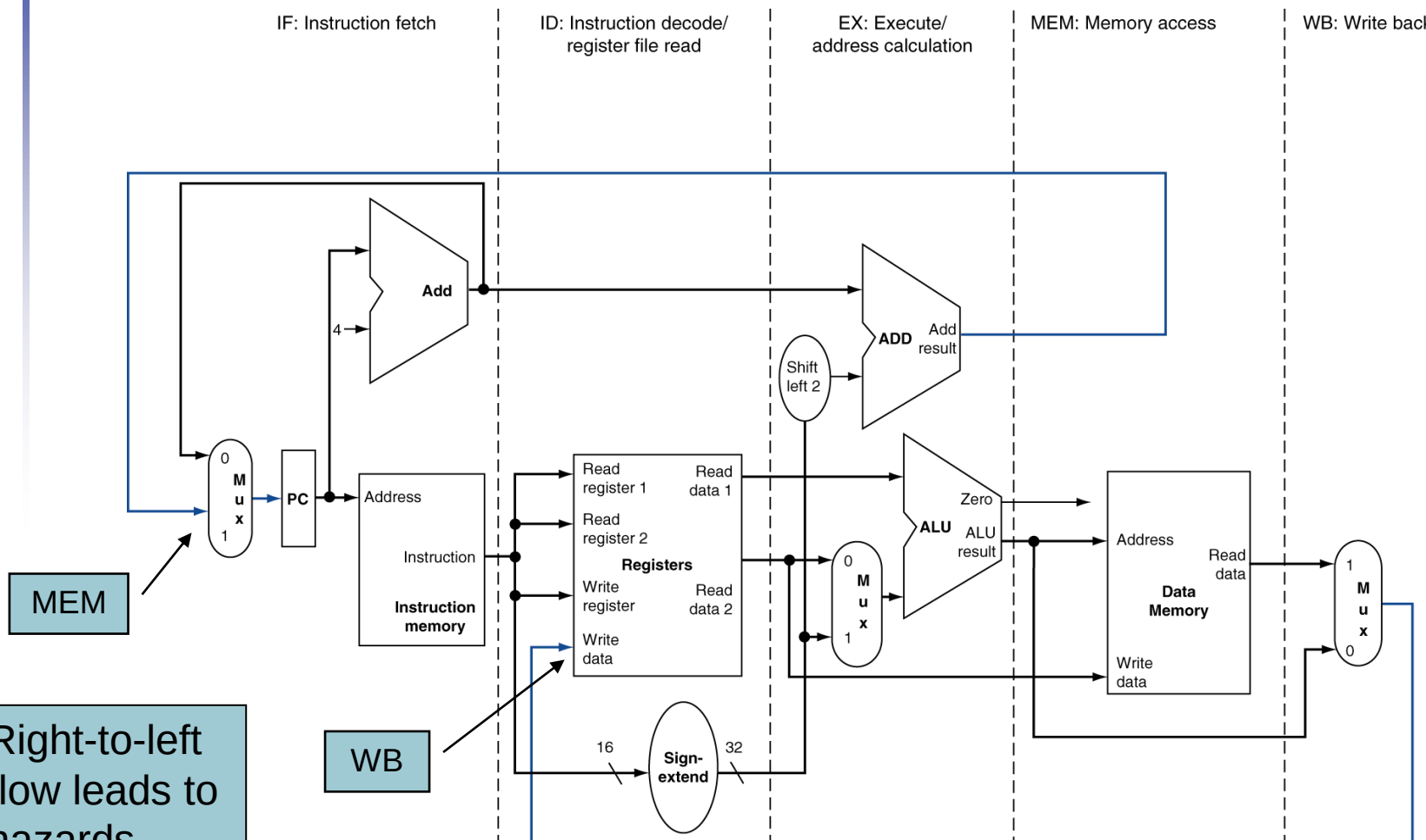
Prediction incorrect



More Realistic Branch Prediction

- Static branch prediction
  - Fixed prediction (like we just saw)
  - Or based on typical branch behaviors
  - 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

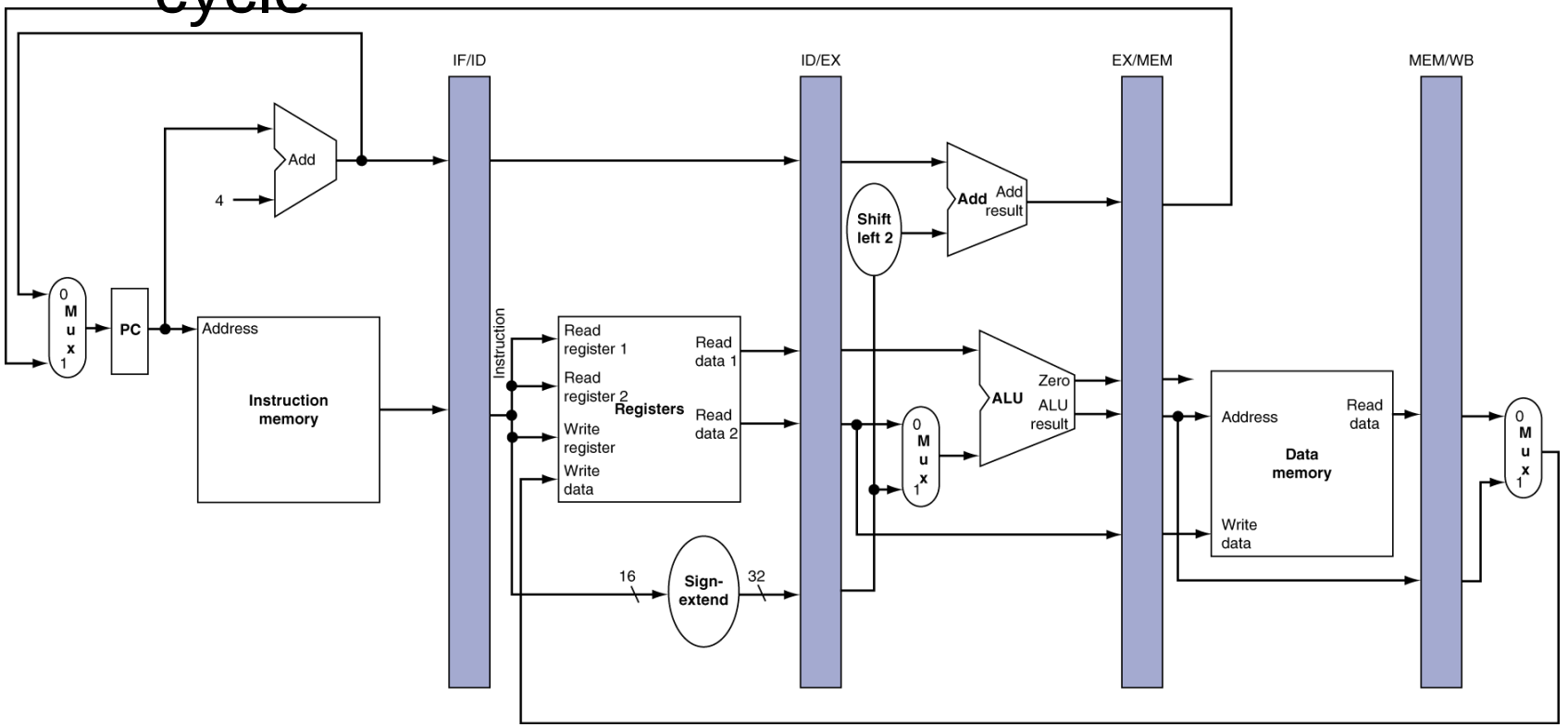
MIPS Pipelined Datapath



Right-to-left flow leads to hazards

Pipeline registers

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

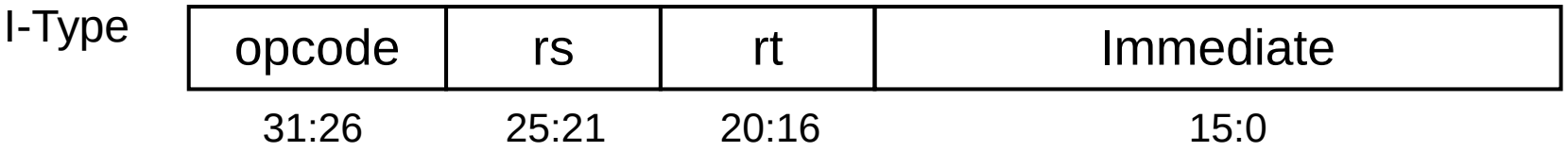


Pipeline Operation

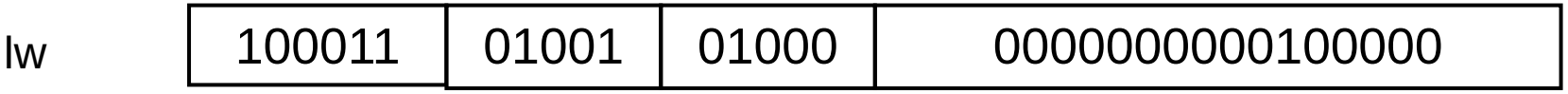
- Cycle-by-cycle flow of instructions through the pipelined data-path
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams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

Recall Load & Store Encoding

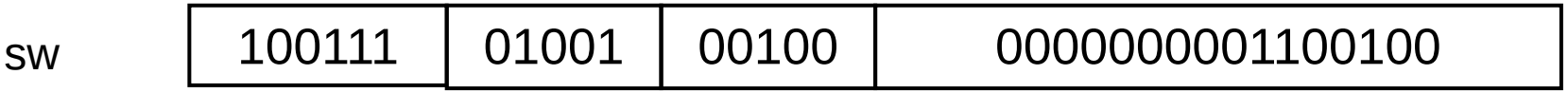
lw, sw, beq, addi, andi, …



lw $8, 32($9)

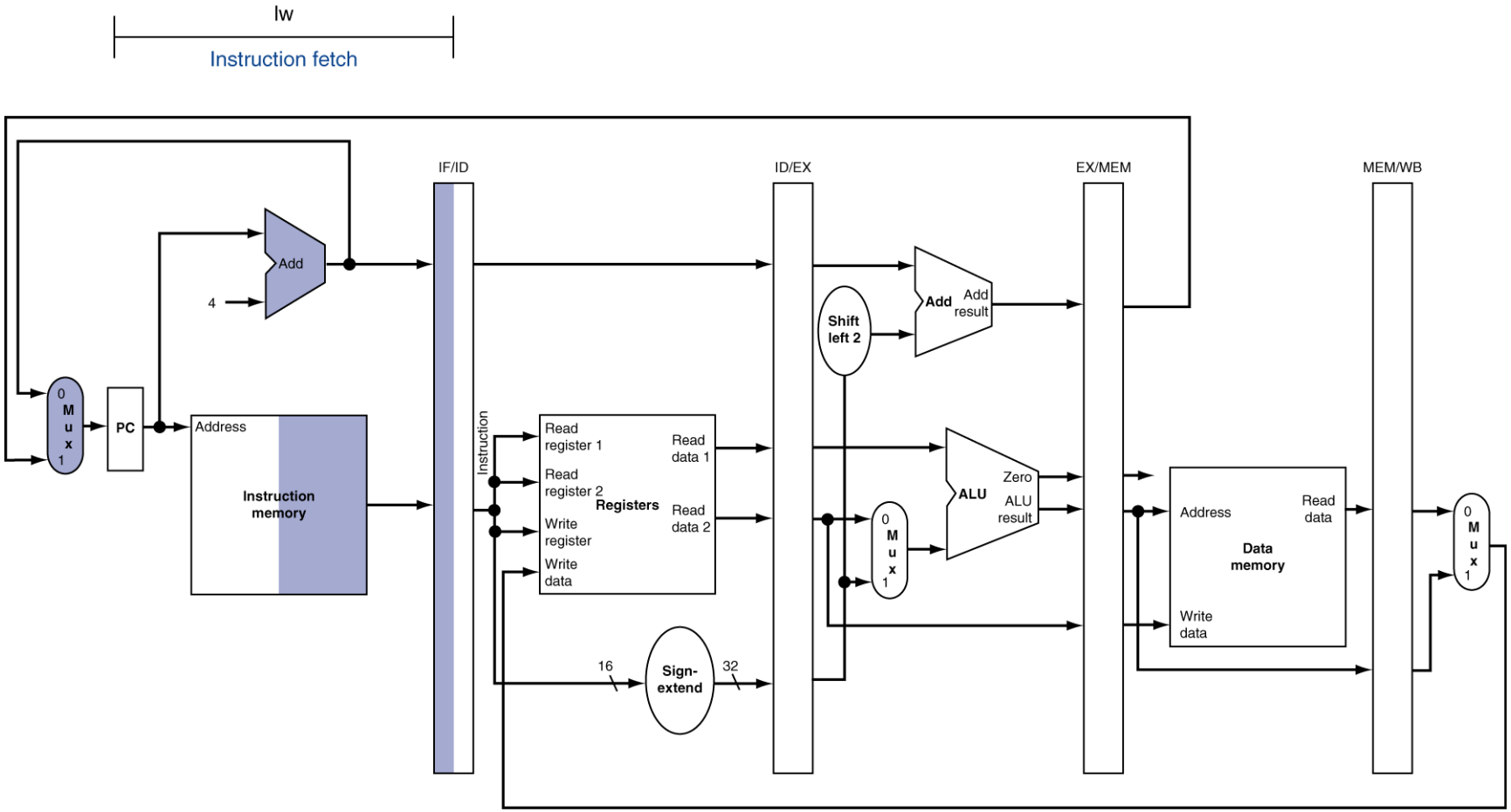


sw $8, 32($9)

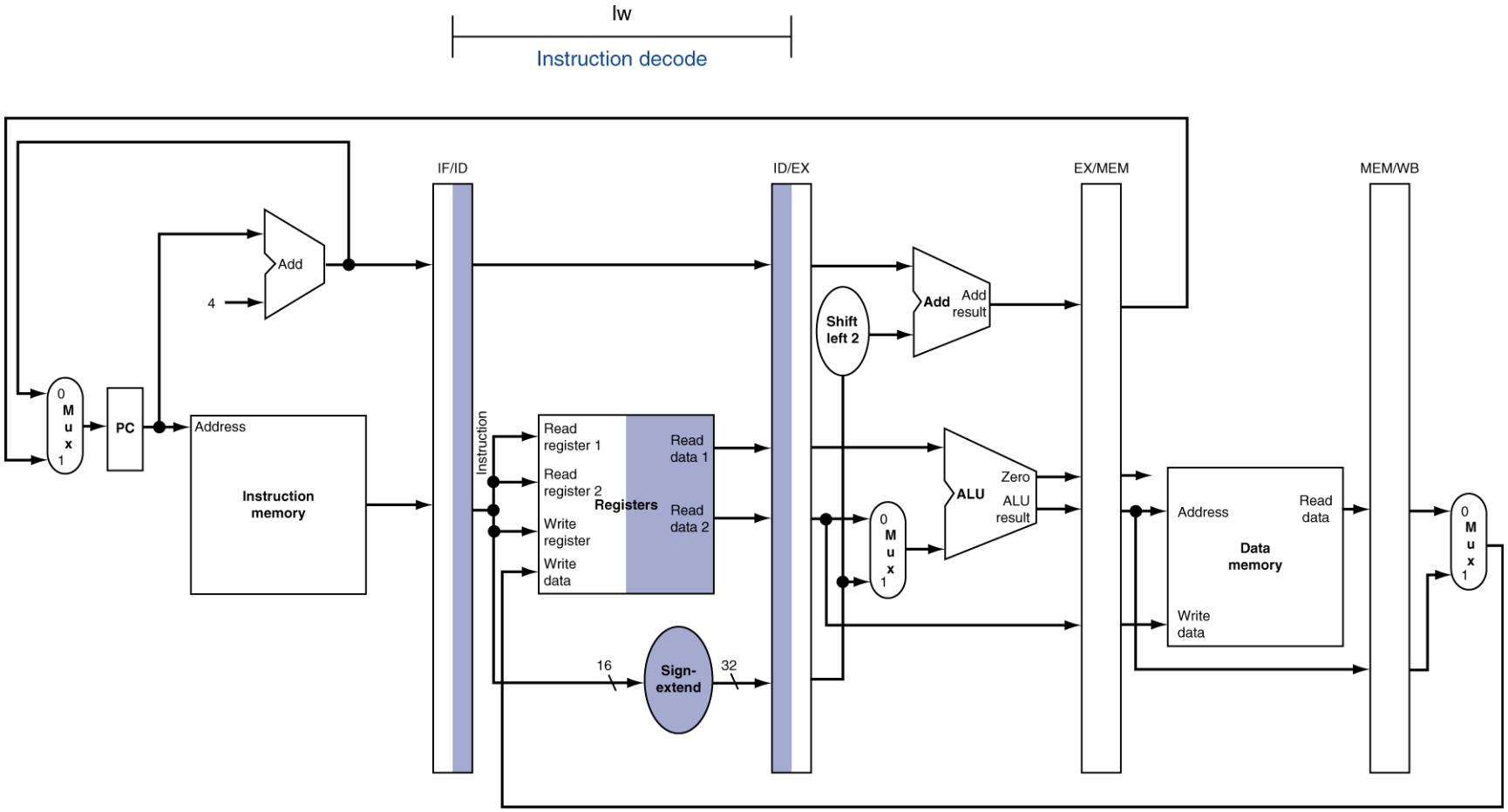


Chapter 4 — The Processor — 69

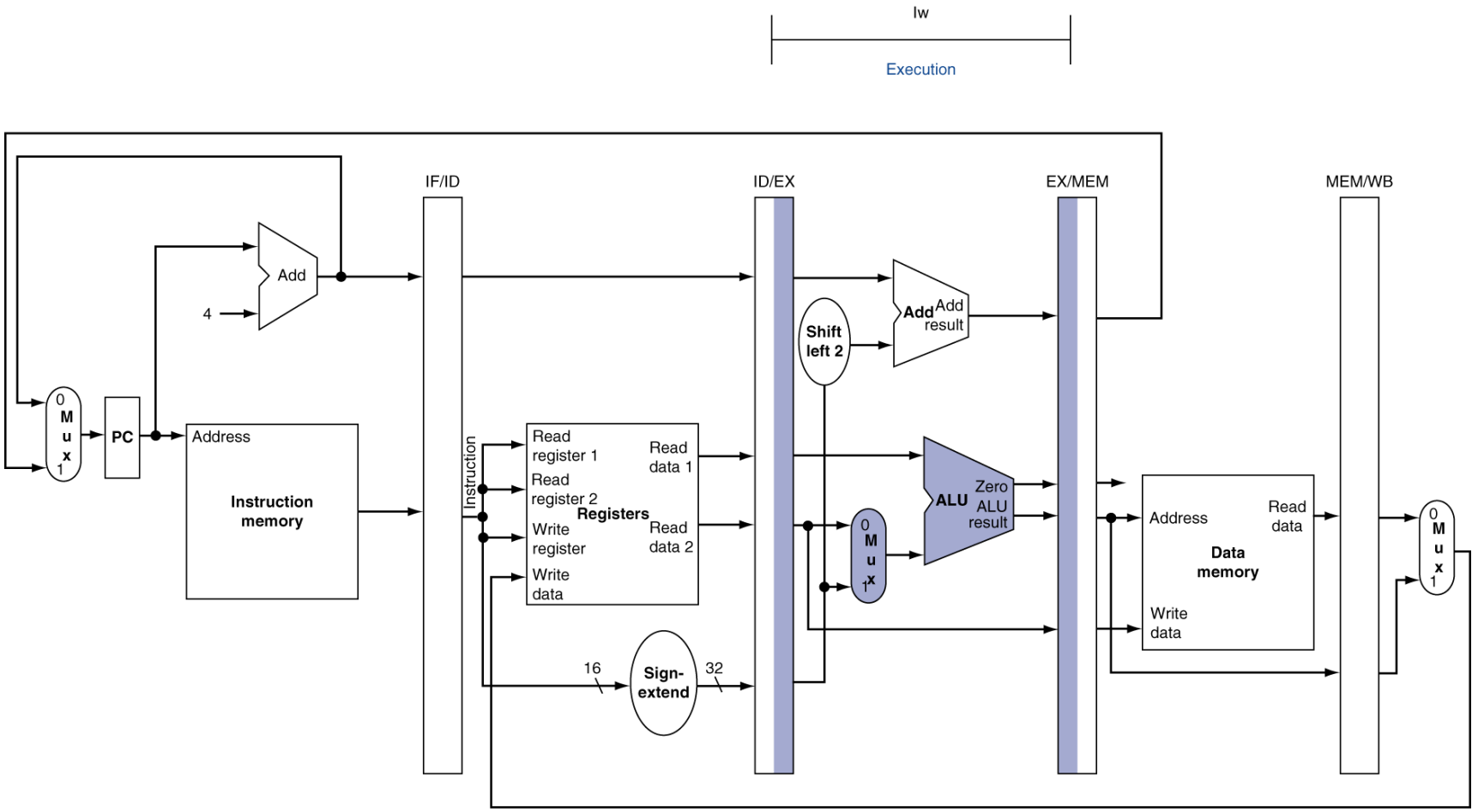
IF for Load, Store, …



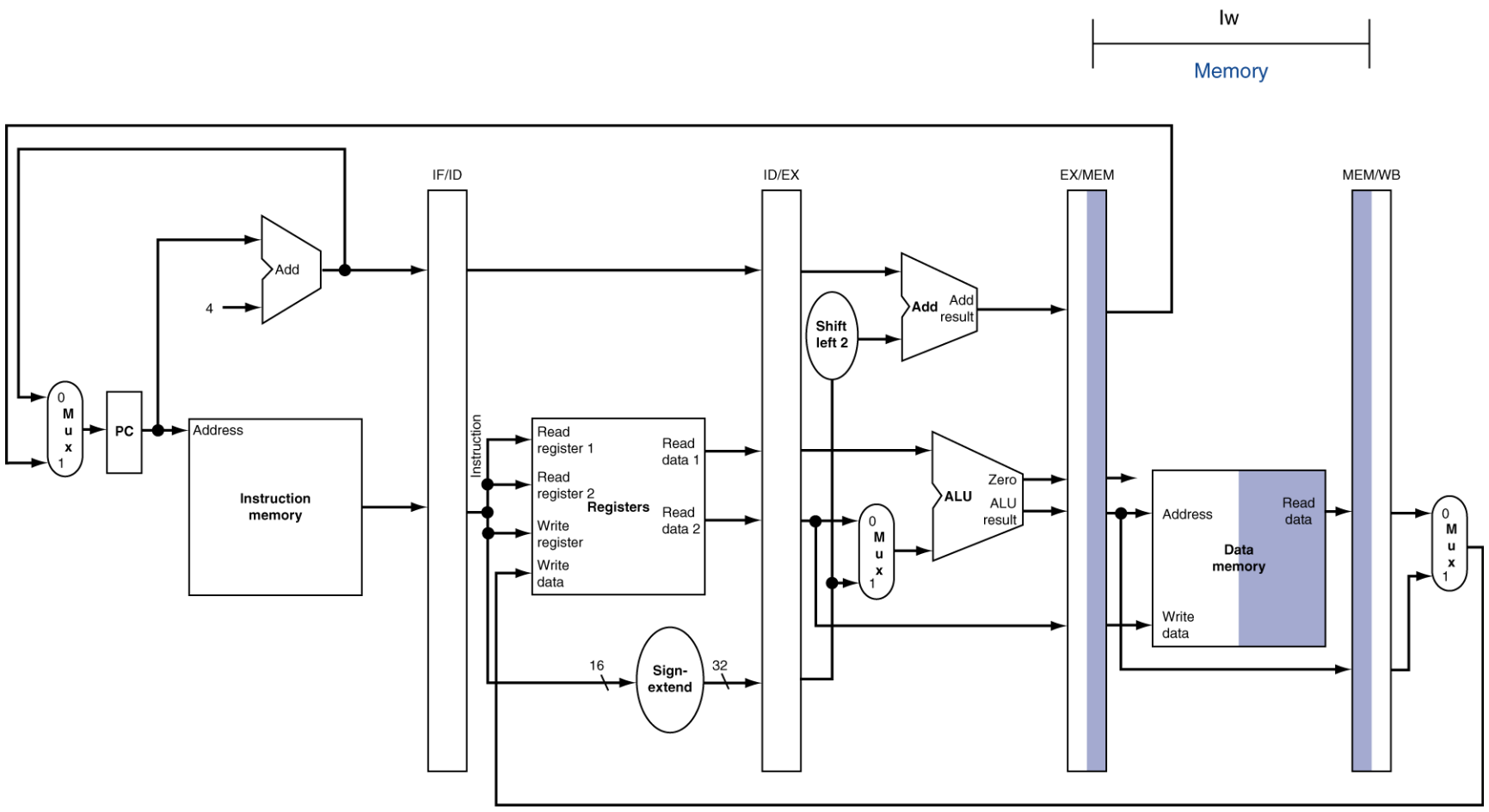
ID for Load, Store, …



EX for Load

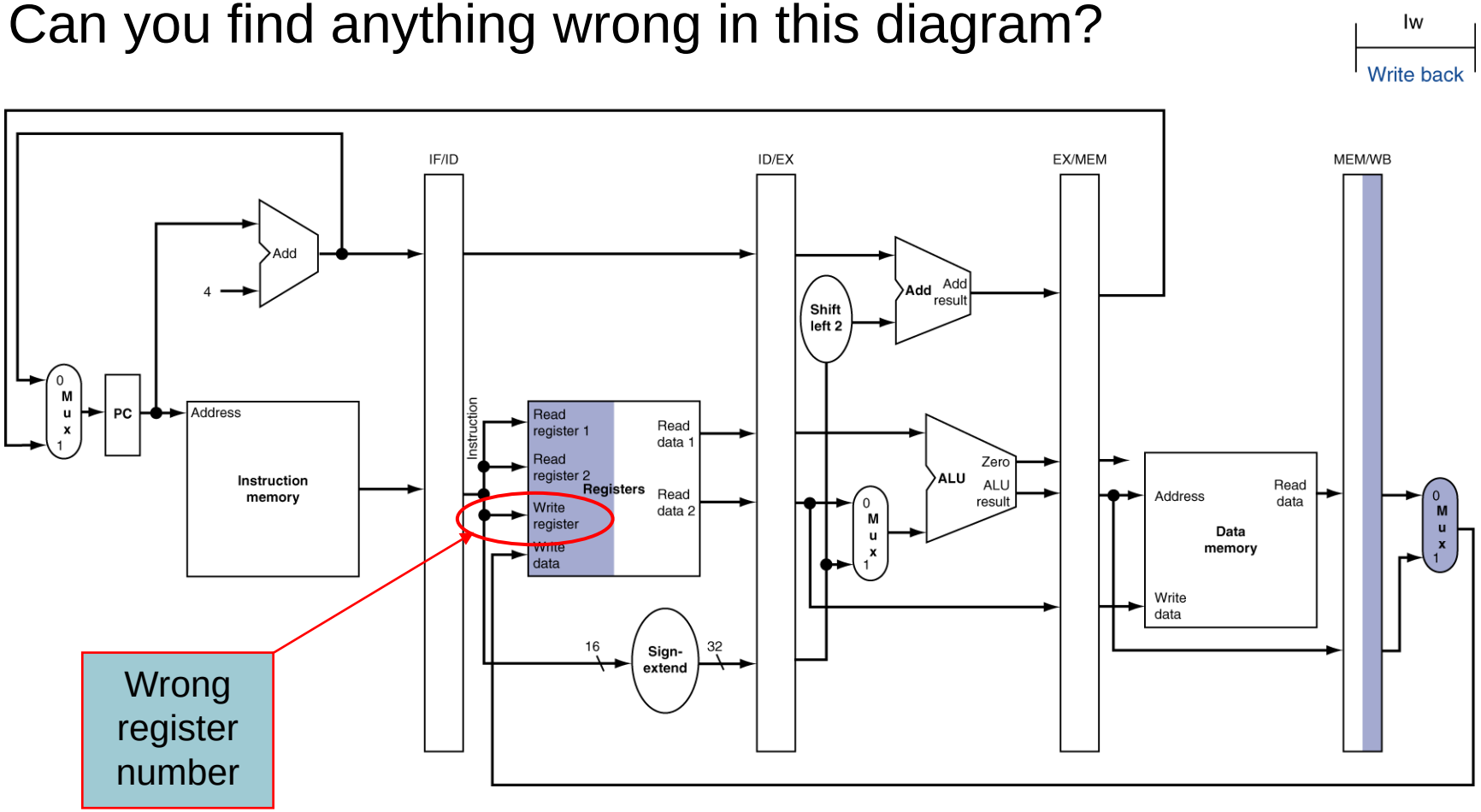

lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

MEM for Load

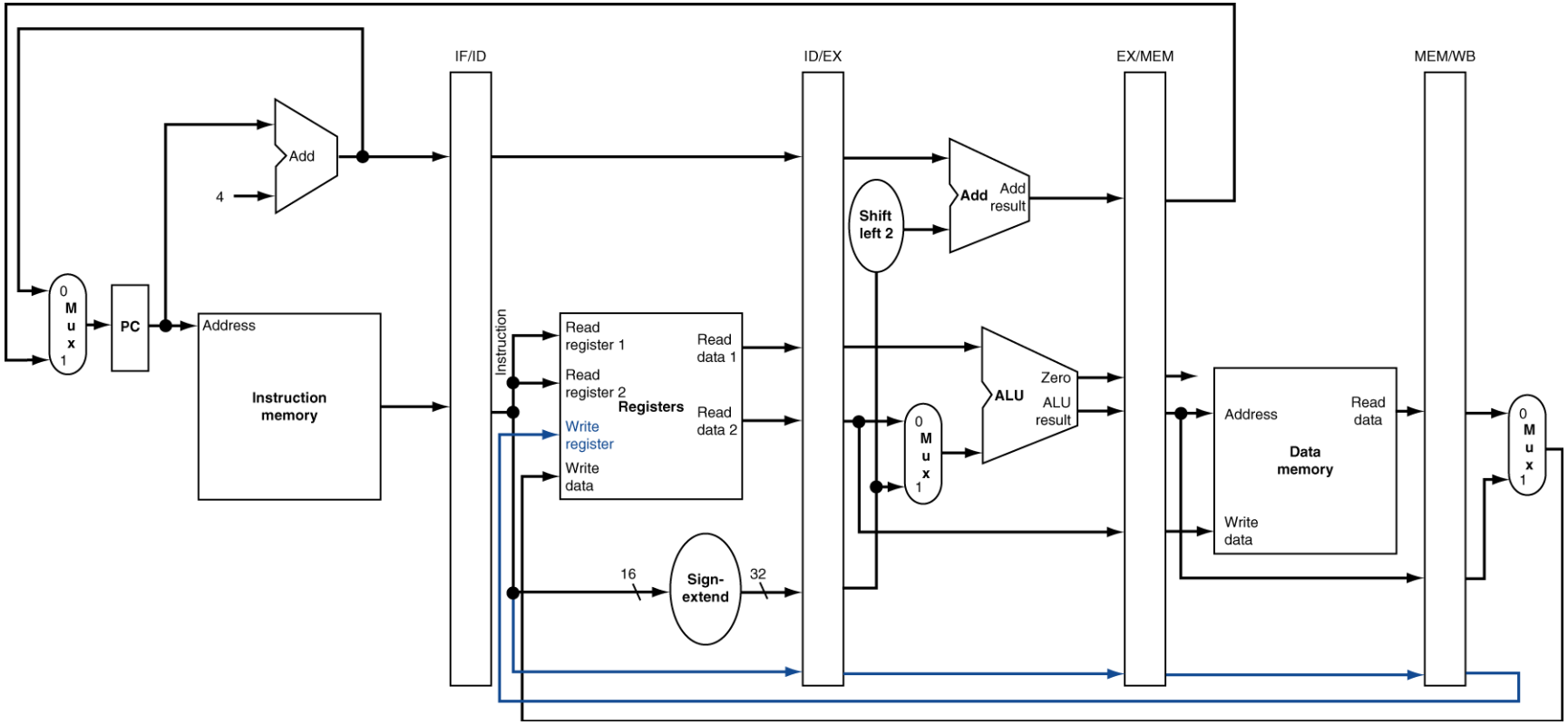

lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

WB for Load

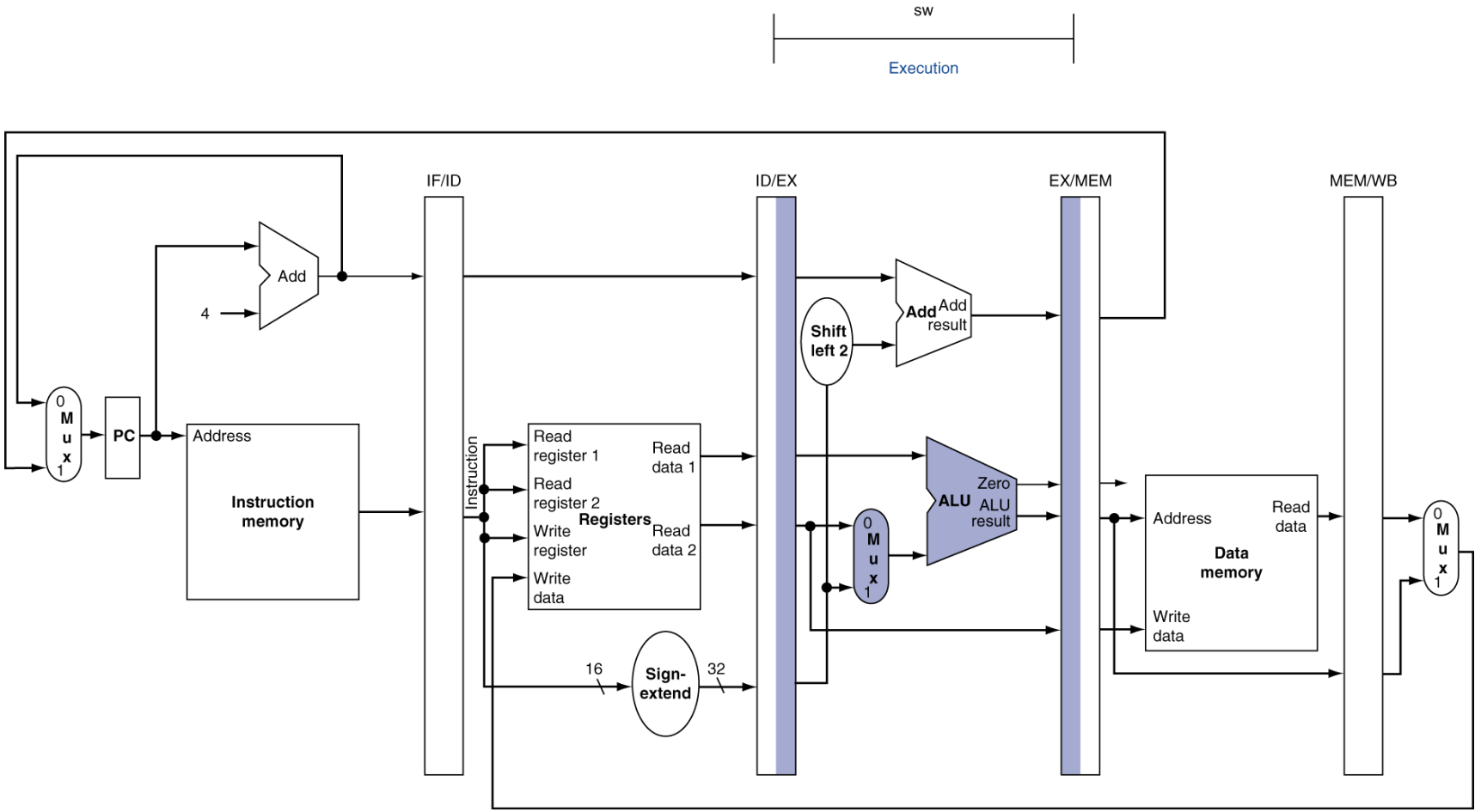
Can you find anything wrong in this diagram?



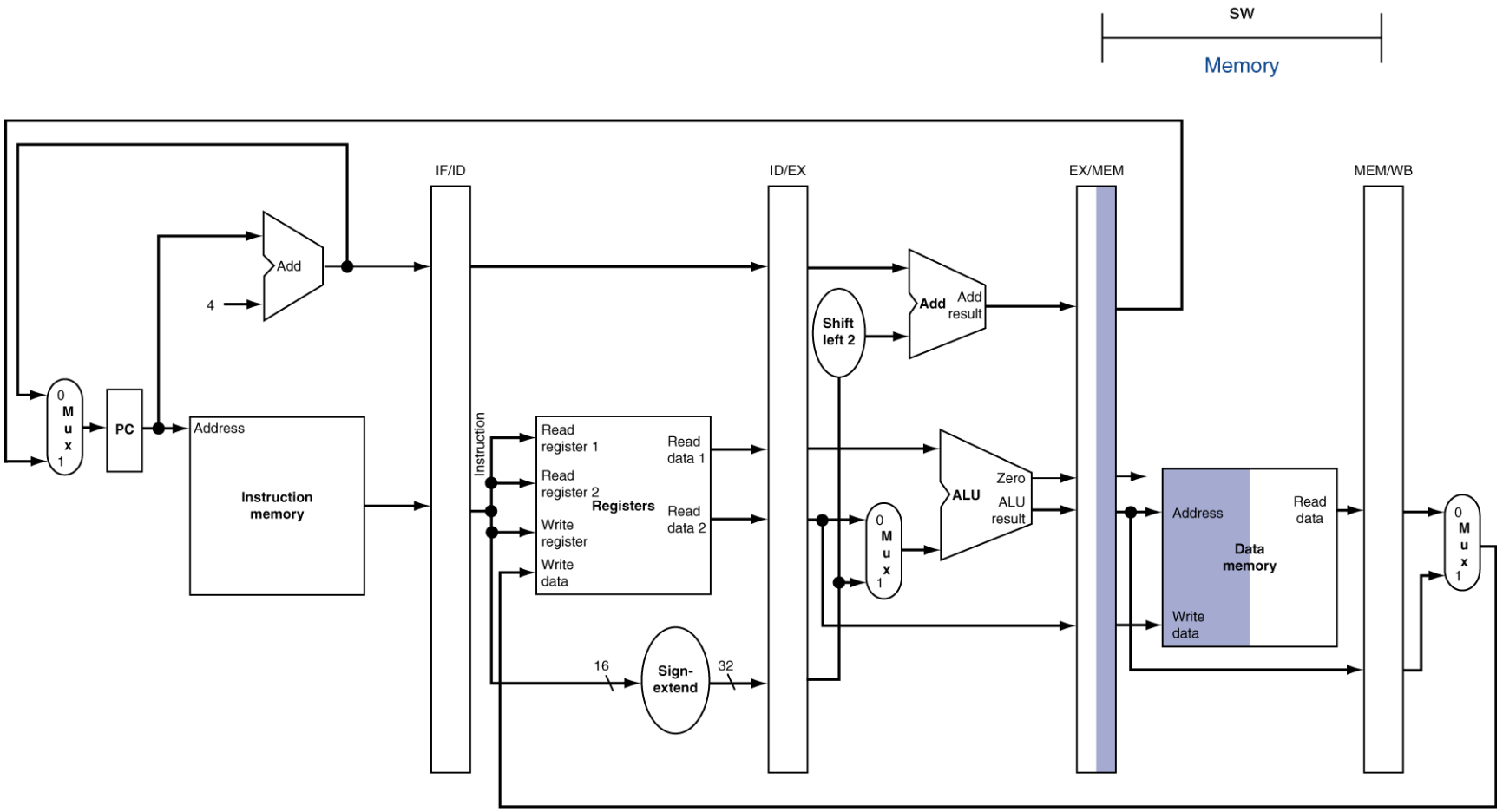
Corrected Datapath for Load



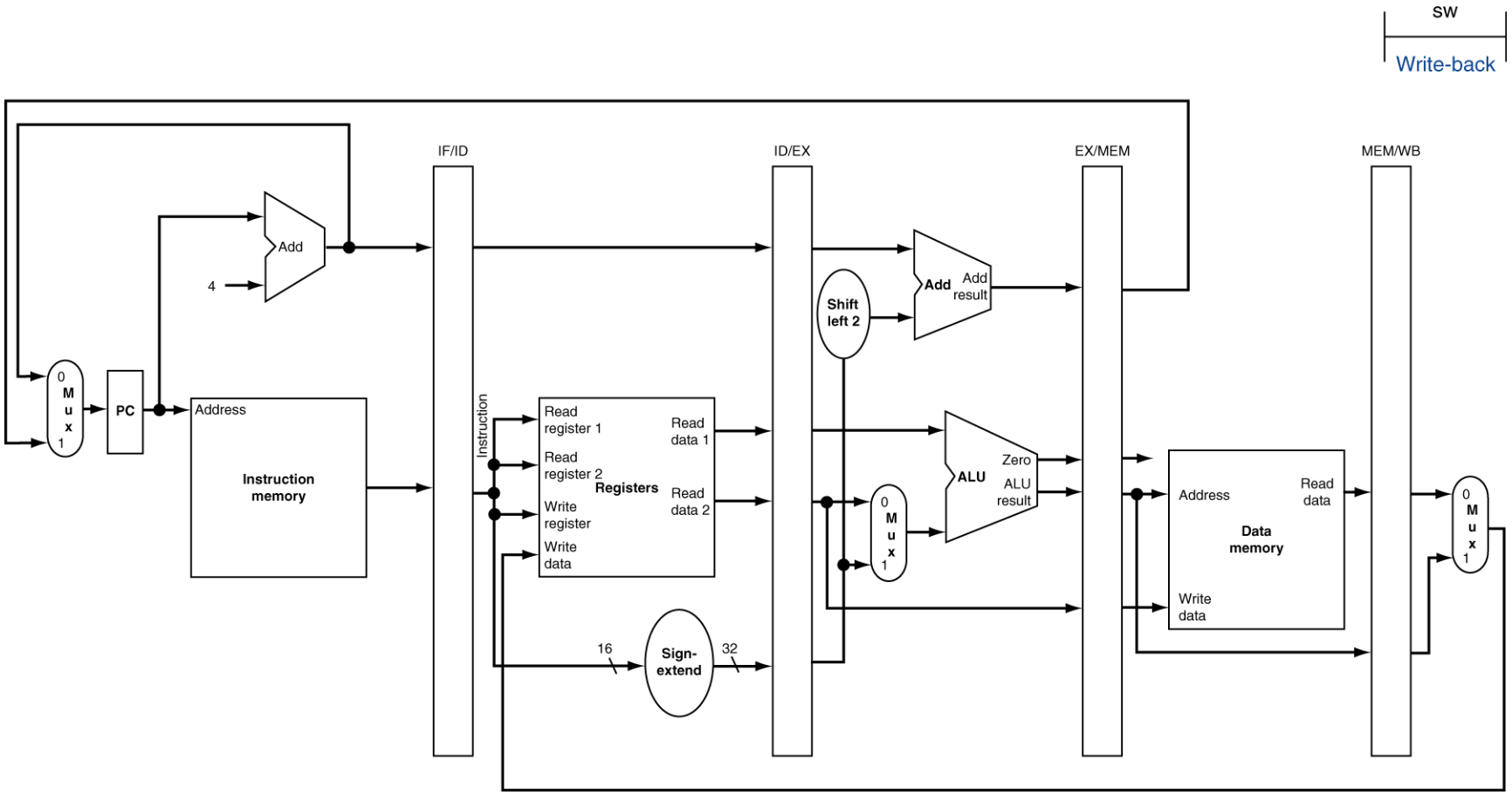
EX for Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

MEM for Store

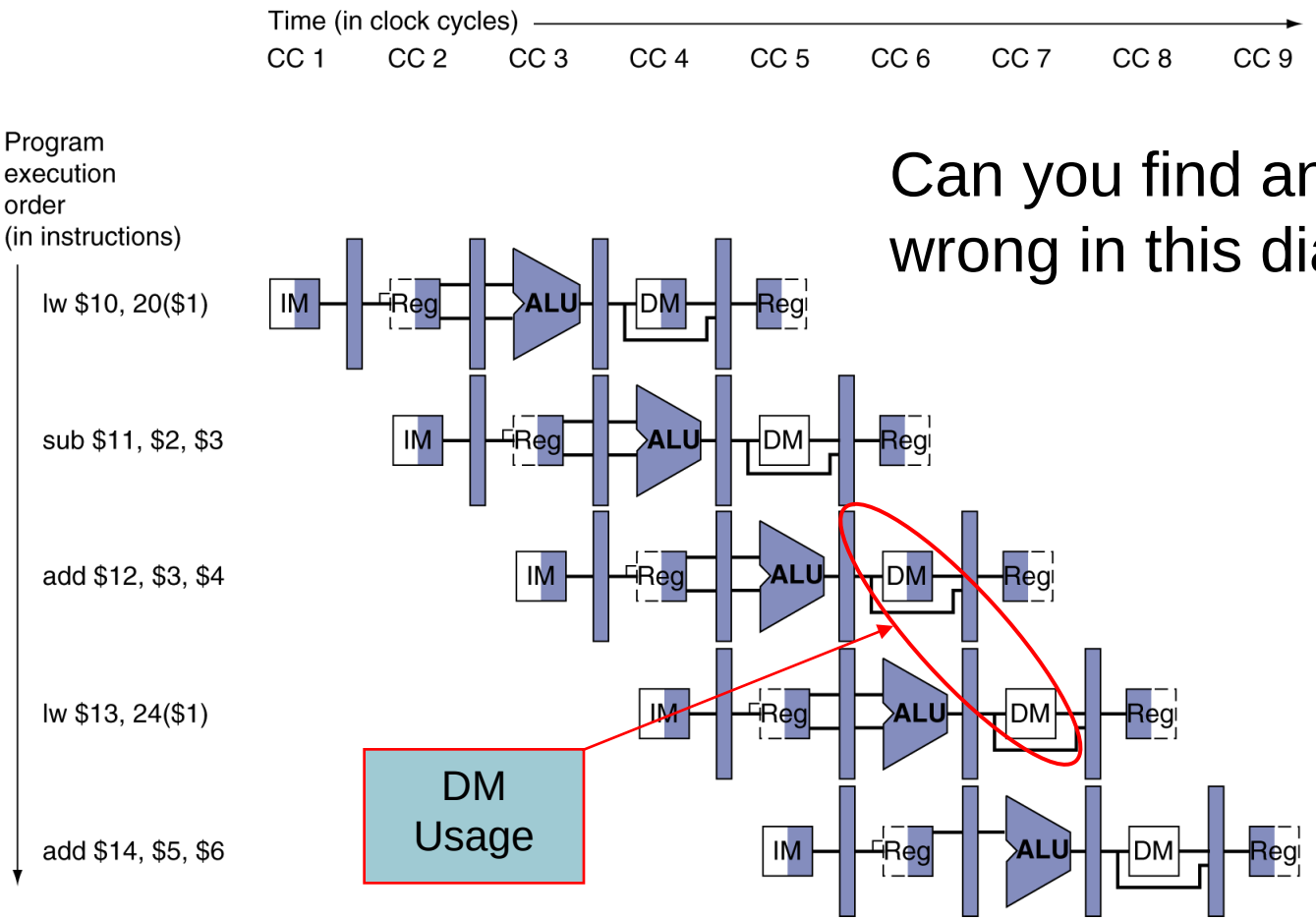


WB for Store


SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

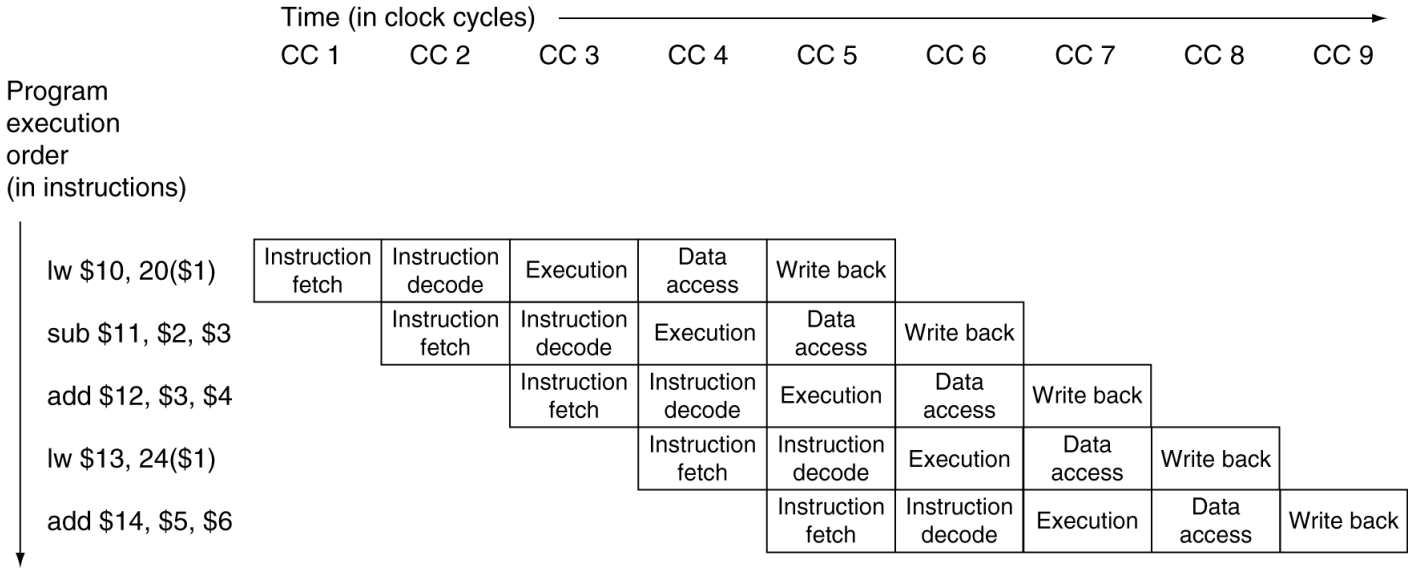
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

- Traditional form



Pipeline Summary

The BIG Picture proves 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

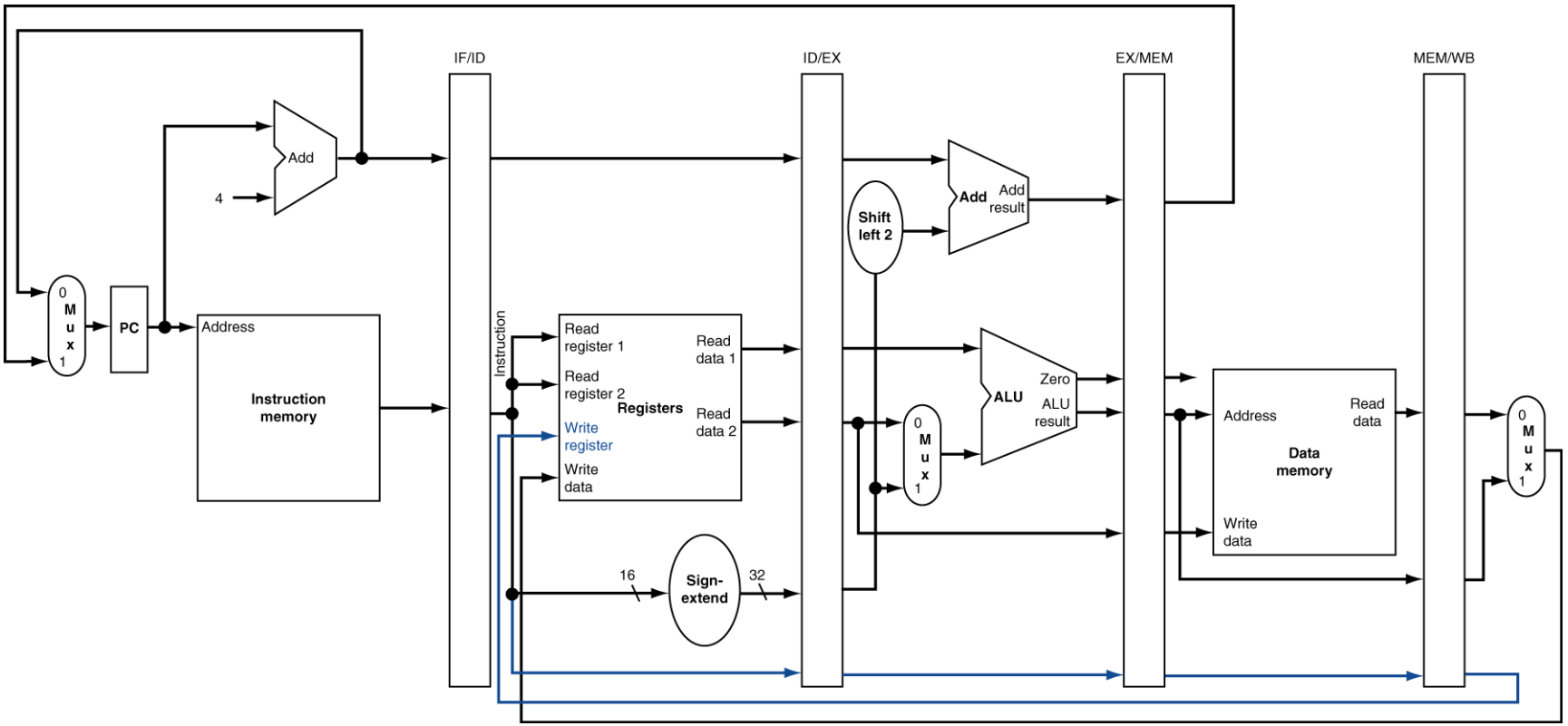
Exercise 1

- The table given below states the latencies of hardware components of a 5 stage pipelined MIPS datapath. Assume memory write latencies are smaller than the corresponding read latencies.
  - Calculate the maximum clock frequency of the processor

Component	I-MEM Read	Adder	MUX	ALU	Reg Read	Reg Write	D-MEM Read	Sign-Extent	Shift-left
Time	200ps	70ps	20ps	90ps	90ps	30ps	250ps	15ps	10ps

  - State any negative impact on performance due to the technique you mentioned above.

Exercise 1



Component	I-MEM Read	Adder	MUX	ALU	Reg Read	Reg Write	D-MEM Read	Sign-Extent	Shift-left
Time	200ps	70ps	20ps	90ps	90ps	30ps	250ps	15ps	10ps

Exercise 2

- For the piece of code given below, derive the result stored in register r6 when executed in case of a standard 5-stage pipelined datapath having no hazard detection unit or forwarding unit. Indicate the hazards and their types. Assume you have a single memory for instruction and data with a single read port. 1 CC is spent on any memory read. You may draw a pipeline diagram to answer this question.

sw r0, 100(r1)

Exercise 3

- The piece of code given rec
number of clock cycles to c
execution. Calculate N and
techniques to reduce N whi
the correctness of the code
new number of clock cycles
Show your work.

```
LD      R1,100(R2)
LD      R3, 200(R2)
ADD     R4, R1, R3
ST      R4, 500(R7)
MUL     R5, R1, R3
ST      R5, 600(R7)
LD      R1, 300(R2)
ADD     R6, R1, R3
ST      R6,400(R7)
```

Exercise 4

- For the piece of code given derive the result stored in register R10 when executed in case of a stanc pipelined datapath having r detection unit and (i) with a (ii) without a forwarding uni

```
LD      R0, 100(R1)
ADD     R2, R0, R3
MUL     R2, R2, R5
LD      R6, 200(R1)
SUB     R7, R6, R7
AND     R10, R7, R8
```

Exercise 5

In the piece of code is given below that is going to be executed in a pipelined processor, identify and explain the possible hazards and classify them.

```
lw   r2, r4, #10     ;       r2 ← M[r4 +10]
add  r1, r2, r3      ;       r1 ← r2 + r3
sub  r5, r1, r6      ;       r5 ← r1 - r6
beq  r5, r8, #100    ;       if  r5 == r8 then PC ← PC + 4*100
or   r9, r11, r10    ;       r9 ← r11 or r10
```

Use a diagram show the code execution (by filling the cells using the pipeline stages: IF, ID, EX, ME, WB) with different solutions existing to solve the hazards you identified above. If you have two solutions for a single hazard draw two separate diagrams to show them.

Exercise 6

a. Consider pipelined processor architecture. Assume that the time taken to execute an ALU operation can be shortened by 25%:

(i) Will it affect the speedup obtained from pipelining? If yes, by how much? Otherwise, why?

(ii) What if the ALU operation now takes 25% more time?

b. A computer architect needs to design the pipeline of a new microprocessor. She has an example workload program code with 10^6 instructions. Each instruction takes 100ps to finish.

(i) How long does it take to execute this program code on a non-pipelined processor?

(ii) The current state-of-the-art microprocessor has about 20 pipeline stages. Assume it is perfectly pipelined. How much speedup will it achieve compared to the non-pipelined processor?

(iii) Real pipelining is not perfect, since implementing pipelining introduces some overhead per pipeline stage. Will this overhead affect the instruction latency or the instruction throughput, or both?

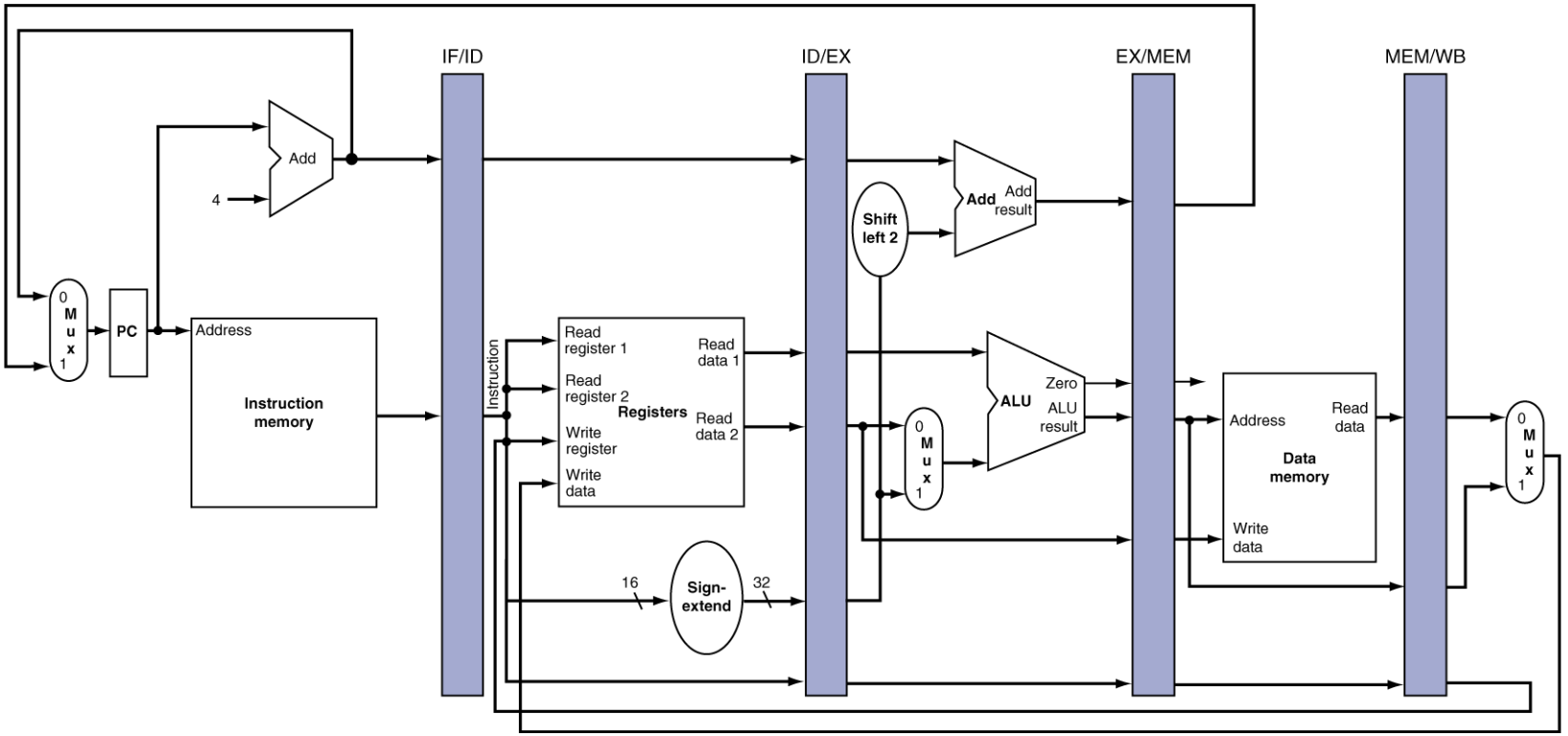
c. By filling a table, show the forwarding paths needed to execute the following four instructions:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

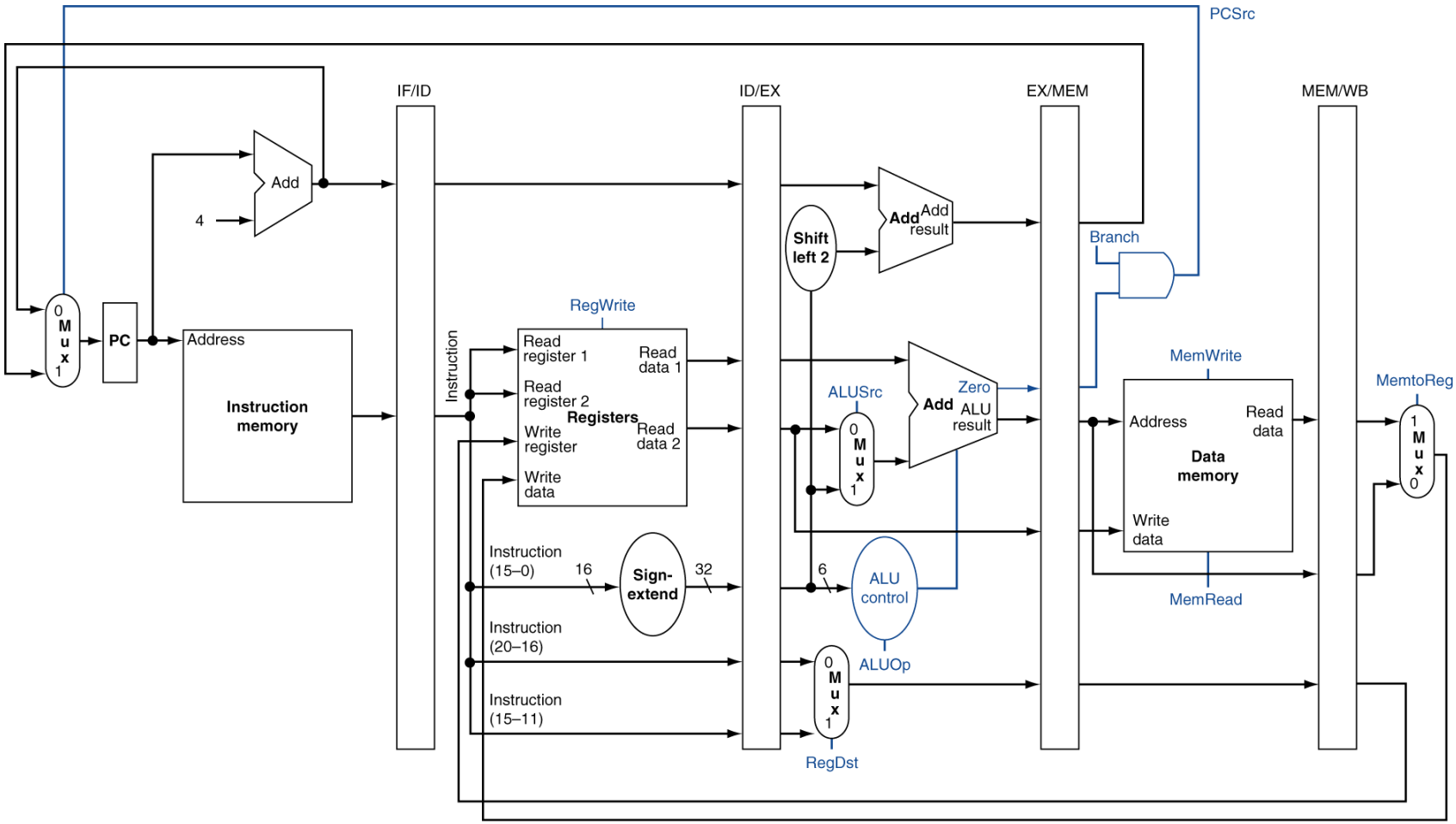
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

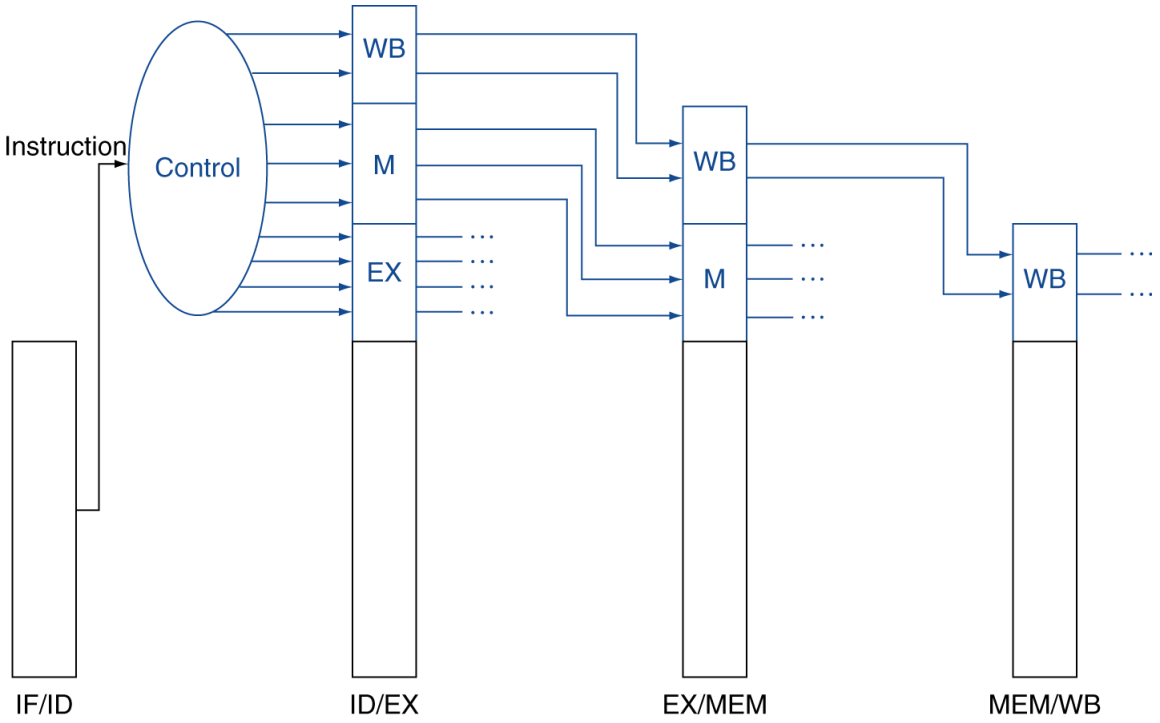


Pipelined Control (Simplified)

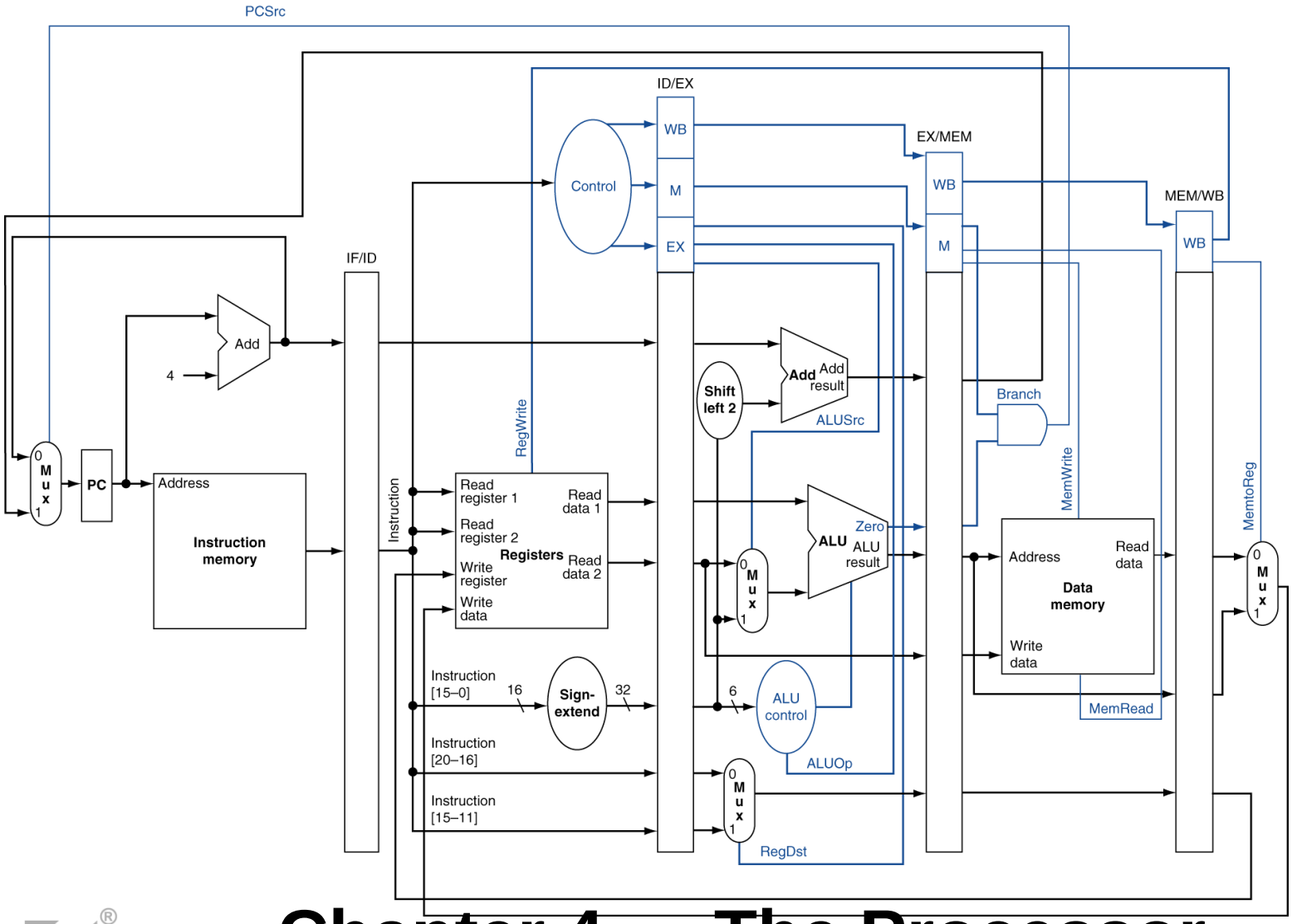


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



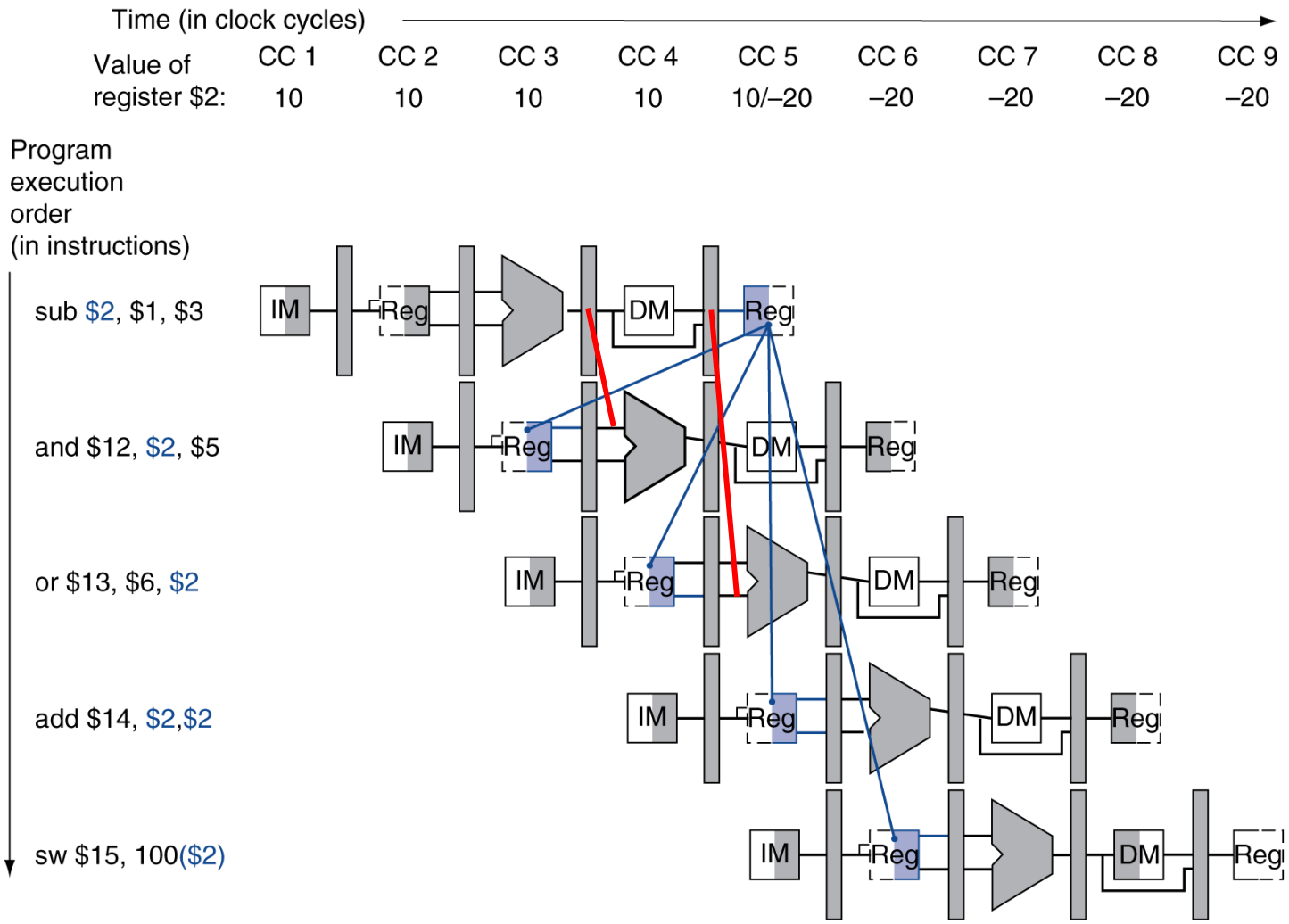
Pipelined Control



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



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

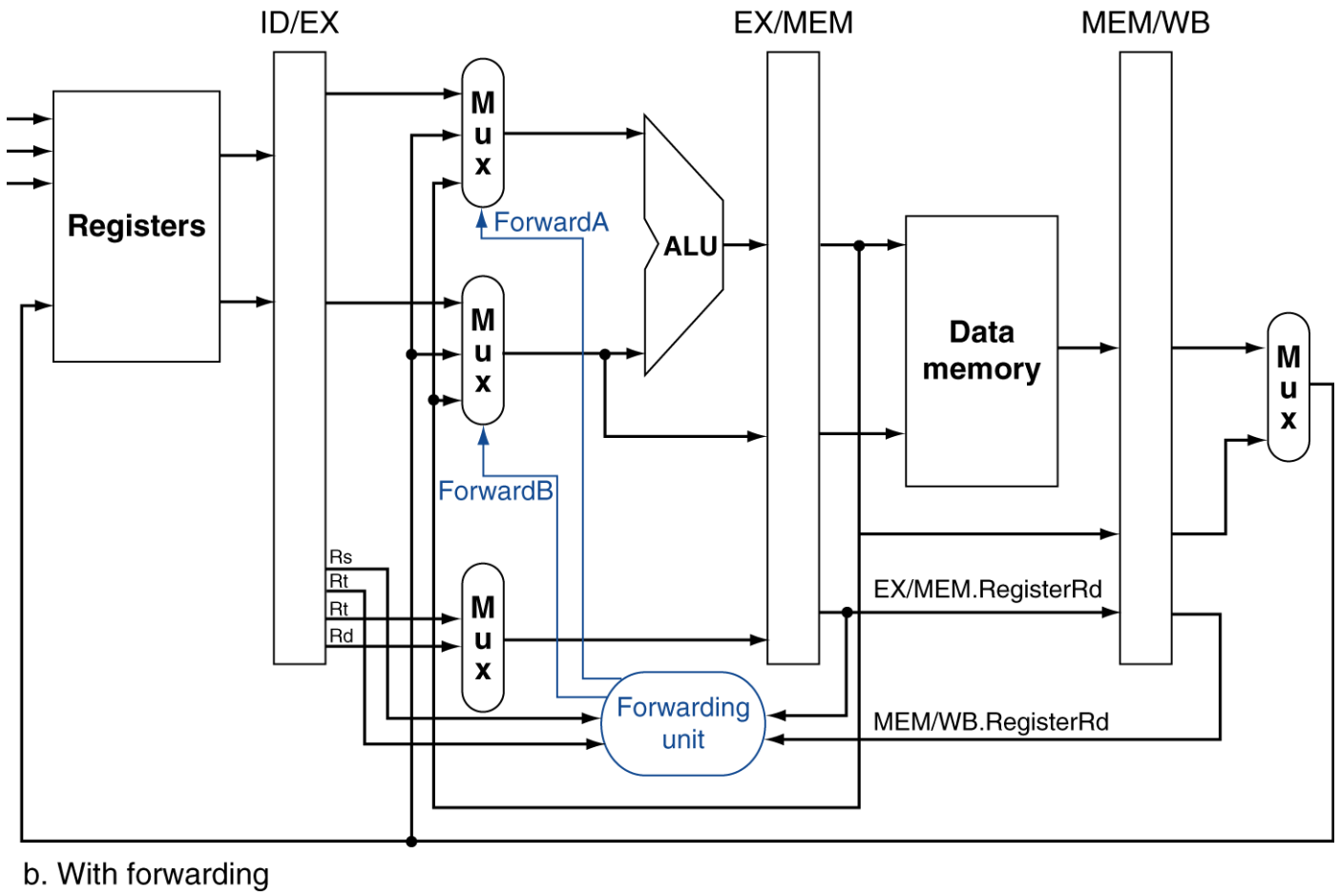
Fwd from EX/MEM pipeline reg

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

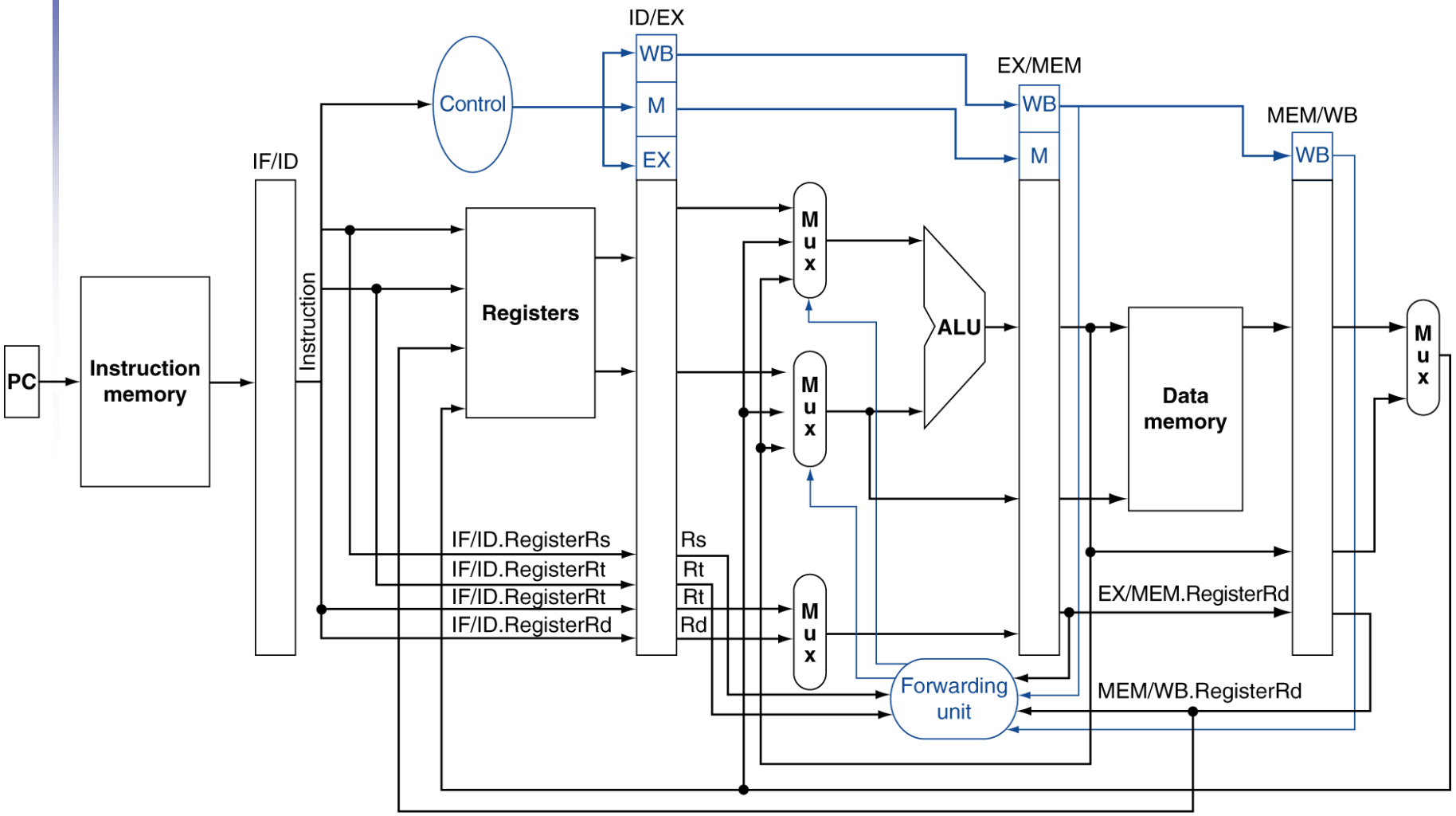
Double Data Hazard

- 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

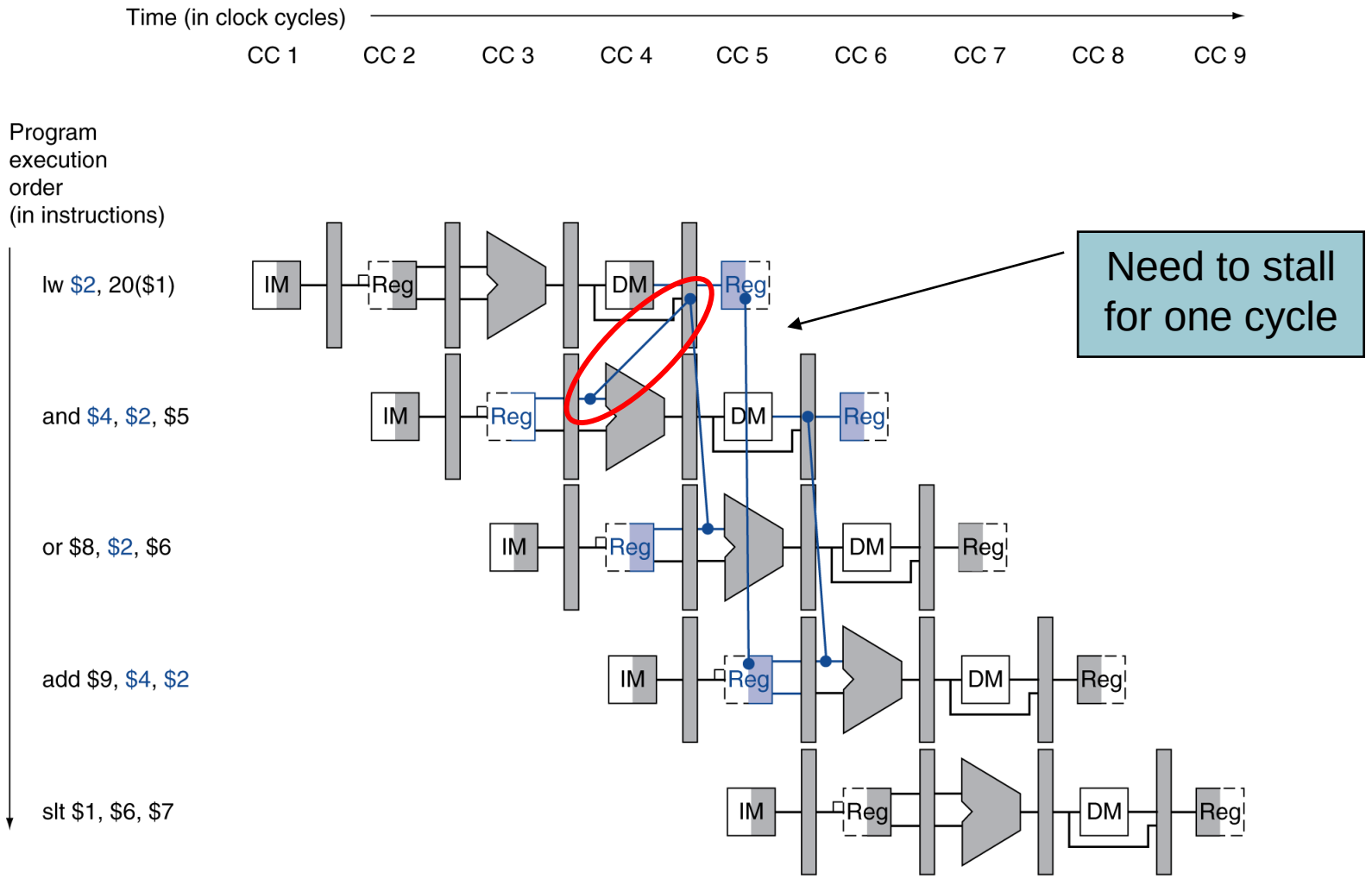
Revised Forwarding Condition

- 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

Datapath with Forwarding



Load-Use Data Hazard



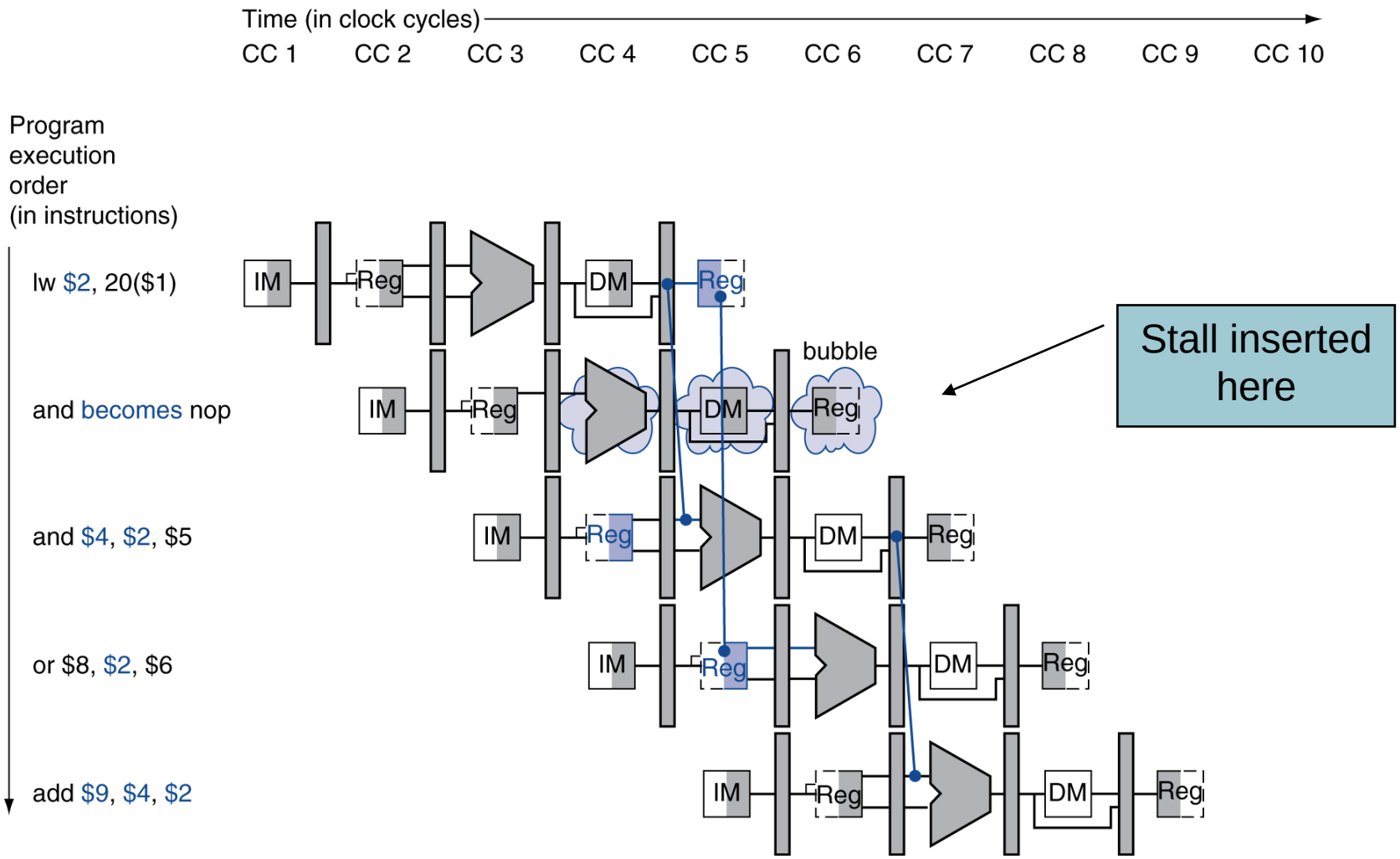
Load-Use Hazard Detection

- 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

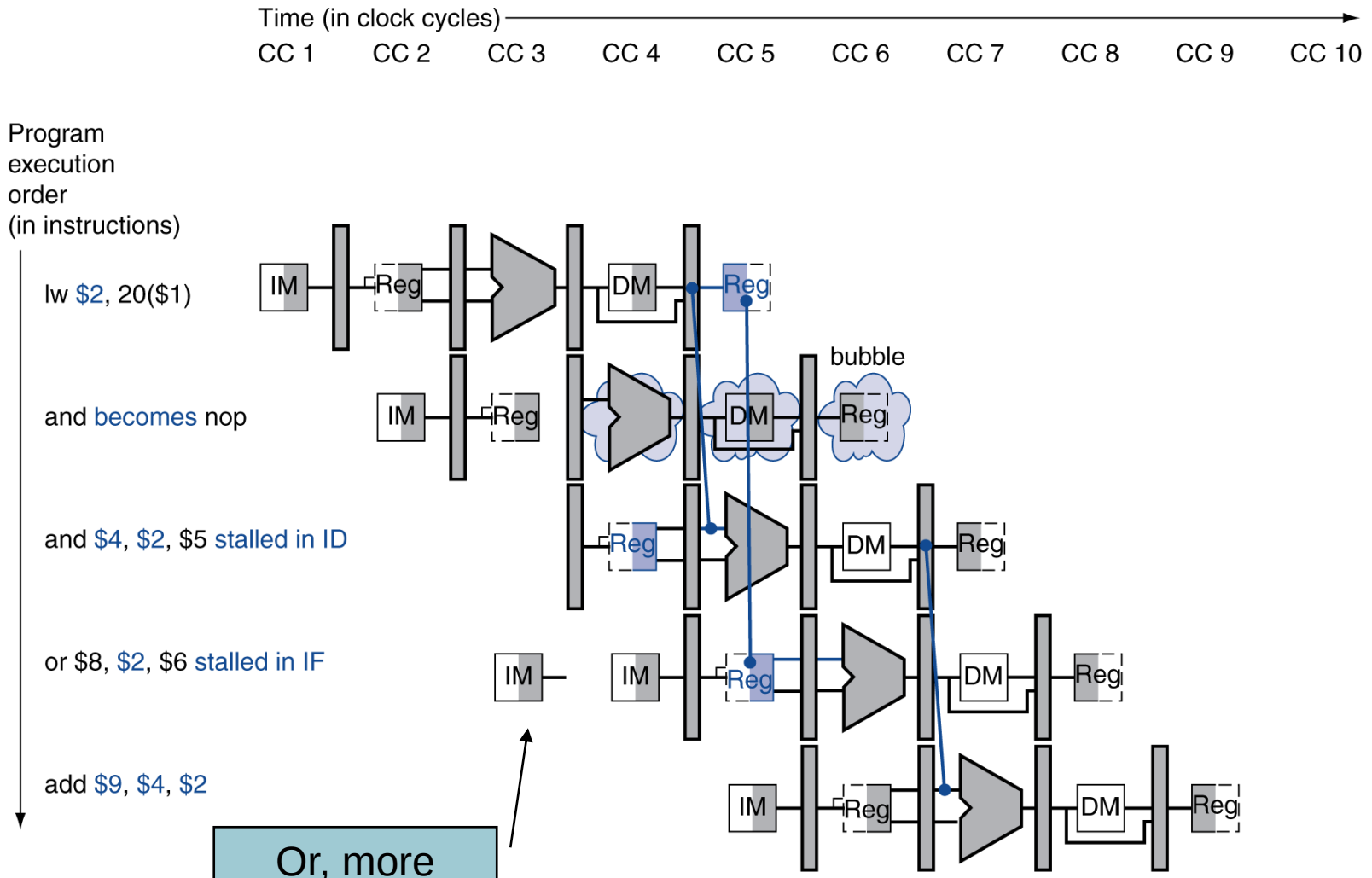
How to Stall the Pipeline

- 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

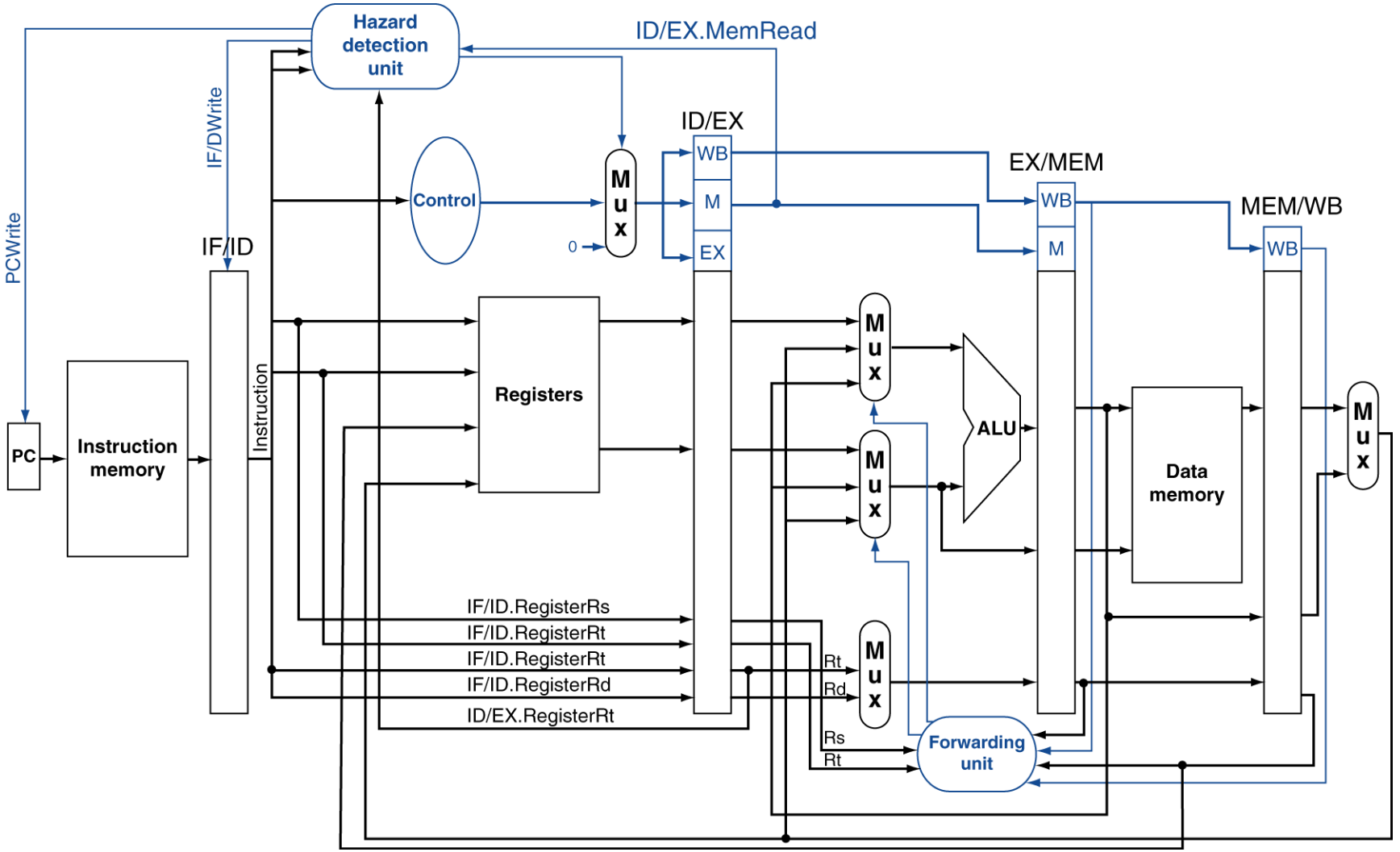
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath With Hazard Detection



Stalls and Performance

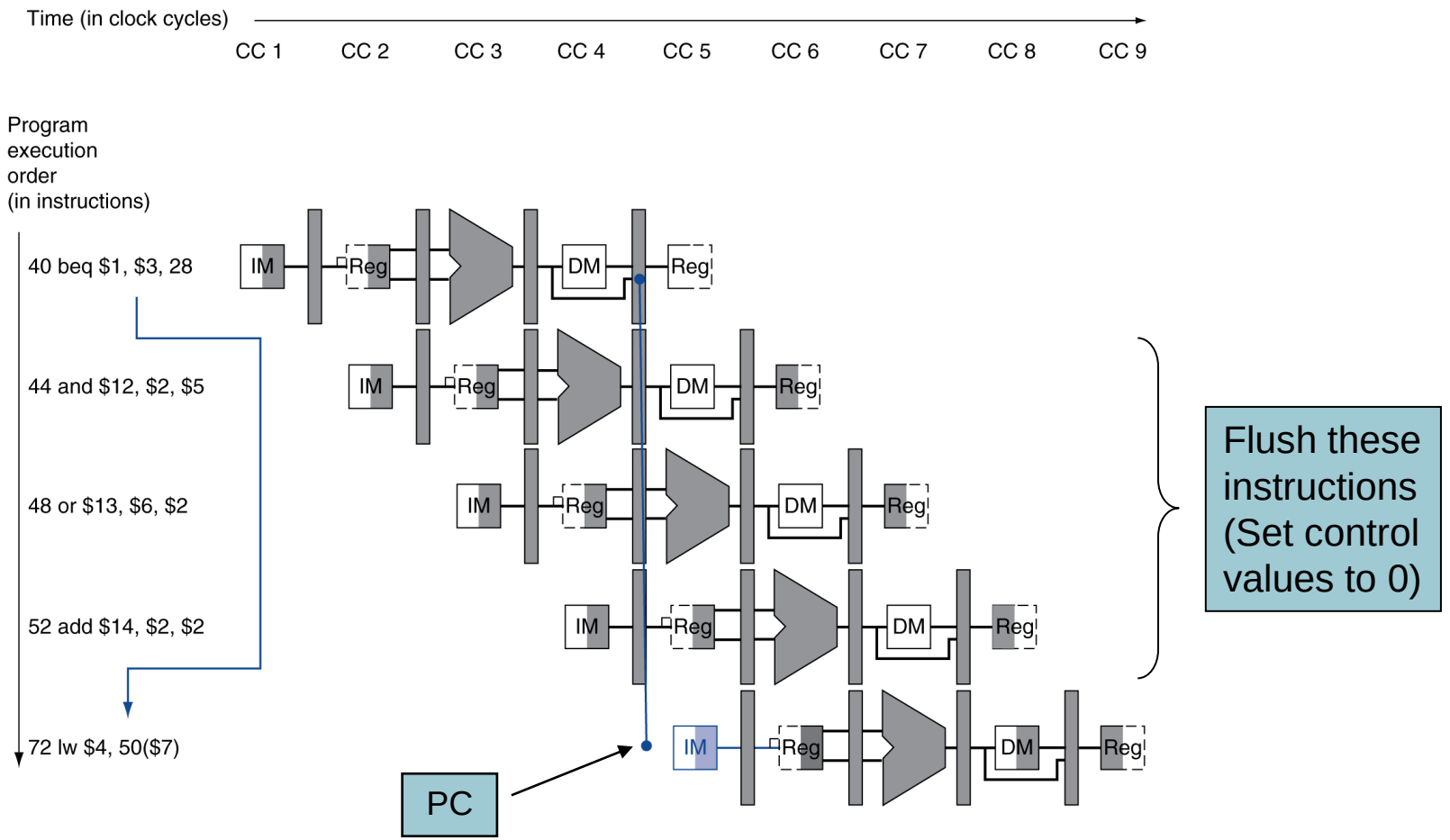
The BIG Picture performance

- But are required to get correct results

- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

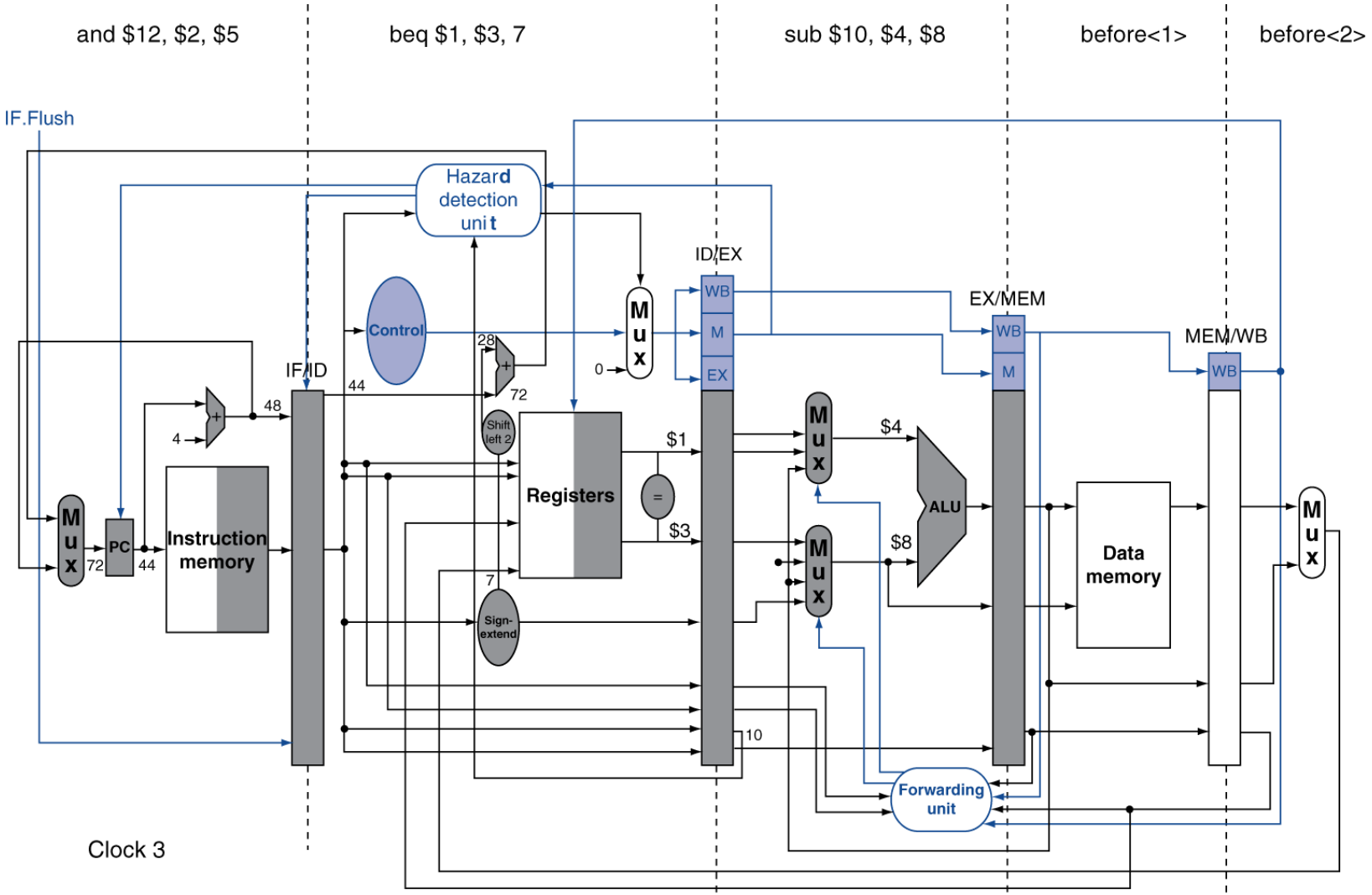
- If branch outcome determined in MEM



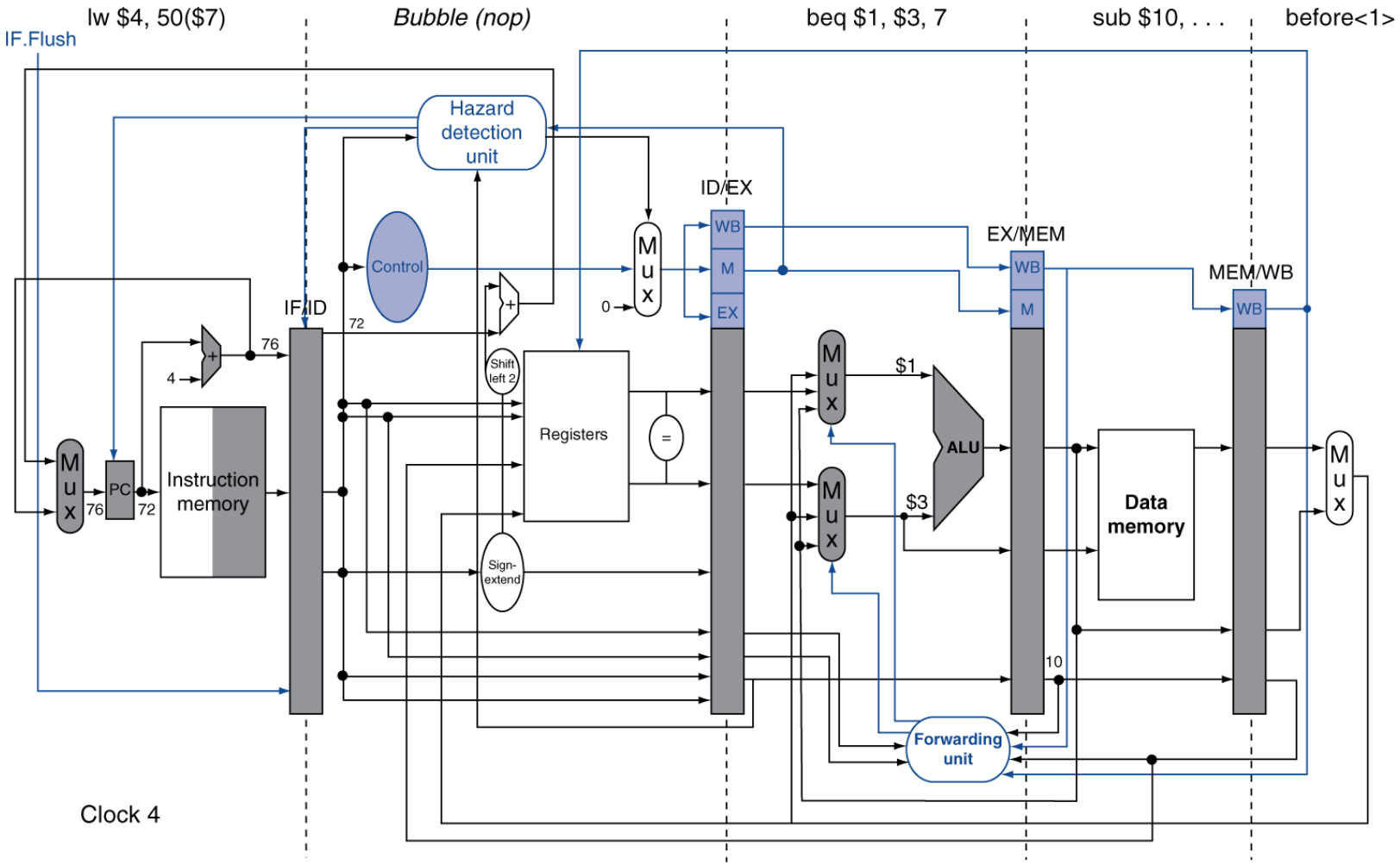
Reducing Branch Delay

- 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
    ...

Example: Branch Taken



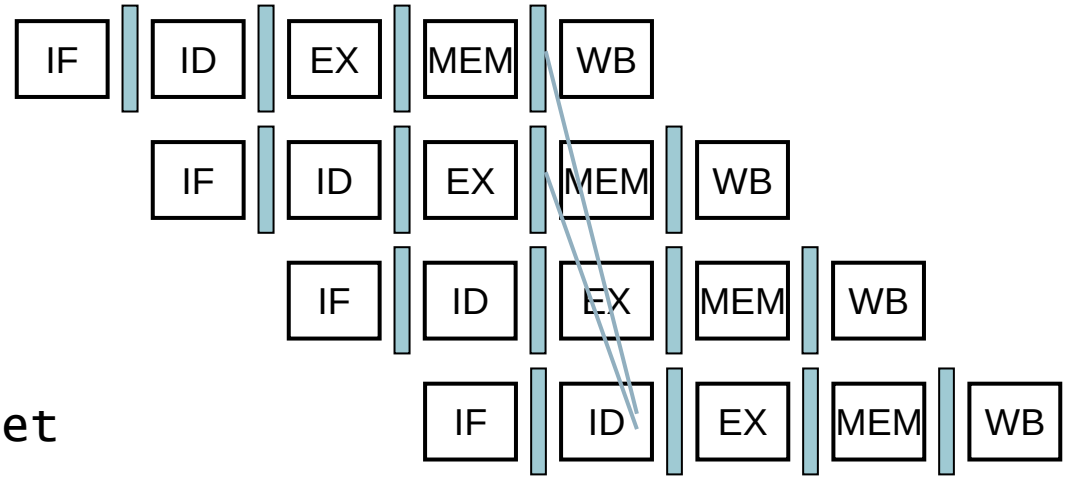
Example: Branch Taken



Data Hazards for Branches

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

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

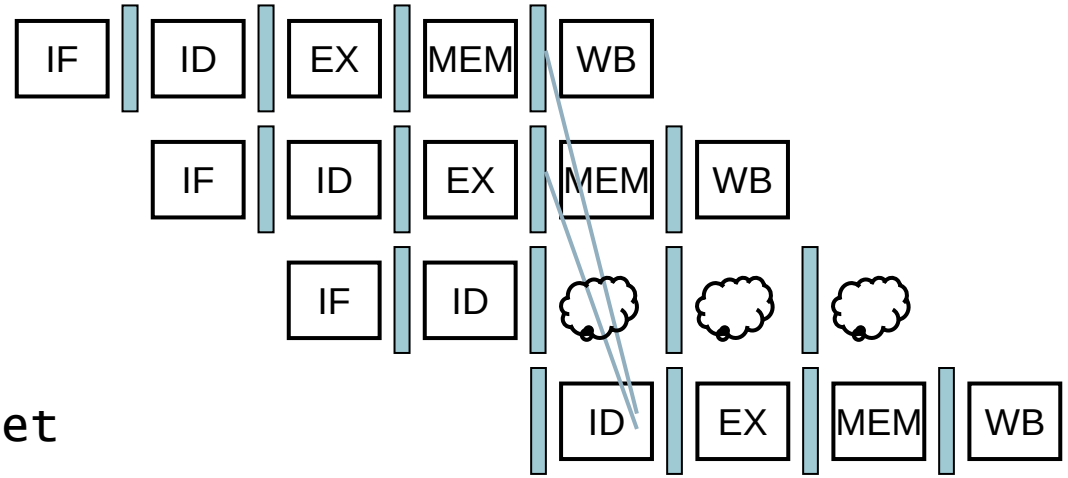


- Can resolve using forwarding

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

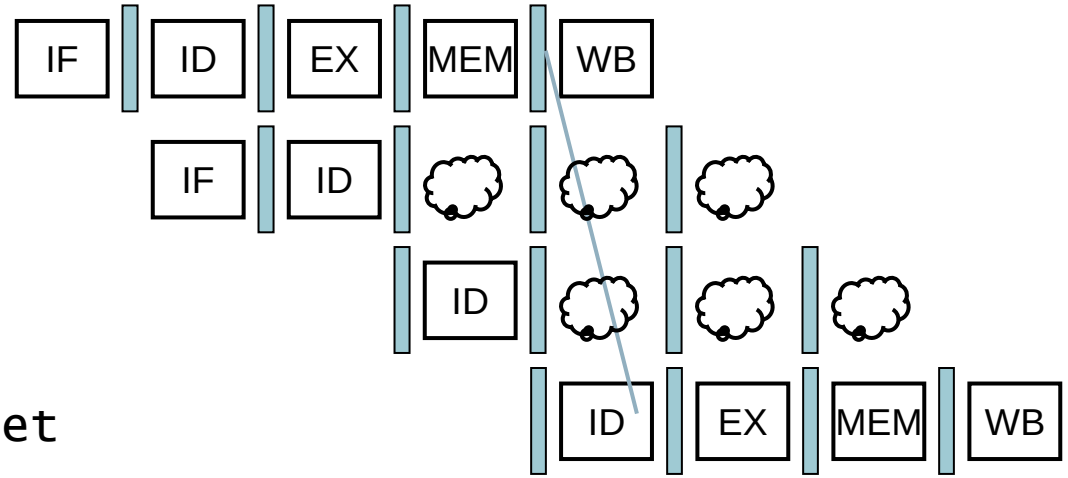
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

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

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```

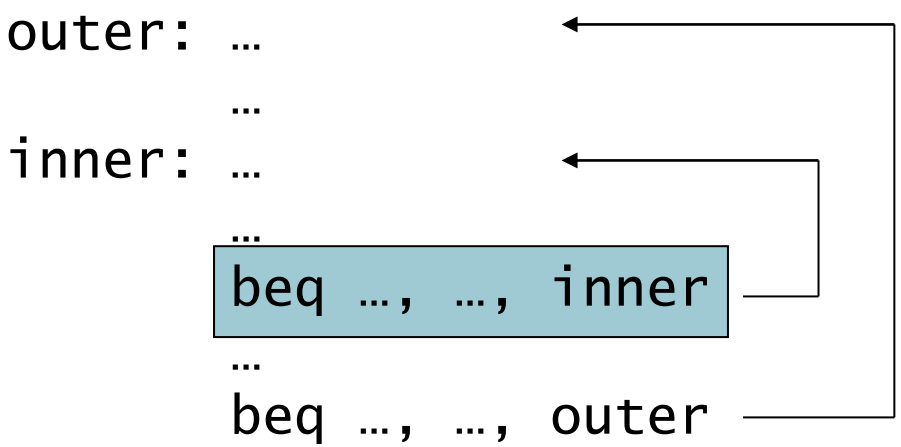


Dynamic Branch Prediction

- 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

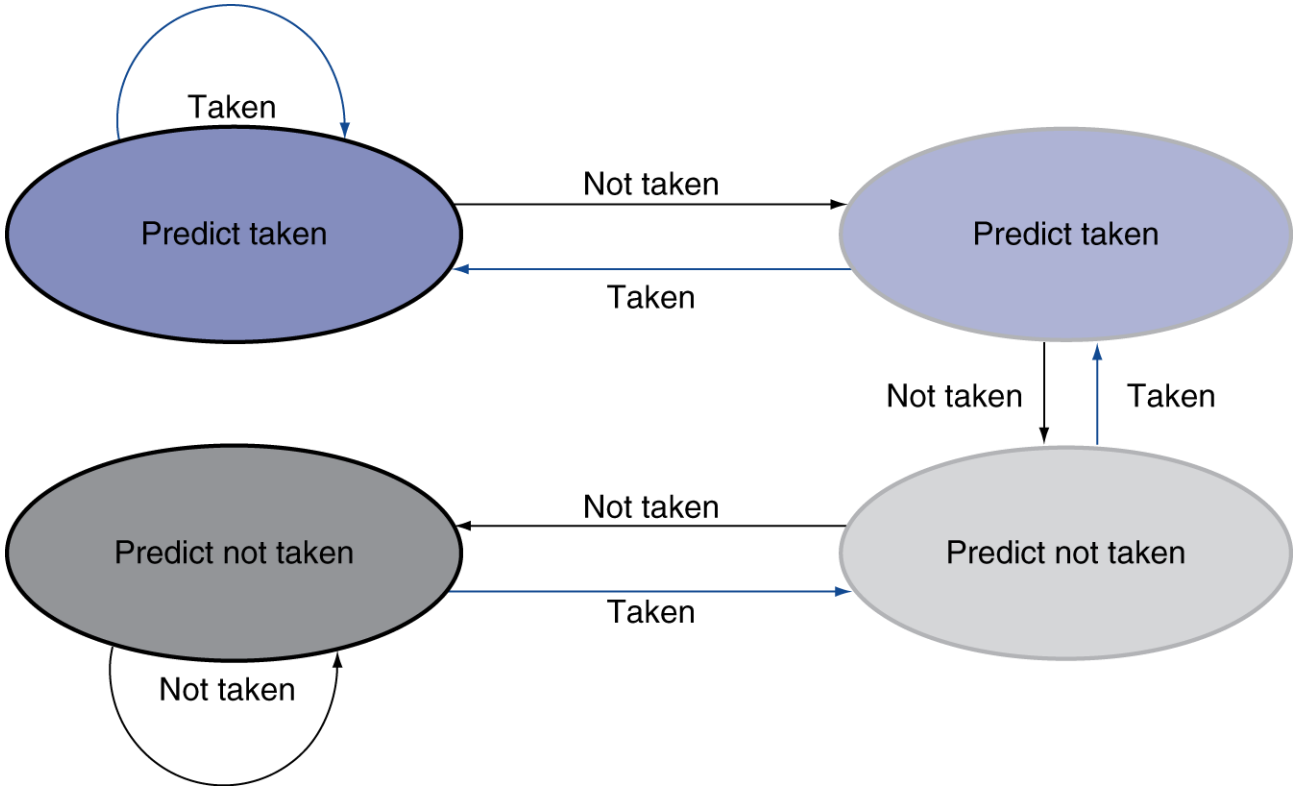
- Inner loop branches mispredicted twice!



  - 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

Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

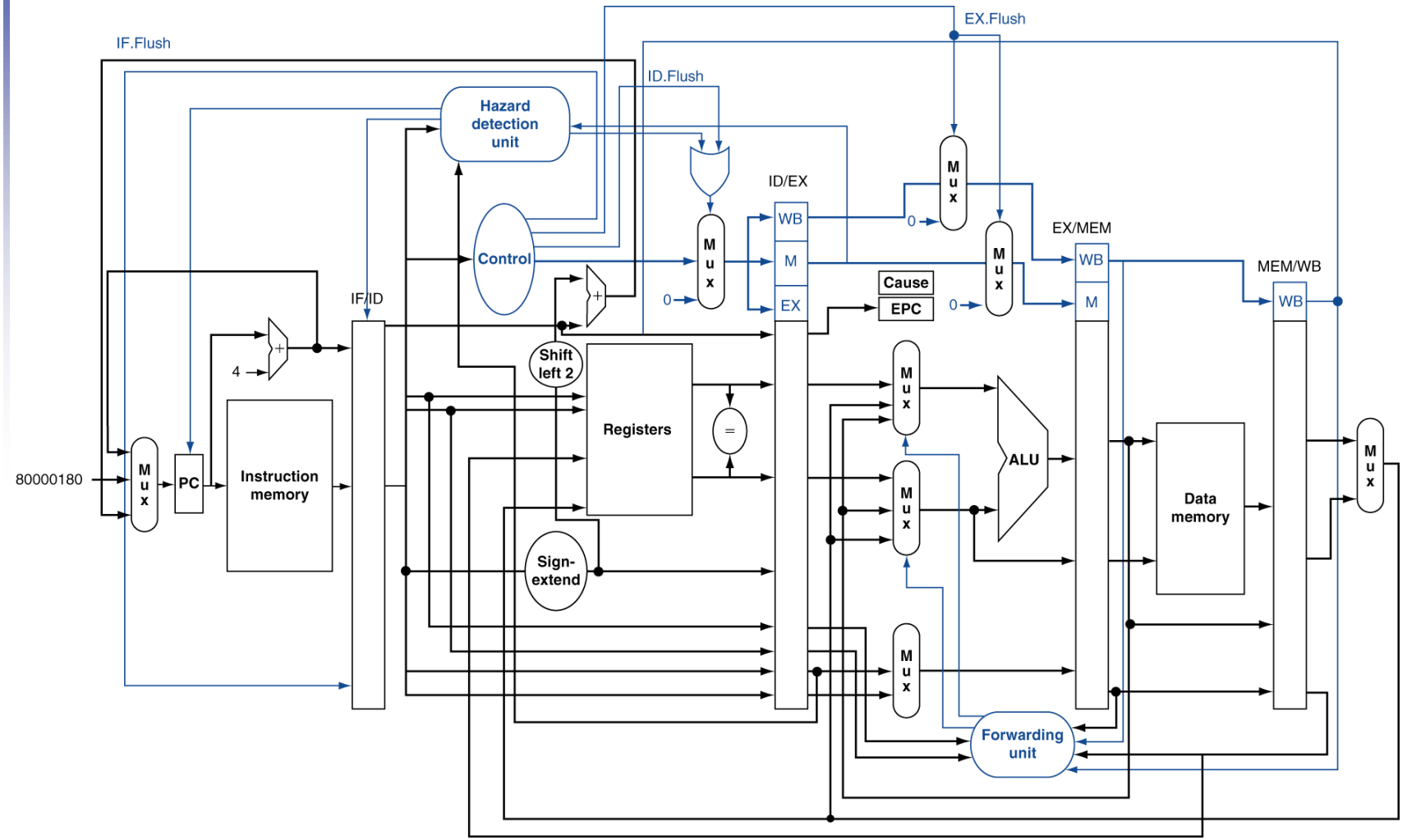
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  - add $1, $2, $1
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



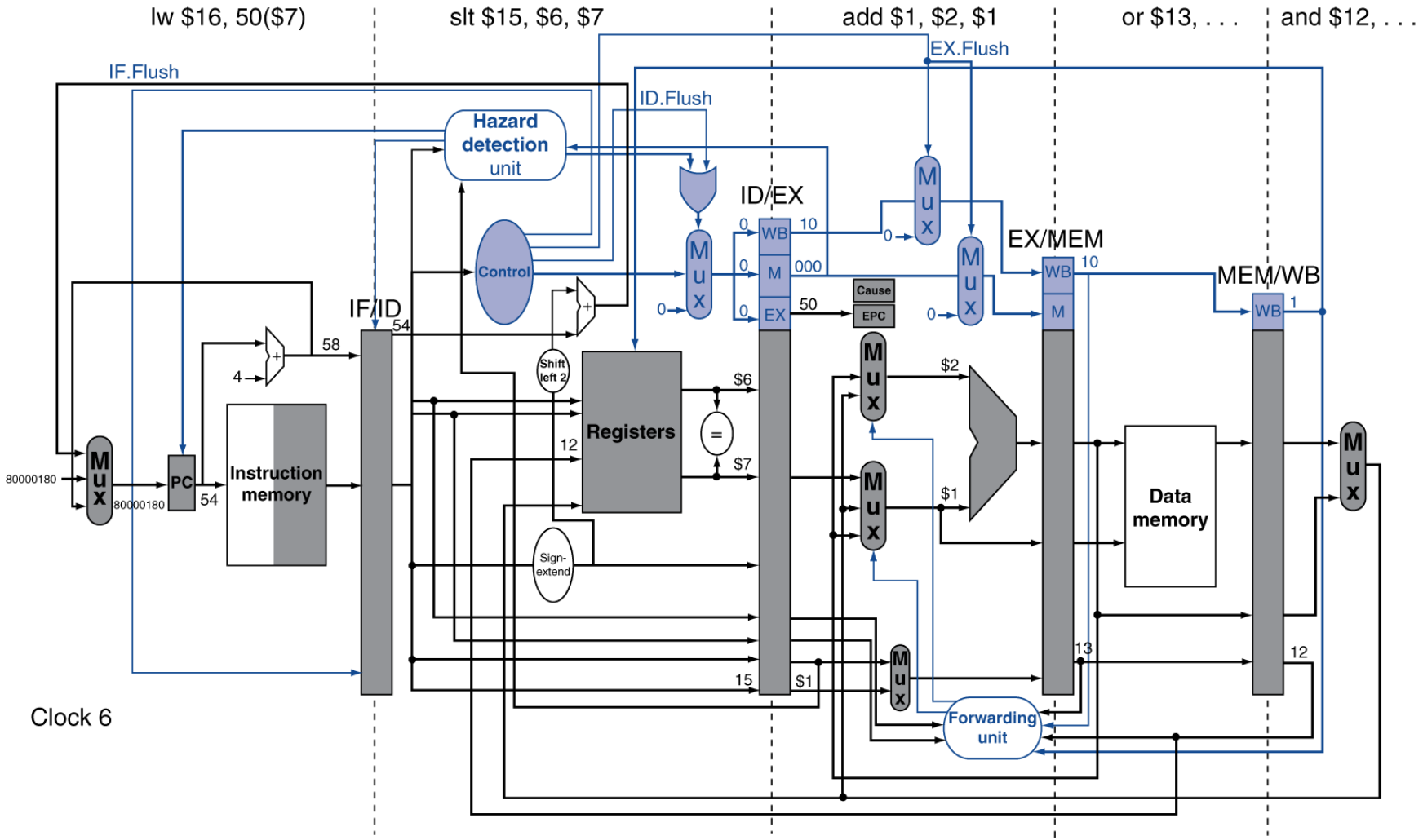
Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

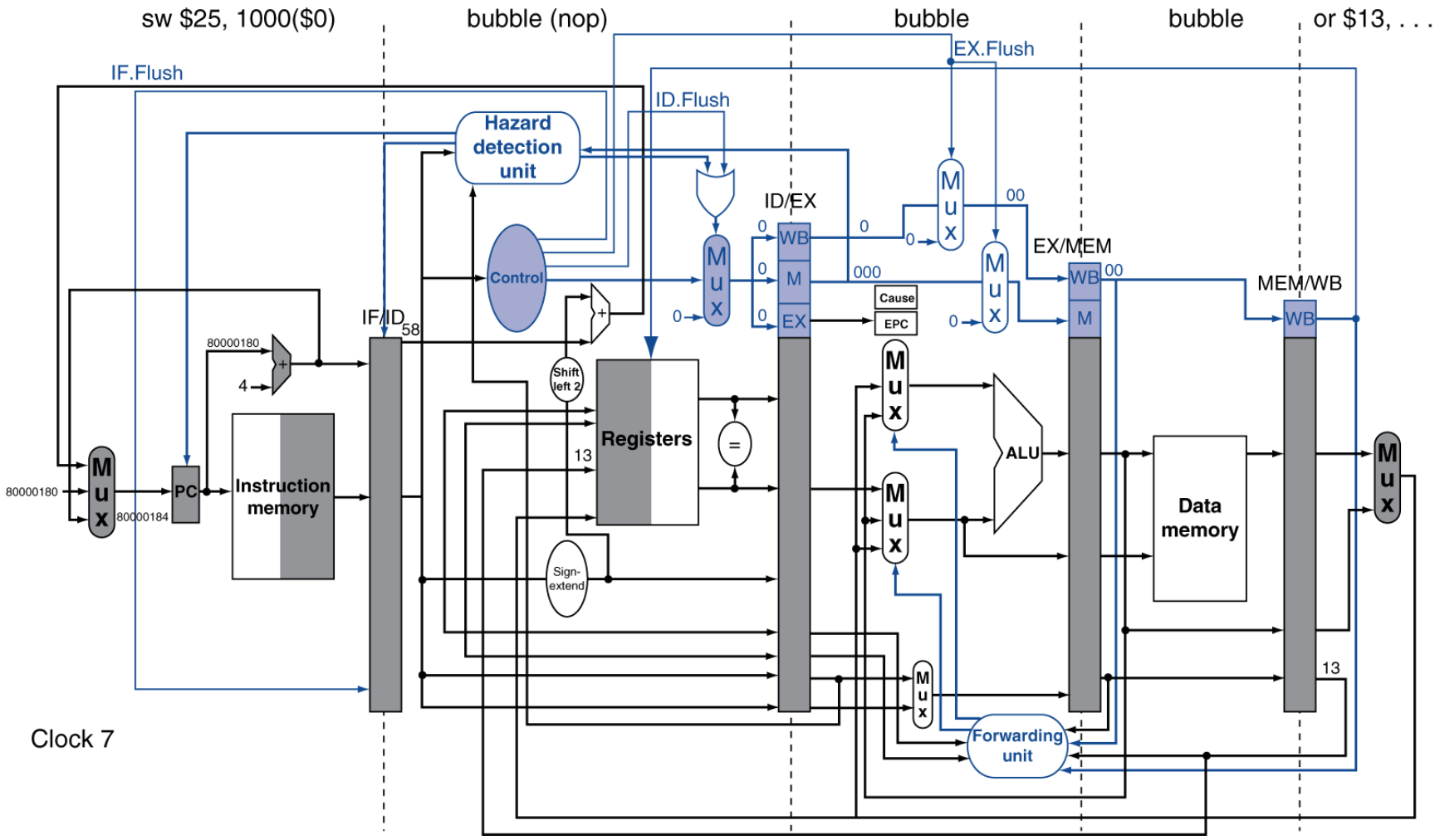
Exception Example

- Exception on add in
  - 40 sub $11, $2, $4
    44 and $12, $2, $5
    48 or $13, $2, $6
    4C add $1, $2, $1
    50 slt $15, $6, $7
    54 lw $16, 50($7)
    …
- Handler
  - 80000180 sw $25, 1000($0)
    80000184 sw $26, 1004($0)
    …

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

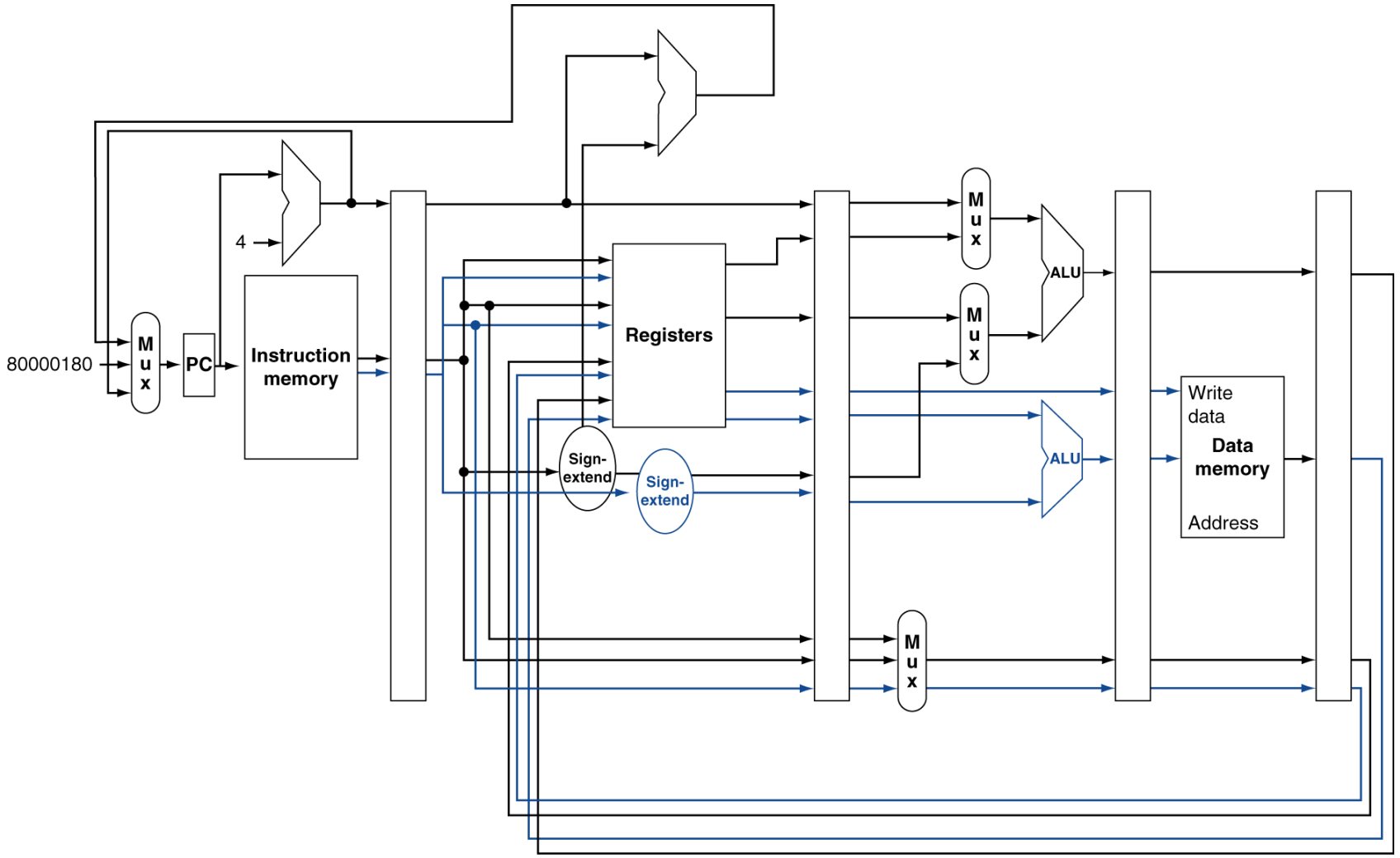
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Hazards in the Dual-issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - add $t0, $s0, $s1
      load $s2, 0($t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
  - More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling Example

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

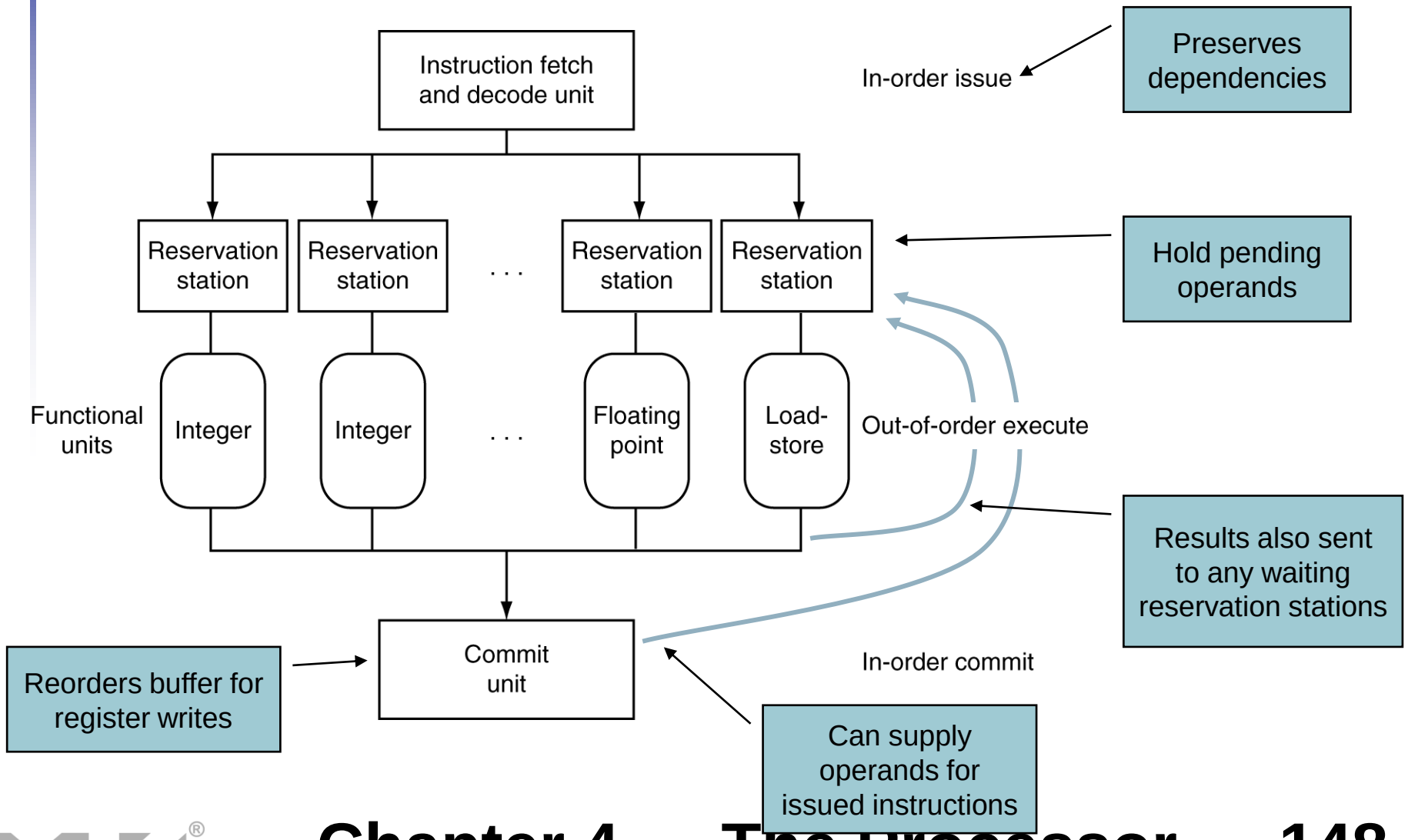
Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example
  - lw $t0, 20($s2)
    addu $t1, $t0, $t2
    sub $s4, $s4, $t3
    slti $t5, $s4, 20
  - Can start sub while addu is waiting for lw

Dynamically Scheduled CPU



Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don’t commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture as much as we'd like

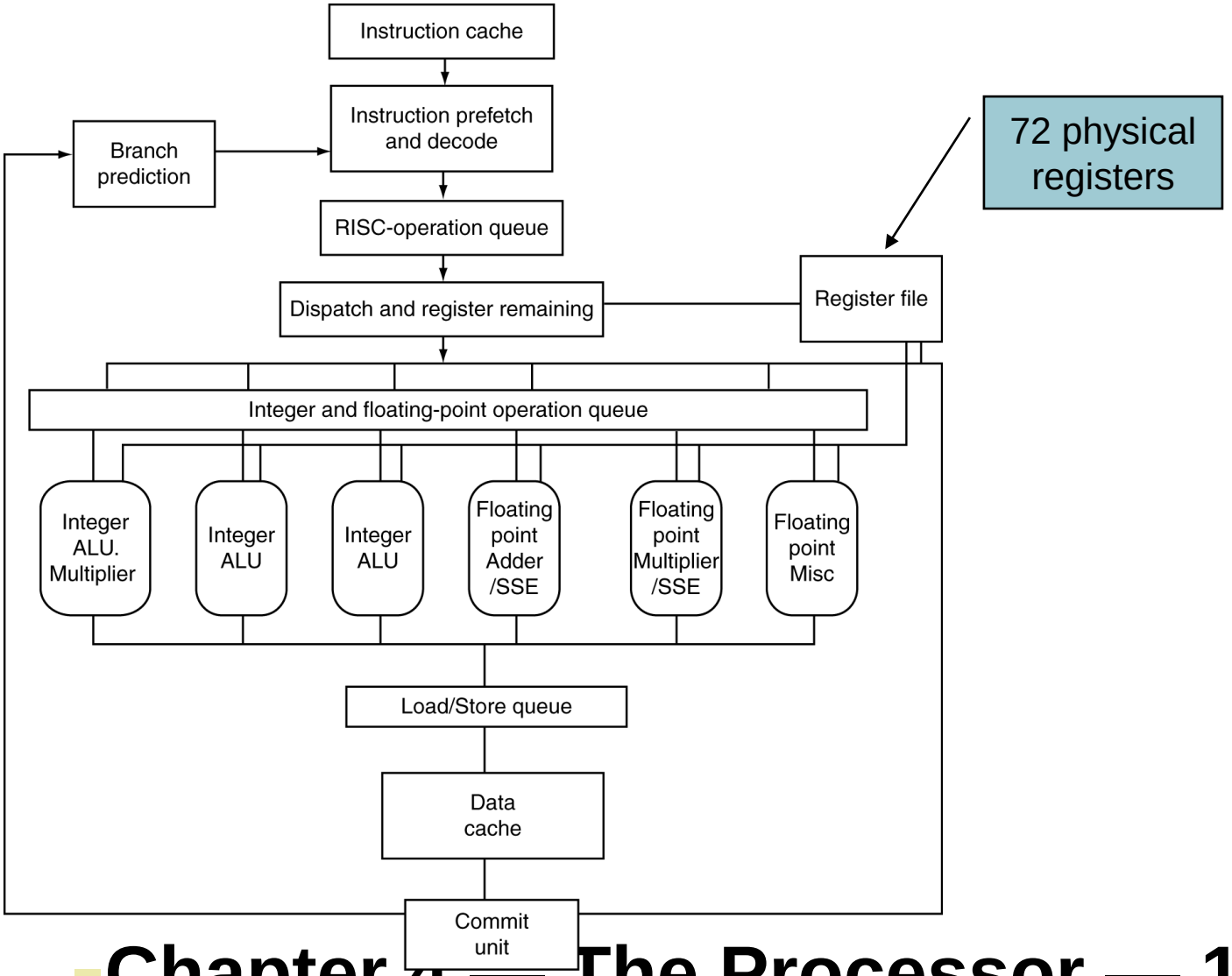
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

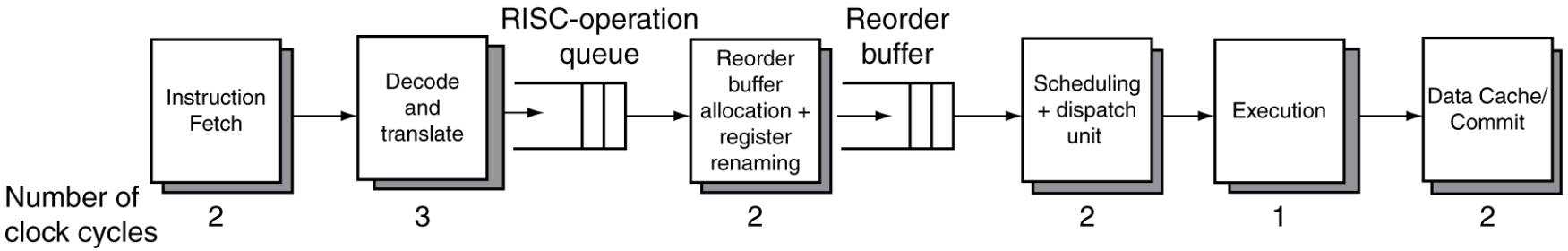
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Microarchitecture



The Opteron X4 Pipeline Flow

- For integer operations



  - FP is 5 stages longer
  - Up to 106 RISC-ops in progress
- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays

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
- Multiple issue and dynamic scheduling (ILP)