Chapter 4

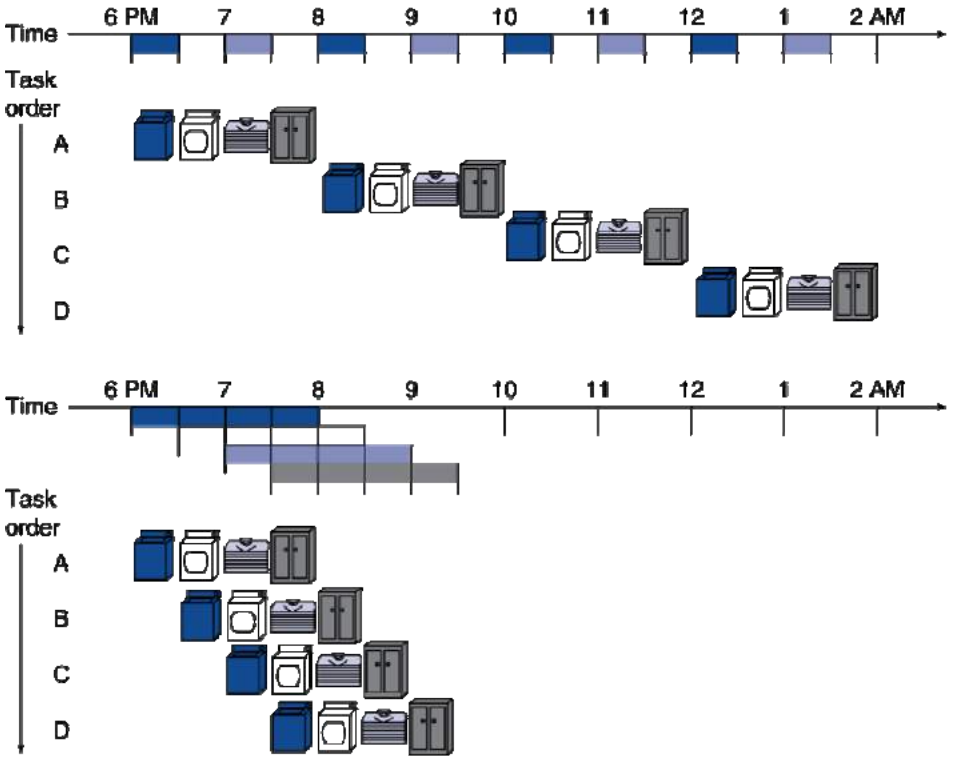
The Processor

Performance Issues

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

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

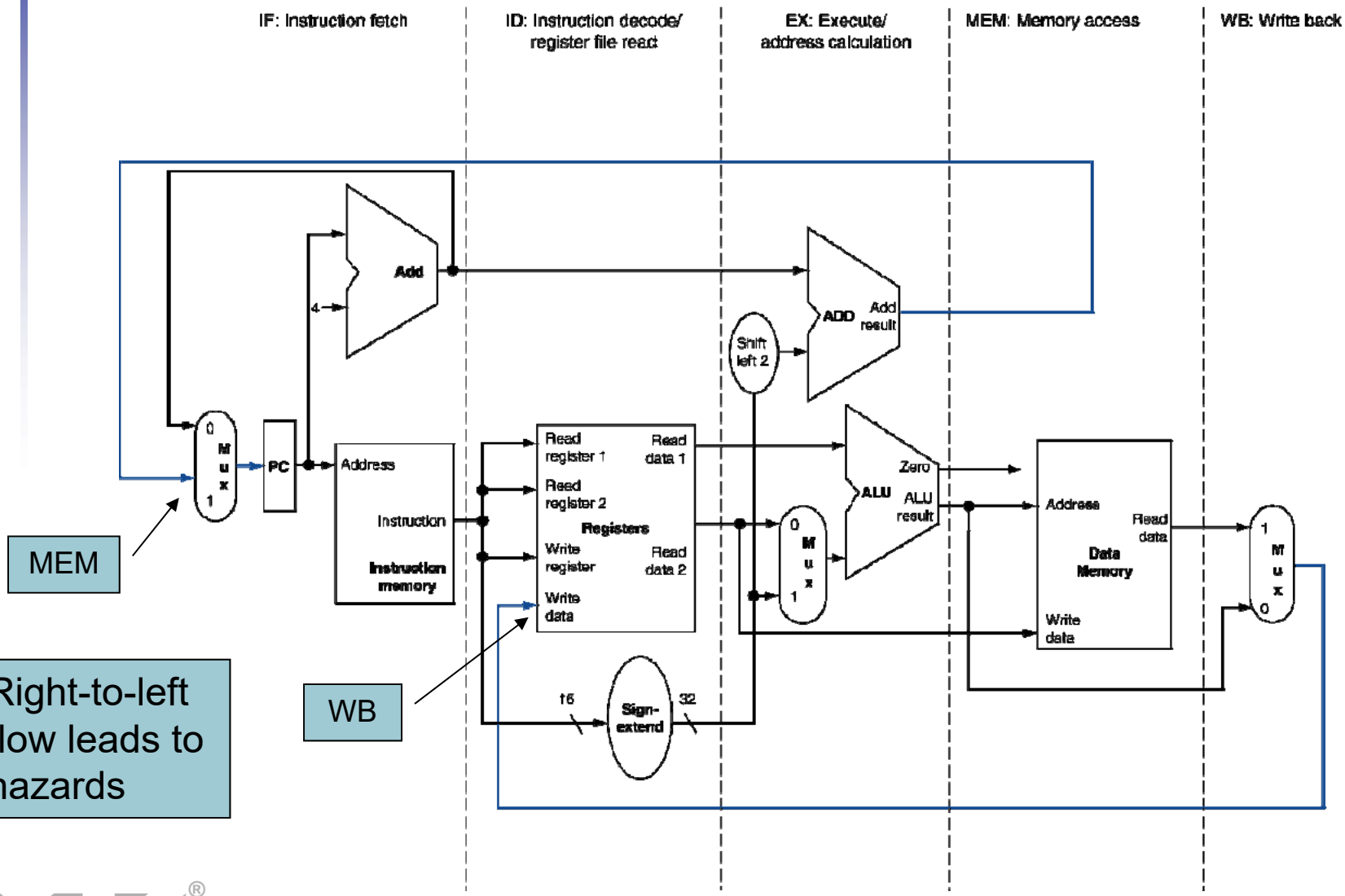


- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

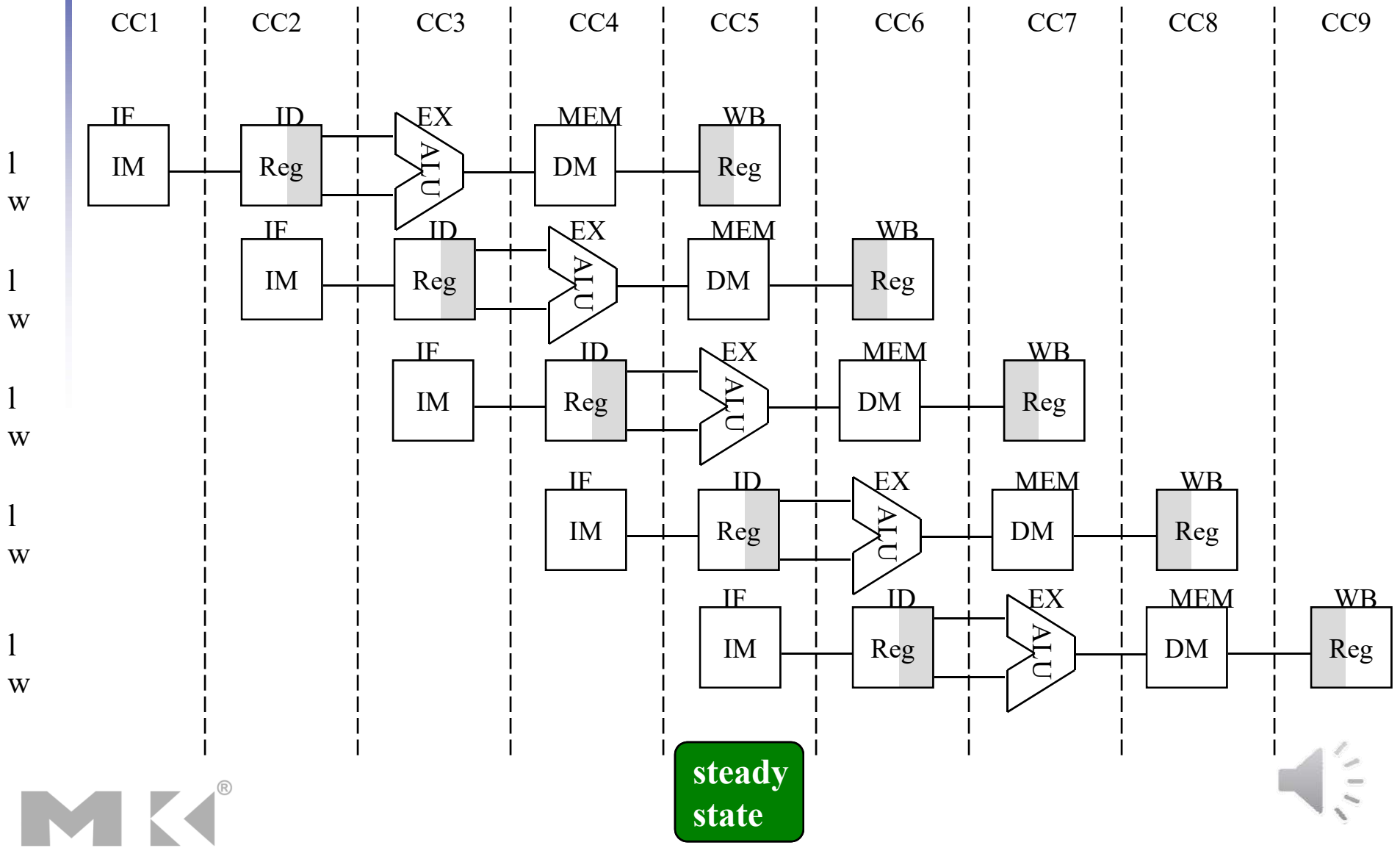
MIPS Pipeline

- 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

MIPS Pipelined Datapath



Execution in a Pipelined Datapath



Chapter 4

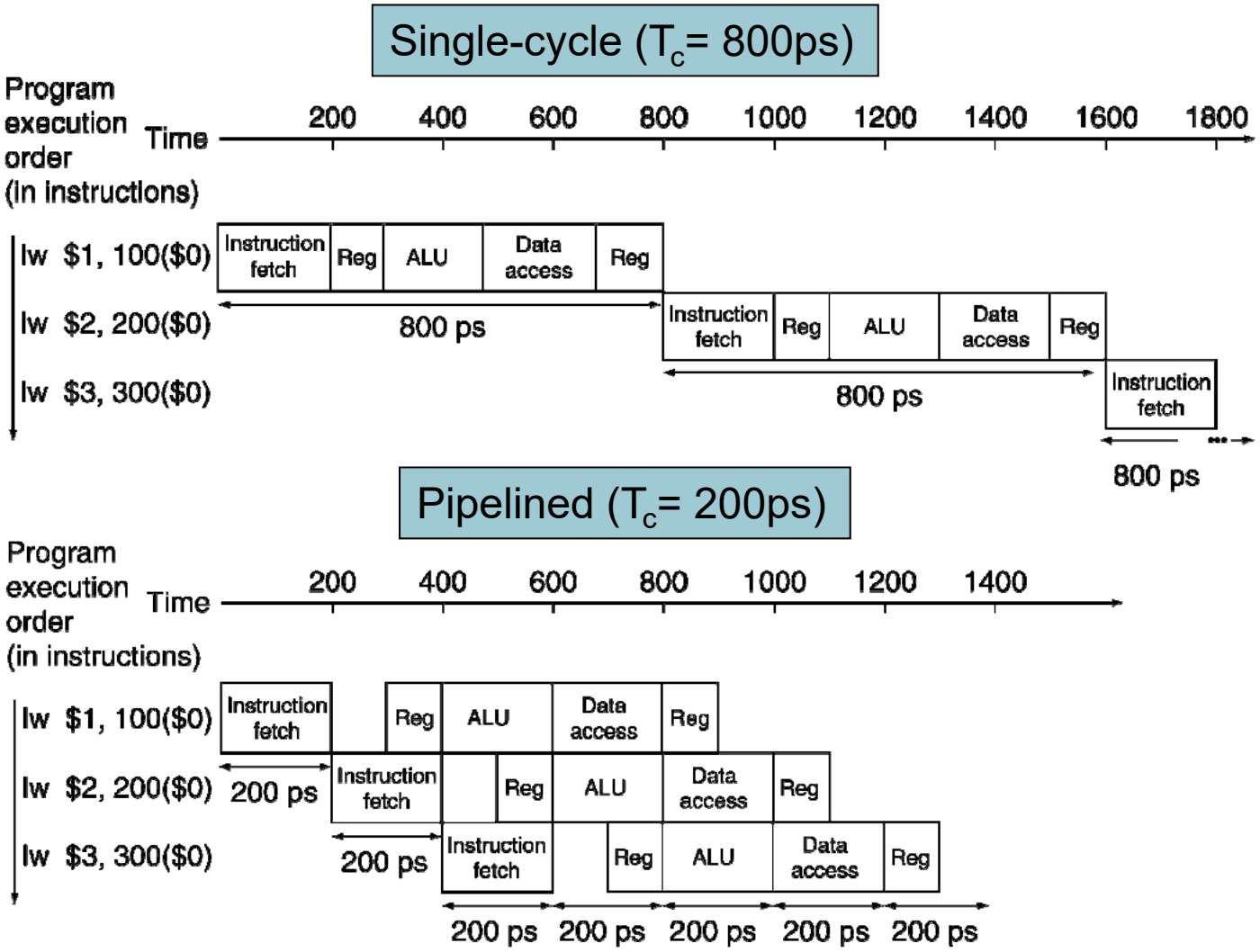
The Processor

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

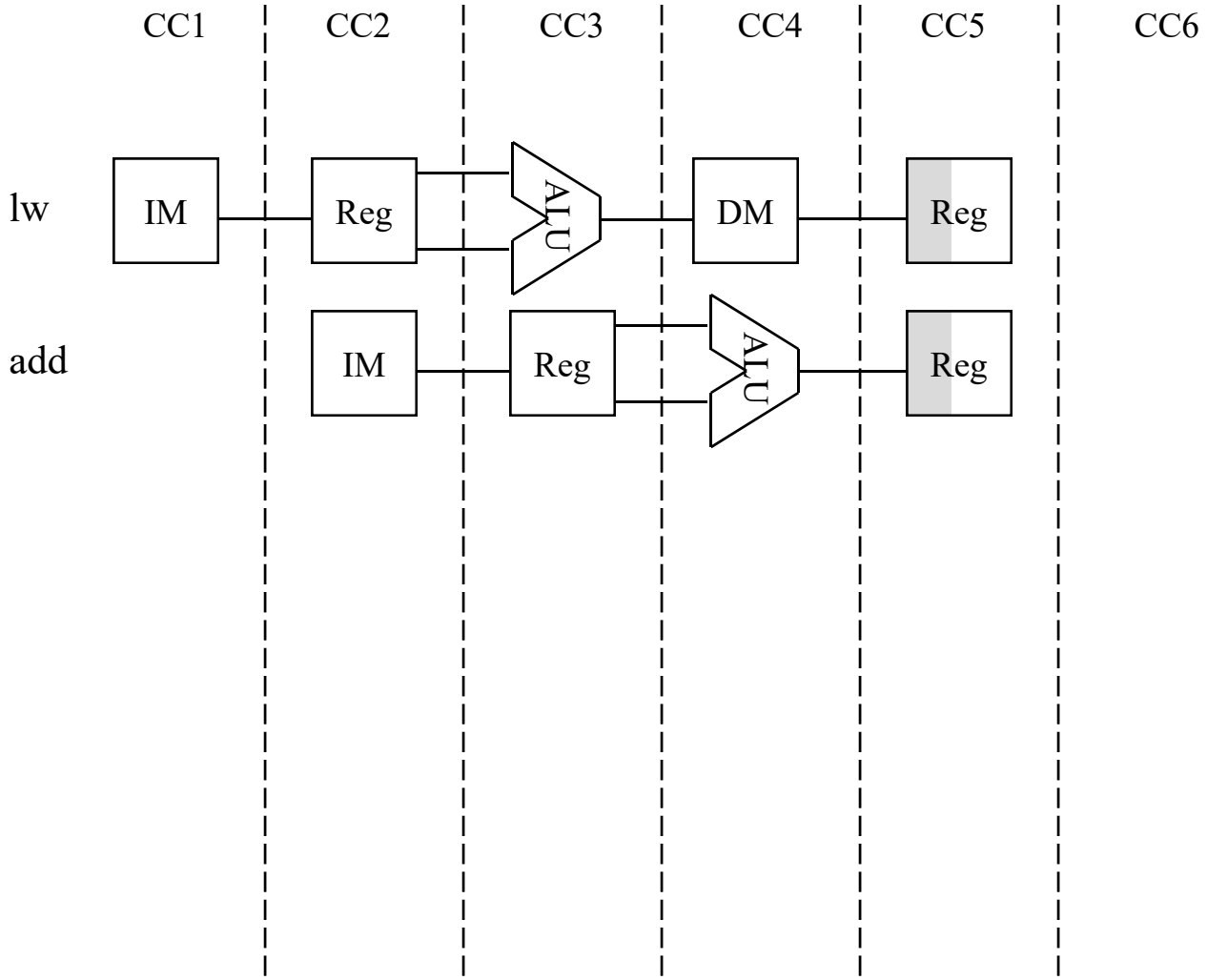
Pipeline Performance



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Mixed Instructions in the Pipeline

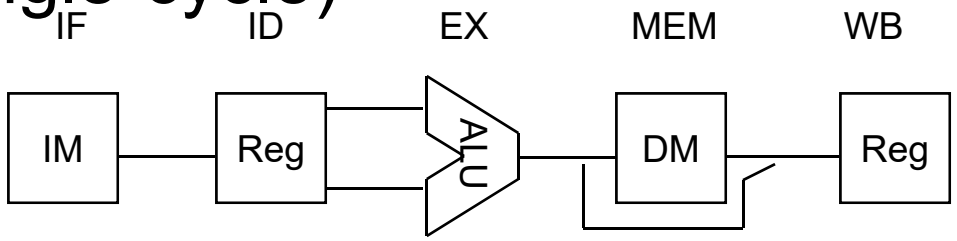


Pipelining and ISA Design

- 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

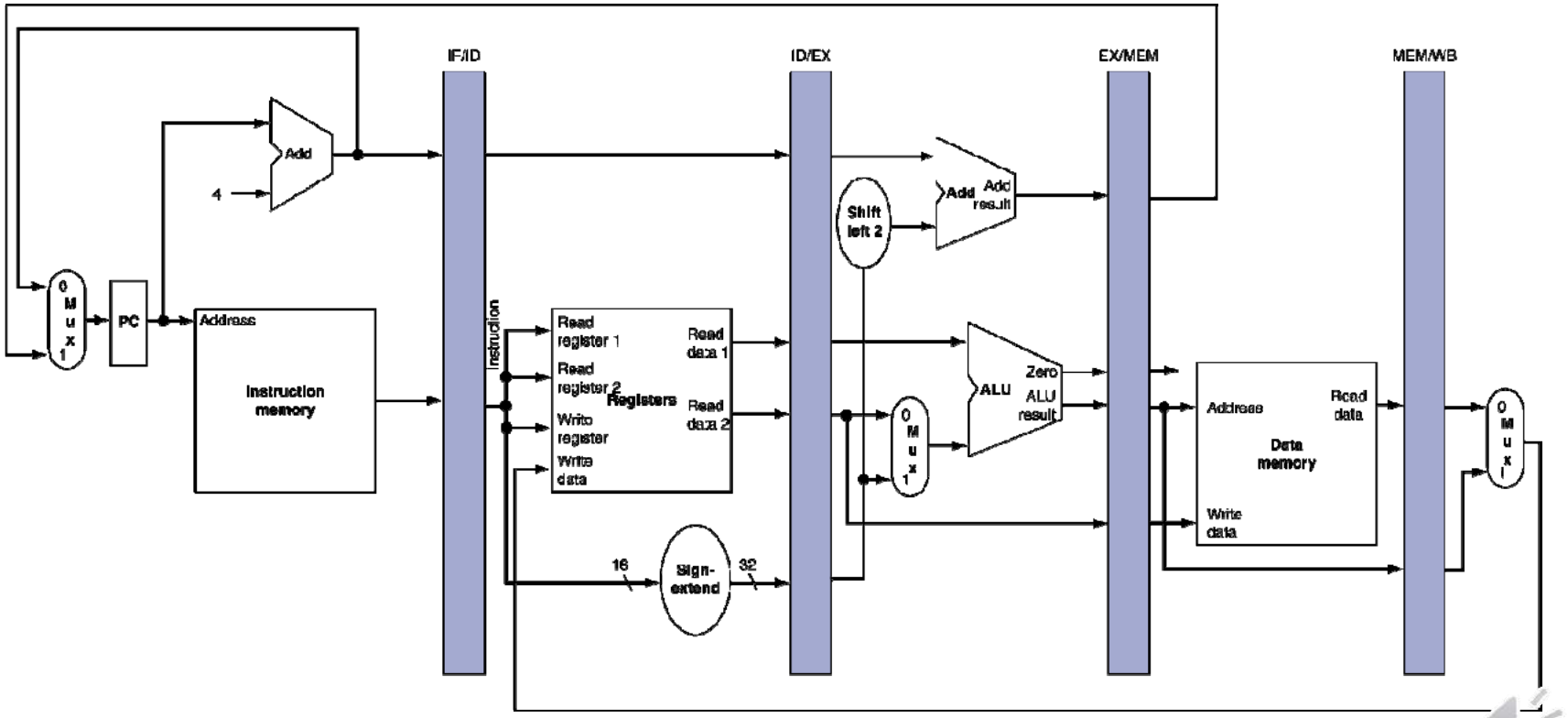
Pipeline Principles

- All instructions that share a pipeline must have the same stages in the same order.
  - therefore, add does nothing during Mem stage
  - sw does nothing during WB stage
- All intermediate values must be latched each cycle.
- There is no functional block reuse
  - example: we need 2 adders and ALU (like in single-cycle)



Pipeline registers

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



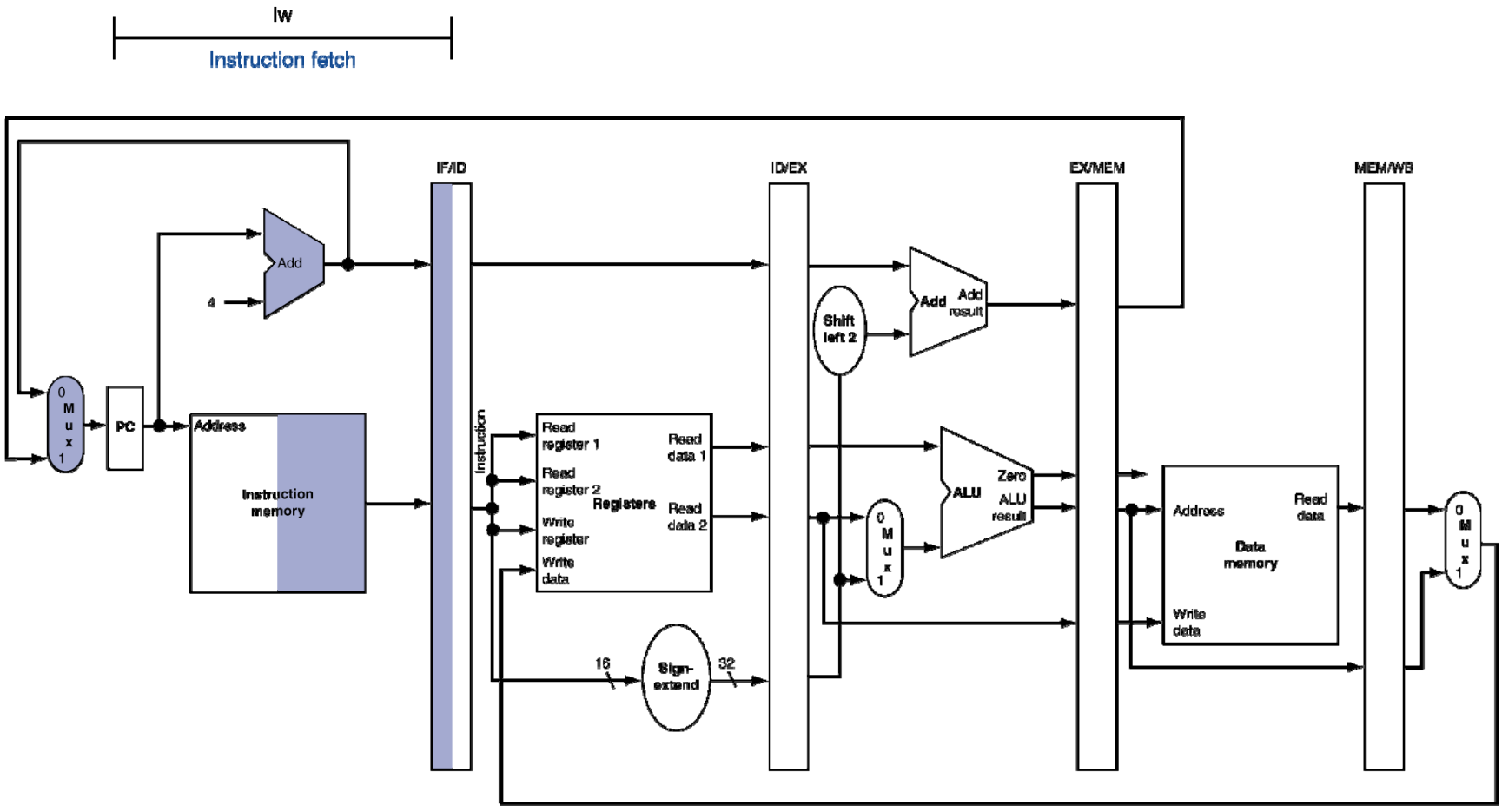
Chapter 4

The Processor

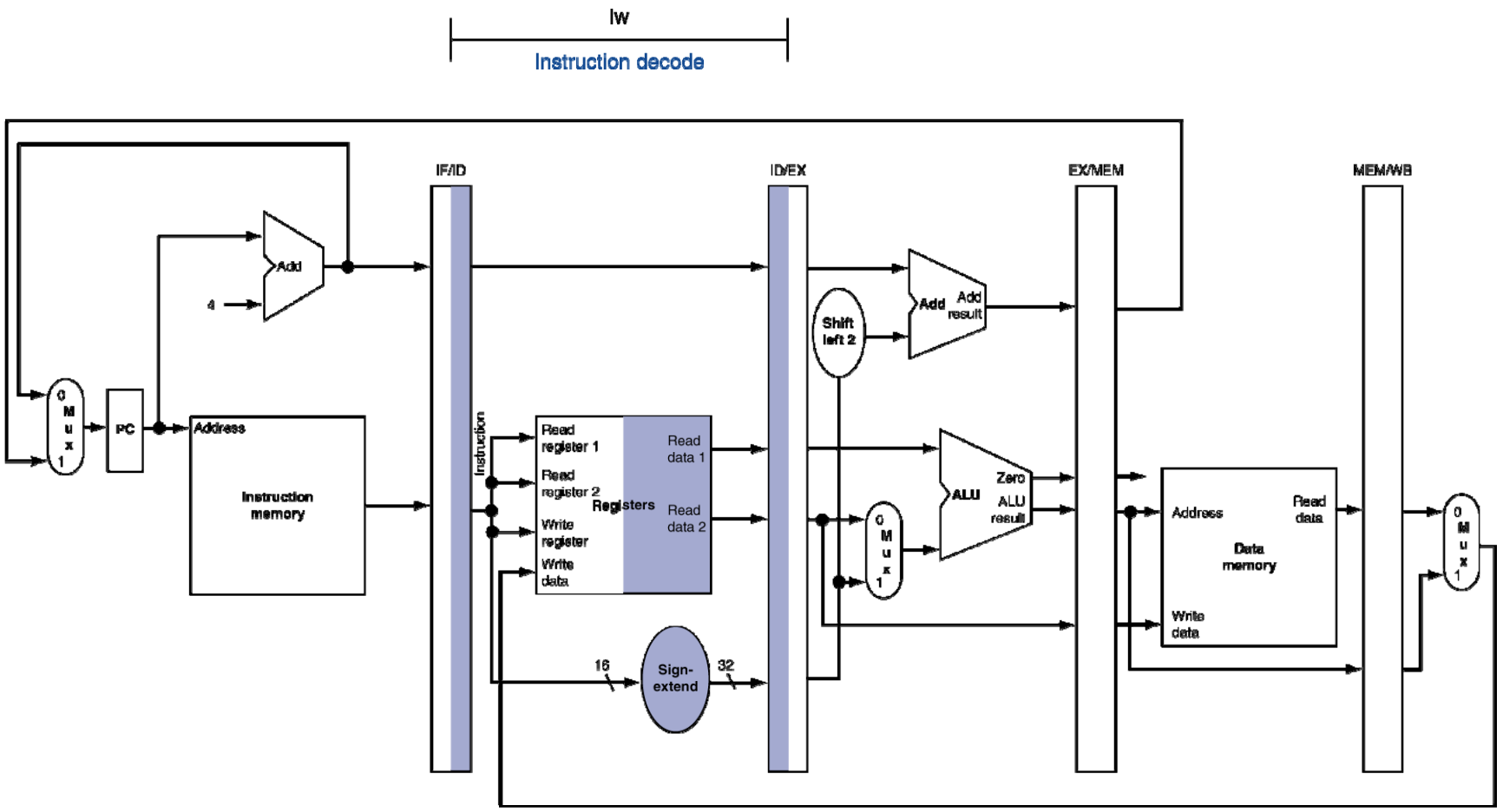
Pipeline Operation

- 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

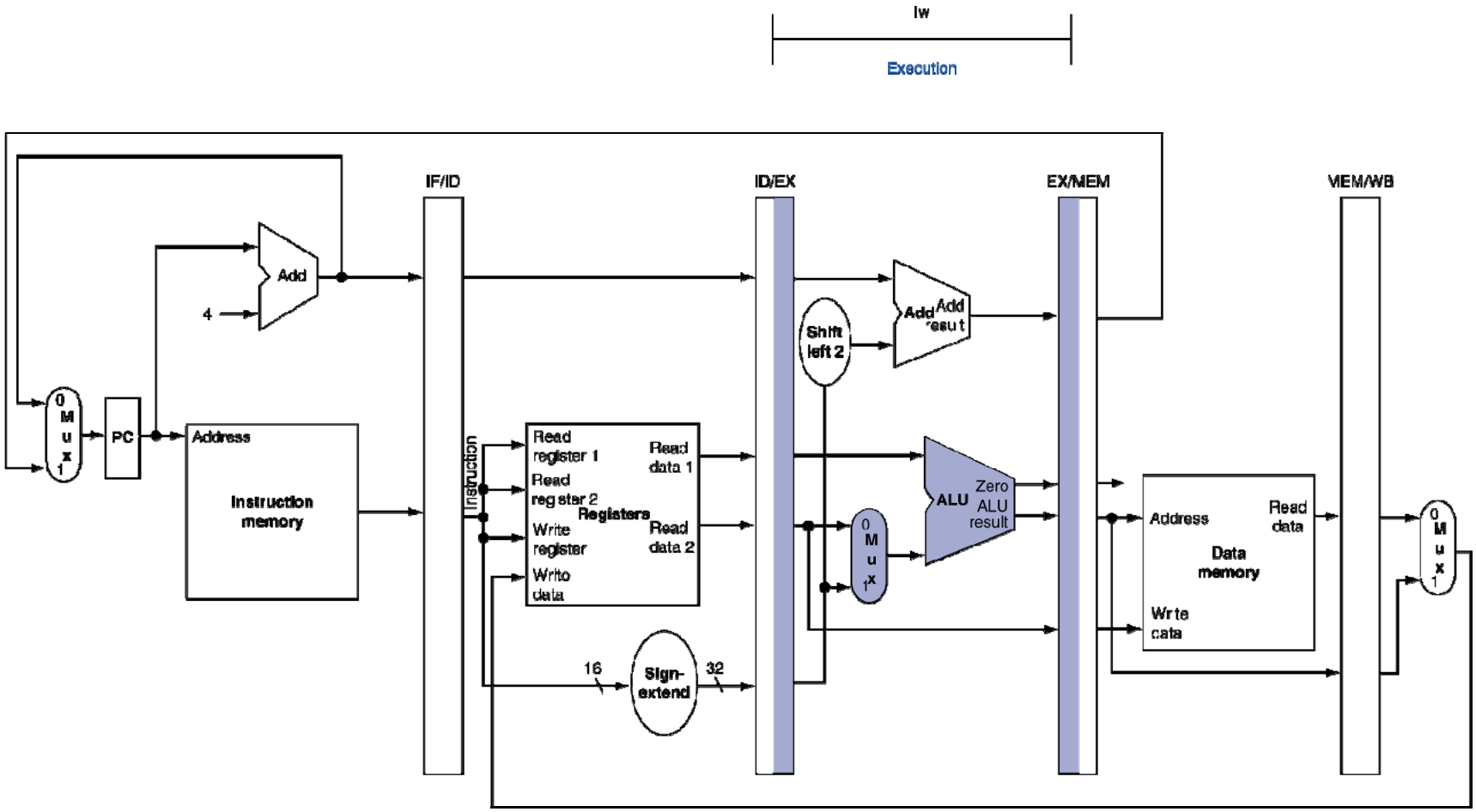
IF for Load, Store, …



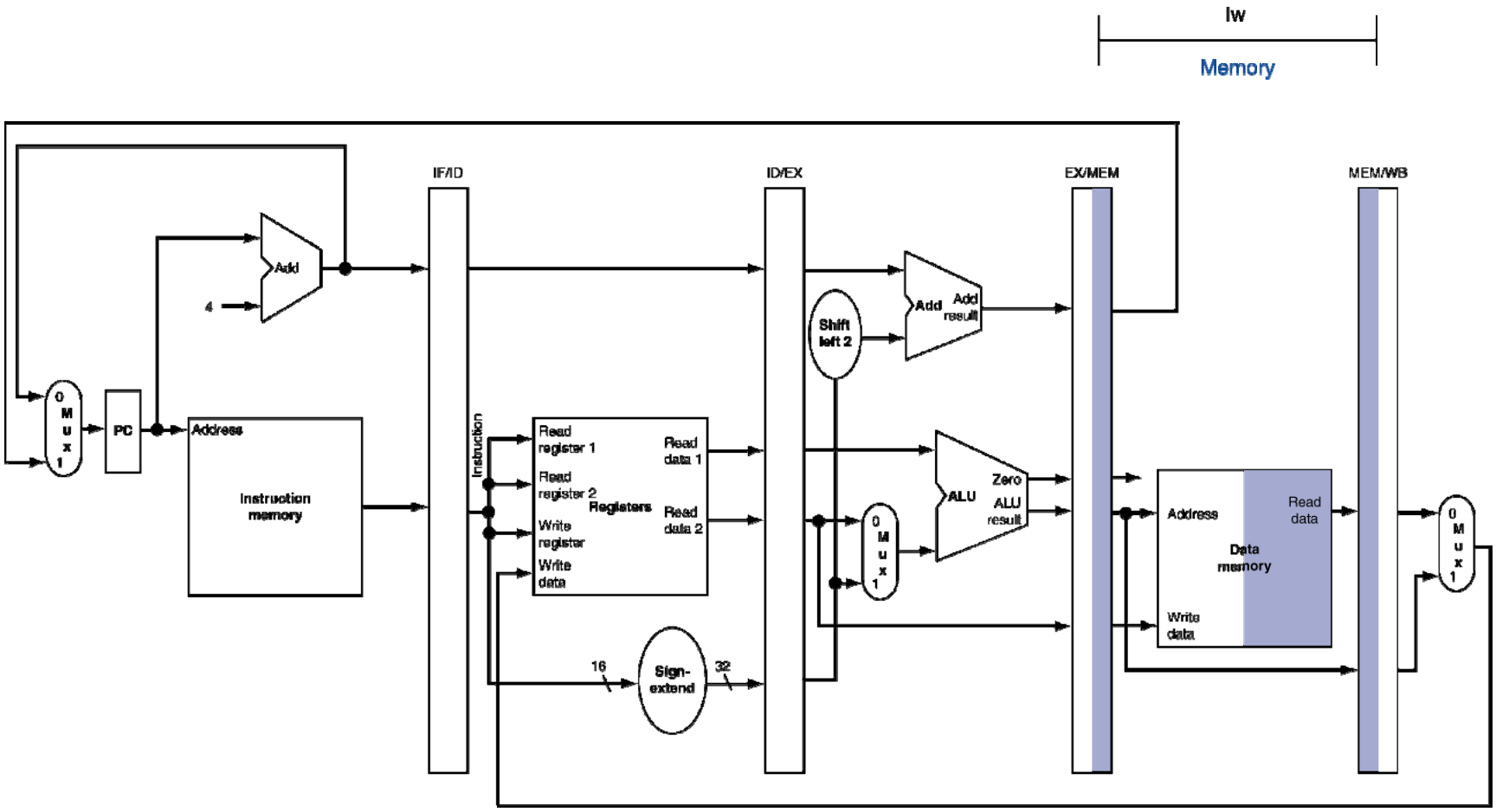
ID for Load, Store, …



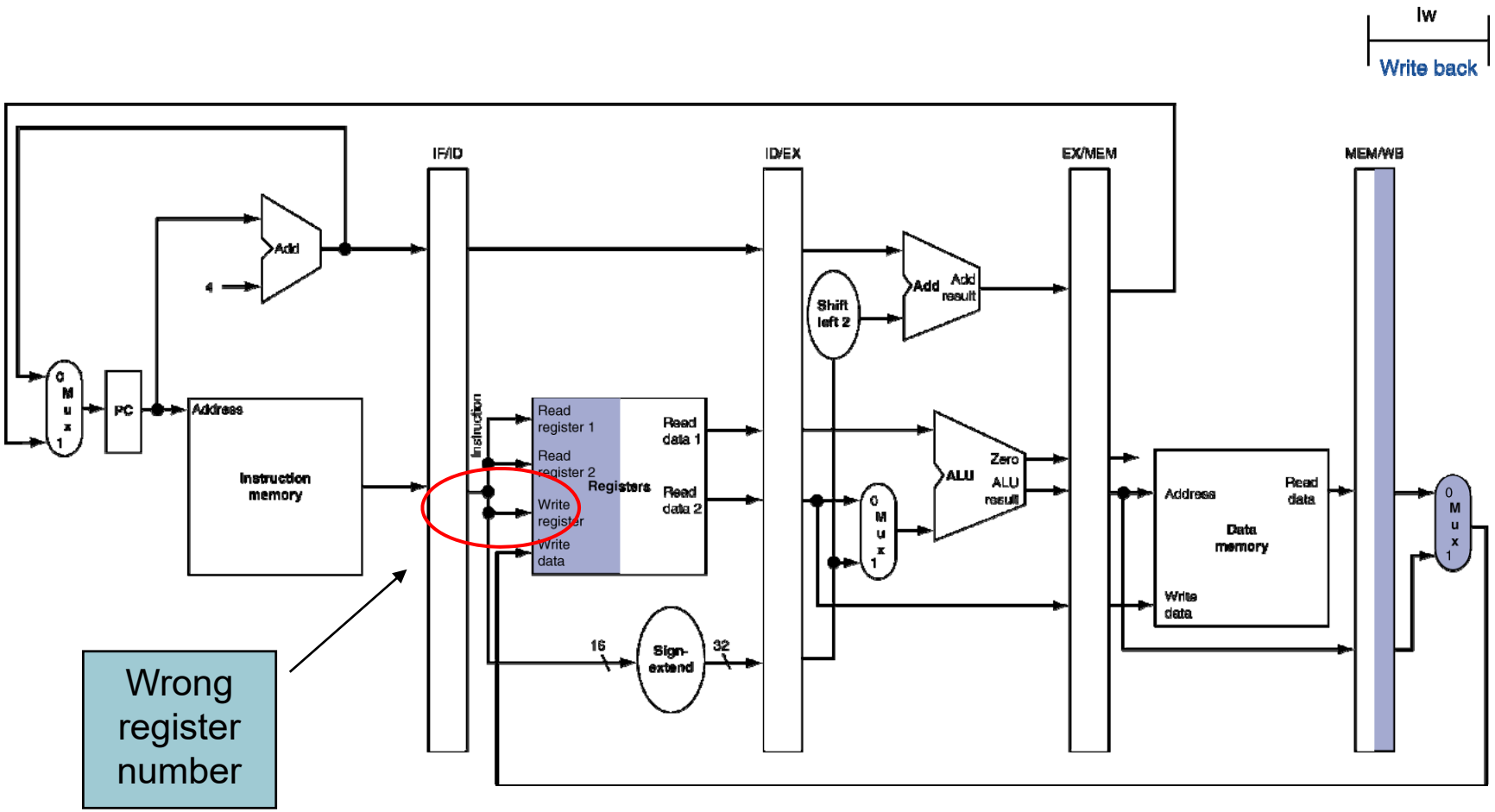
EX for Load


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

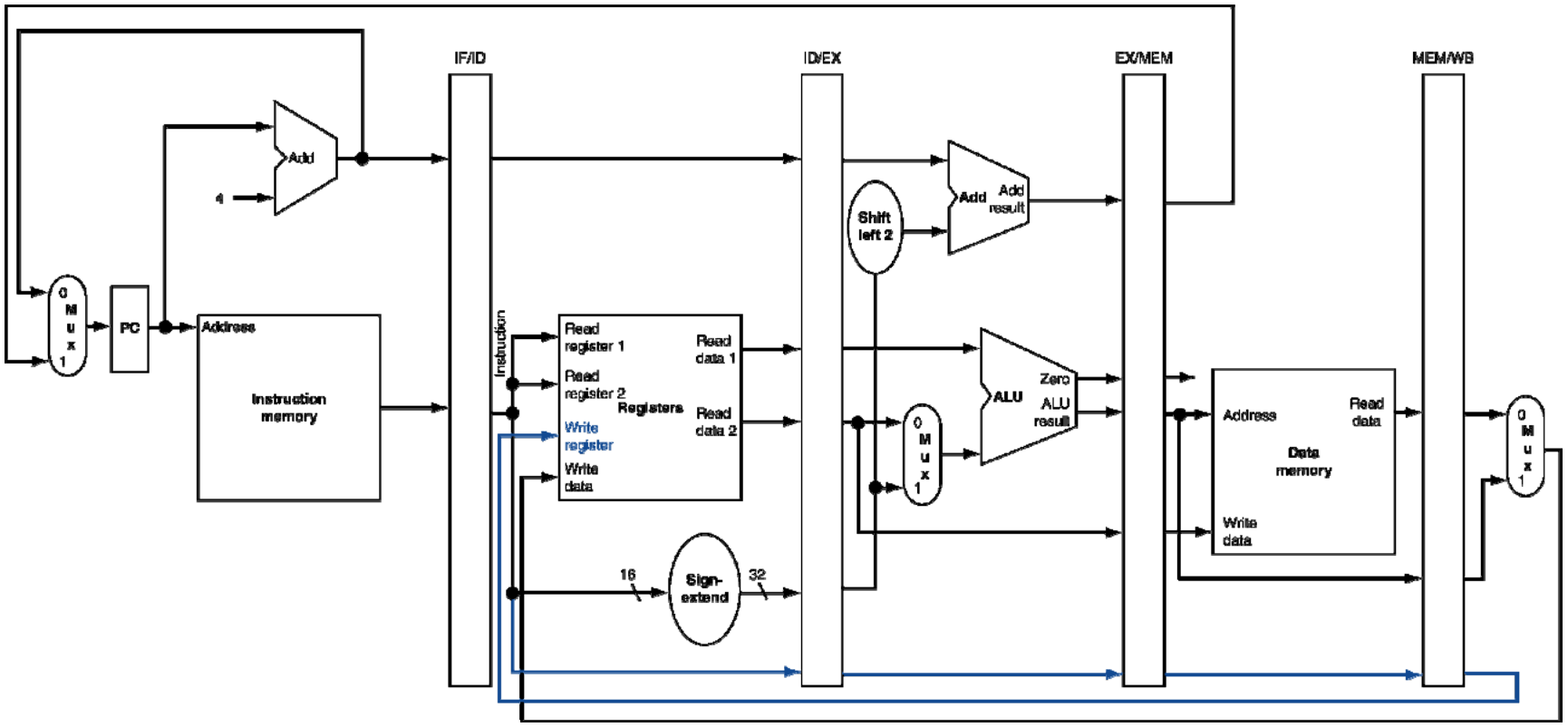
MEM for Load


lw
Memory
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

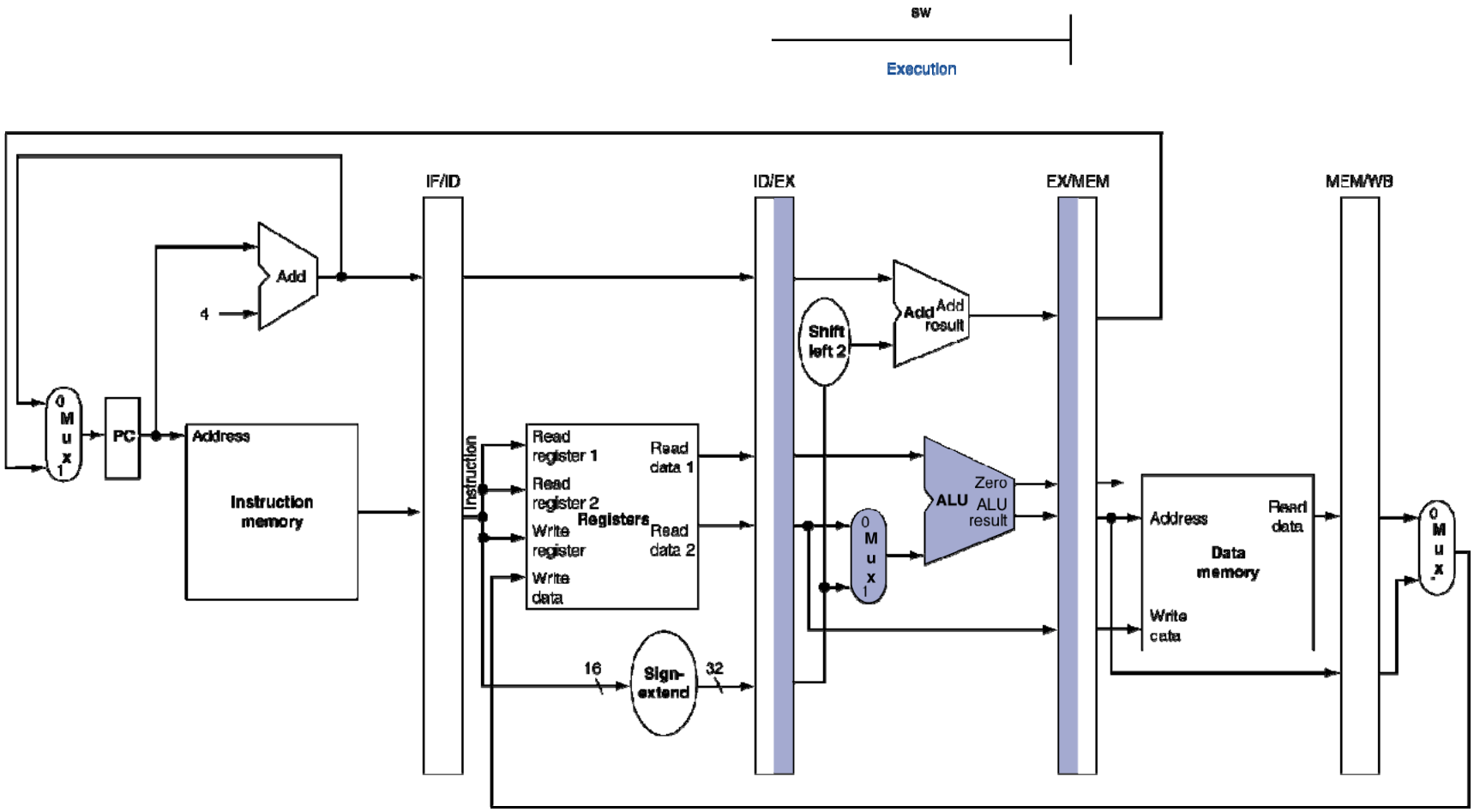
WB for Load


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

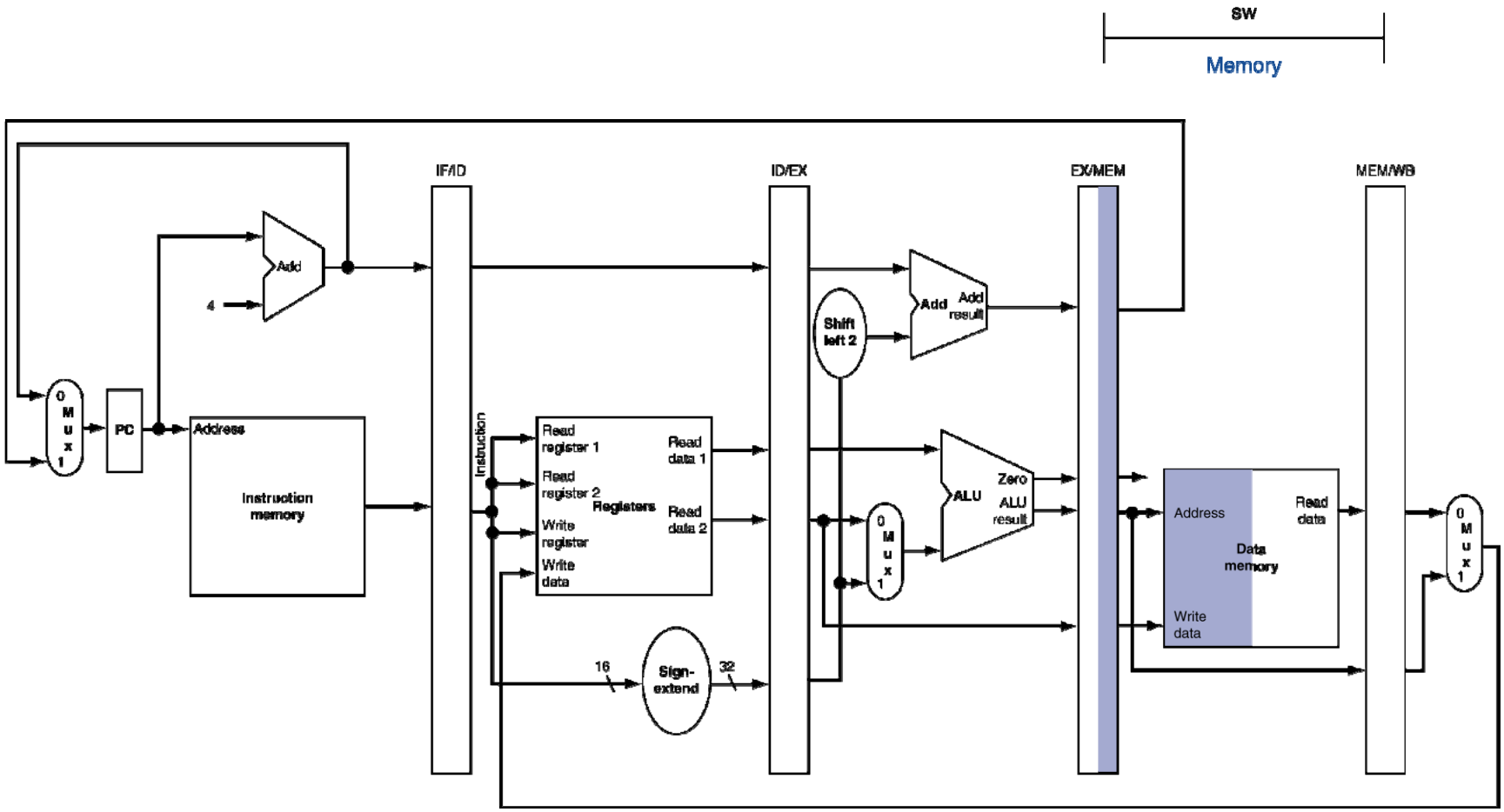
Corrected Datapath for Load



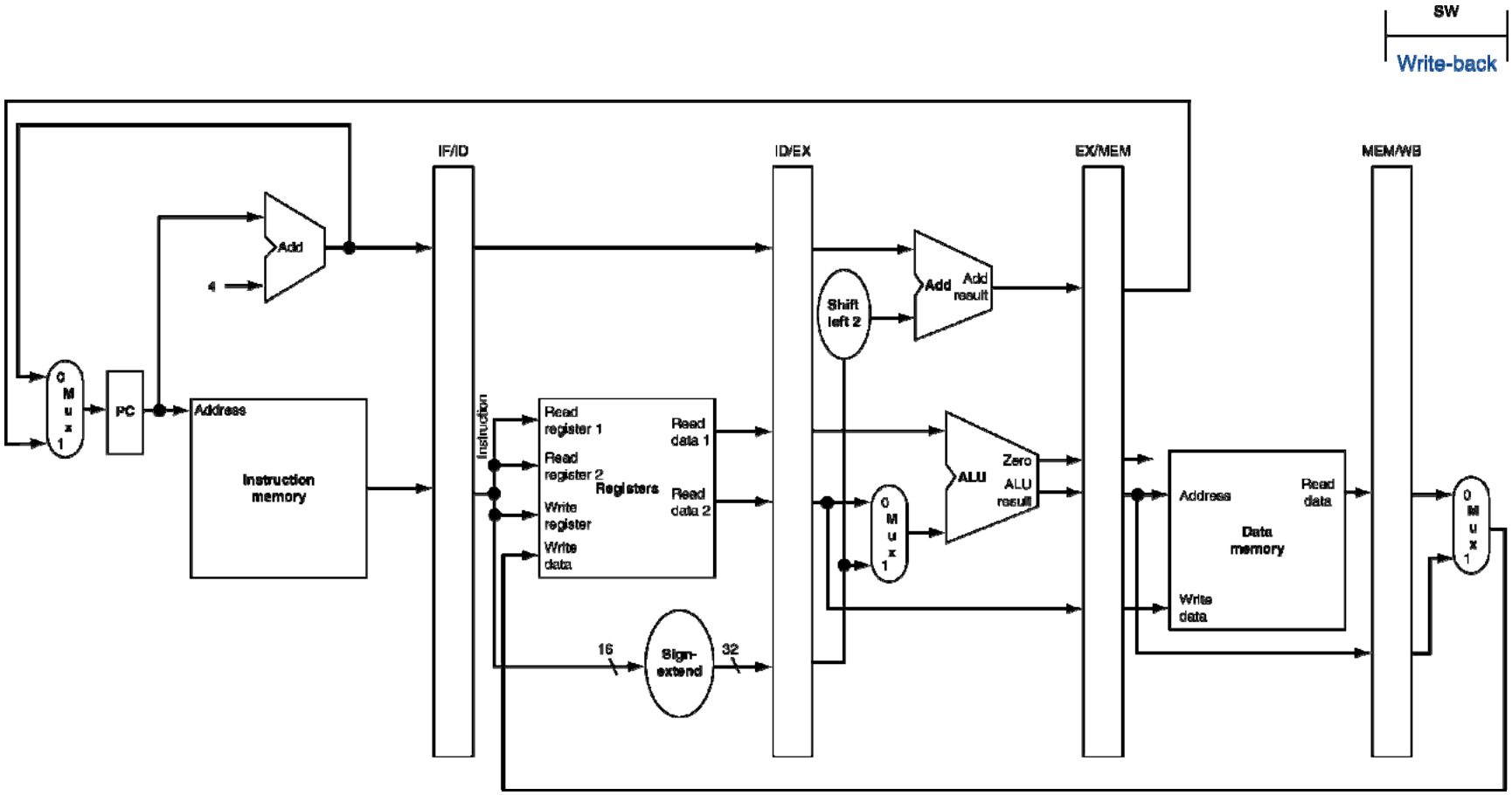
EX for Store


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

MEM for Store



WB for Store

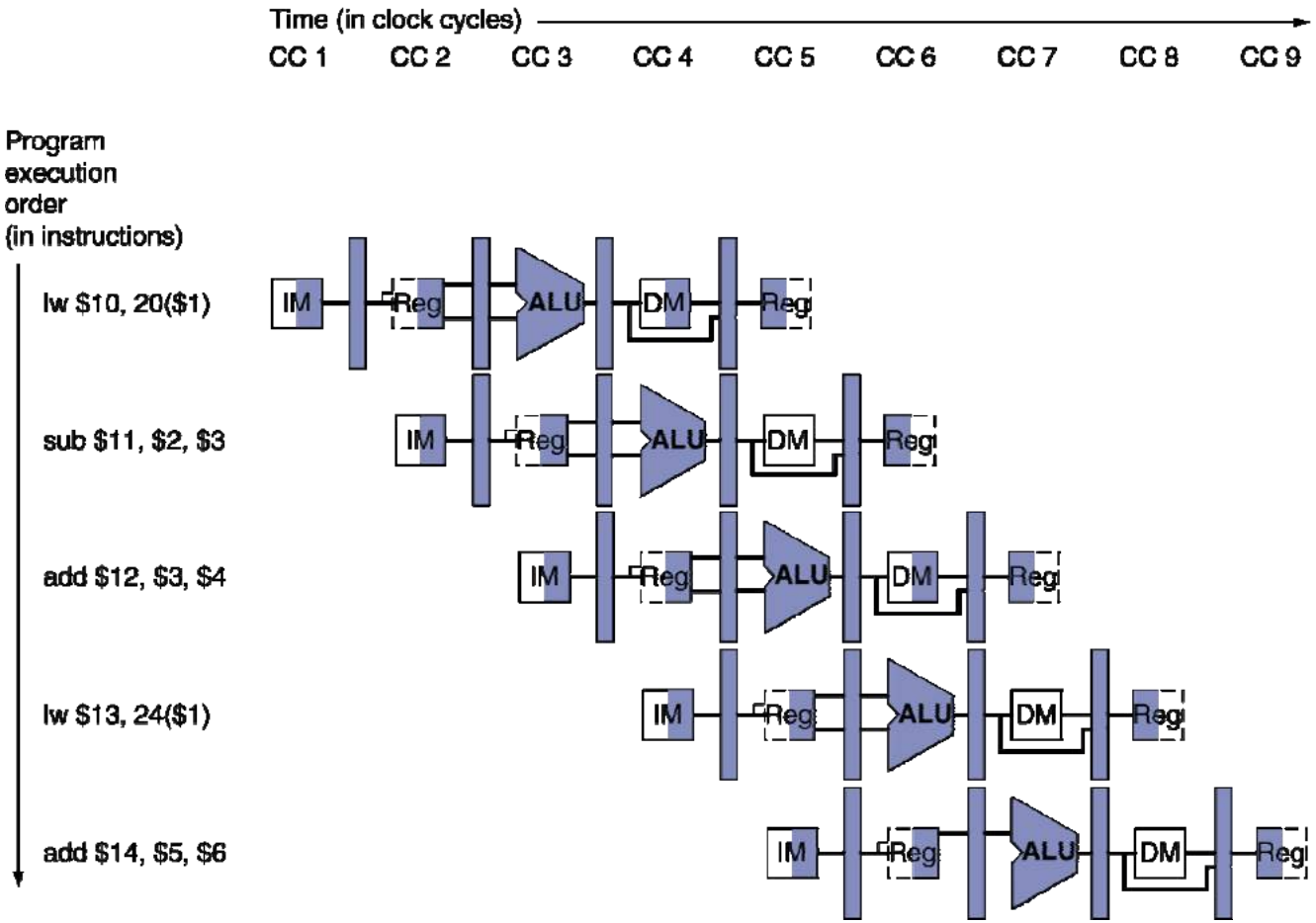

SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

Chapter 4

The Processor

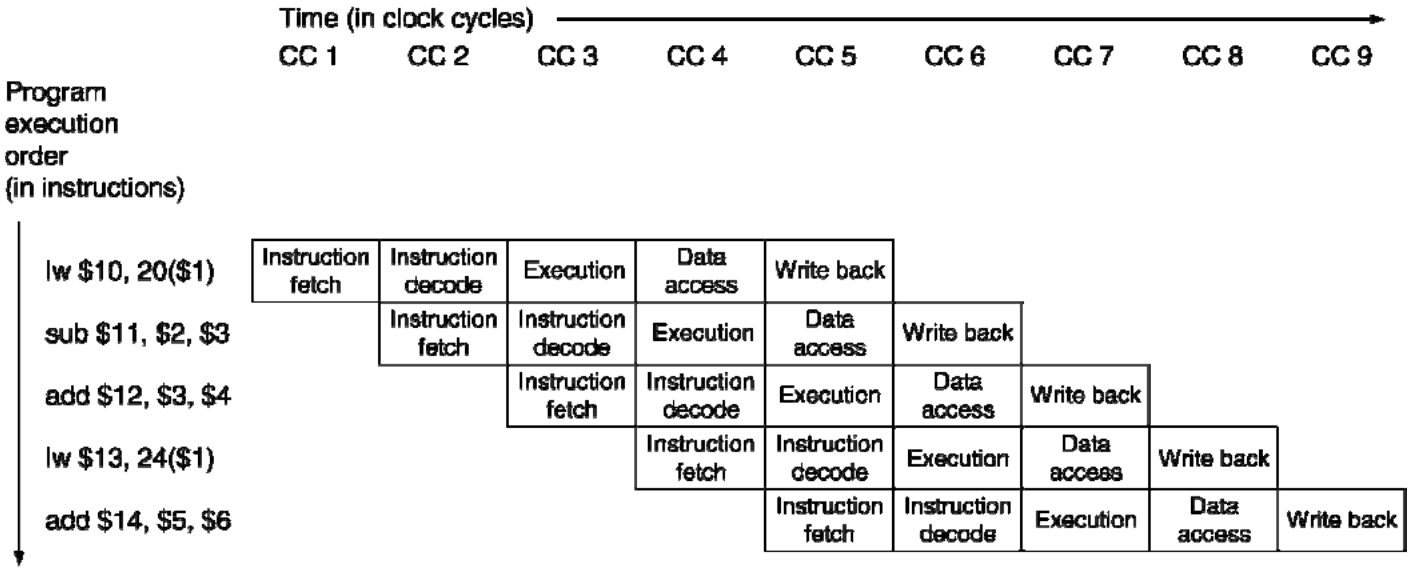
Multi-Cycle Pipeline Diagram

- Form showing resource usage



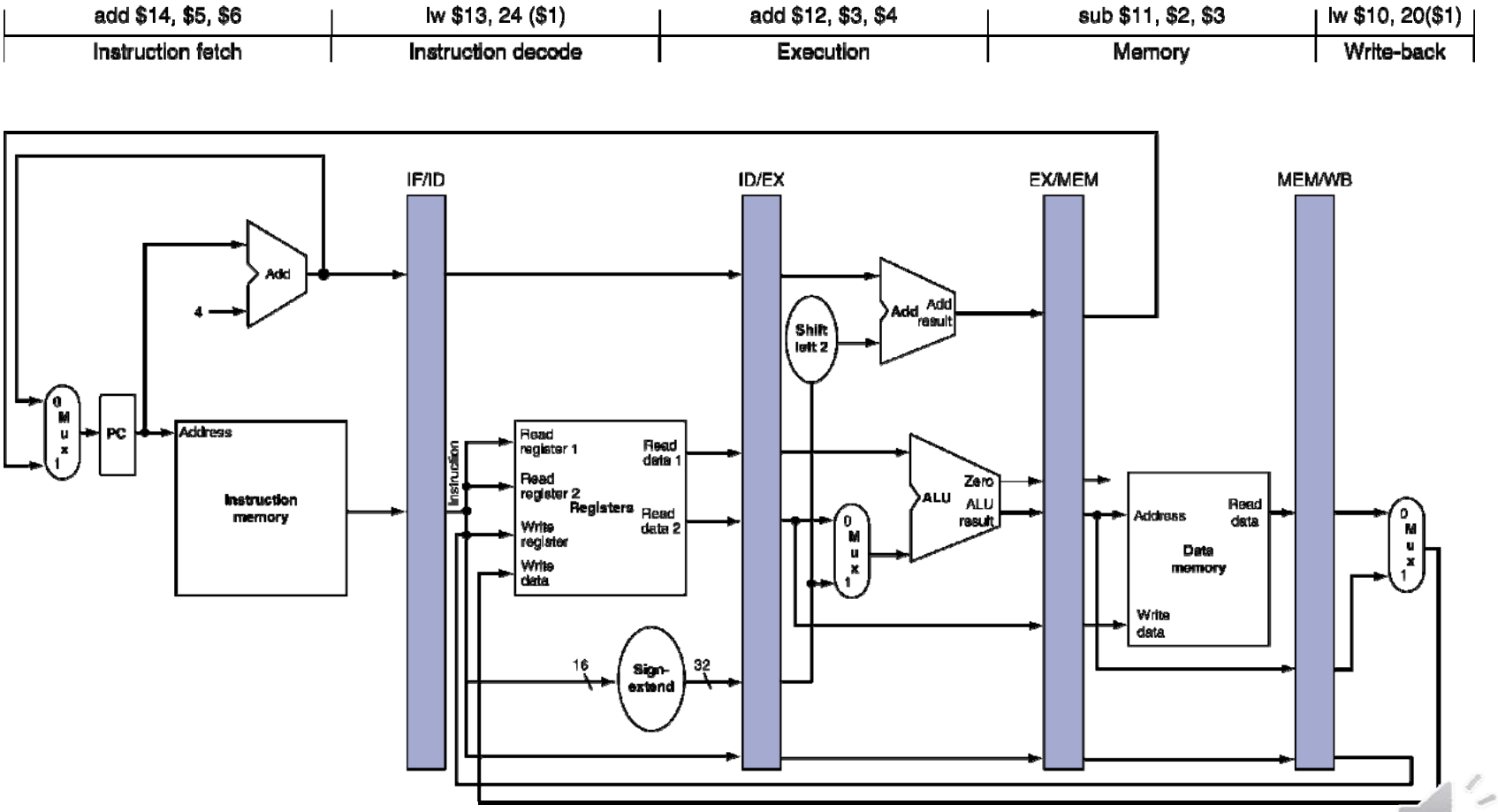
Multi-Cycle Pipeline Diagram

- Traditional form

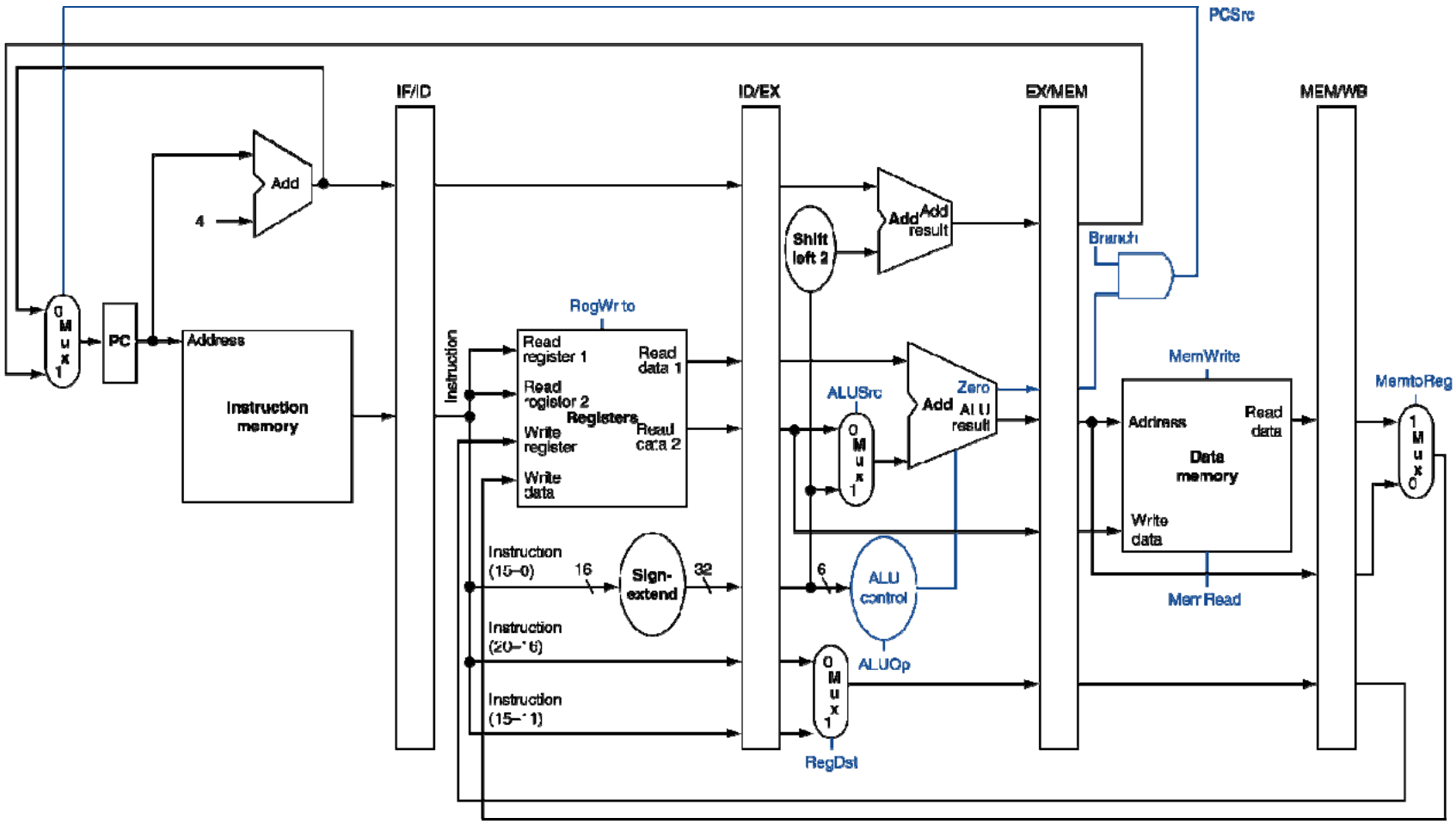


Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

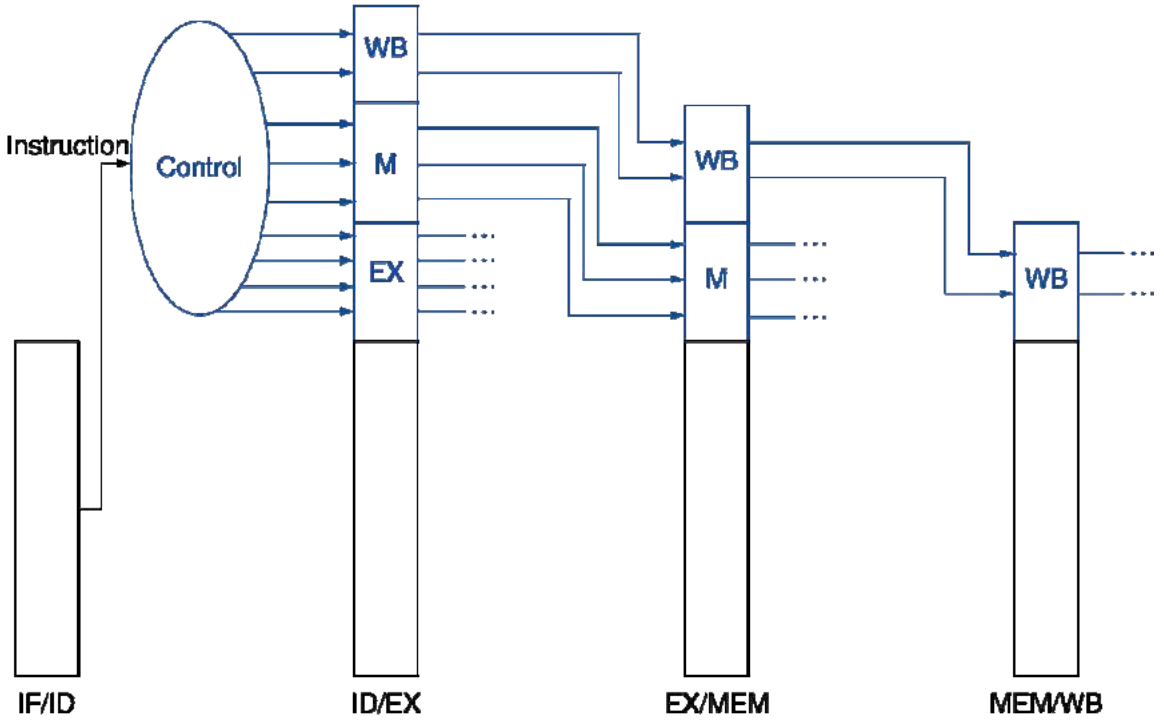


Pipelined Control (Simplified)

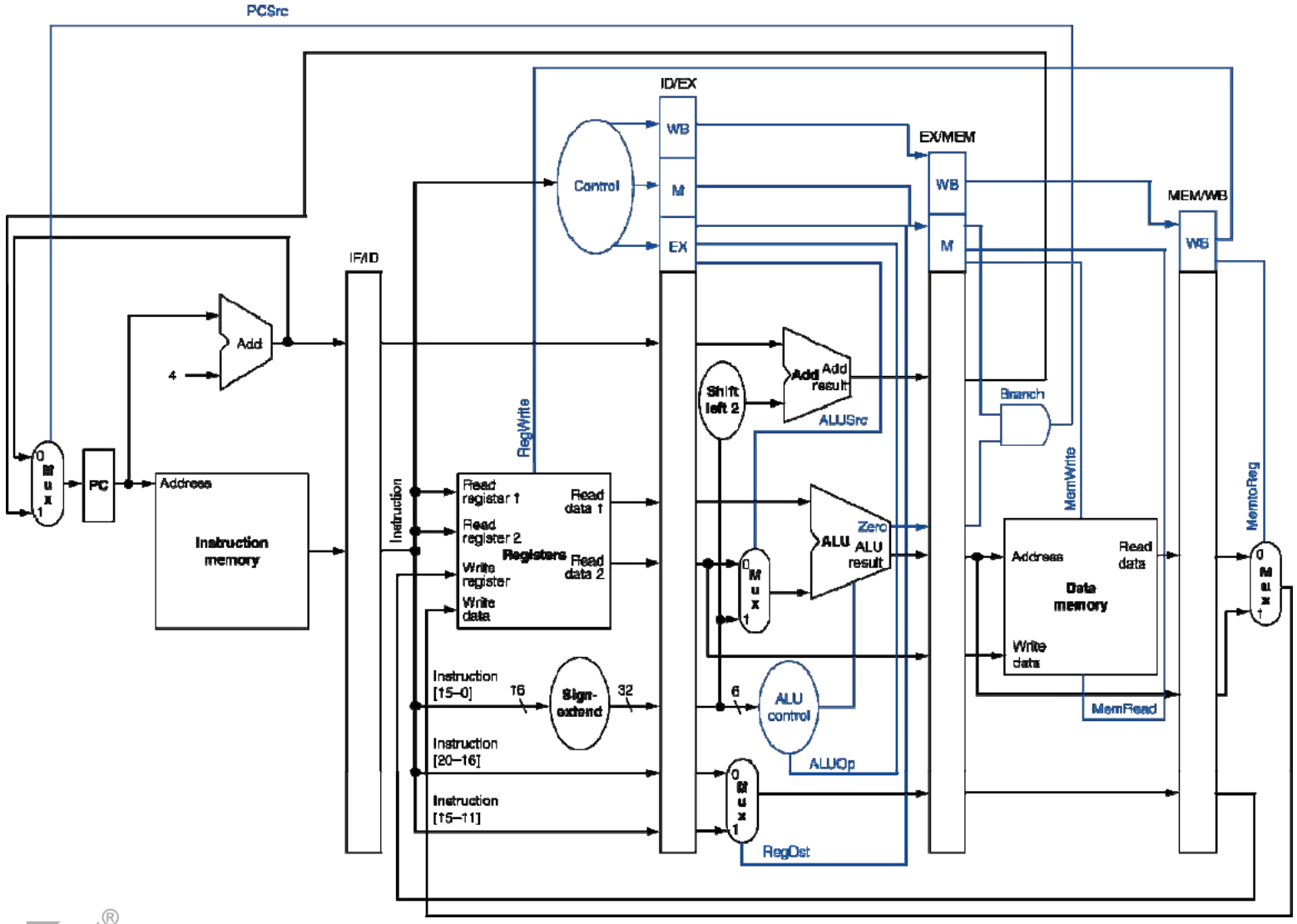


Pipelined Control

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



Pipelined Control



Pipelined Control Signals

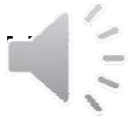
	Execution Stage Control Lines				Memory Stage Control Lines			Write Back Stage Control Lines	
Instruction	RegDst	ALU Op1	ALU Op0	ALUSrc	Branch	Mem Read	Mem Write	RegWrite	MemtoReg
R-Format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	x	0	0	1	0	0	1	0	x
beq	x	0	1	0	1	0	0	0	x

Chapter 4

The Processor

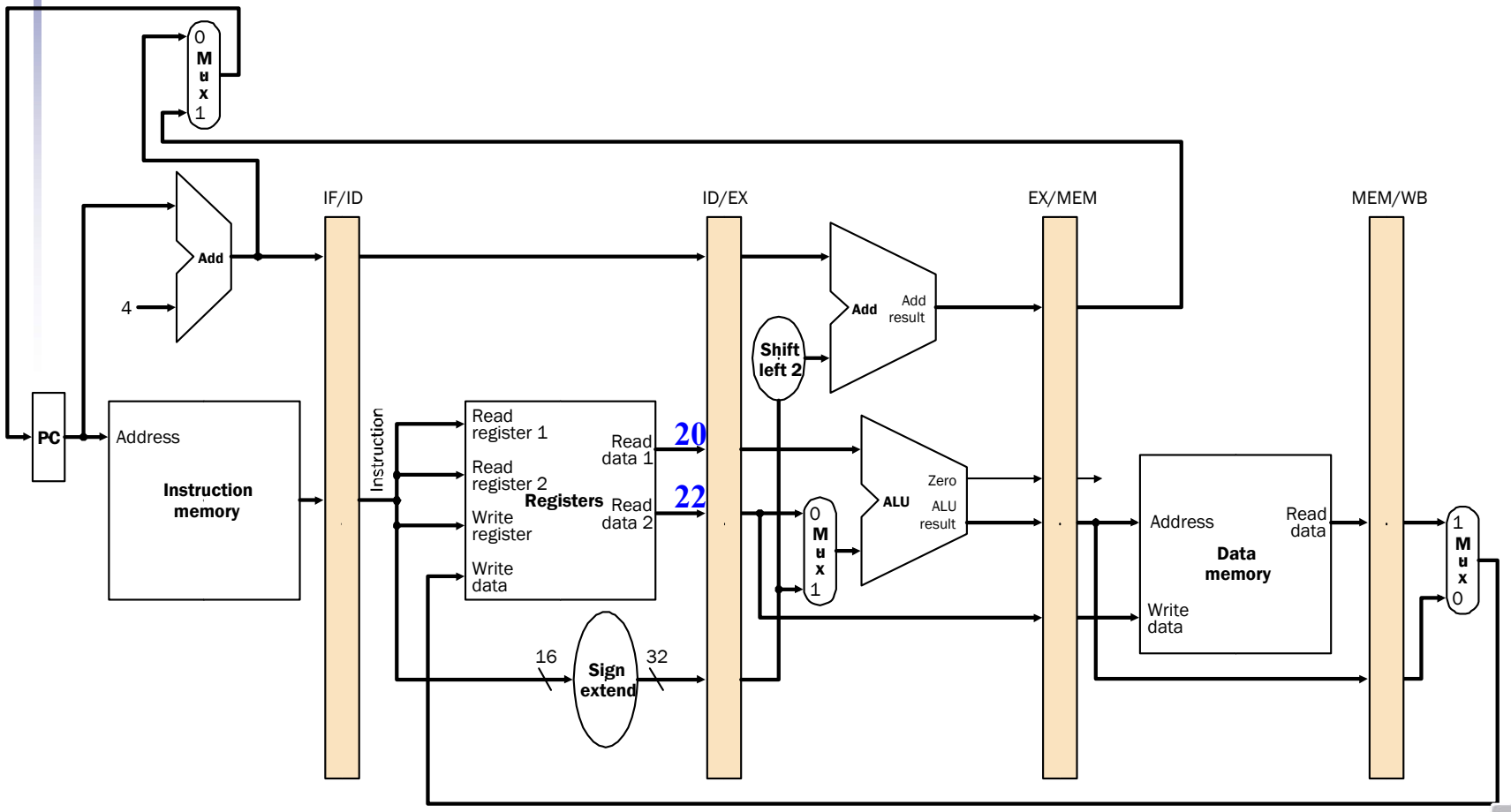
Hazards

- Suppose initially, register $i holds the number 2i
- What happens when we see the following dynamic instruction sequence:
  - **add $3, $10, $11**
    - this should add 20 + 22, putting result 42 into $3
  - **lw $8, 50($3)**
    - this should load memory location 92 (42+50) into $8
  - **sub $11, $8, $7**
    - this should subtract 14 from that just-loaded val



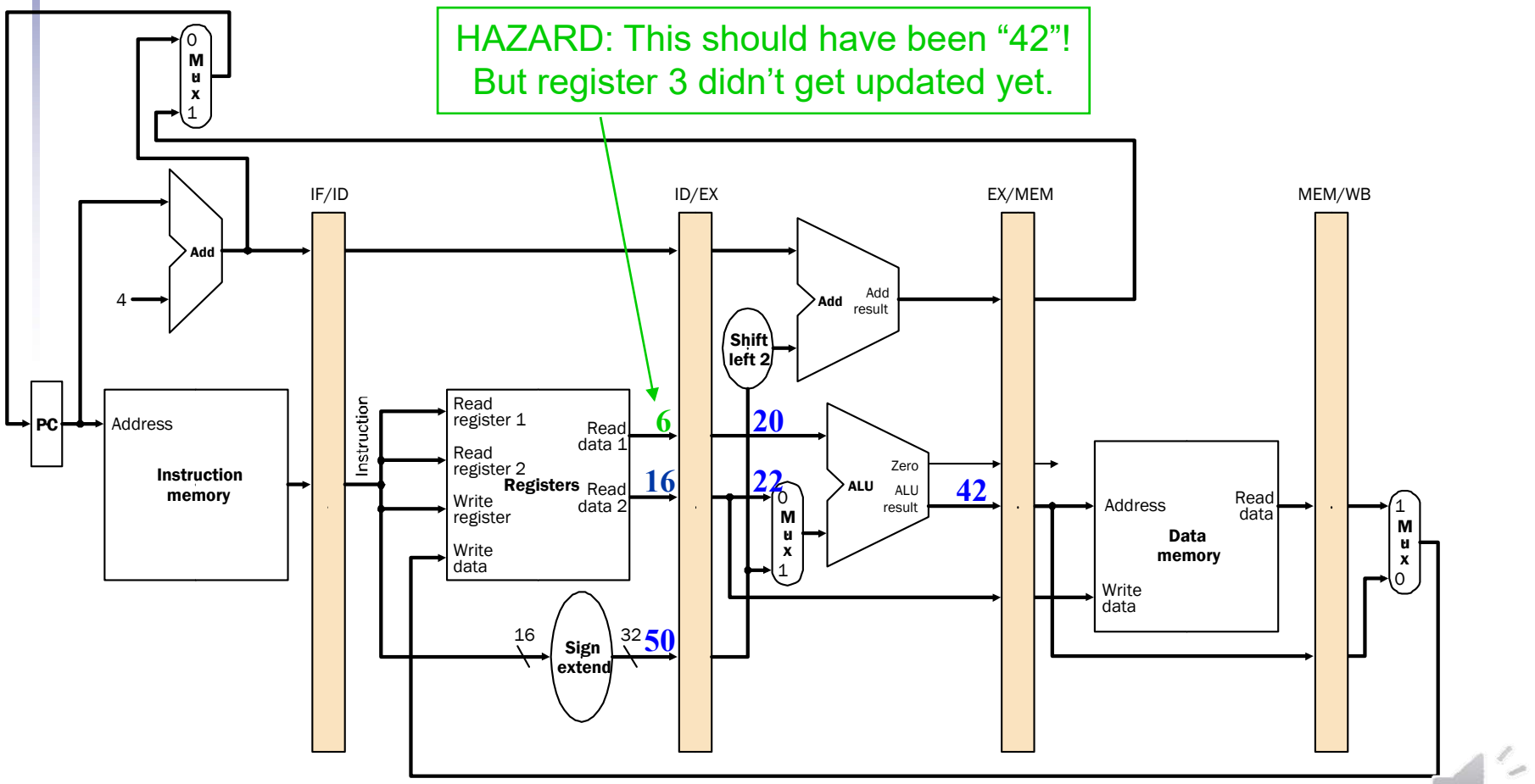
The Pipeline in Execution

lw $8, 50($3) **add $3, $10, $11** Execute/ Address Calculation Memory Access Write Back

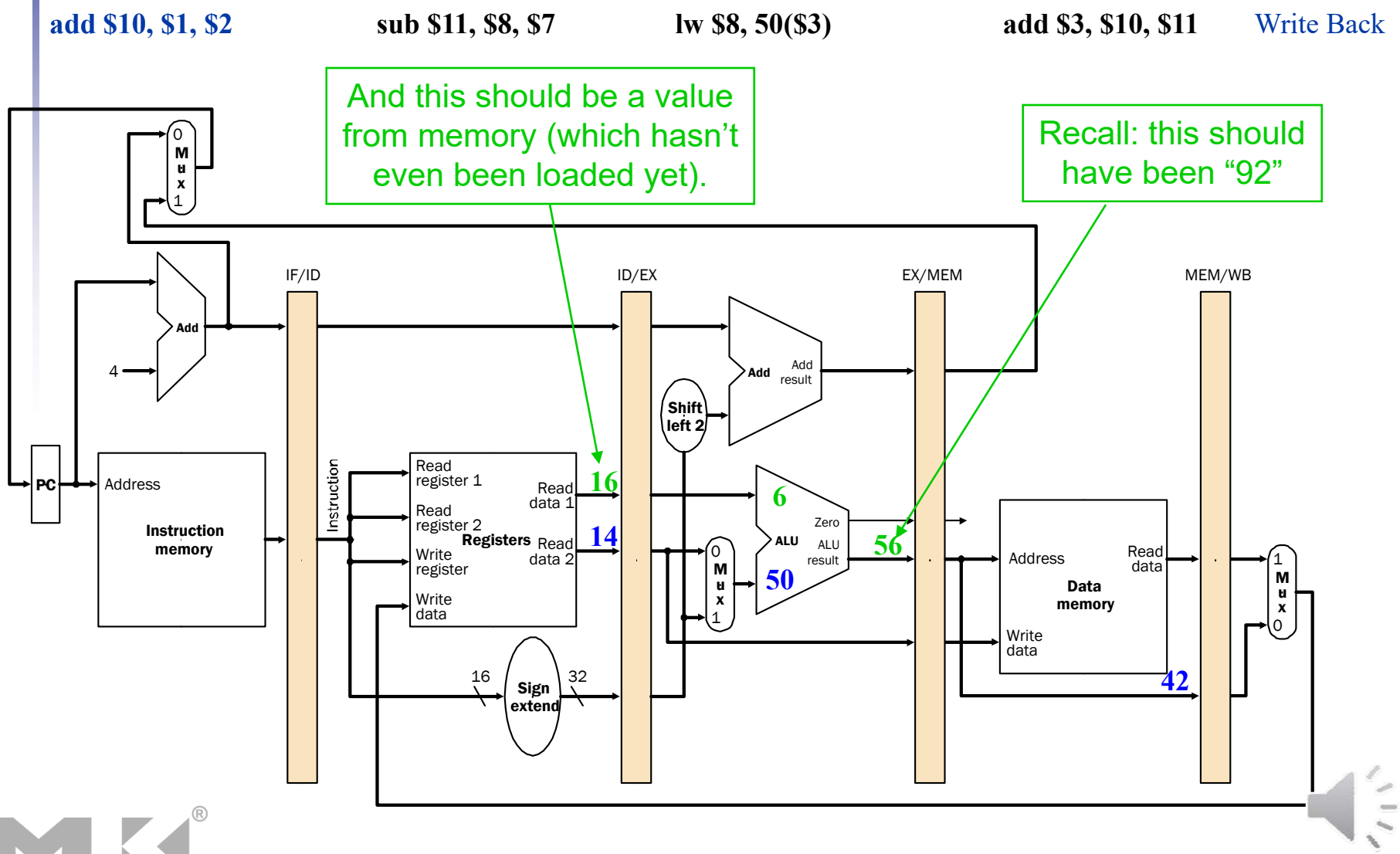


The Pipeline in Execution

sub $11, $8, $7 **lw $8, 50($3)** **add $3, $10, $11** Memory Access Write Back



The Pipeline in Execution



Hazards

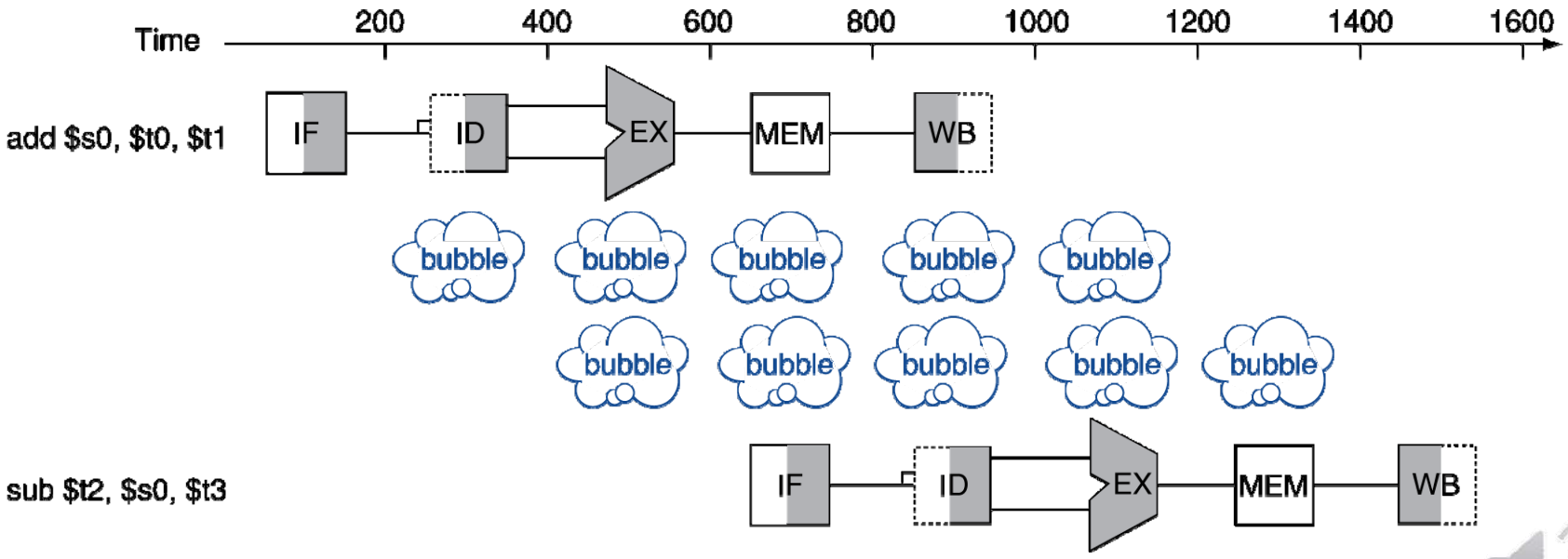
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

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

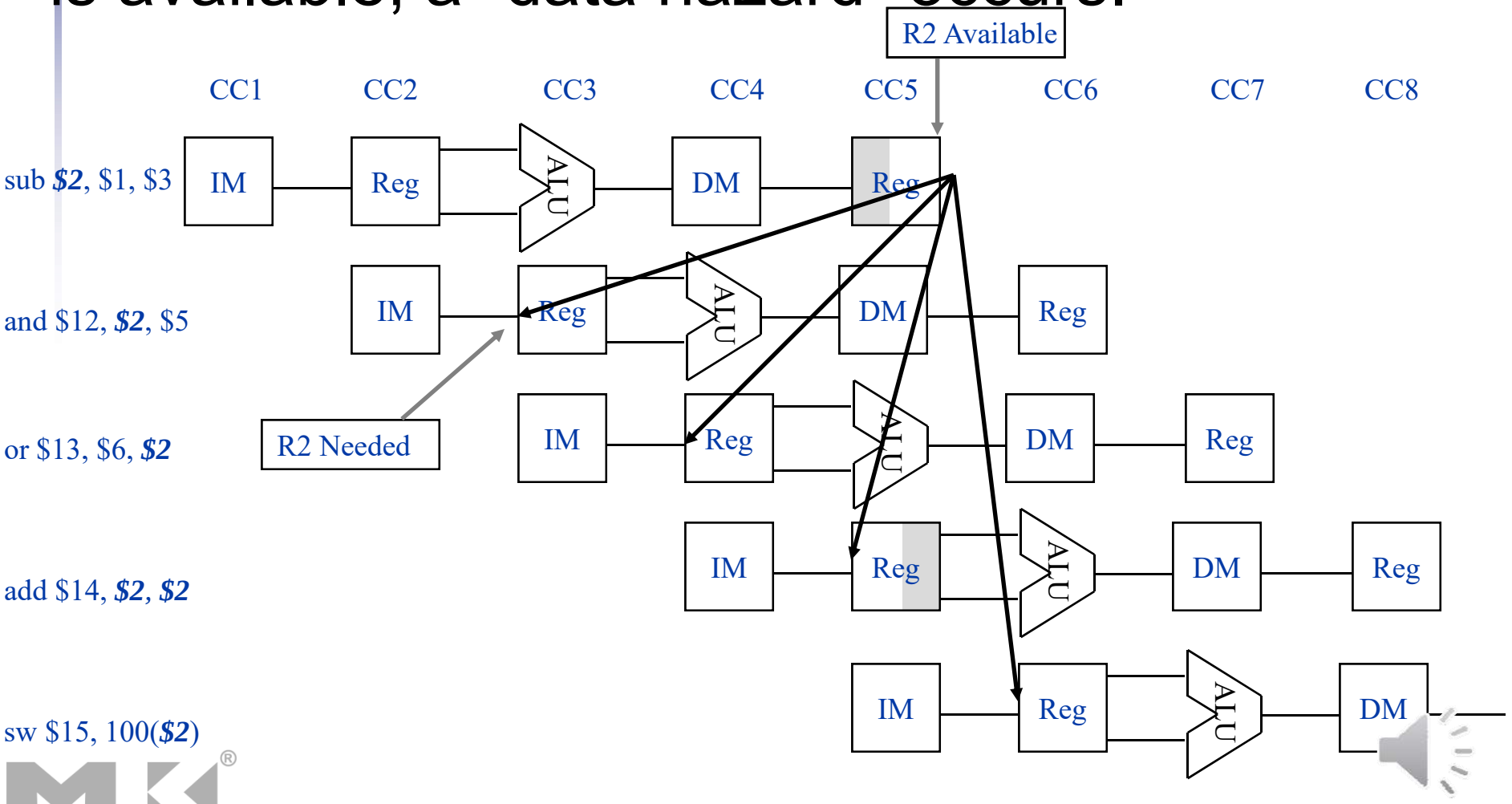
Data Hazards

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



Data Hazards

- When a result is needed in the pipeline before it is available, a “data hazard” occurs.



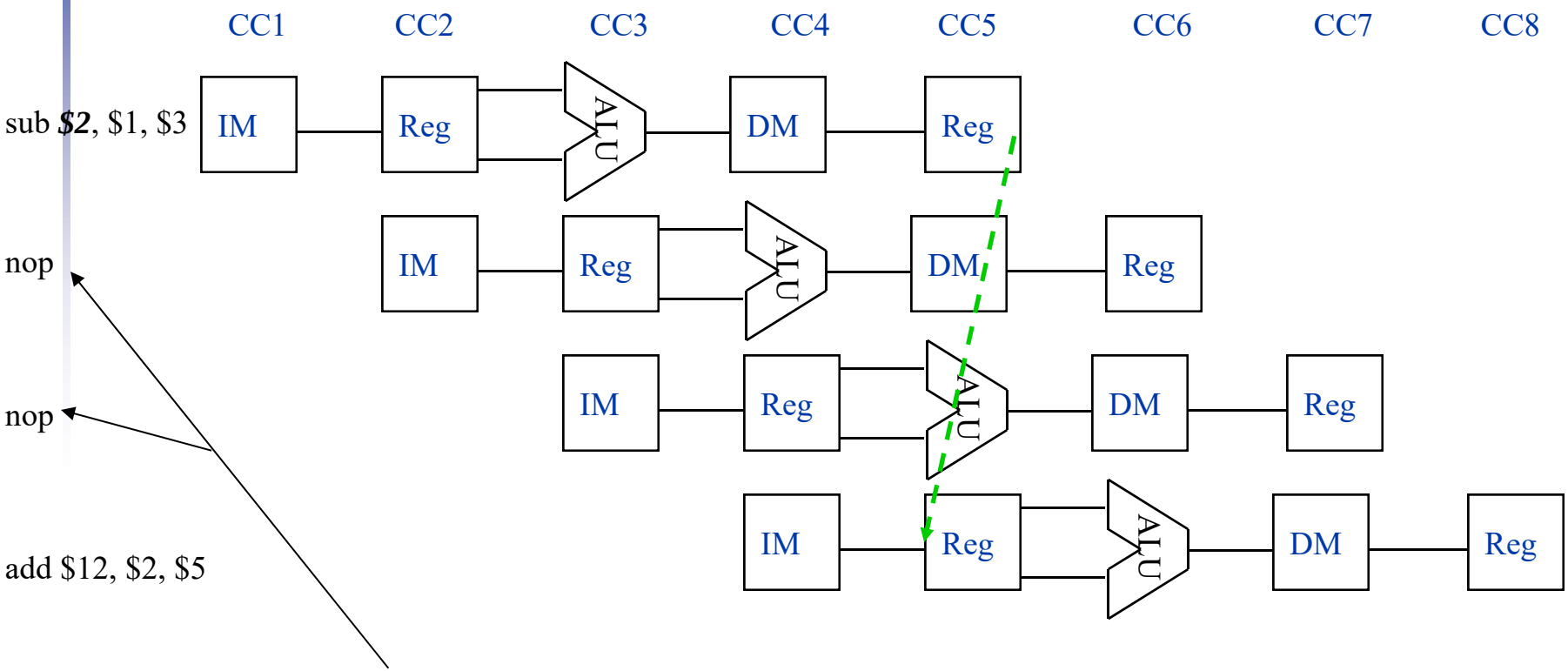
Chapter 4

The Processor

Dealing with Data Hazards

- In Software
  - insert independent instructions (or no-ops)
- In Hardware
  - insert bubbles (i.e. stall the pipeline)
  - data forwarding

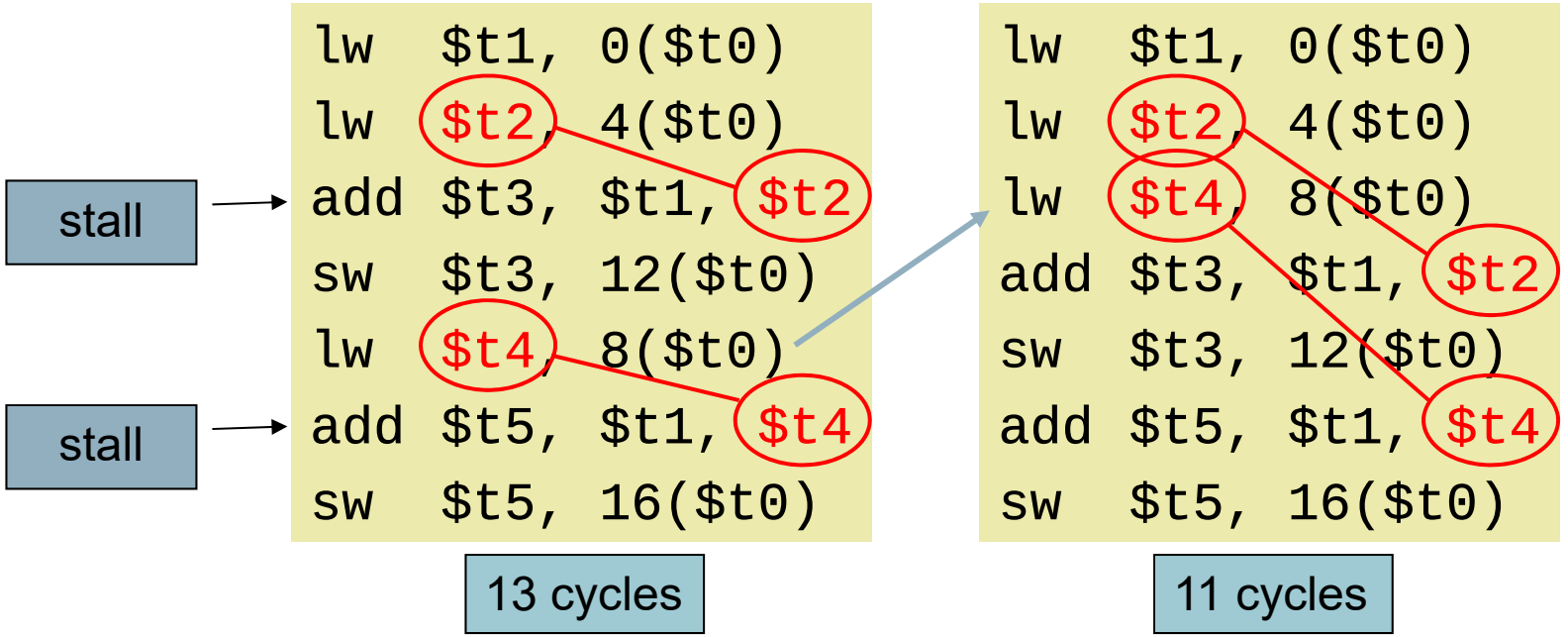
Dealing with Data Hazards in Software



Insert enough no-ops (or other instructions that don't use register 2) so that data hazard doesn't occur,

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;`



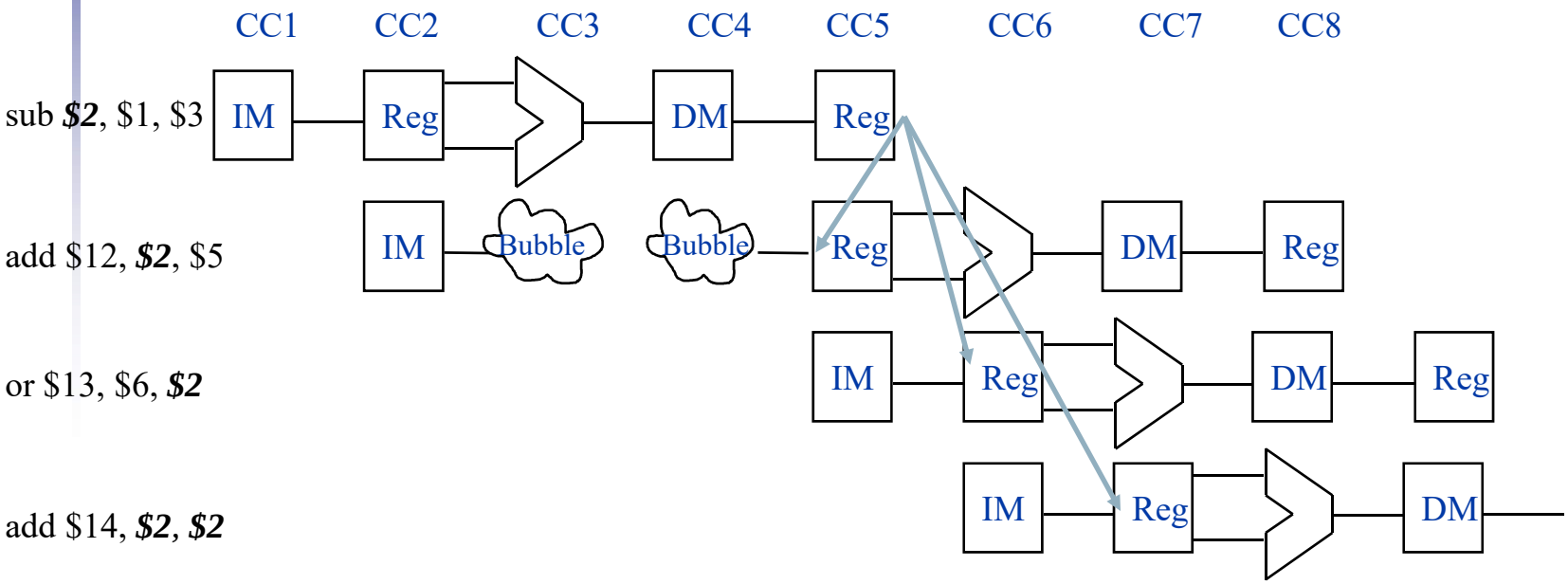
Where are No-ops needed?

- sub $2, $1,$3
- and $4, $2,$5
- or $8, $2,$6
- add $9, $4,$2
- slt $1, $6,$7

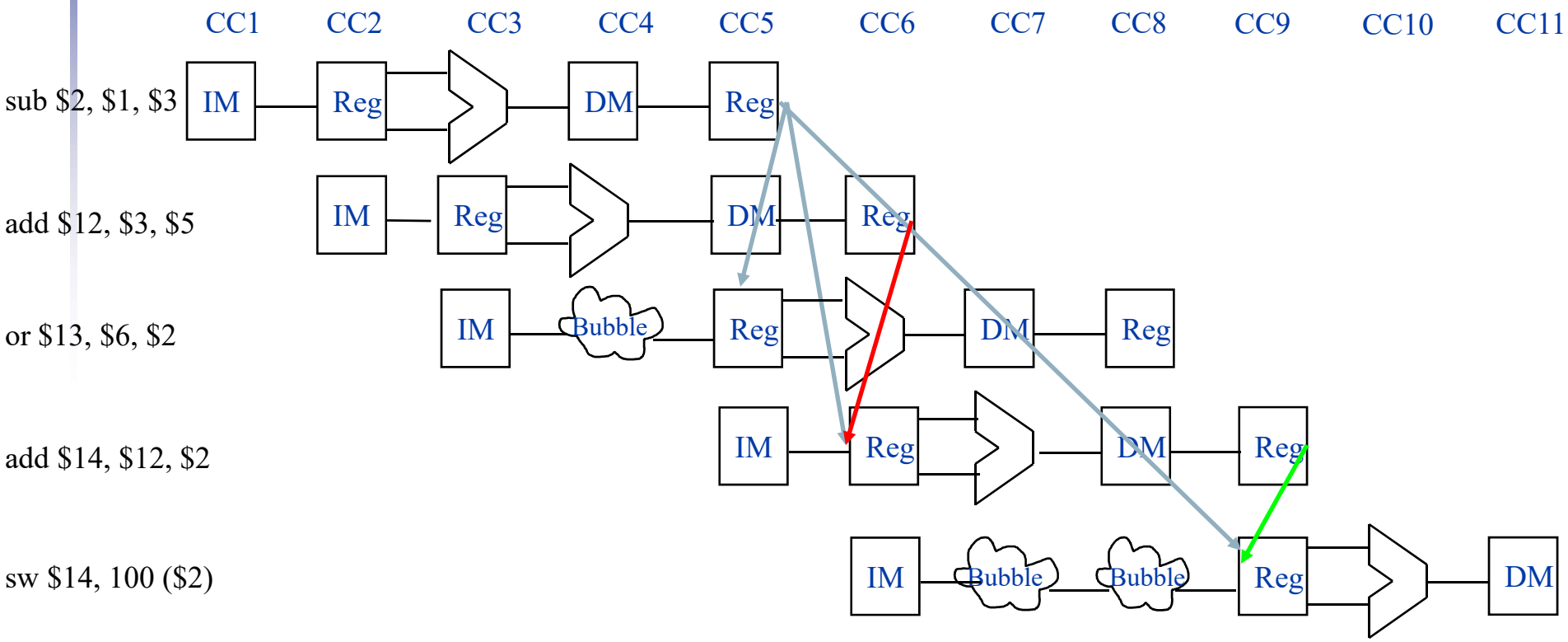
- Are no-ops really necessary?

Handling Data Hazards in Hardware

- Stall the pipeline



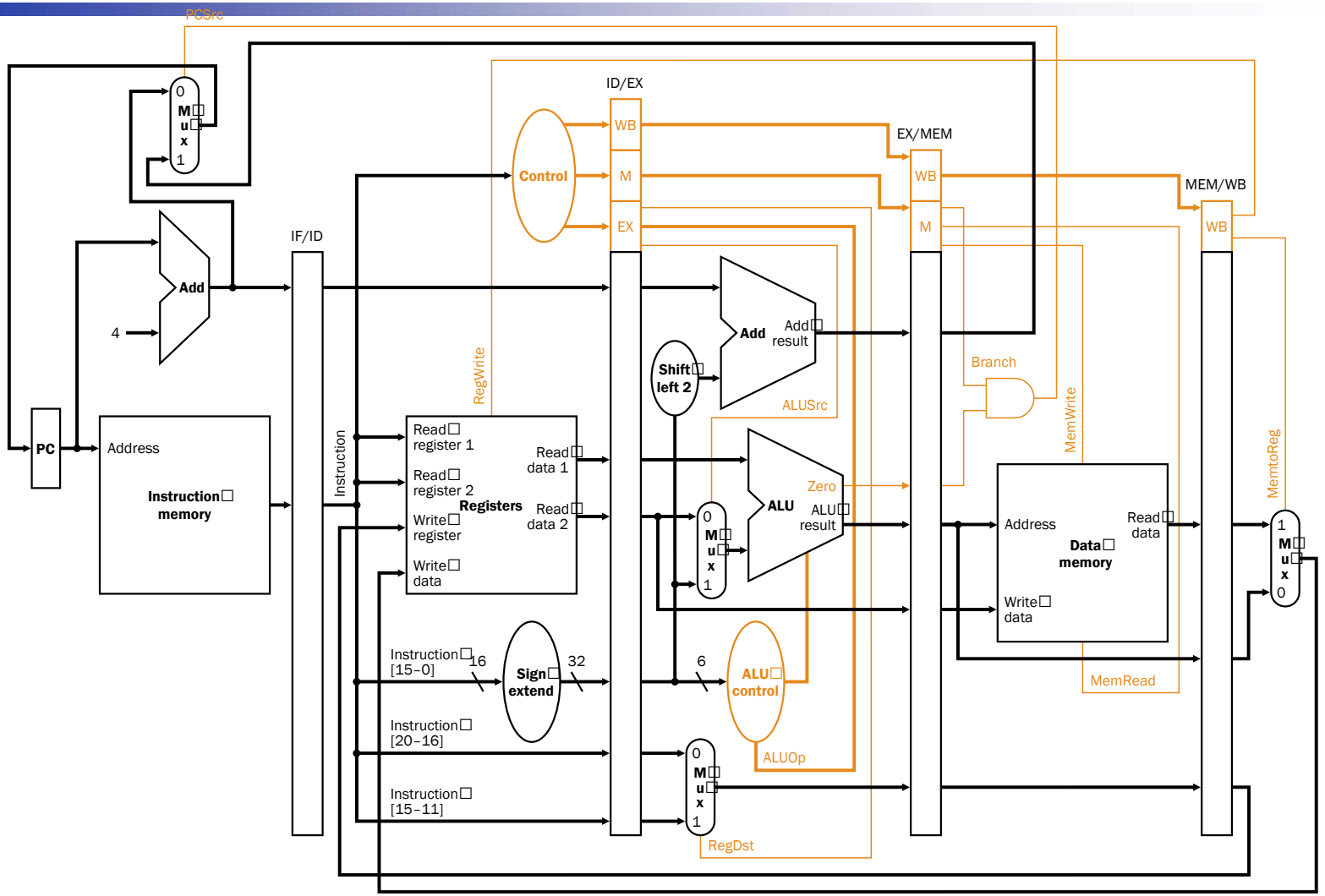
Handling Data Hazards in Hardware



Pipeline Stalls

- To insure proper pipeline execution in light of register dependences, we must:
  - Detect the hazard
  - Stall the pipeline
    - prevent the IF and ID stages from making progress
      - the ID stage because we can't go on until the dependent instruction completes correctly
      - the IF stage because we do not want to lose any instructions.

The Pipeline



What comparisons tell us when to stall?

Stalling the Pipeline

- Prevent the IF and ID stages from proceeding
  - don't write the PC (PCWrite = 0)
  - don't rewrite IF/ID register (IF/IDWrite = 0)
- Insert "nops"
  - set all control signals propagating to EX/MEM/WB to zero