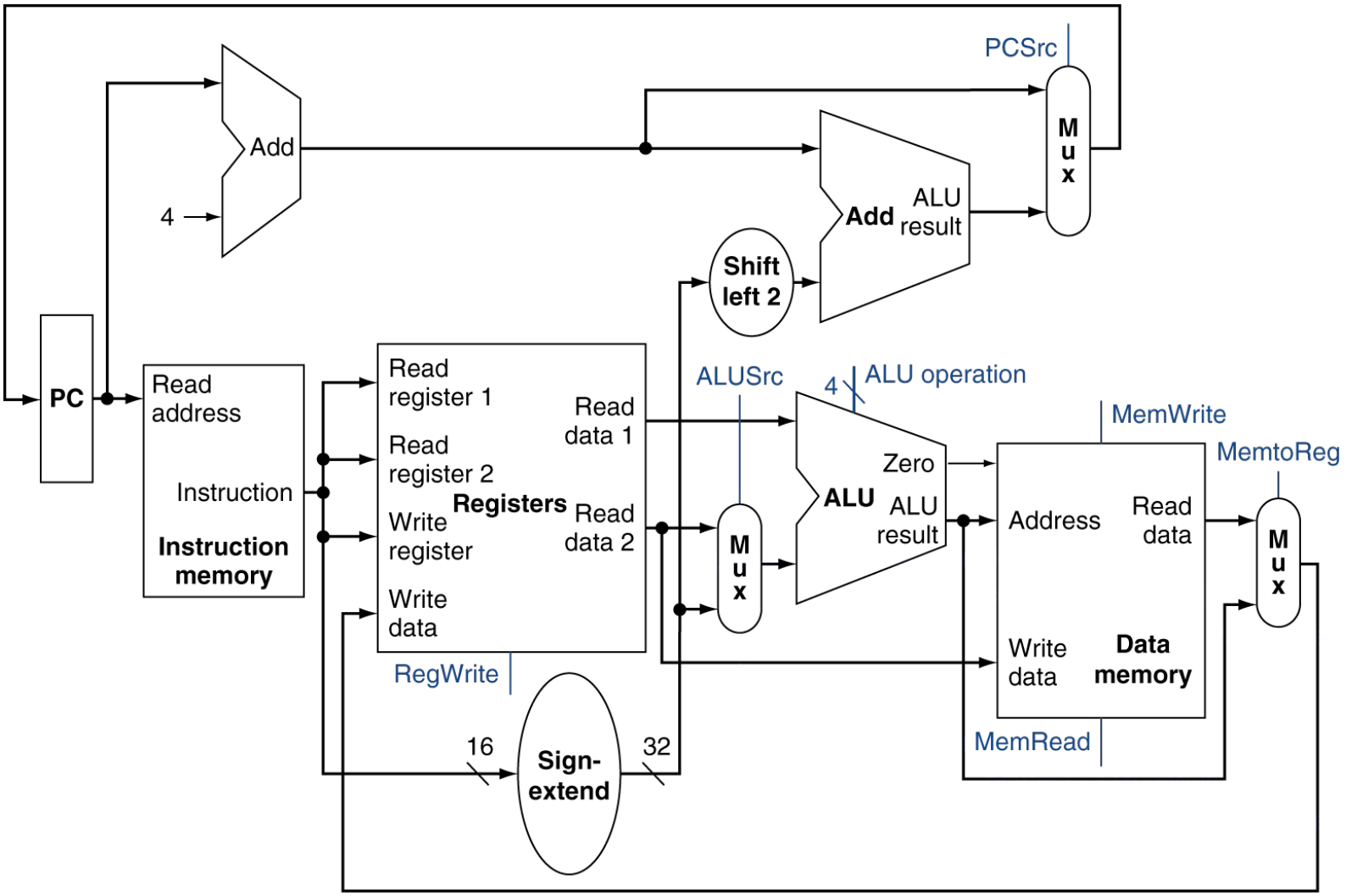
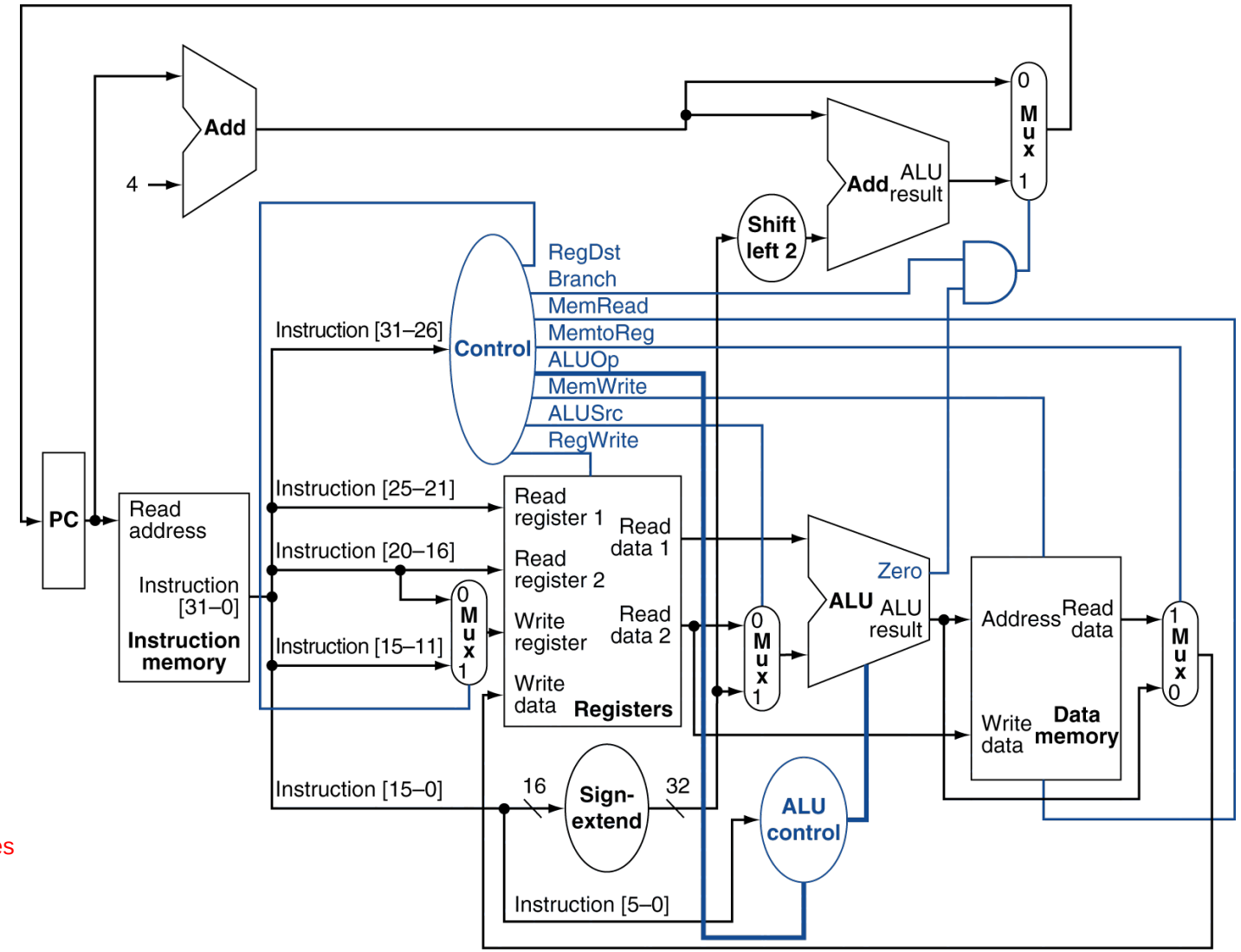
Pipelining

Full Datapath



Datapath With Control



5 fases

Performance Issues

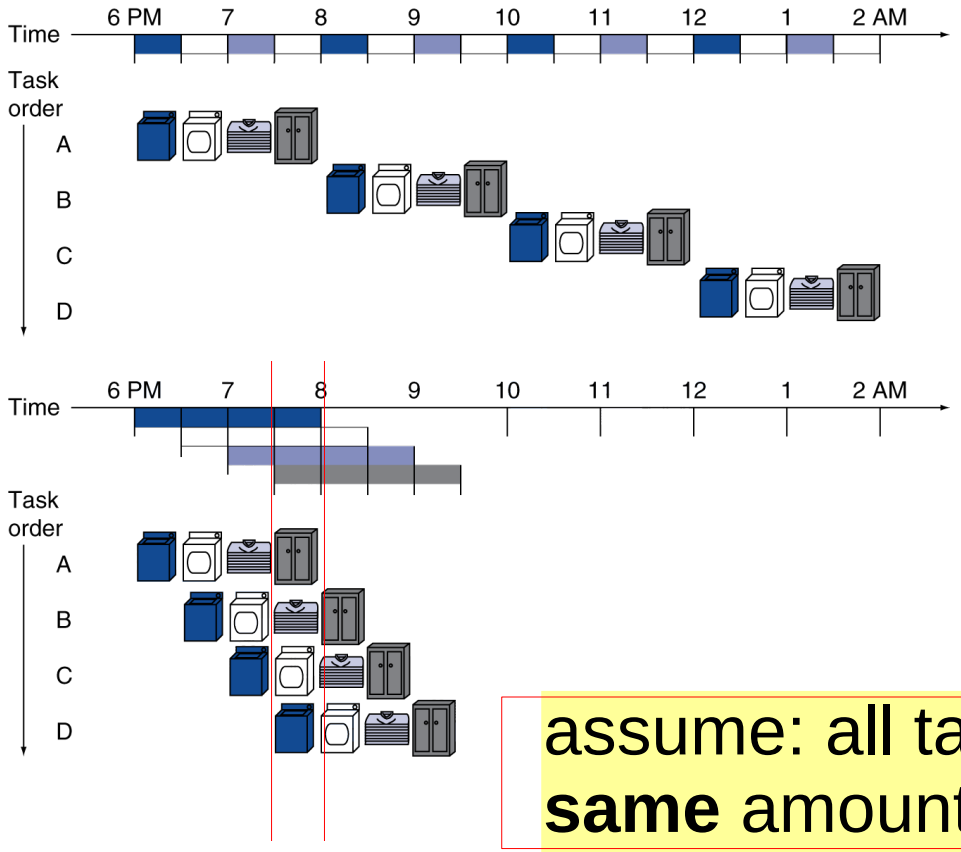
- **Longest delay** determines **clock period**
  - **Critical path**: **load** instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not desirable to vary period for different instructions

  Violates the **regularity** design principle
- We will improve performance by **pipelining**, exploiting **instruction-level parallelism (ILP)**

Pipelining Analogy

Pipelined laundry: **overlapping** execution

- **Parallelism** improves performance



- 4 loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup (for very large n) = Kn/(n + (K-1)) ≈ K

K = number of stages

assume: all tasks take **same** amount of time

MIPS Pipeline

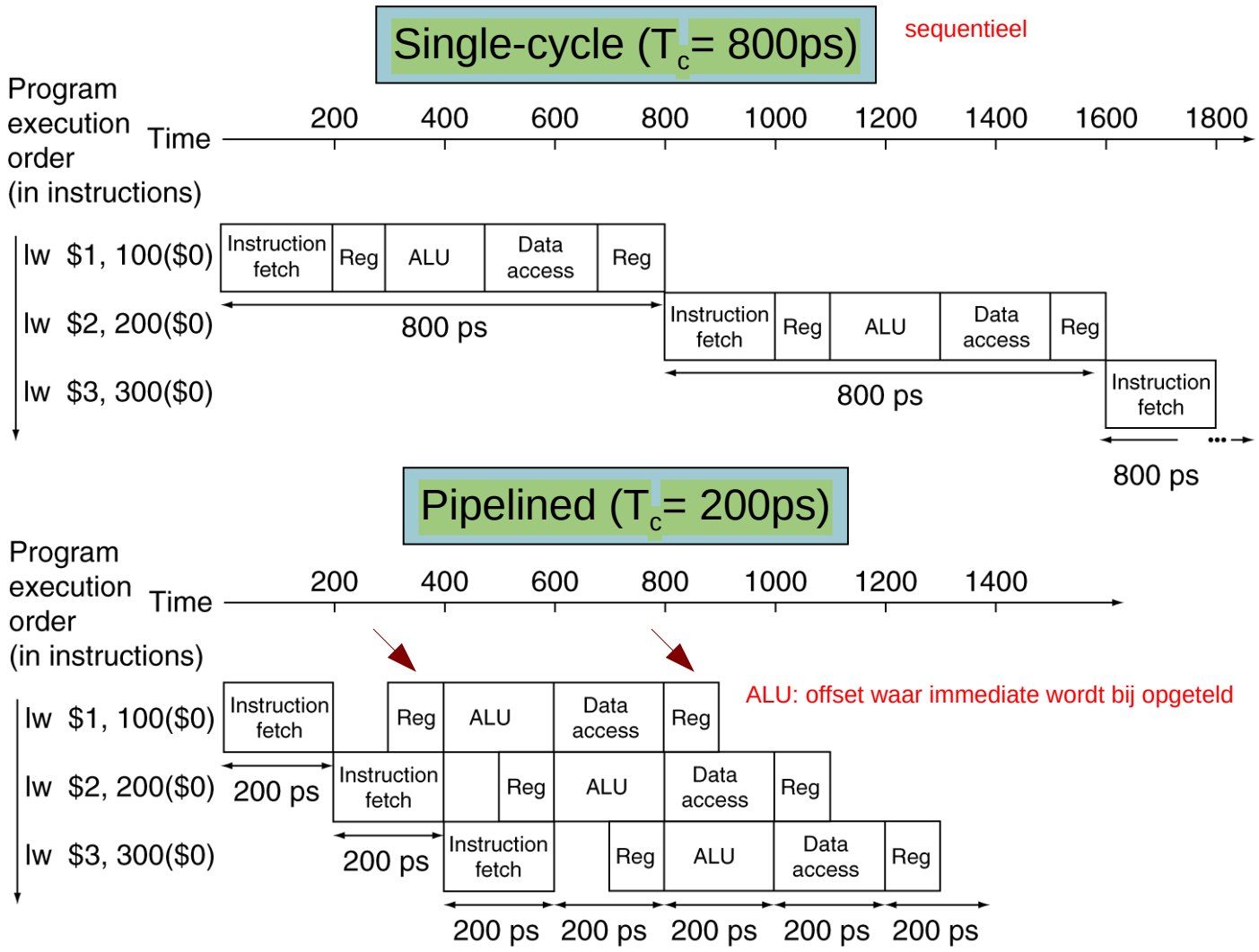
- Five **stages**, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode and register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

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!

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - **All** instructions are **same length** (32-bits)
    - Easier to fetch and decode in **one cycle**
    - cf. 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

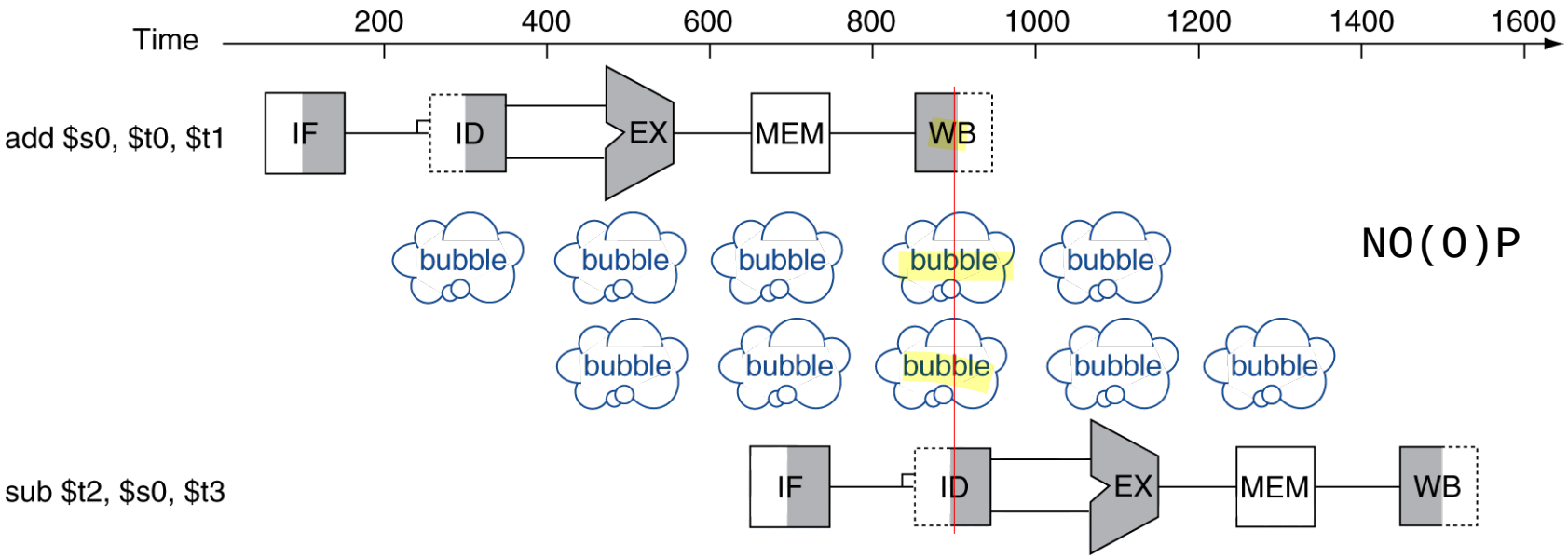
- Situations that **prevent** starting the next instruction in the next cycle. Need to wait!
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for a previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on a previous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

Structure Hazards

- Conflict for **use of a resource**
- In MIPS pipeline with a **single memory**
  - Load/store requires data access
  - Instruction fetch requires memory acces too → 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**

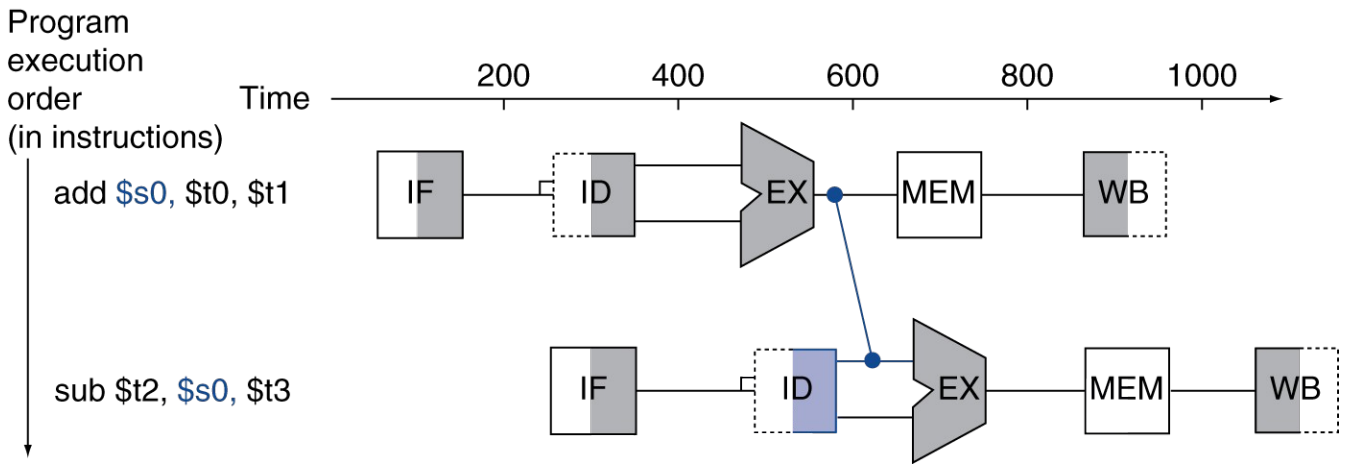
Data Hazards

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



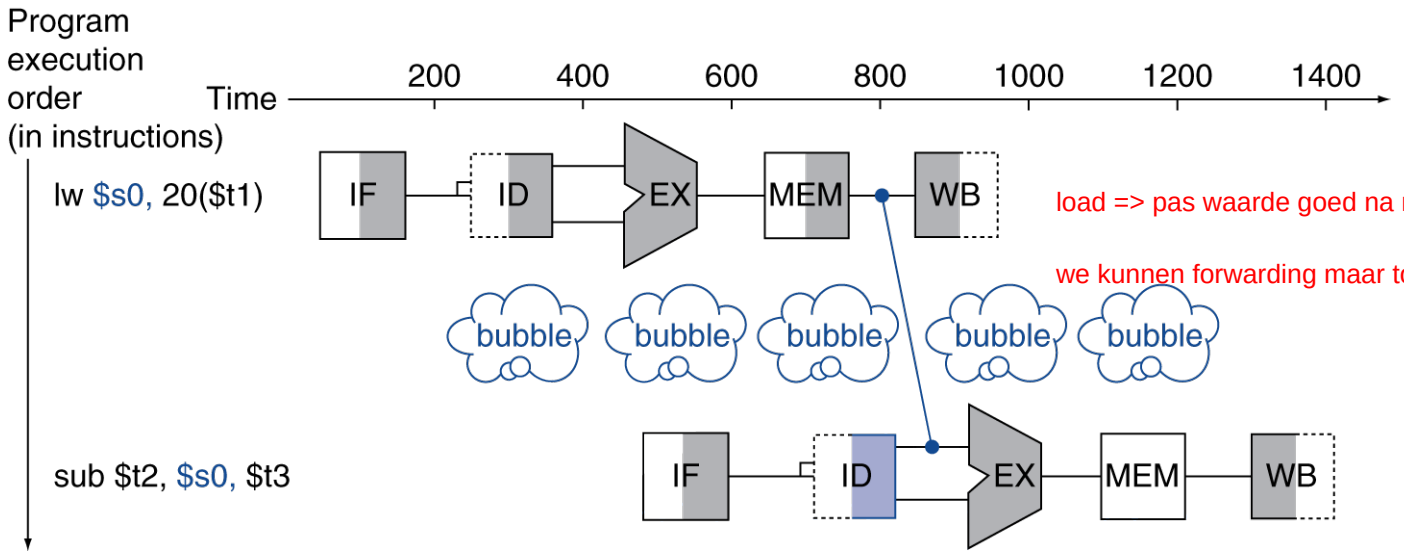
Forwarding (aka Bypassing)

- Use result when it is computed
  - **Do not wait** for it to be **stored** in a register
  - Requires **extra connections** in the datapath (**extra hardware** to **detect** and **fix** hazard)



Load-Use Data Hazard

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



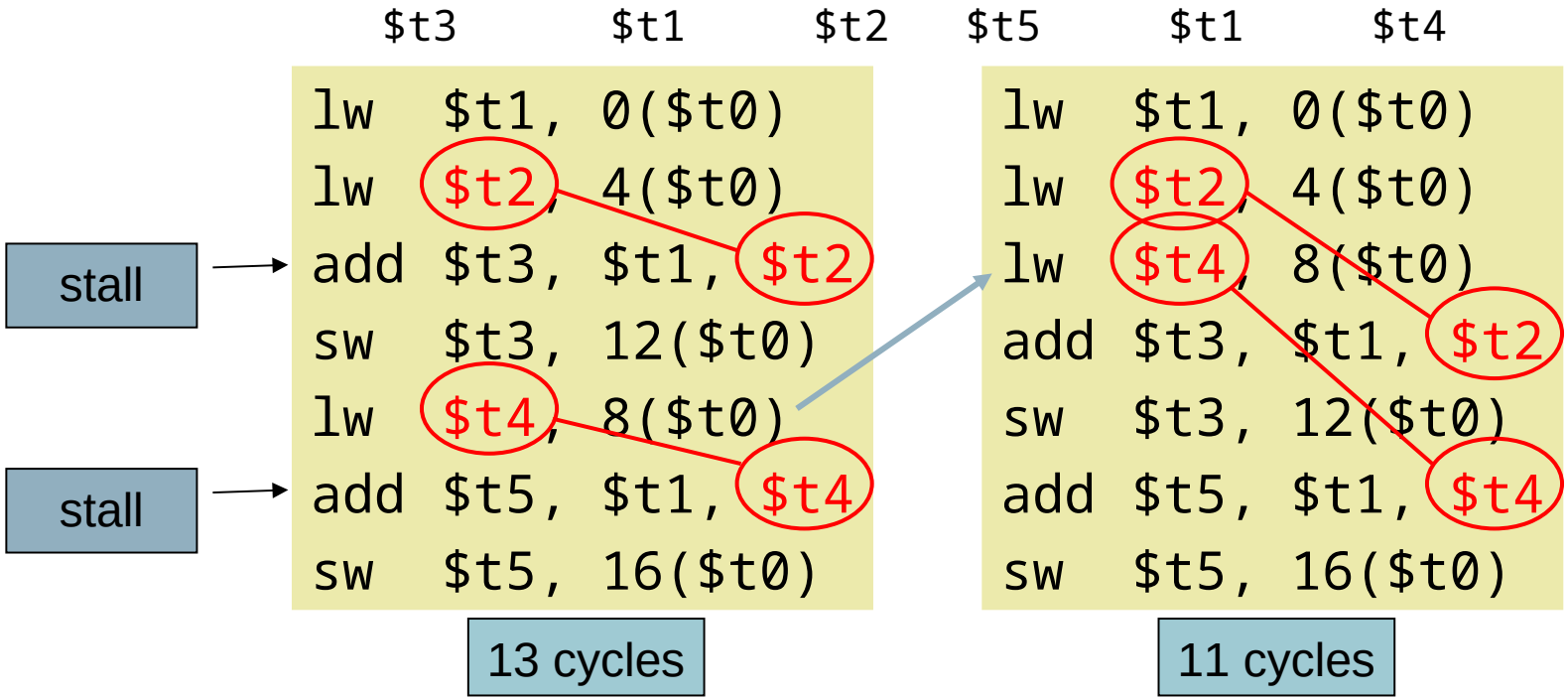
load => pas waarde goed na memory write

we kunnen forwarding maar toch nog 1 bubble

load gevolgd door add dus 1x 200ps delay met forwarding
anders verkeerde waarde

Code Scheduling to **Avoid** Stalls

- **Re-order** code to avoid use of load result in the next instruction
- code for `A = B + E; C = B + F;`



n + (K-1)
K stages, n instructies => 7 + (5-1) = 11

ipv tijd te waste, plaats andere instructie tussen de tijd dat we moeten wachten zodat er minder tijd/cycles zijn

Control Hazards

- **Branch** determines **flow of control**
  - **Fetching next instruction**

    depends on test 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

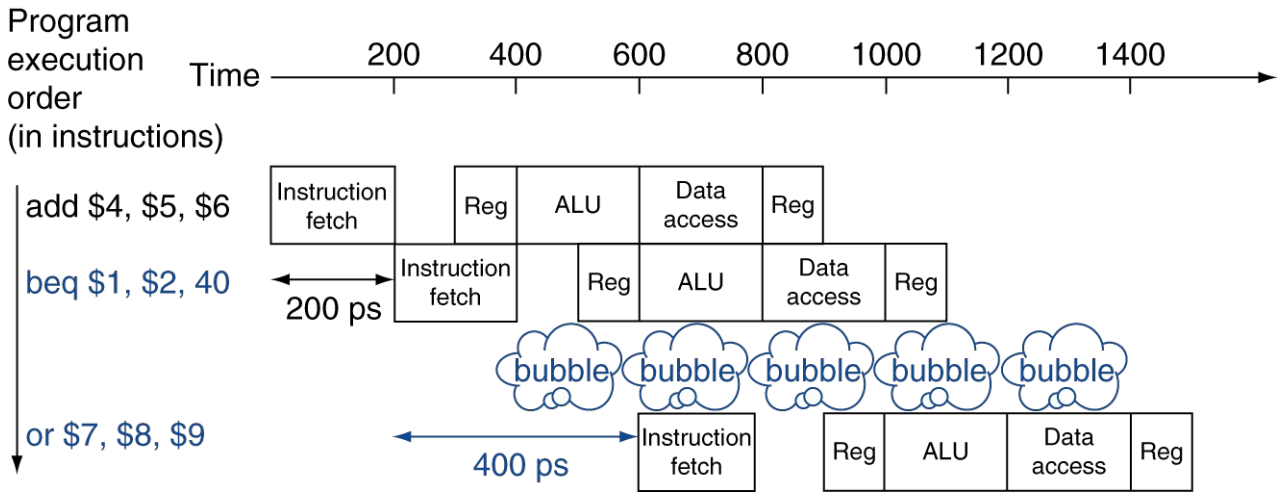
  →

  add **hardware** to do it **in ID stage**

  (ID = Instruction Decode and Register Read)

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

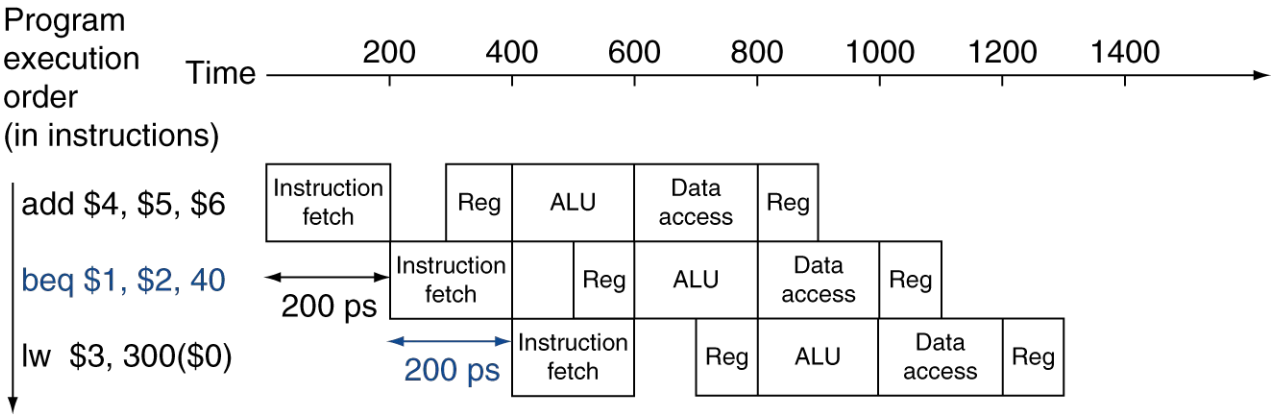


Branch Prediction

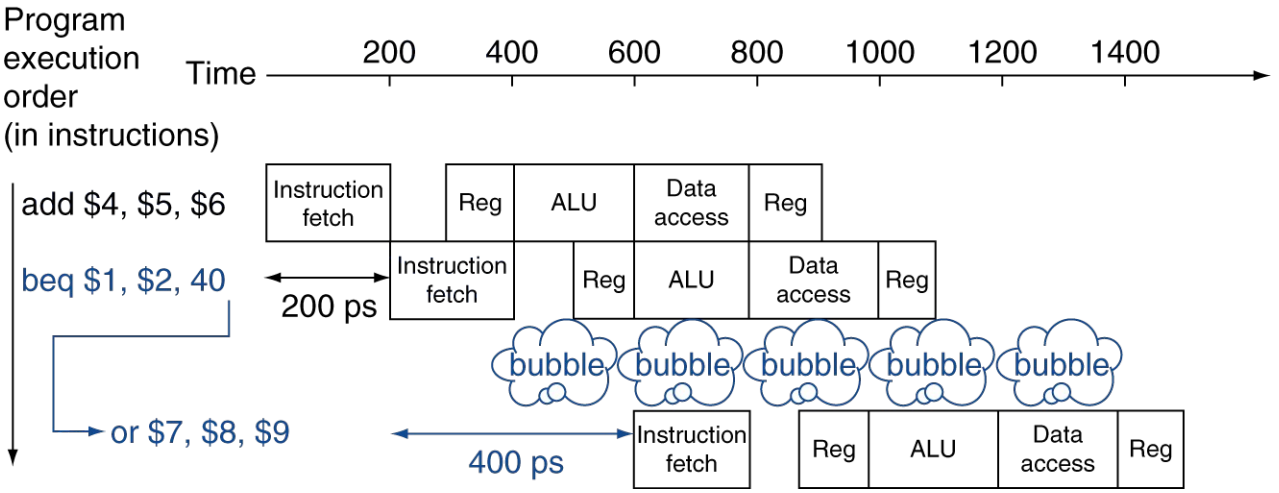
- Longer pipelines **can't** readily **determine** branch outcome **early**
  - Stall penalty becomes unacceptable
- **Predict** outcome of branch
  - **Only** stall **if prediction is wrong**
- In MIPS pipeline
  - Can predict **branches not taken**
  - Fetch instruction after branch, with **no delay**

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



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
  - Extrapolate:

    assume future behavior will continue the trend
    - When wrong, stall while **re-fetching**, and **update history**

bubble

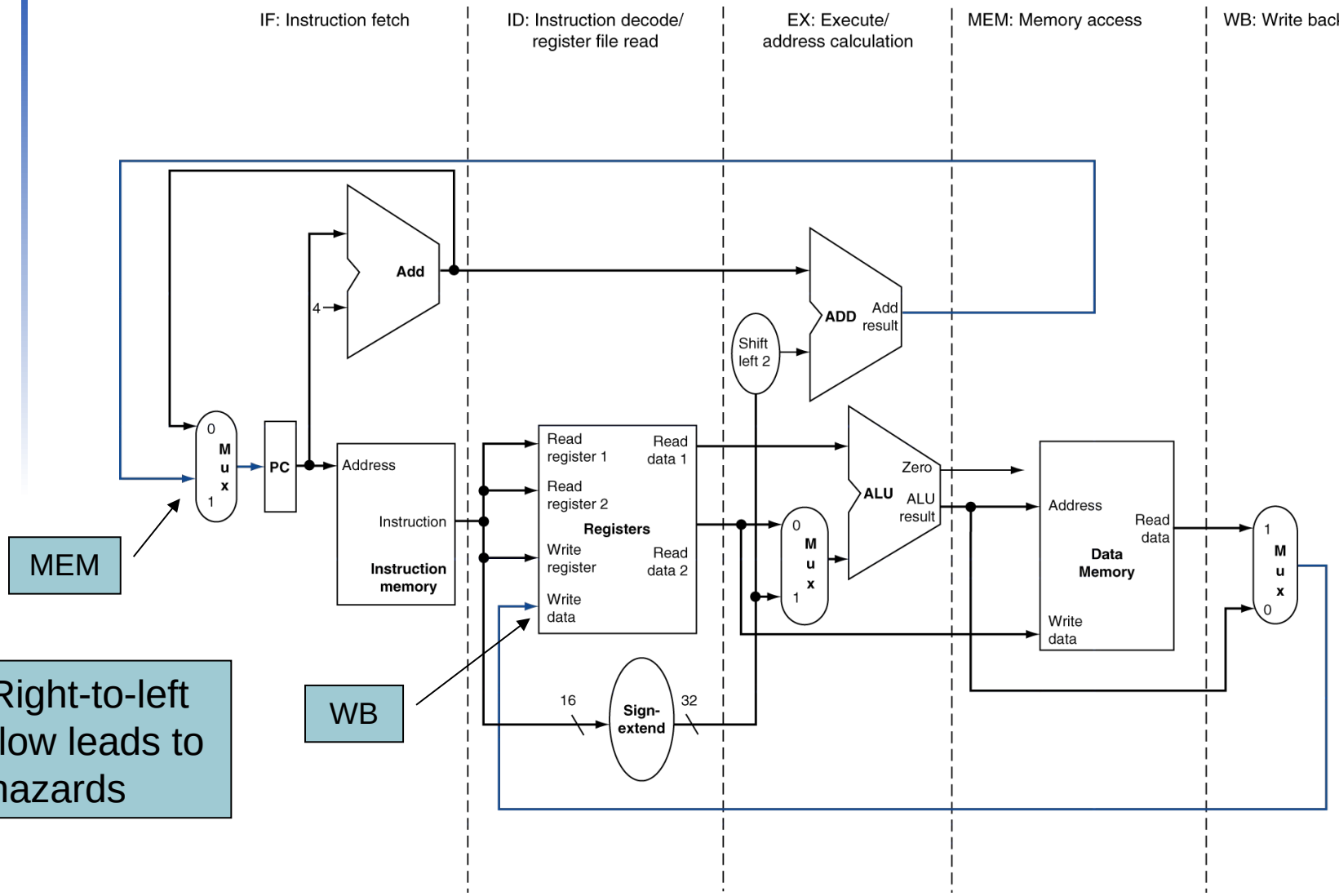
! BRANCH DELAY SLOT SLIDE NOT IN HERE !

Pipeline Summary

The BIG Picture

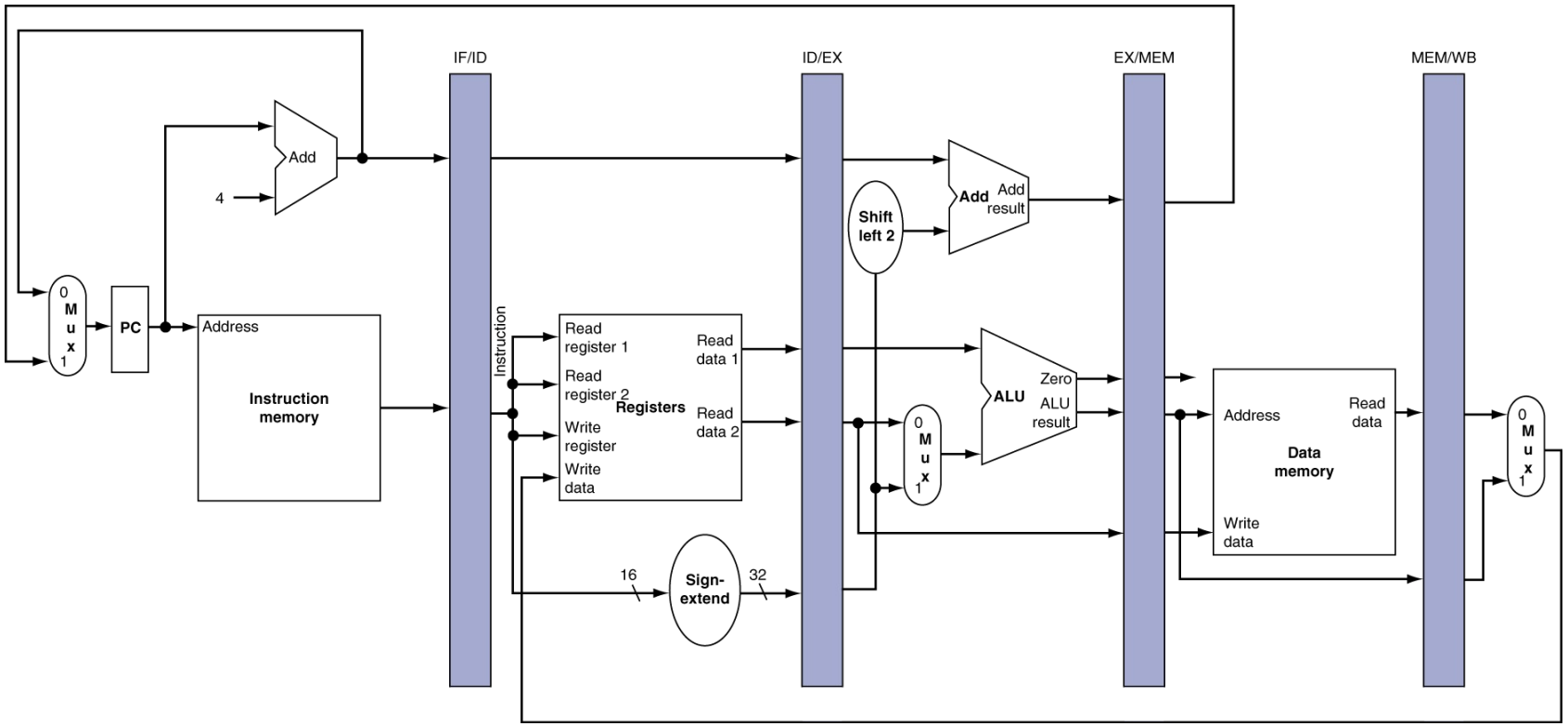
- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - structure, data, control
- Instruction set design (**ISA**) affects **complexity** of pipeline implementation

MIPS Pipelined Datapath



Pipeline registers

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

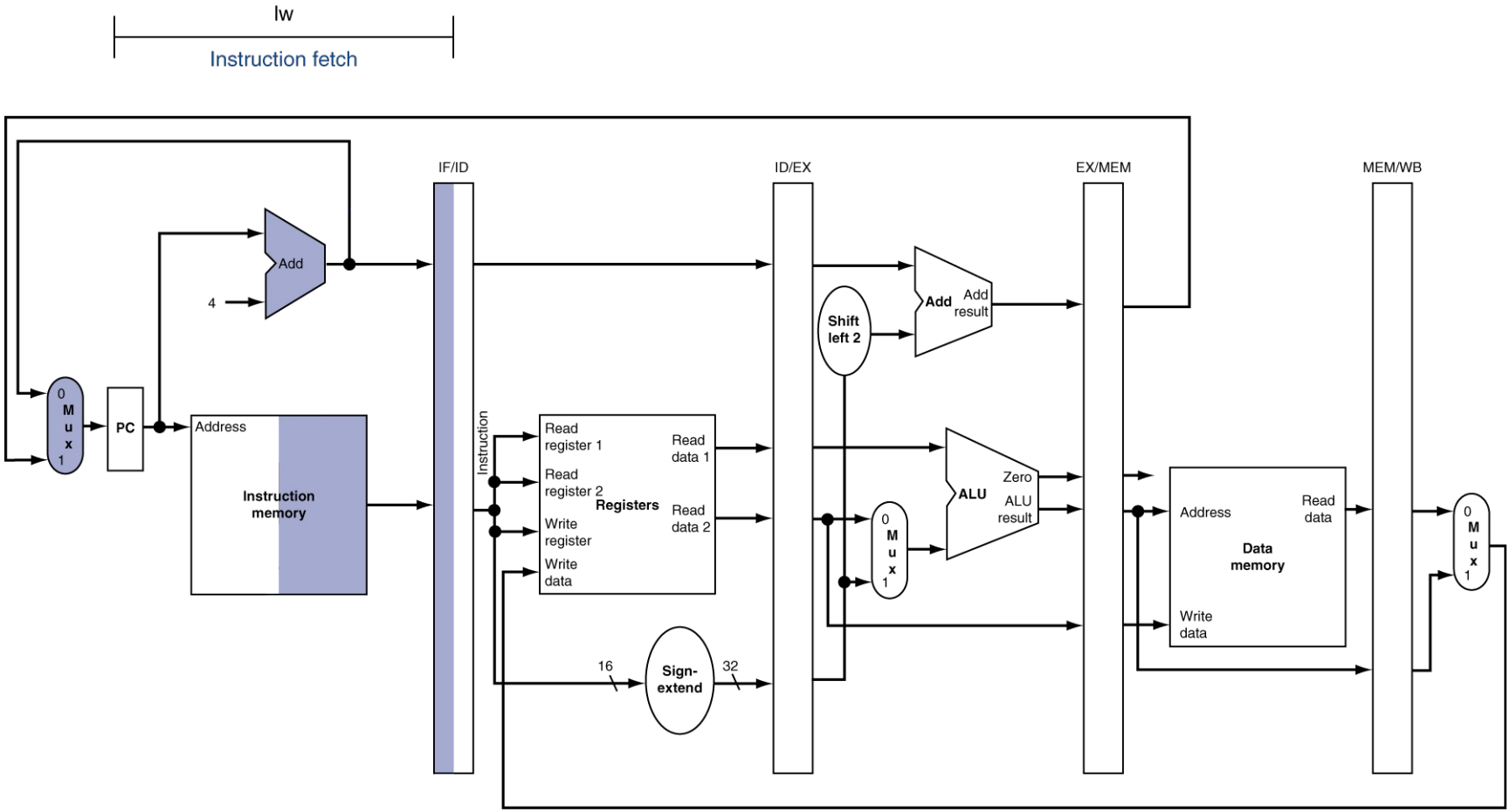


Pipeline Operation

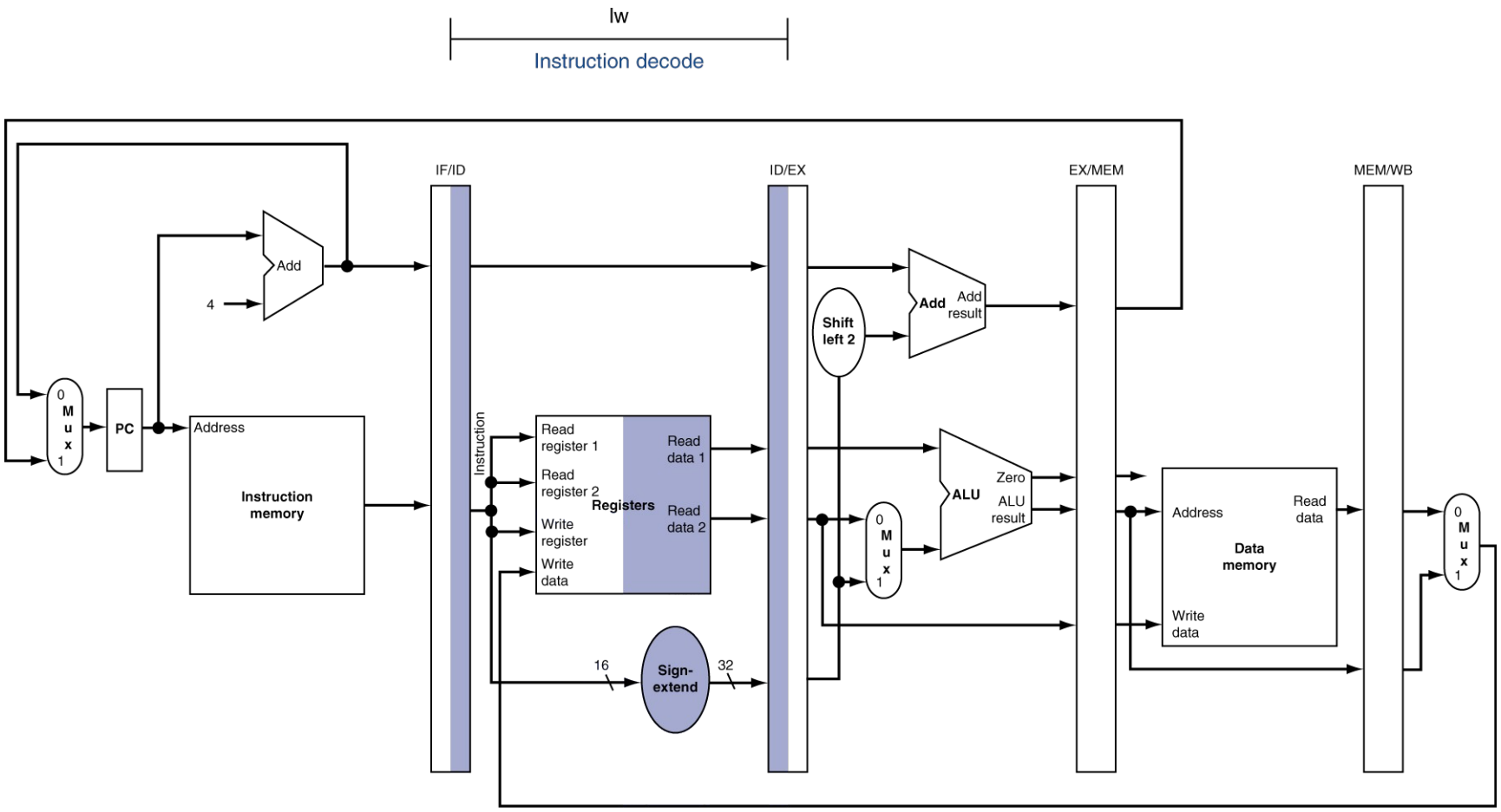
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “single-clock-cycle” pipeline diagram
    - shows pipeline usage in a single cycle: “snapshot”
    - highlight **resources** used
  - vs. “multi-clock-cycle” diagram
    - shows operation **over time**

- next: look at “single-clock-cycle” diagrams for load and store

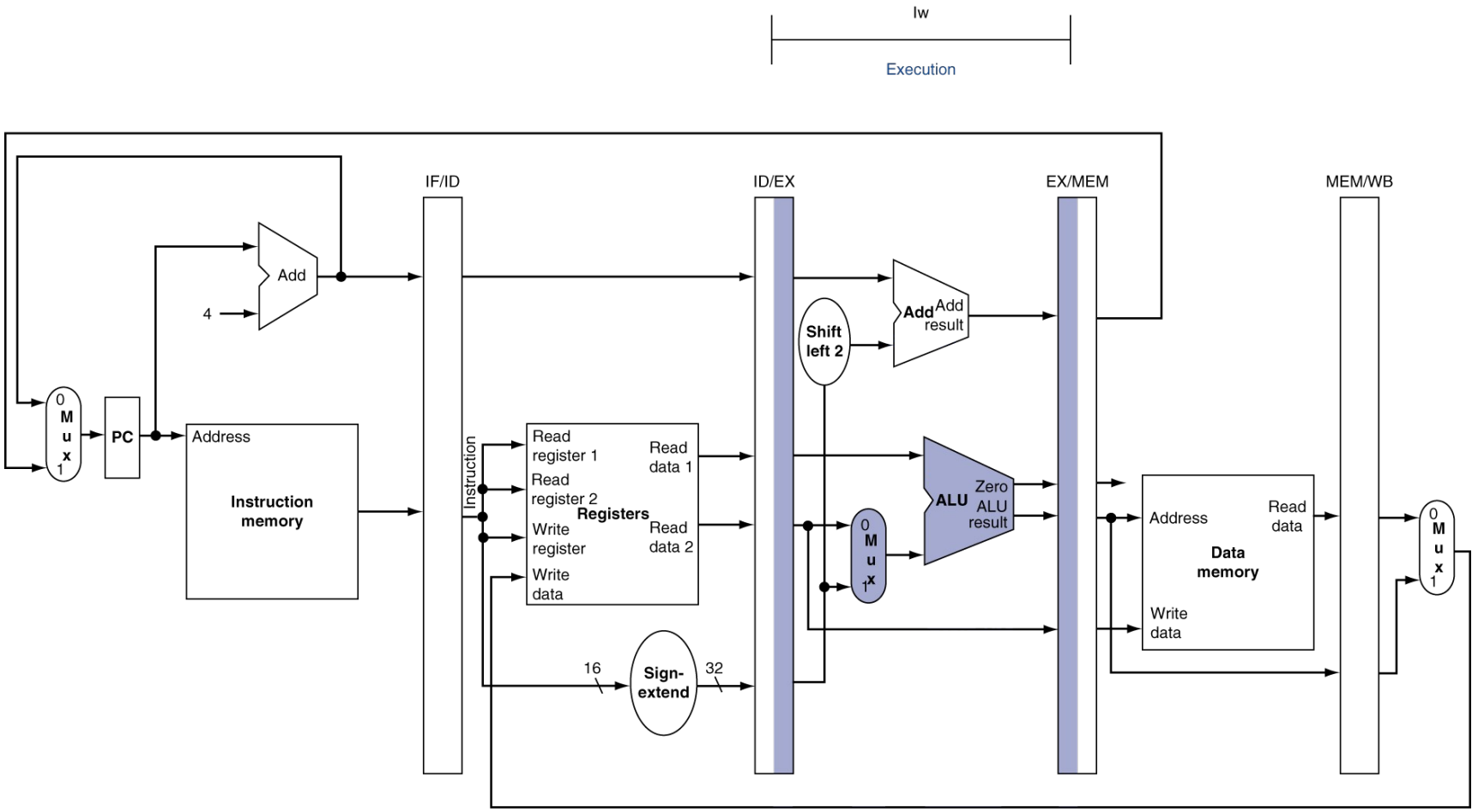
IF for Load, Store, …



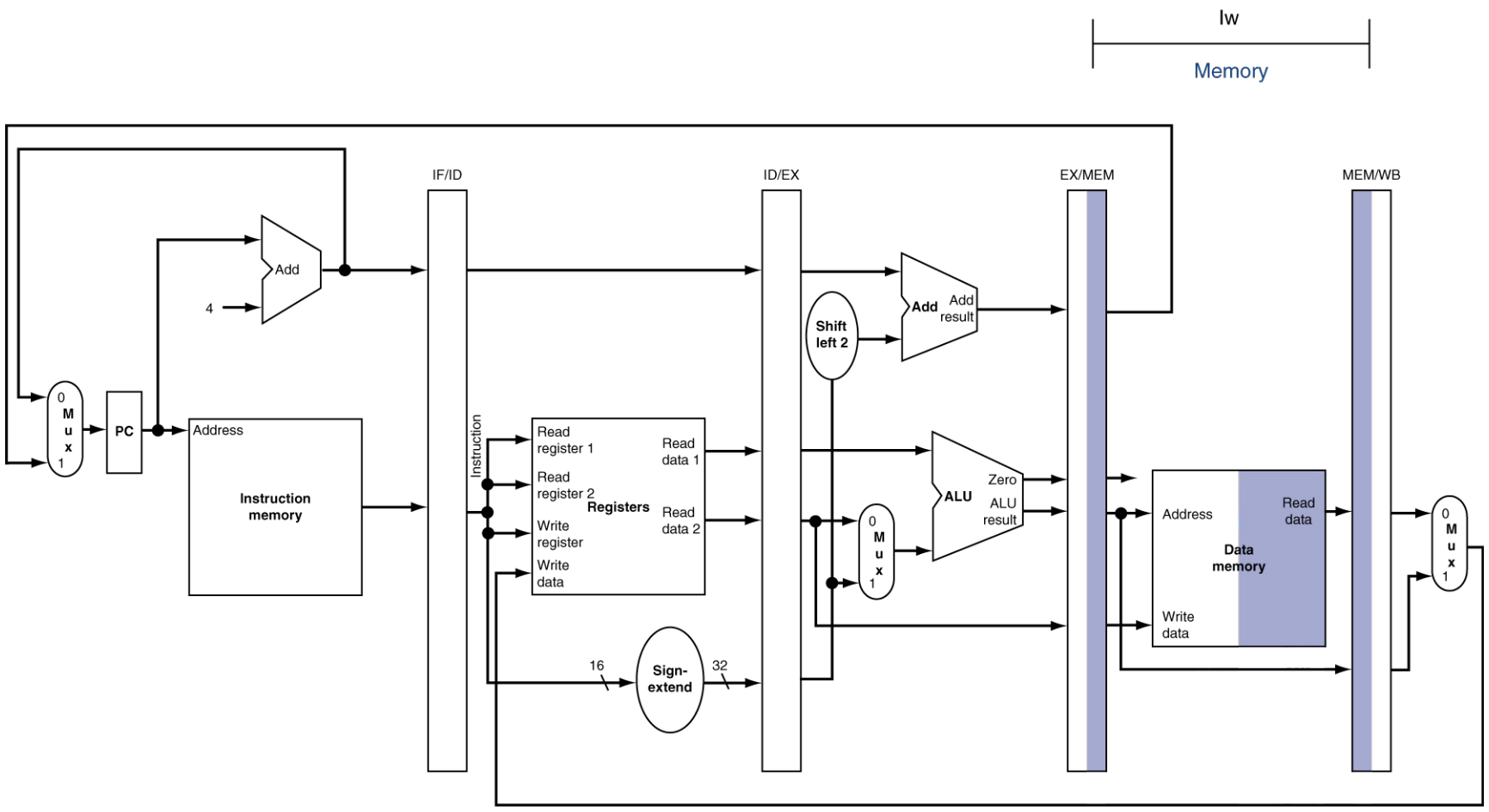
ID for Load, Store, ...



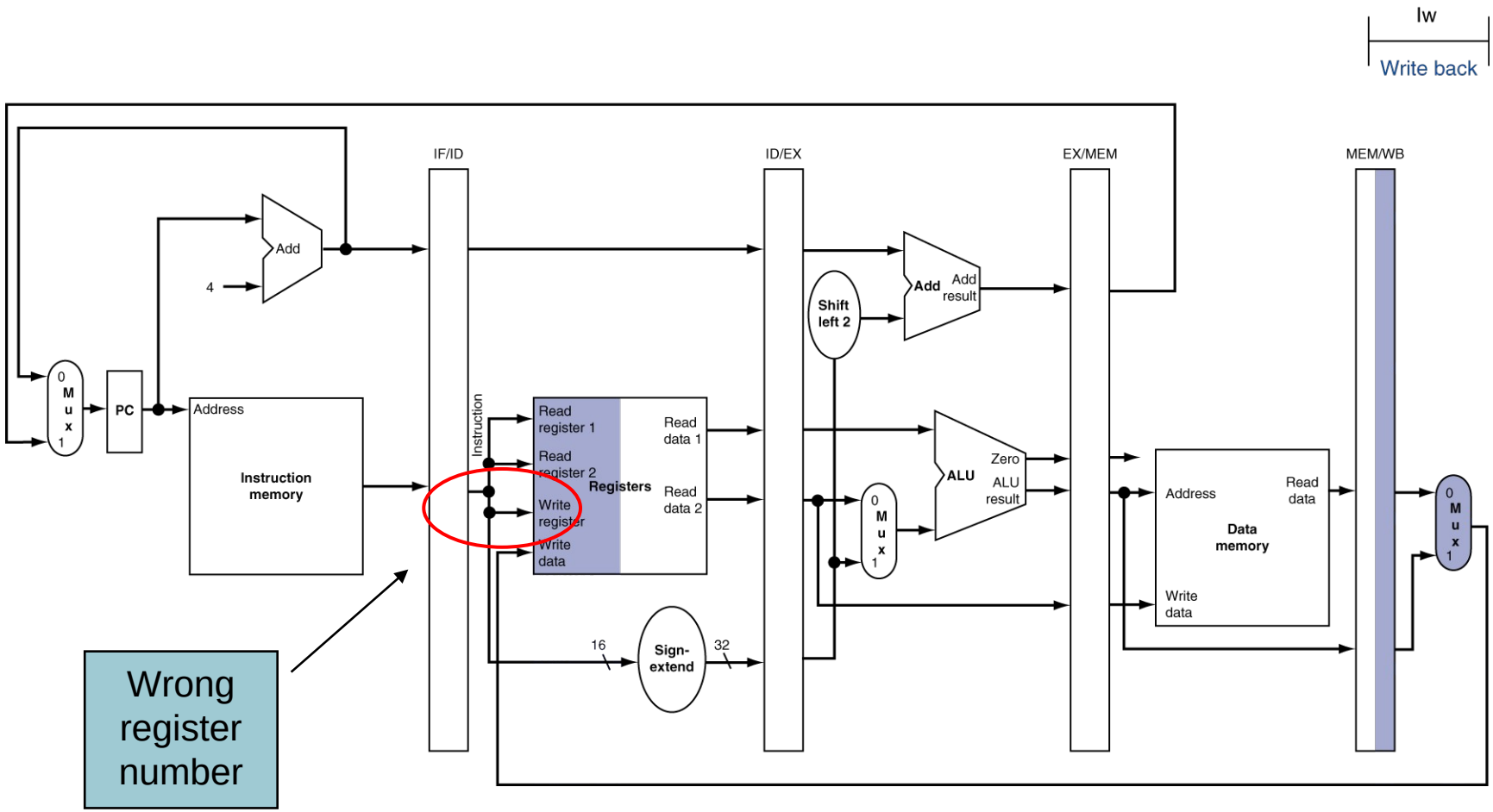
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
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

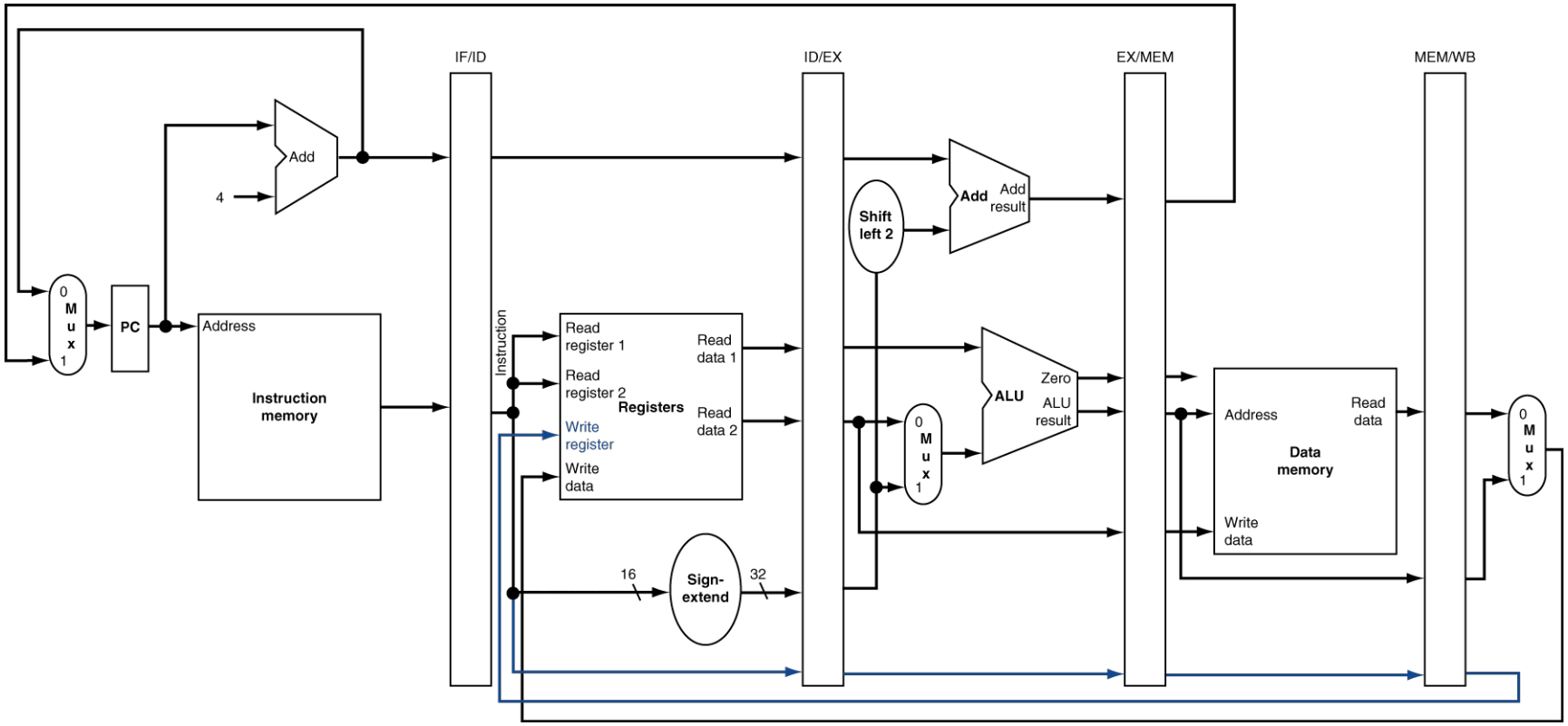
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
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
Data memory
Read data
Write data

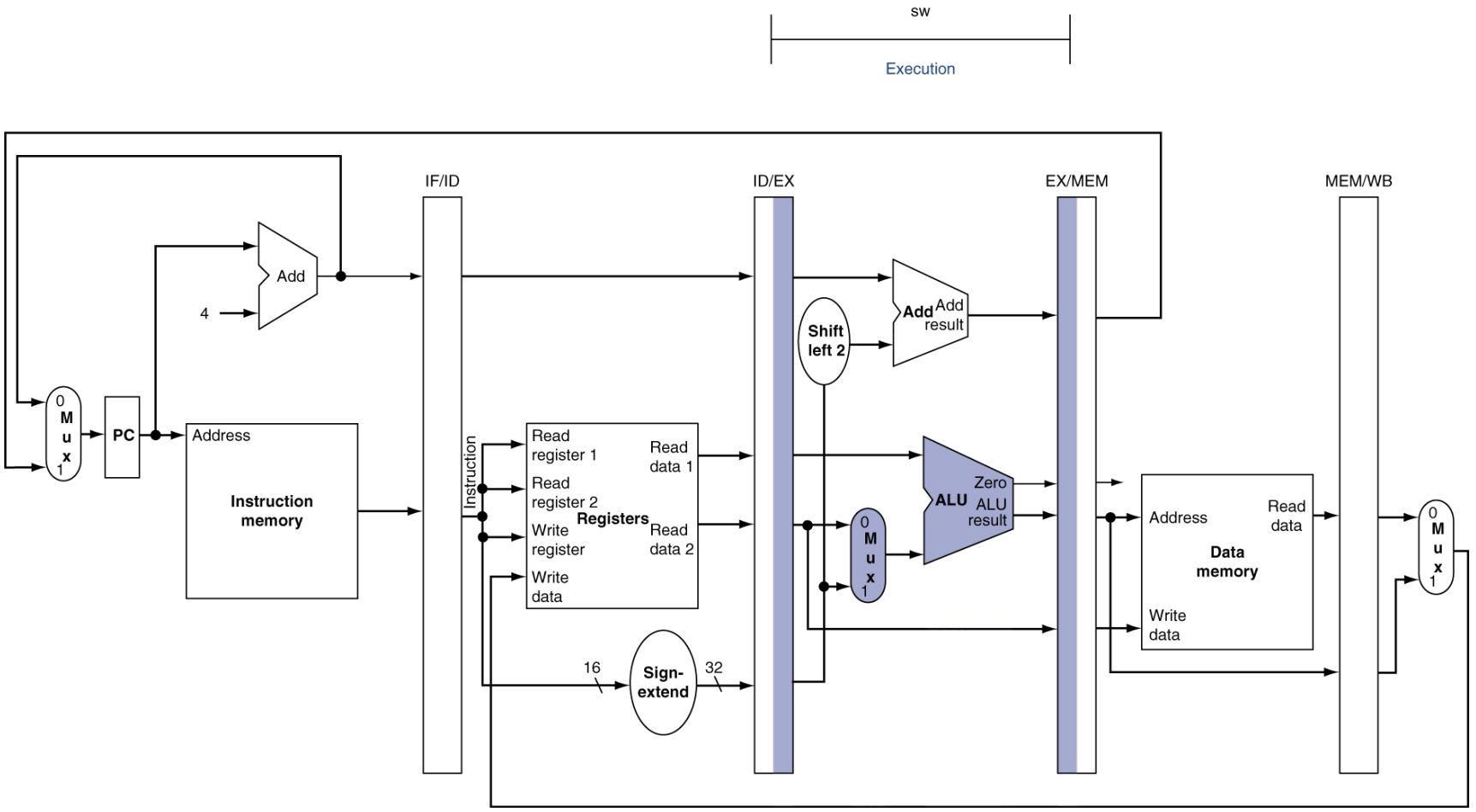
WB for Load


lw
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
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Wrong register number

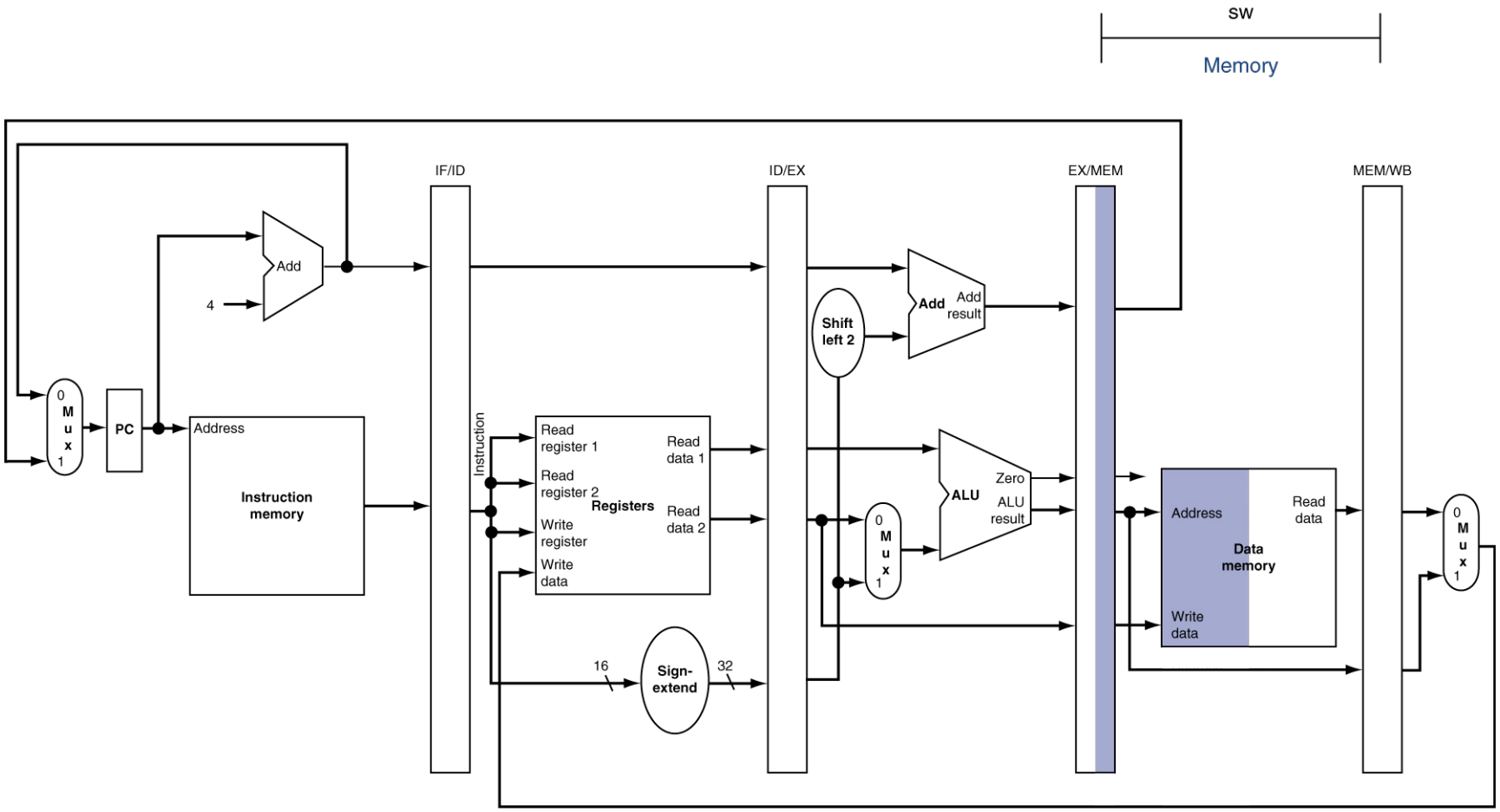
Corrected Datapath for Load



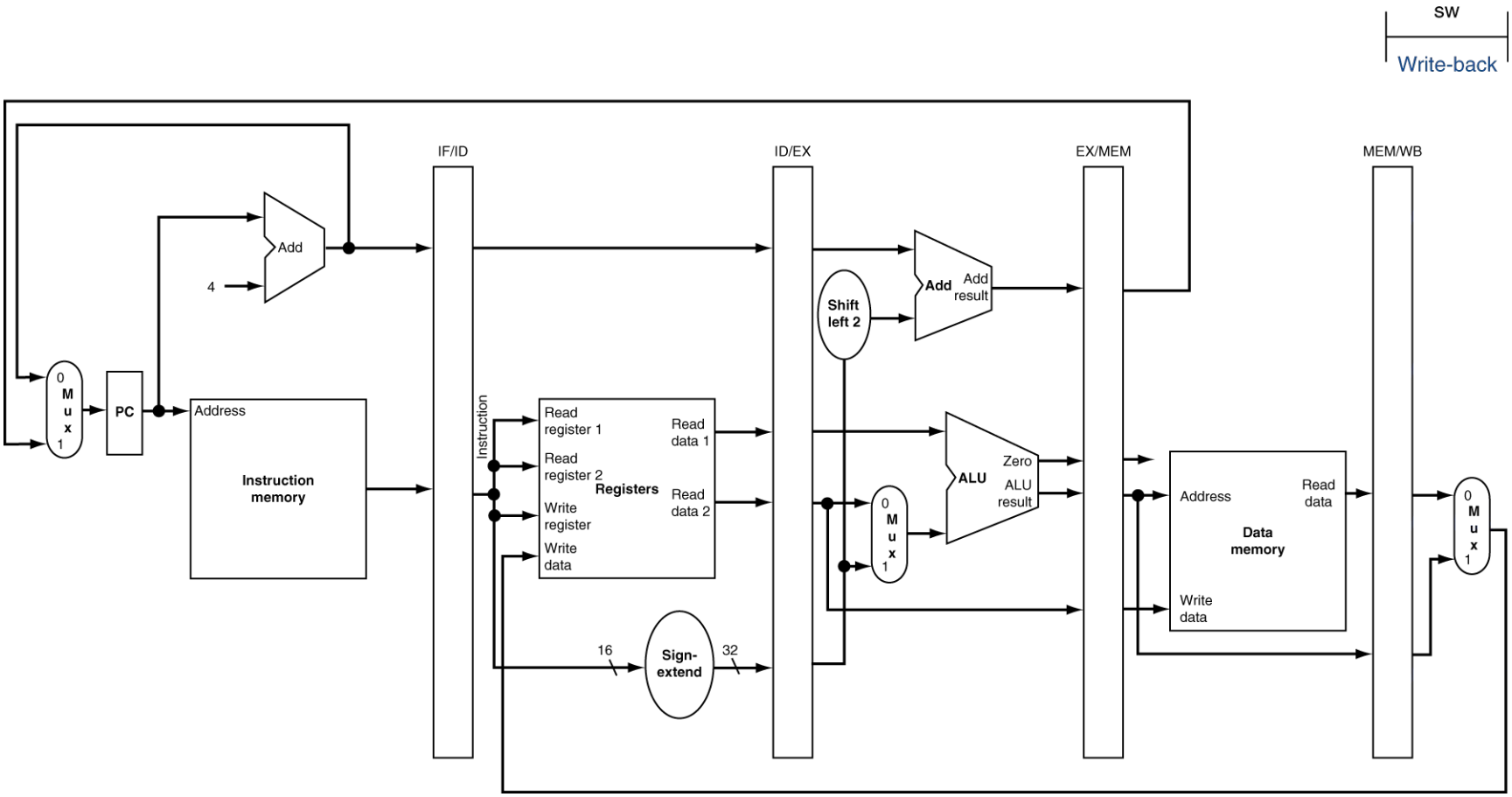
EX for Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
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

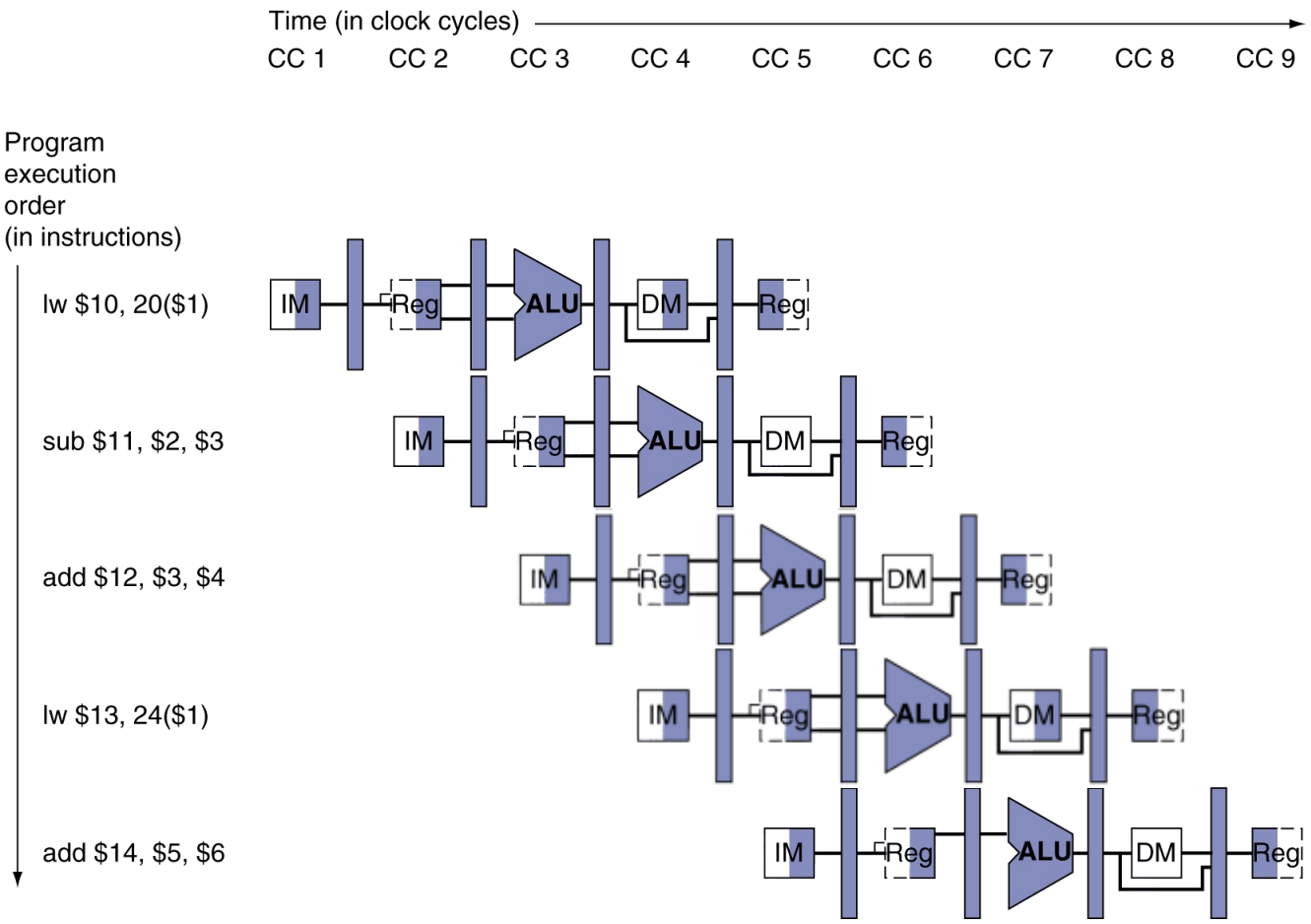
MEM for Store



WB for Store

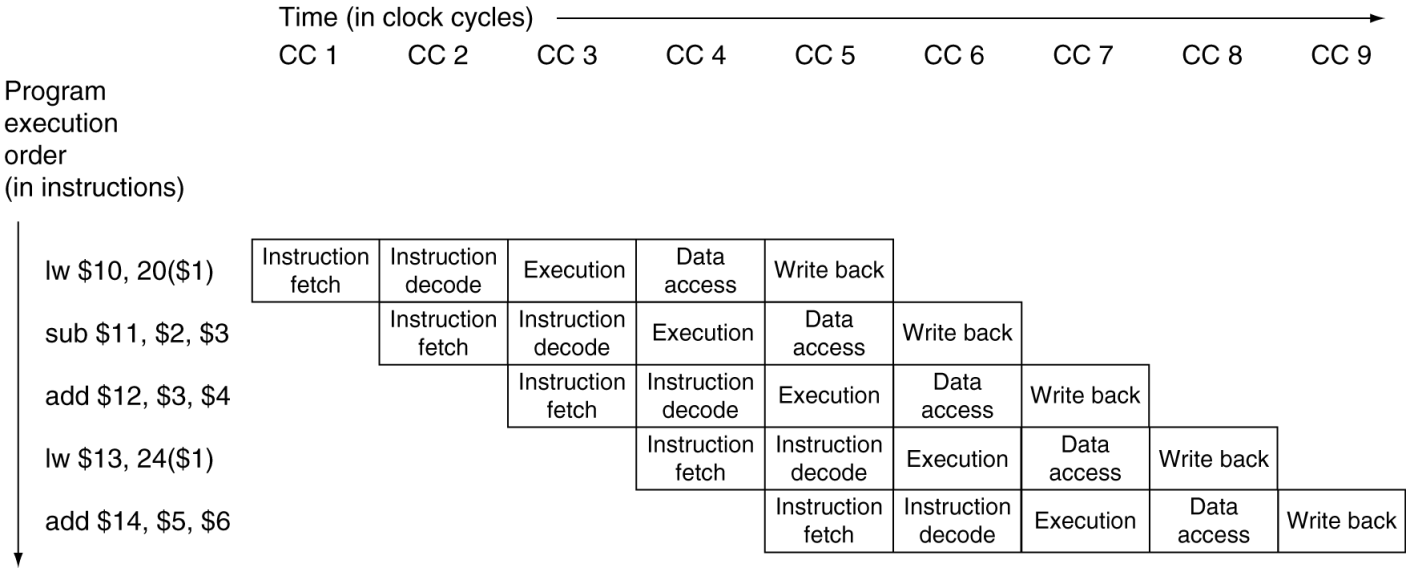

SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

Multi-Cycle Pipeline Diagram



Multi-Cycle Pipeline Diagram

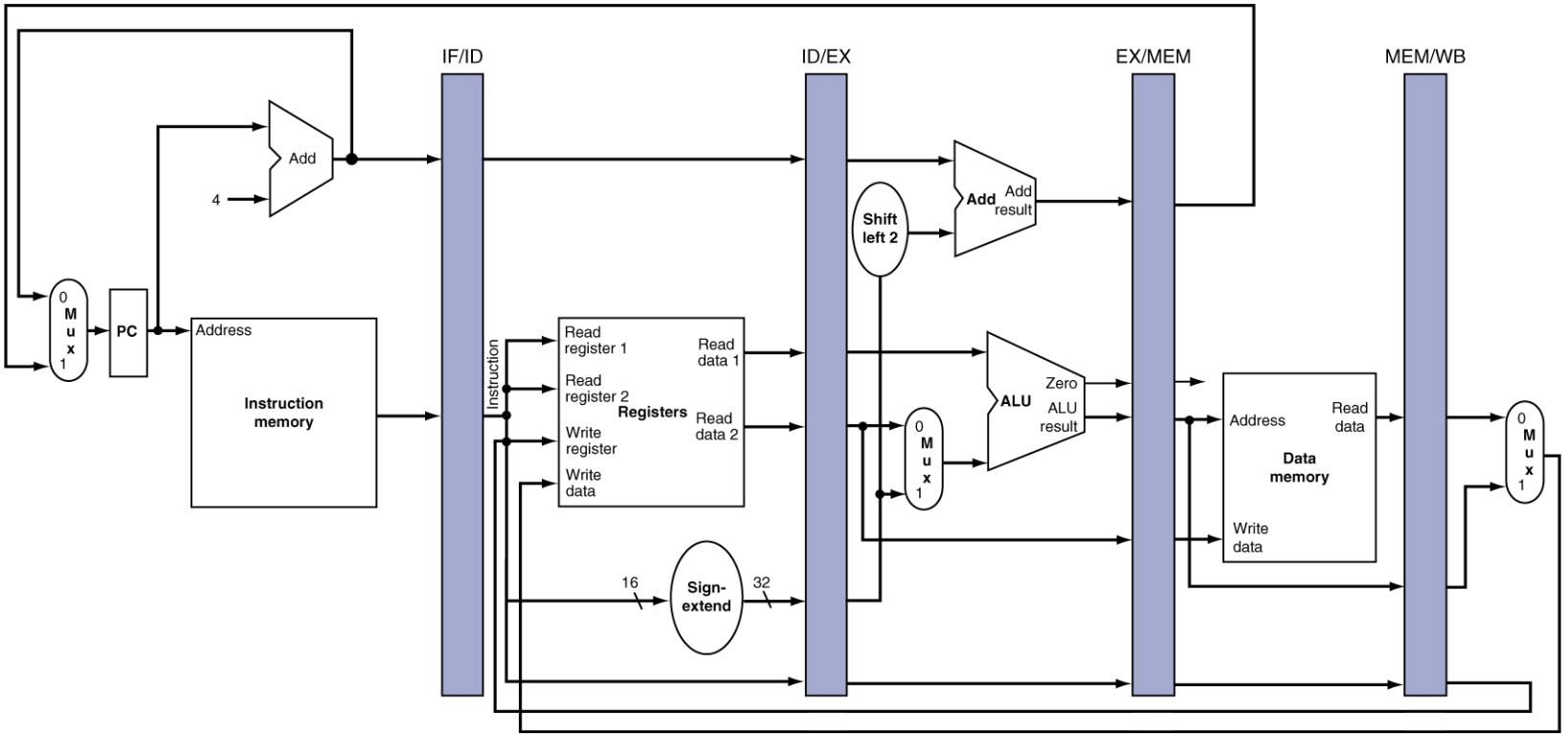
traditional form



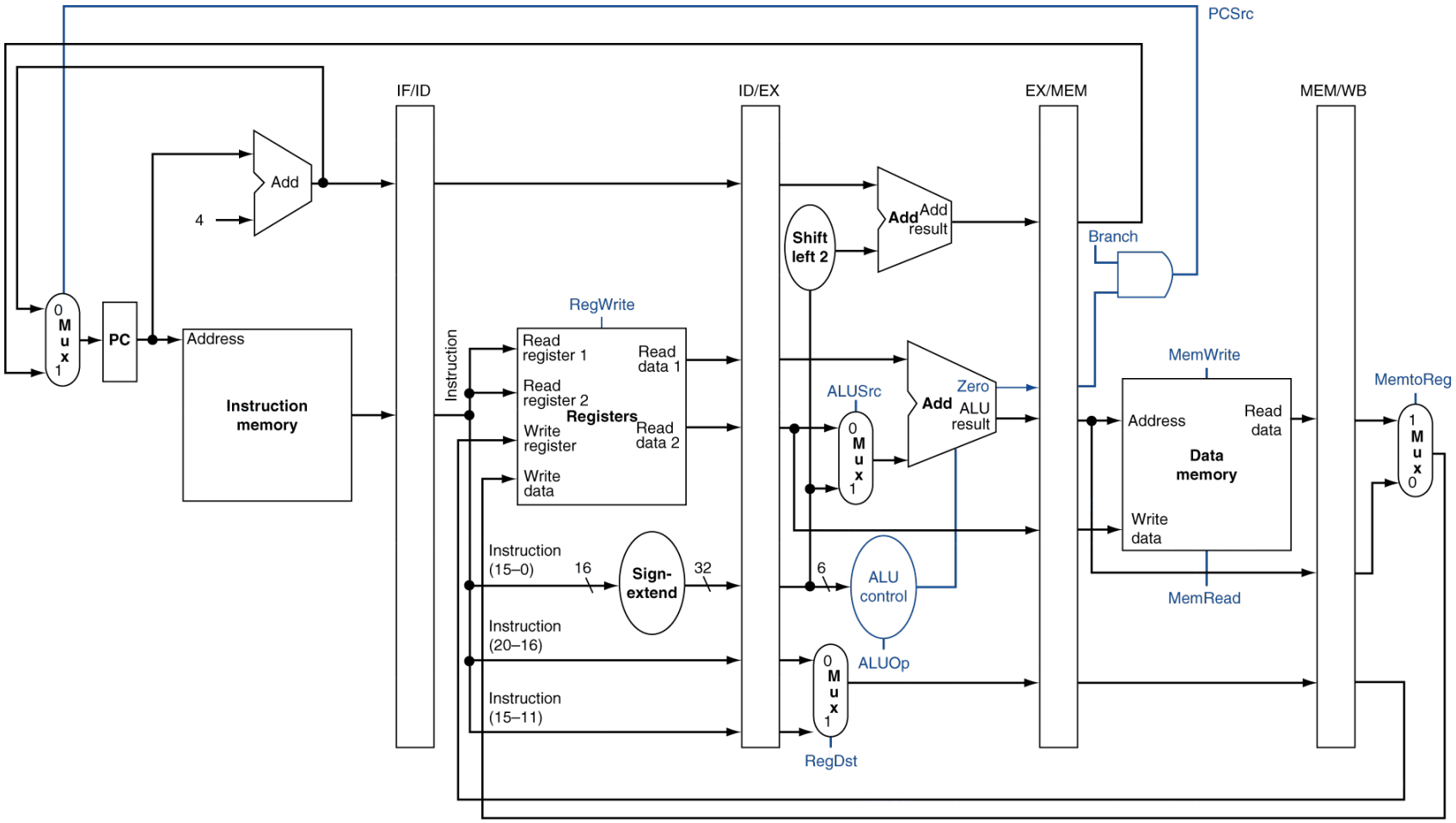
Single-Cycle Pipeline Diagram

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



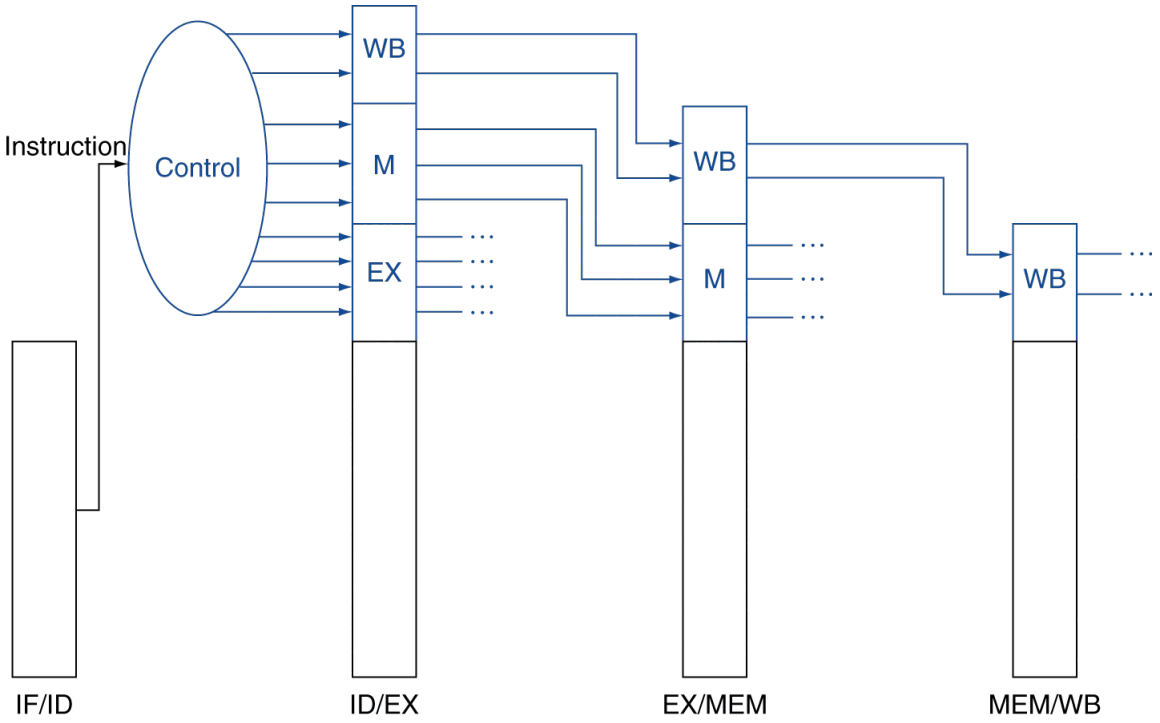
Pipelined Control (simplified)



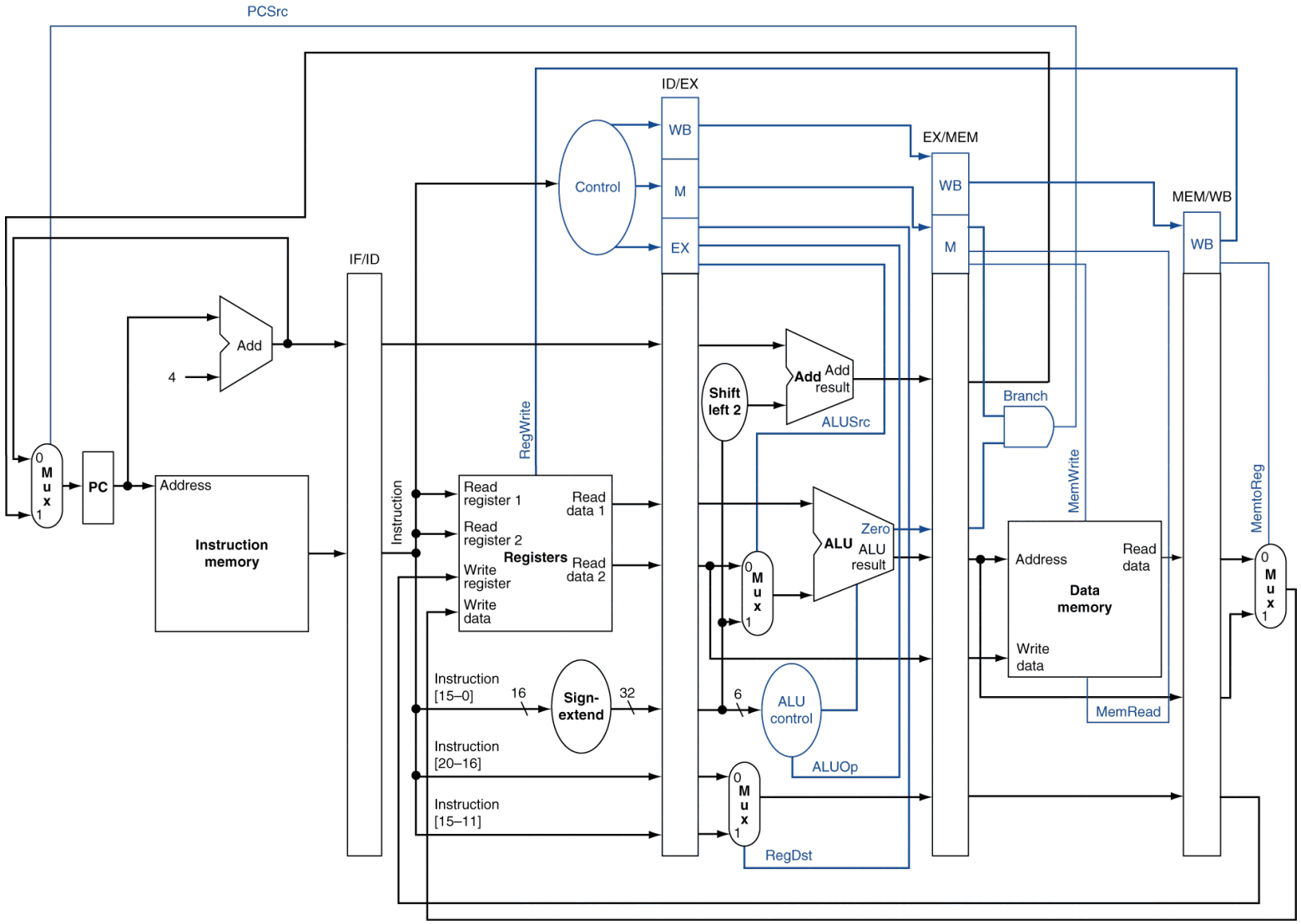
Pipelined Control

control signals derived from instruction

as in single-cycle implementation



Pipelined Control



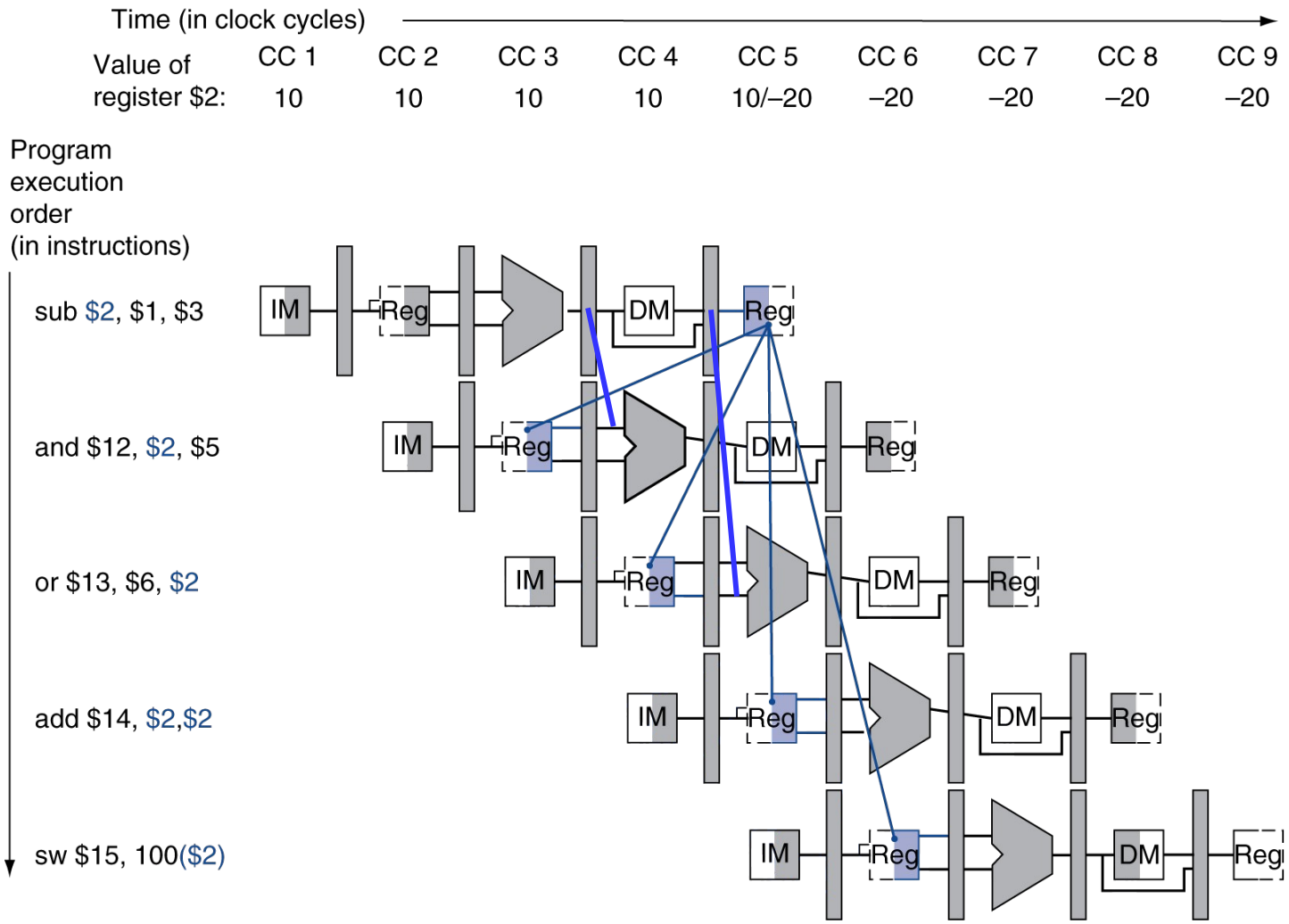
Data Hazards in ALU Instructions

- 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



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

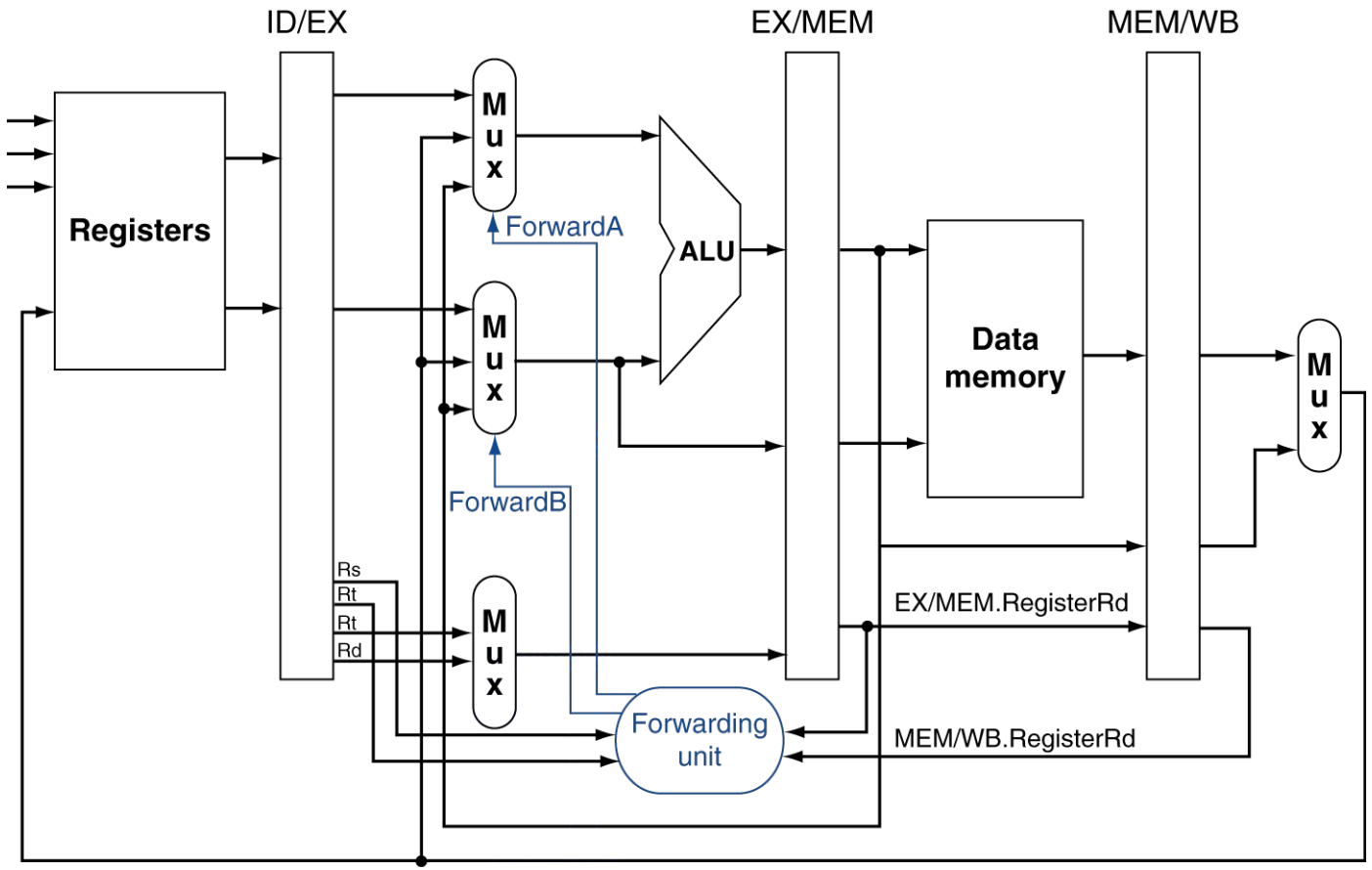
Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- but **only** if forwarding instruction will **write** to a register (e.g., not for `sw`)!
  - `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



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`

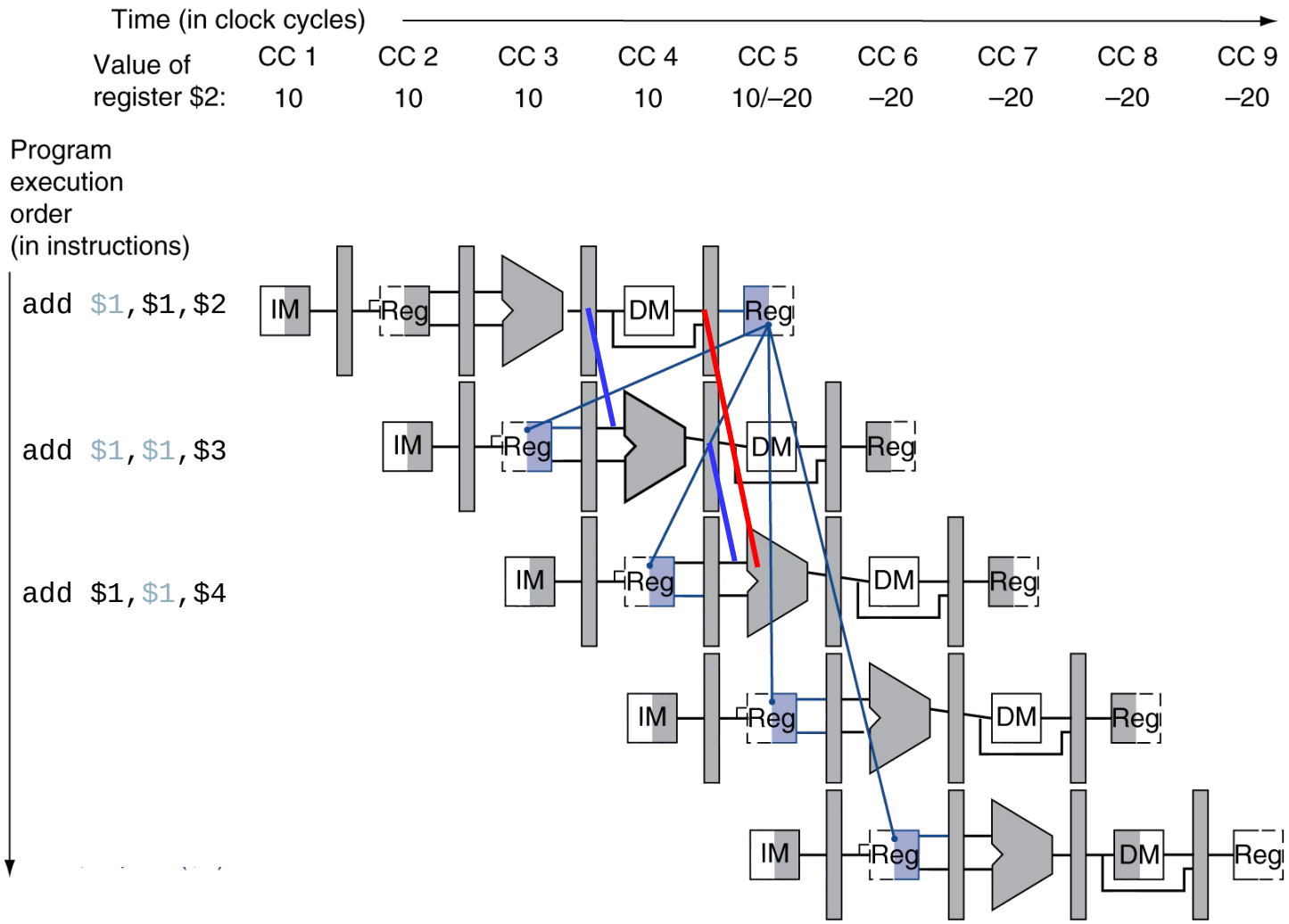
Double Data Hazard

- 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 is not true

Double Data Hazard

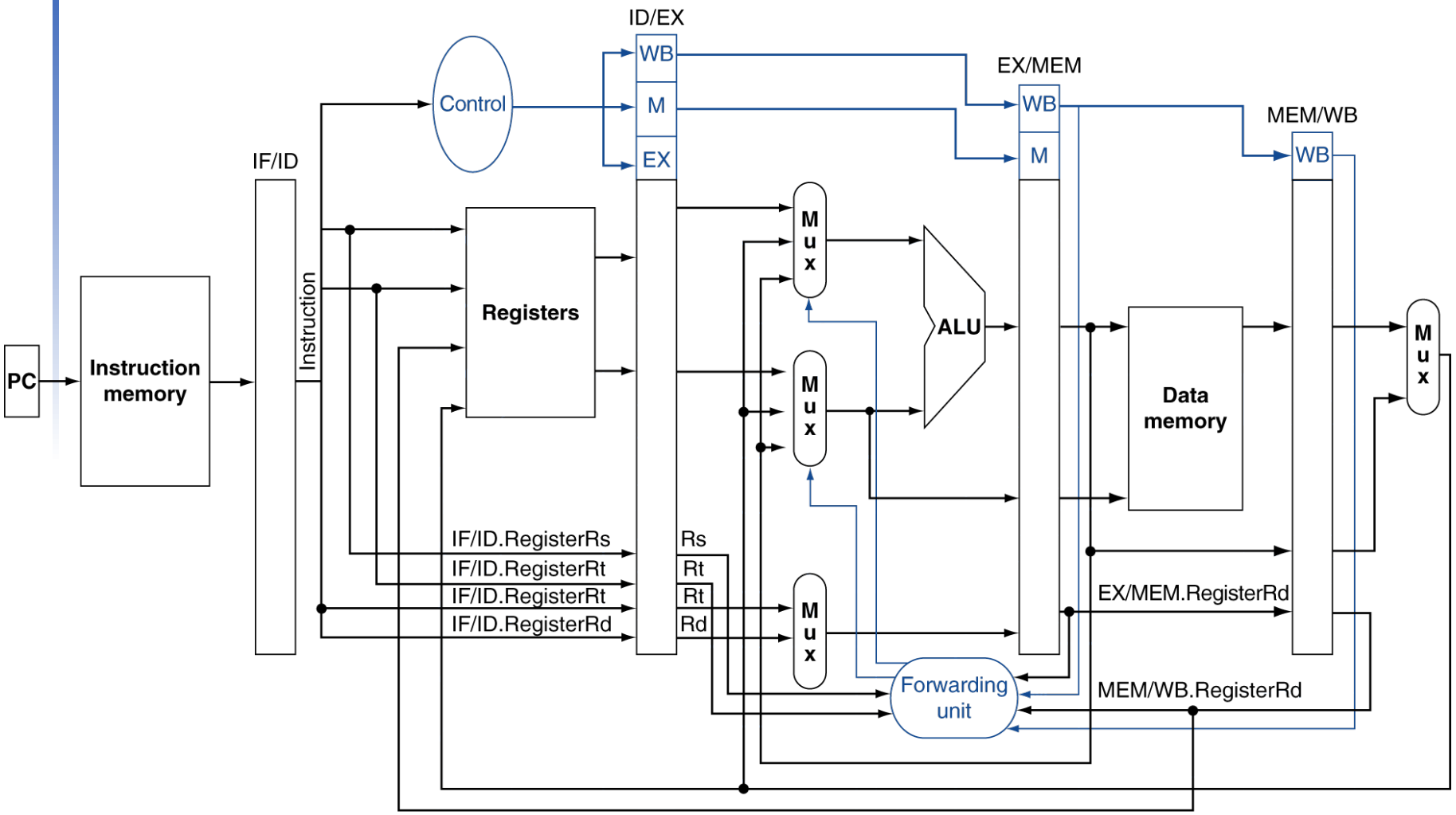


Revised Forwarding Condition

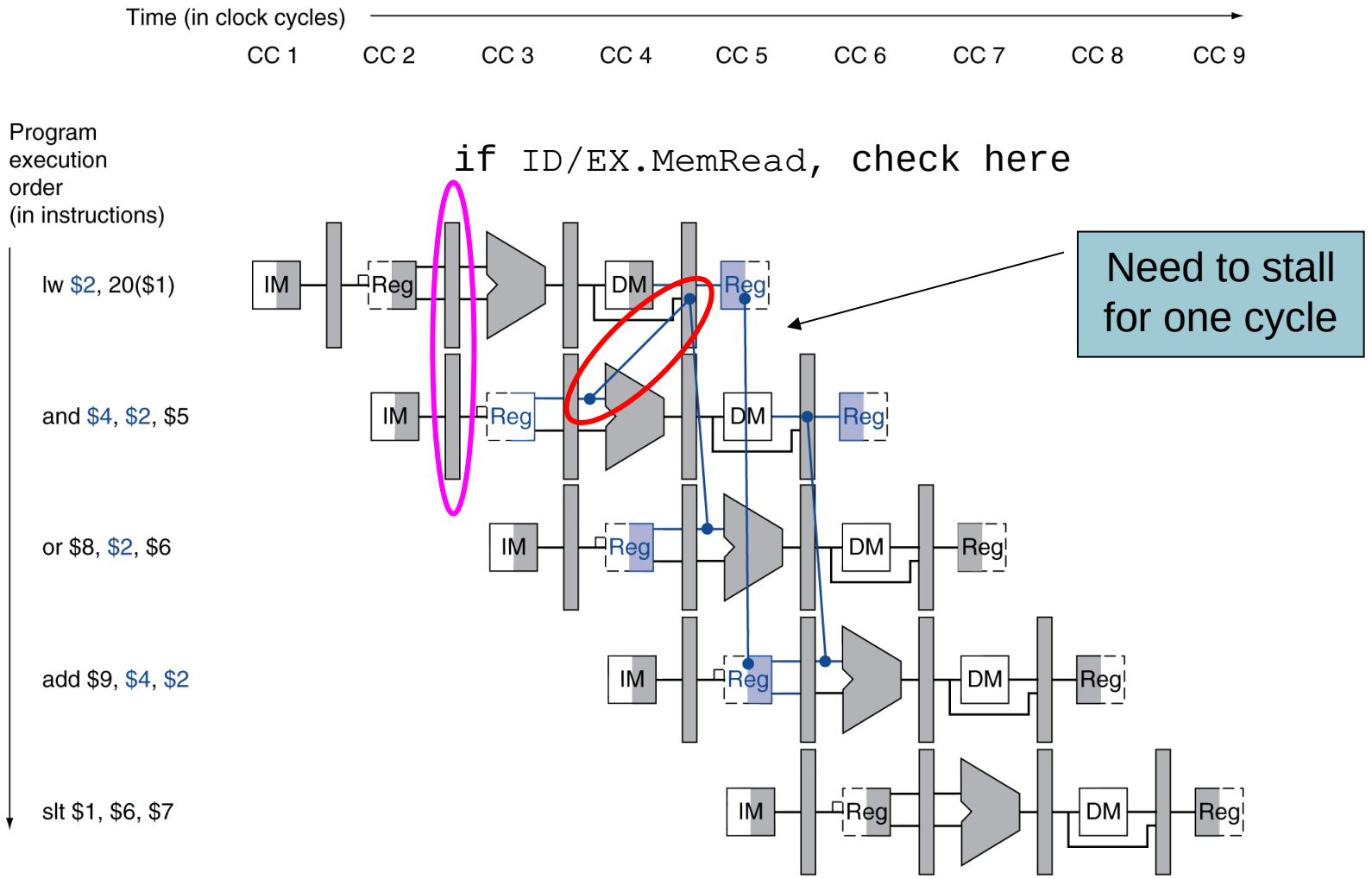
MEM hazard

- 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



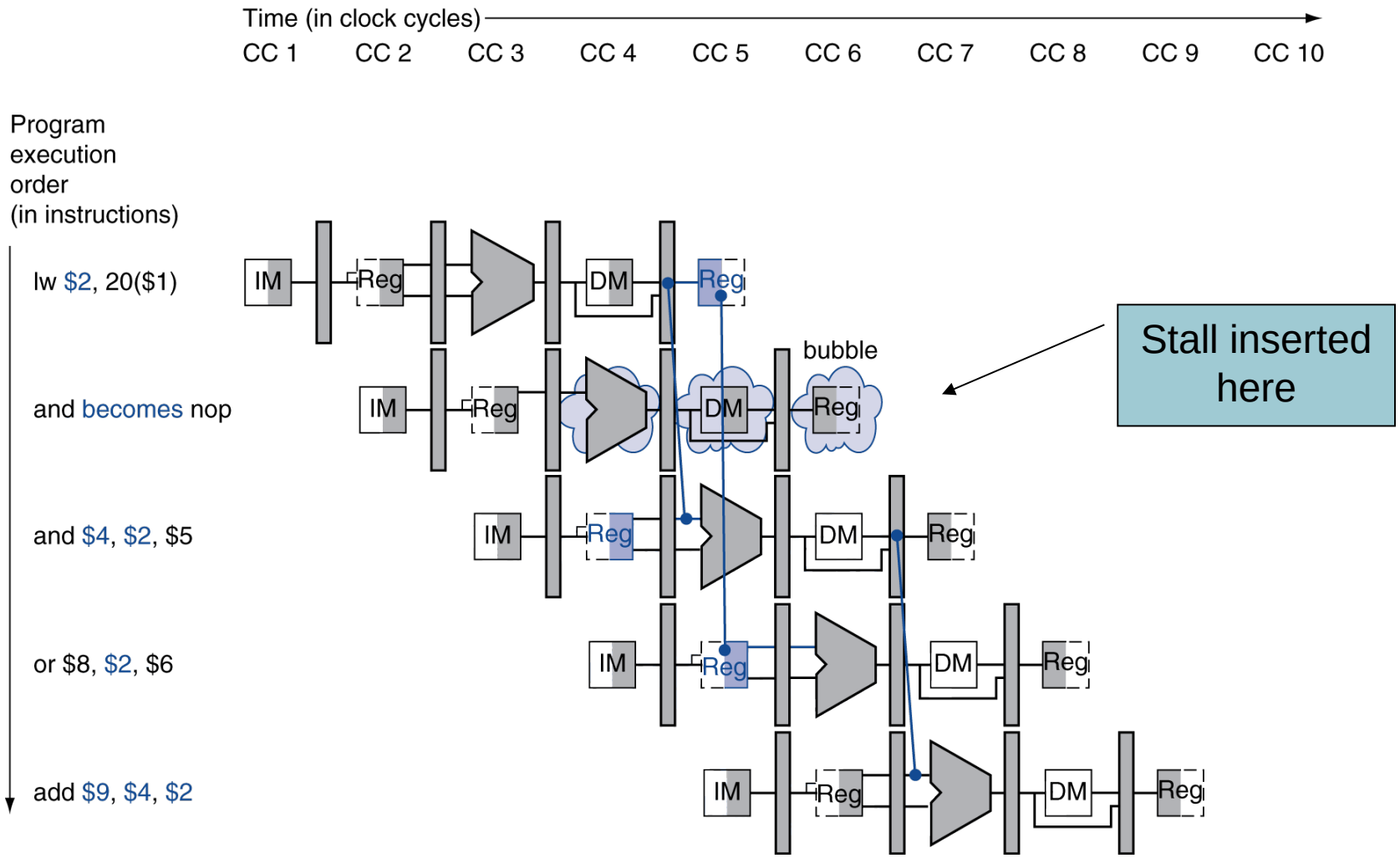
Load-Use Hazard Detection

- **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 target of load = input of computation
  - `ID/EX.MemRead and`
    `((ID/EX.RegisterRt = IF/ID.RegisterRs) or`
    `(ID/EX.RegisterRt = IF/ID.RegisterRt))`
- If detected, stall and insert bubble

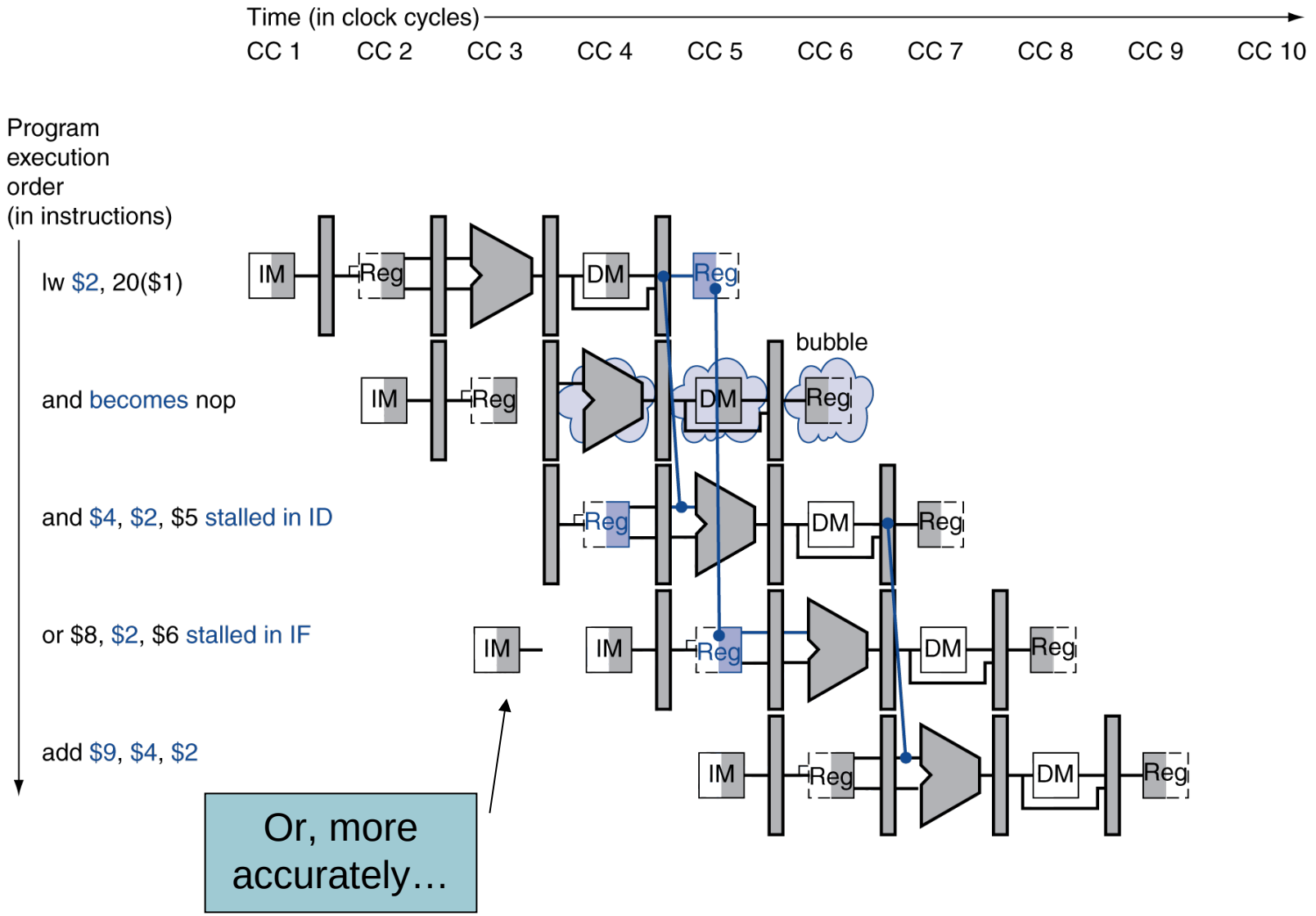
How to Stall the Pipeline?

- Force control values in ID/EX register to 0
  → EX, MEM and WB do `NOOP` (no-operation)
- Prevent update of PC and IF/ID register
  - Current 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

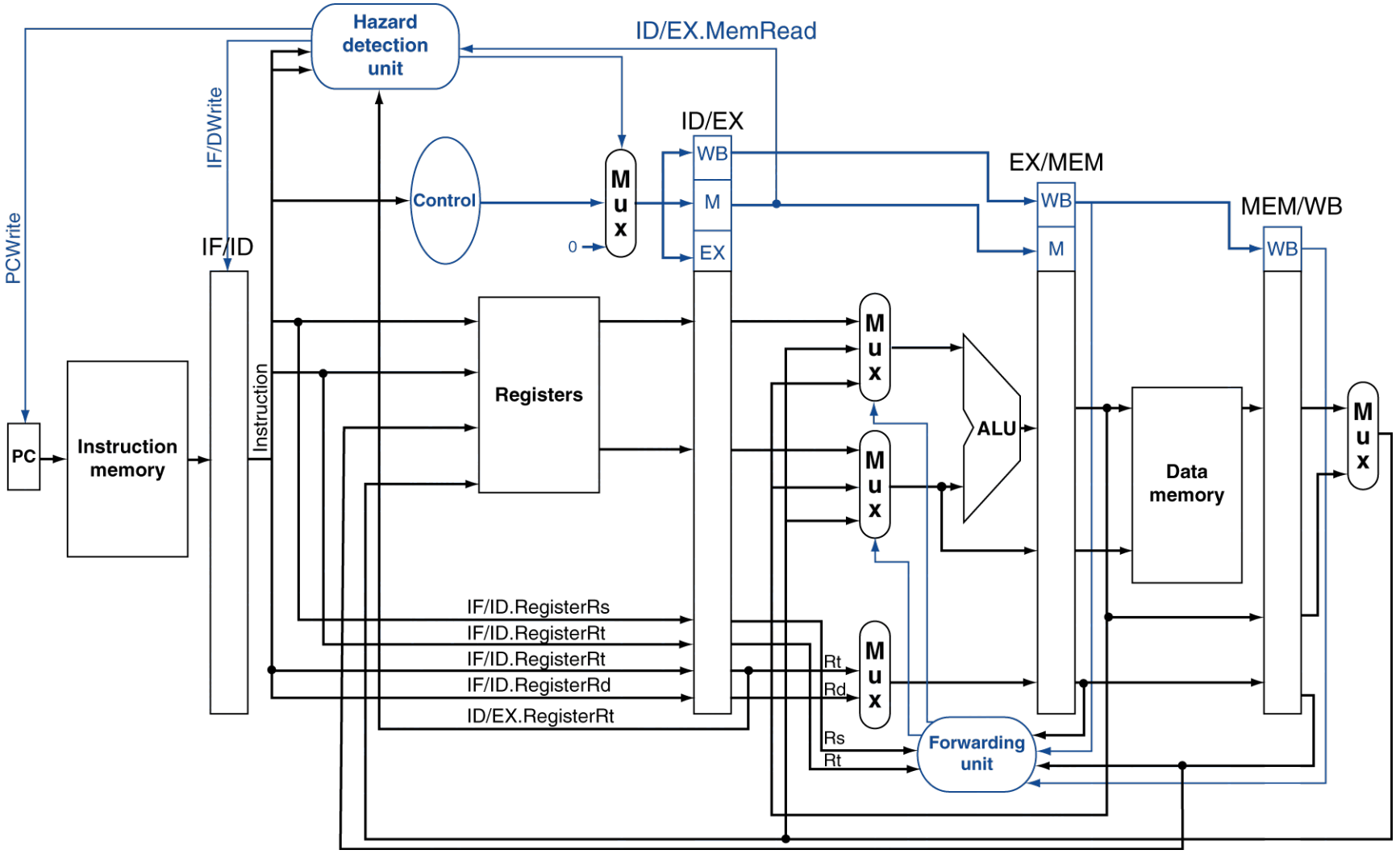
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



Stalls and Performance

The BIG Picture

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

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

Pipeline with Exceptions