Pipelined CPU

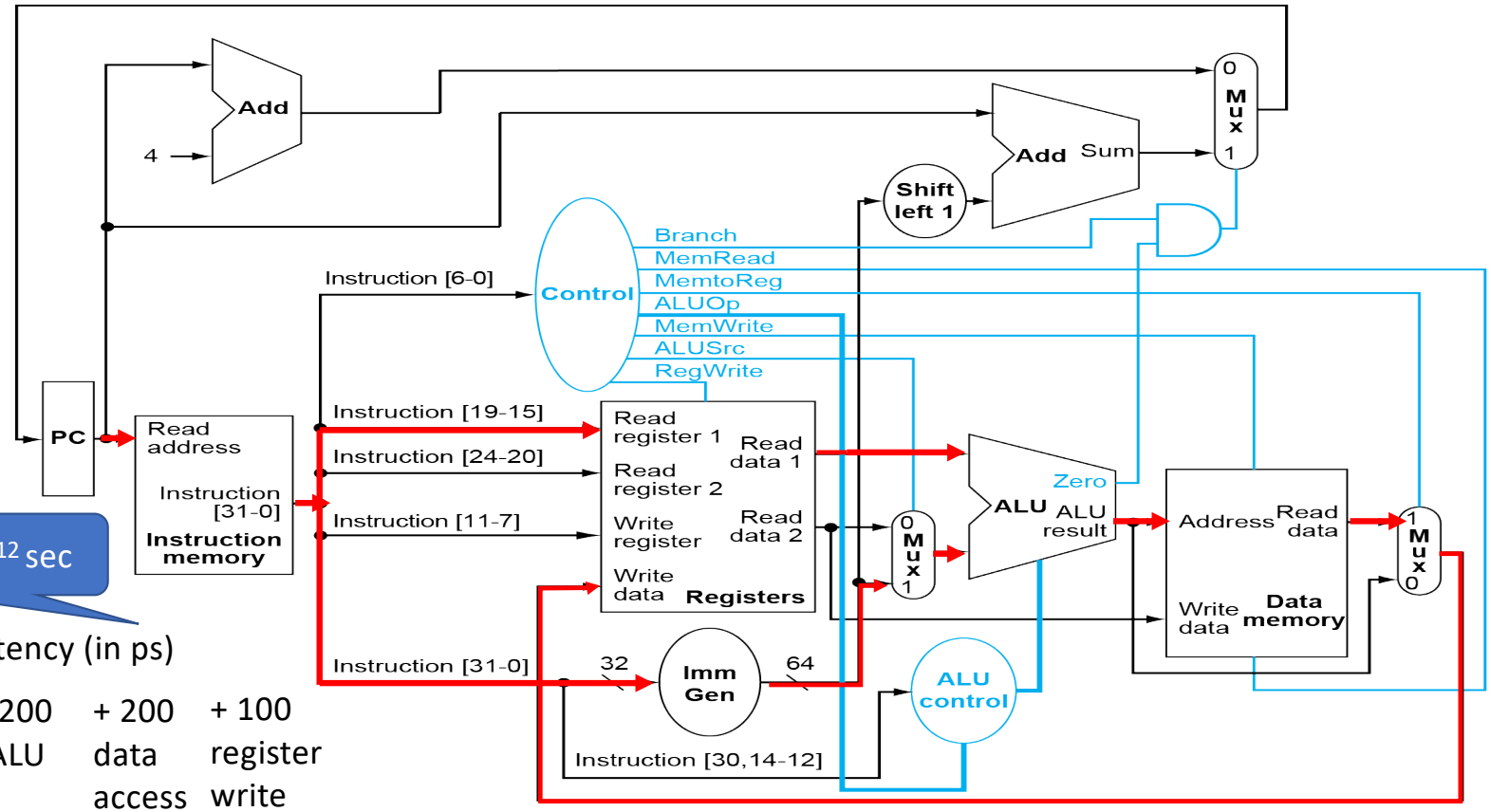
Jinyang Li

Based on the slides of Patterson and Hennessy

What we've learnt so far

- Combinatorial logic
  - ALU
- Sequential logic
  - Clocks and basic state elements (SR latch, D latch, flip-flop)
- The single-cycle CPU design

Single-cycle CPU uses a slow clock



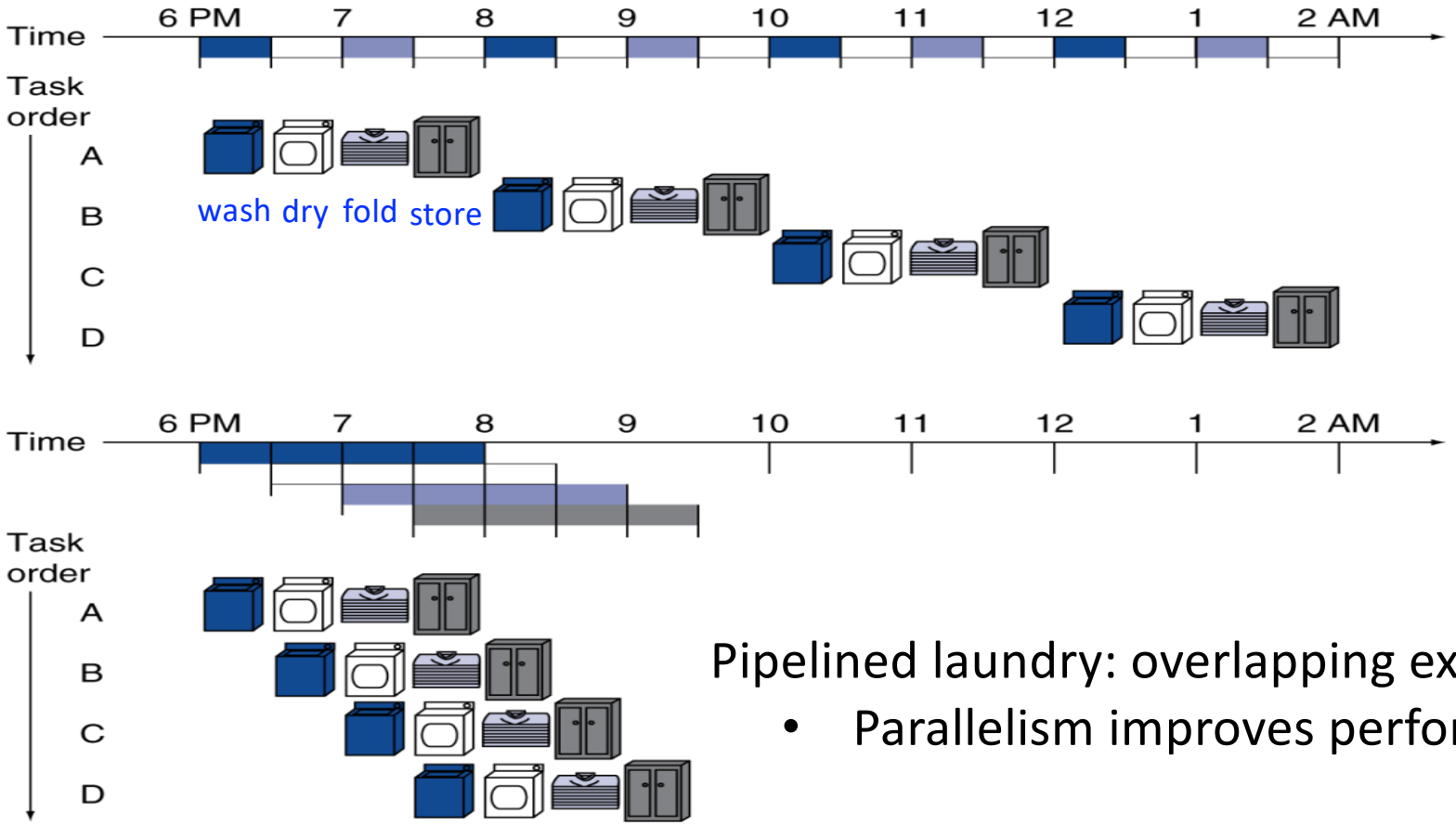
1 picosecond = 10^{-12} sec

Load instruction latency (in ps)

200	+ 100	+ 200	+ 200	+ 100
instruction fetch	register read	ALU	data access	register write

Clock cycle >= 800 ps

Pipelining: a laundry analogy



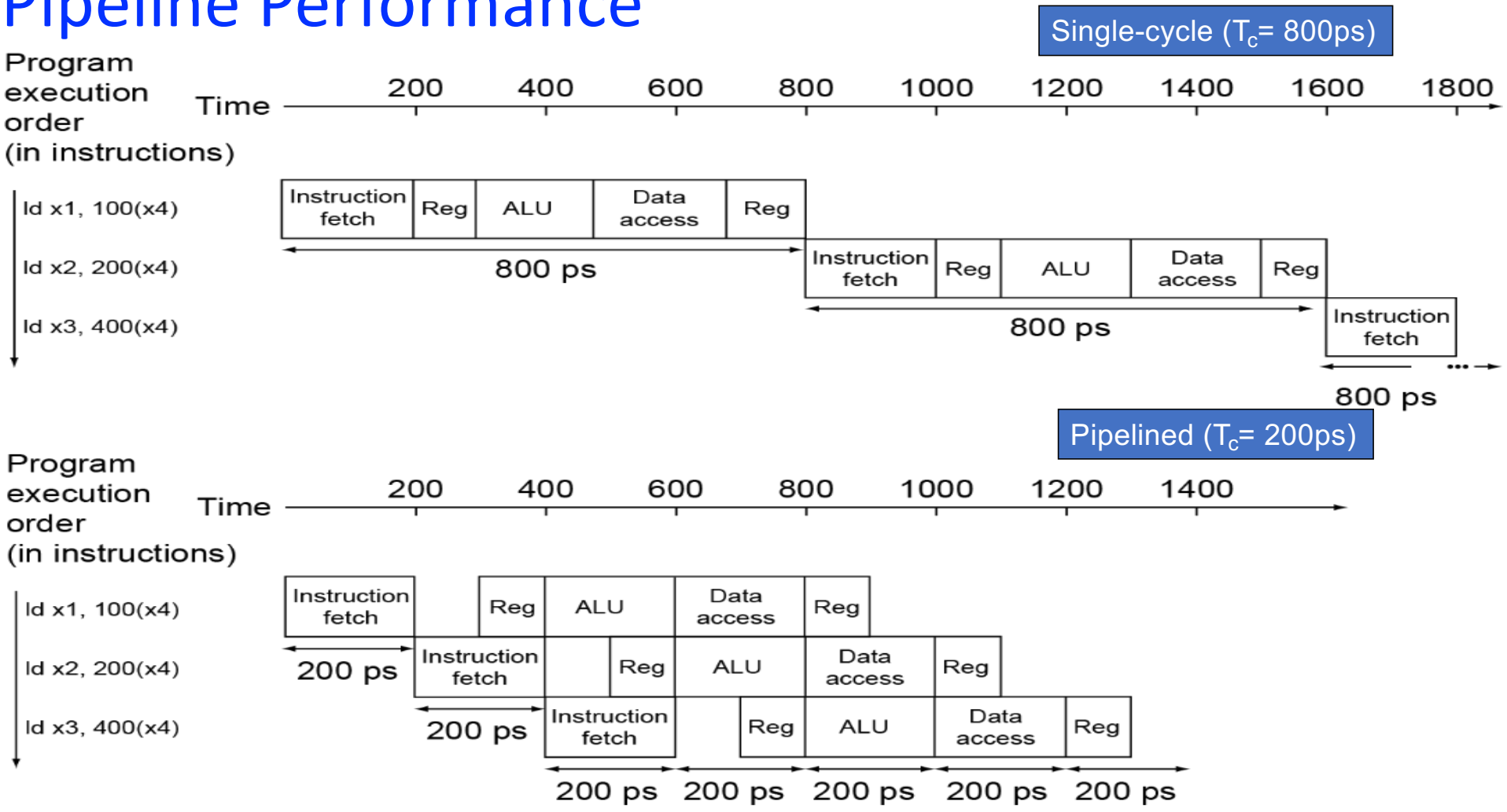
Pipelined laundry: overlapping execution

- Parallelism improves performance

RISC-V Pipeline

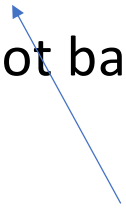
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance



Pipeline Speedup

- Pipelining increases throughput (instructions/sec)
  - Latency (time for each instruction) does not decrease
- If all stages are balanced (i.e., all take the same time)
  - $\text{throughput}_{\text{pipelined}} = \text{number-of-stages} * \text{throughput}_{\text{nonpipelined}}$
  - If not balanced, speedup is less



Throughput = 1/time between instructions

Pipelining and ISA Design

- RISC-V 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

Pipeline challenges: 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

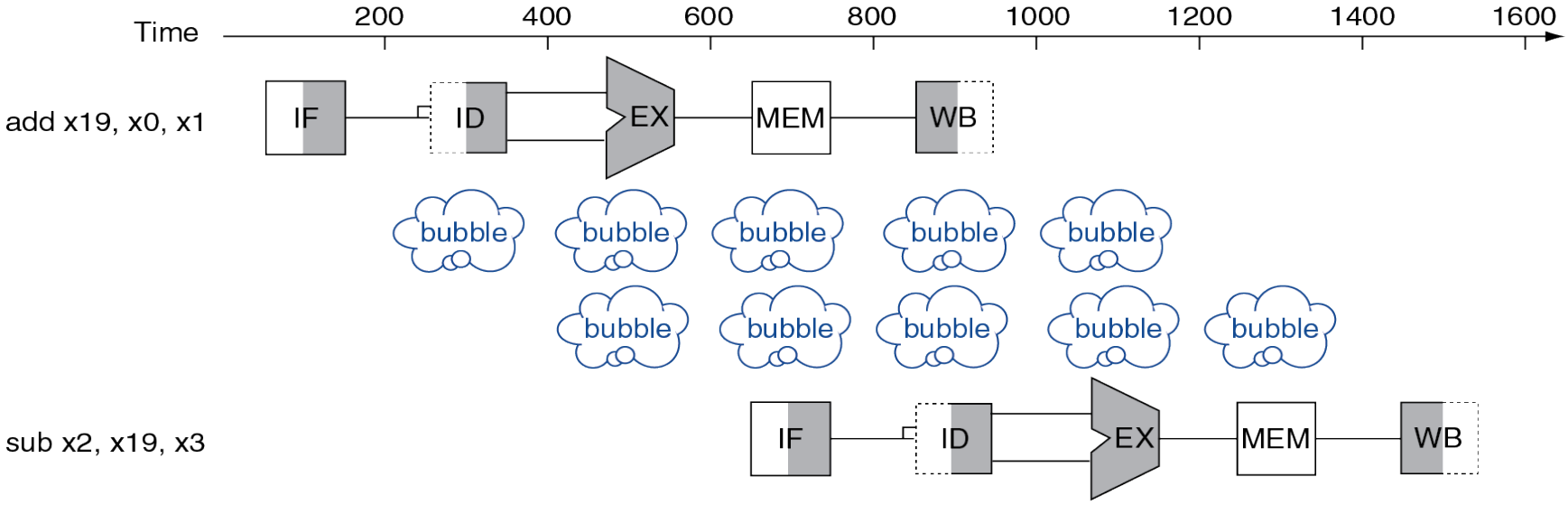
Structure Hazards

- Conflict use of a single resource
- Example: Suppose CPU uses a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Solution: Use separate instruction/data memories

Data Hazards

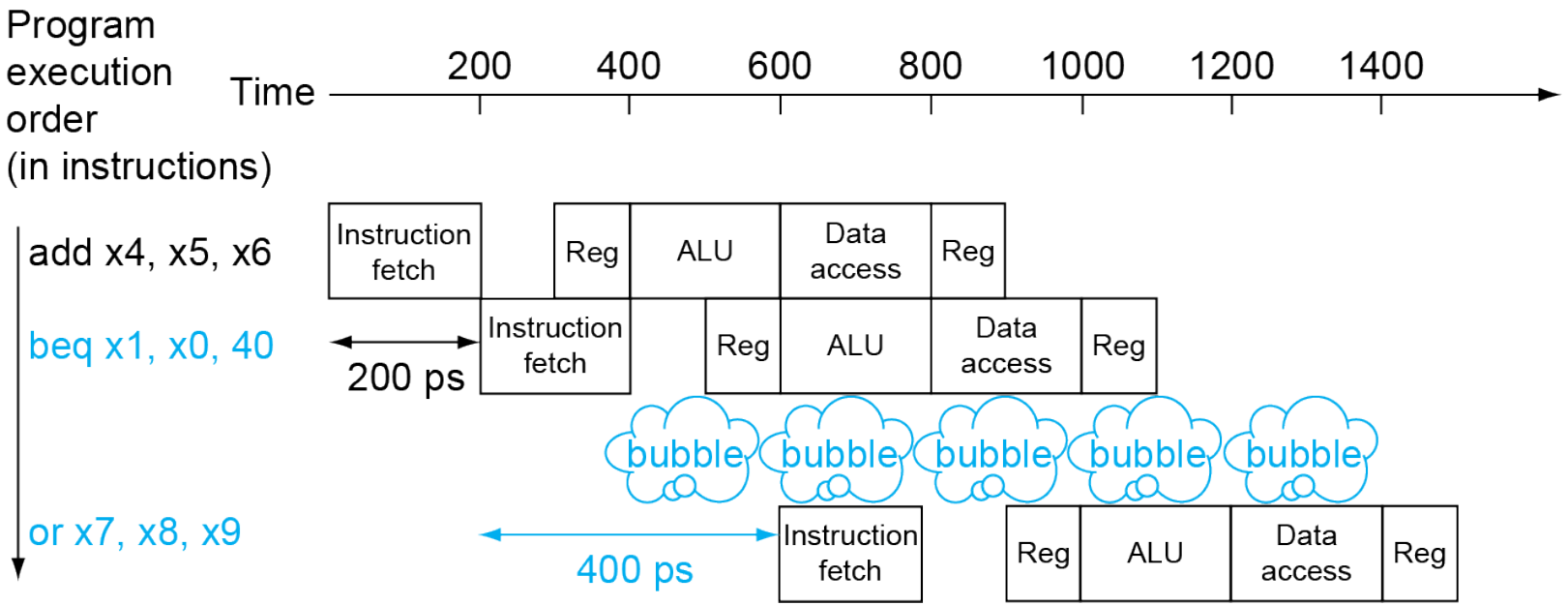
- An instruction depends on the previous instruction to complete its data read/write

```
add    x19, x0, x1
sub    x2, x19, x3
```



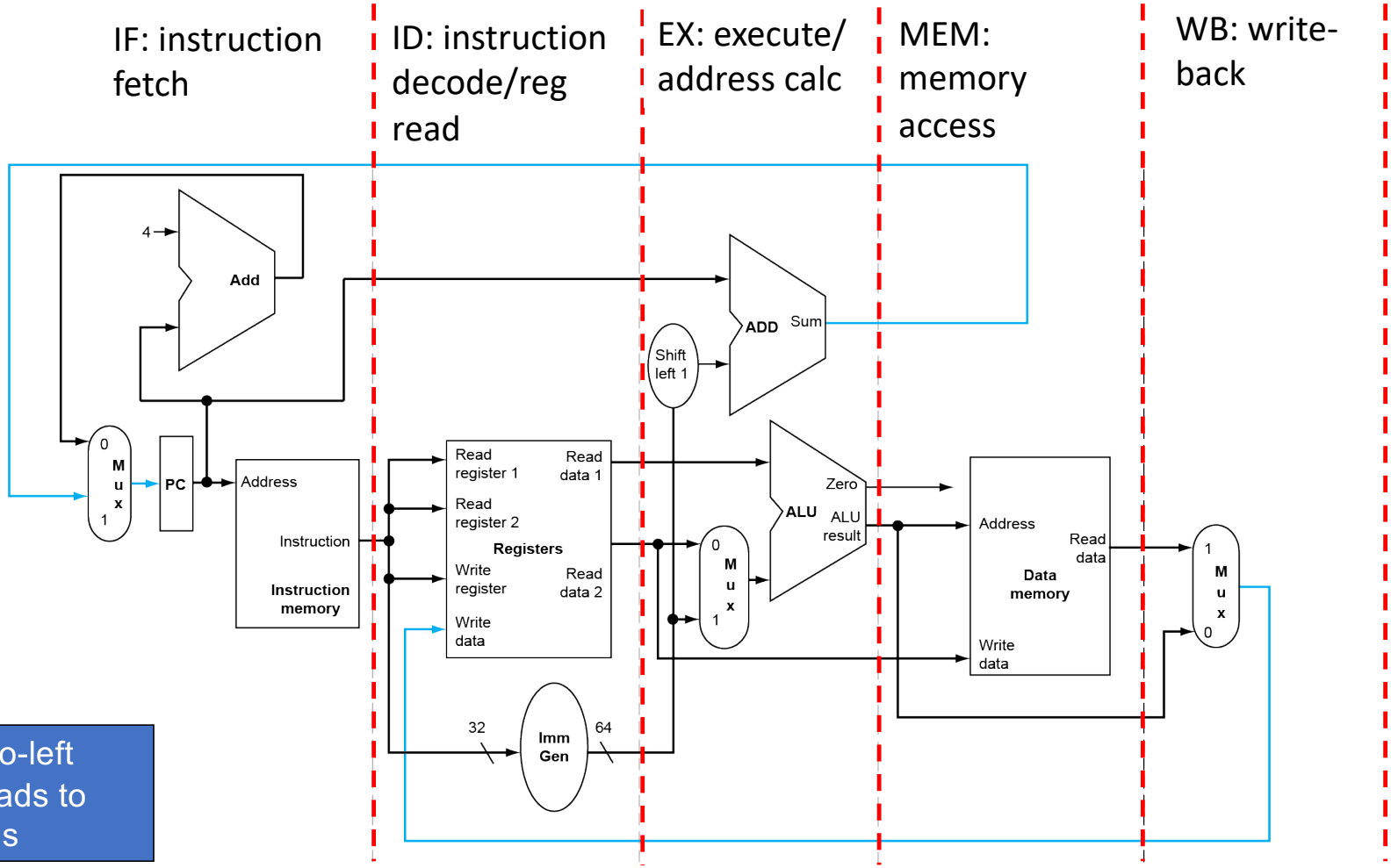
Control hazard

- Wait until branch outcome determined before fetching next instruction



A basic pipelined RISC-V CPU

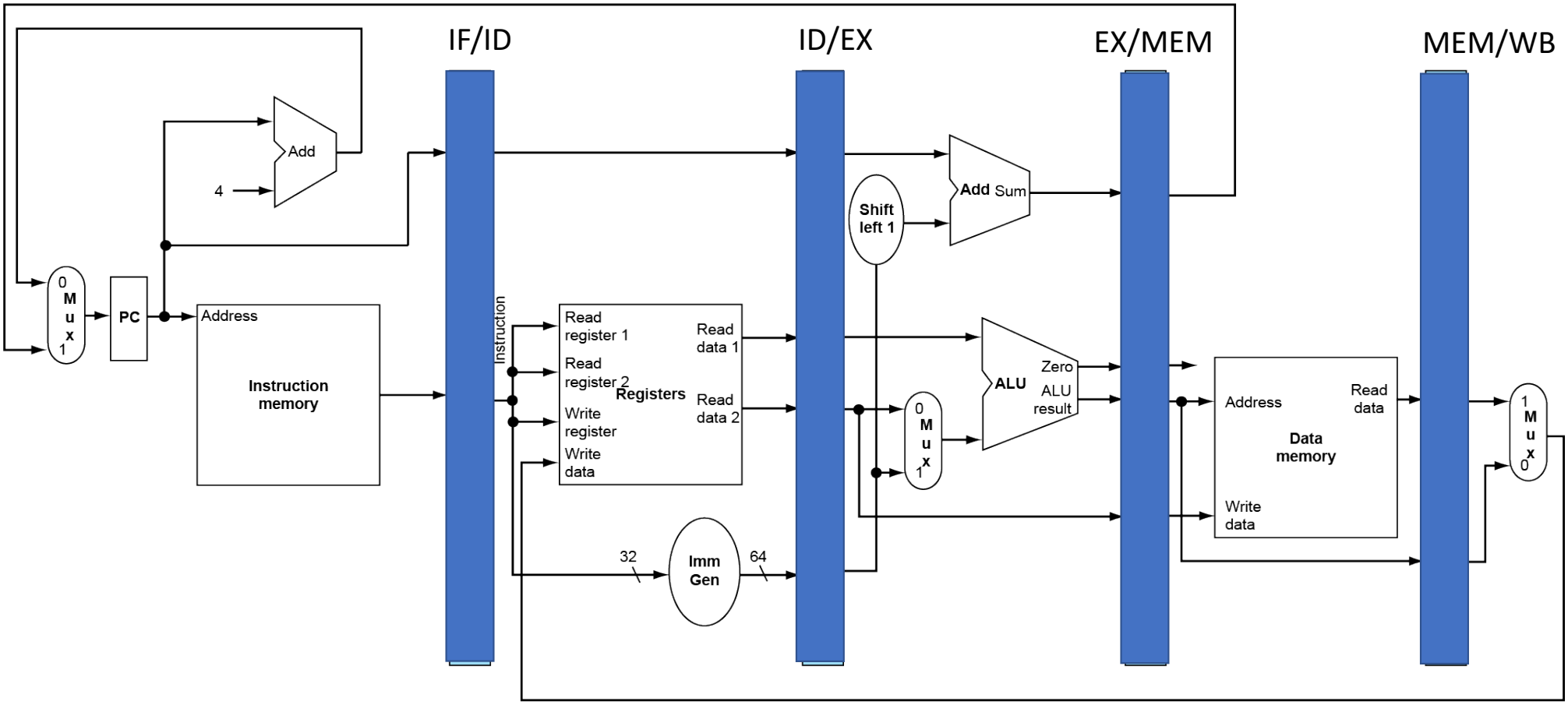
Pipelined Datapath



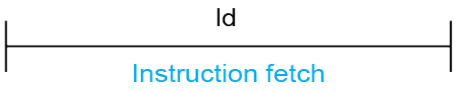
Right-to-left flow leads to hazards

Pipeline registers

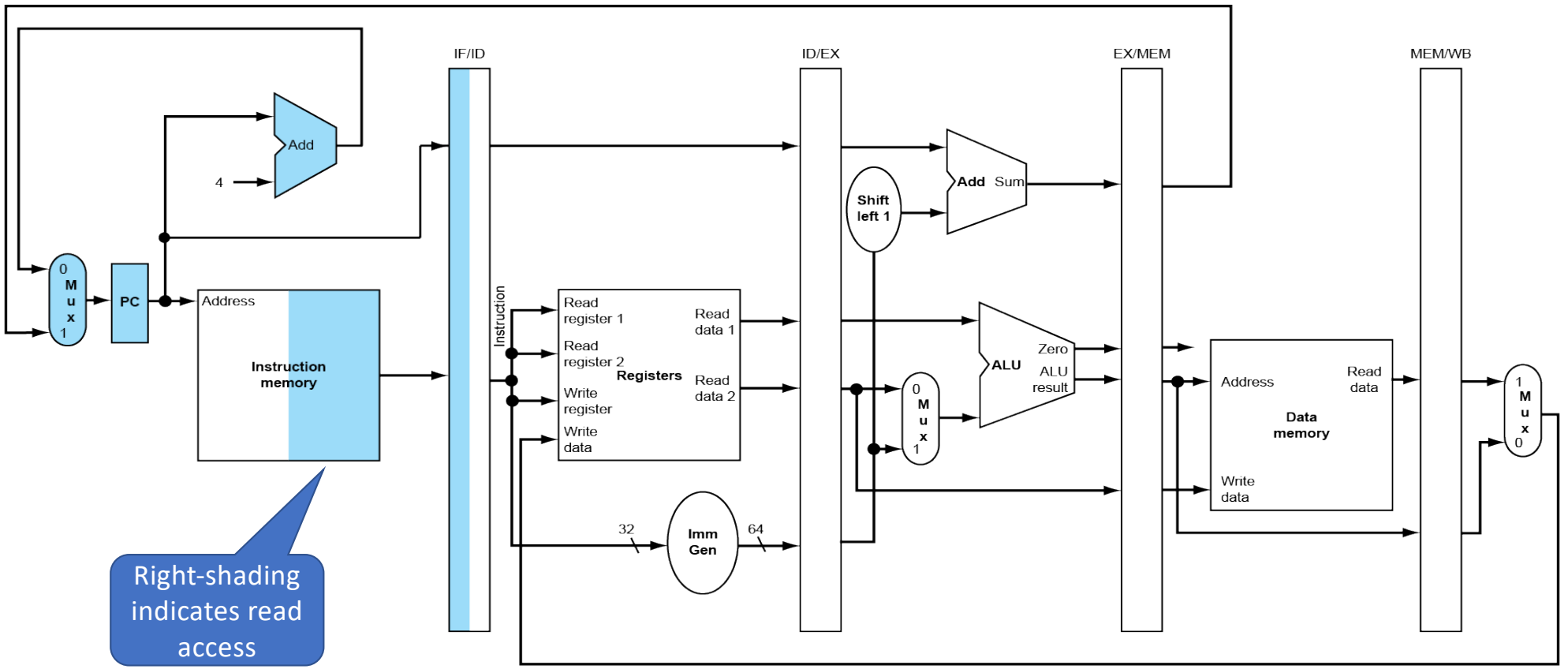
Needed to hold information produced in previous cycle



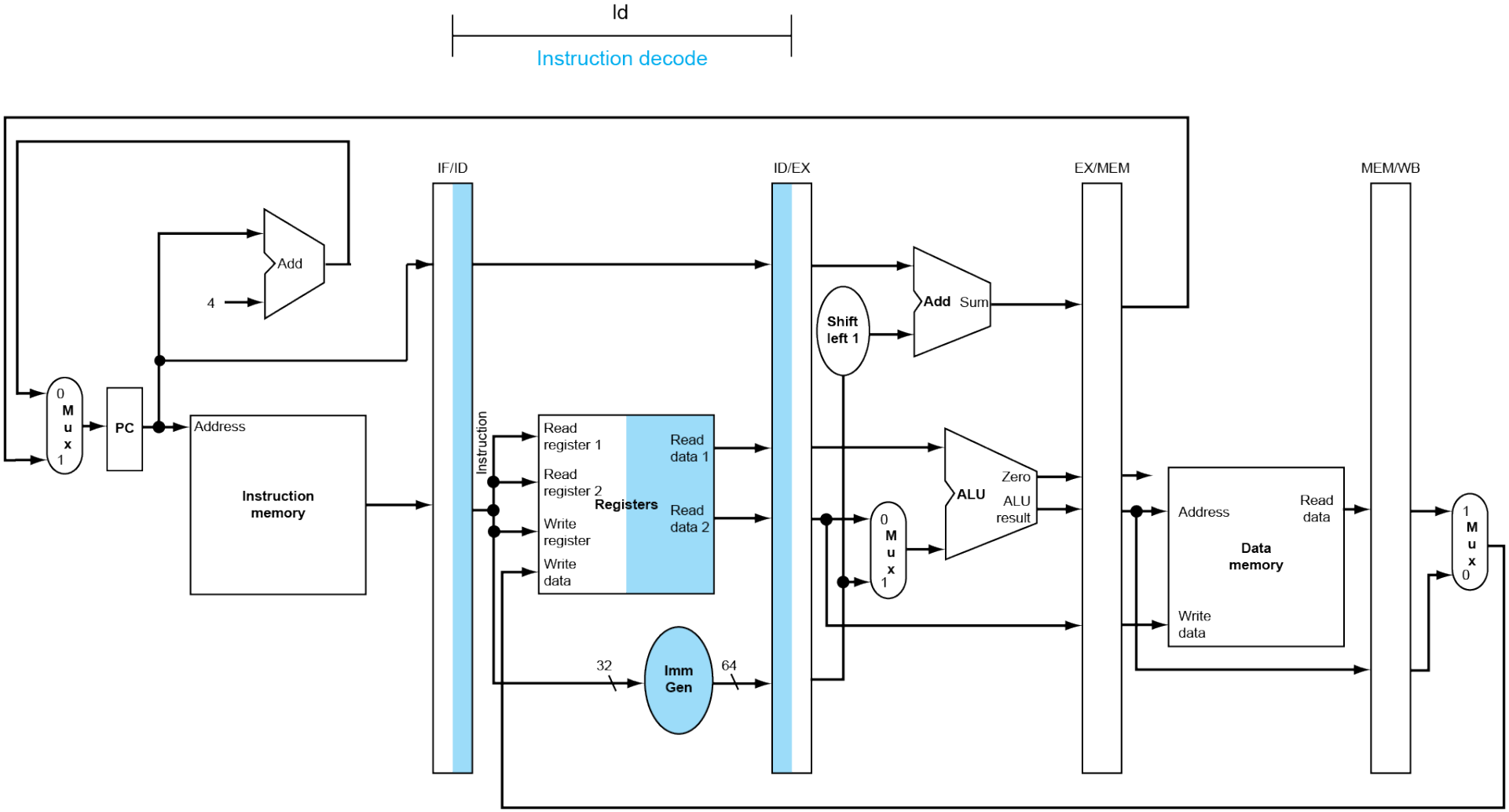
IF for Load, Store



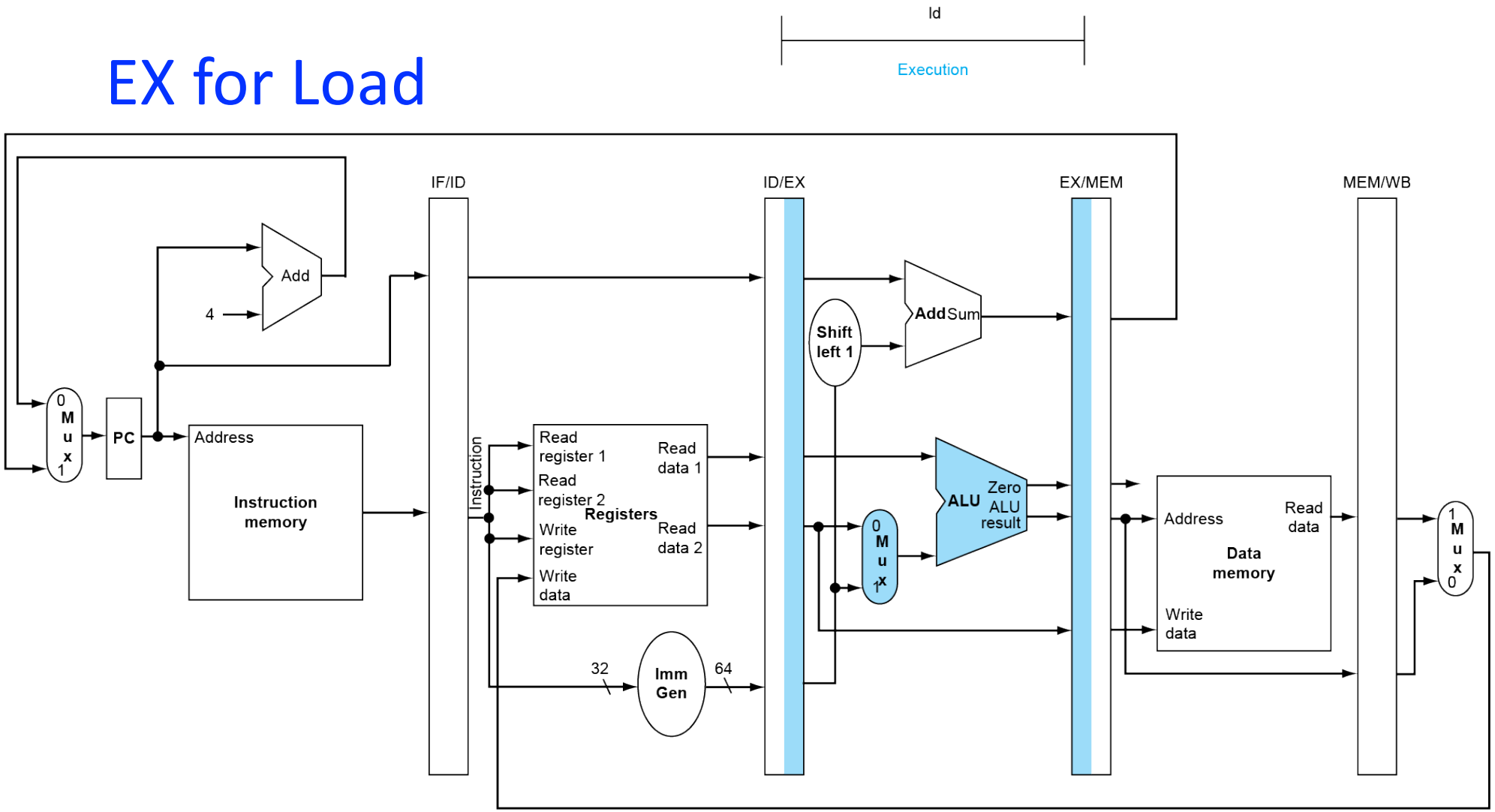
Single-clock-cycle diagram shows the state of an entire datapath during a single clock cycle



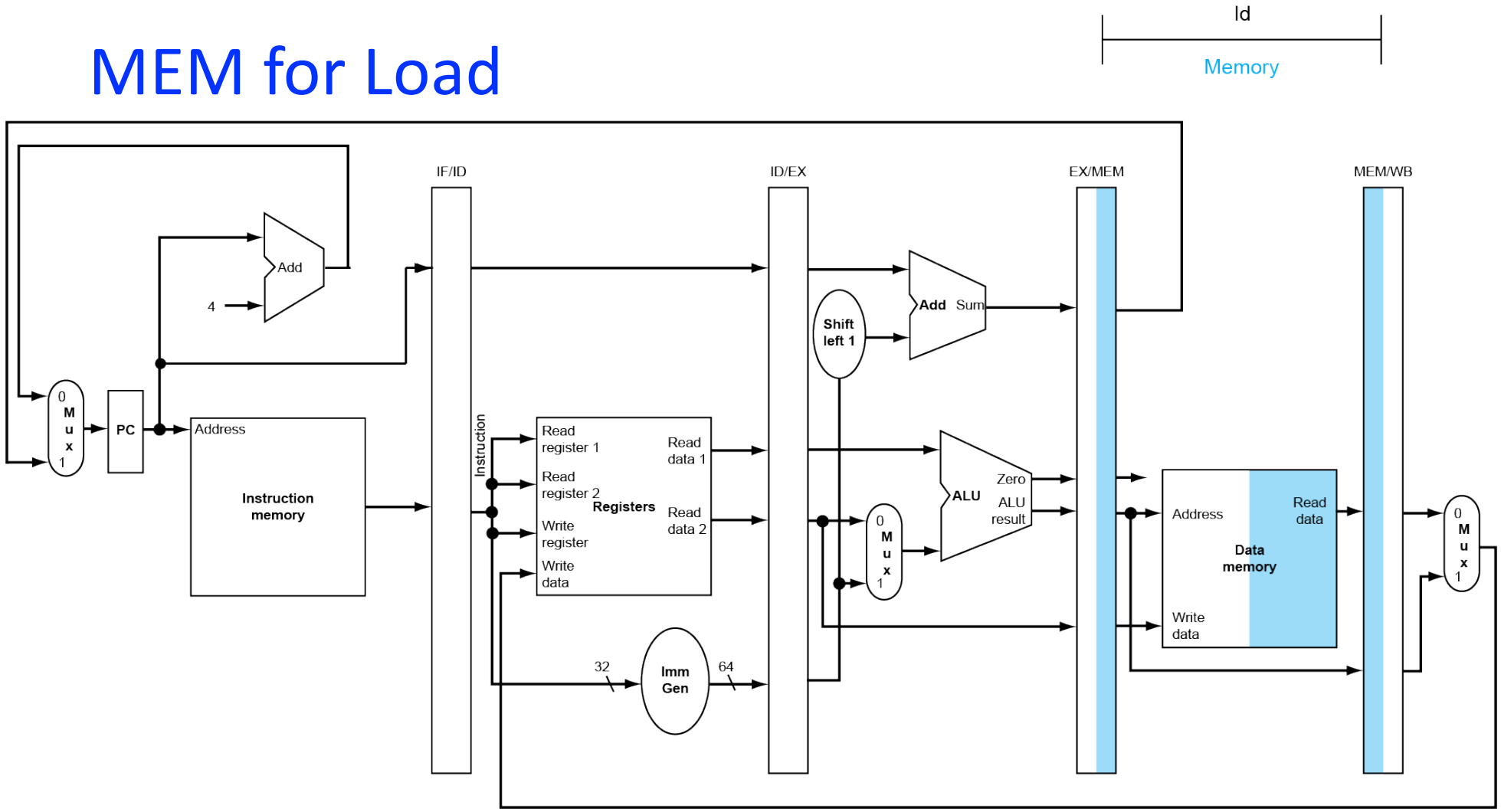
ID for Load, Store, …



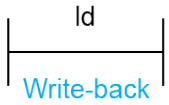
EX for Load

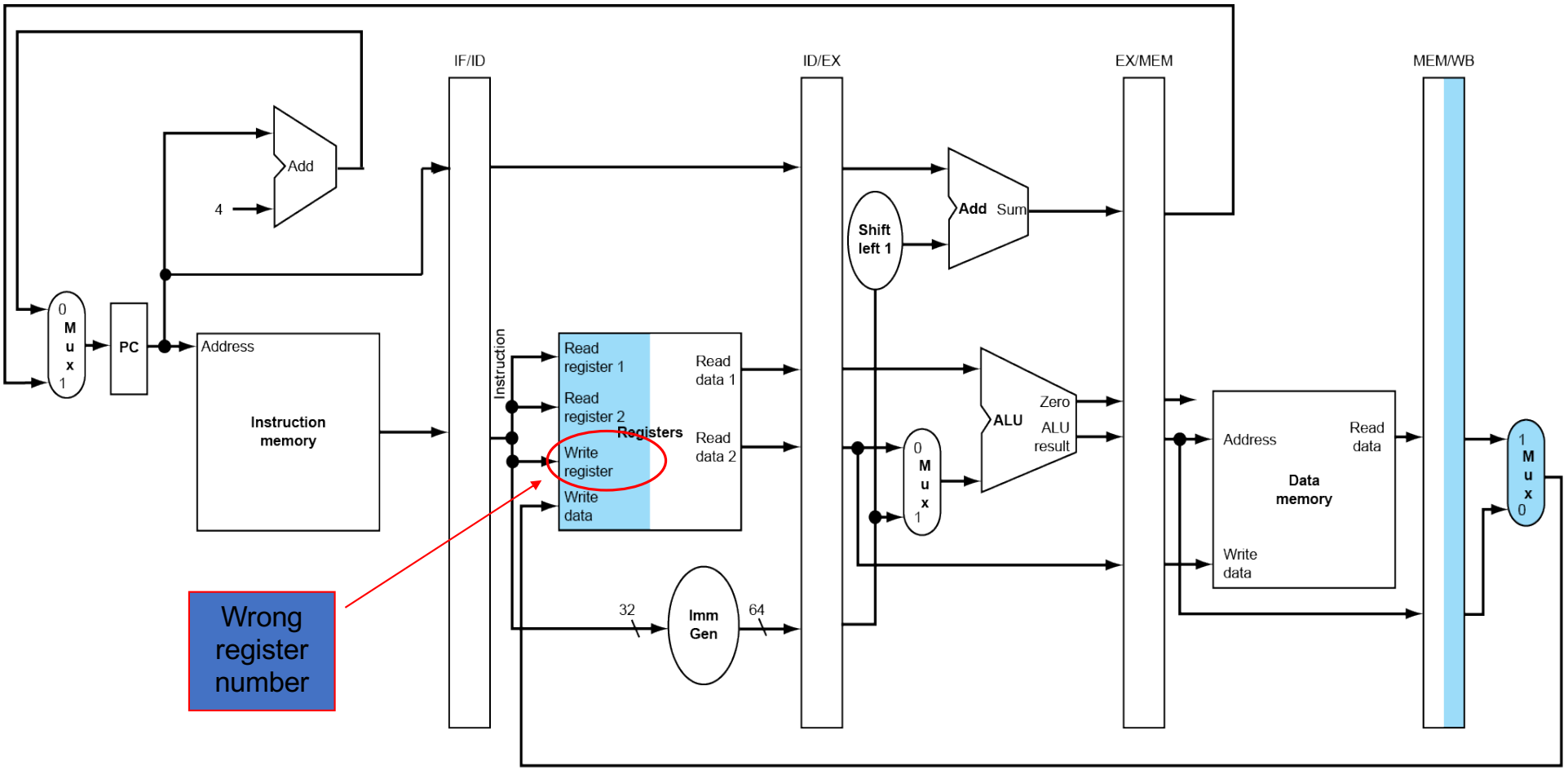

ld
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add
Sum
0
M
u
x
1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
1
M
u
x
0

MEM for Load

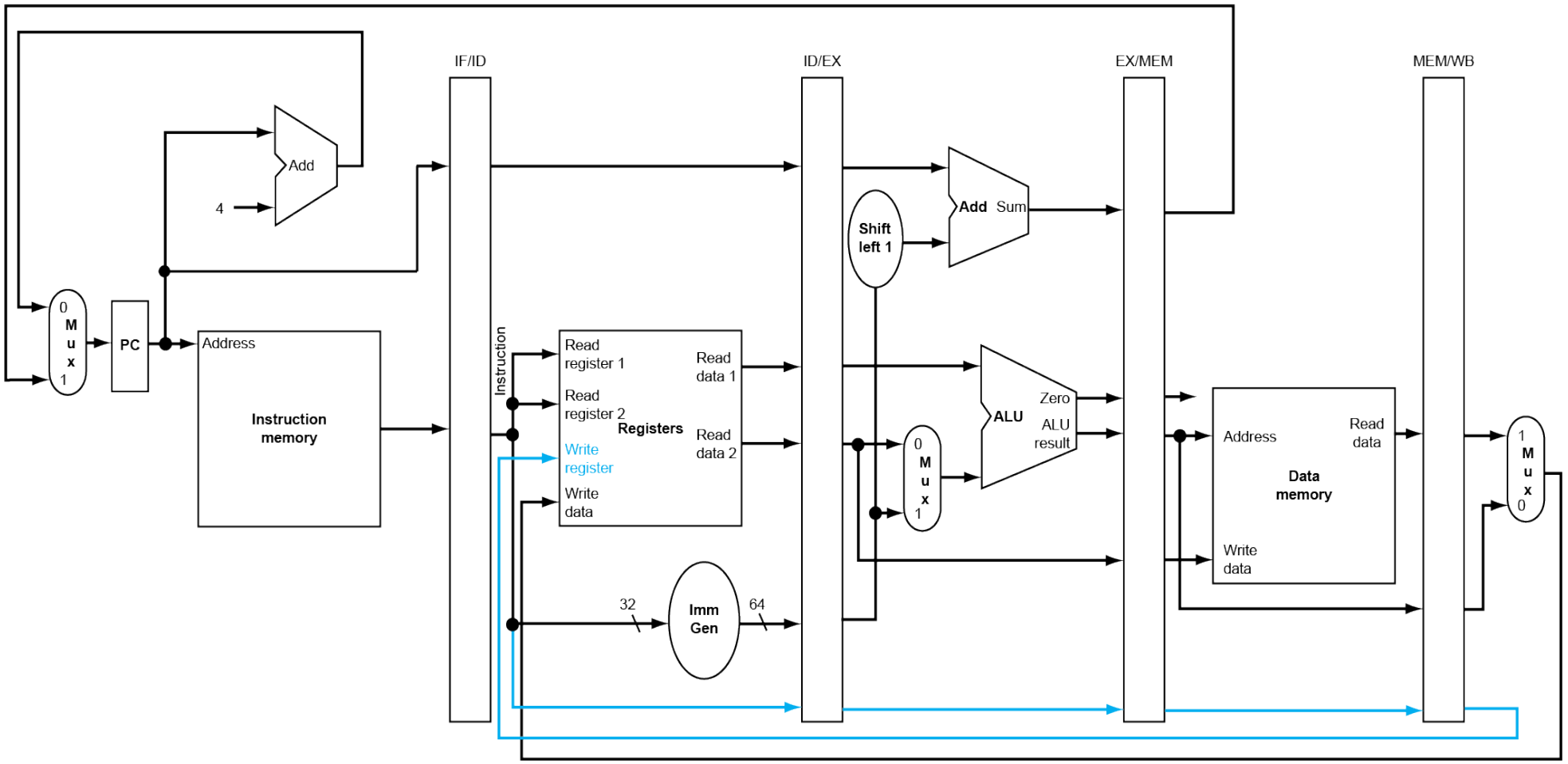

ld
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
32
Imm Gen
64
Shift left 1
Add
Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

WB for Load

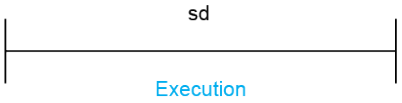

ld
Write-back

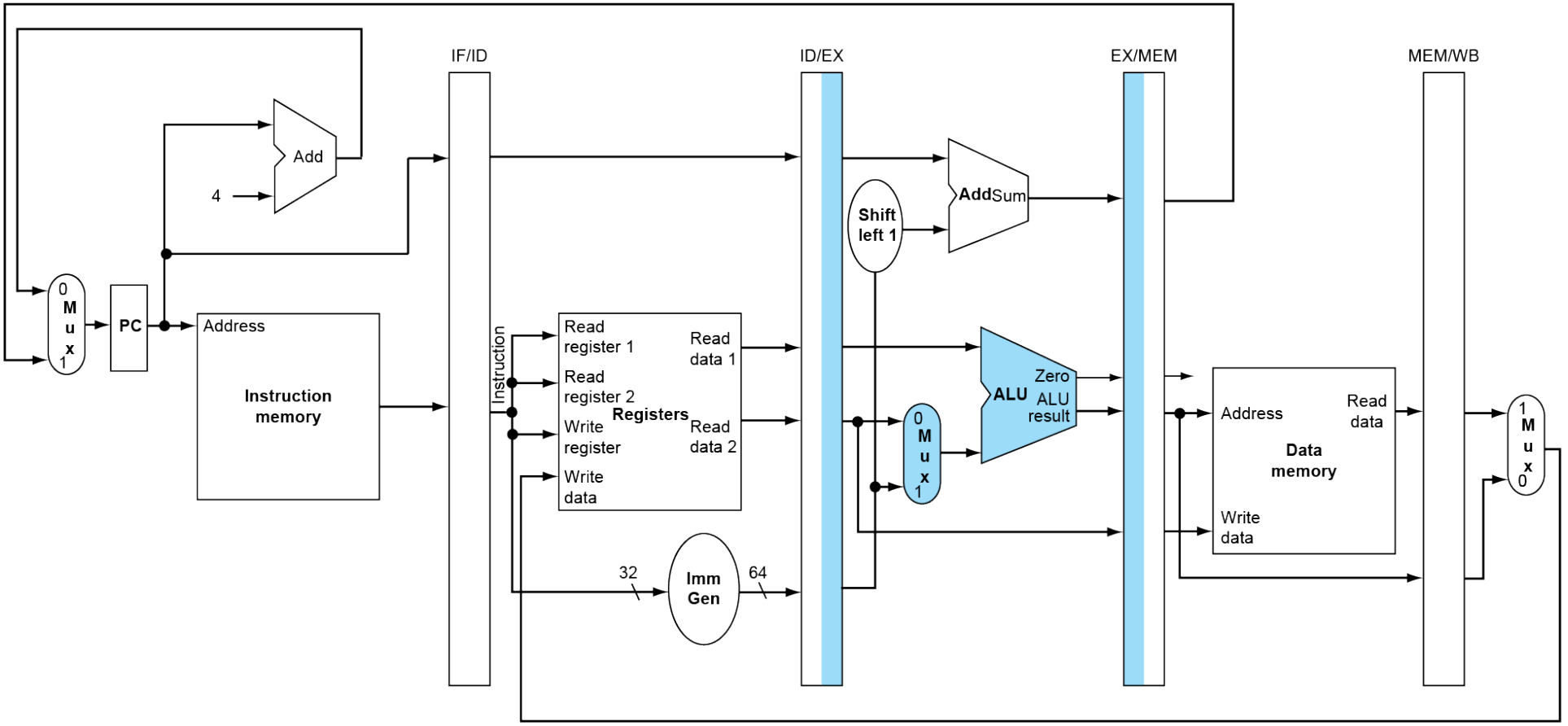

IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add
Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
32
Imm Gen
64
Wrong register number

Corrected Datapath for Load

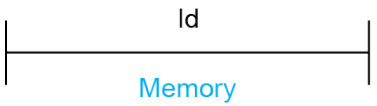


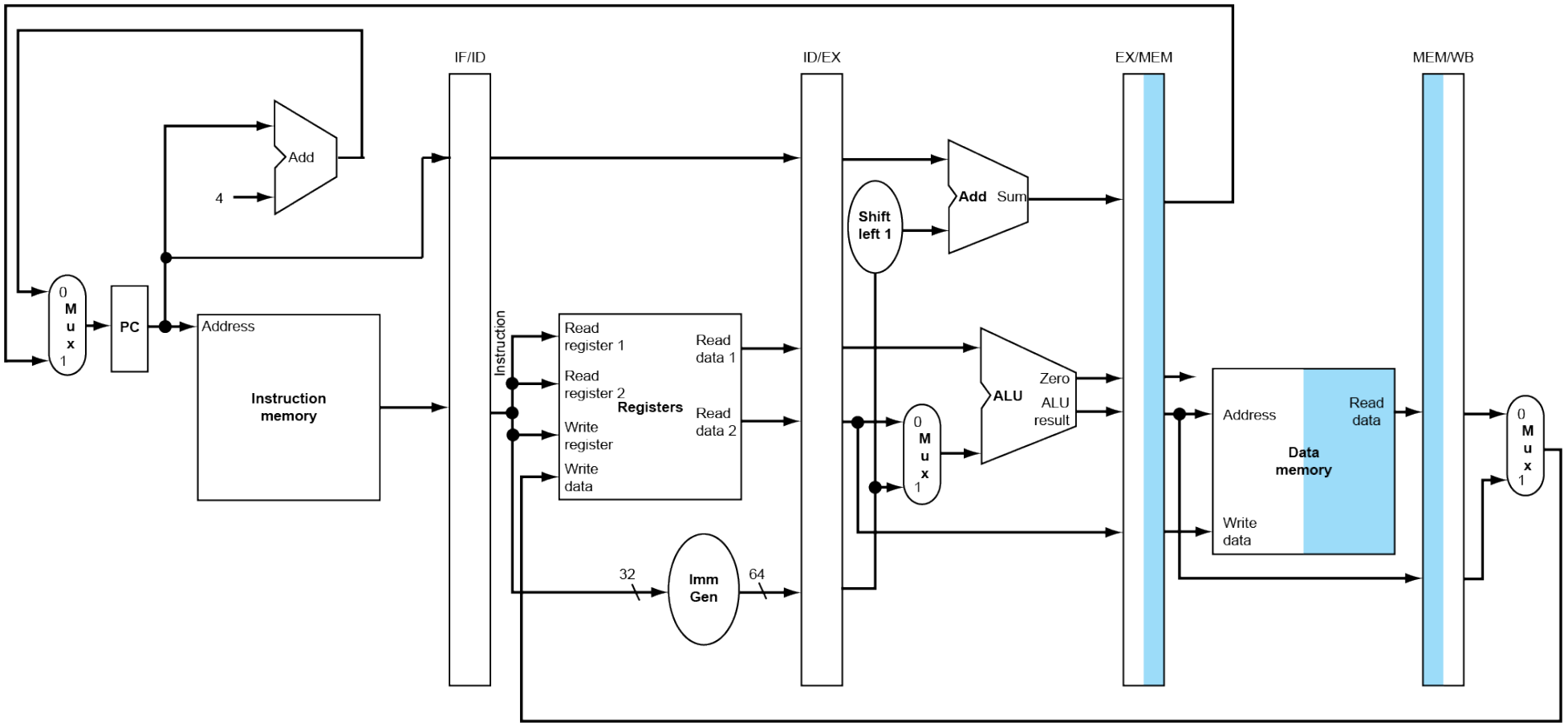
EX for Store


sd
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
32
Imm Gen
64
Shift left 1
AddSum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
M u x
1
0
M u x
0
1

MEM for Store

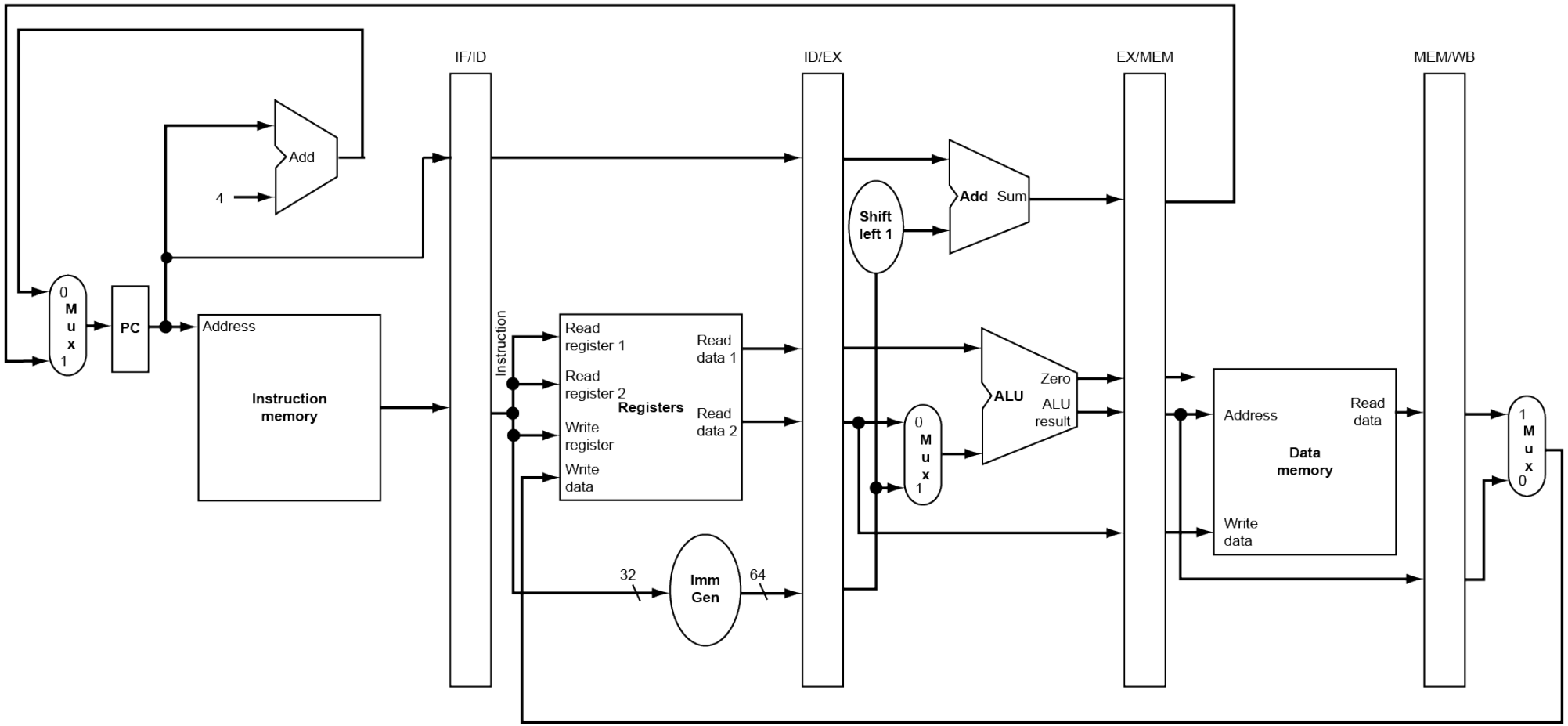




WB for Store

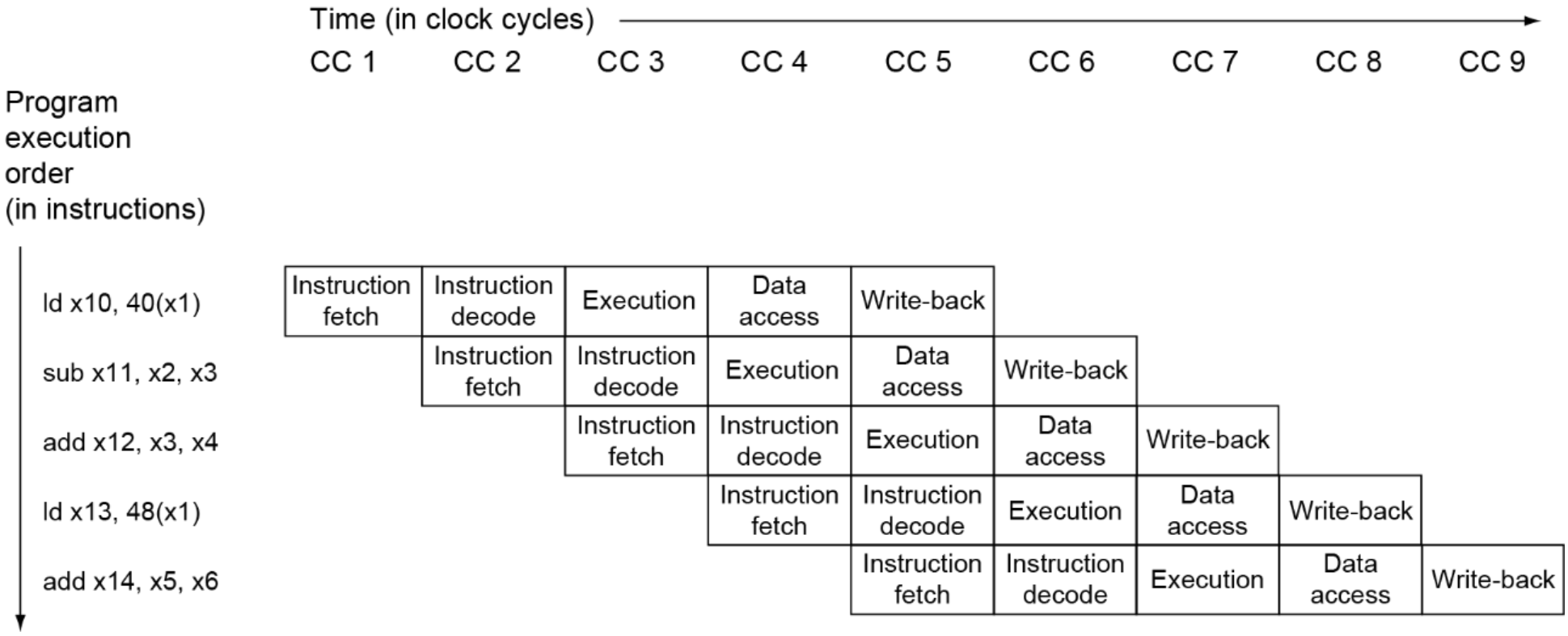
sd

Write-back



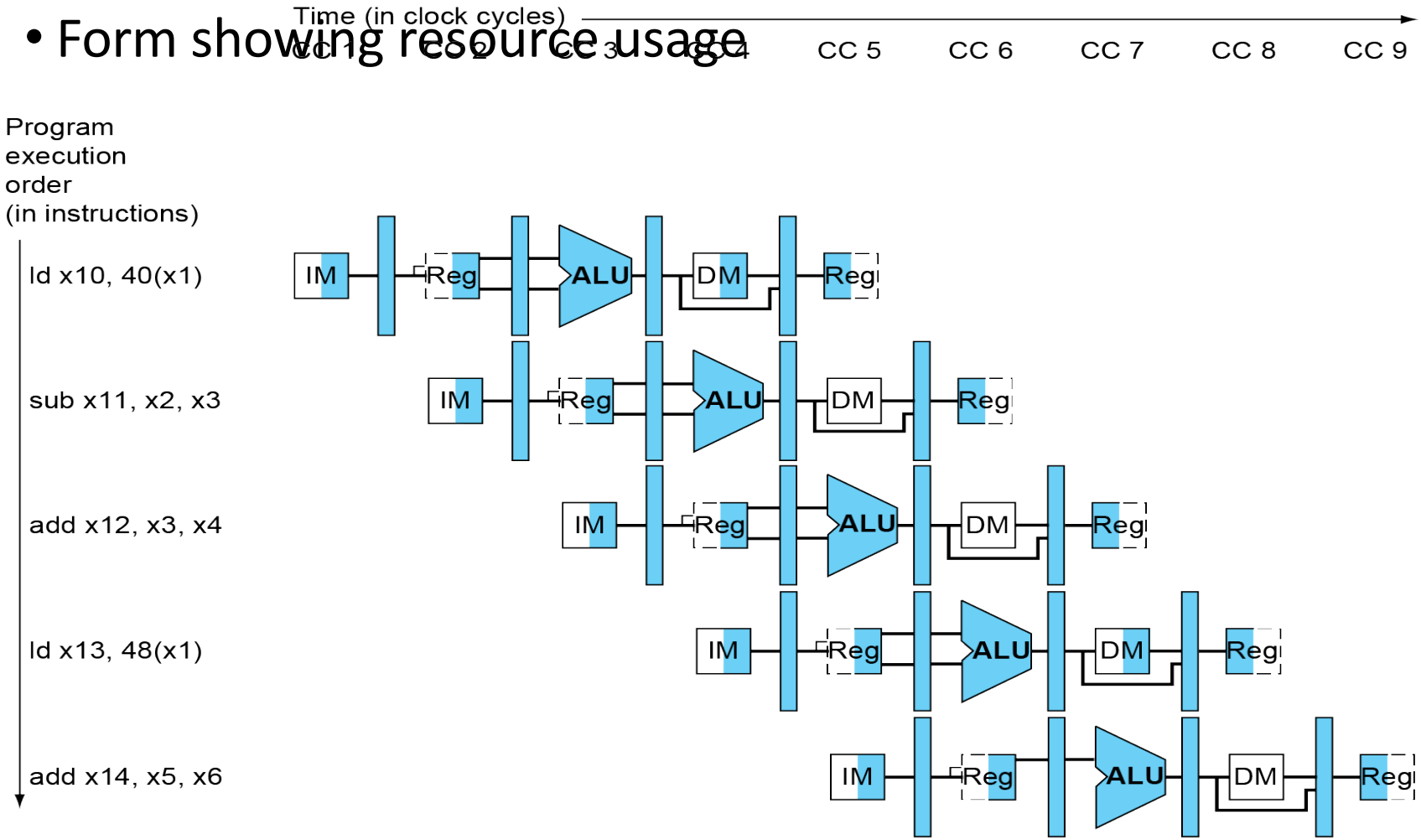
Multi-Cycle Pipeline Diagram

- Traditional form



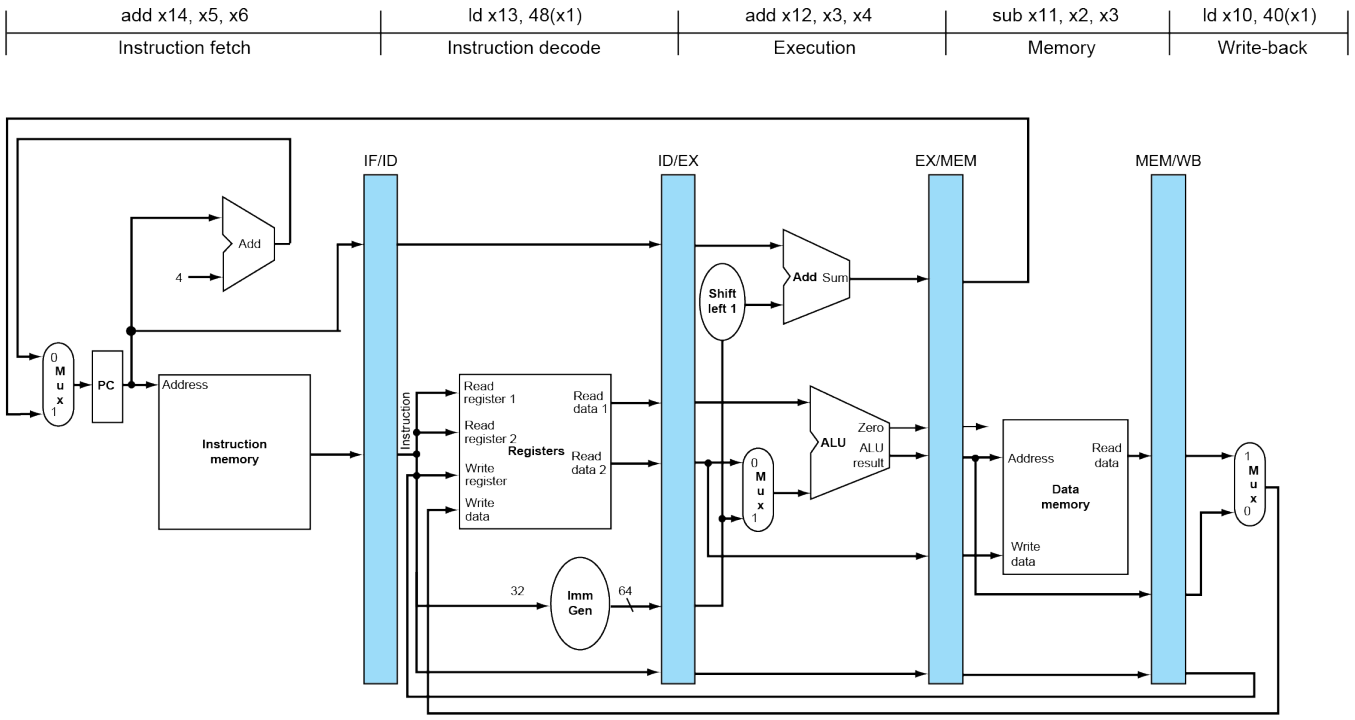
Multi-Cycle Pipeline Diagram

- Form showing resource usage

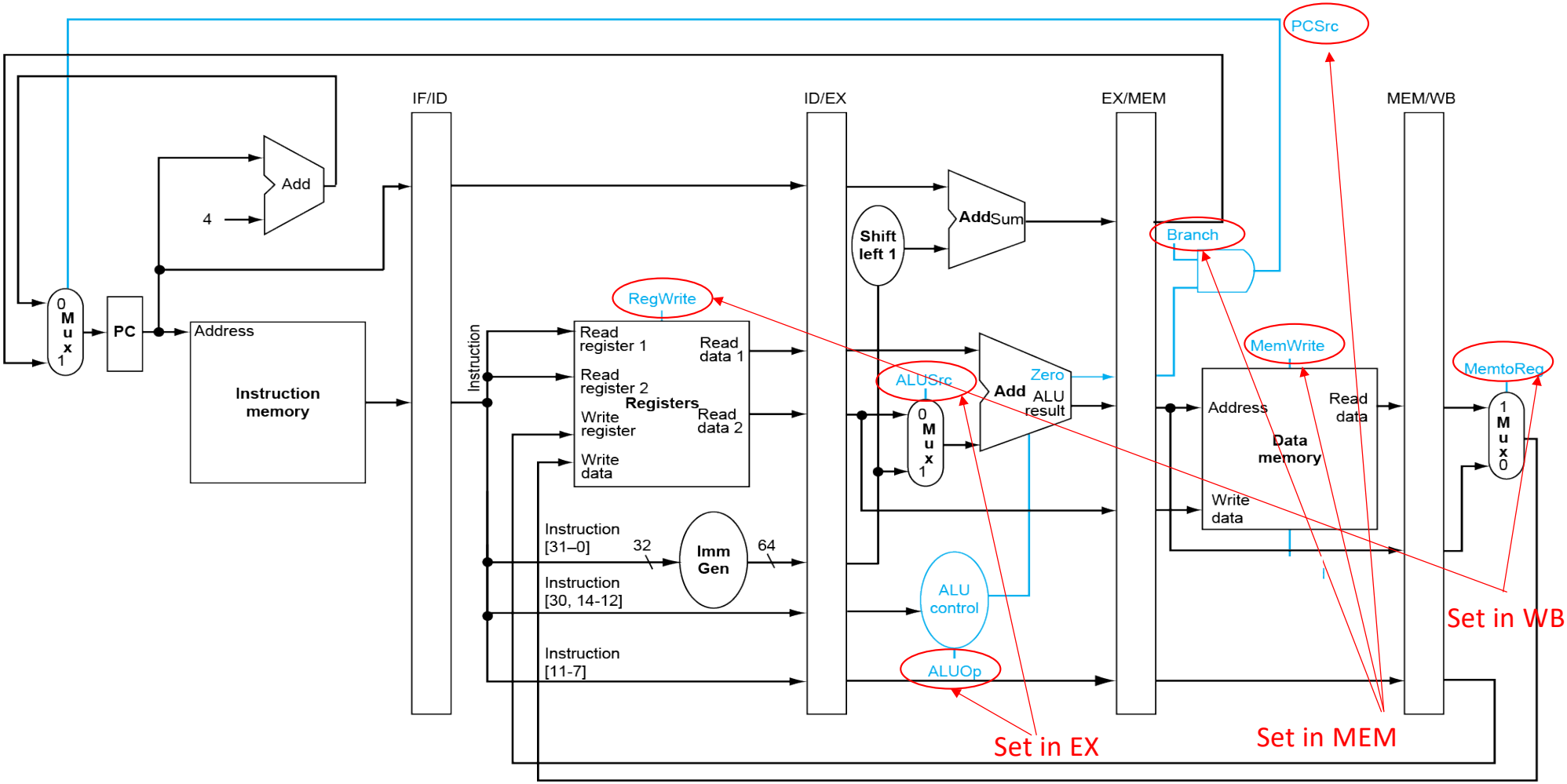


Single-Cycle Pipeline Diagram

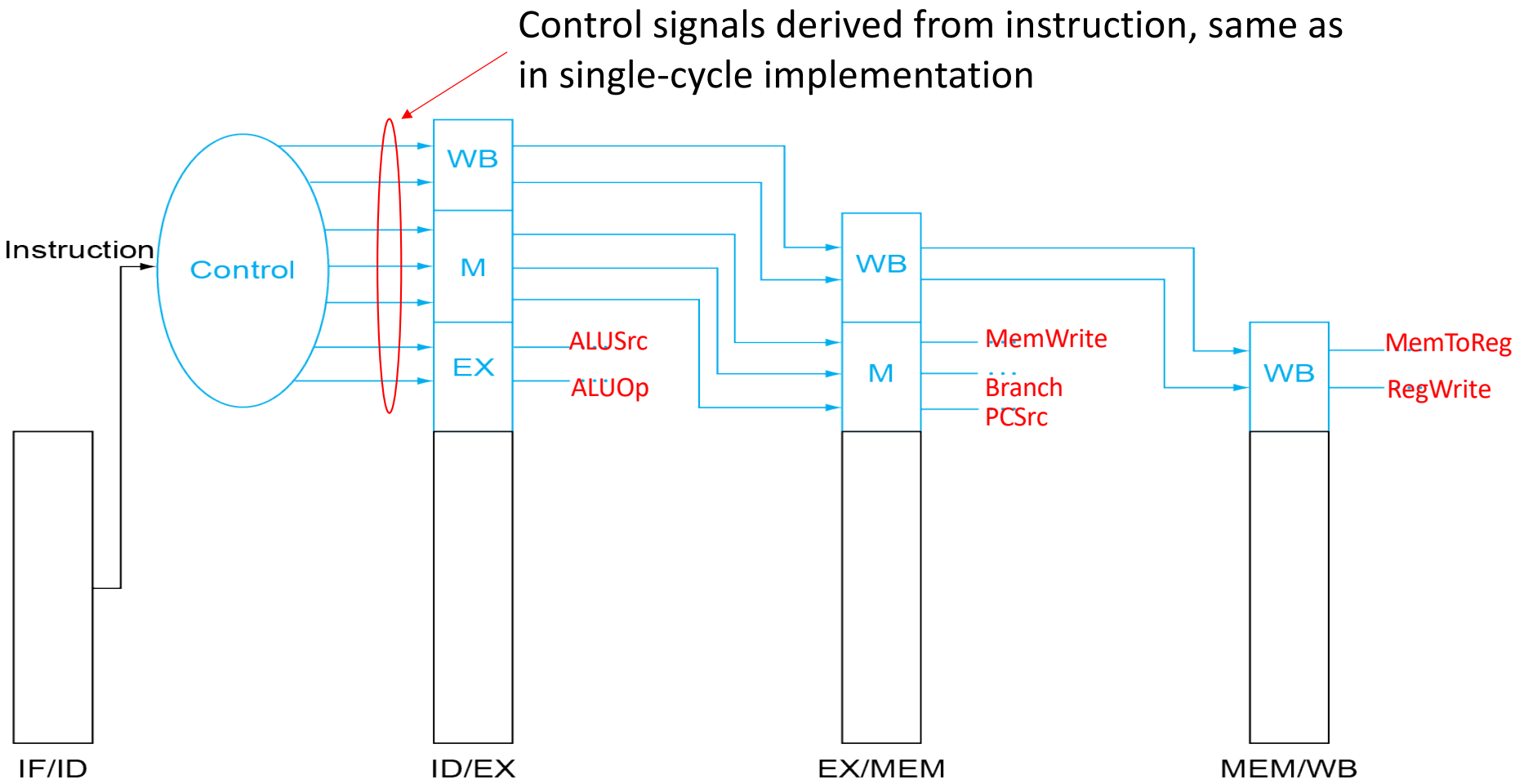
- State of pipeline in a given cycle



Pipelined Control (Simplified)



Pipelined Control



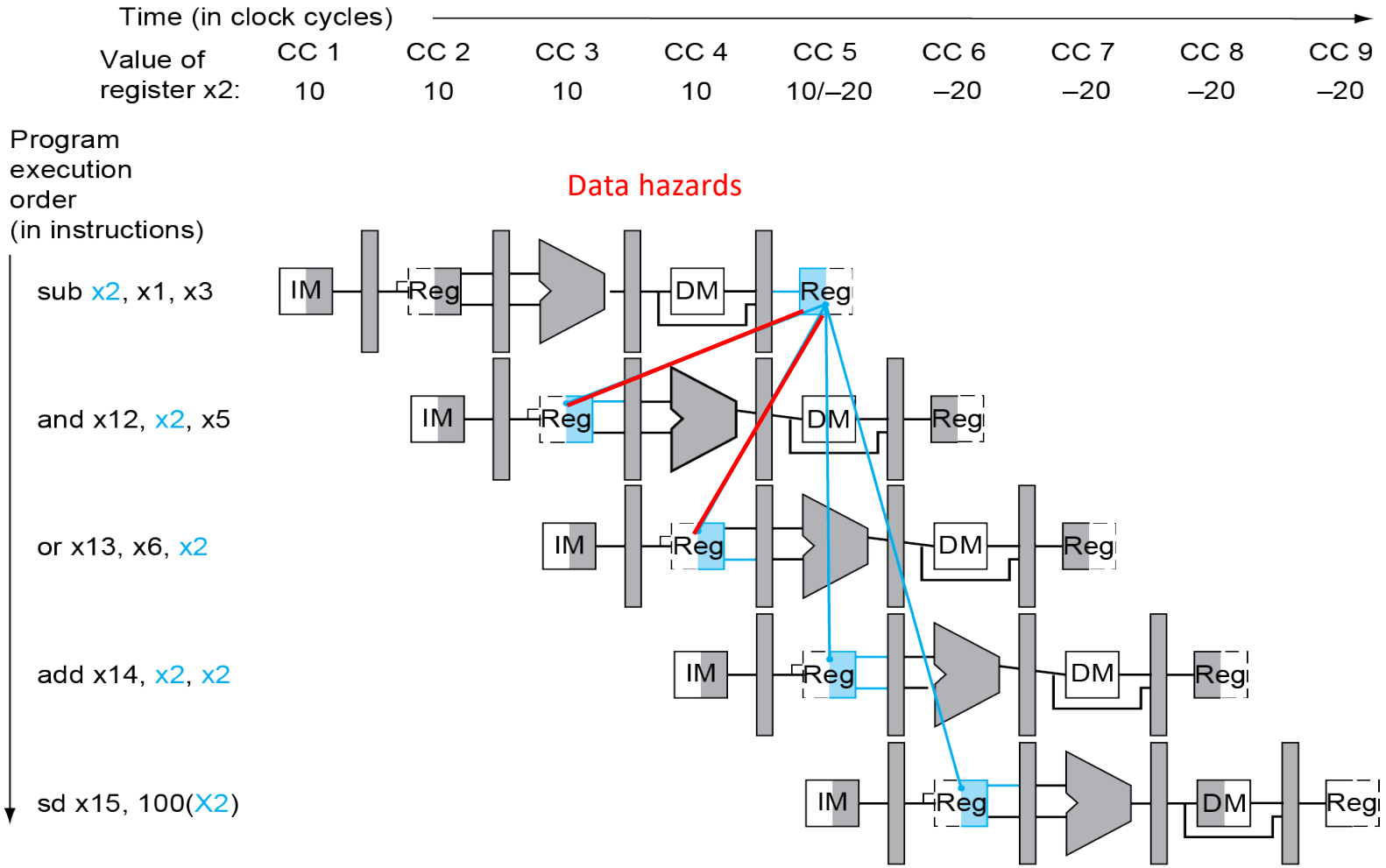
Data Hazards in ALU Instructions

- Example instruction sequence:

```
sub   x2, x1,x3
and   x12,x2,x5
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

- Solution: forwarding (aka bypassing)
  - Use result when it's computed; don't wait for it to be stored in register

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1: register number for Rs1 in ID/EX pipeline register
- ALU operand register numbers (in EX stage) are:
  - ID/EX.RegisterRs1
  - ID/EX.RegisterRs2
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

EX-hazard: Fwd from EX/MEM pipeline reg

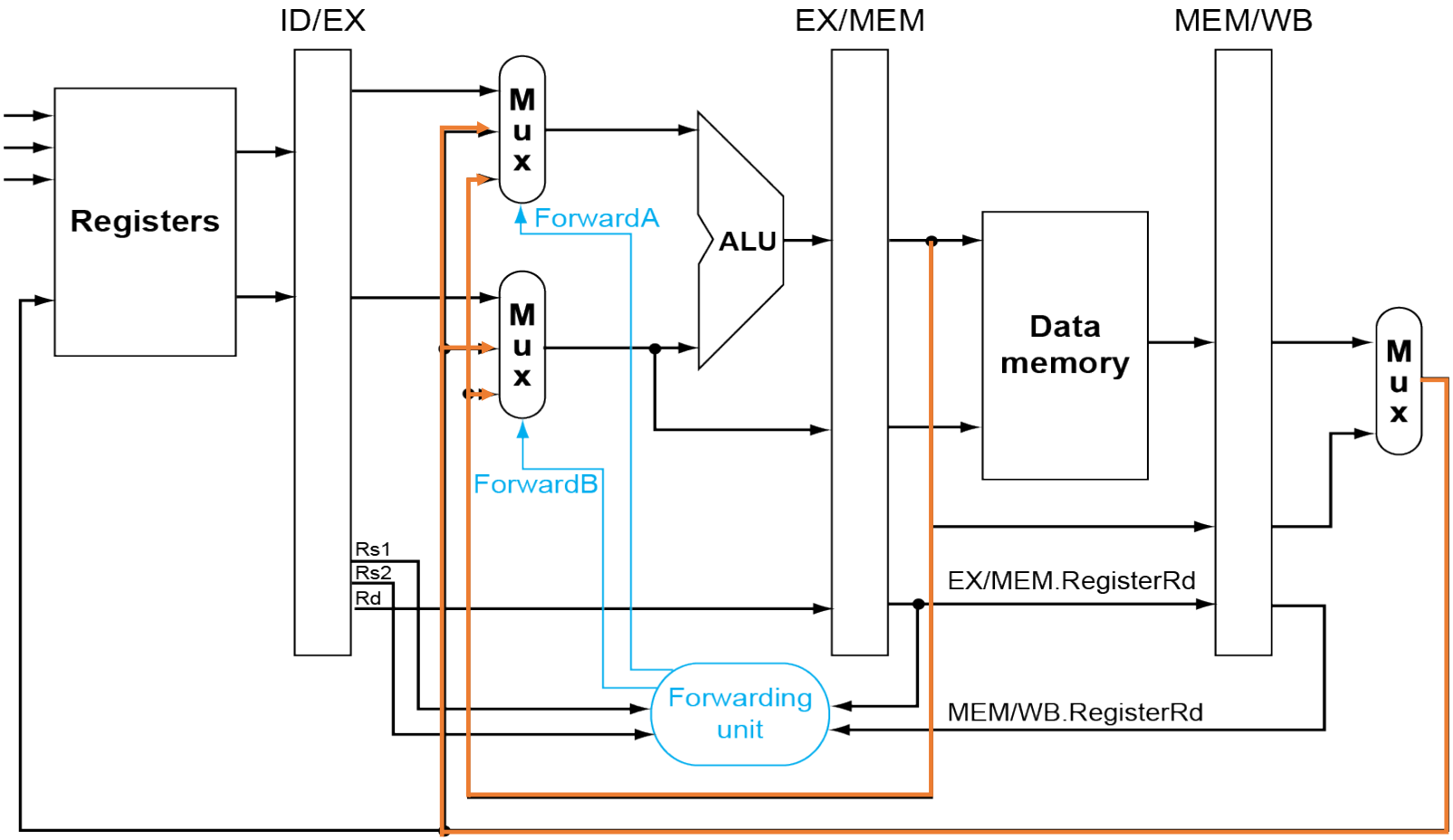
Mem-hazard: Fwd from MEM/WB pipeline reg

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 x0
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0

Forwarding Paths

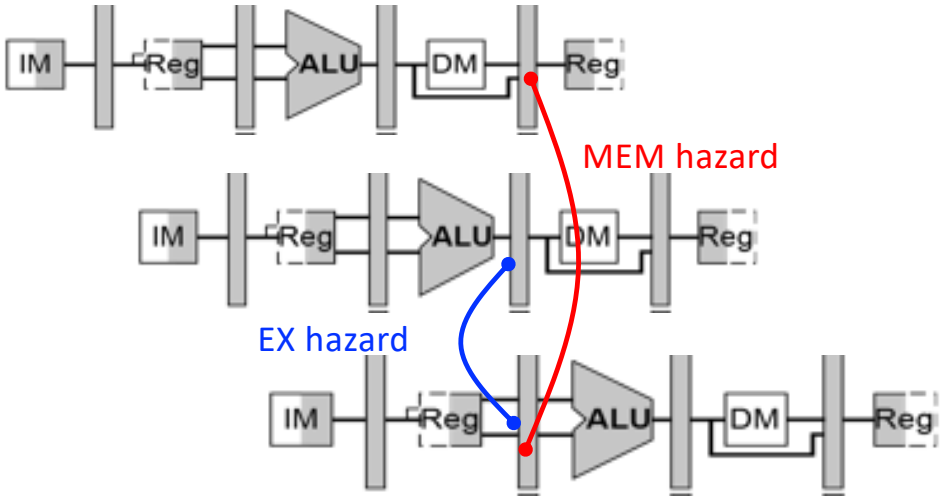


Double Data Hazard

- Consider the sequence:

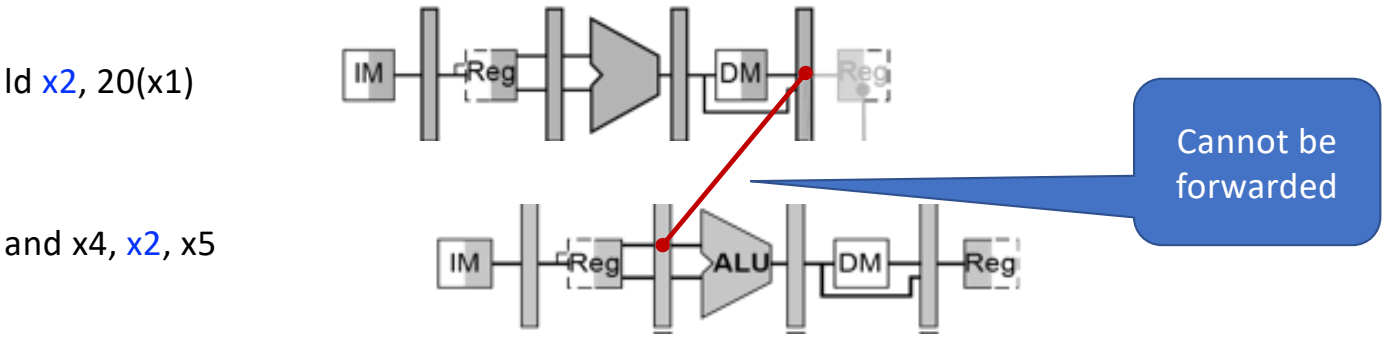
```
add  x1,x1,x2
add  x1,x1,x3
add  x1,x1,x4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true



Load-Use Hazard Detection

- Load-use hazard cannot be resolved using forwarding alone

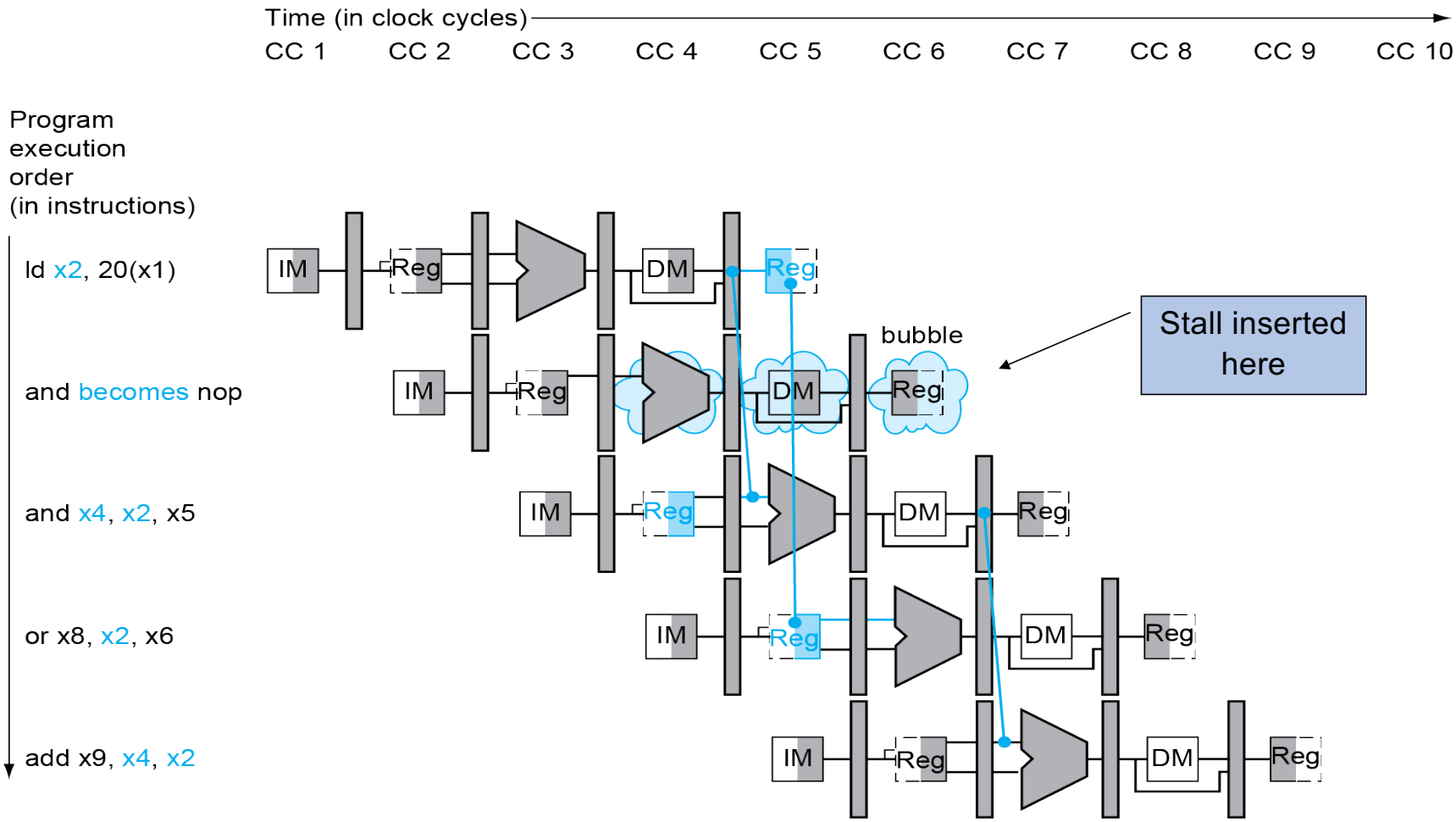


- Check load-use hazard using condition
  - ID/EX.MemRead and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs1))
- If detected, stall and insert bubble

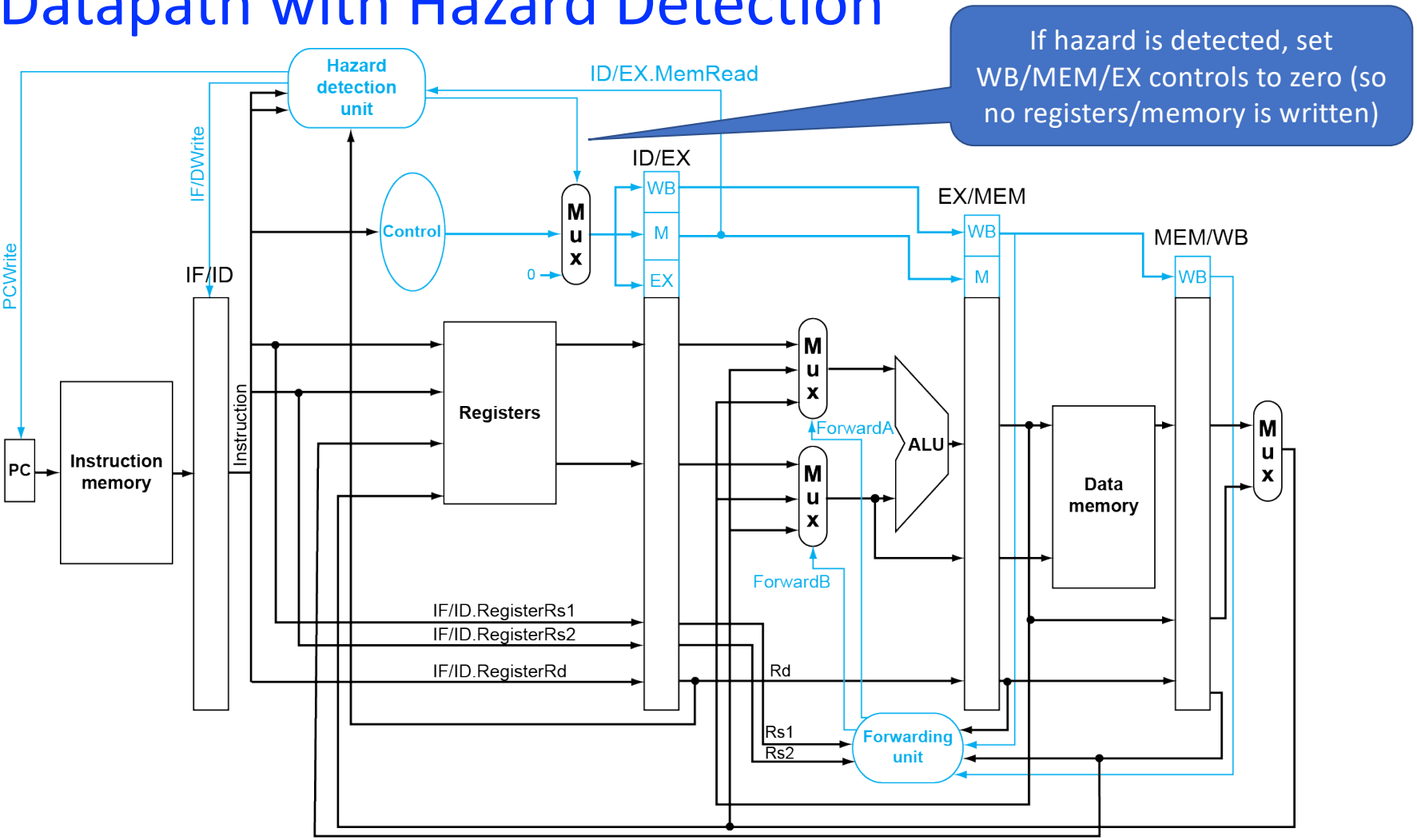
How to Stall the Pipeline

- 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 ld
    - Can subsequently forward to EX stage

Load-Use Data Hazard



Datapath with Hazard Detection



Summary

- Pipeline increases throughput by overlapping execution of multiple instructions
- Pipeline hazard
  - Structure (solution: add resources)
  - Data (solution: forwarding)
  - Control (next class)
- Pipeline stalls