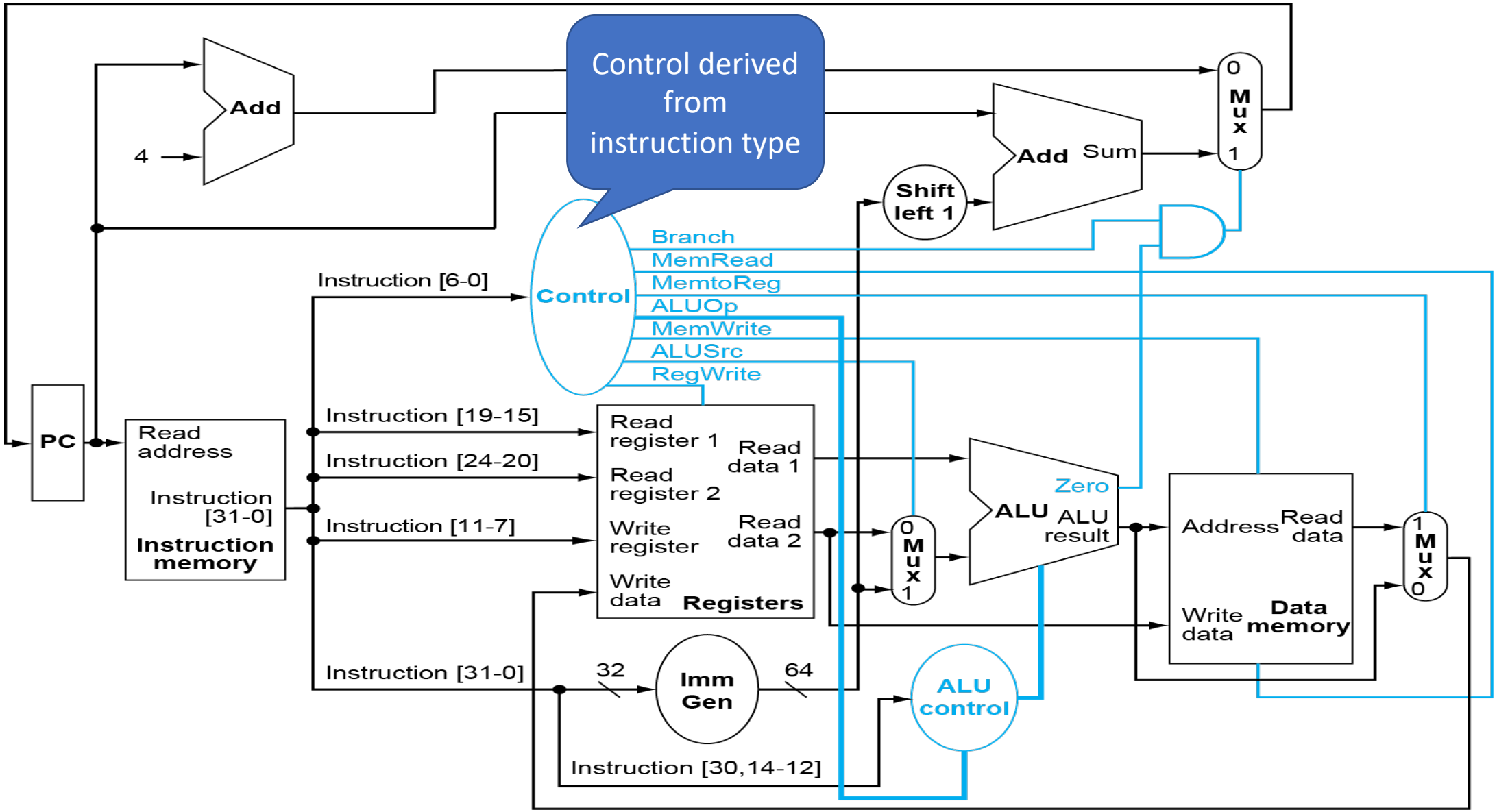
Basic/Pipelined processor Implementation

Jinyang Li

Today’s lesson plan

- Basic single-cycle CPU design, continued
- Pipelining idea and challenges

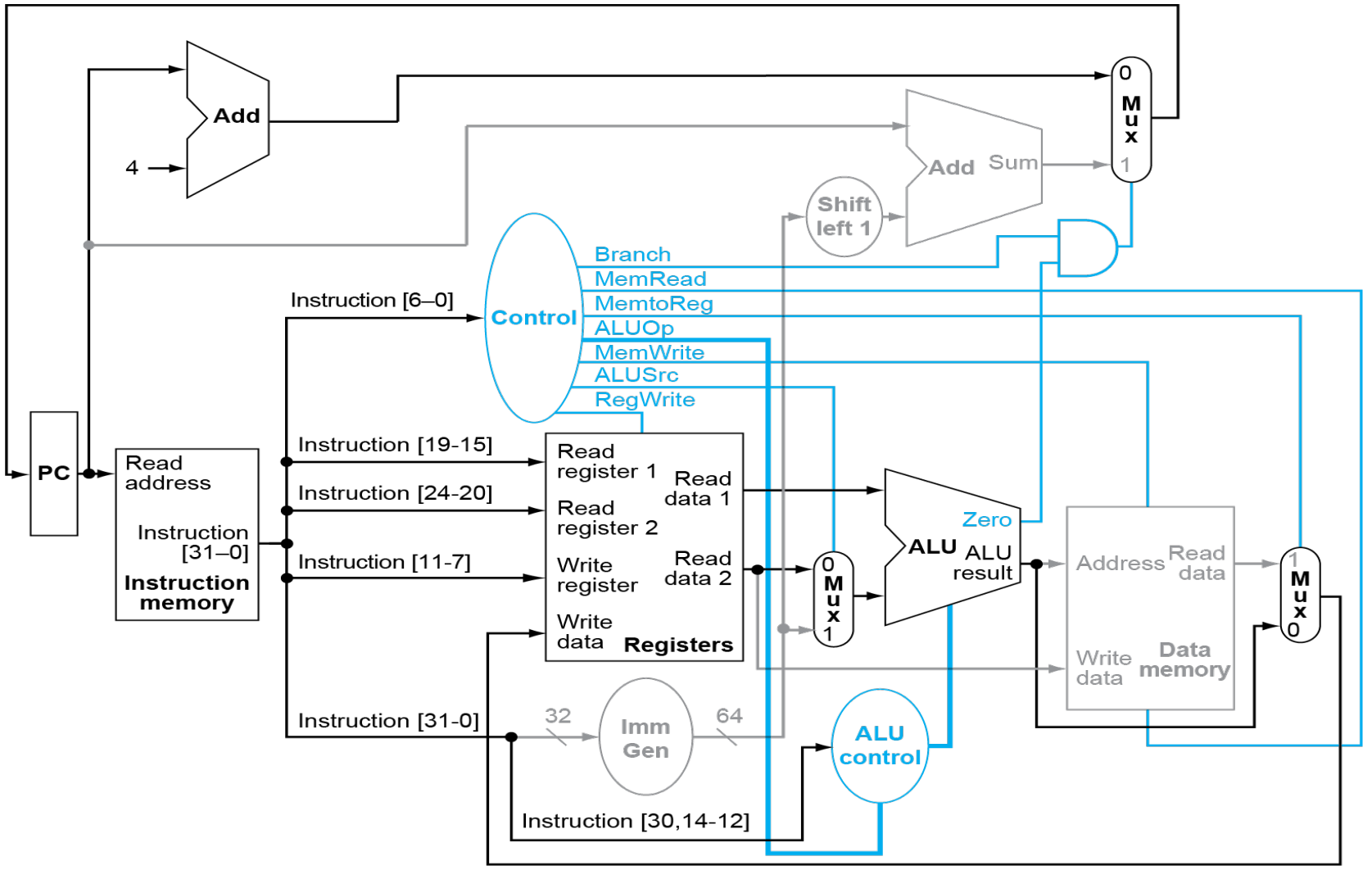
Recall our basic RISC-V CPU: datapath w/ control



```
add x5, x6, x7
```

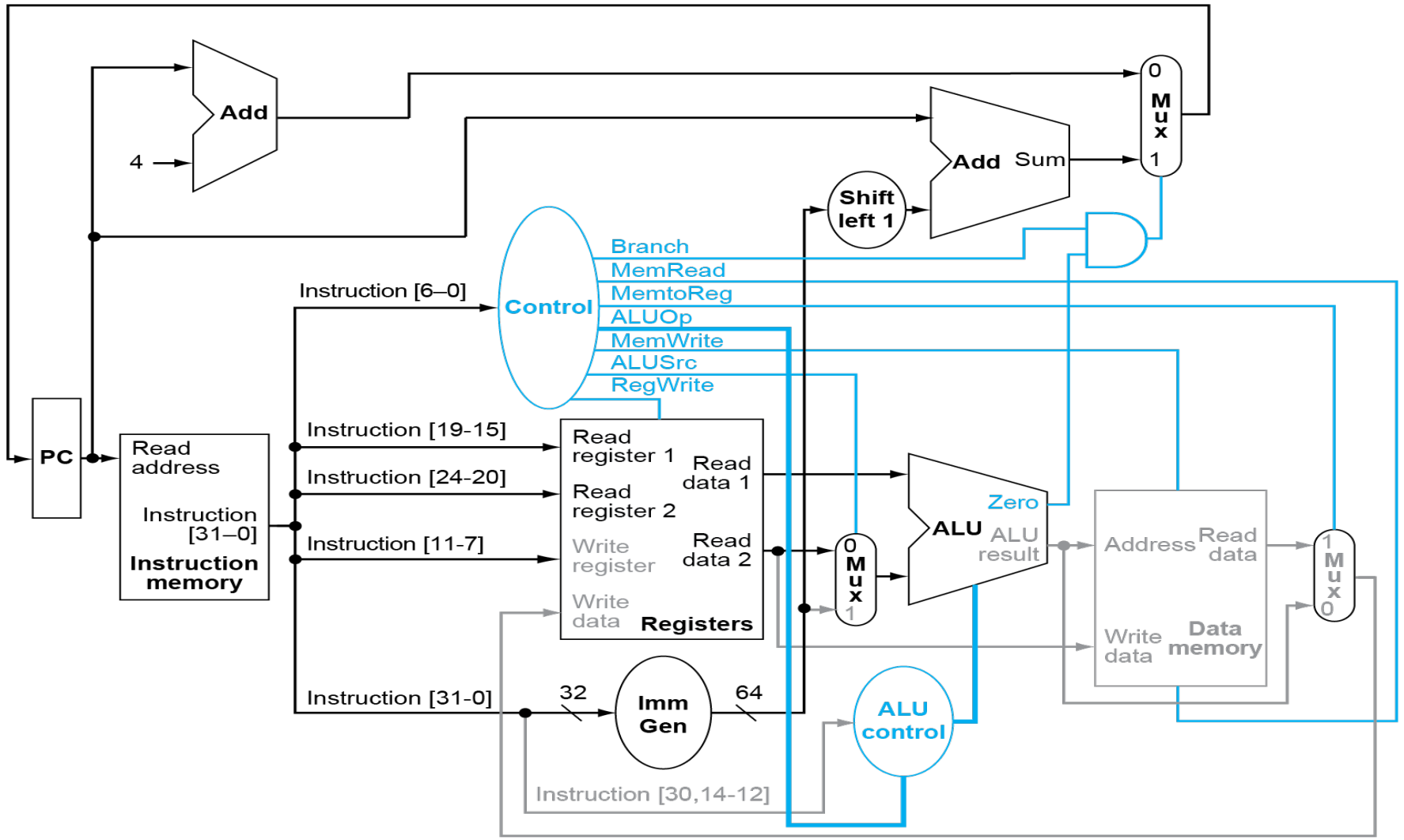
R-Type Inst

funct7	rs2	rs1	funct3	rd	opcode



BEQ Instruction

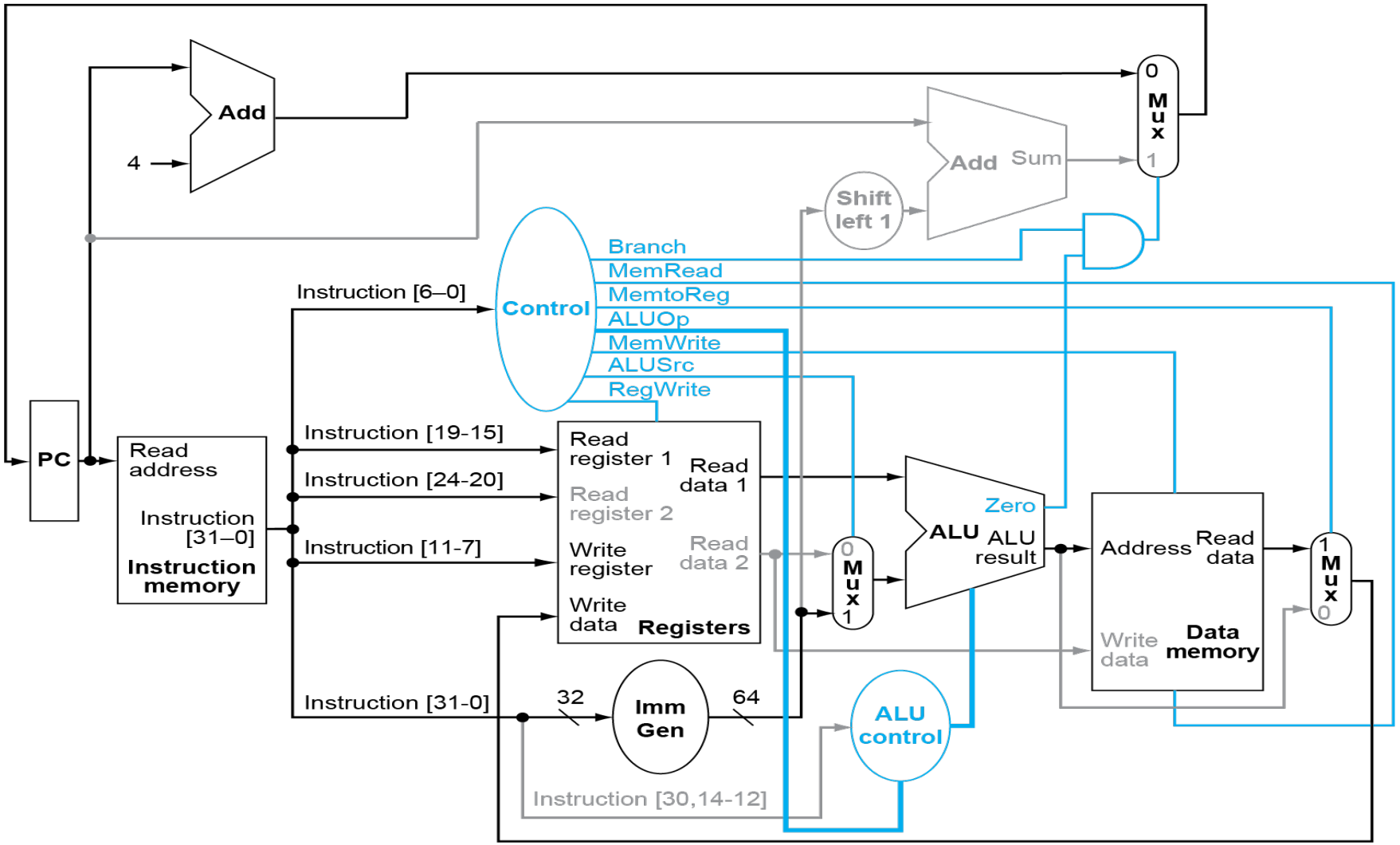
```
beq x5, x6, 100
```



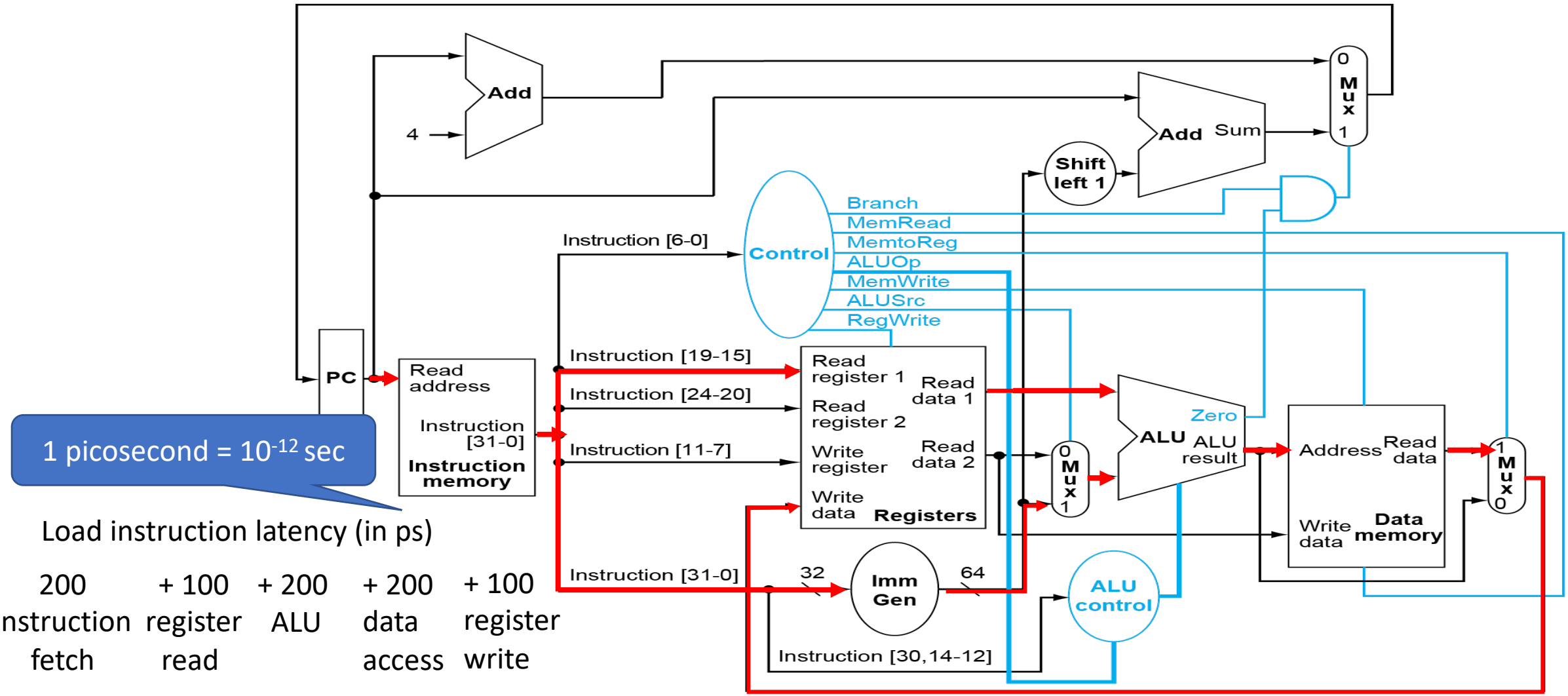
```
ld x5, 40(x6)
```

Load Inst

immediate	rs1	funct3	rd	opcode



Basic CPU must finish an instruction in one clock cycle → use a "slow" clock



1 picosecond = 10^{-12} sec

Load instruction latency (in ps)

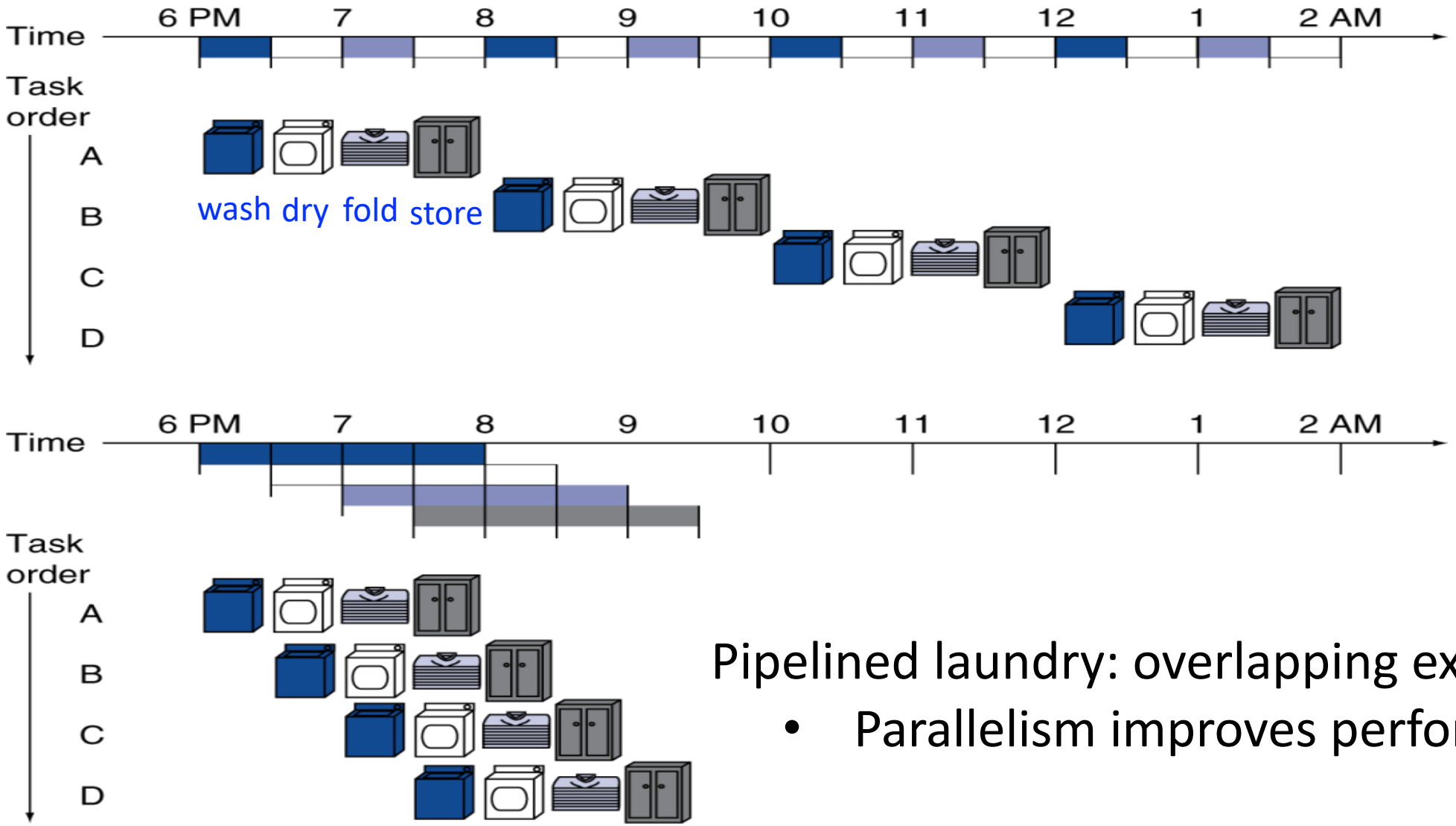
200	+ 100	+ 200	+ 200	+ 100
instruction fetch	register read	ALU	data access	register write

Clock cycle >= 800 ps

Our basic design is slow

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary clock period for different instructions
- Next: improve performance by pipelining

Pipelining: a laundry analogy



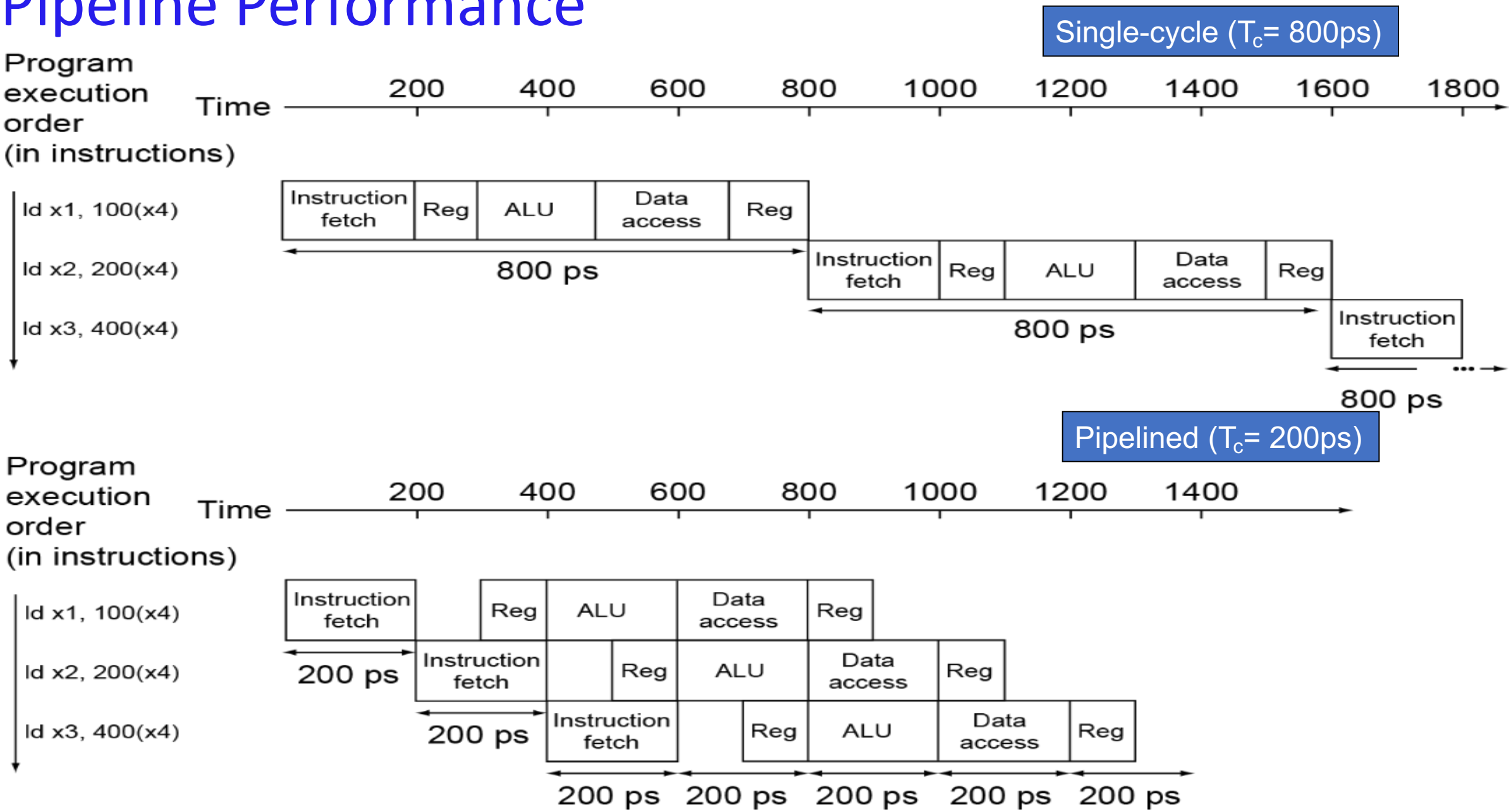
Pipelined laundry: overlapping execution

- Parallelism improves performance

RISC-V Pipeline

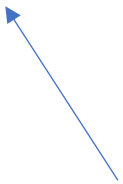
- Five stages:
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

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 is 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

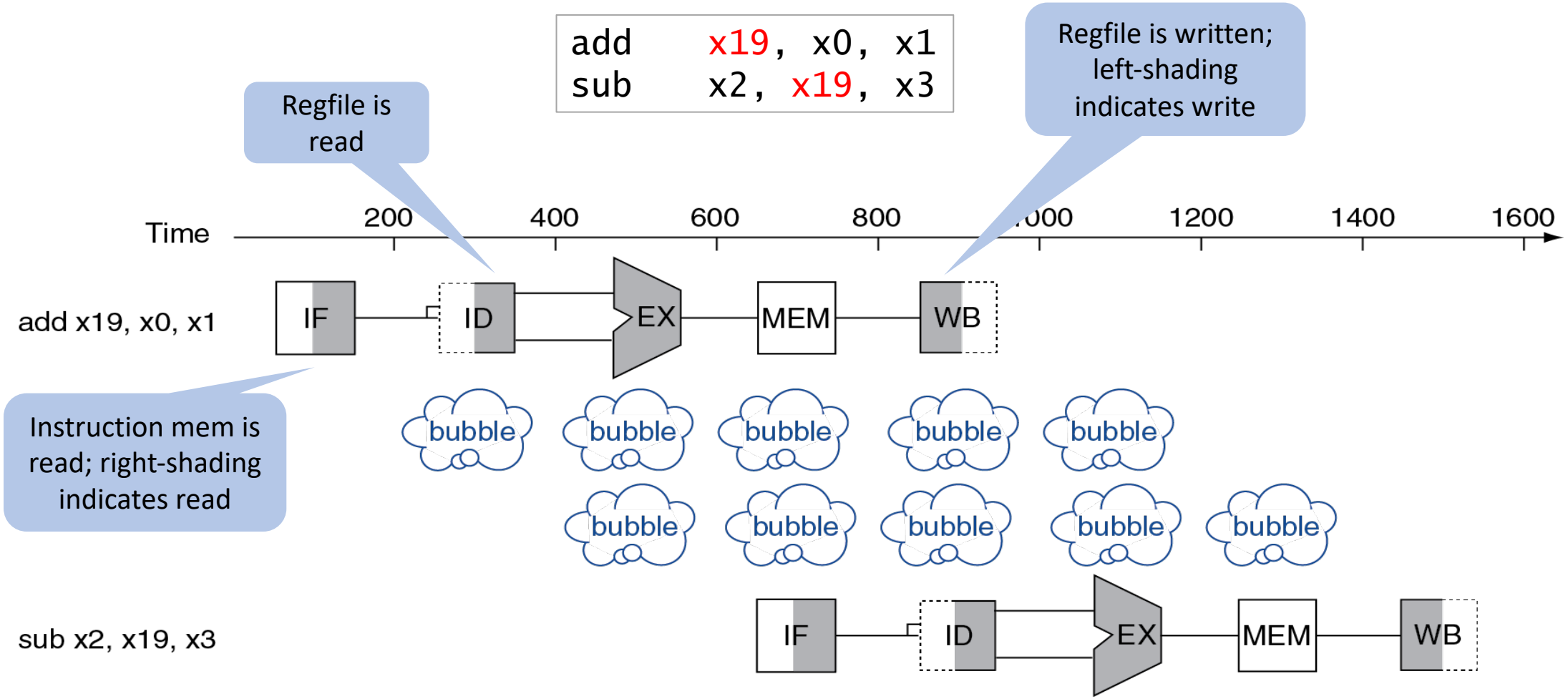
- Situations that prevent starting next instruction in the next cycle
- Structure hazard
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its write
- Control hazard
  - Which instruction to execute depends on previous instruction

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

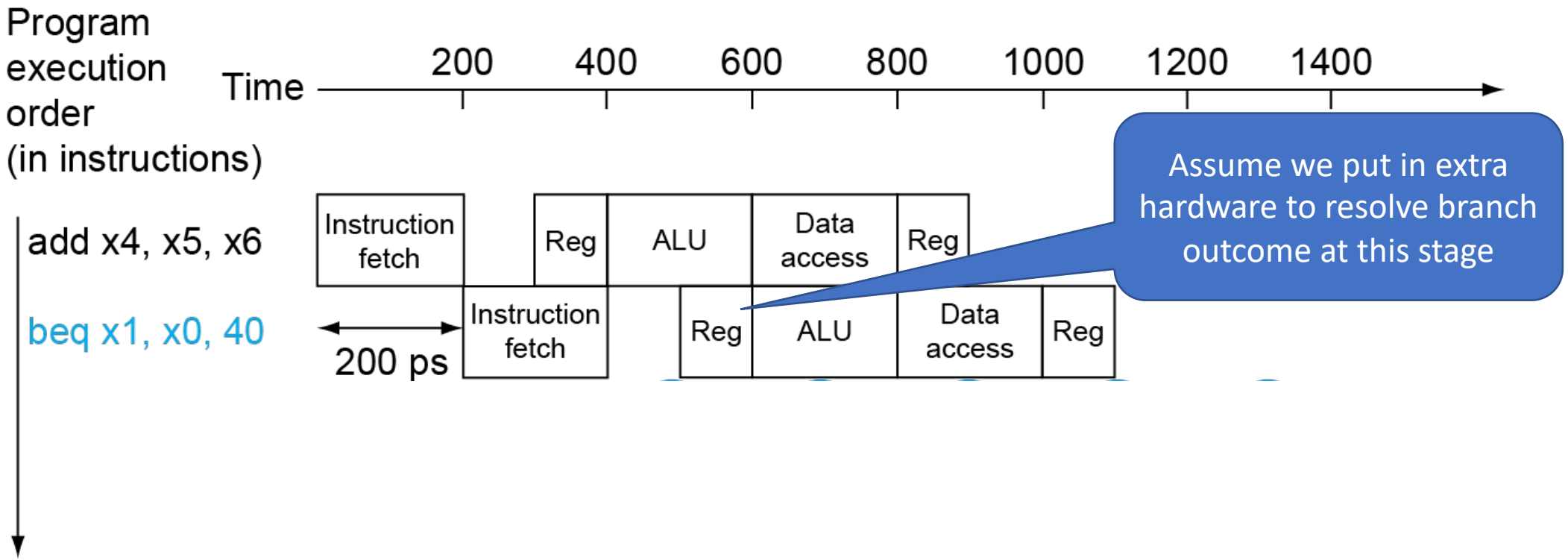
Data Hazards

- An instruction depends on the previous instruction to complete its write



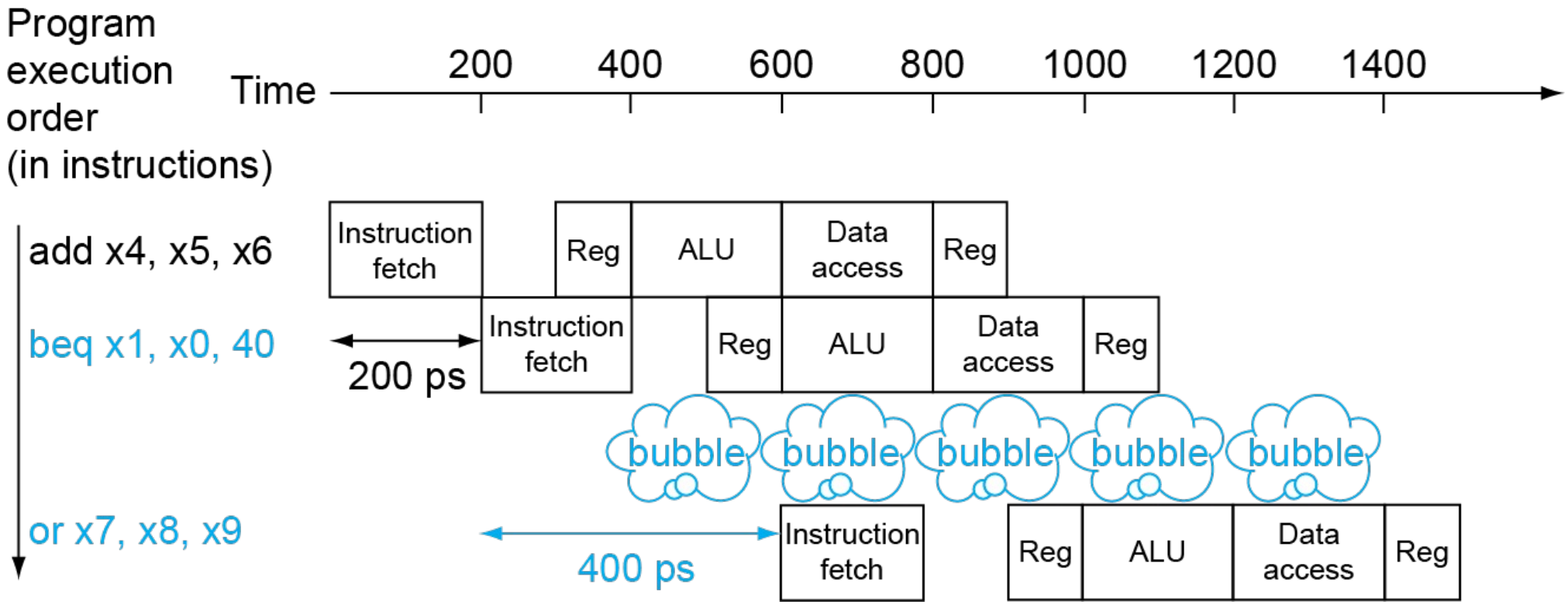
Control hazard

- Wait until branch outcome is determined before fetching next instruction



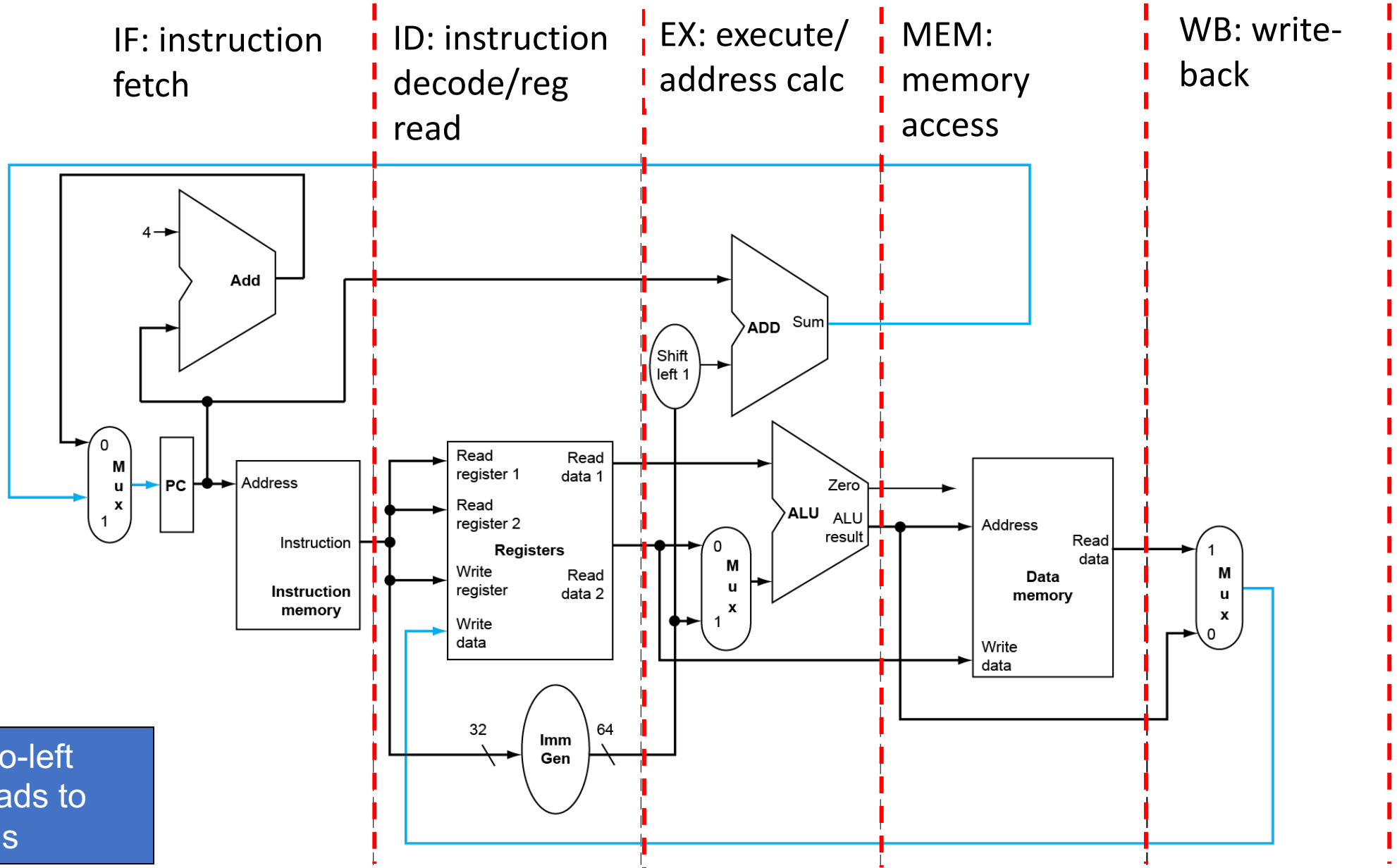
Control hazard

- Wait until branch outcome is determined before fetching next instruction



A basic pipelined RISC-V CPU

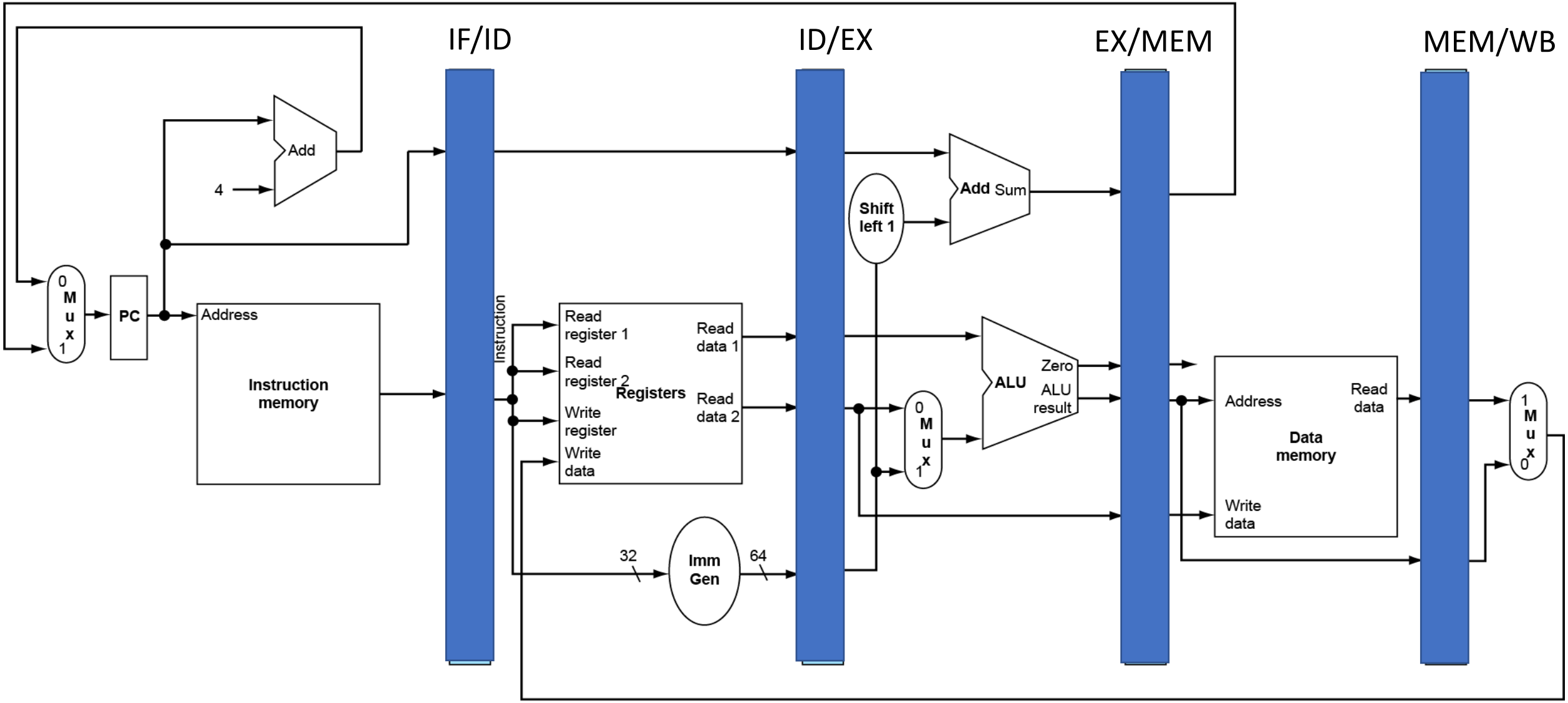
Pipelined Datapath



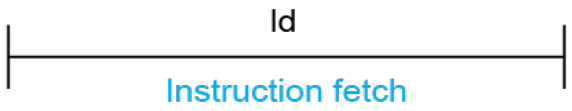
Right-to-left flow leads to hazards

Pipeline registers

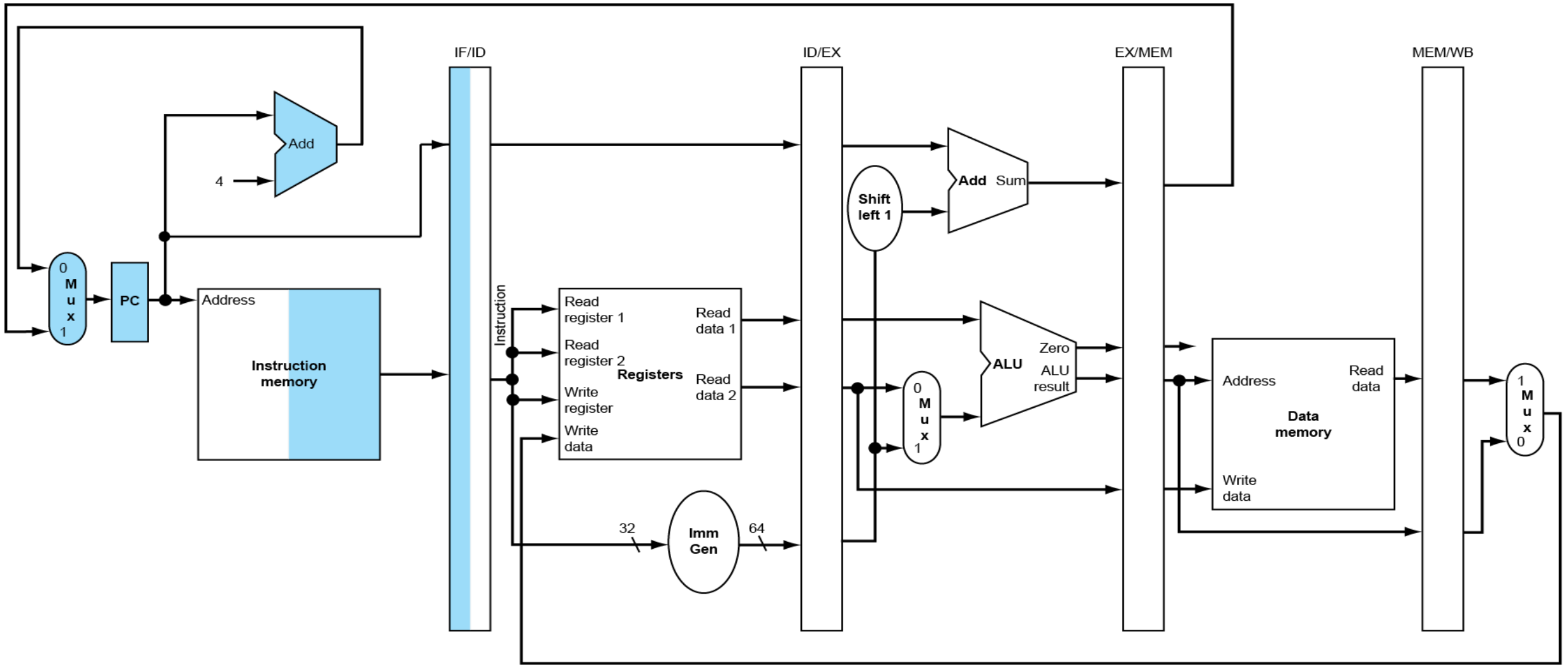
: needed to hold data produced in previous cycle



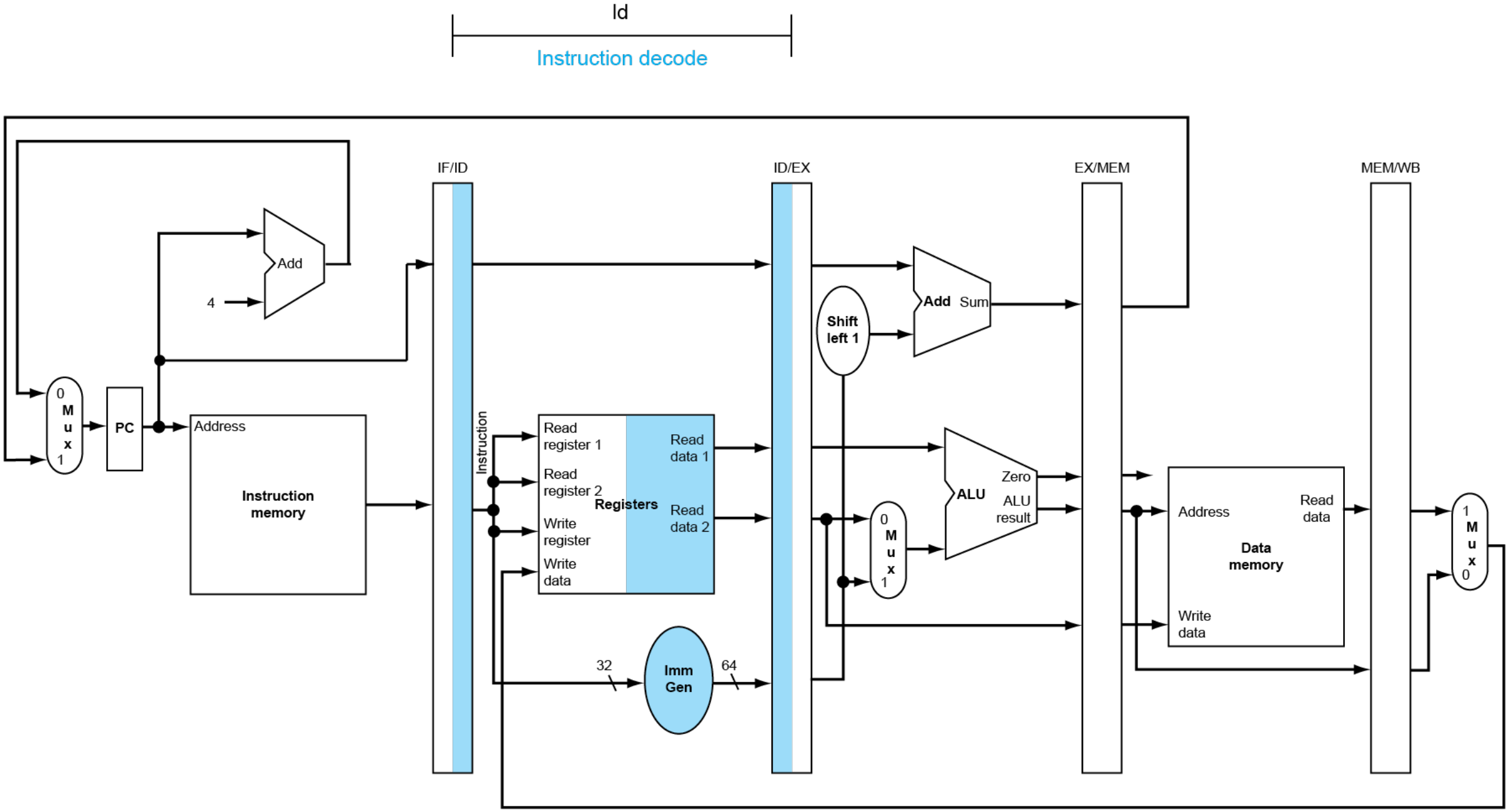
IF for Load, Store



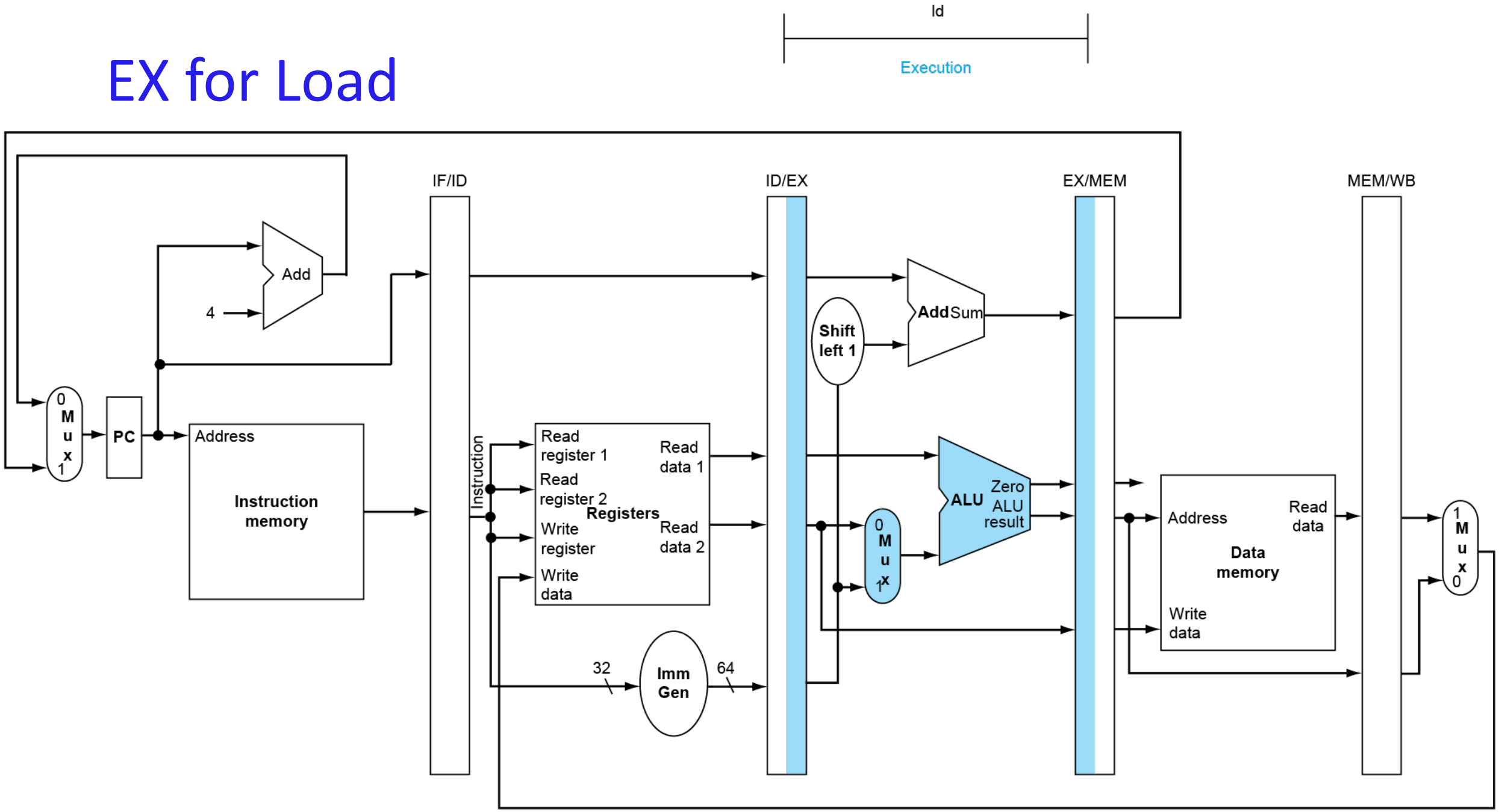
Single-clock-cycle diagram shows the state of an entire datapath during a single clock cycle



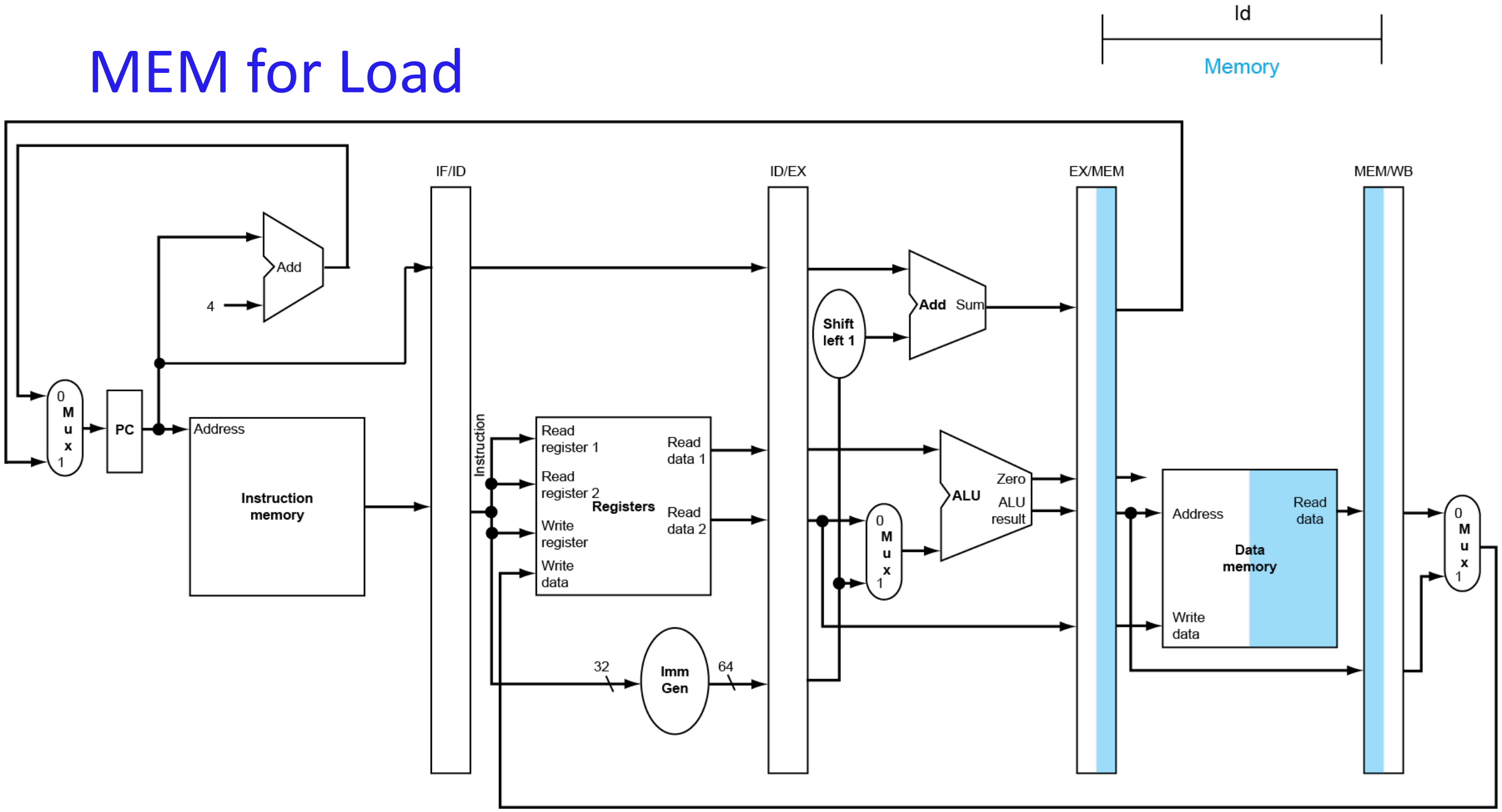
ID for Load, Store, …



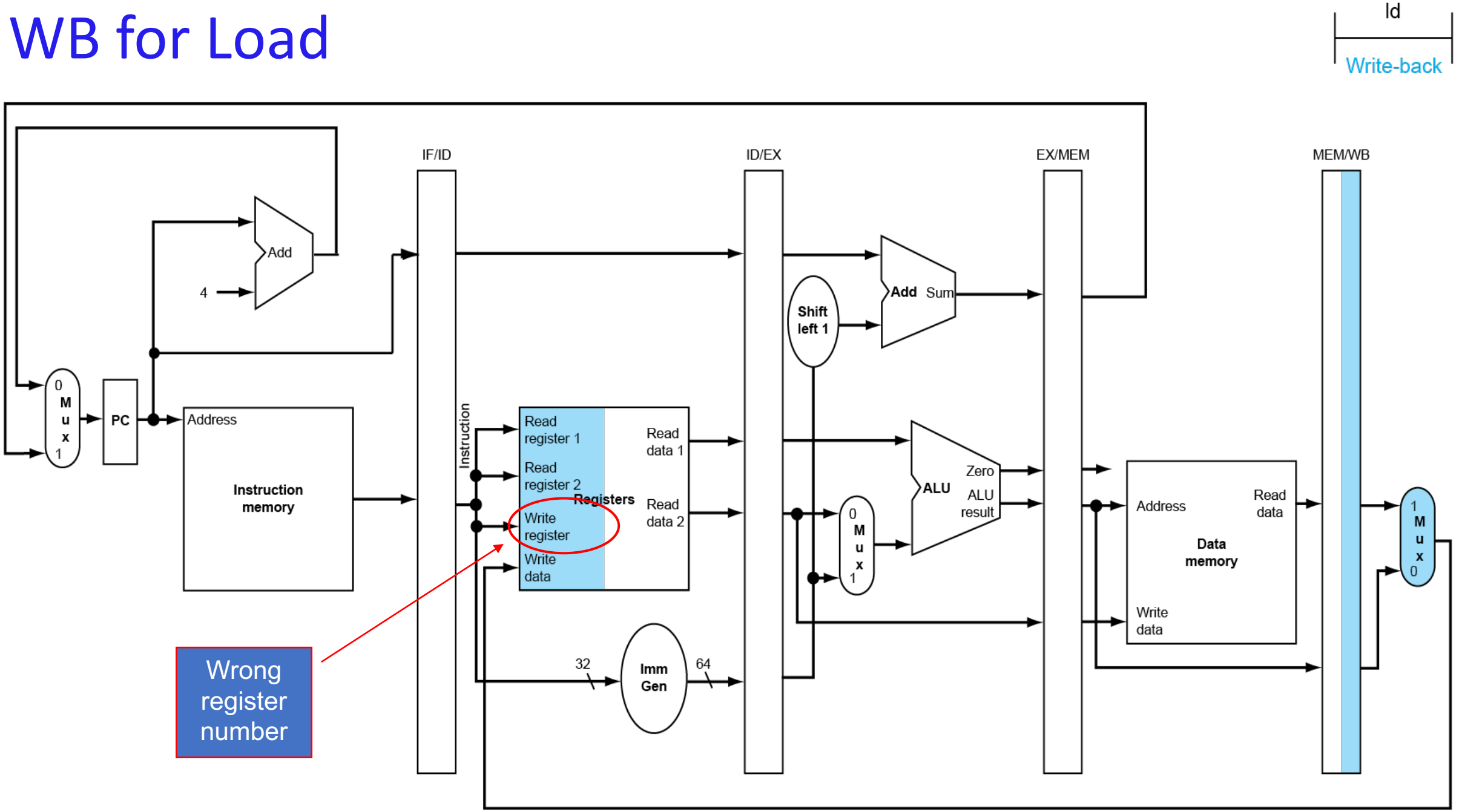
EX for Load


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

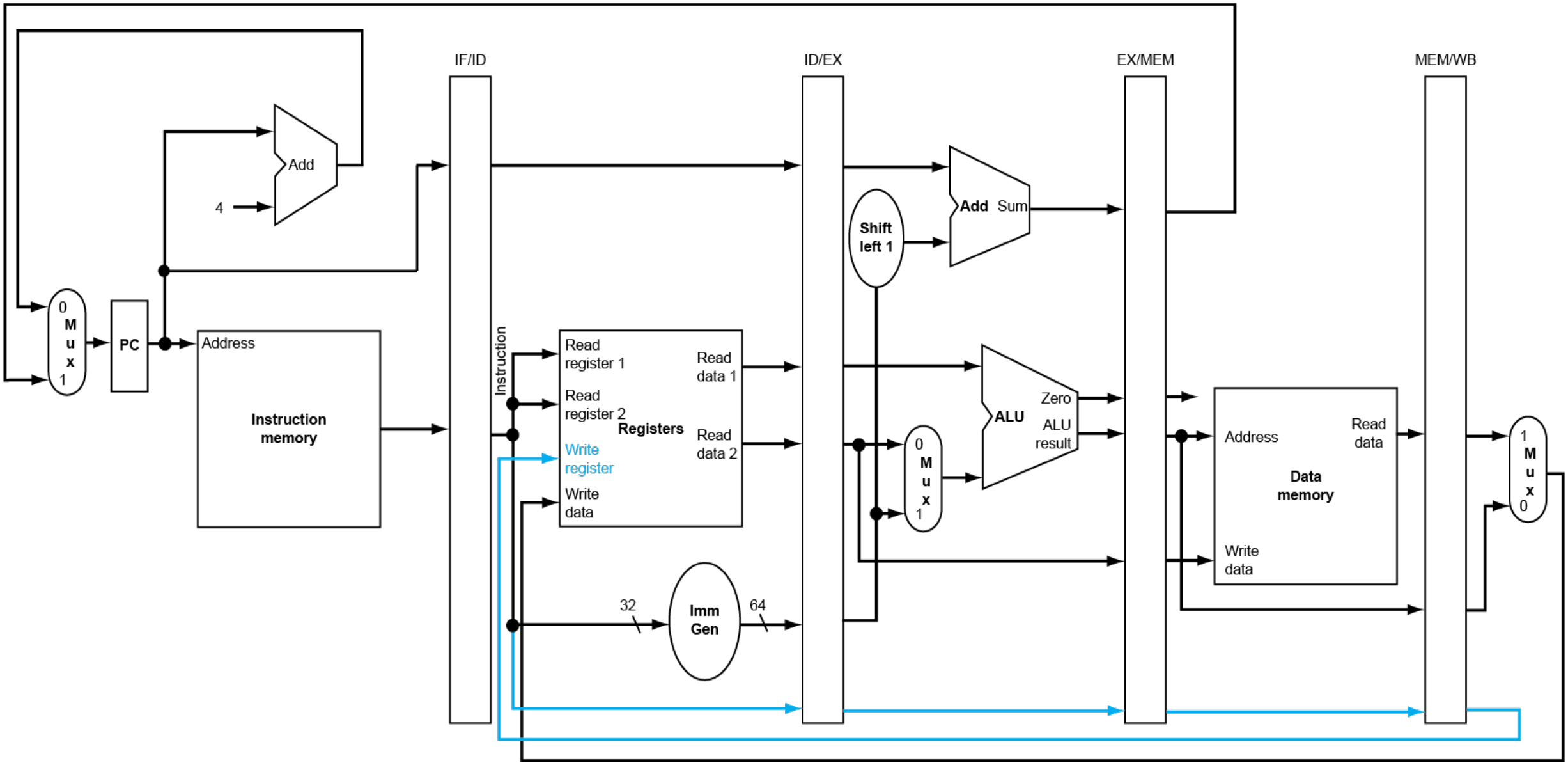
MEM for Load


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
Data memory
Read data
Write data

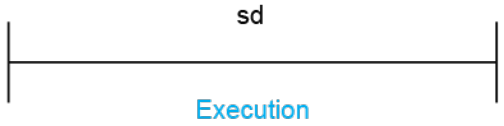
WB for Load

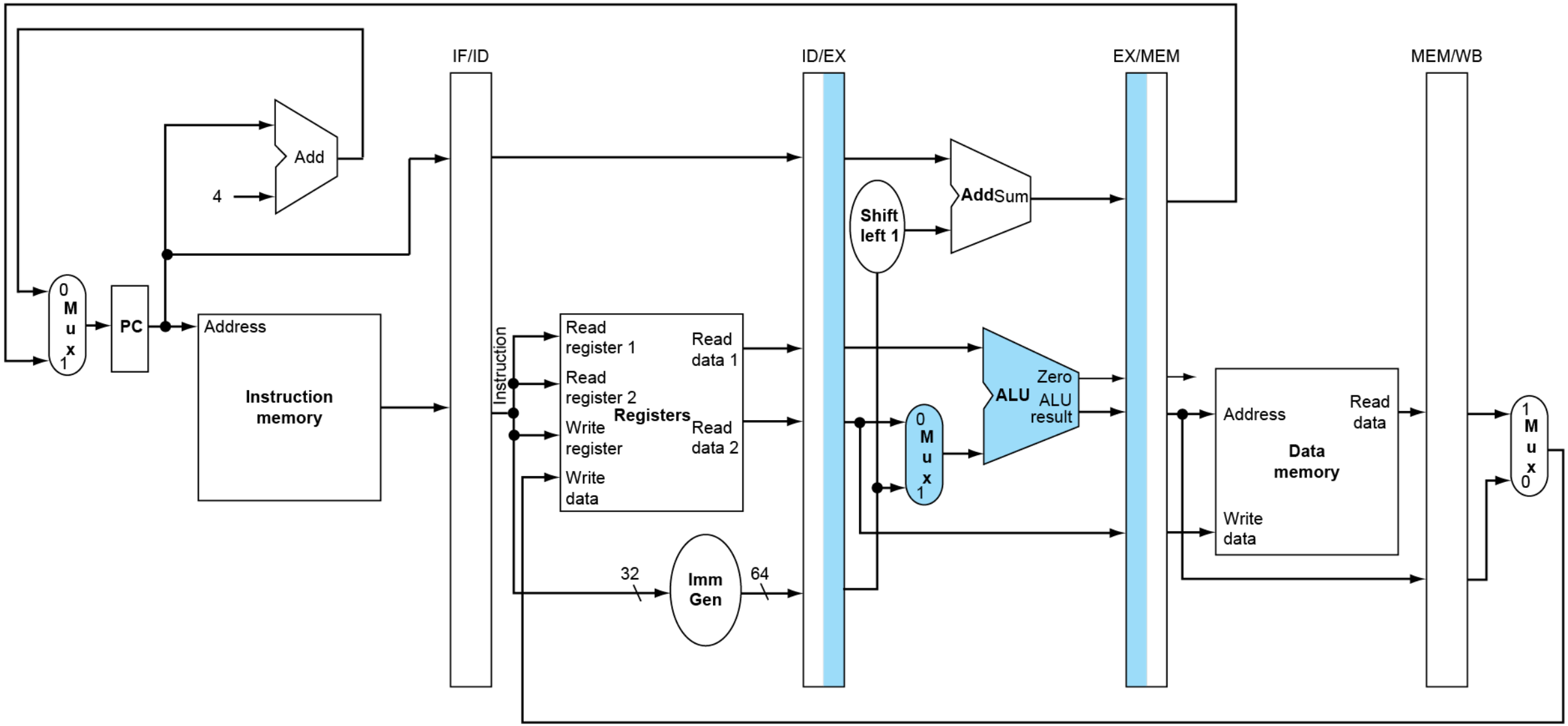

ld
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 1
Add
Sum
ALU
Zero
ALU result
0
M
u
x
1
32
Imm Gen
64
Address
Read data
Data memory
Write data
1
M
u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

Corrected Datapath for Load

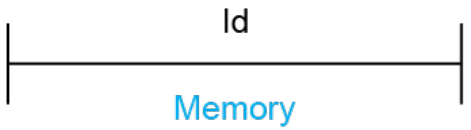


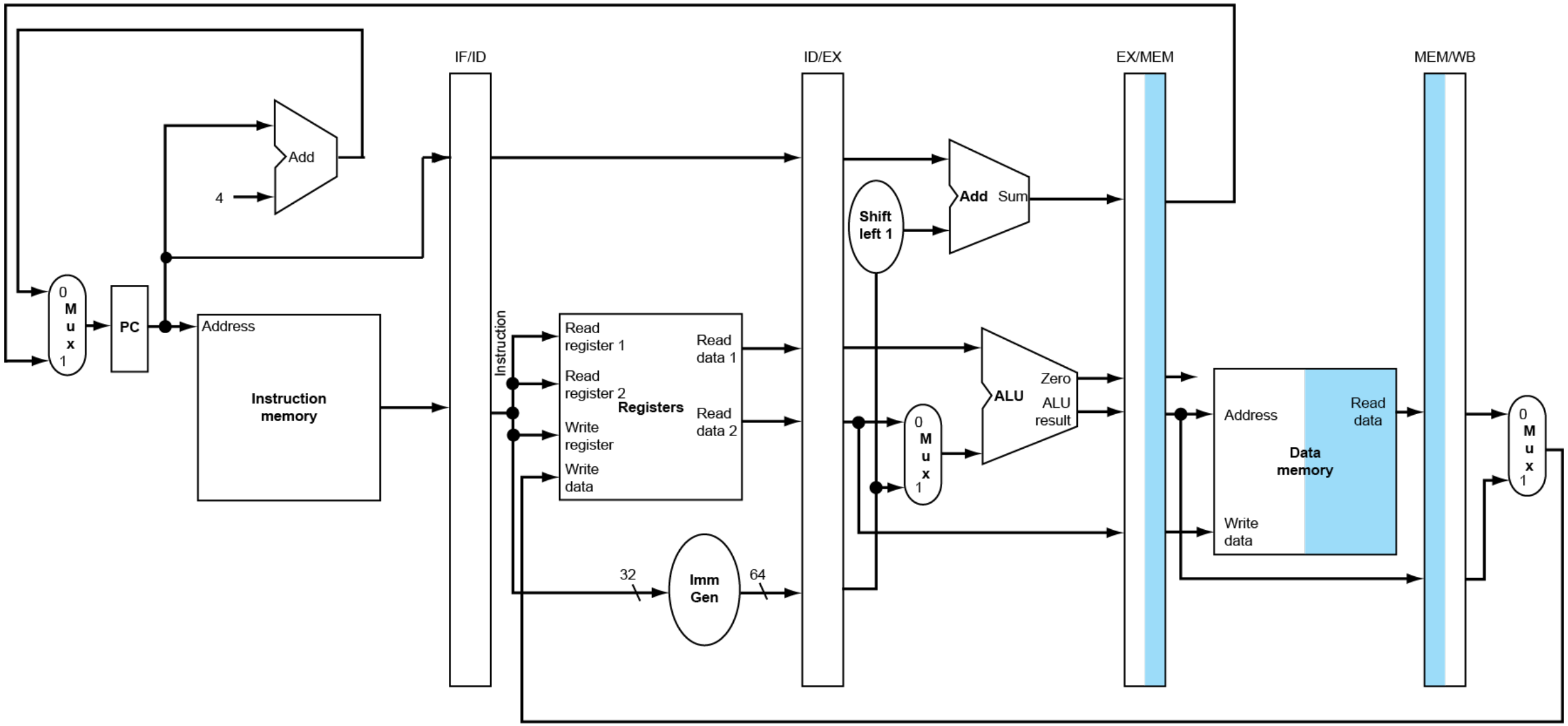
EX for Store


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

MEM for Store

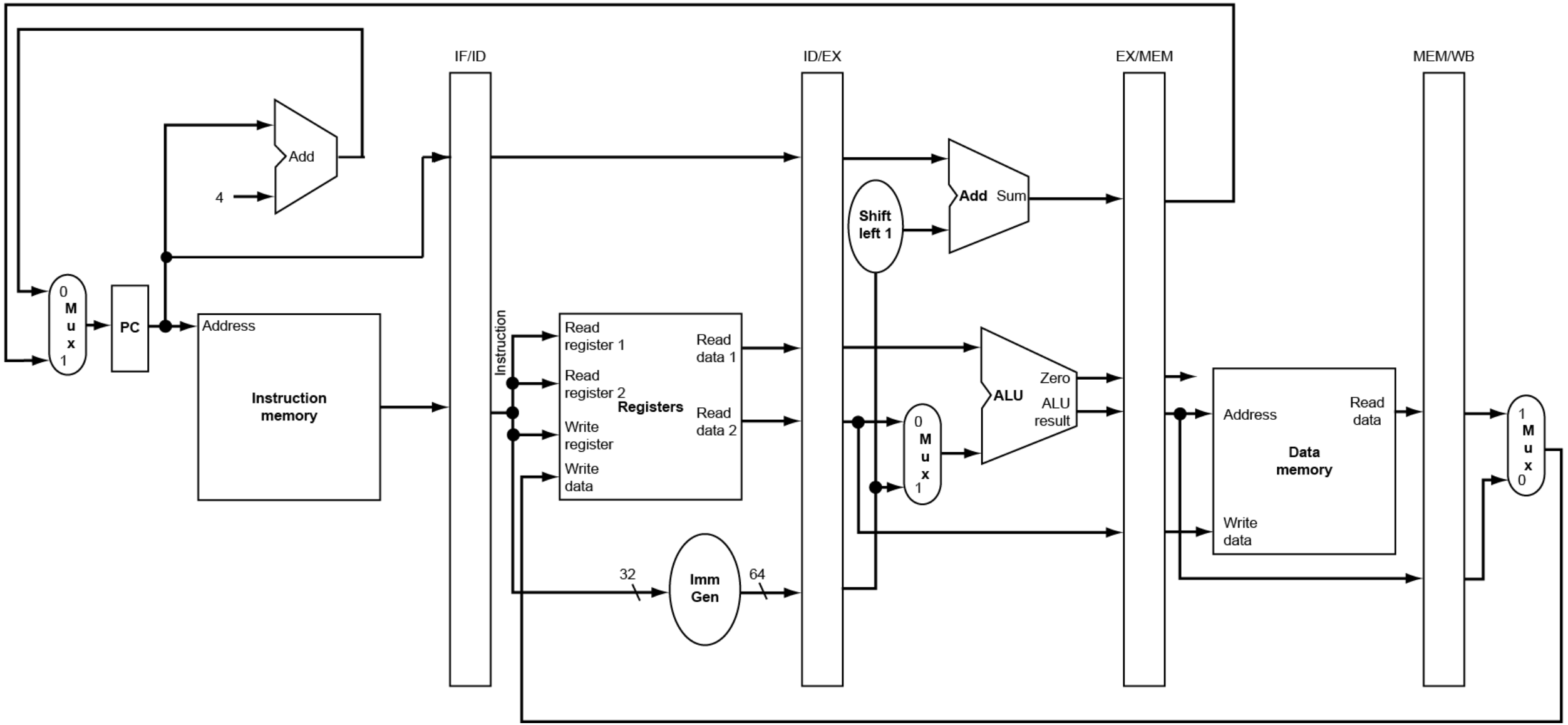




WB for Store

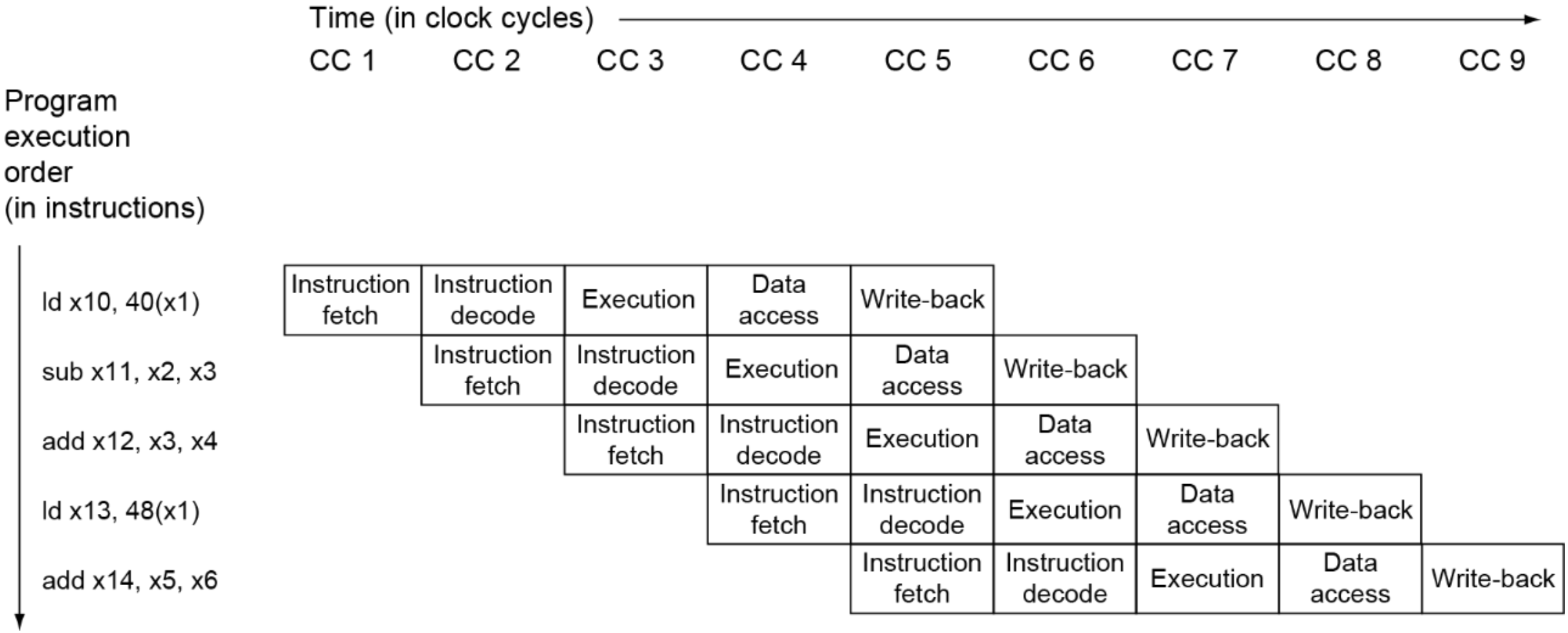
sd

Write-back



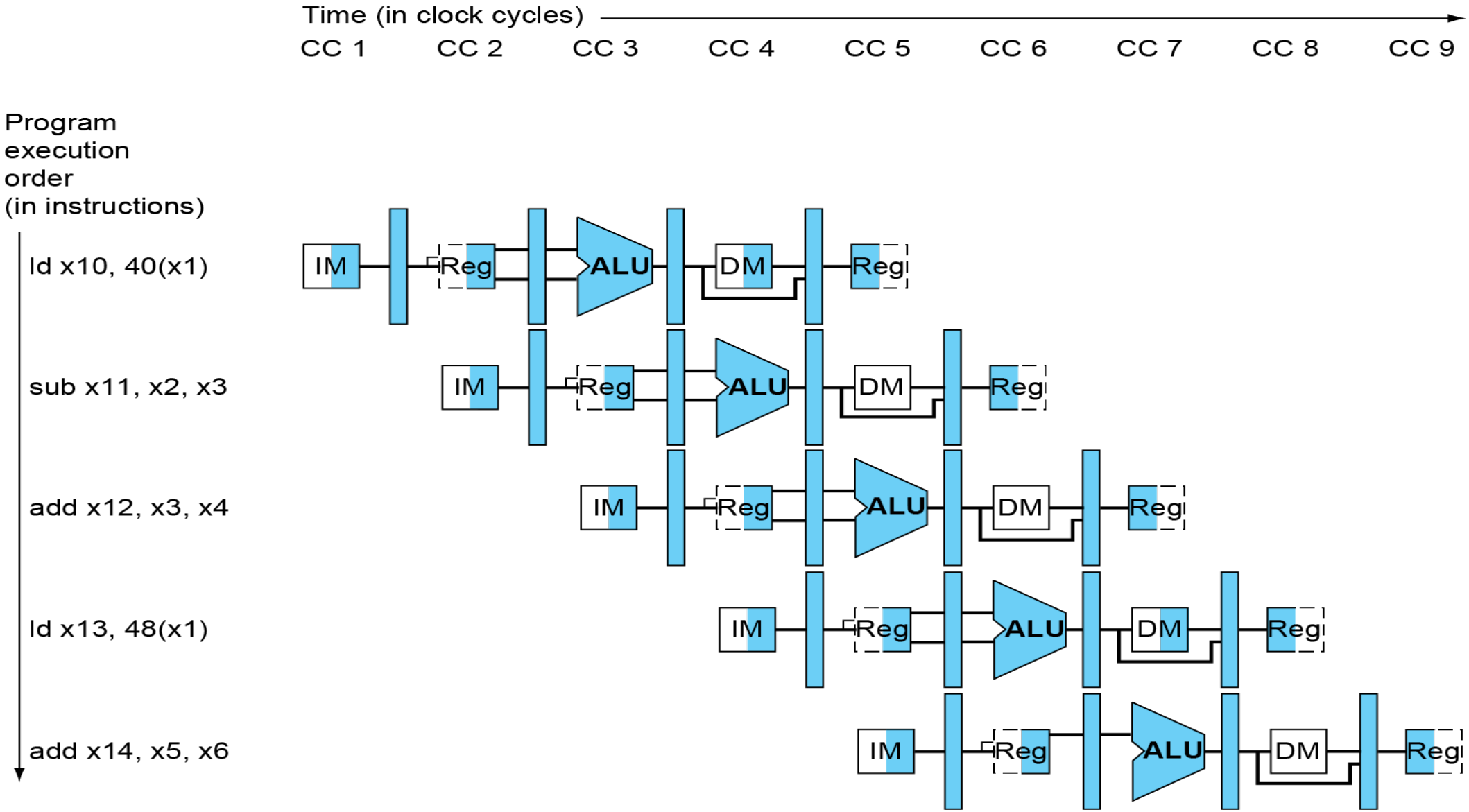
Multi-Cycle Pipeline Diagram

- Traditional form

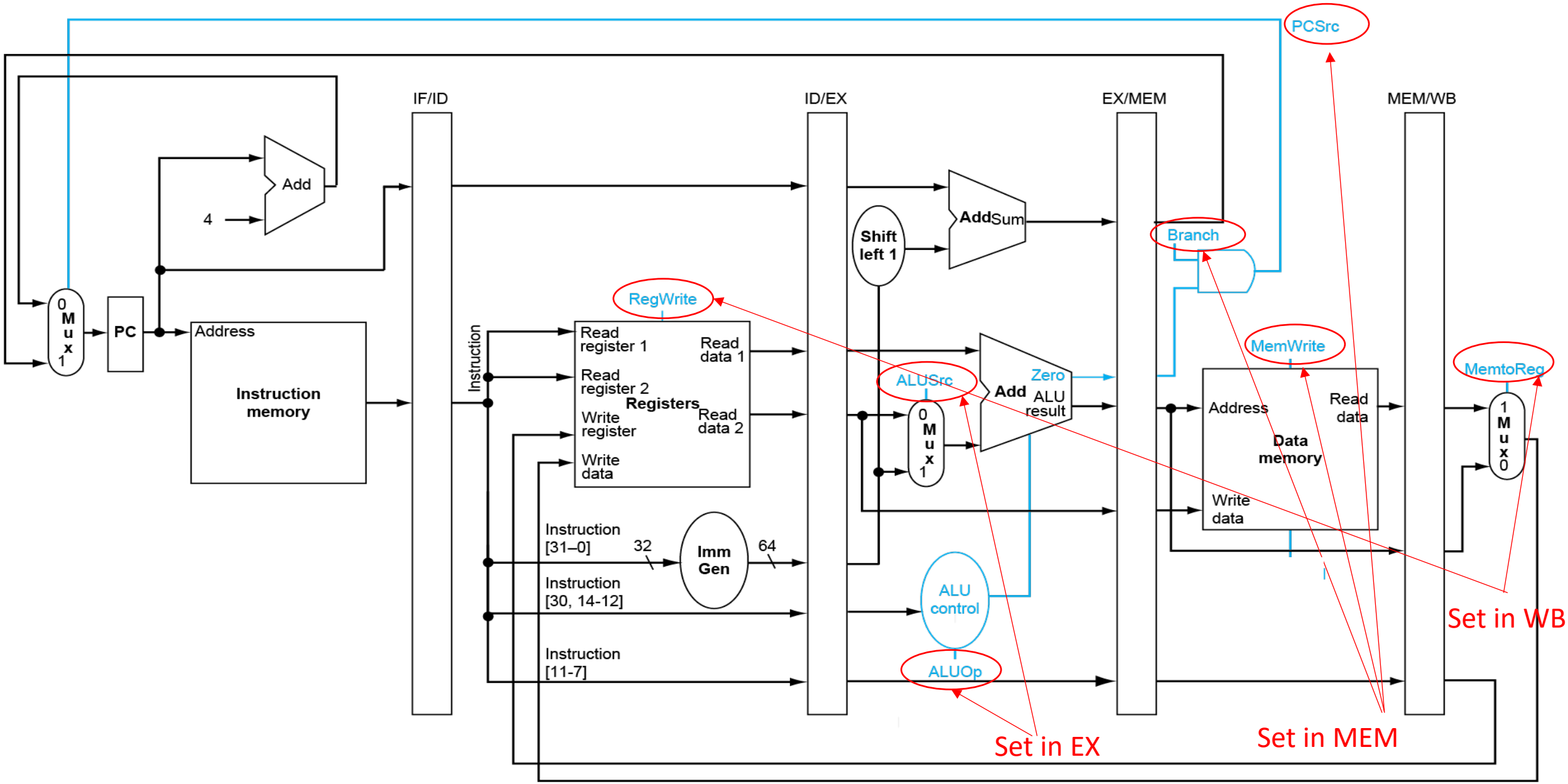


Multi-Cycle Pipeline Diagram

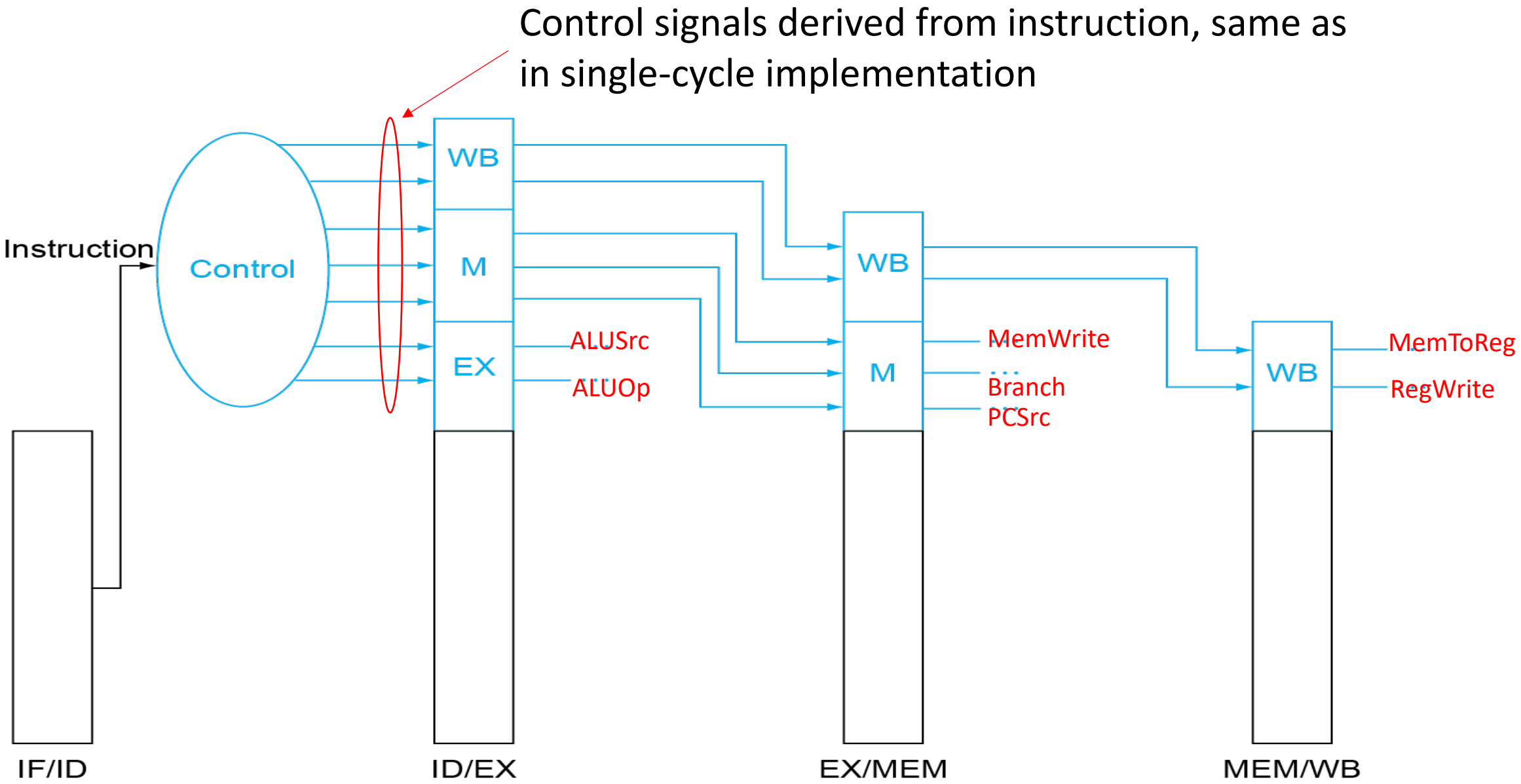
- Form showing resource usage



Pipelined Control (Simplified)



Pipelined Control



Summary

- Basic single-cycle CPU design
  - Data path vs. control path
  - Clock frequency is limited by the longest delay
- Basic 5-stage pipelined design:
  - Main idea: Parallel processing of different stages of an instruction's execution
  - RISC-V 5-stage pipeline (IF, ID, EXE, MEM, WB)
  - Pipeline hazards: structure, data, control