Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

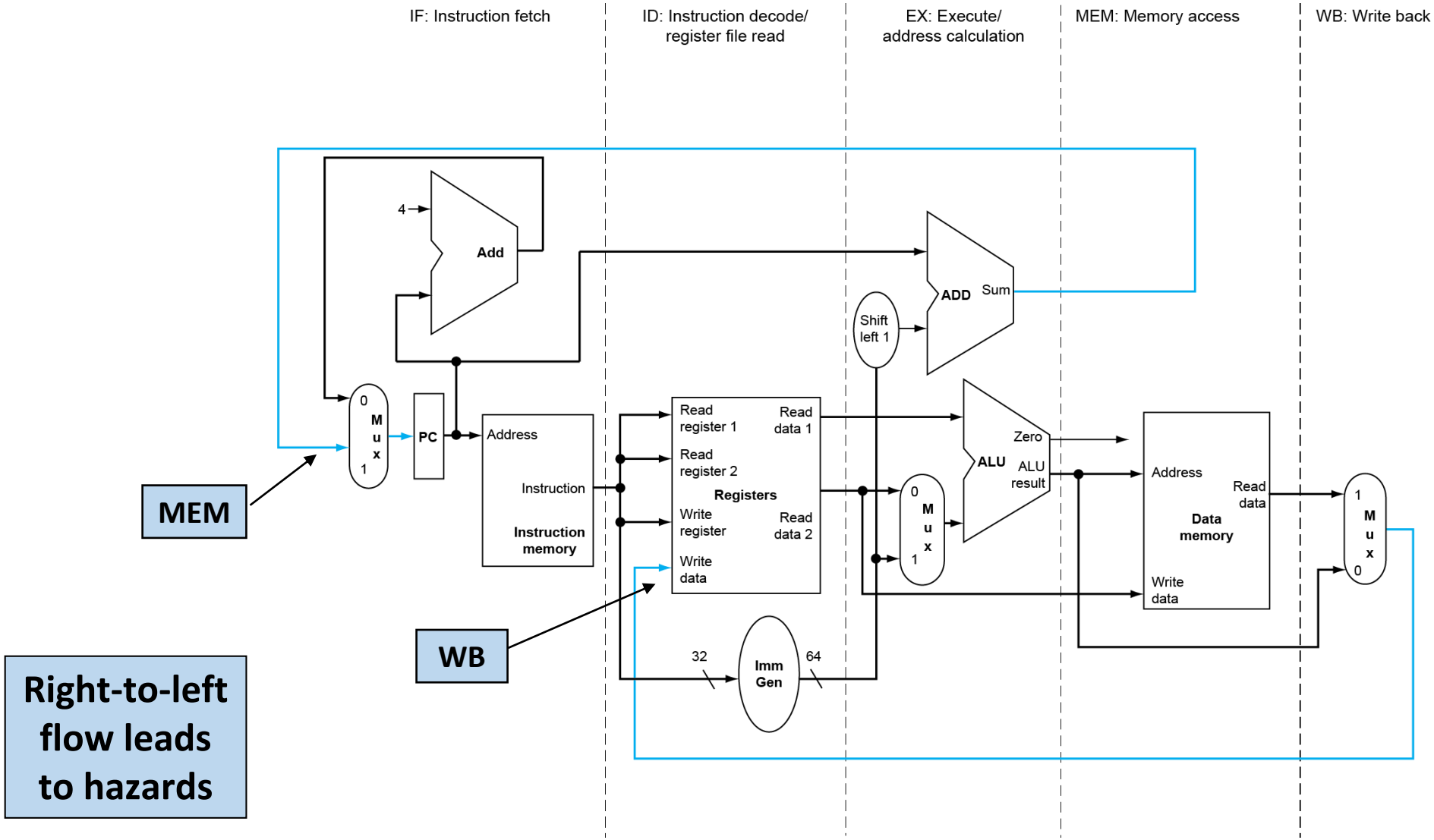
Fall 2019

Pipelined Processor

Pipelined Datapath and Control

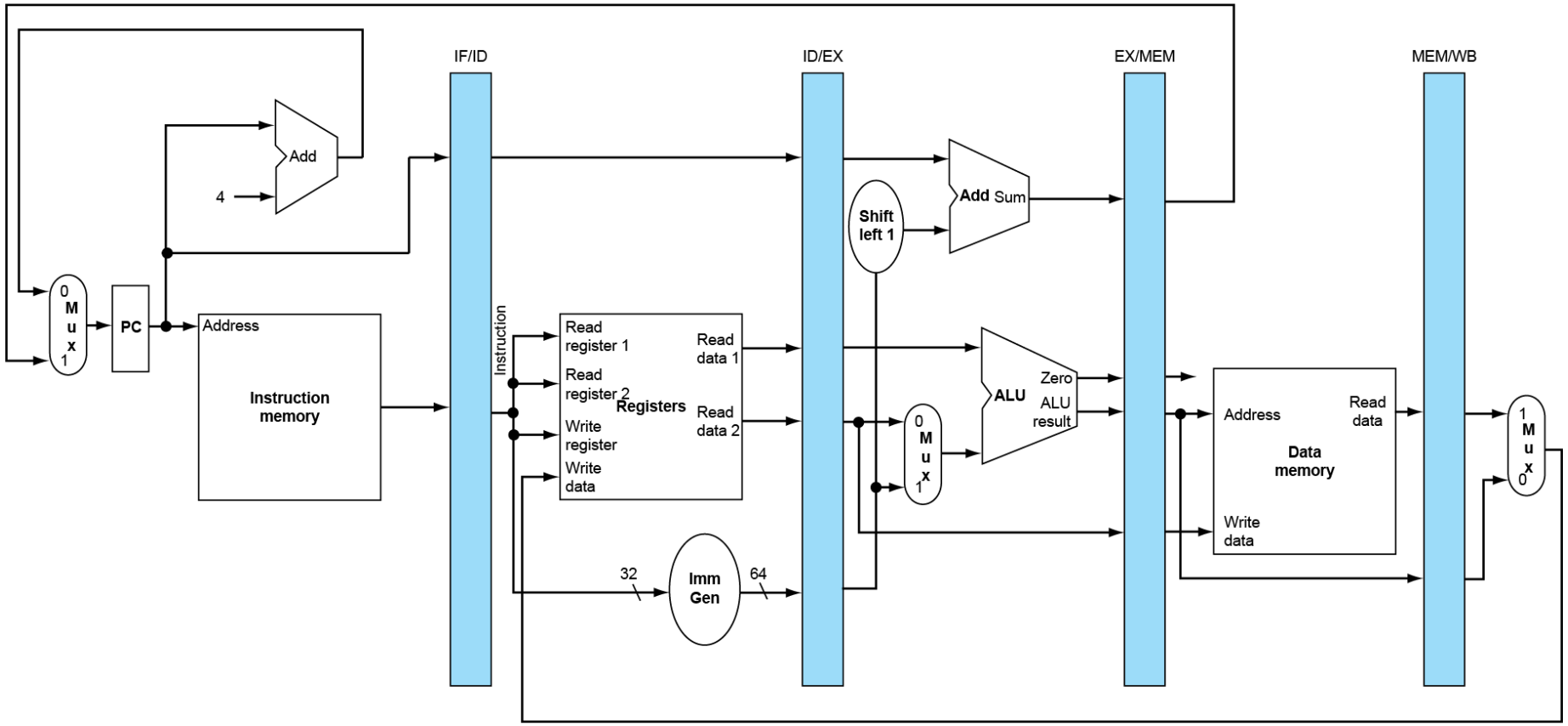
Chap. 4.6

RISC-V Pipelined Datapath



Pipeline Registers

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



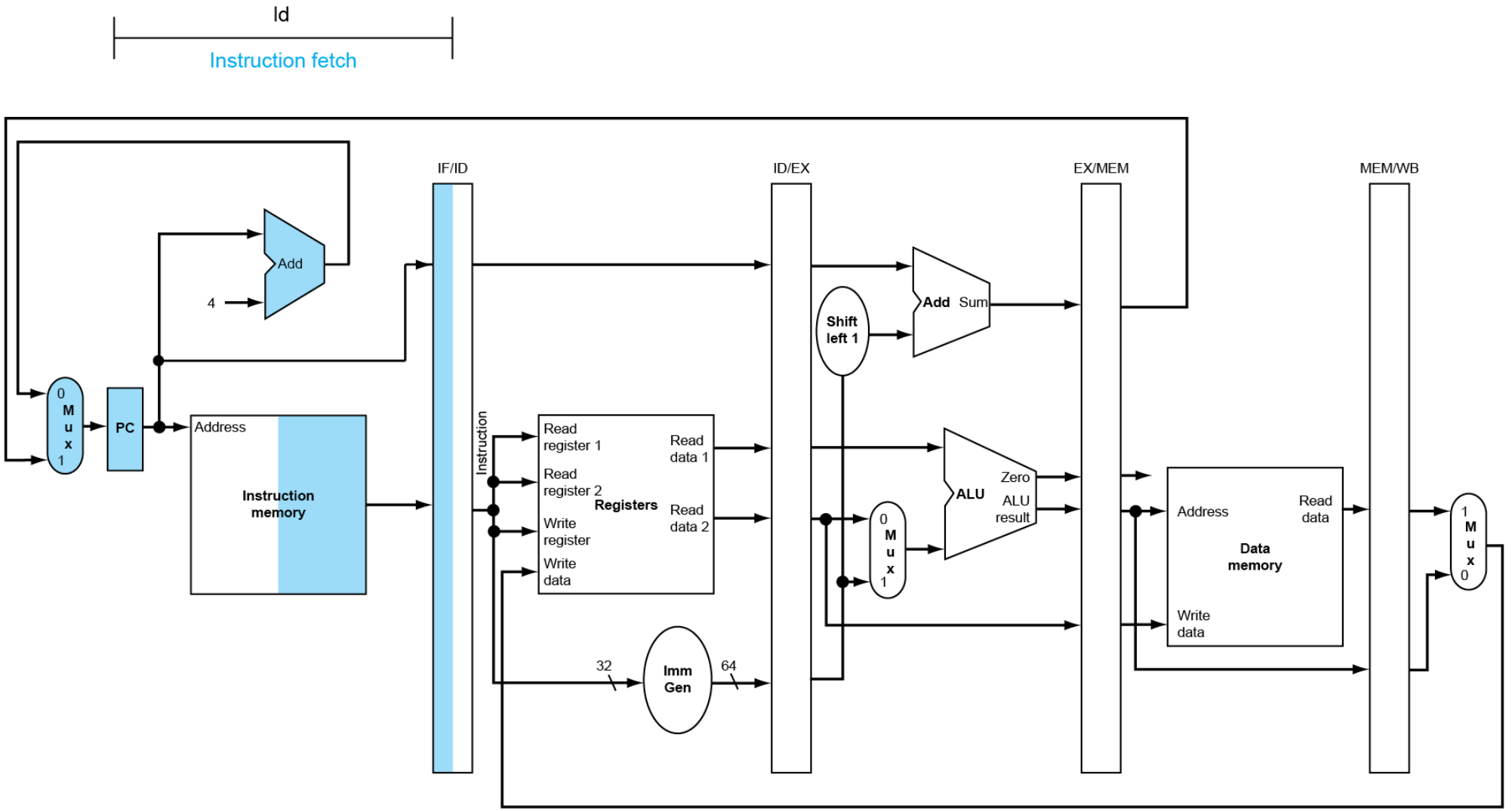
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath

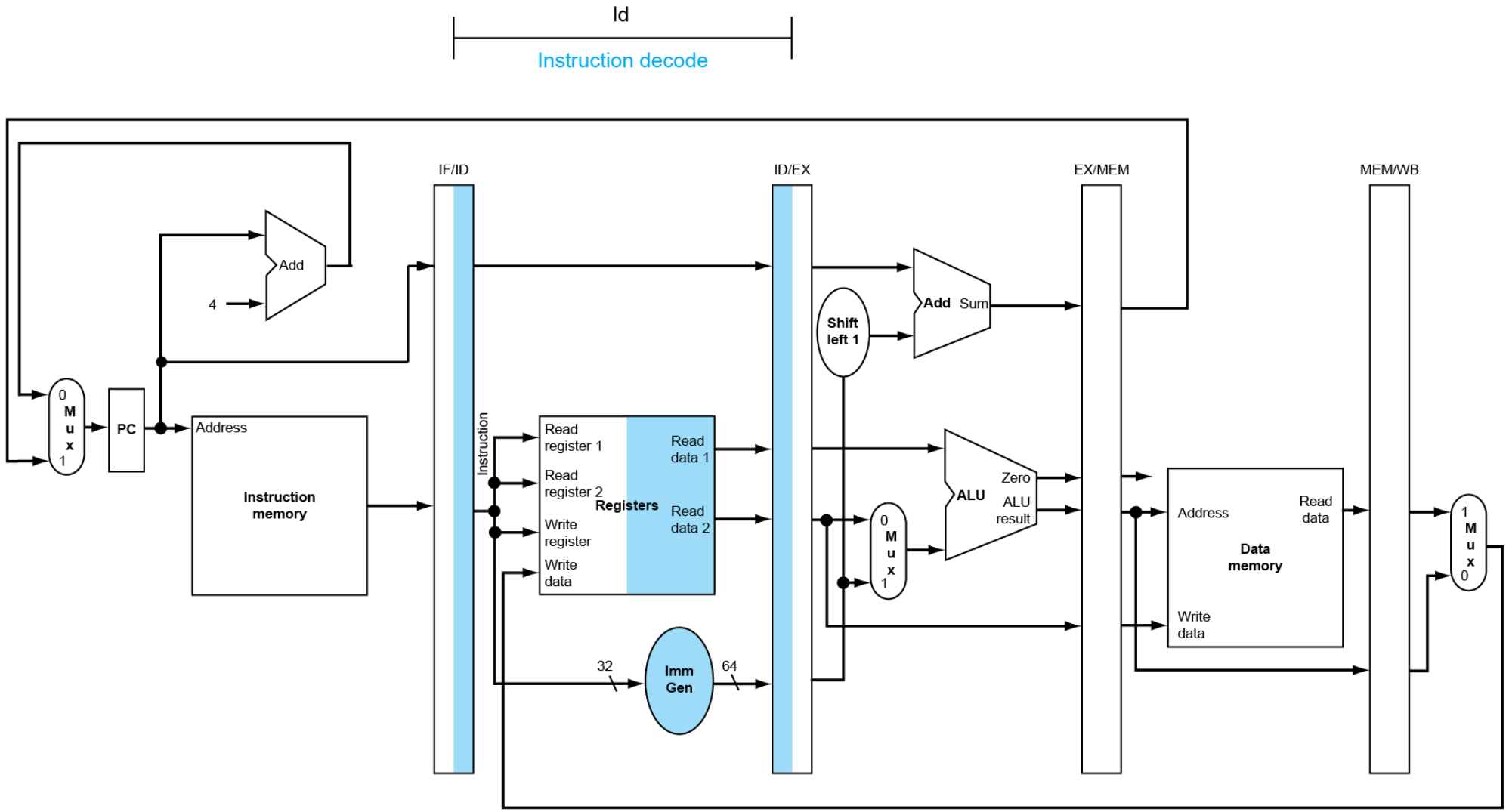
- "Single-clock-cycle" pipeline diagram
  - Shows pipeline usage in a single cycle
  - Highlight resource used
  - (cf.) "multi-clock-cycle" diagram: graph of operation over time

- We'll look at "single-clock-cycle" diagrams for load & store

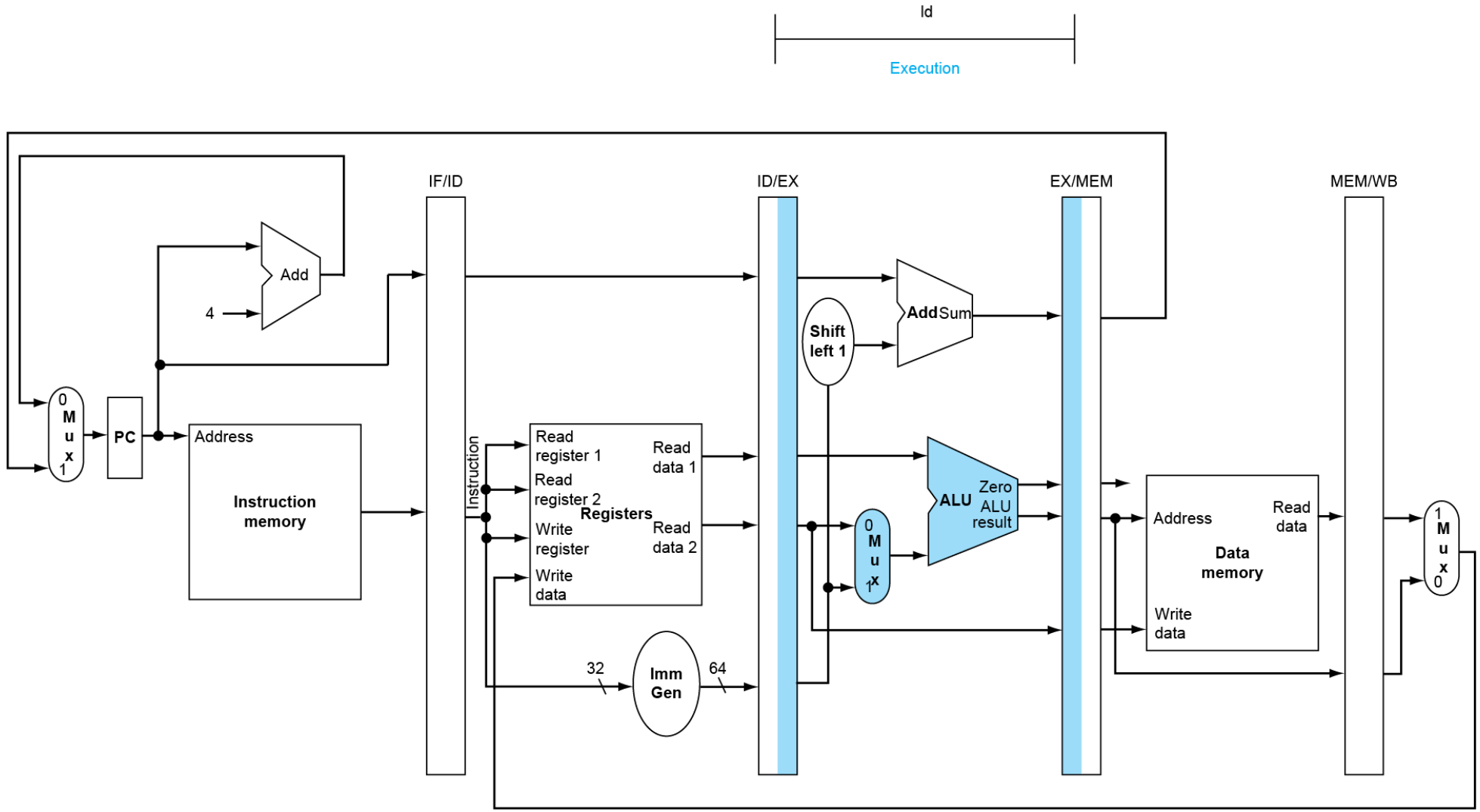
IF for Load


ld
Instruction fetch
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
1
M
u
x
0

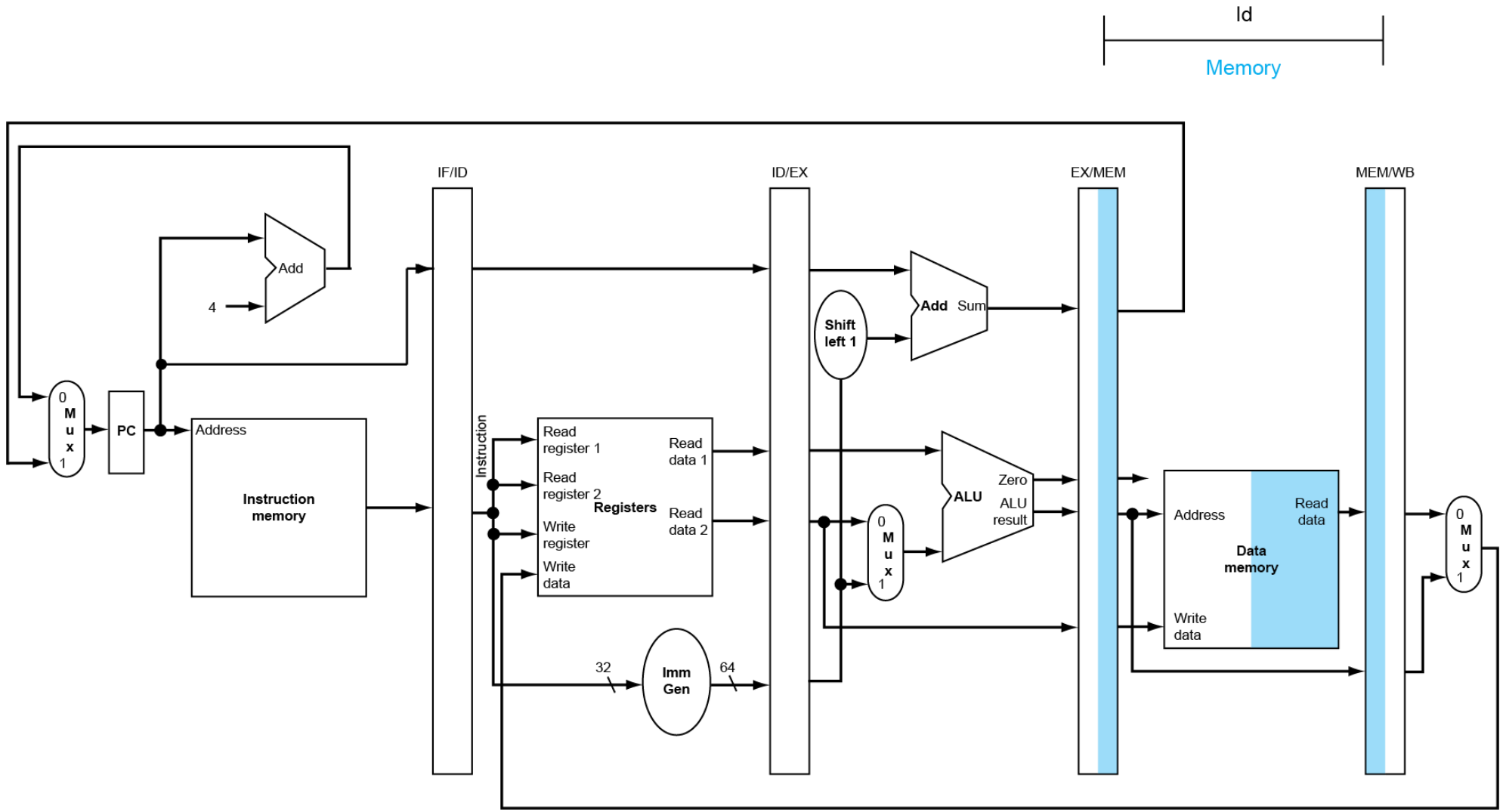
ID for Load


Id
Instruction decode
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
1
M
u
x
0

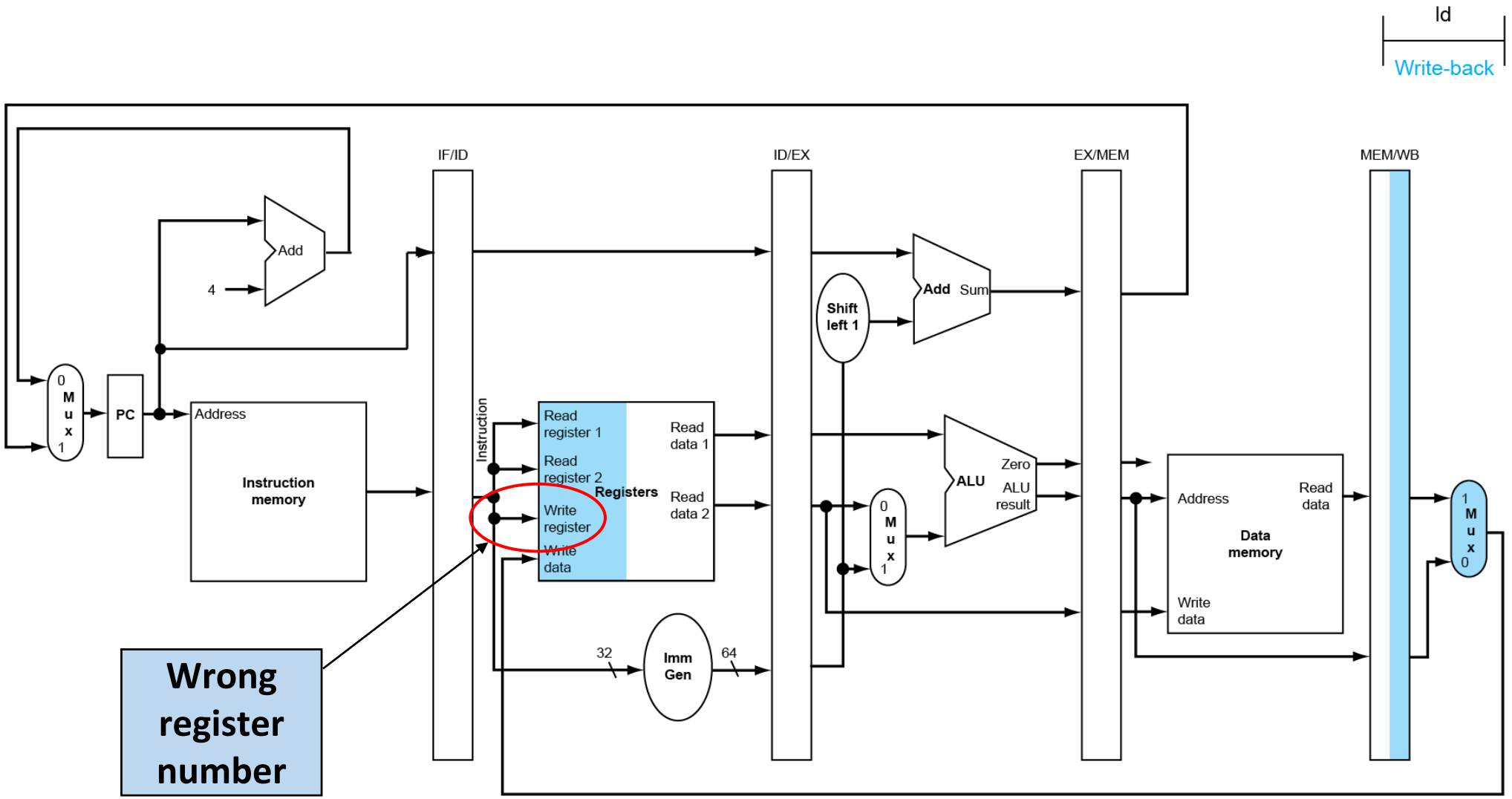
EX for Load


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
ALU
Zero
ALU result
0
M
u
x
1
Address
Read data
Data memory
Write data
1
M
u
x
0

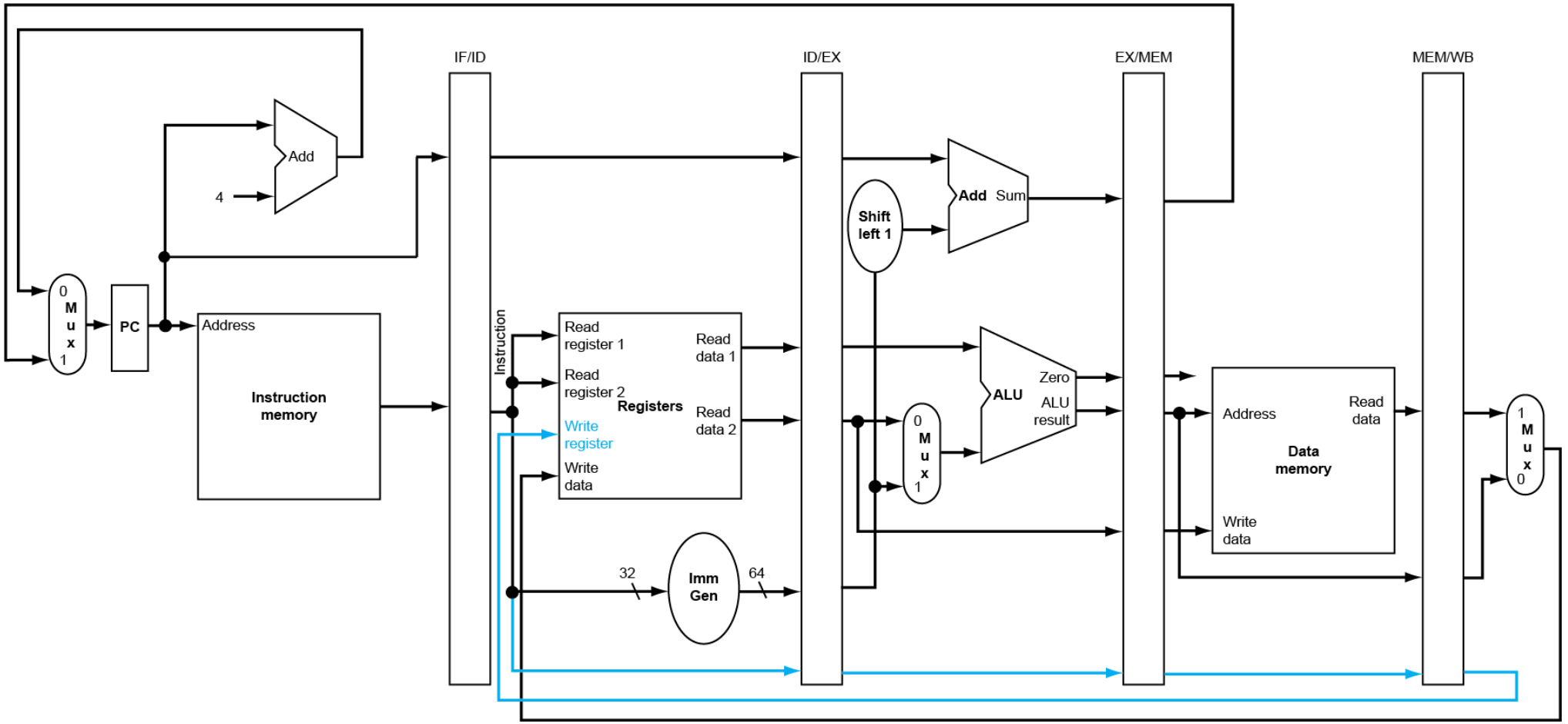
MEM for Load


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

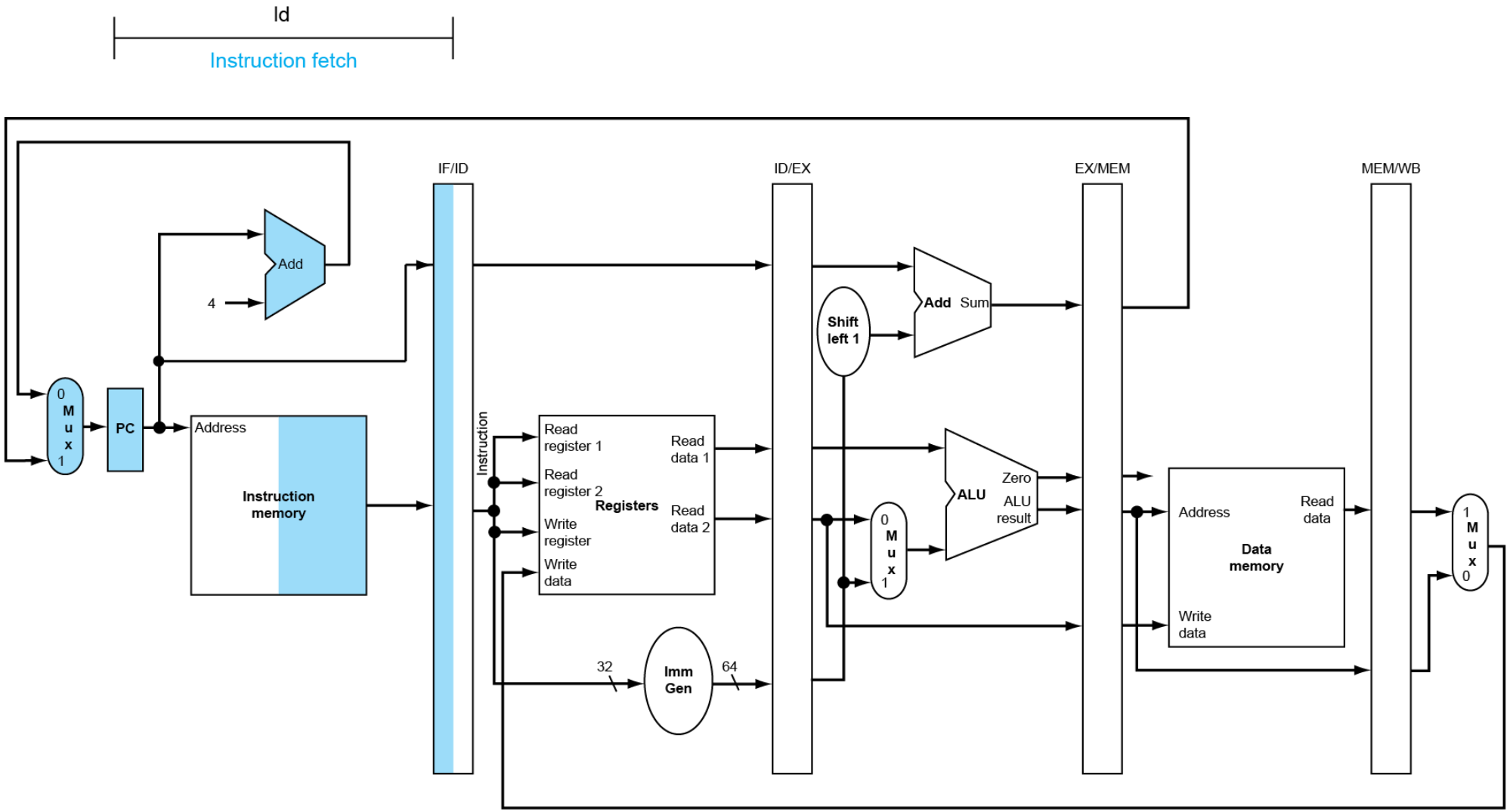
WB for Load


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
32
Imm Gen
64
Address
Read data
Data memory
Write data
1
M
u
x
0
Wrong register number

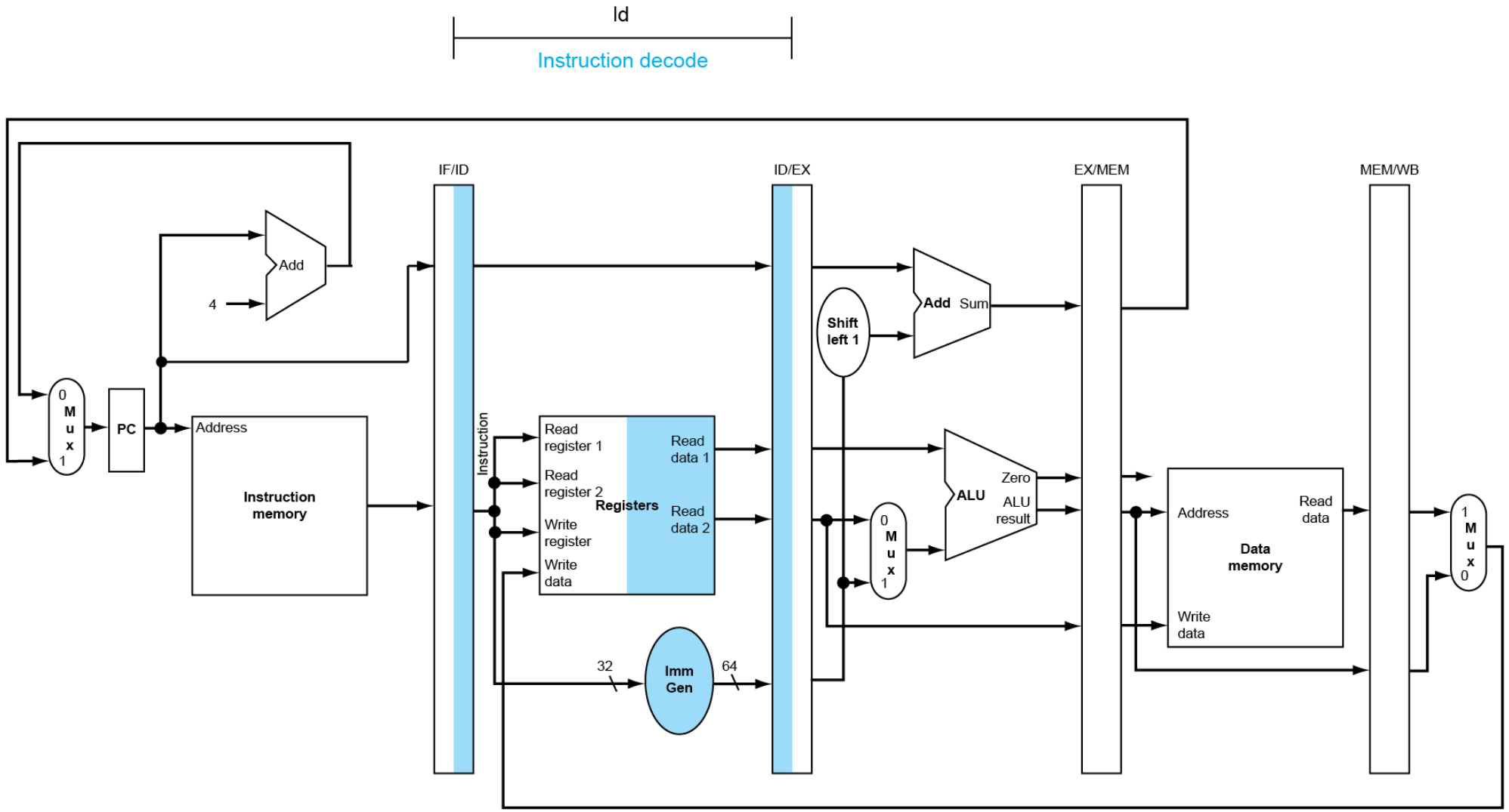
Corrected Datapath for Load



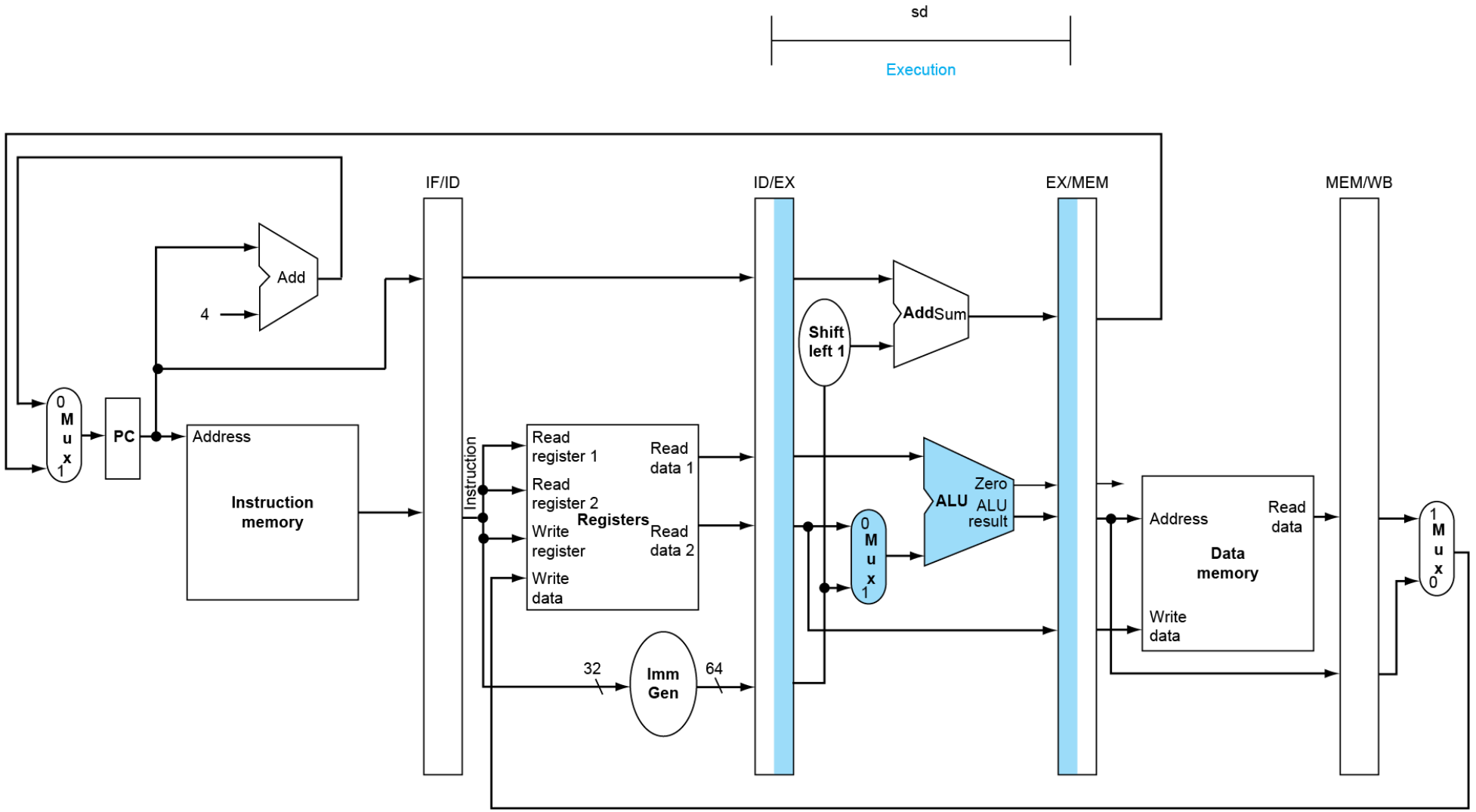
IF for Store


Id
Instruction fetch
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
1
M
u
x
0

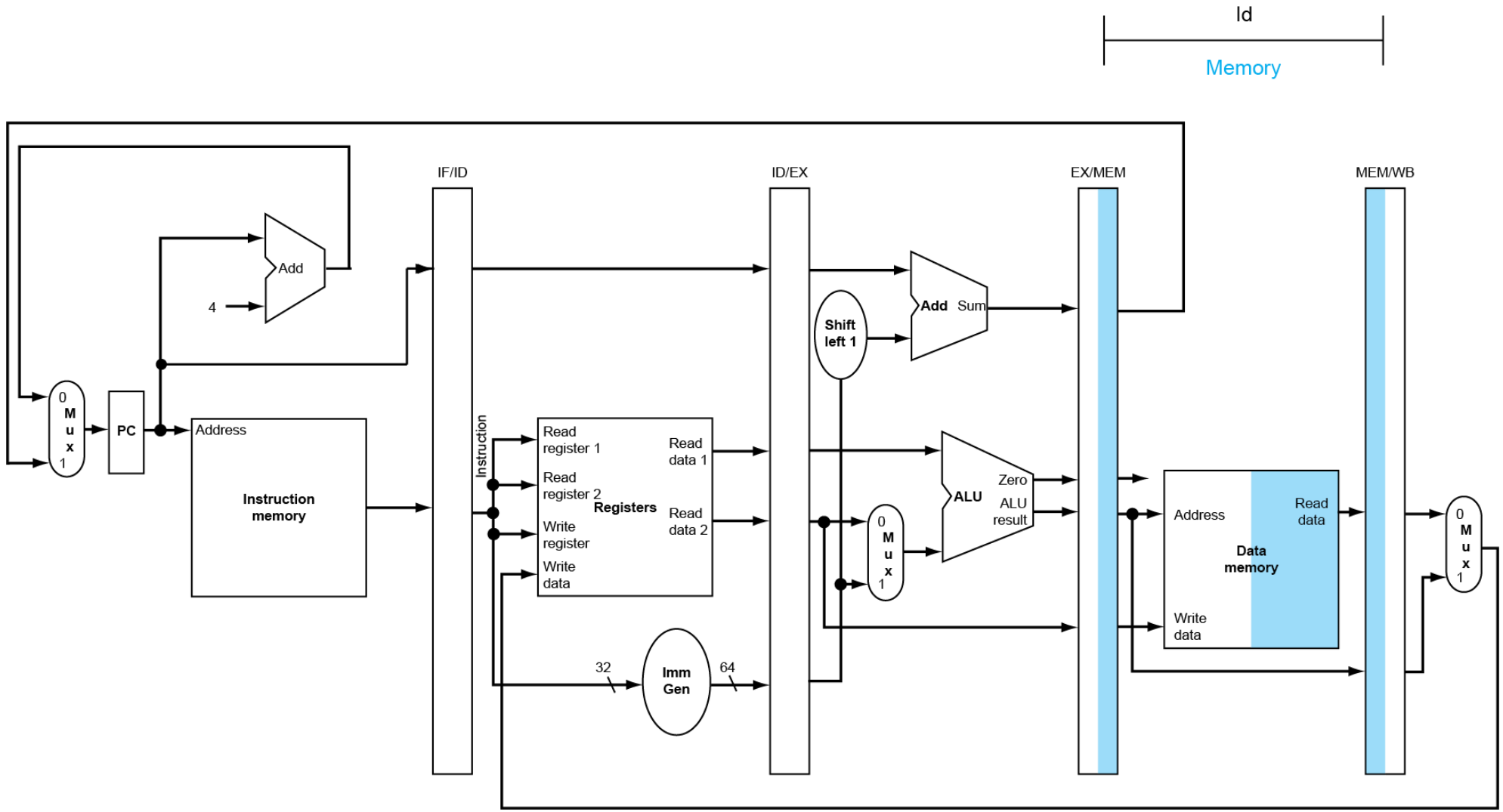
ID for Store


Id
Instruction decode
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 1
Add
Sum
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
32
Imm Gen
64

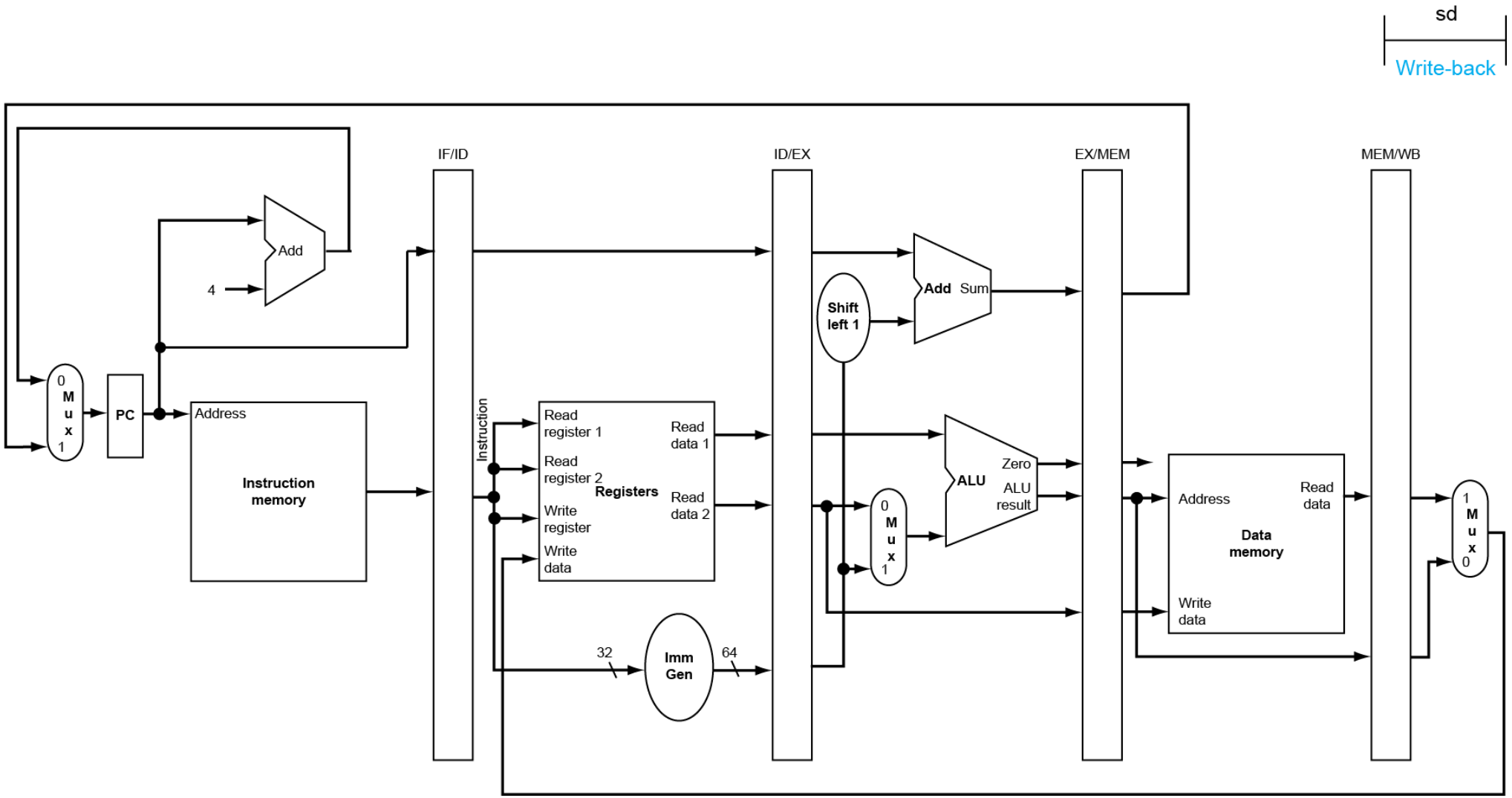
EX for Store


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

MEM for Store

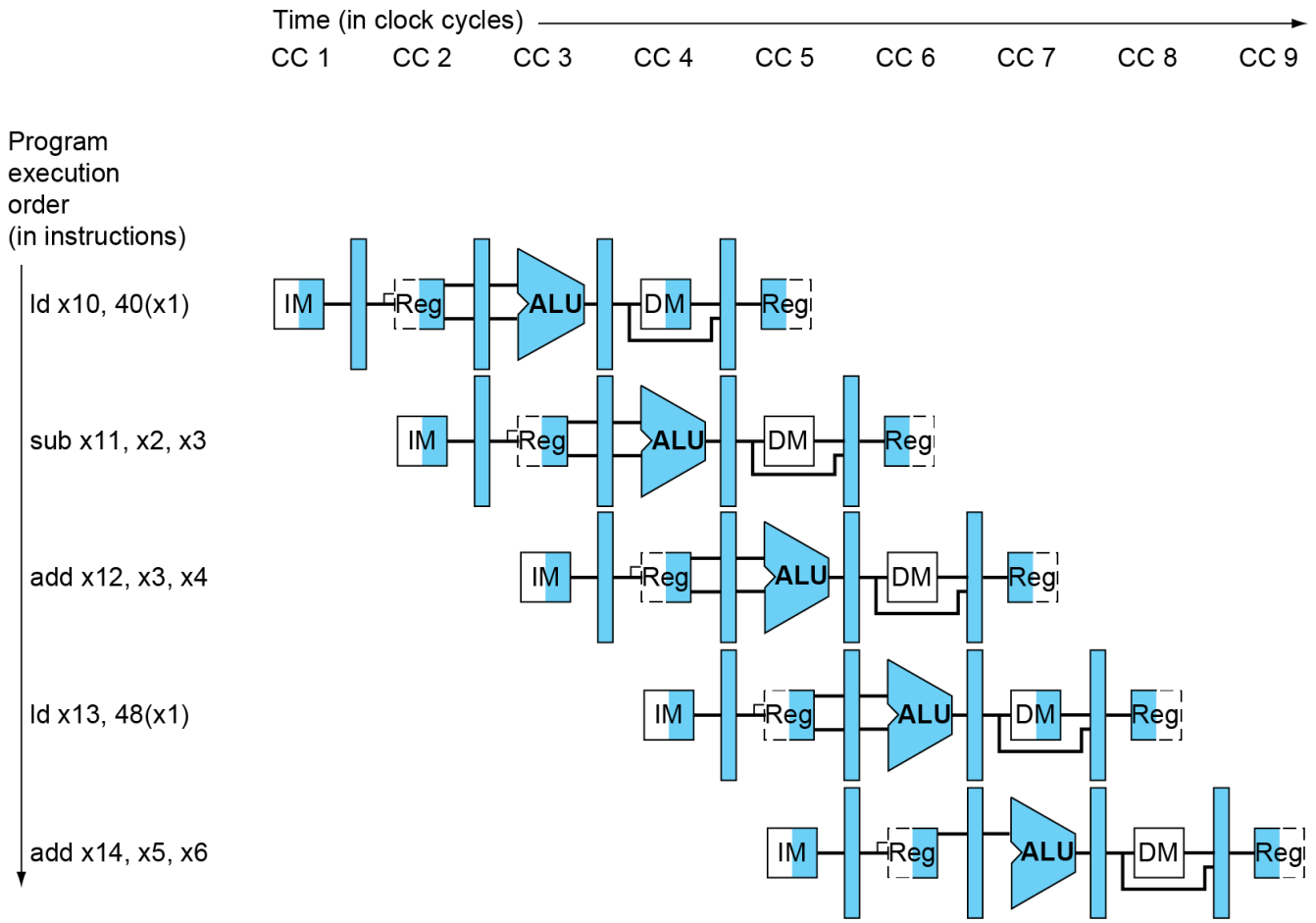


WB for Store


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

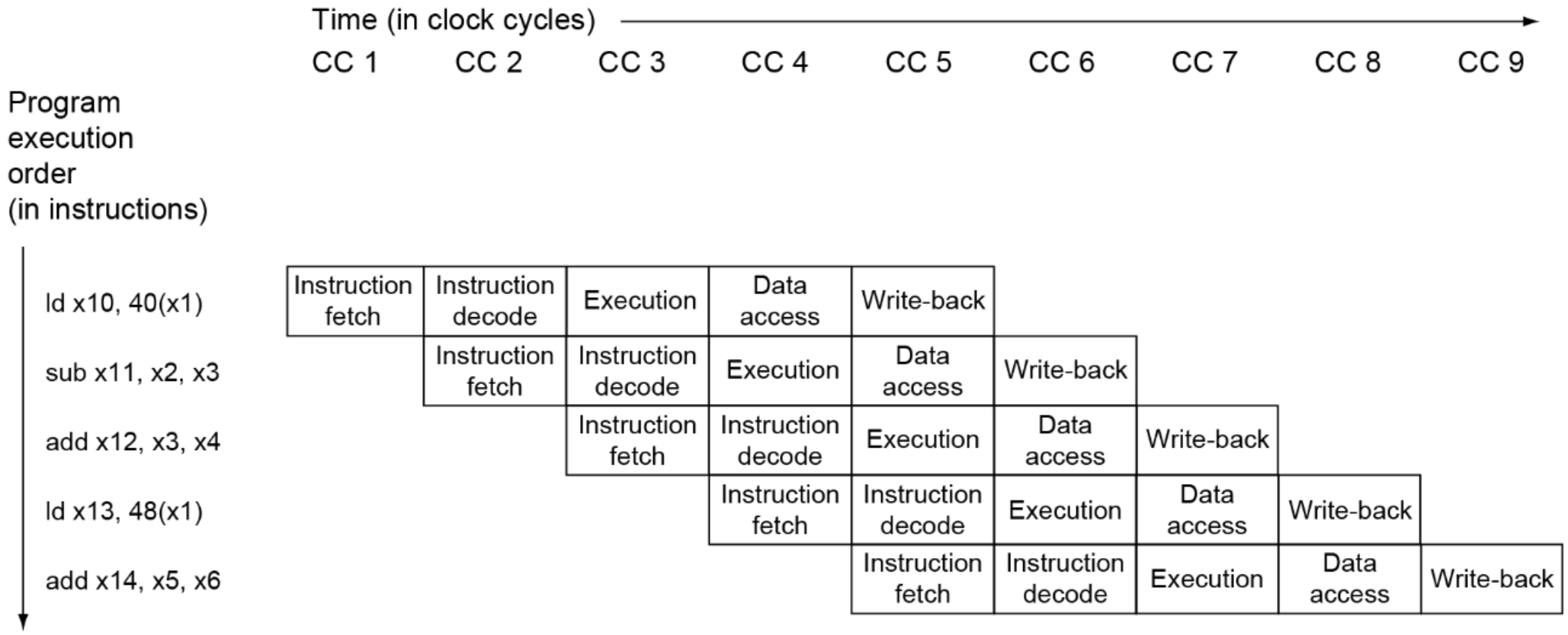
Multi-Cycle Pipeline Diagram

- Form showing resource usage



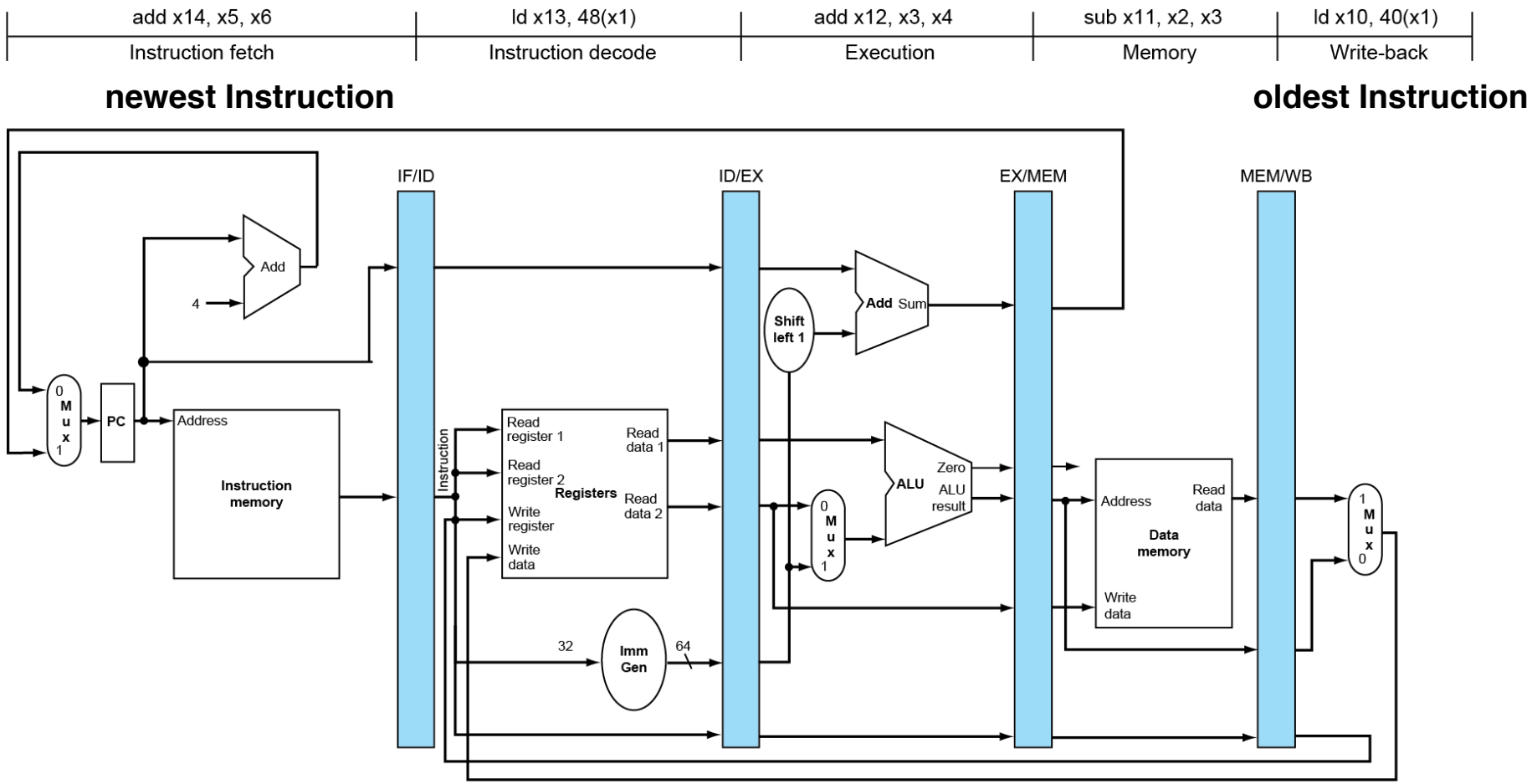
Multi-Cycle Pipeline Diagram

- Traditional form

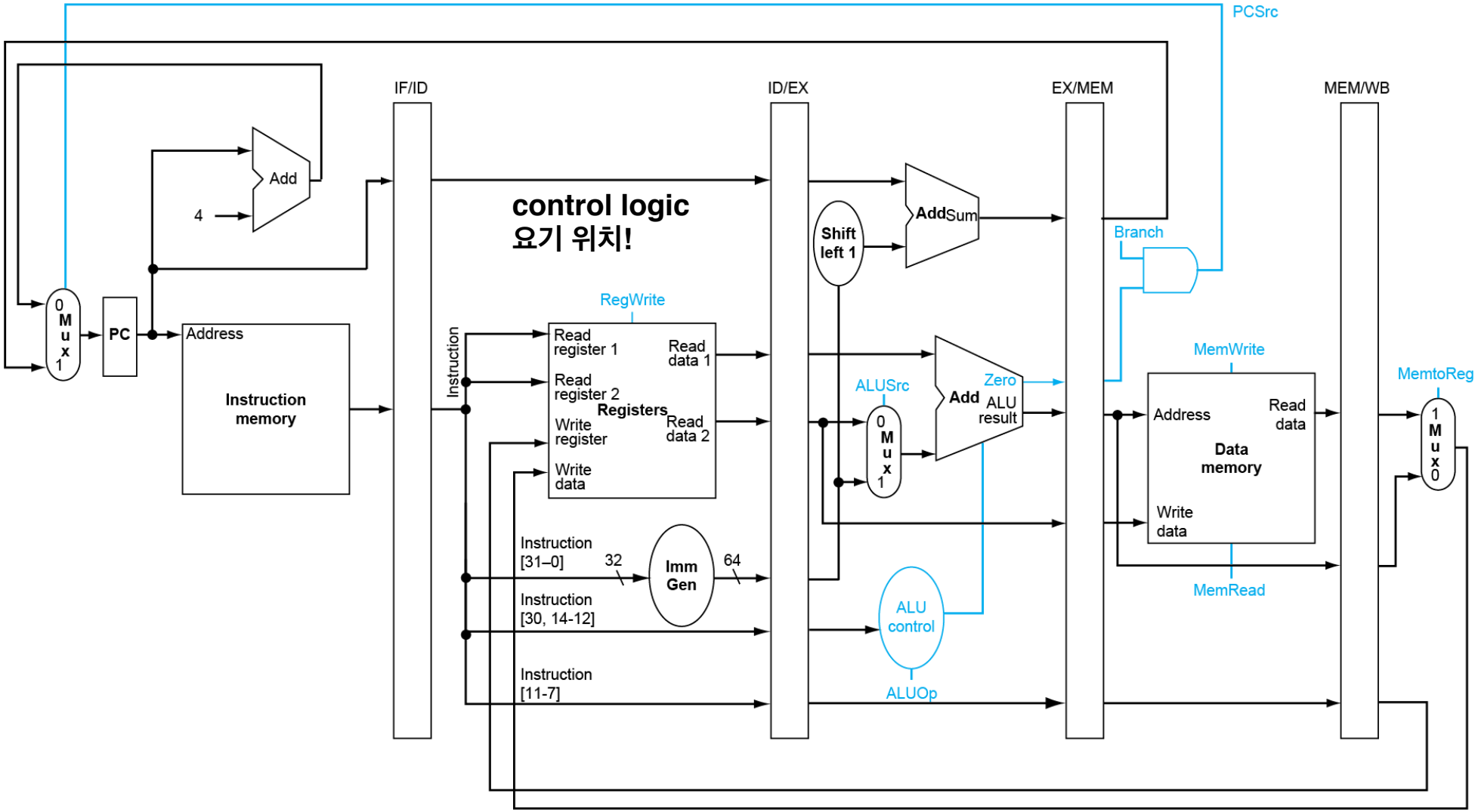


Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle



Pipelined Control (Simplified)

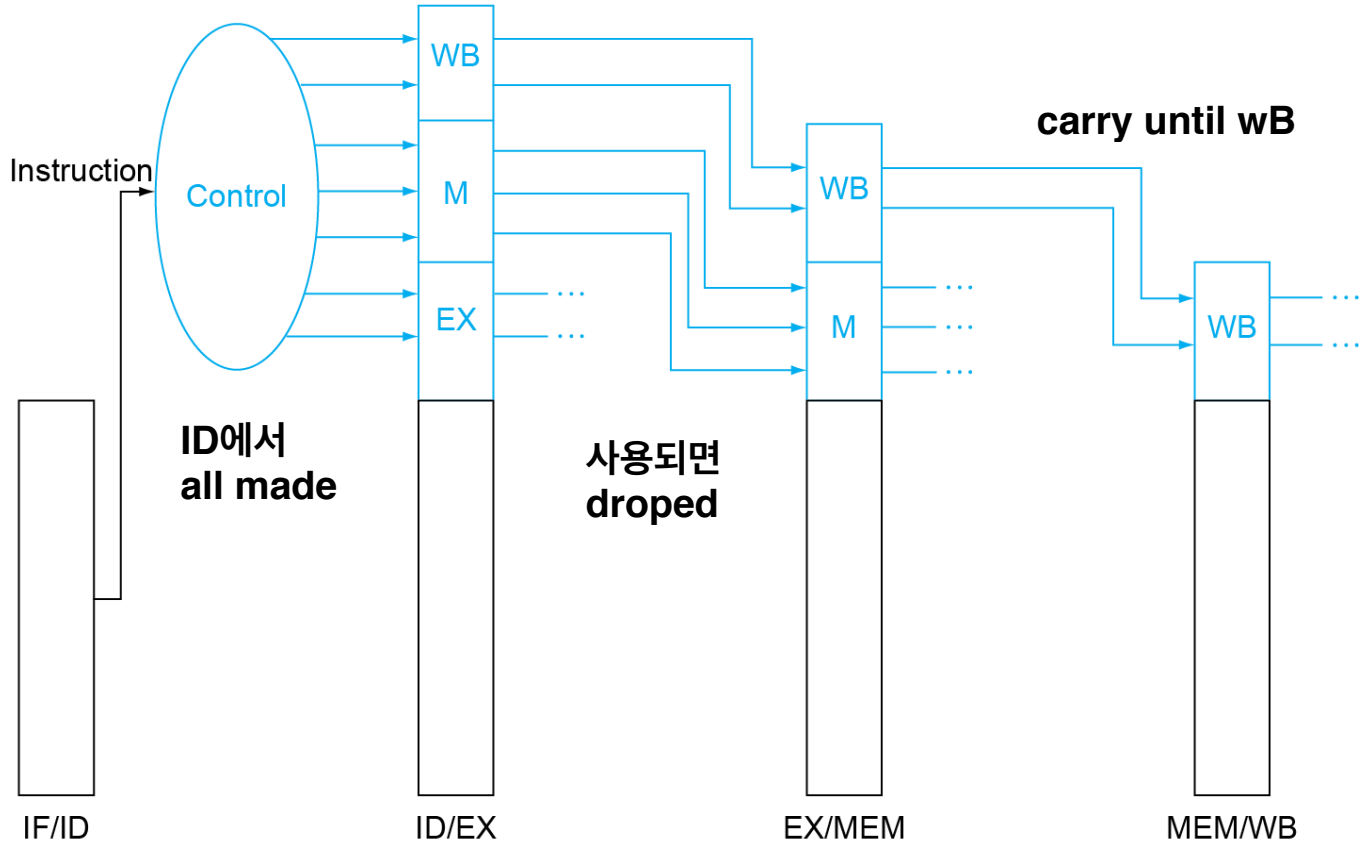


Generating Control Signals

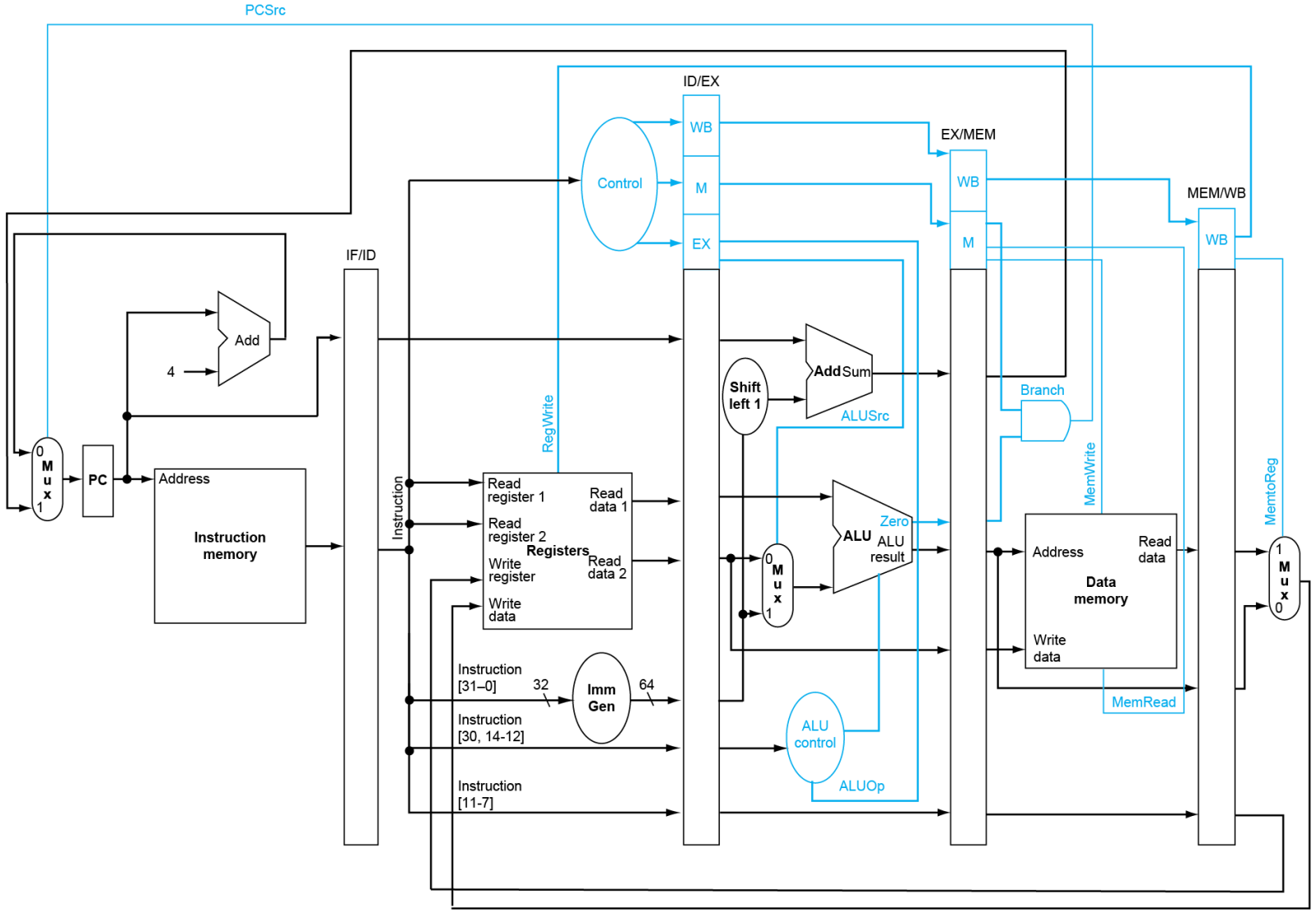
Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Pipelined Control

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



Pipelined Control



Data Hazards

Chap. 4.7

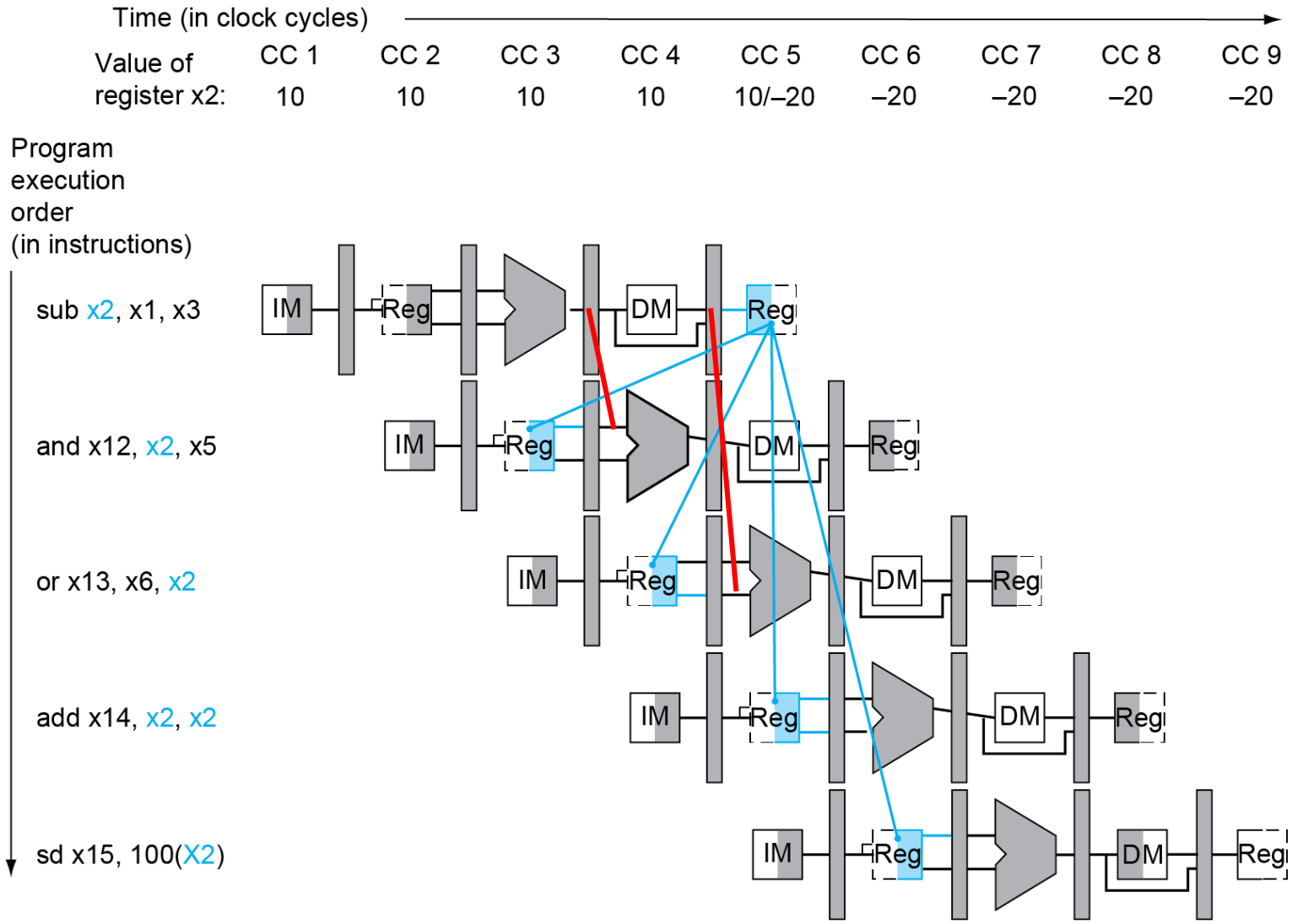
Data Hazards in ALU Instructions

- Consider this sequence:

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

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies and Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., `ID/EX.RegisterRs1` = register # for `Rs1` sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - `ID/EX.RegisterRs1`, `ID/EX.RegisterRs2`
- Data hazards when

des num of reg in the mem stage — src num in ex

```
EX/MEM.RegisterRd = ID/EX.RegisterRs1
EX/MEM.RegisterRd = ID/EX.RegisterRs2
```

Forward from EX/MEM pipeline register

des num of reg in the wb stage

```
MEM/WB.RegisterRd = ID/EX.RegisterRs1
MEM/WB.RegisterRd = ID/EX.RegisterRs2
```

Forward from MEM/WB pipeline register

Detecting the Need to Forward (cont'd)

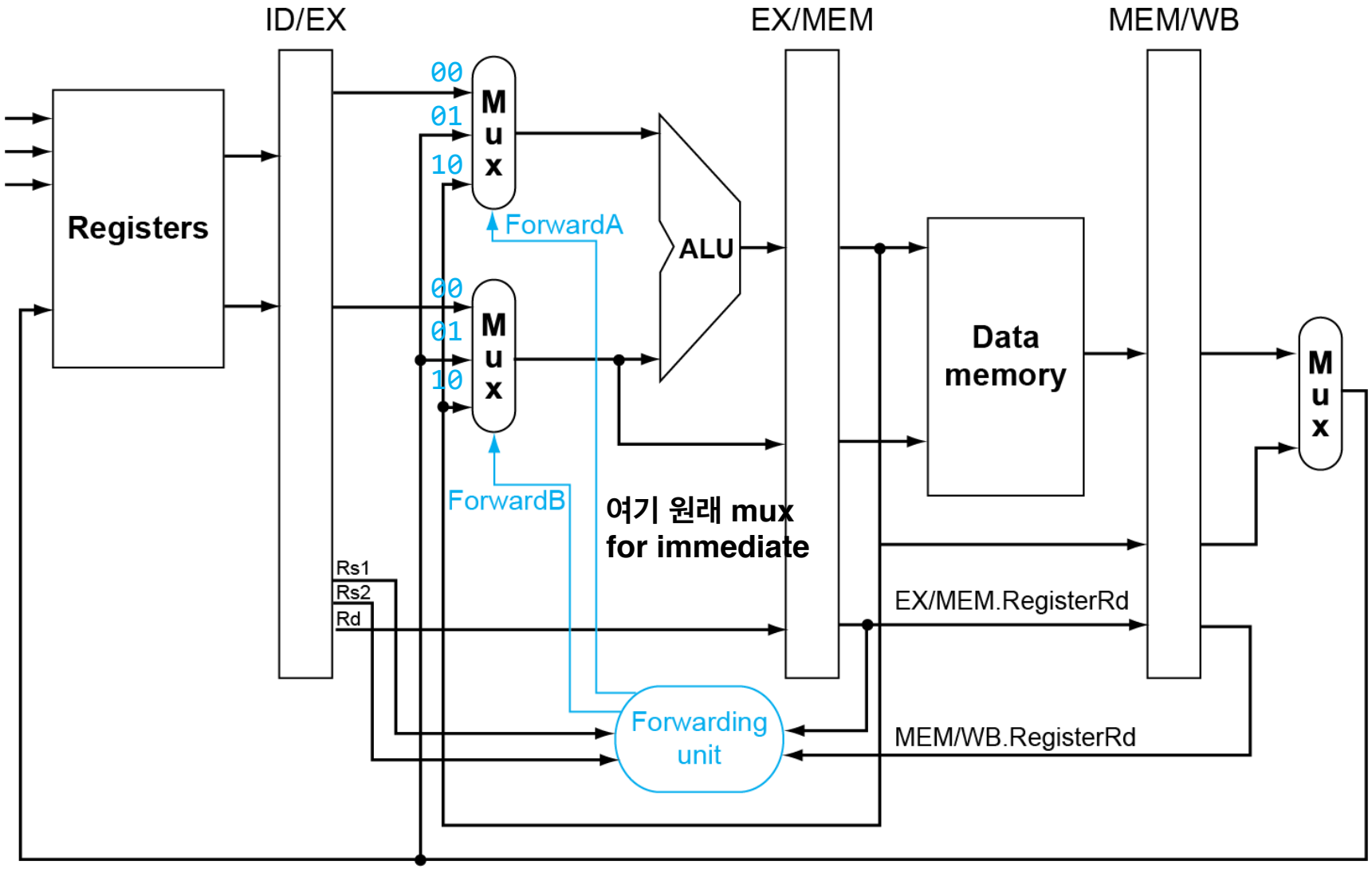
- But only if forwarding instruction will write to a register!

```
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



Forwarding Conditions

- EX hazard

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
   forwardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
   forwardB = 10
```

- MEM hazard

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
   forwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
   forwardB = 01
```

Forwarding Control

MUX control	Source	Example
`ForwardA = 00`	`ID/EX`	The first ALU operand comes from the register file.
`ForwardA = 10`	`EX/MEM`	The first ALU operand is forwarded from the prior ALU result.
`ForwardA = 01`	`MEM/WB`	The first ALU operand is forwarded from data memory or an earlier ALU result.
`ForwardB = 00`	`ID/EX`	The second ALU operand comes from the register file.
`ForwardB = 10`	`EX/MEM`	The second ALU operand is forwarded from the prior ALU result.
`ForwardB = 01`	`MEM/WB`	The second ALU operand is forwarded from data memory or an earlier ALU result.

Double Data Hazard

- Consider this 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 forward if EX hazard condition isn't true

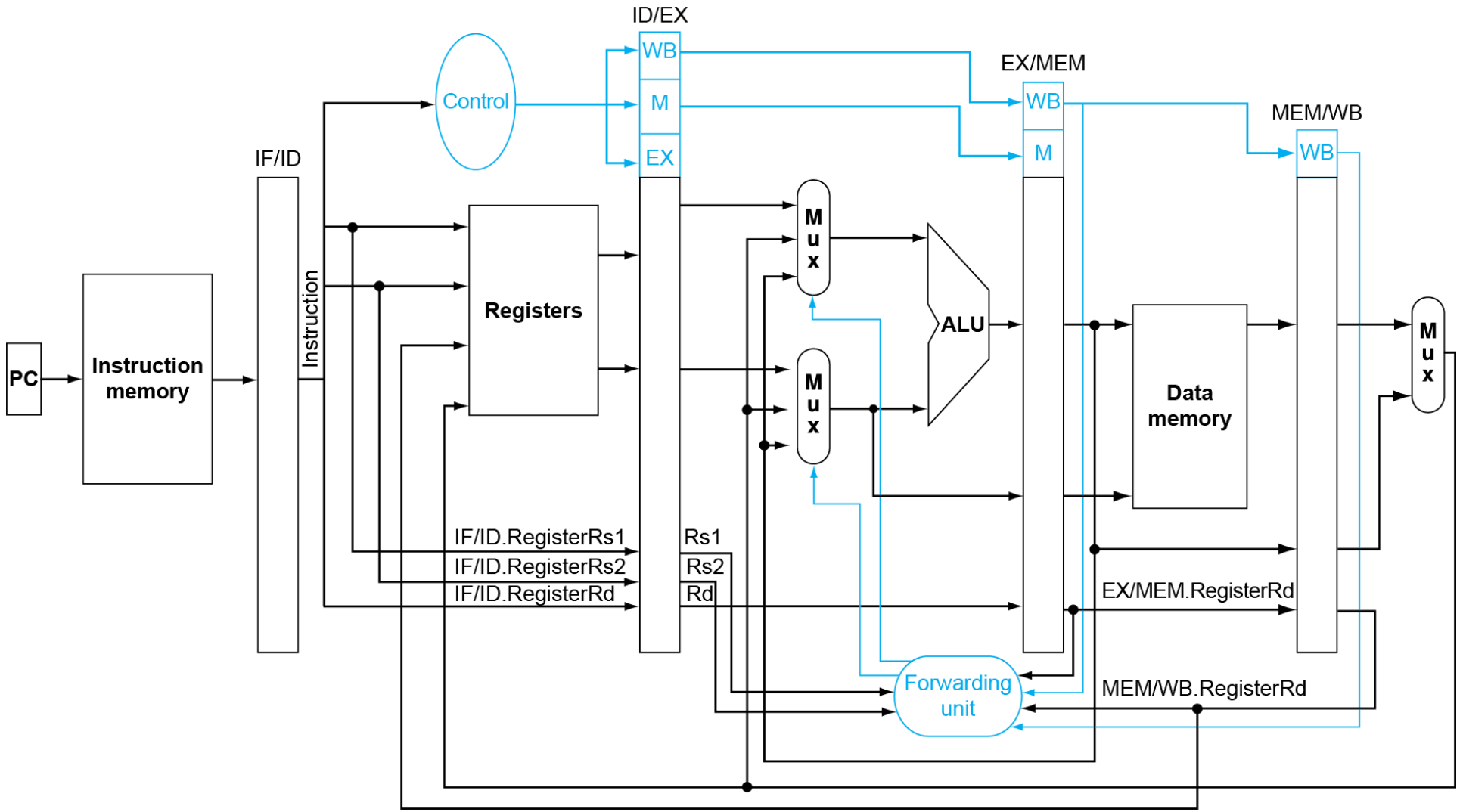
Revised Forwarding Conditions

- MEM hazard

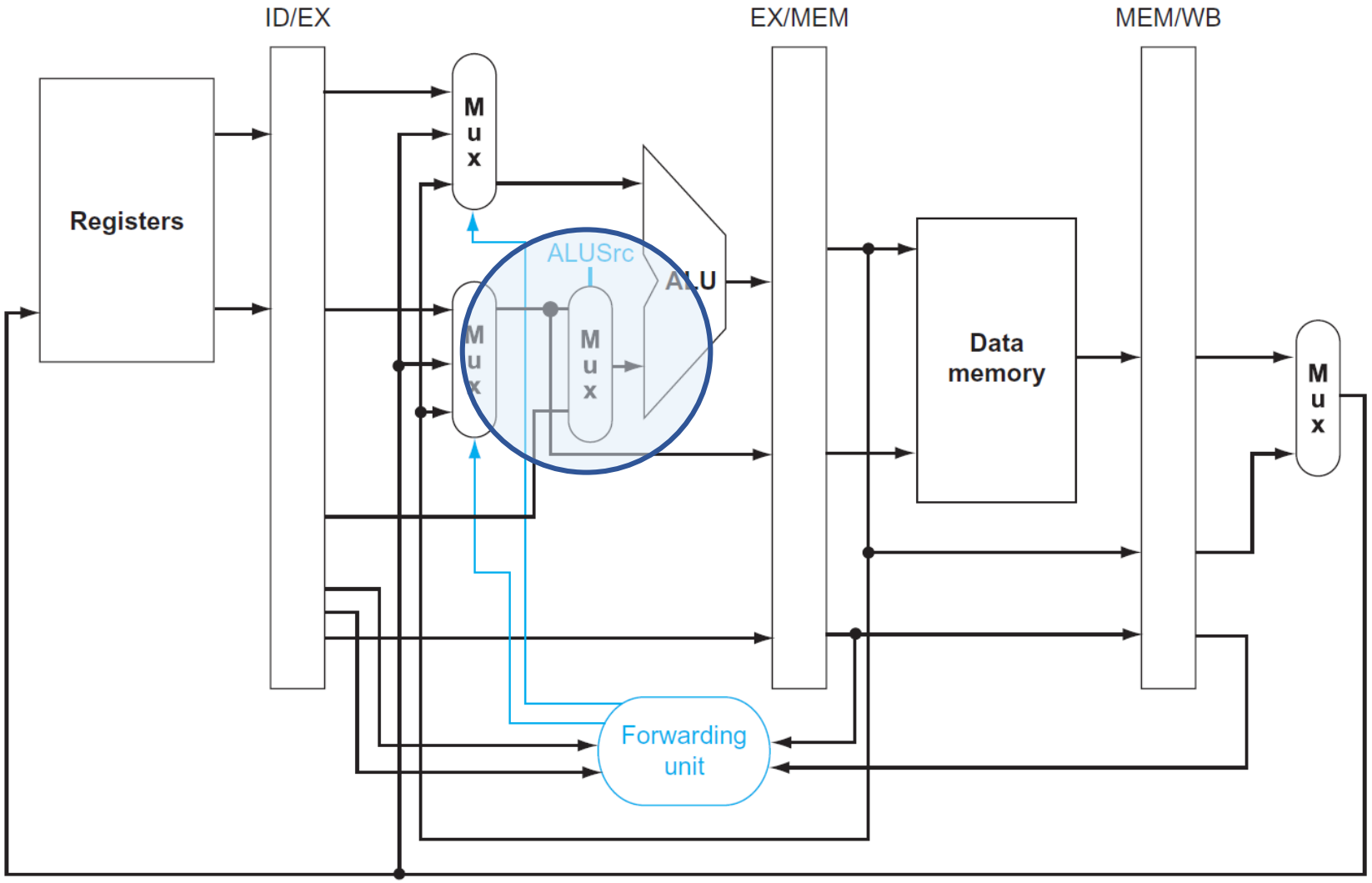
```
 if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    forwardA = 01

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.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    forward = 01
```

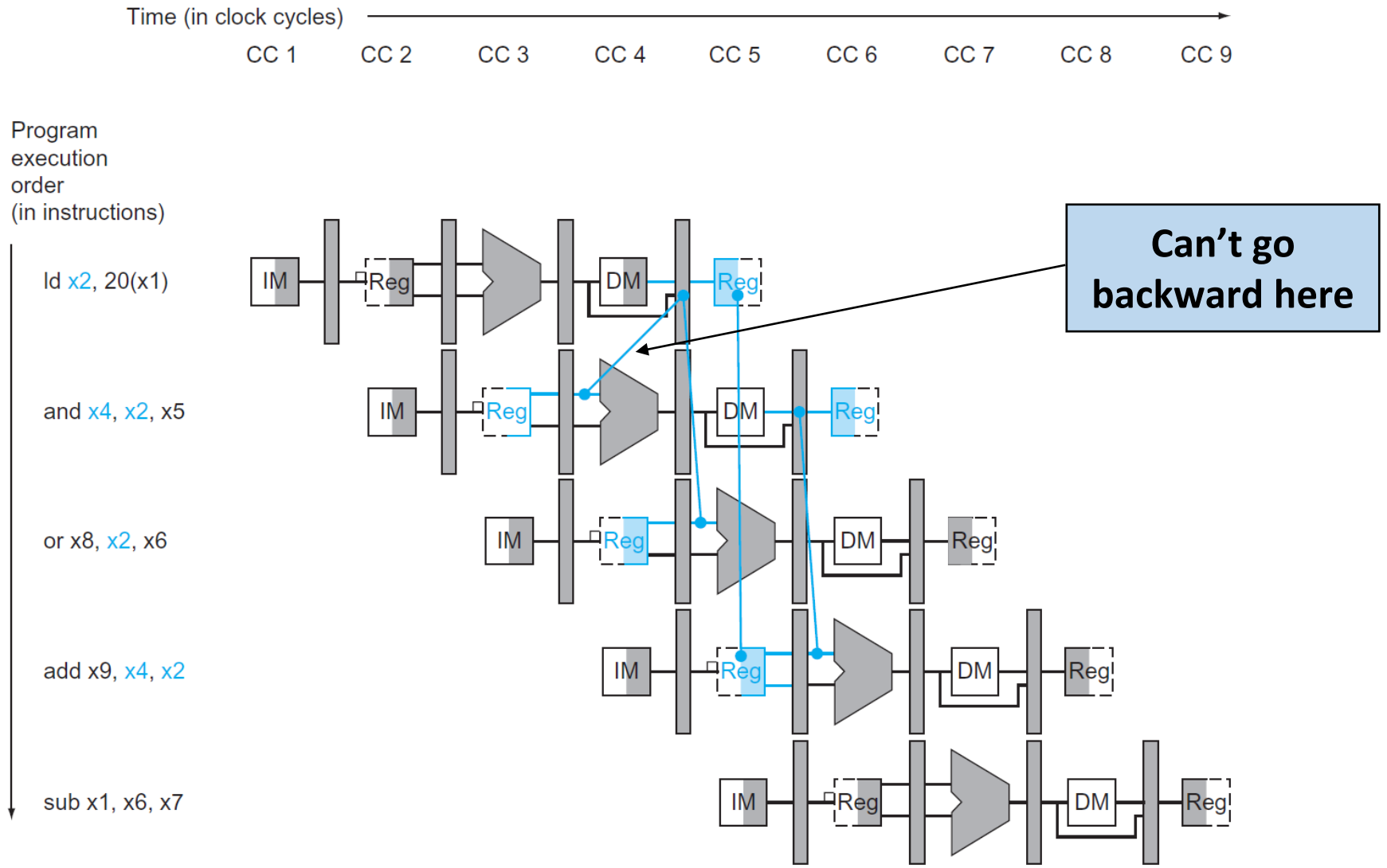
Datapath with Forwarding



Complete Datapath with Forwarding



Load-Use 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.RegisterRs1, IF/ID.RegisterRs2`
- Load-use hazard when

```
ID/EX.MemRead and
   ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs2))
```

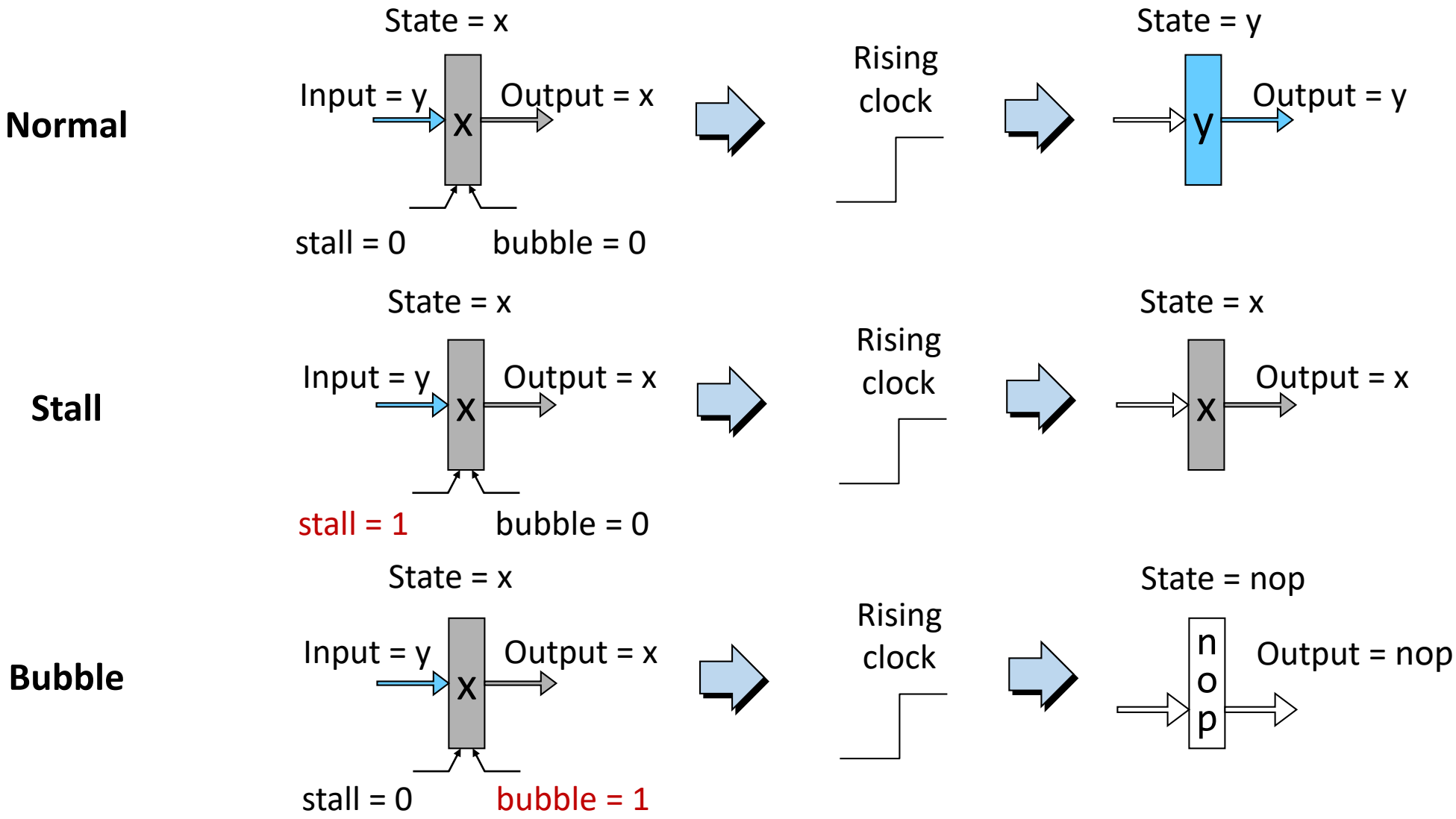
- If detected, stall and insert bubble

How to Stall the Pipeline

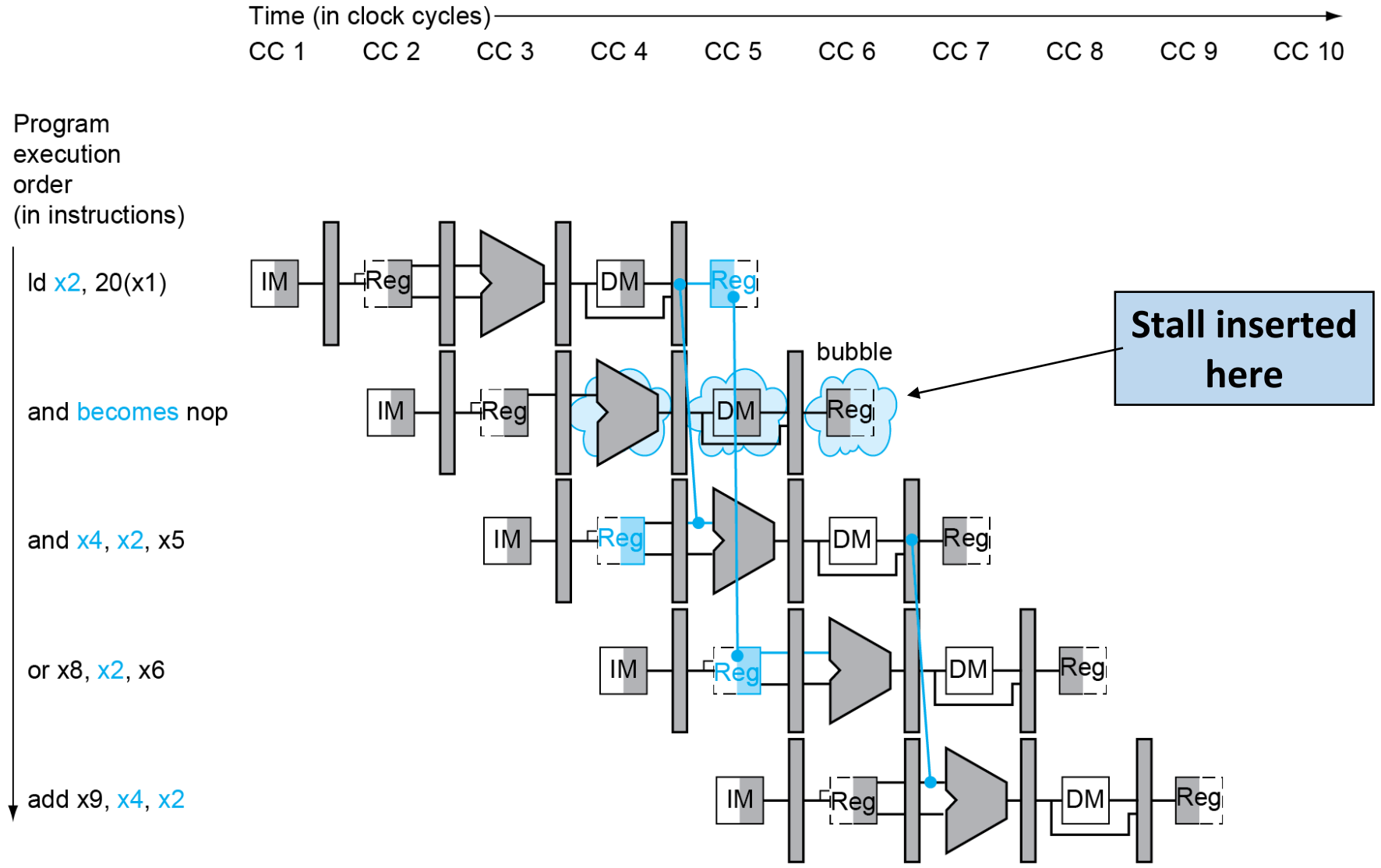
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)

- 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

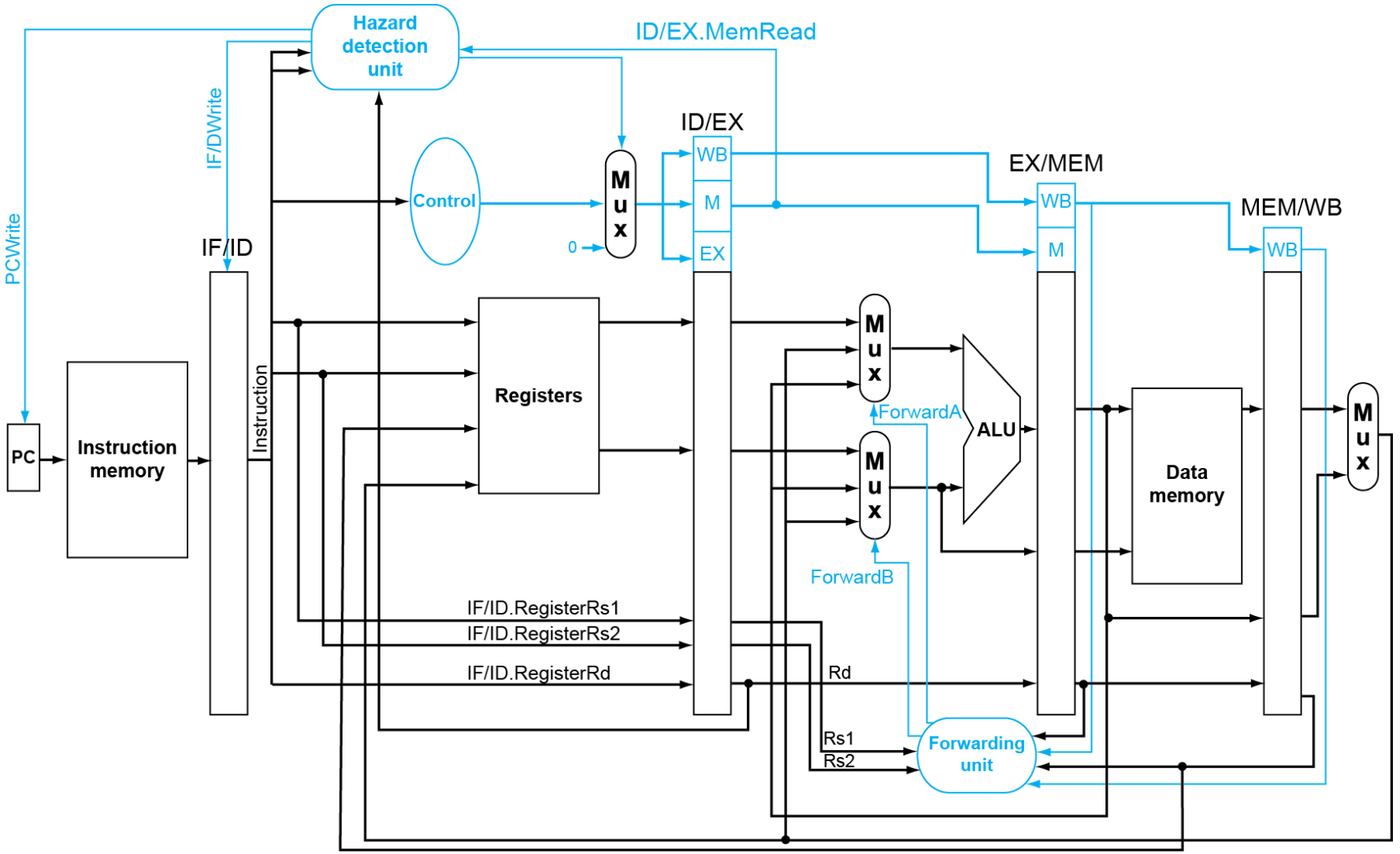
Pipeline Register Modes



Load-Use Data Hazard



Datapath with Hazard Detection



Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results

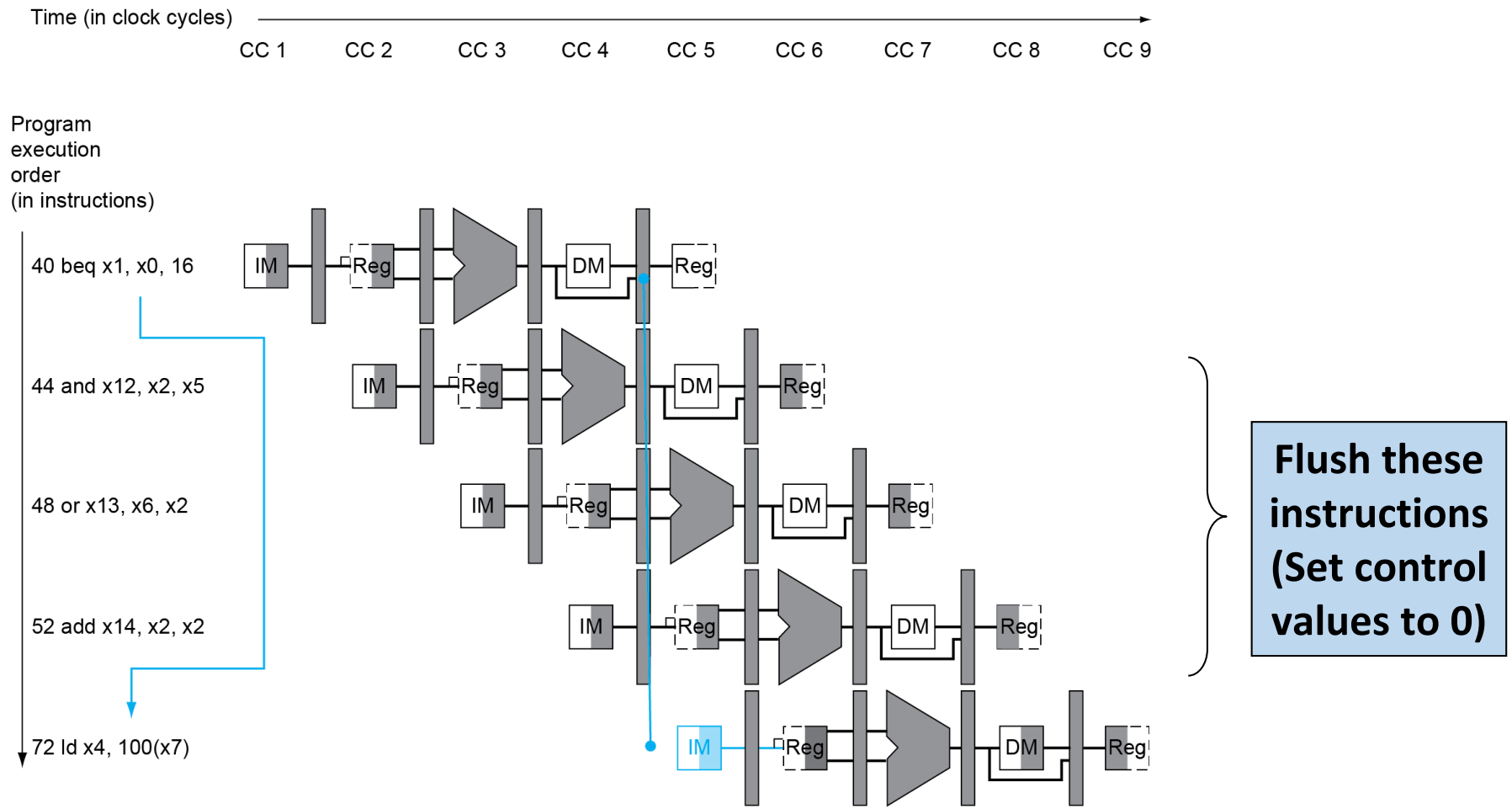
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards

Chap. 4.8

Control Hazards

- If branch outcome determined in MEM (with always-not-taken prediction)



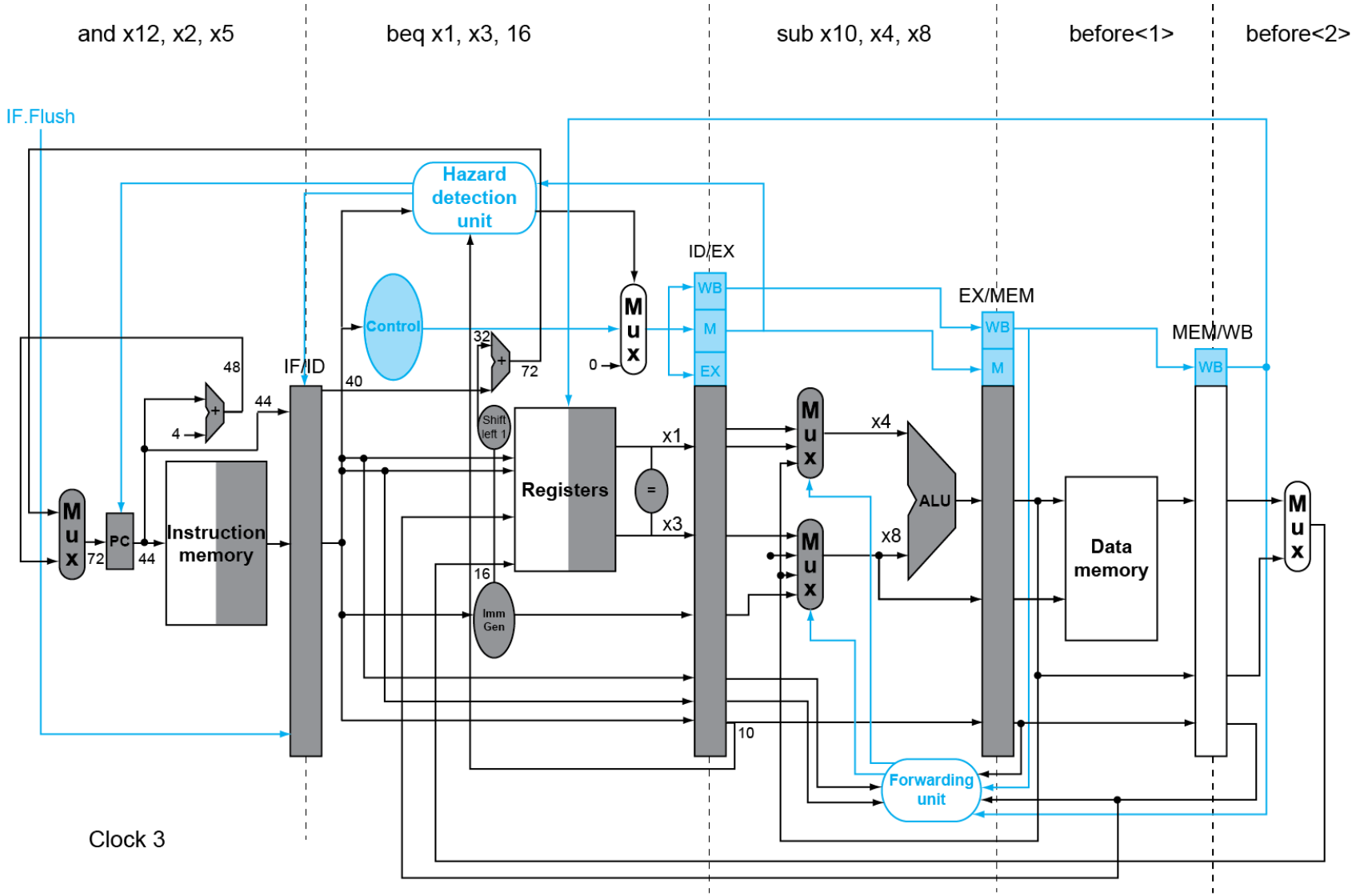
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator

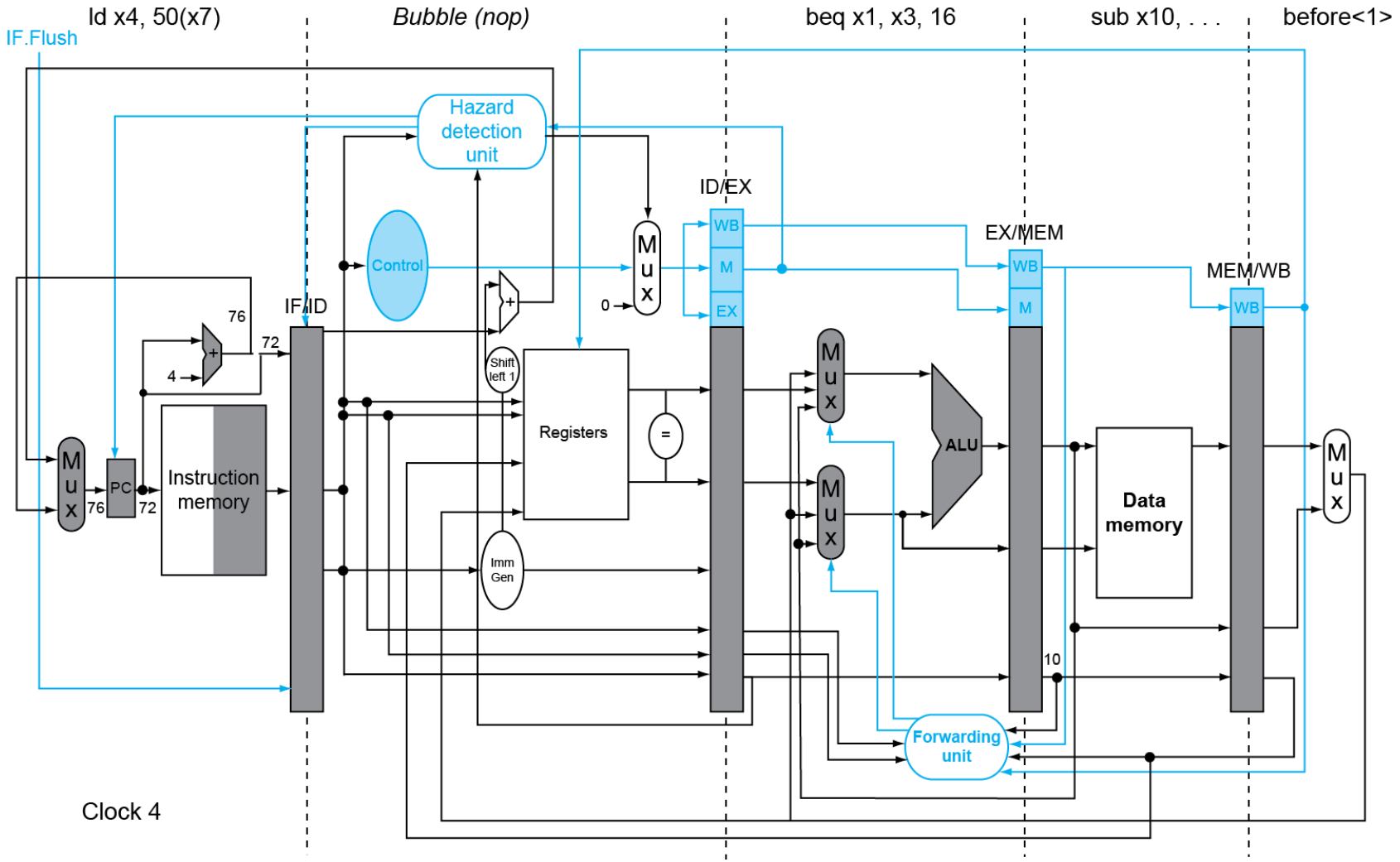
- Example: branch taken

```
36:    sub    x10, x4, x8
40:    beq    x1, x3, 16     // PC-relative branch to 40+16*2 = 72
44:    and    x12, x2, x5
48:    or     x13, x2, x6
52:    add    x14, x4, x2
56:    sub    x15, x6, x7
       ...
72:    ld     x4, 50(x7)
```

Example: Branch Taken



Example: Branch Taken (cont'd)

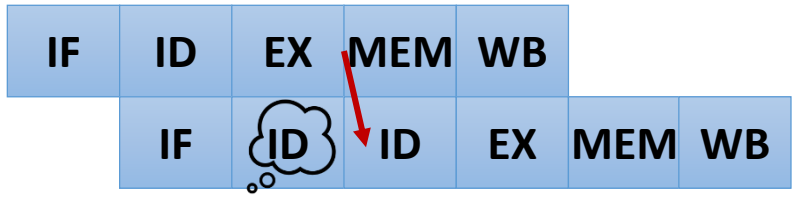


Another Cost of Branch Test in ID

- Register operands may require forwarding
  - New forwarding logic from EX/MEM or MEM/WB pipeline registers to ID needed

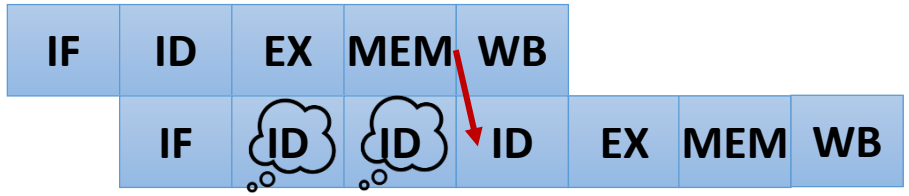
- Stalls due to data hazard
  - 1-cycle stall if the preceding instruction is an ALU instruction

```
add  x3,x4,x5
beq  x8,x3,10
```



  - 2-cycle stall if the preceding instruction is the load instruction

```
ld   x3,0(x4)
beq  x8,x3,10
```



Dynamic Branch Prediction

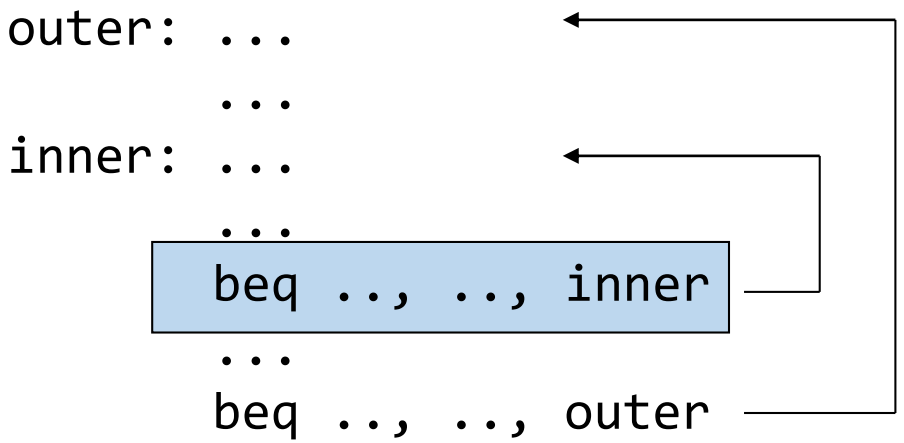
- In deeper and superscalar pipelines, branch penalty is more significant

- Use dynamic prediction
  - Branch prediction buffer (or 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

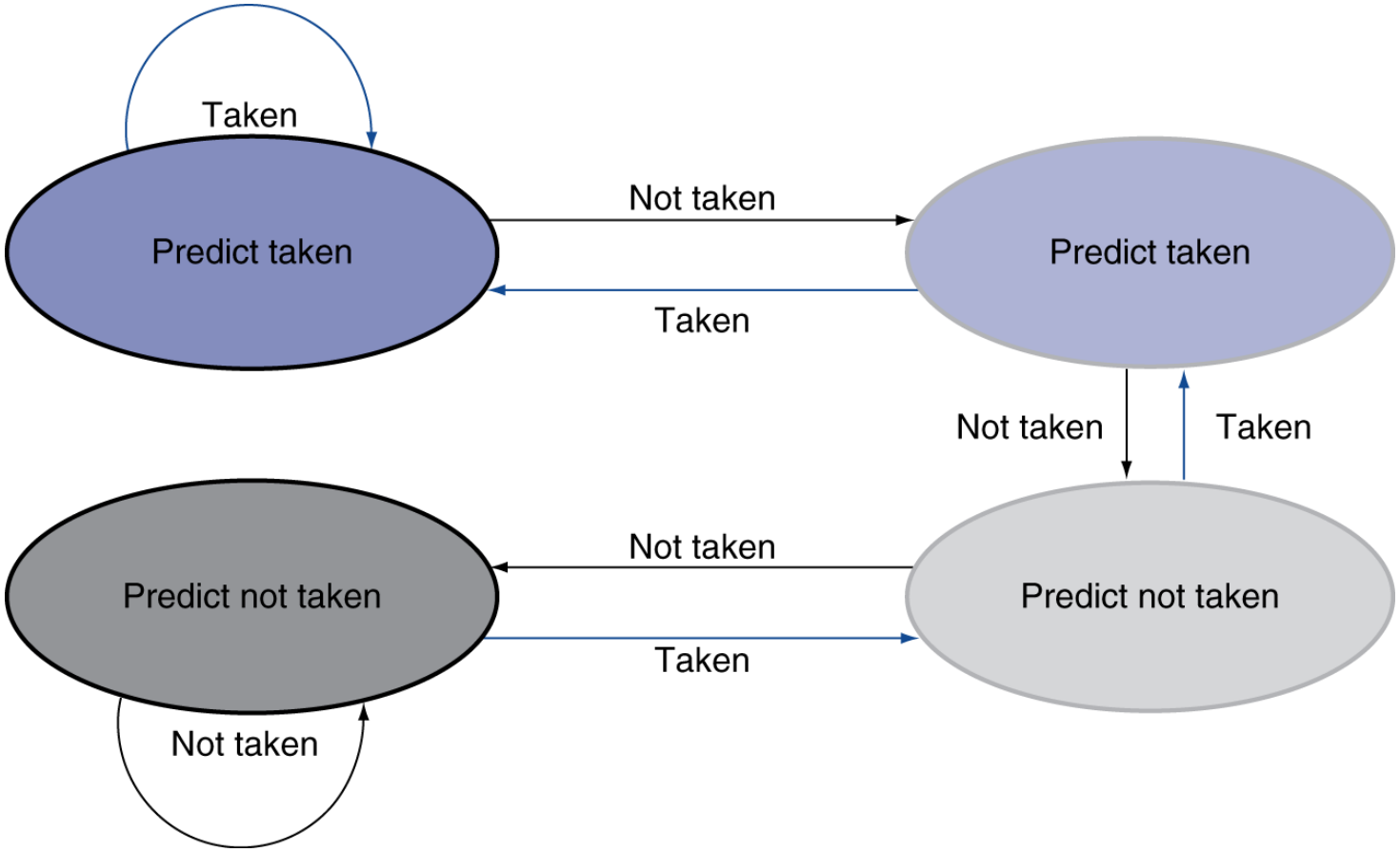
- Inner loop branches mispredicted twice!



- 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 Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch

- Branch target buffer (BTB)
  - Cache of target addresses
  - Indexed by PC when instruction fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately

Exceptions

Chap. 4.9

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - RISC-V: any unexpected change in control flow (either internal or external)
  - Internal exception arises within the CPU (e.g., undefined opcode, syscall, …)
- Interrupt
  - RISC-V: event from an external I/O controller
  - e.g., hard disks, network adapters, keyboard, …
- Dealing with them without sacrificing performance is hard

Handling Exceptions in RISC-V

- Save PC of offending (or interrupted) instruction
  - Supervisor Exception Program Counter (SEPC)

- Save indication of the problem
  - Supervisor Exception Cause Register (SCAUSE)
  - 64-bits, but most bits unused
  - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …

- Jump to handler
  - Assume at `0x0000 0000 1C09 0000`

An Alternate Mechanism

- Vectored interrupts
  - Handler address determined by the cause

- Exception vector address to be added to a vector table base register:
  - Undefined opcode: 00 0100 0000
  - Hardware malfunction: 01 1000 0000

- 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 SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …

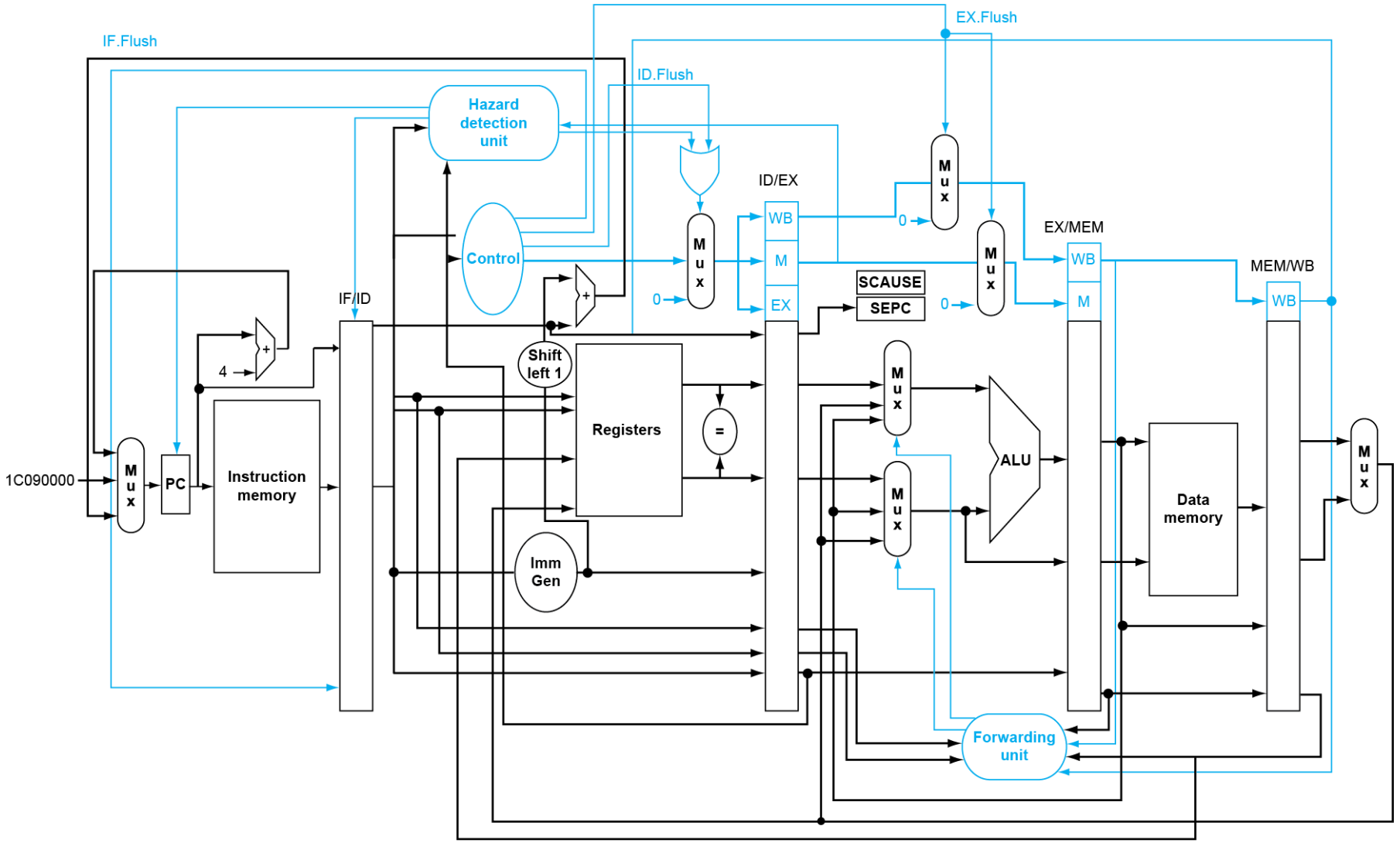
Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage

```
add   x1, x2, x1
```

  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set SEPC and SCAUSE 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 SEPC register
  - Identifies causing instruction

Exception Example (1)

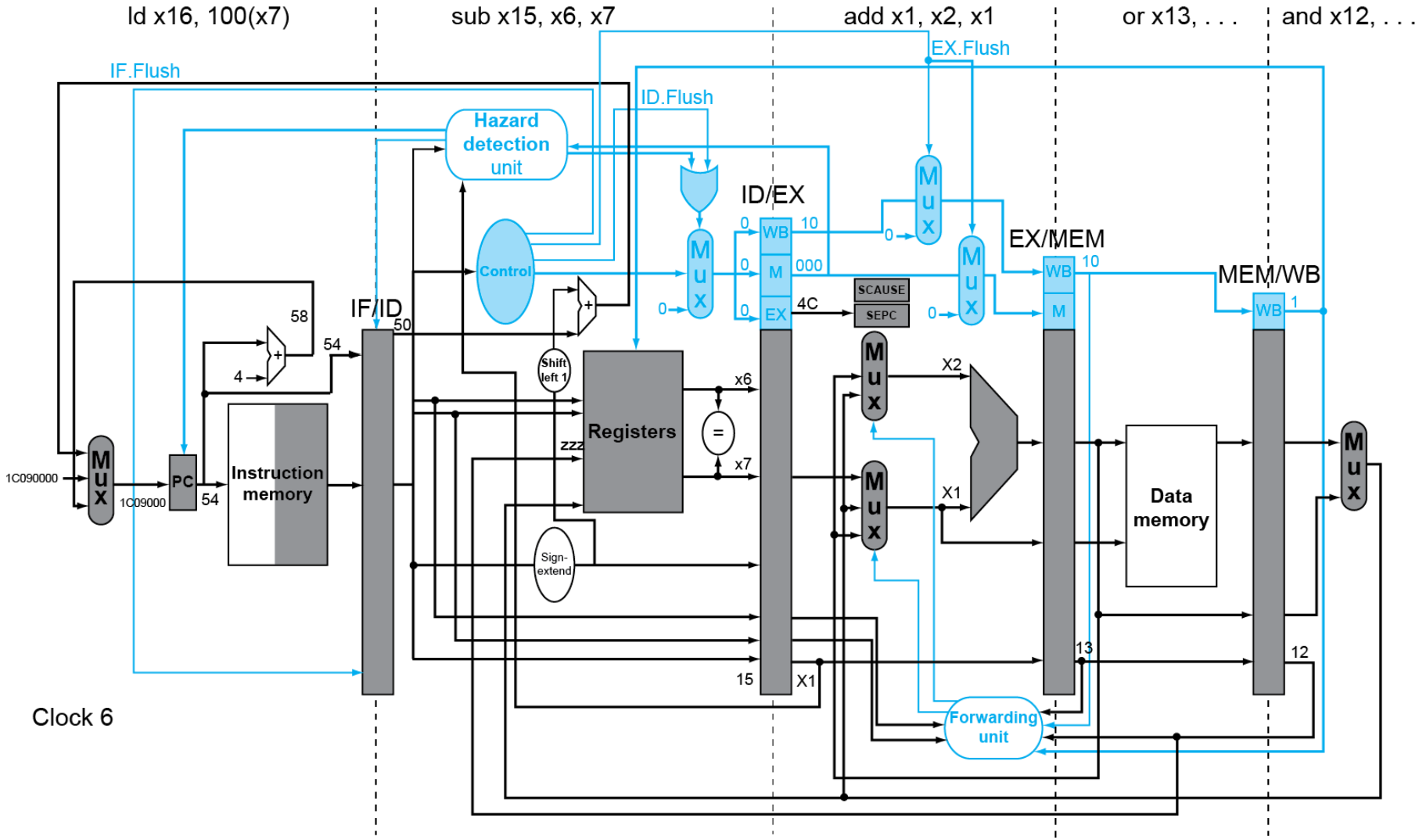
- Exception on add in

```
40:    sub    x11, x2, x4
44:    and    x12, x2, x5
48:    or     x13, x2, x6
4c:    add    x1, x2, x1
50:    sub    x15, x6, x7
54:    ld     x16, 100(x7)
```

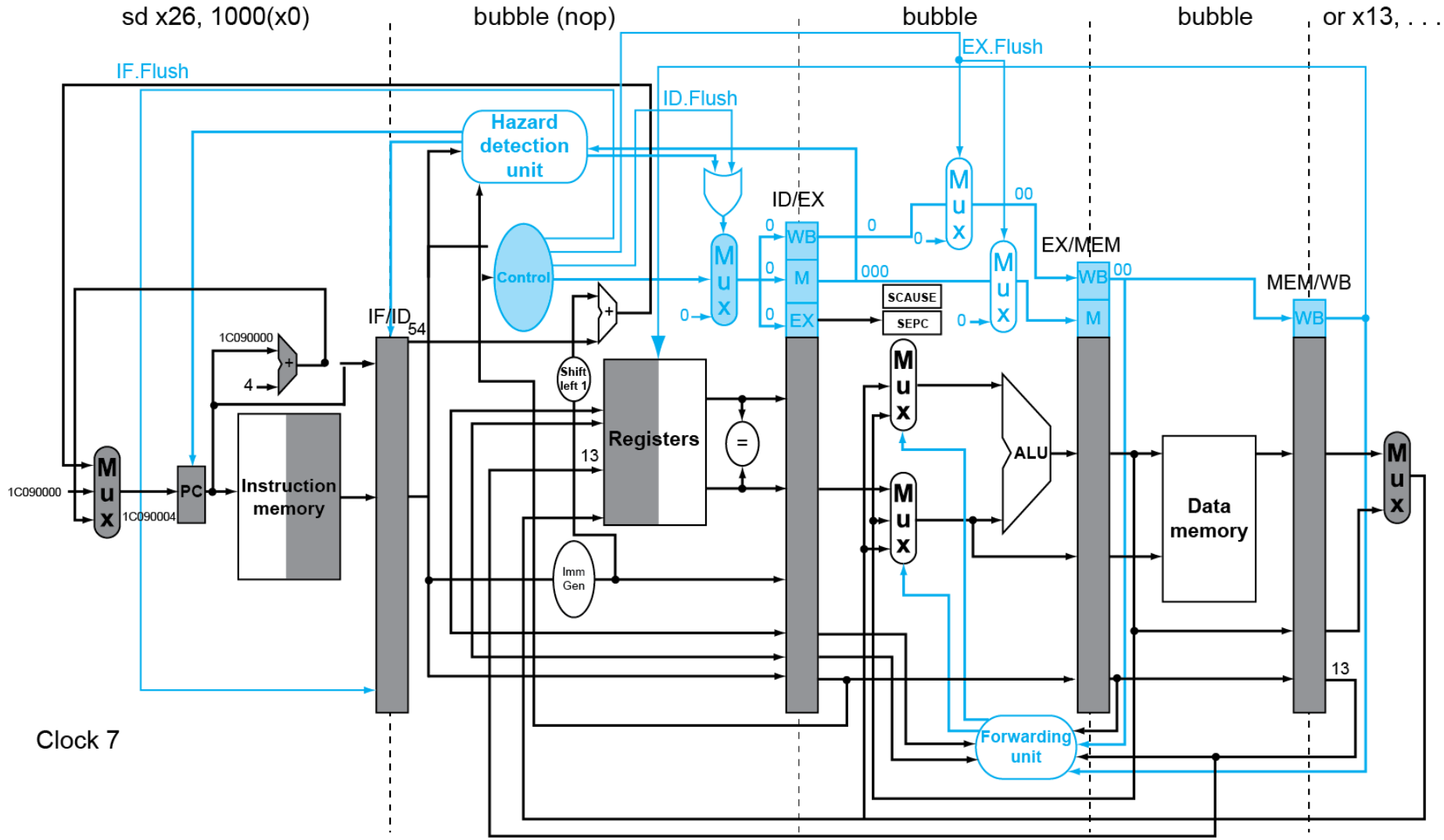
- Handler

```
1c090000       sd    x26, 1000(x10)
1c090004       sd    x27, 1008(x10)
...
```

Exception Example (2)



Exception Example (3)



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