Pipeline Hazards

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

Systems Software &
Architecture Lab.

Seoul National University

Fall 2020

Hazards

- Situations that prevent starting the next instruction in the next cycle

- Structural hazard
  - A required resource is busy

- Data hazard
  - Need to wait (or *stall*) for previous instruction to complete its data read/write

- Control hazard
  - Deciding on control action depends on previous instruction

Structural Hazard

- Conflict for use of a resource

- In RISC-V 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)

- Register file also requires multiple ports (for 2 reads and 1 write)
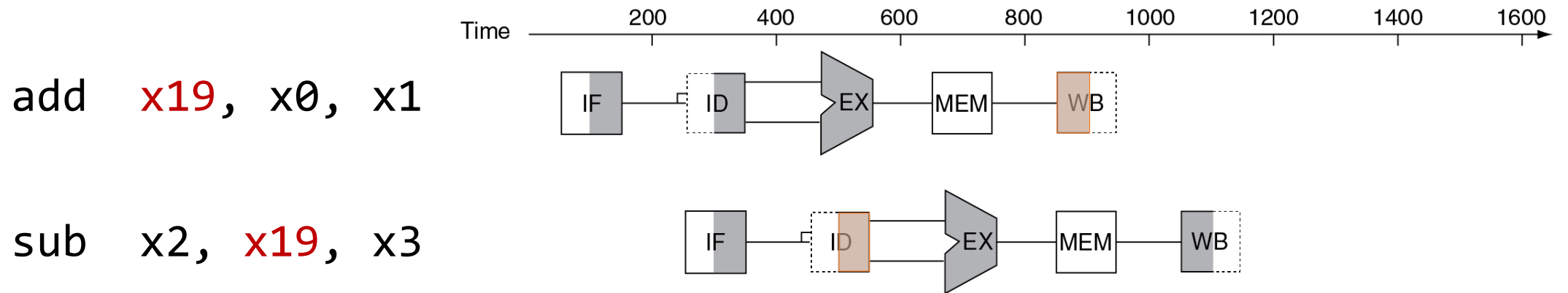
IF	ID	EX	MEM	WB	**Register write**
	IF	ID	EX	MEM	WB	**Memory read/write**
		IF	ID	EX	MEM	WB
Register read		IF	ID	EX	MEM	WB
Memory read			IF	ID	EX	MEM	WB

Data Hazards

Chap. 4.7

Data Hazard

- An instruction depends on completion of data access by a previous instruction

- Also called "Read-After-Write (RAW)" hazard

- This hazard results from an actual need for communication
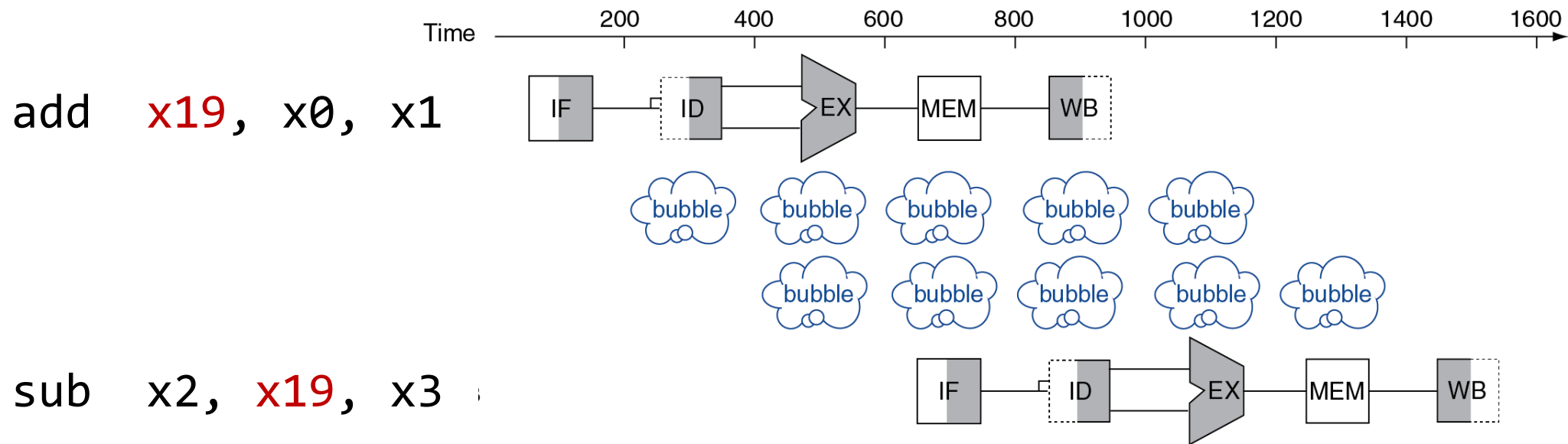


```
add   x19, x0, x1

sub   x2, x19, x3
```

Solutions to Data Hazard

- Freezing the pipeline

- Forwarding

- Compiler scheduling

- Out-Of-Order execution (discussed later)

Freezing the Pipeline

- Stall the pipeline until dependences are resolved
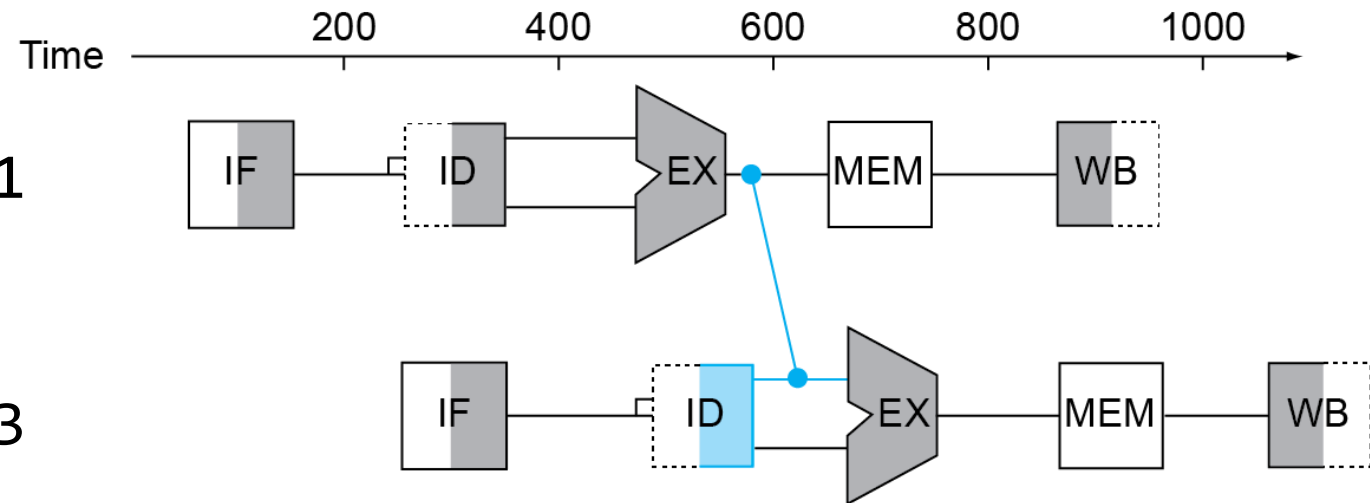
- ALU result to next instruction (2 stalls)

Forwarding (or Bypassing)

- **Use result when it is computed**
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

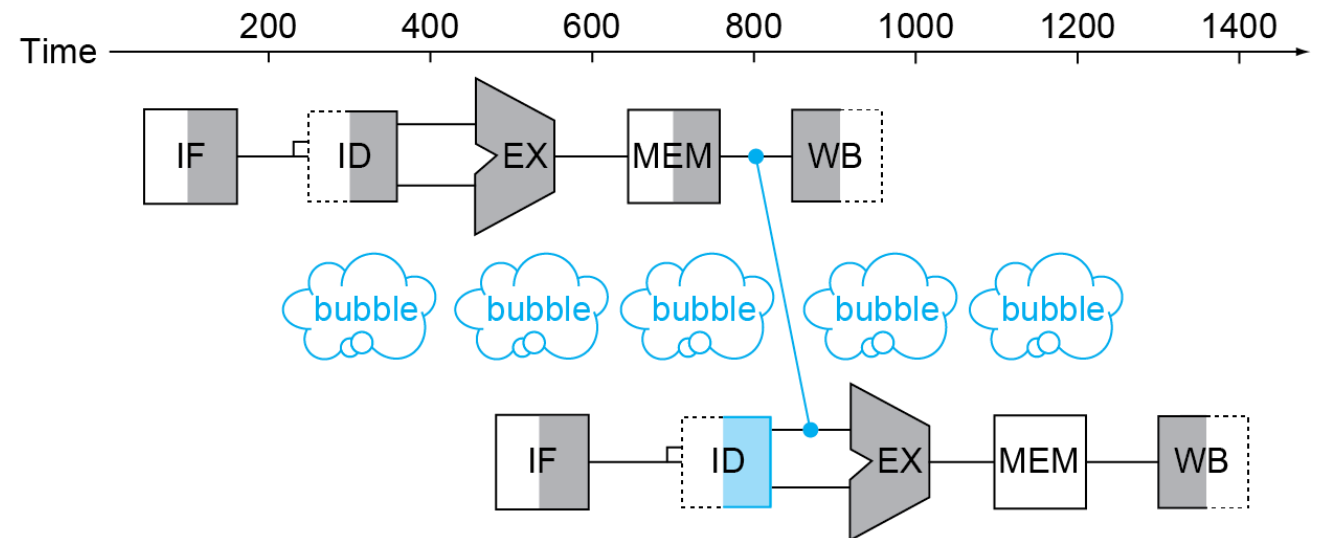# Forwarding: Load-Use Data Hazard

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

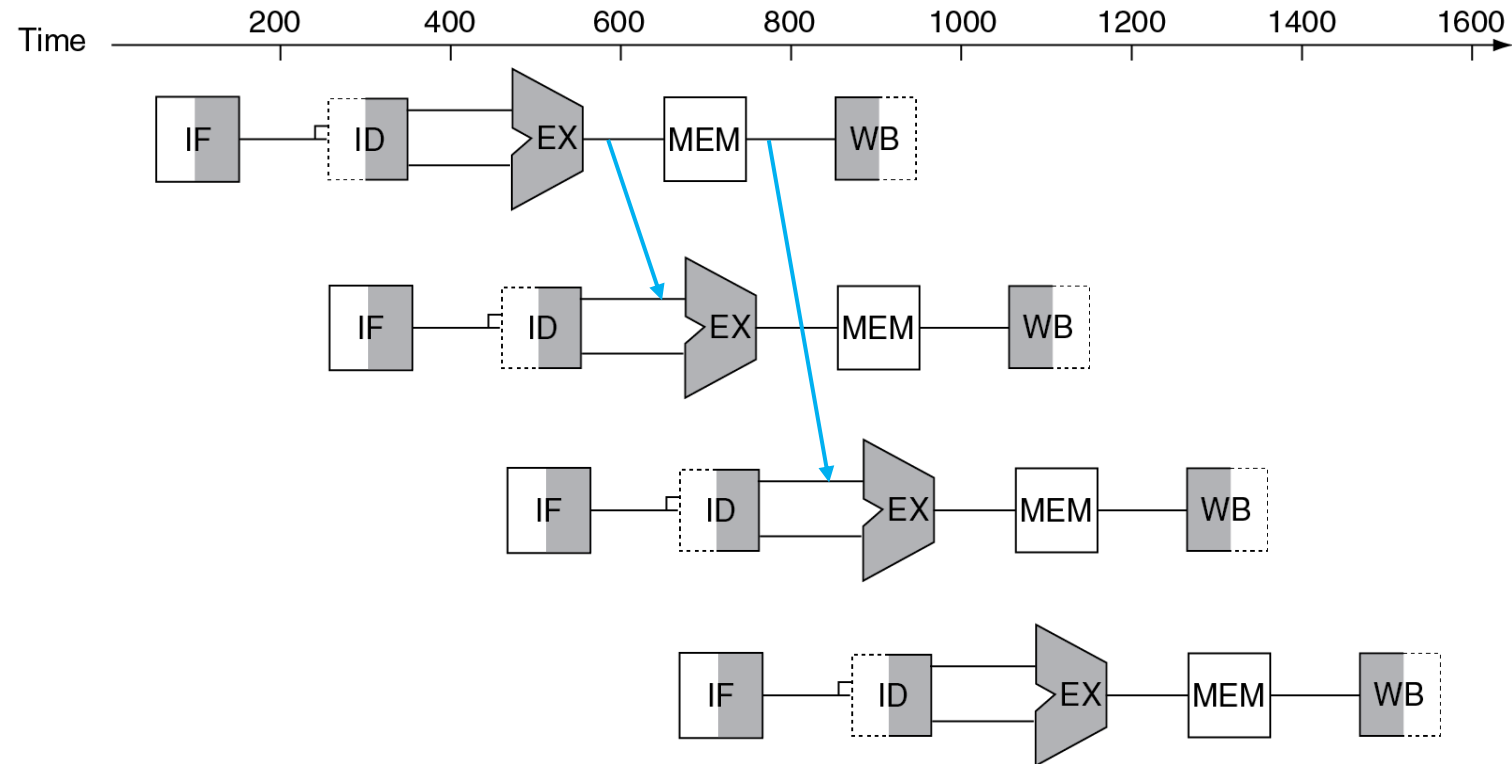# Forwarding: Multiple Readers

add x10, x4, x5

sub x6, x10, x4

and x7, x10, x0

xor x8, x10, x3

Forwarding: Multiple Writers

add   x10, x4, x5

sub   x10, x5, x6

addi x10, x2, 1

xor   x8, x10, x7
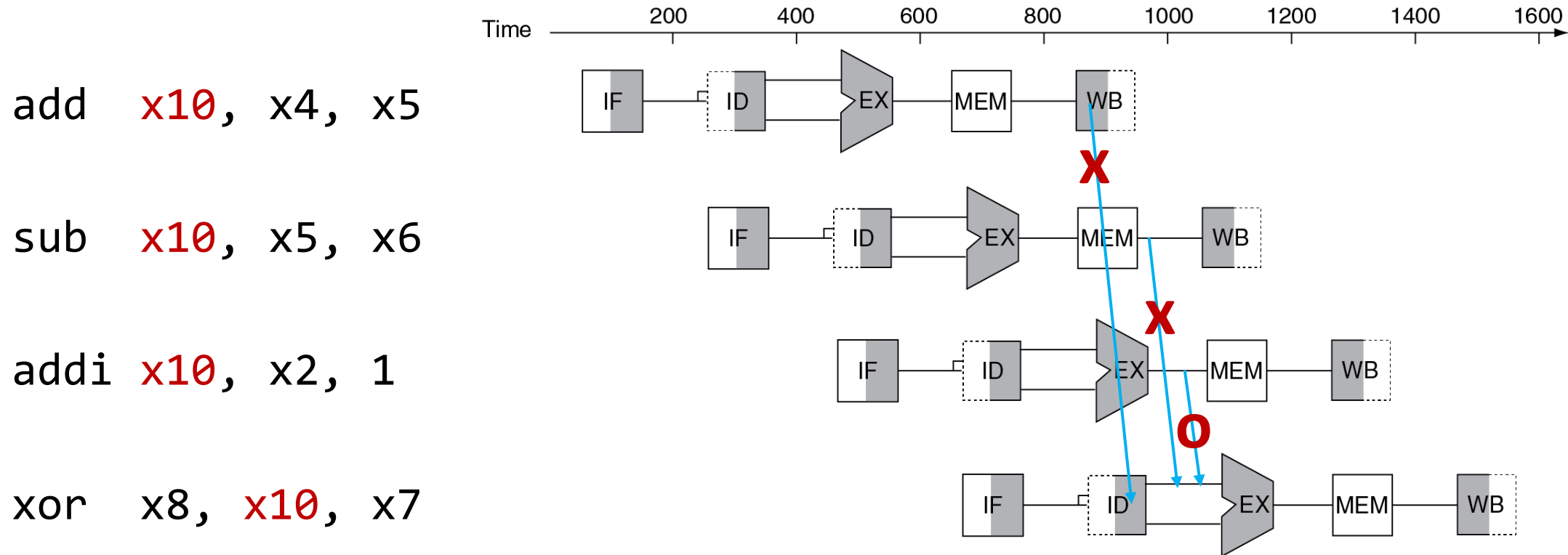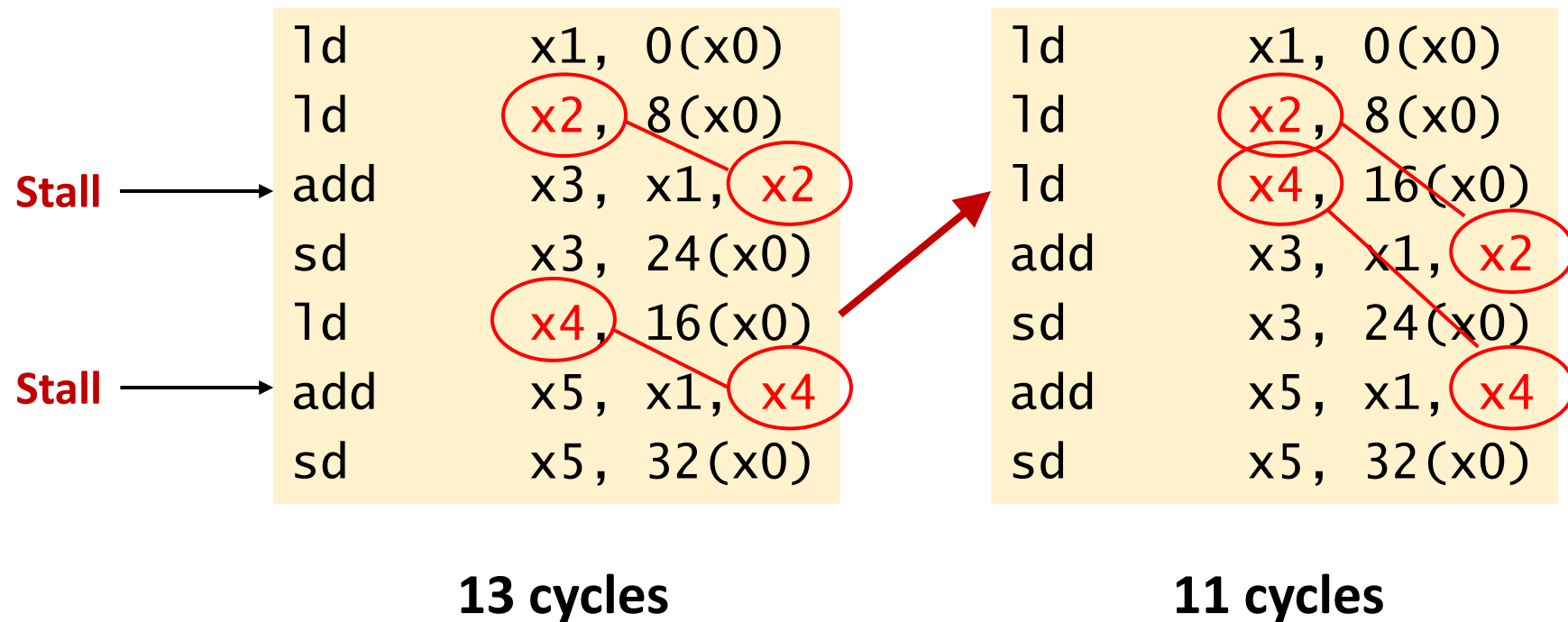
Compiler Scheduling

- Reorder code to avoid use of load result in the next instruction
- C code for `v[3] = v[0] + v[1];  v[4] = v[0] + v[2];`

```
ld      x1, 0(x0)
ld      x2, 8(x0)
add     x3, x1, x2
sd      x3, 24(x0)
ld      x4, 16(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

Stall → add
Stall → add

13 cycles

```
ld      x1, 0(x0)
ld      x2, 8(x0)
ld      x4, 16(x0)
add     x3, x1, x2
sd      x3, 24(x0)
add     x5, x1, x4
sd      x5, 32(x0)
```

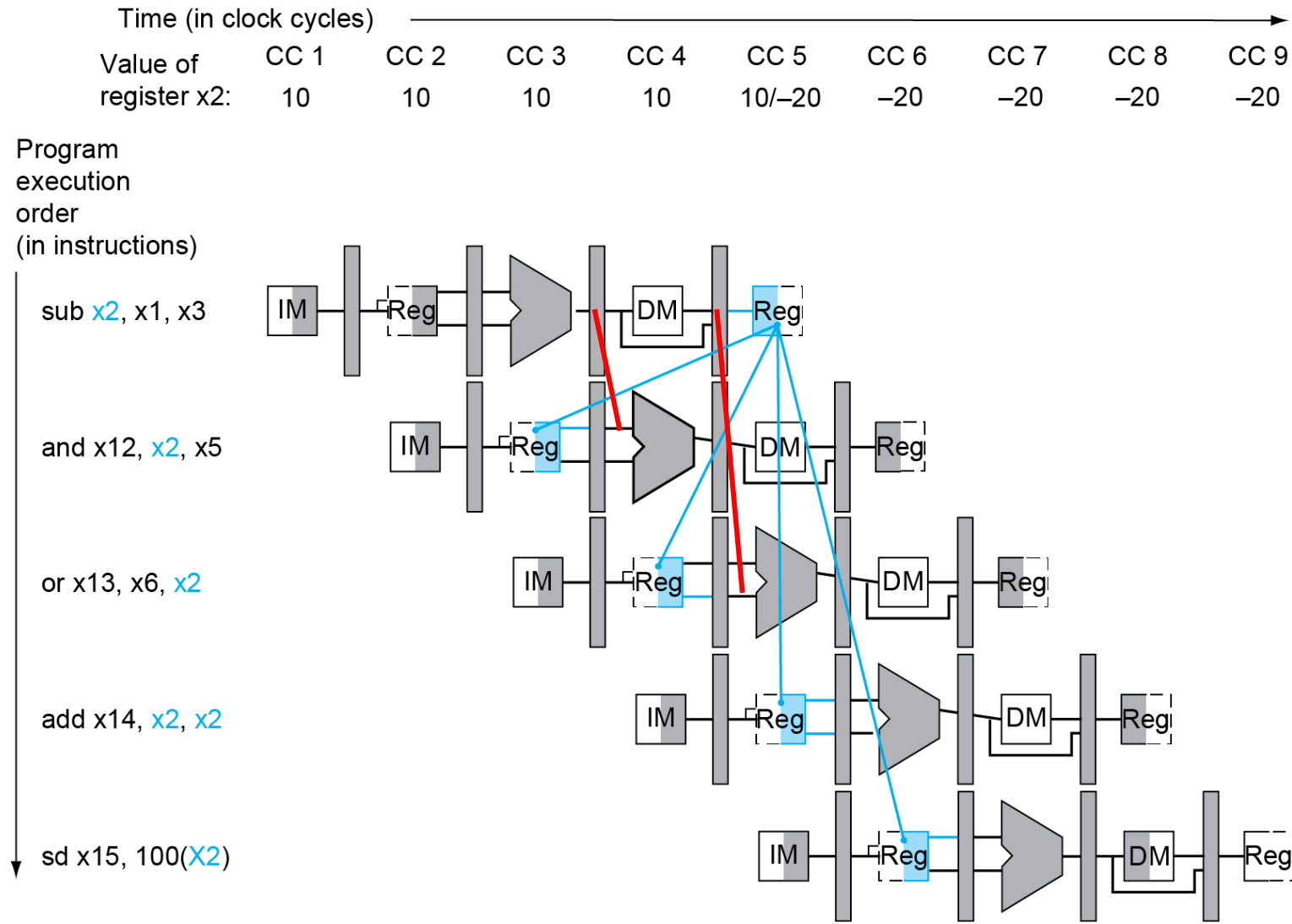**11 cycles**

Data Hazards in RISC-V

- 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

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


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

Forward from EX/MEM pipeline register

Forward from MEM/WB pipeline register
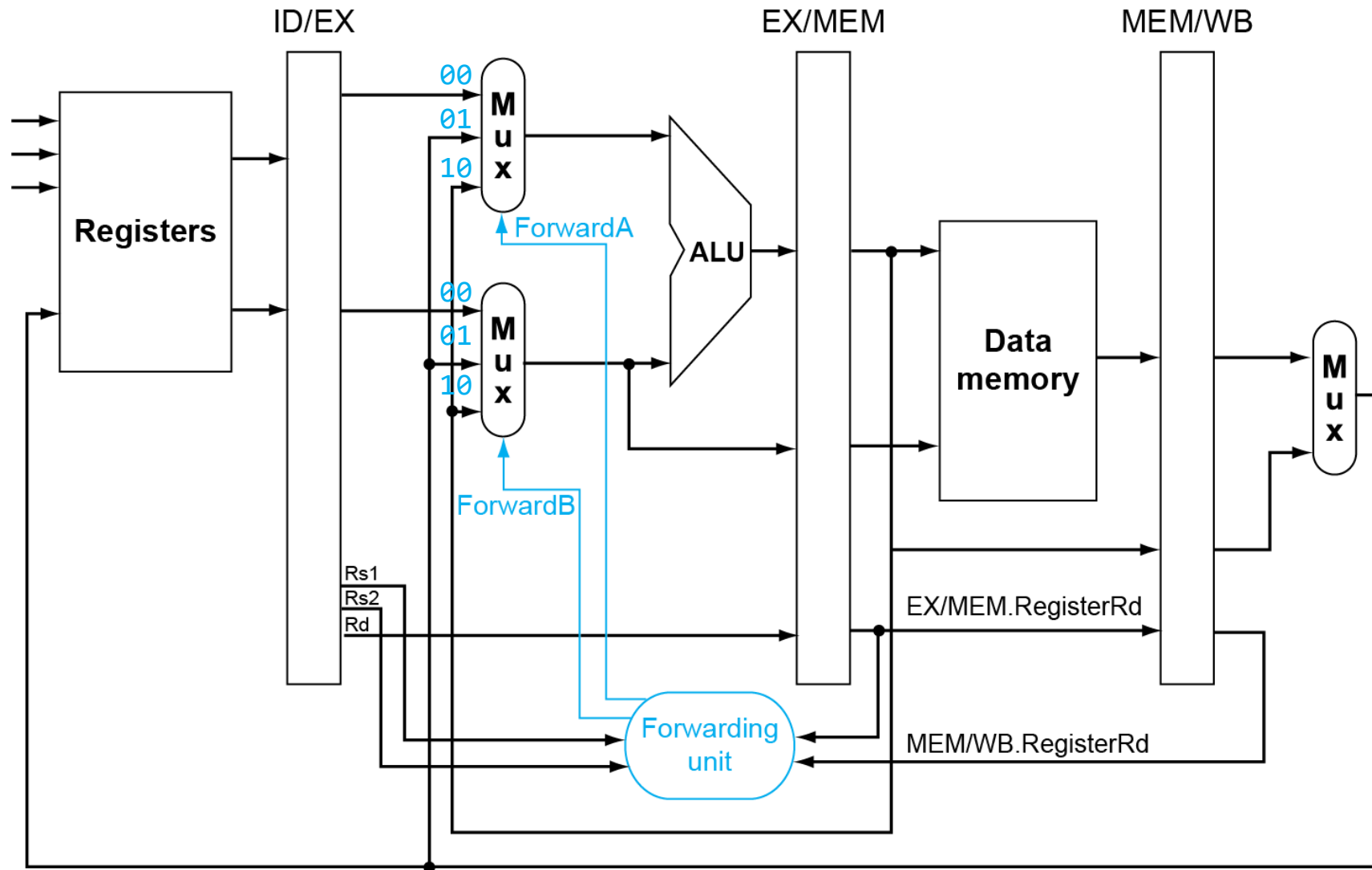
Detecting the Need to Forward (cont'd)

- 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
```
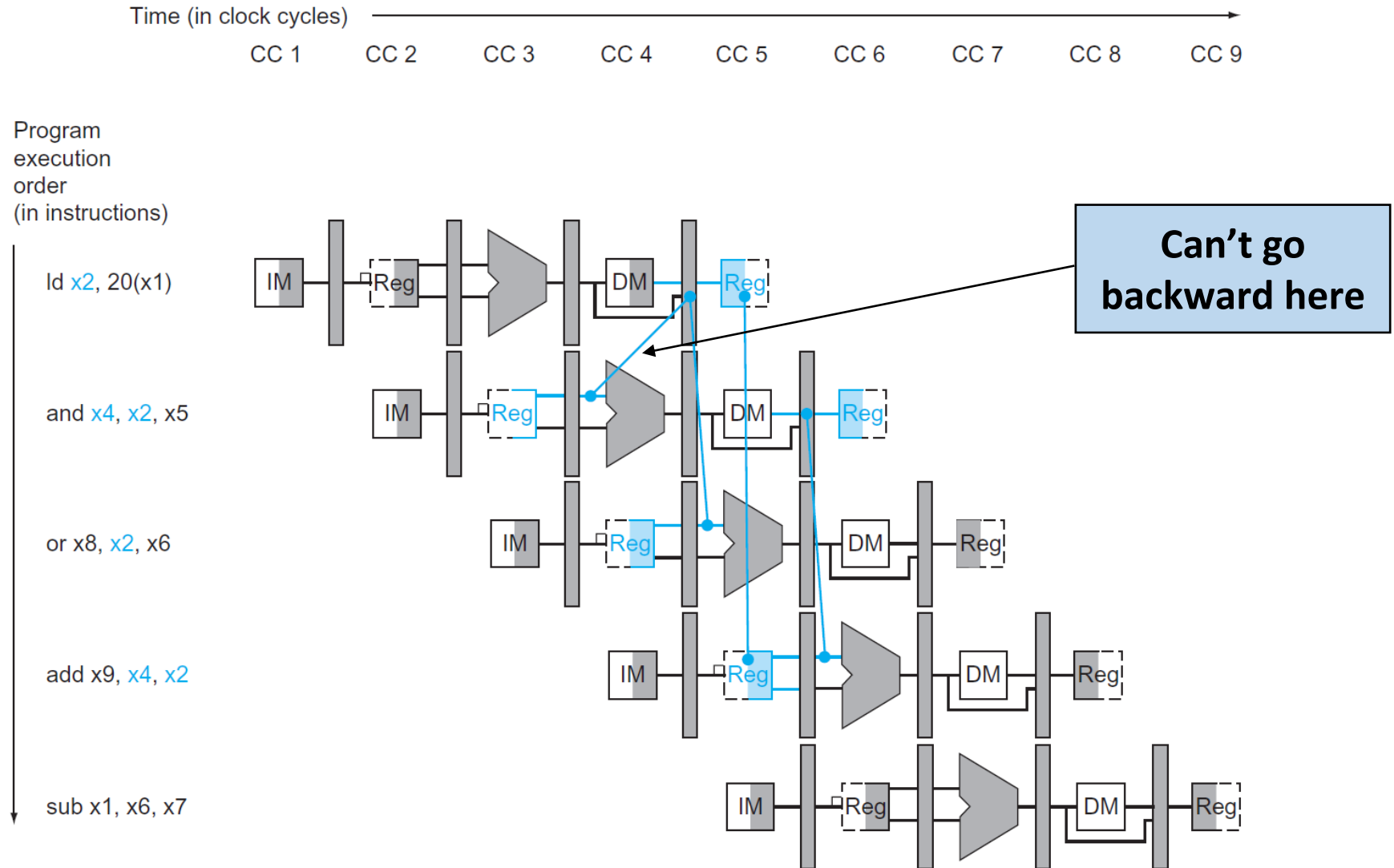
Datapath with Forwarding
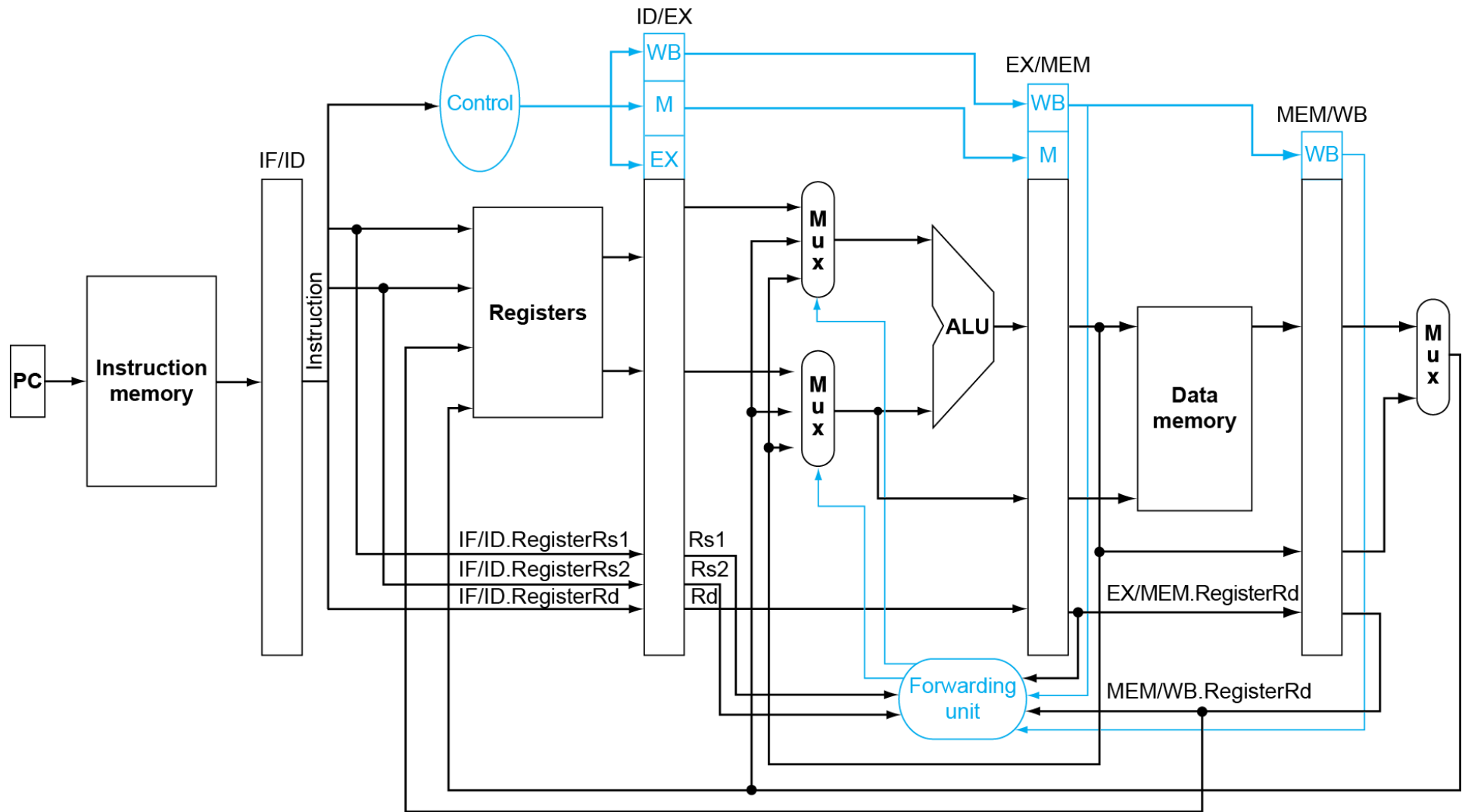
Complete Datapath with Forwarding
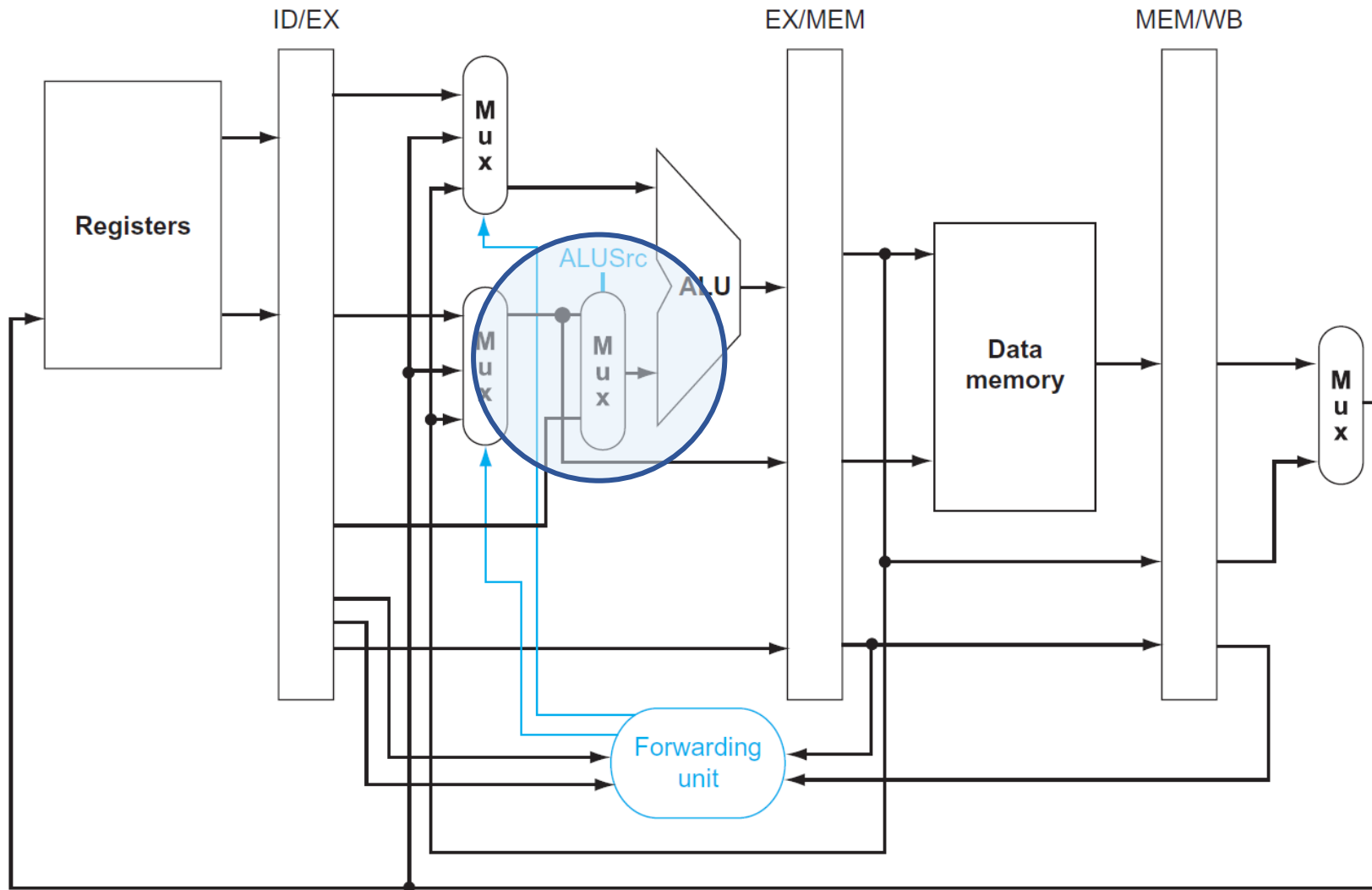
Load-Use Hazard

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`
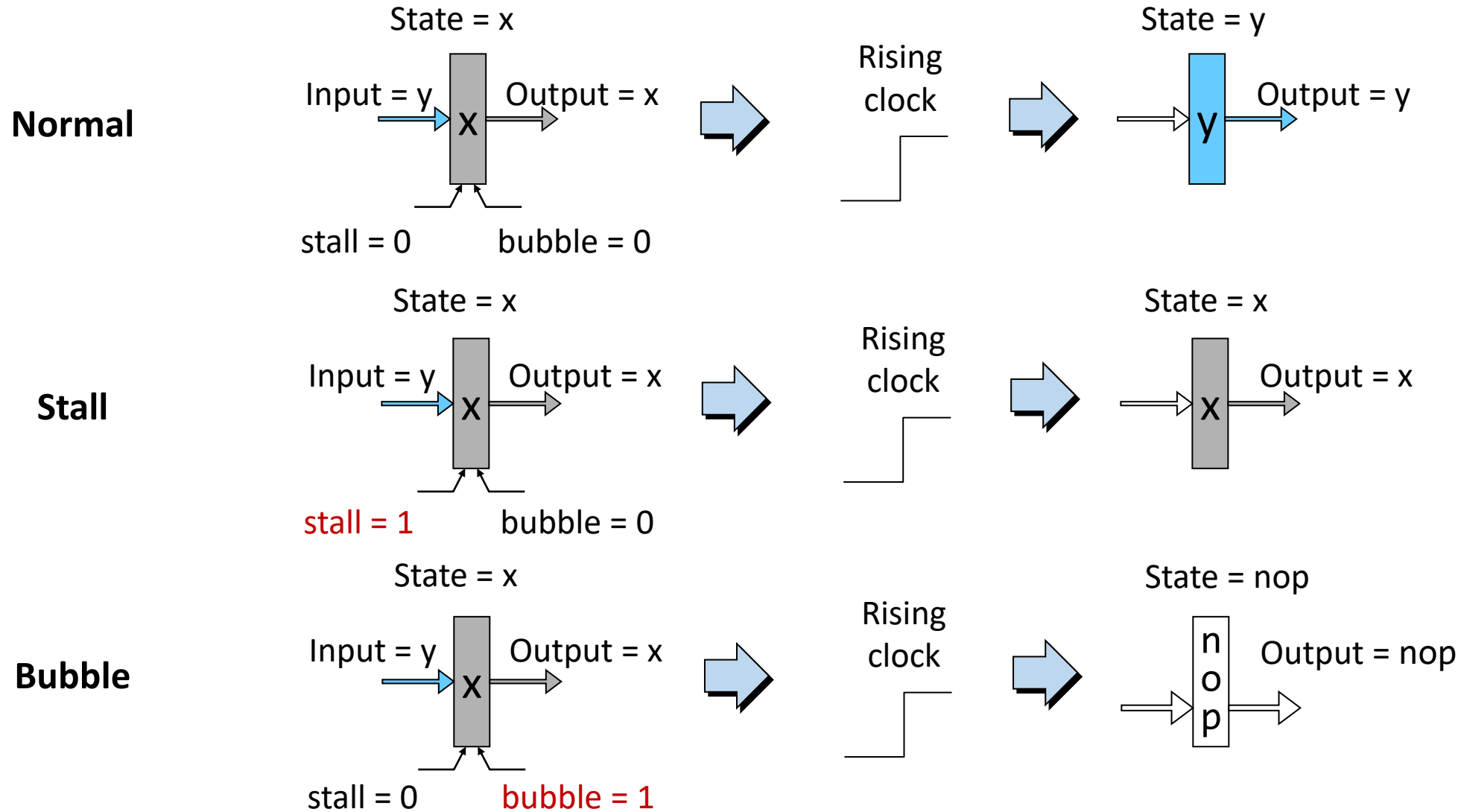
- Load-use hazard when

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

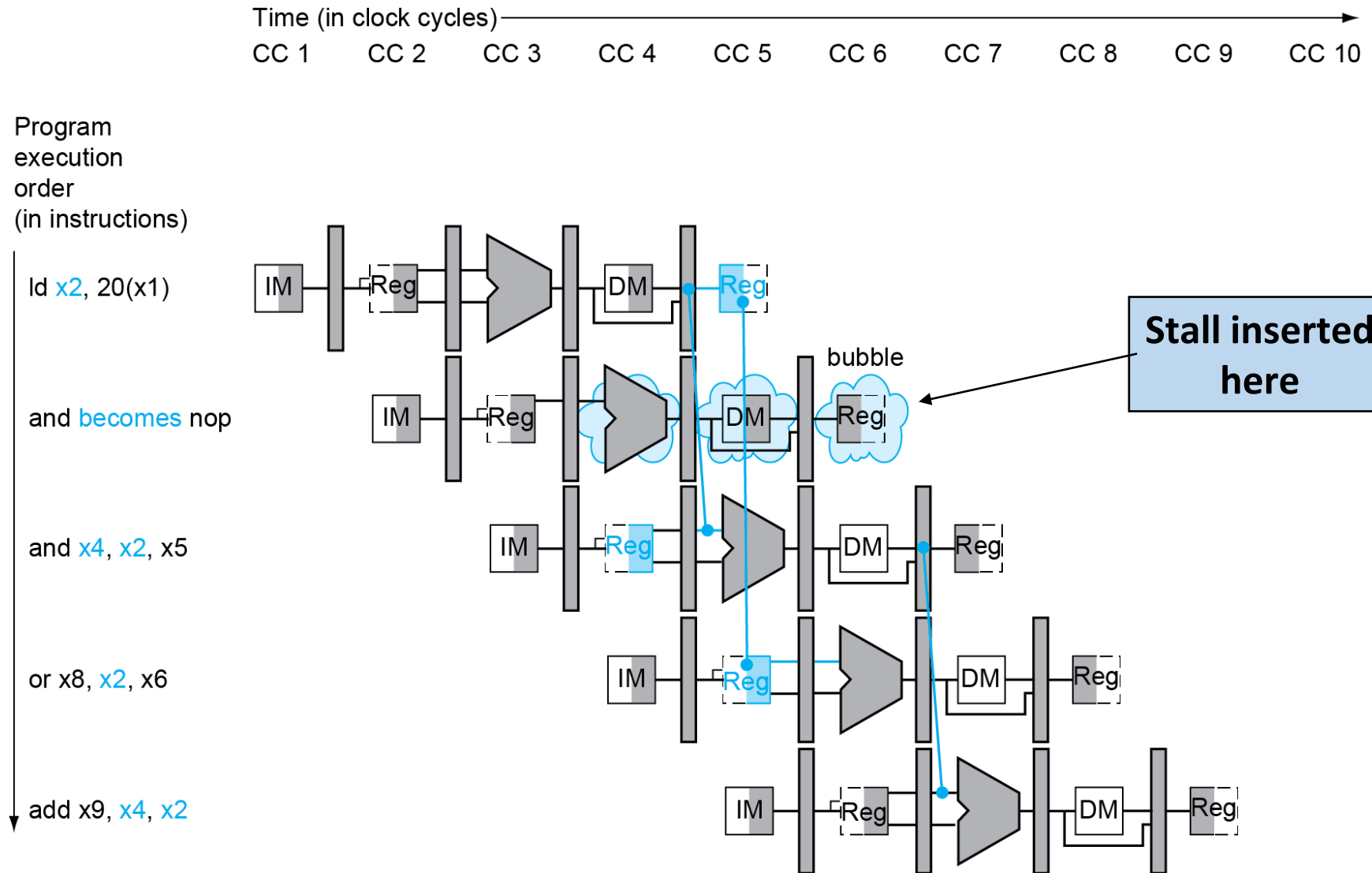- If detected, stall and insert bubble

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

Normal

State = x

Input = y   [x]   Output = x

stall = 0   bubble = 0

Rising clock

State = y

[y]   Output = y

Stall

State = x

Input = y   [x]   Output = x

stall = 1   bubble = 0

Rising clock

State = x

[x]   Output = x

Bubble

State = x

Input = y   [x]   Output = x

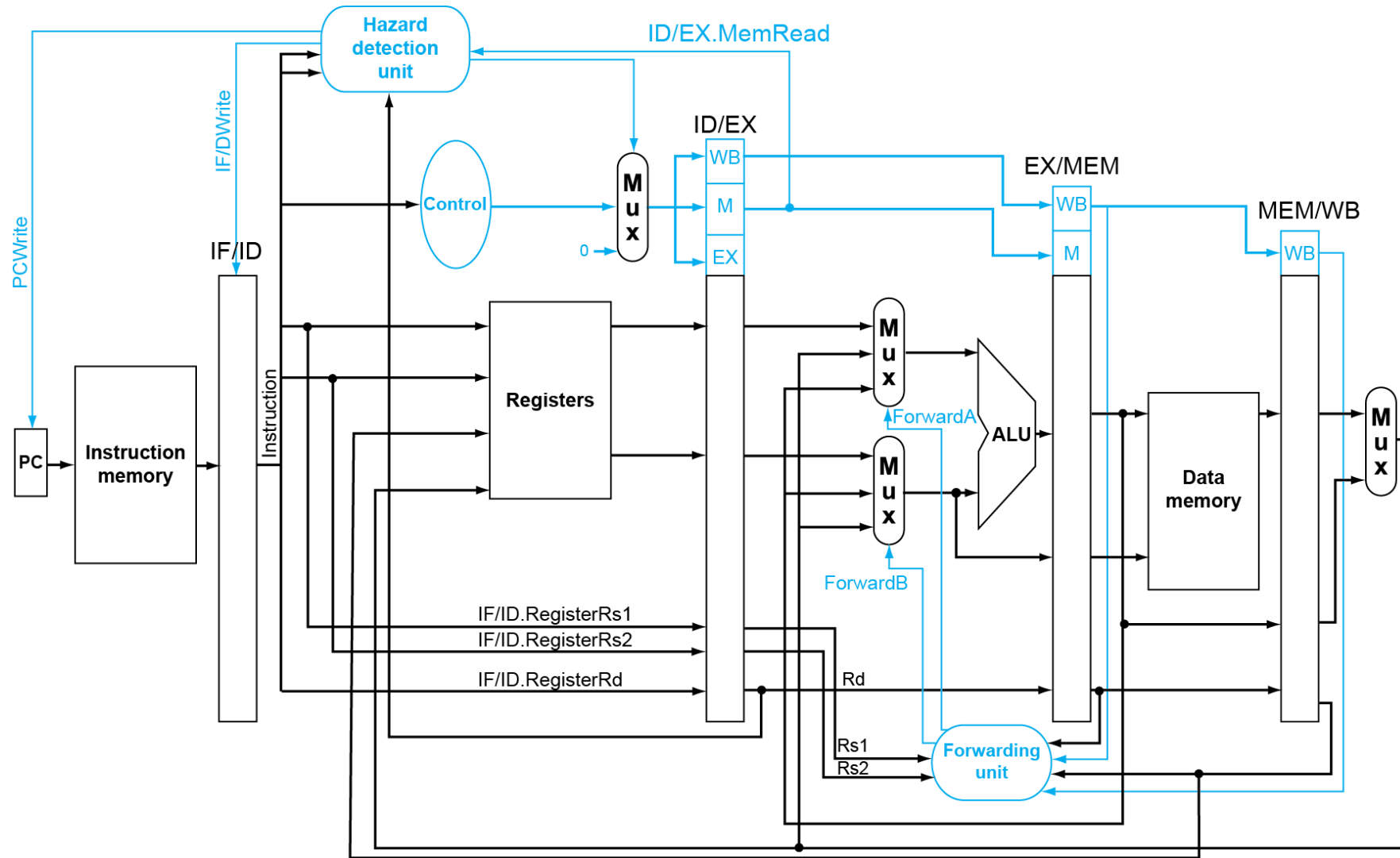stall = 0   bubble = 1

Rising clock

State = nop

[nop]   Output = nop

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 Hazard

- **Branch determines flow of control**
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction: still working on ID stage of branch

- **In RISC-V pipeline**
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Solutions to Control Hazard

- Stall on branch

- Branch prediction

- Delayed branch (compiler scheduling to avoid stalls)

Stall on Branch

- Wait until branch outcome determined before fetching next instruction
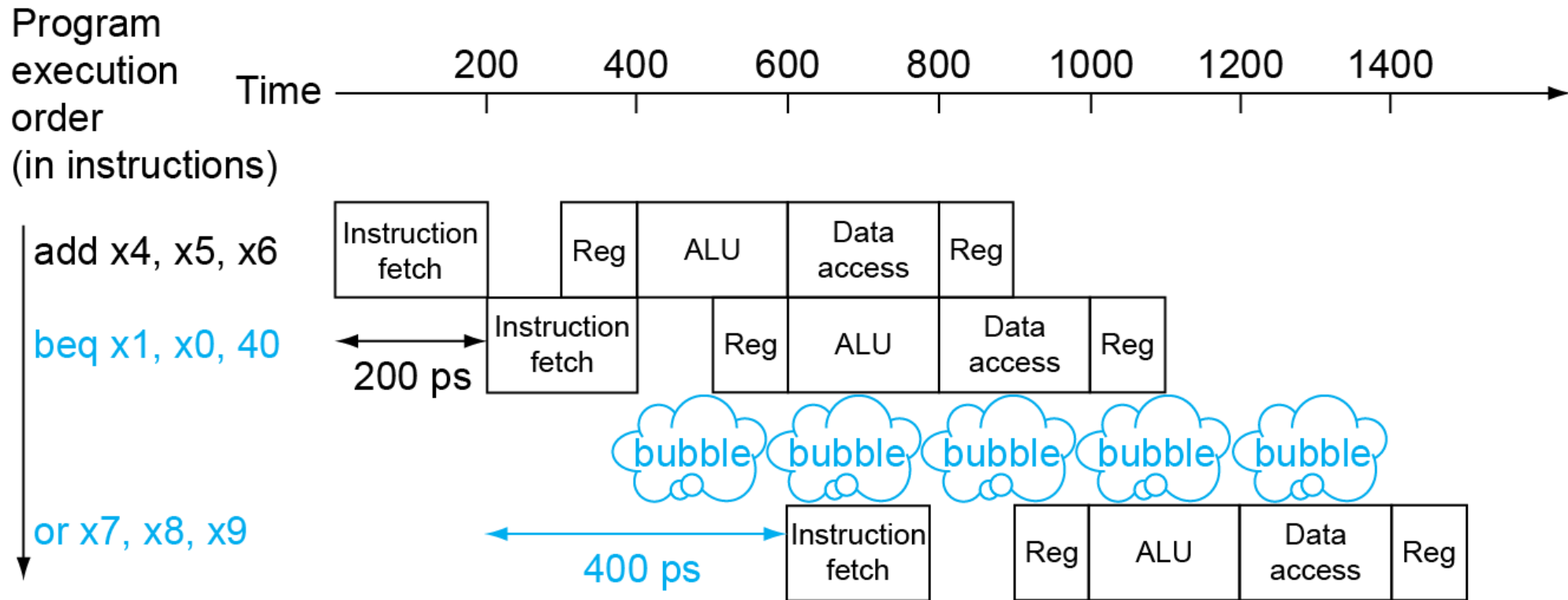
Branch Prediction

- Longer pipelines can't readily determine branch outcome early

  - Stall penalty becomes unacceptable

- Predict outcome of branch

  - Only stall if prediction is wrong

- In RISC-V pipeline

  - Can predict branches not taken

  - Fetch instruction after branch, with no delay

  - Cancel the fetched instruction if the prediction was wrong

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

Control Hazards in RISC-V

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

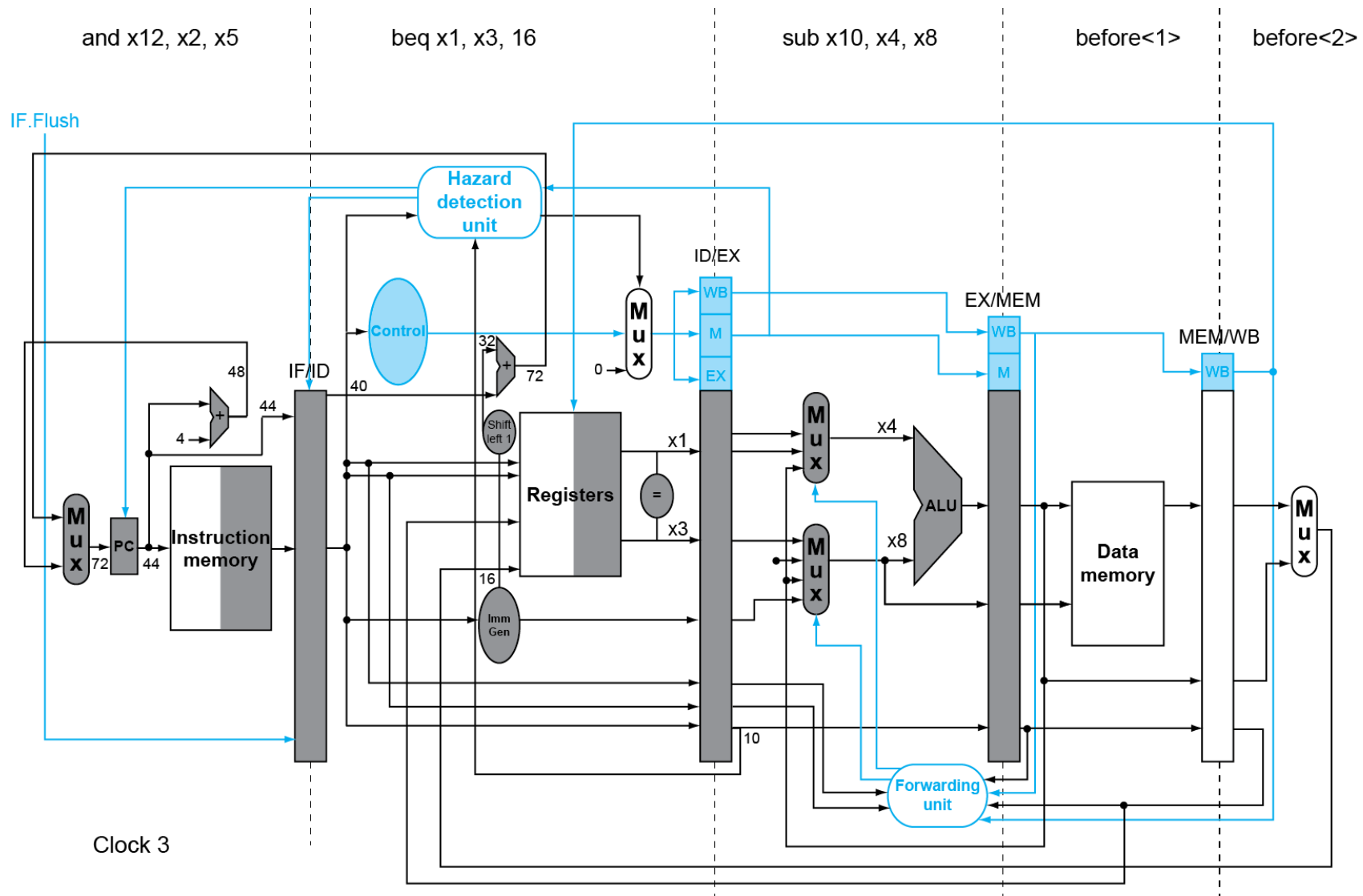Flush these instructions (Set control values to 0)
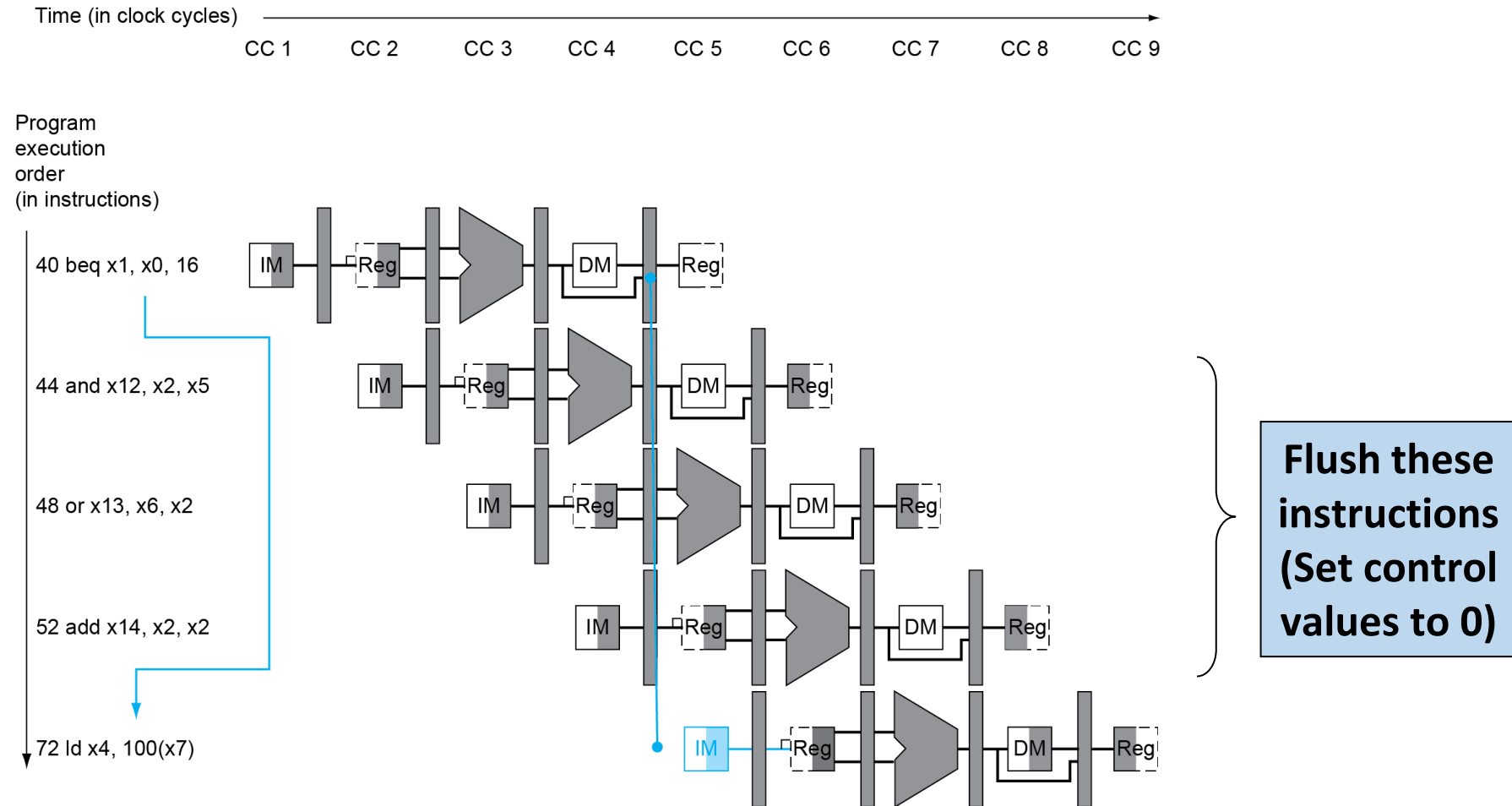
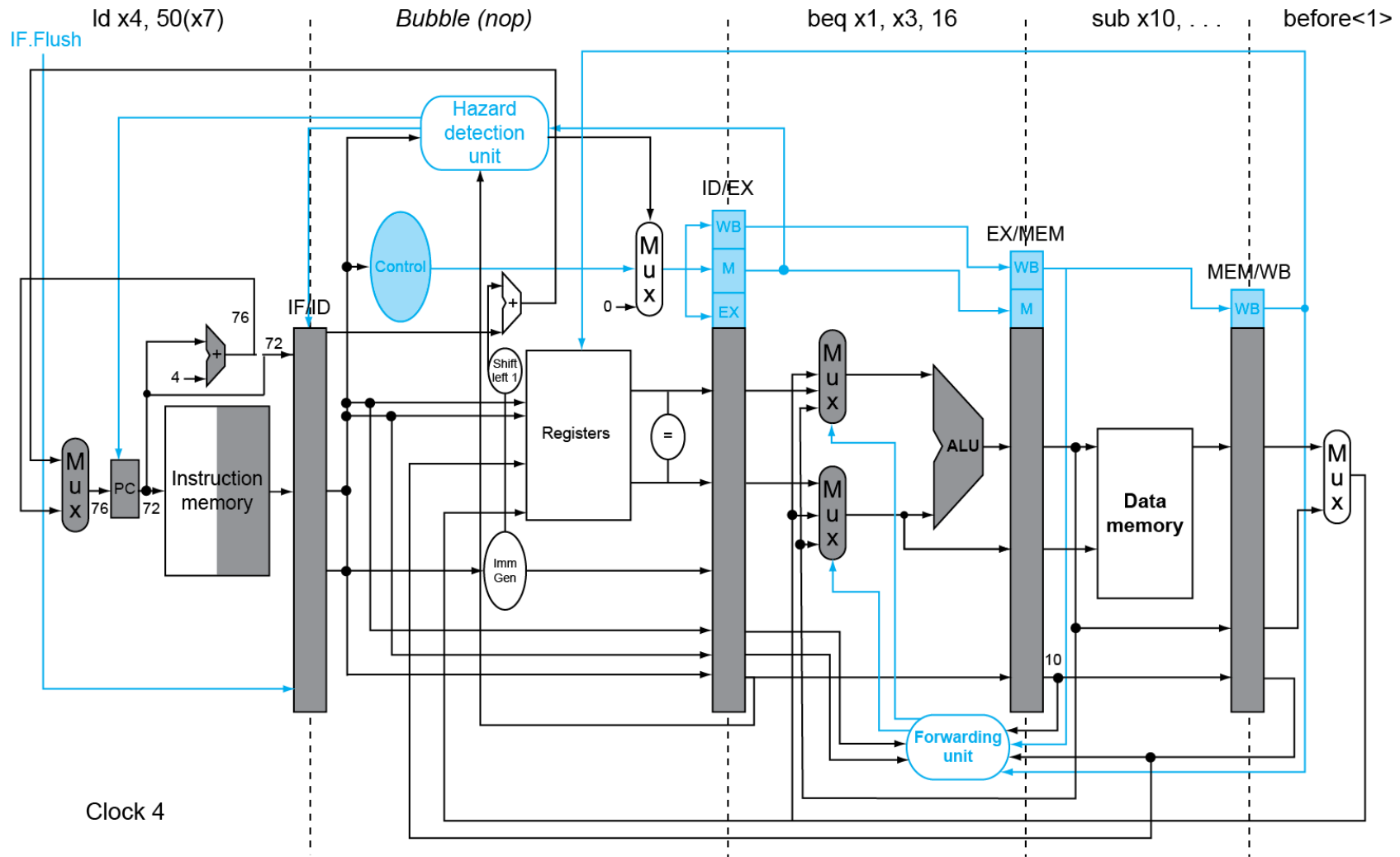# Reducing Branch Delay

- 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

and x12, x2, x5   beq x1, x3, 16   sub x10, x4, x8   before<1>   before<2>

IF.Flush

Hazard detection unit

ID/EX

WB

Control   EX/MEM

M   WB

IF/ID   EX   M   MEM/WB

48   WB

44

4   Shift left 1   x1   Mux   x4

Registers   =   ALU

Mux   x3   Mux   x8

PC   Instruction memory   Imm Gen   Data memory   Mux

72   44   16
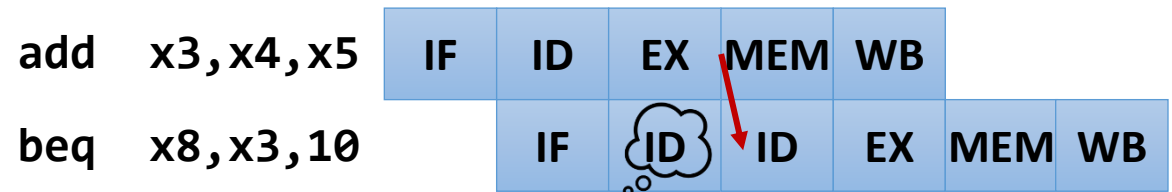
10

Forwarding unit

Clock 3
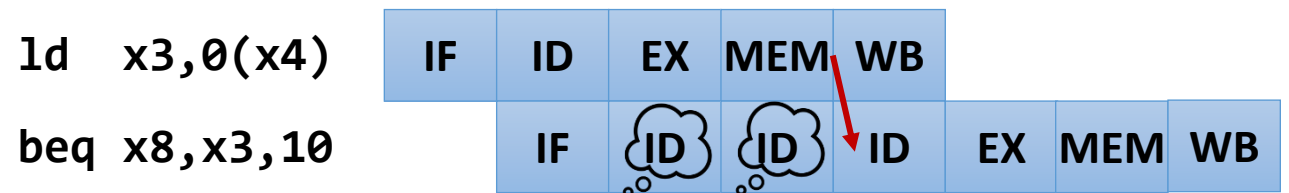
Example: Branch Taken (cont'd)

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
```

| add x3,x4,x5 | IF | ID | EX | MEM | WB | | | |
| beq x8,x3,10 | | IF | ID | ID | EX | MEM | WB |

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

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

| ld x3,0(x4) | IF | ID | EX | MEM | WB | | | |
| beq x8,x3,10 | | IF | ID | ID | ID | EX | MEM | WB |

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
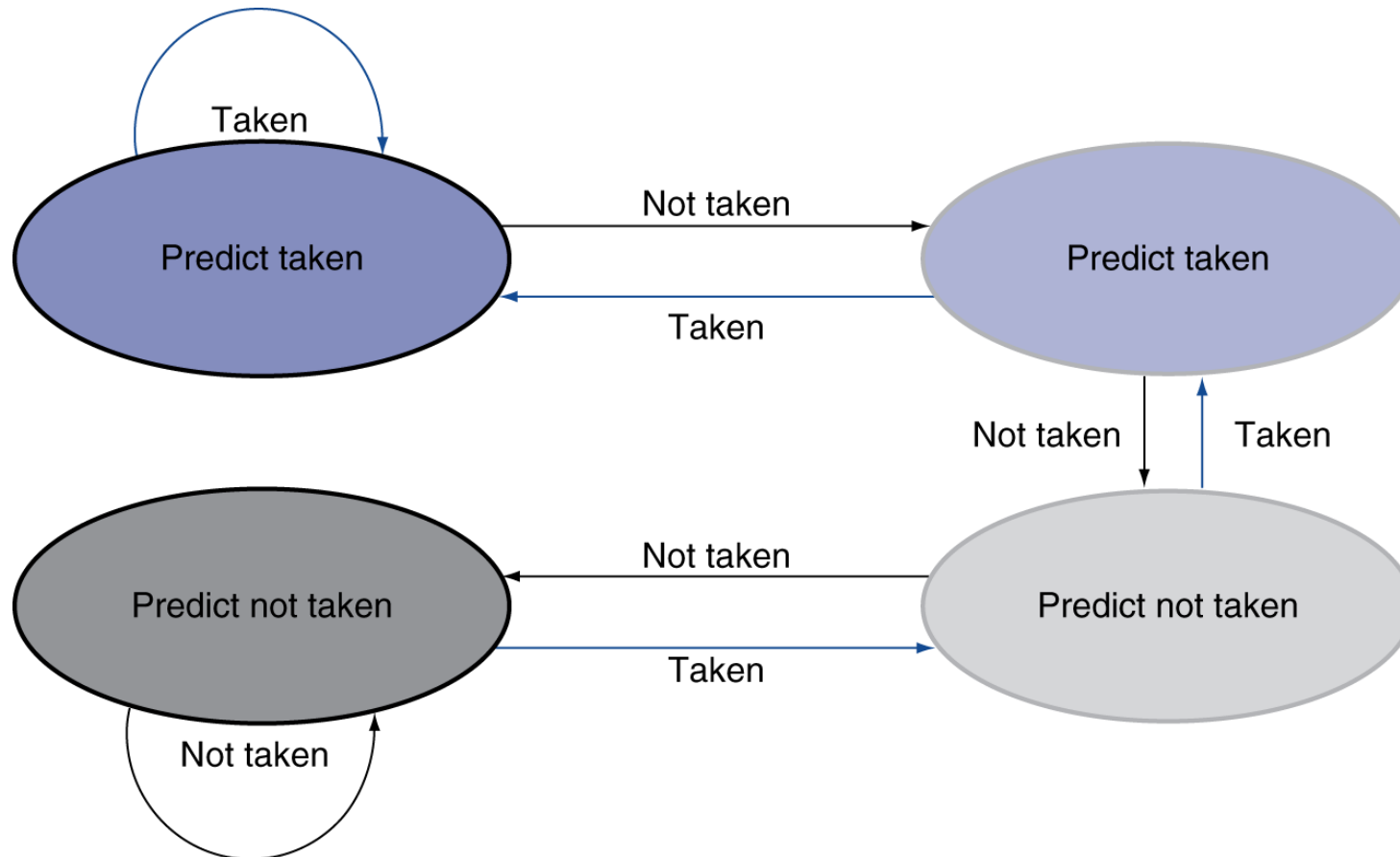
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

```
outer: ...
       ...
inner: ...
       ...
       beq .., .., inner
       ...
       beq .., .., outer
```
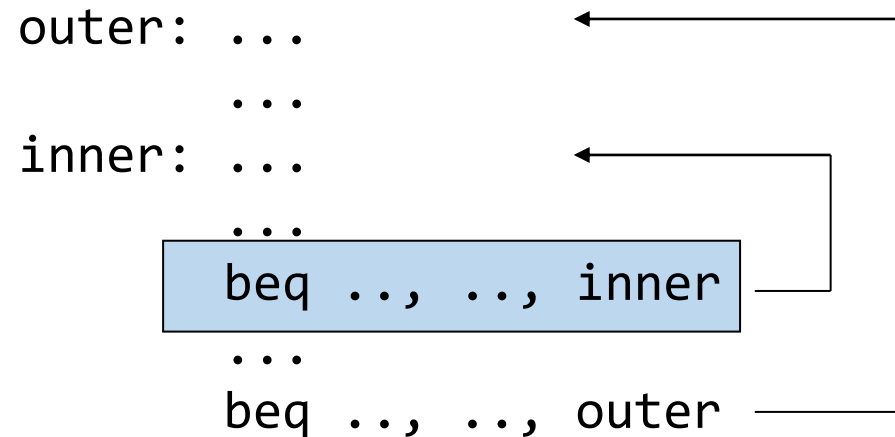
- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around

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

Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency

- Subject to hazards
  - Structural, data, control

- Instruction set design affects complexity of pipeline implementation