

Pipelining and Hazards

Prof. Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

See P&H Chapter: 4.6-4.8

Announcements

Prelim next week

Tuesday at 7:30.

Go to location based on netid

[a-g]* → MRS146: Morrison Hall 146

[h-l]* → RRB125: Riley-Robb Hall 125

[m-n]* → RRB105: Riley-Robb Hall 105

[o-s]* → MVRG71: M Van Rensselaer Hall G71

[t-z]* → MVRG73: M Van Rensselaer Hall G73

Prelim reviews

TODAY, Tue, Feb 24 @ 7:30pm in Olin 255

Sat, Feb 28 @ 7:30pm in Upson B17

Prelim conflicts

Contact Deniz Altinbuken <deniz@cs.cornell.edu>

Announcements

Prelim1:

- Time: We will start at 7:30pm sharp, so come early
- Location: on previous slide
- Closed Book
 - Cannot use electronic device or outside material
- Practice prelims are online in CMS

- Material covered everything up to end of this week
 - Everything up to and including data hazards
 - Appendix B (logic, gates, FSMs, memory, ALUs)
 - Chapter 4 (pipelined [and non] MIPS processor with hazards)
 - Chapters 2 (Numbers / Arithmetic, simple MIPS instructions)
 - Chapter 1 (Performance)
 - HW1, Lab0, Lab1, Lab2, C-Lab0, C-Lab1

Goals for Today

RISC and Pipelined Processor: Putting it all together

Data Hazards

- Data dependencies
- Problem, detection, and solutions
 - (delaying, stalling, forwarding, bypass, etc)
- Hazard detection unit
- Forwarding unit

Next time

- Control Hazards
 - What is the next instruction to execute if a branch is taken? Not taken?

MIPS Design Principles

Simplicity favors regularity

- 32 bit instructions

Smaller is faster

- Small register file

Make the common case fast

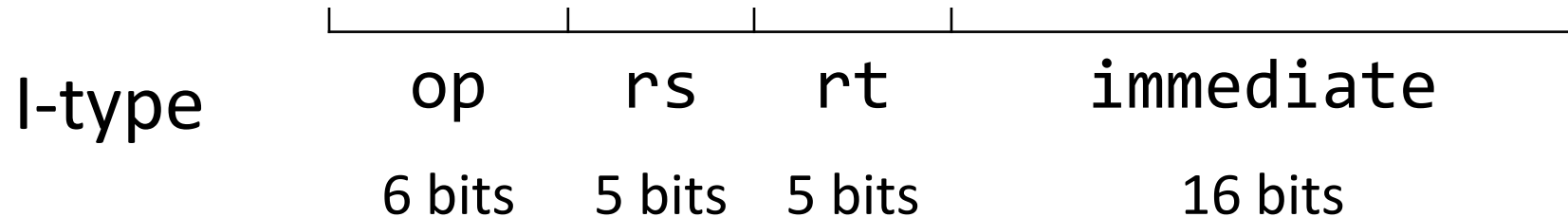
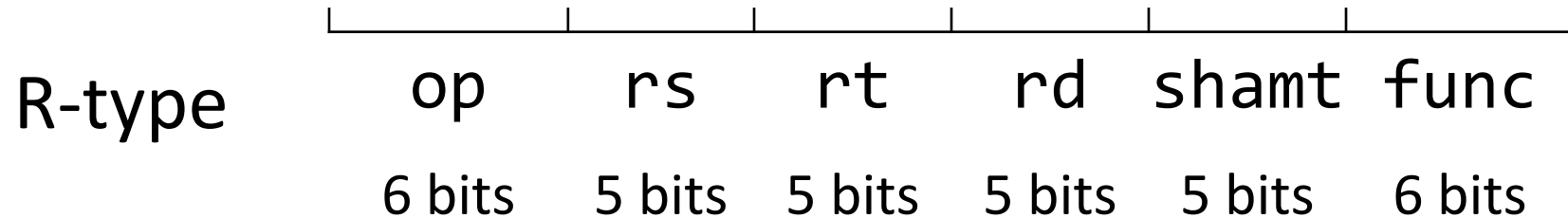
- Include support for constants

Good design demands good compromises

- Support for different type of interpretations/classes

Recall: MIPS instruction formats

All MIPS instructions are 32 bits long, has 3 formats



Recall: MIPS Instruction Types

Arithmetic/Logical

- R-type: result and two source registers, shift amount
- I-type: 16-bit immediate with sign/zero extension

Memory Access

- load/store between registers and memory
- word, half-word and byte operations

Control flow

- conditional branches: pc-relative addresses
- jumps: fixed offsets, register absolute

Recall: MIPS Instruction Types

Arithmetic/Logical

- ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
- ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
- MULT, DIV, MFLO, MTLO, MFHI, MTHI

Memory Access

- LW, LH, LB, LHU, LBU, LWL, LWR
- SW, SH, SB, SWL, SWR

Control flow

- BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
- J, JR, JAL, JALR, BEQL, BNEL, BLEZL, BGTZL

Special

- LL, SC, SYSCALL, BREAK, SYNC, COPROC

Pipelining

Principle:

Throughput increased by parallel execution

Balanced pipeline very important

Else slowest stage dominates performance

Pipelining:

- Identify *pipeline stages*
- Isolate stages from each other
- Resolve pipeline *hazards* (this and next lecture)

Basic Pipeline

Five stage “RISC” load-store architecture

1. Instruction fetch (IF)

- get instruction from memory, increment PC

2. Instruction Decode (ID)

- translate opcode into control signals and read registers

3. Execute (EX)

- perform ALU operation, compute jump/branch targets

4. Memory (MEM)

- access memory if needed

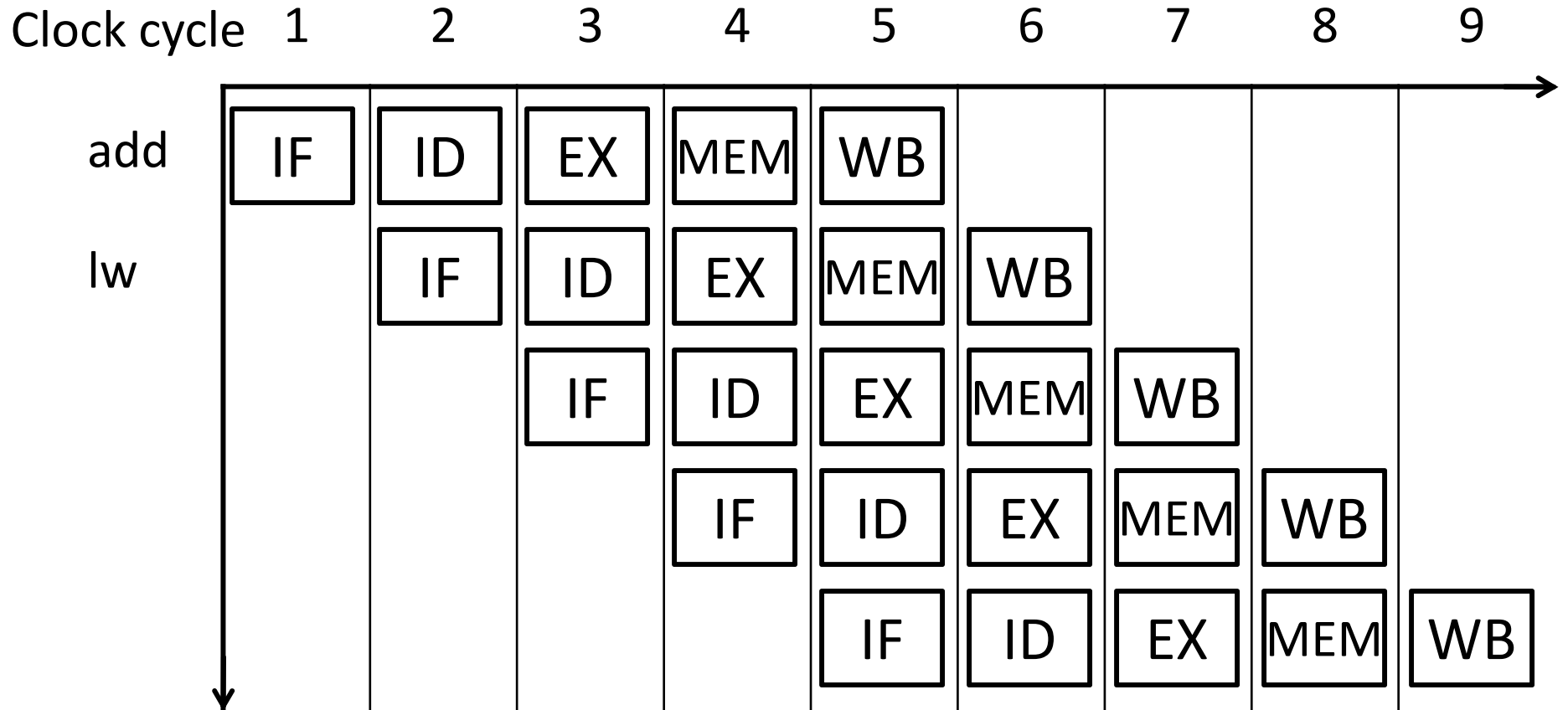
5. Writeback (WB)

- update register file

Pipelined Implementation

- Each instruction goes through the 5 stages
 - Each stage takes one clock cycle
 - So slowest stage determines clock cycle time

Time Graphs



iClicker

The pipeline achieves

A) Latency: 1, throughput: 1 instr/cycle

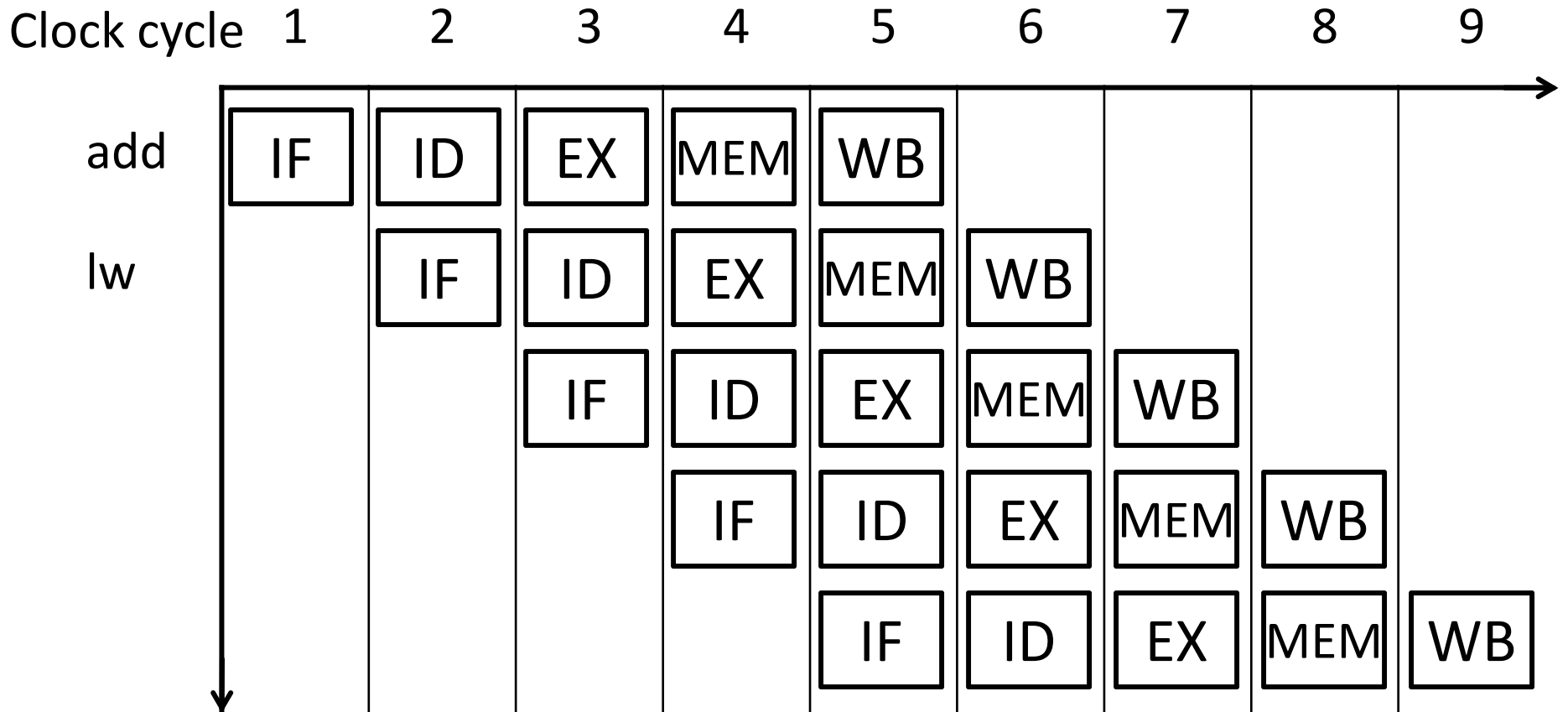
B) Latency: 5, throughput: 1 instr/cycle

C) Latency: 1, throughput: 1/5 instr/cycle

D) Latency: 5, throughput: 5 instr/cycle

E) None of the above

Time Graphs



Latency: 5 cycles

Throughput: 1 instr/cycle

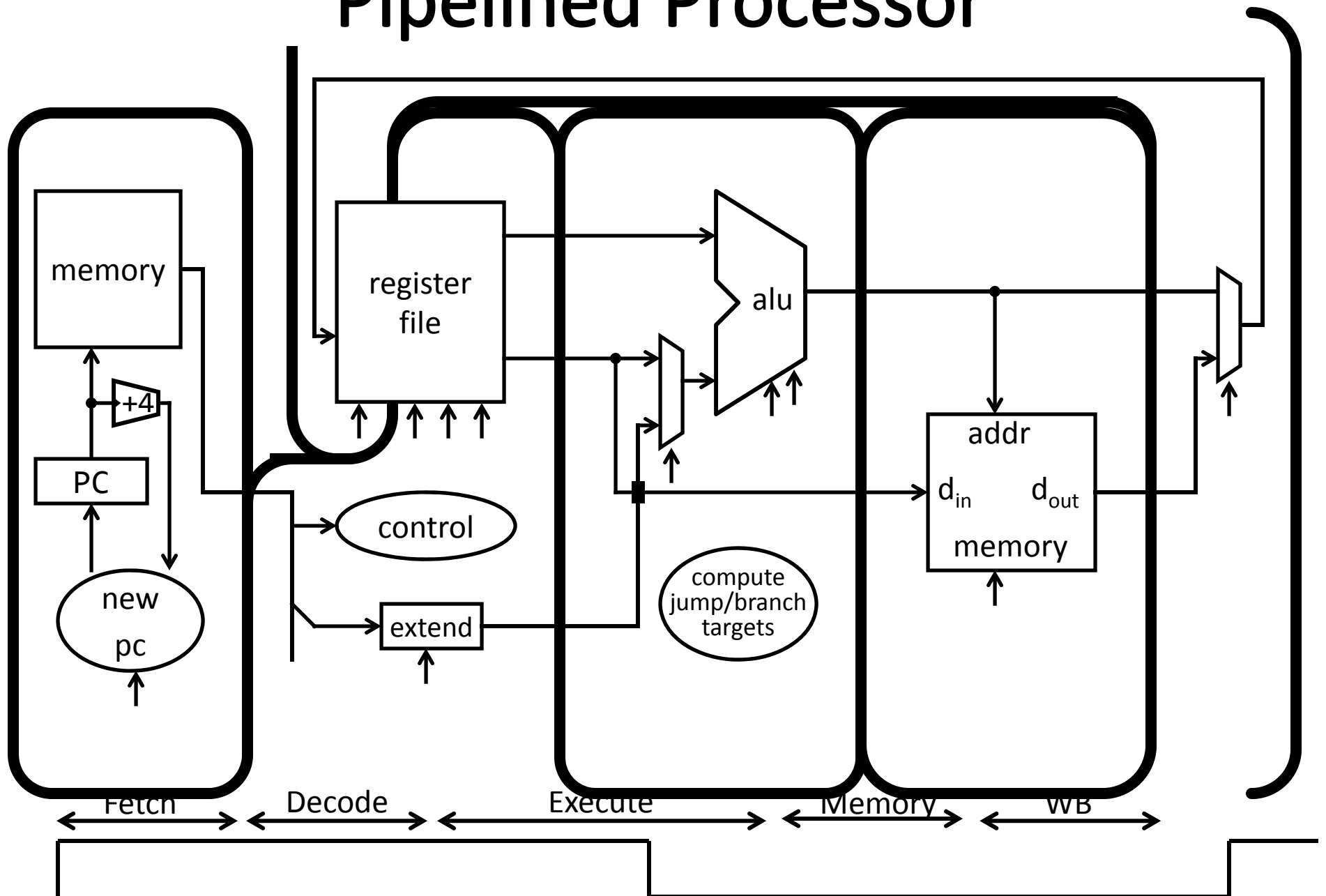
Concurrency: 5

CPI = 1

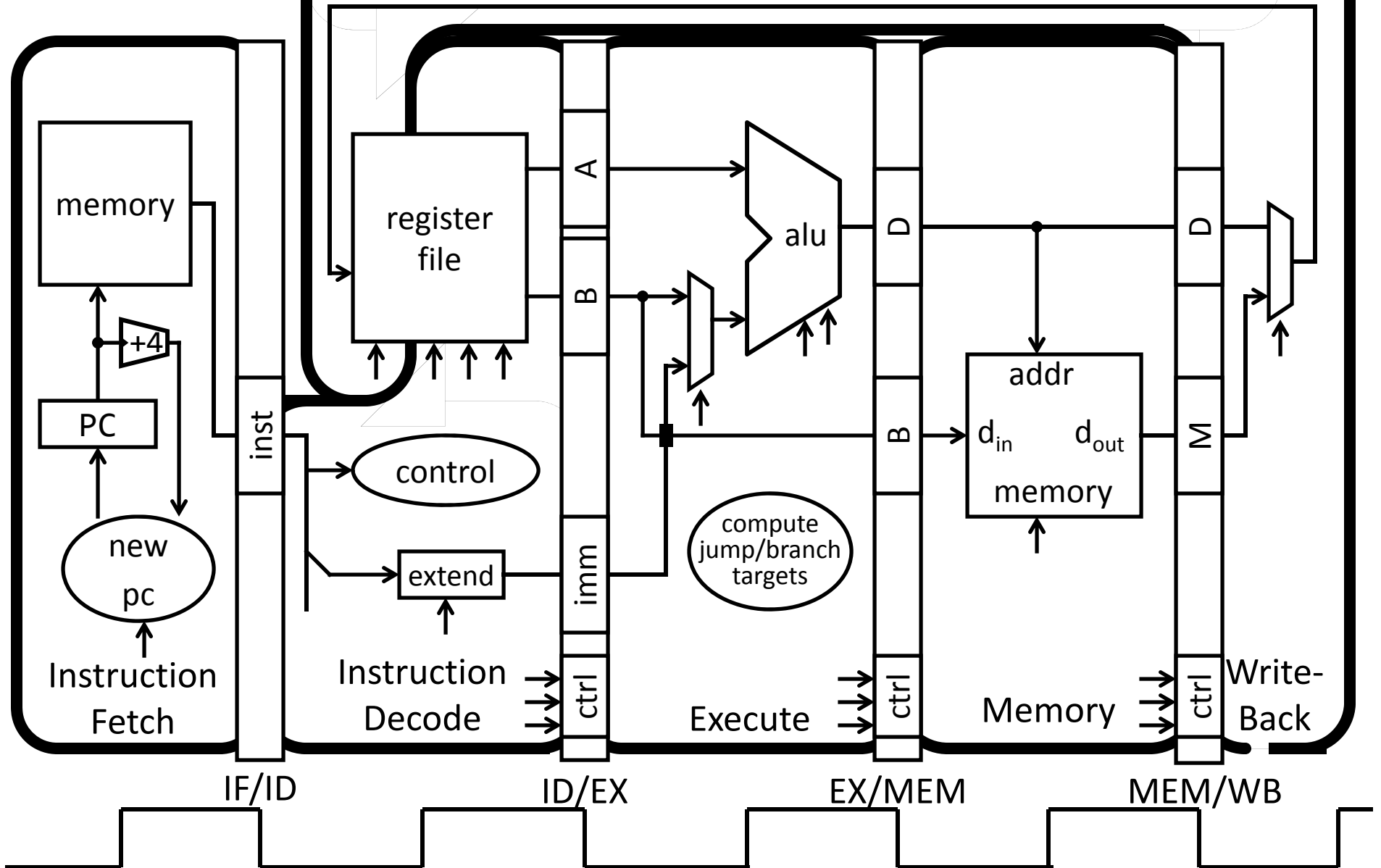
Pipelined Implementation

- Each instruction goes through the 5 stages
 - Each stage takes one clock cycle
 - So slowest stage determines clock cycle time
- Stages must share information. How?
 - Add pipeline registers (flip-flops) to pass results between different stages

Pipelined Processor



Pipelined Processor



Pipelined Implementation

- Each instruction goes through the 5 stages
 - Each stage takes one clock cycle
 - So slowest stage determines clock cycle time
- Stages must share information. How?
 - Add pipeline registers (flip-flops) to pass results between different stages

And is this it?

Not quite....

Hazards

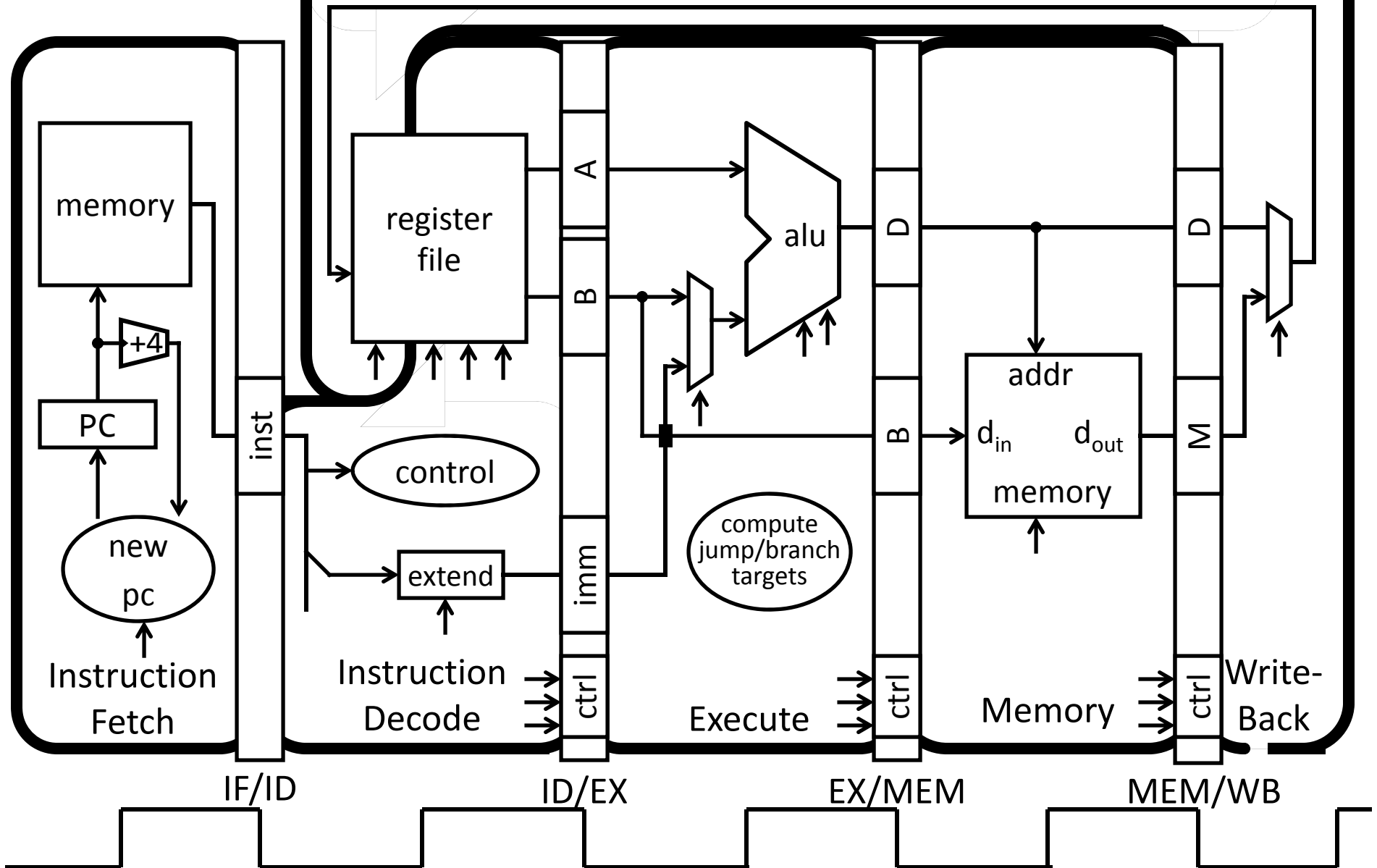
3 kinds

- Structural hazards
 - Multiple instructions want to use same unit
- Data hazards
 - Results of instruction needed before ready
- Control hazards
 - Don't know which side of branch to take

Will get back to this

First, how to pipeline when no hazards

Pipelined Processor



Example: : Sample Code (Simple)

```
add    r3, r1, r2;  
nand   r6, r4, r5;  
lw     r4, 20(r2);  
add    r5, r2, r5;  
sw     r7, 12(r3);
```

Example: Sample Code (Simple)

Assume eight-register machine

Run the following code on a pipelined datapath

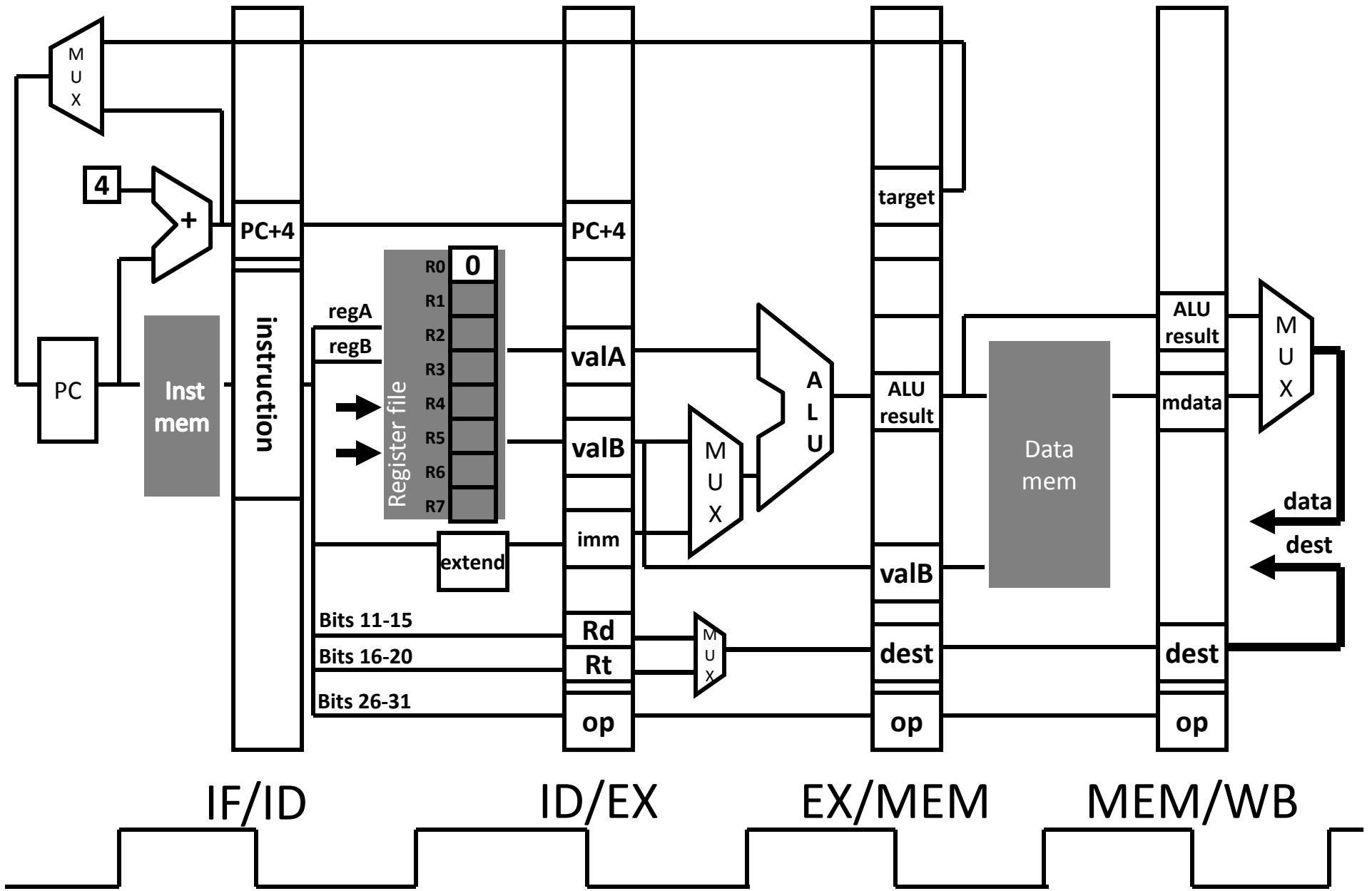
```
add    r3 r1  r2 ; reg 3 = reg 1 + reg 2
```

```
nand   r6 r4  r5 ; reg 6 = ~(reg 4 & reg 5)
```

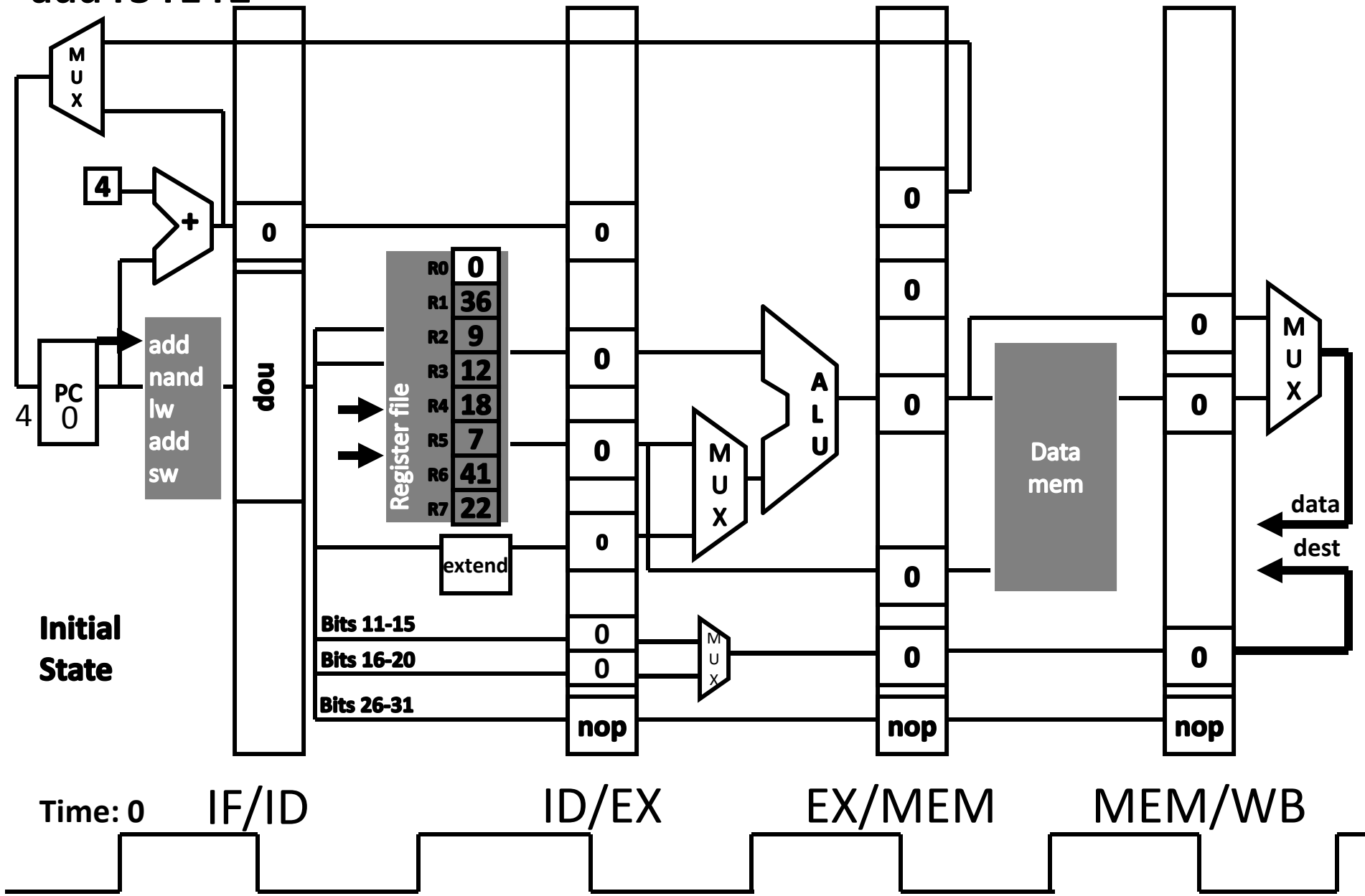
```
lw     r4 20 (r2) ; reg 4 = Mem[reg2+20]
```

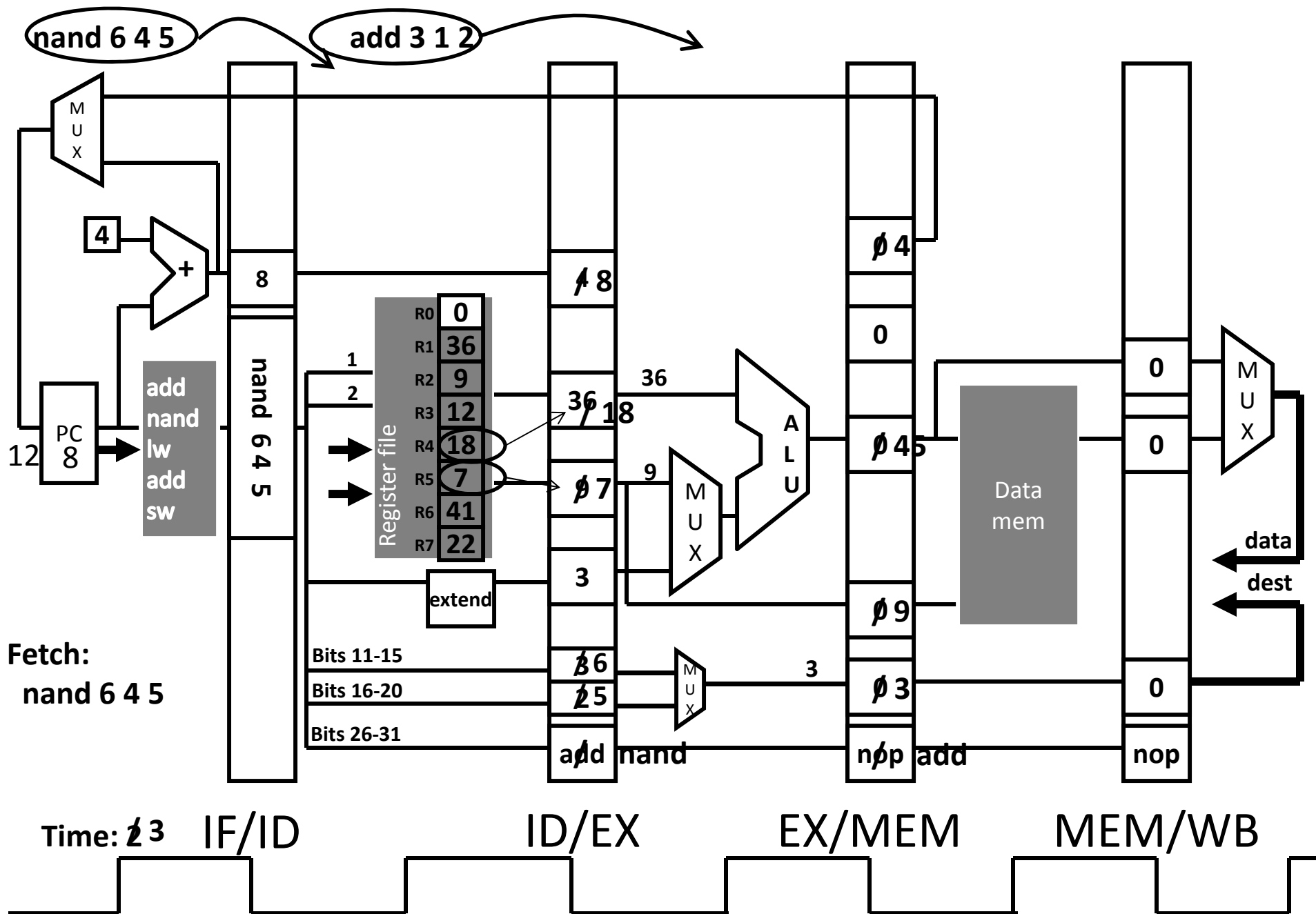
```
add    r5 r2  r5 ; reg 5 = reg 2 + reg 5
```

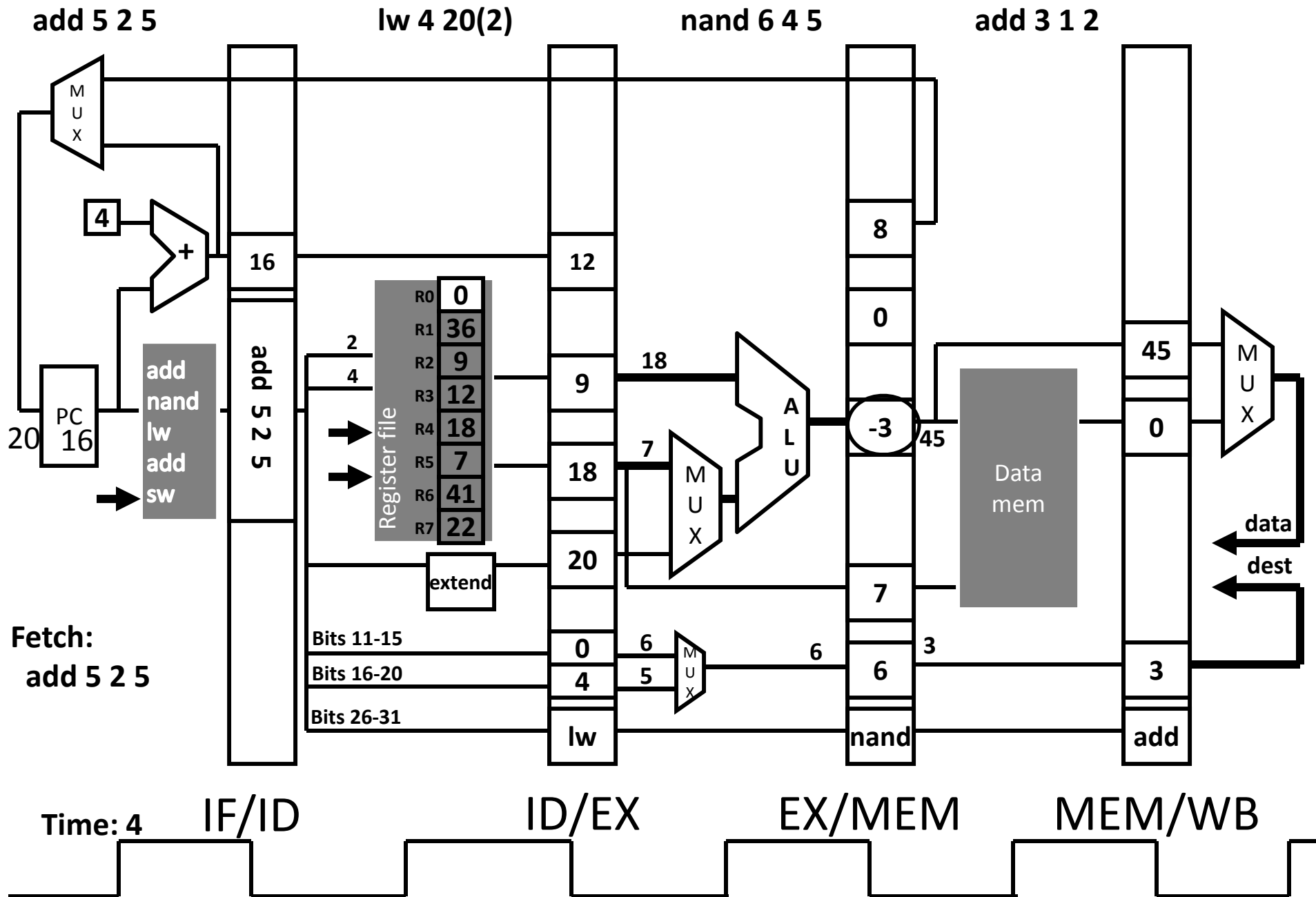
```
sw     r7 12(r3) ; Mem[reg3+12] = reg 7
```

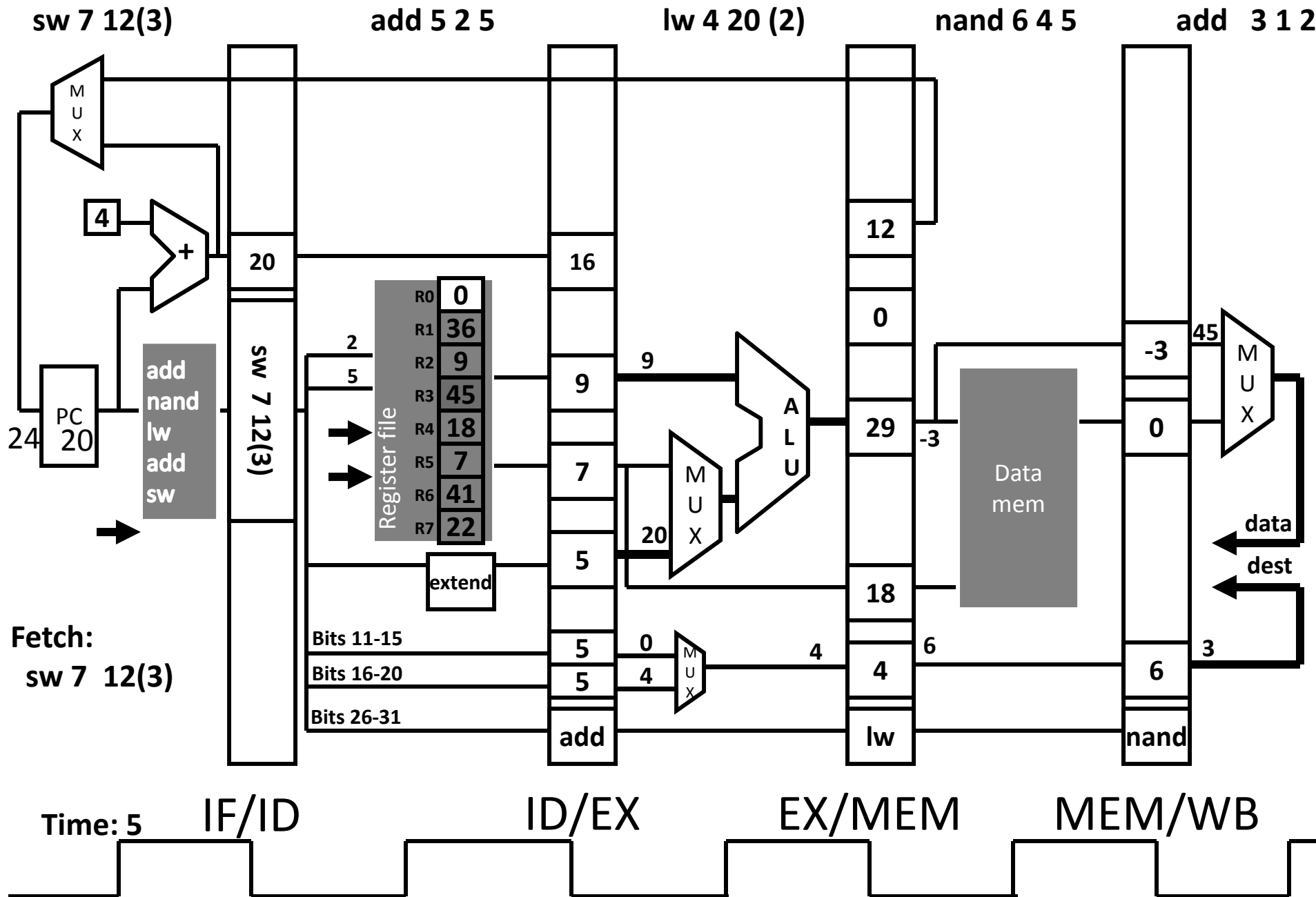


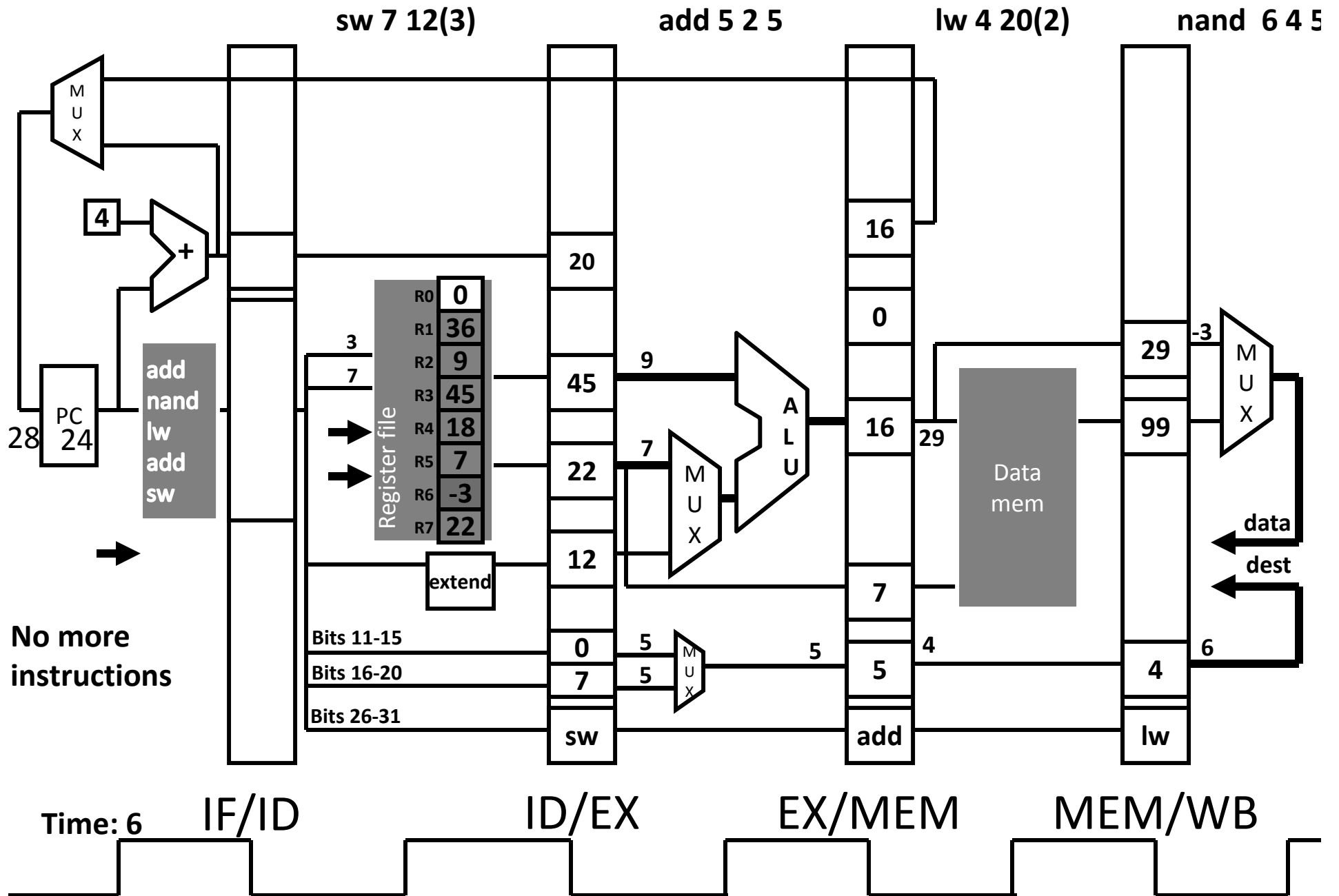
At time 1,
Fetch
add r3 r1 r2



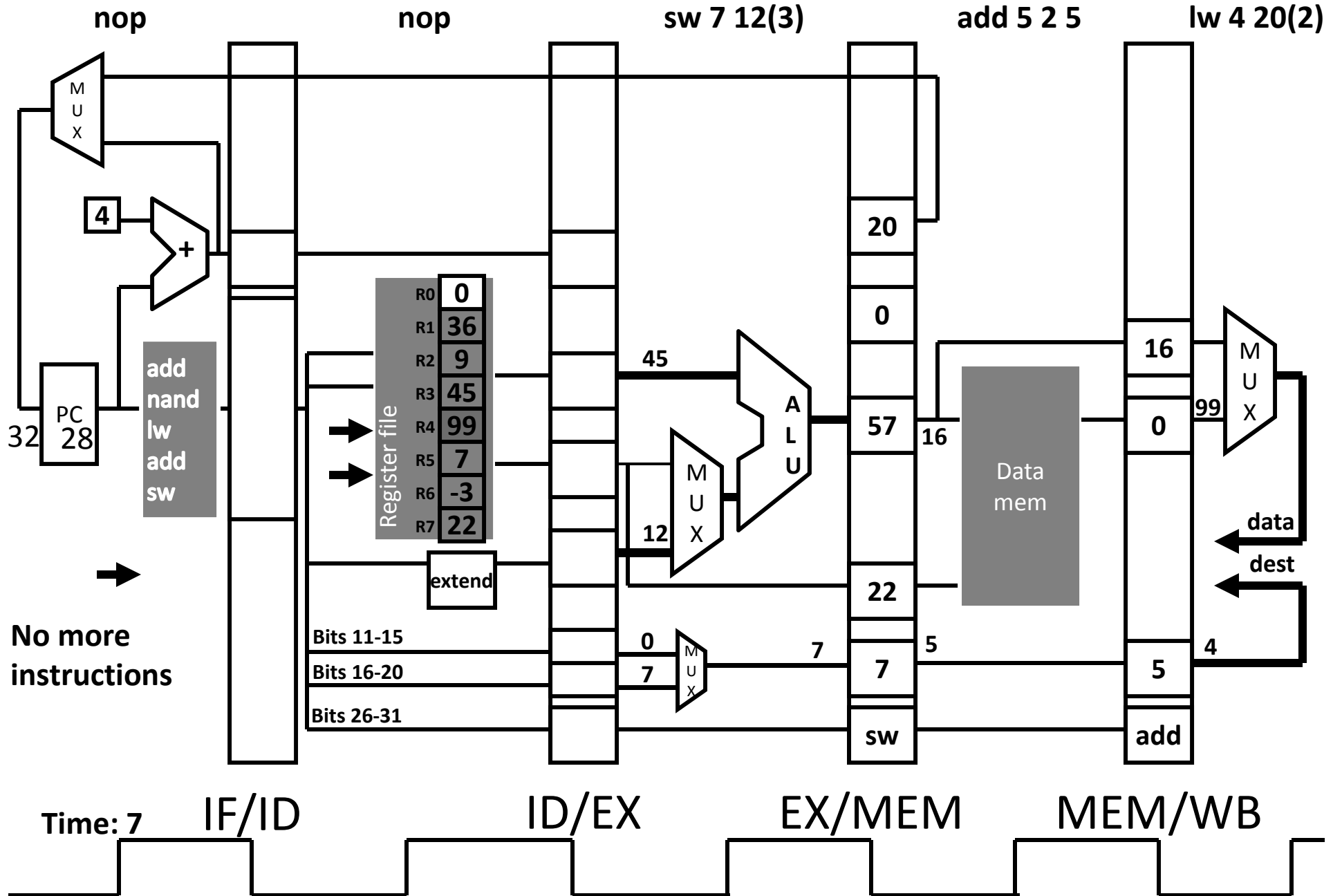


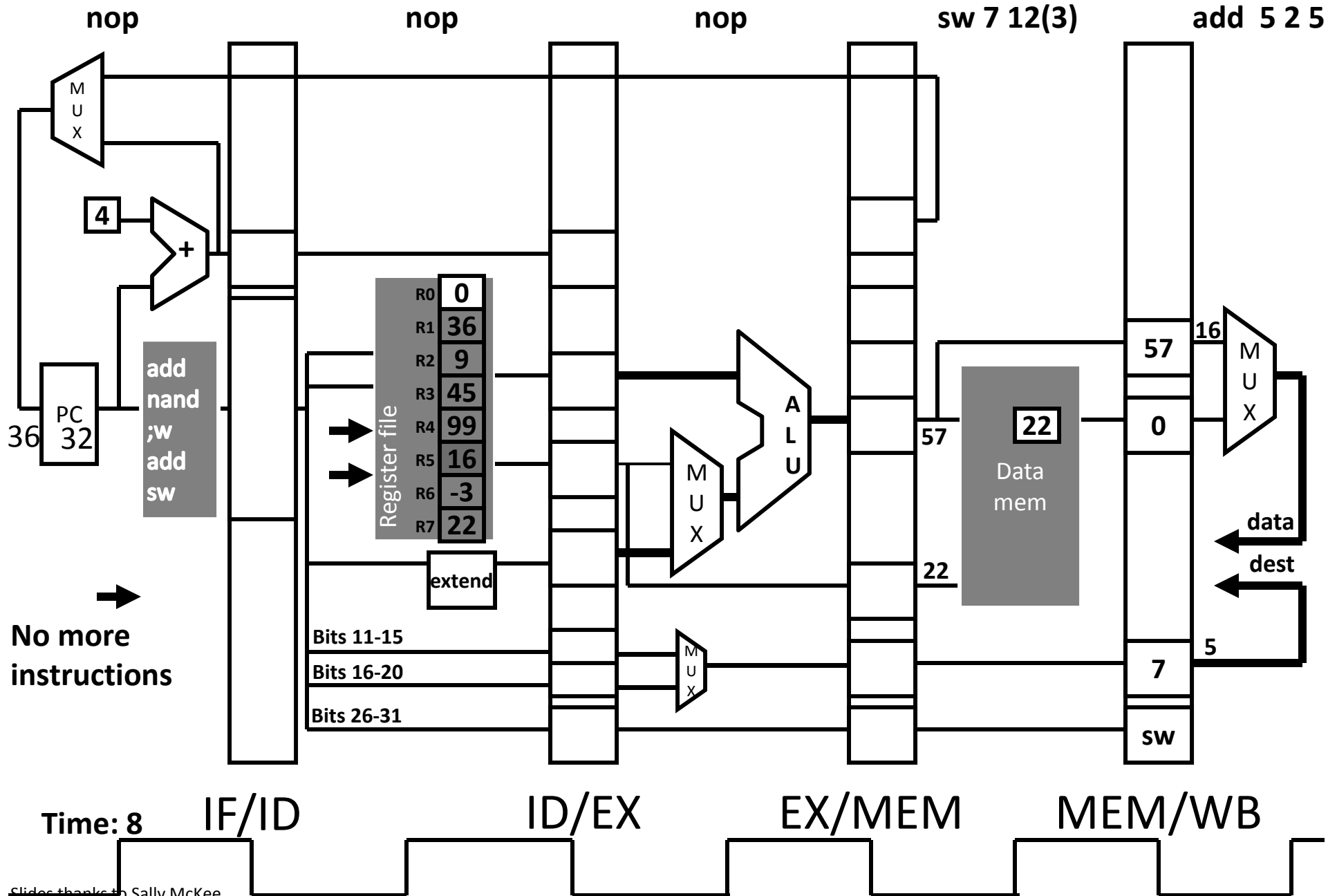






No more instructions





Takeaway

Pipelining is a powerful technique to mask latencies and increase throughput

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel
 - Instruction level parallelism

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)

Hazards

See P&H Chapter: 4.7-4.8

Hazards

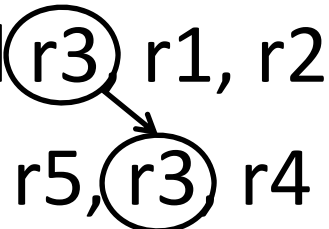
3 kinds

- Structural hazards
 - Multiple instructions want to use same unit
- Data hazards
 - Results of instruction needed before
- Control hazards
 - Don't know which side of branch to take

Next Goal

What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. add(r3) r1, r2
sub r5, (r3) r4



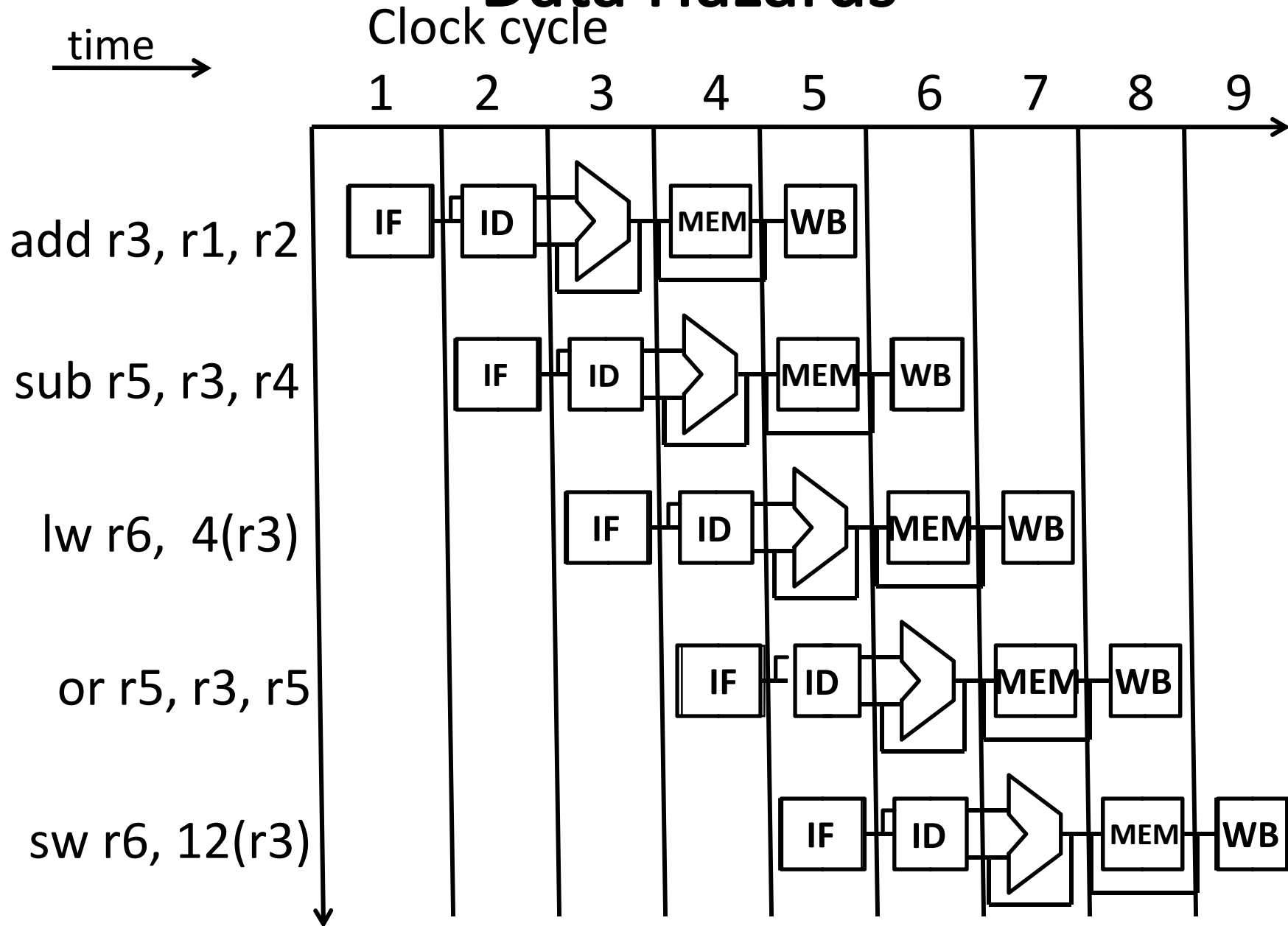
Need to detect and then fix such hazards

Data Hazards

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written
 - i.e instruction may need values that are being computed further down the pipeline
 - *in fact, this is quite common*

Data Hazards



iClicker

How many data hazards due to r3 only

add r3, r1, r2

sub r5, r3, r4

lw r6, 4(r3)

or r5, r3, r5

sw r6, 12(r3)

A) 1

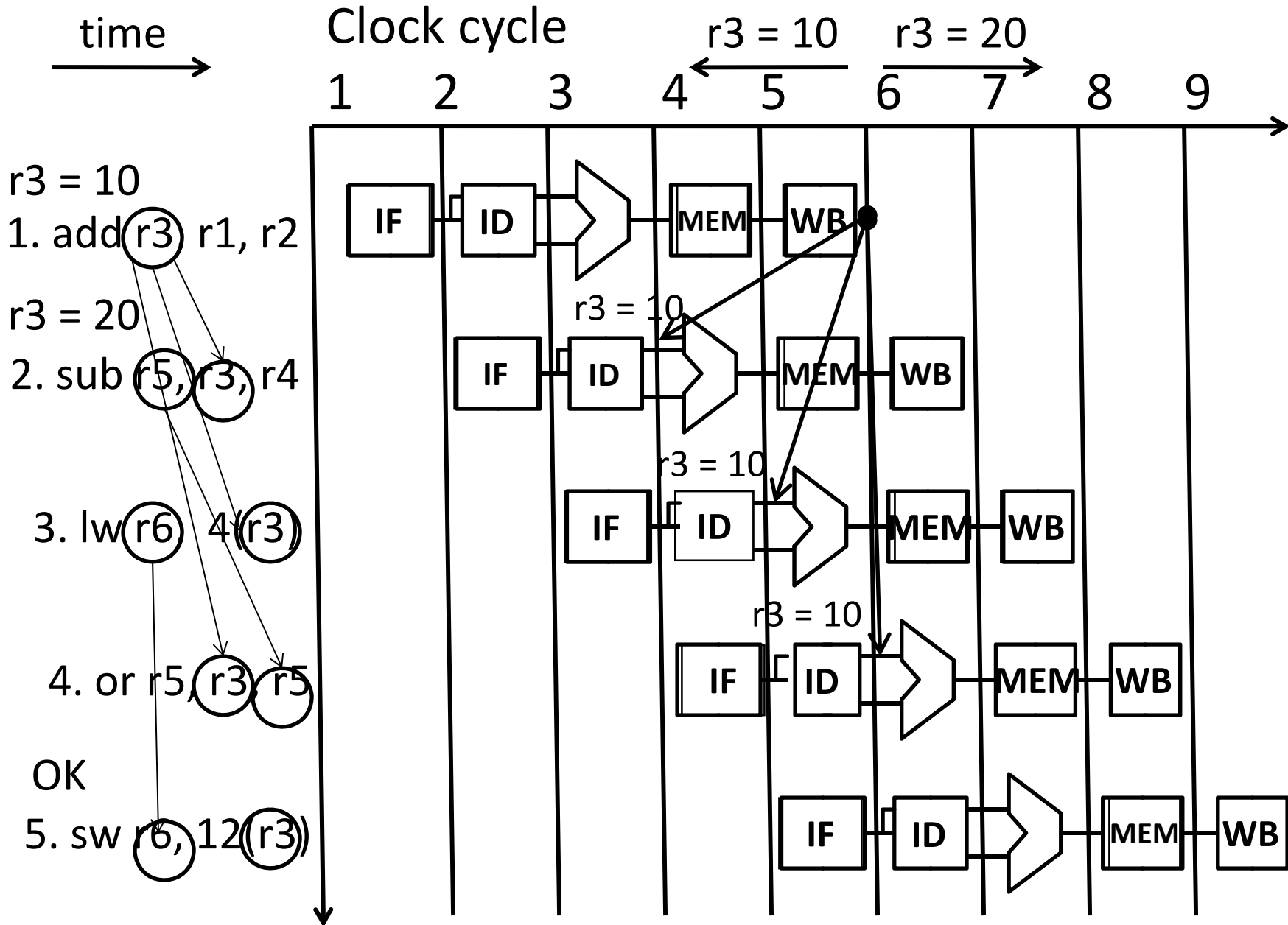
B) 2

C) 3

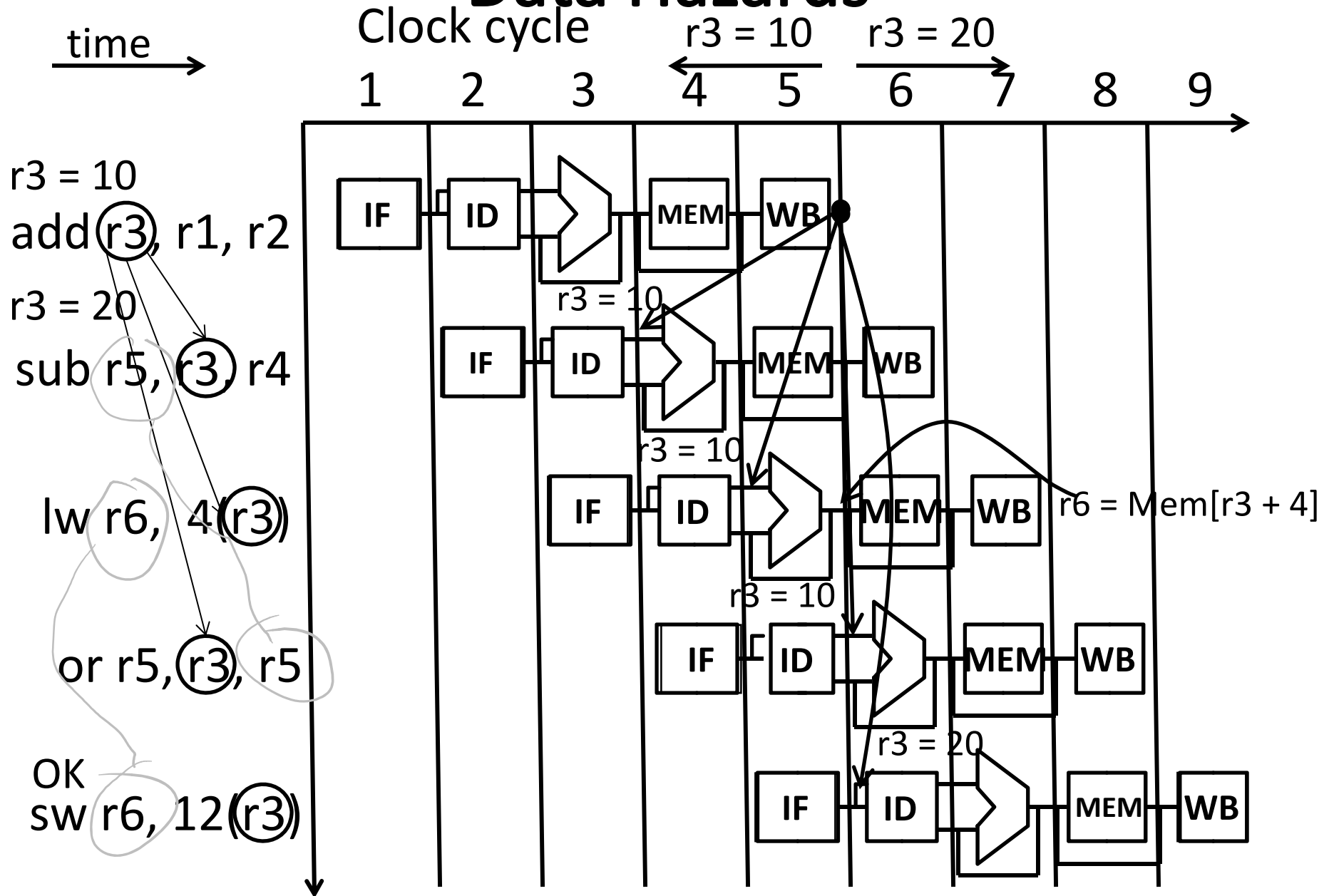
D) 4

E) 5

Data Hazards



Data Hazards



Data Hazards

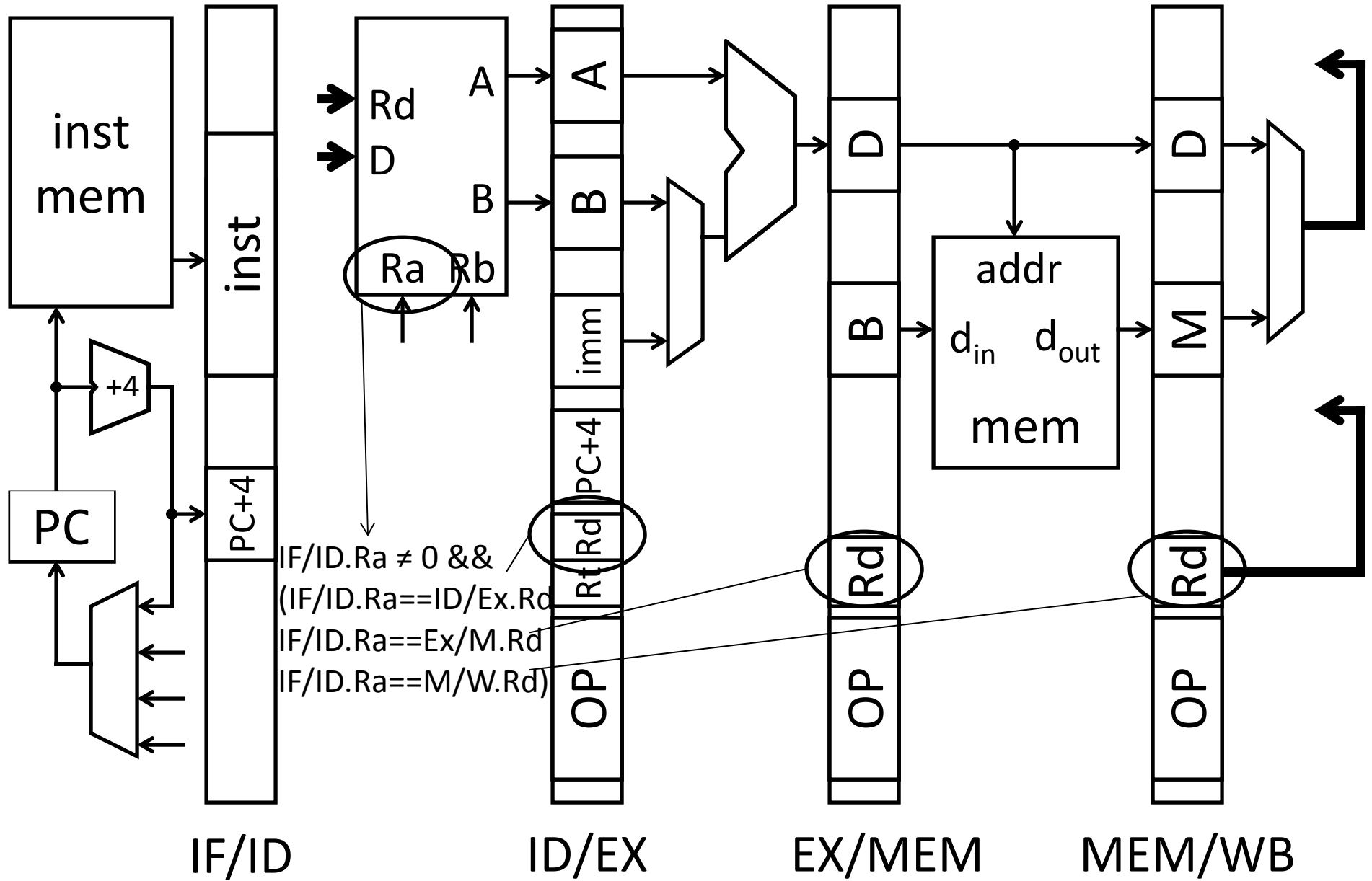
Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

i.e. add (r3), r1, r2
sub r5, (r3), r4

How to detect?

Detecting Data Hazards



Data Hazards

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

How to detect? Logic in ID stage:

stall = (IF/ID.Ra != 0 &&

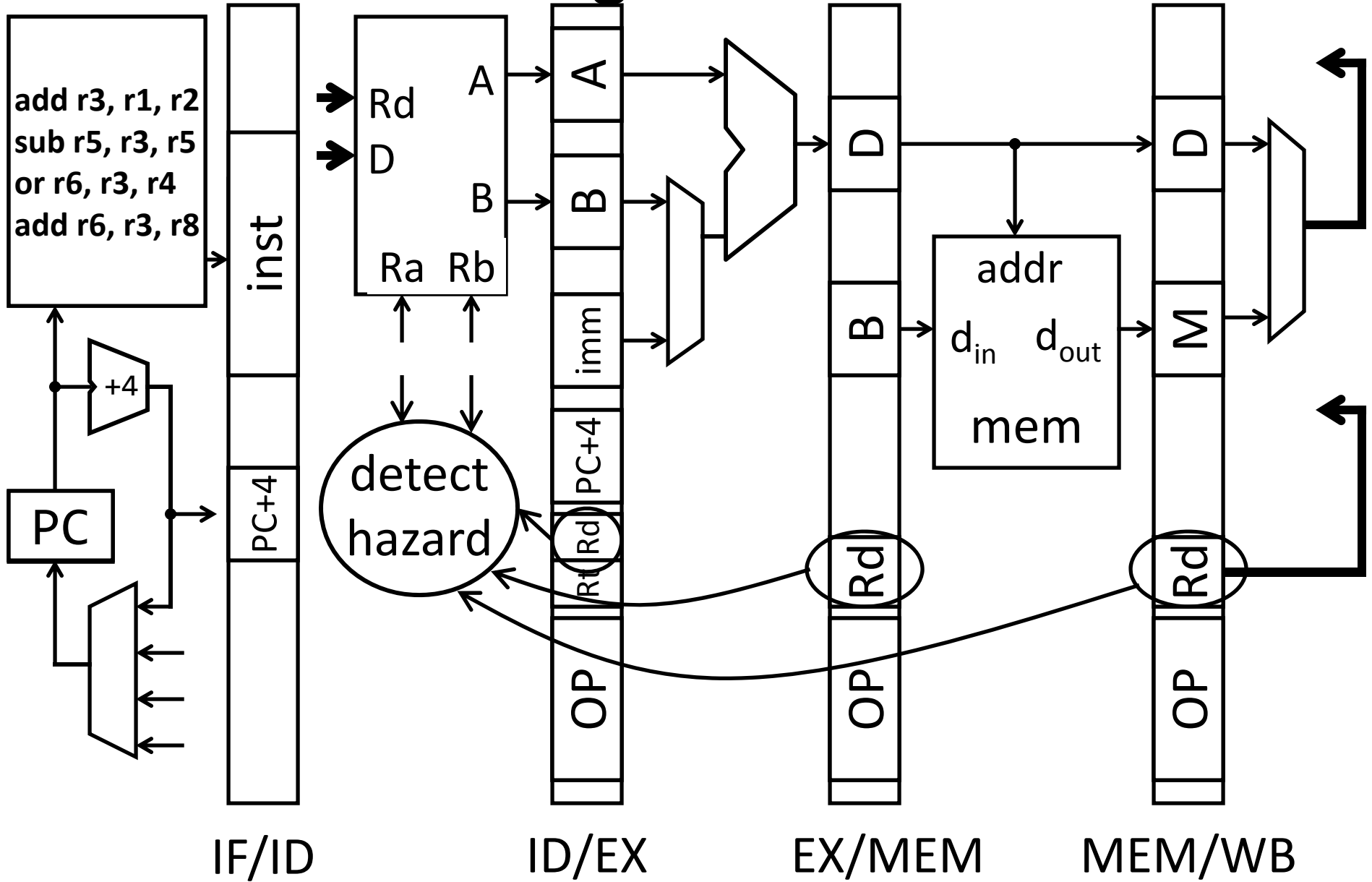
(IF/ID.Ra == ID/EX.Rd ||

IF/ID.Ra == EX/M.Rd ||

IF/ID.Ra == M/WB.Rd))

|| (same for Rb)

Detecting Data Hazards



Takeaway

Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Next Goal

What to do if data hazard detected?

Resolving Data Hazards

What to do if data hazard detected?

A) Wait/Stall

B) Reorder in Software (SW)

C) Forward/Bypass

D) All the above

E) None. We will use some other method

Resolving Data Hazards

What to do if data hazard detected?

A) Wait/Stall

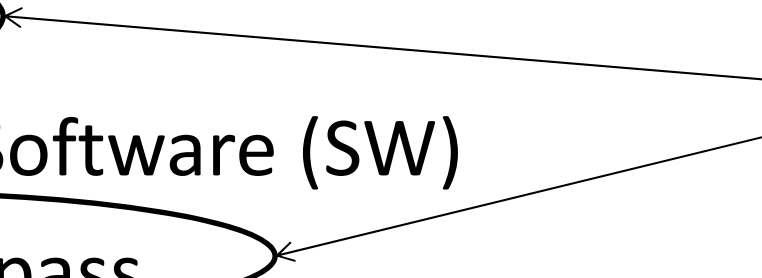
B) Reorder in Software (SW)

C) Forward/Bypass

D) All the above

E) None. We will use some other method

Discuss today



Next Goal

What to do if data hazard detected?

Options

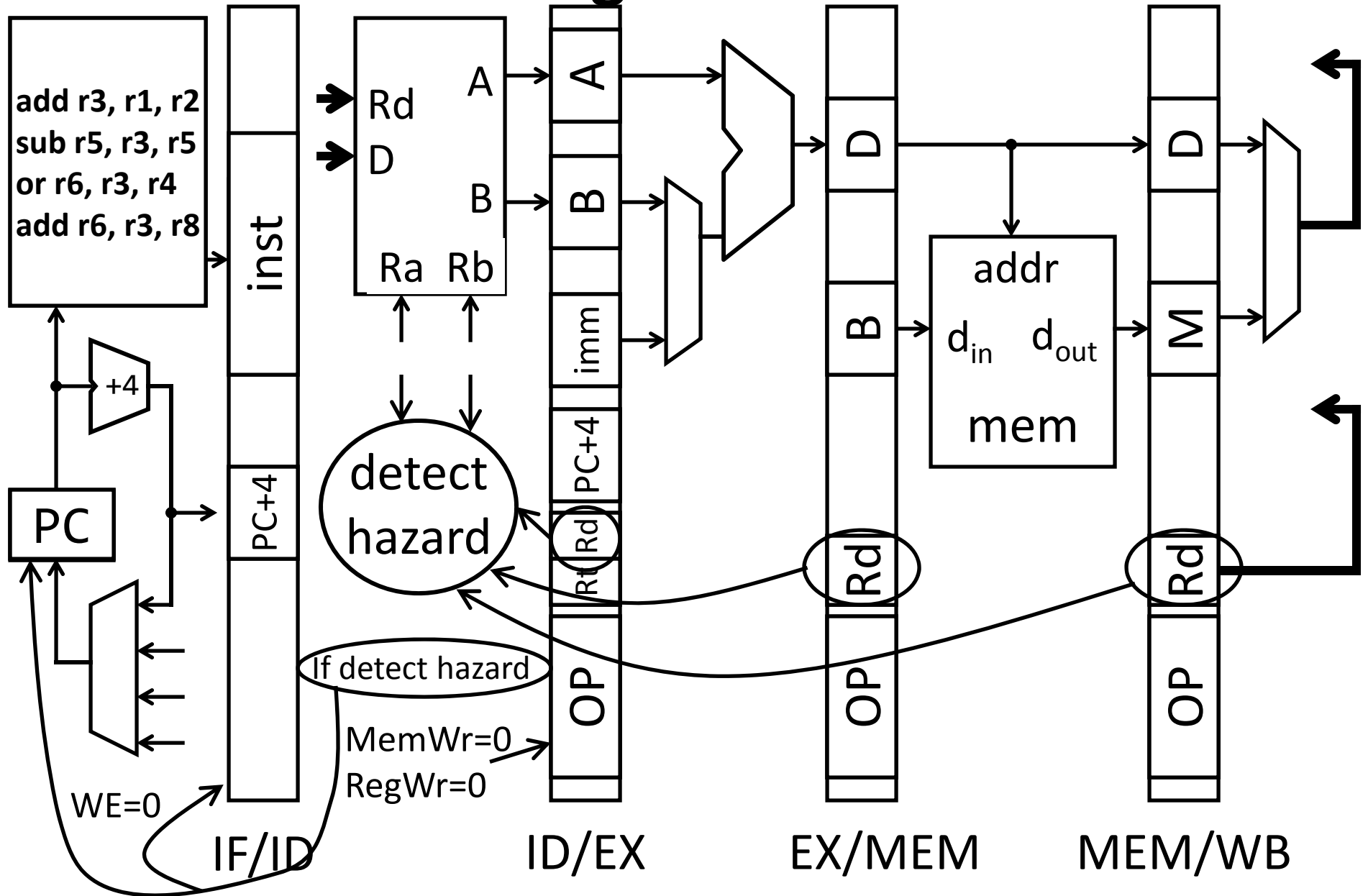
- Nothing
 - Change the ISA to match implementation
- Stall
 - Pause current and subsequent instructions till safe
- Forward/bypass
 - Forward data value to where it is needed

Stalling

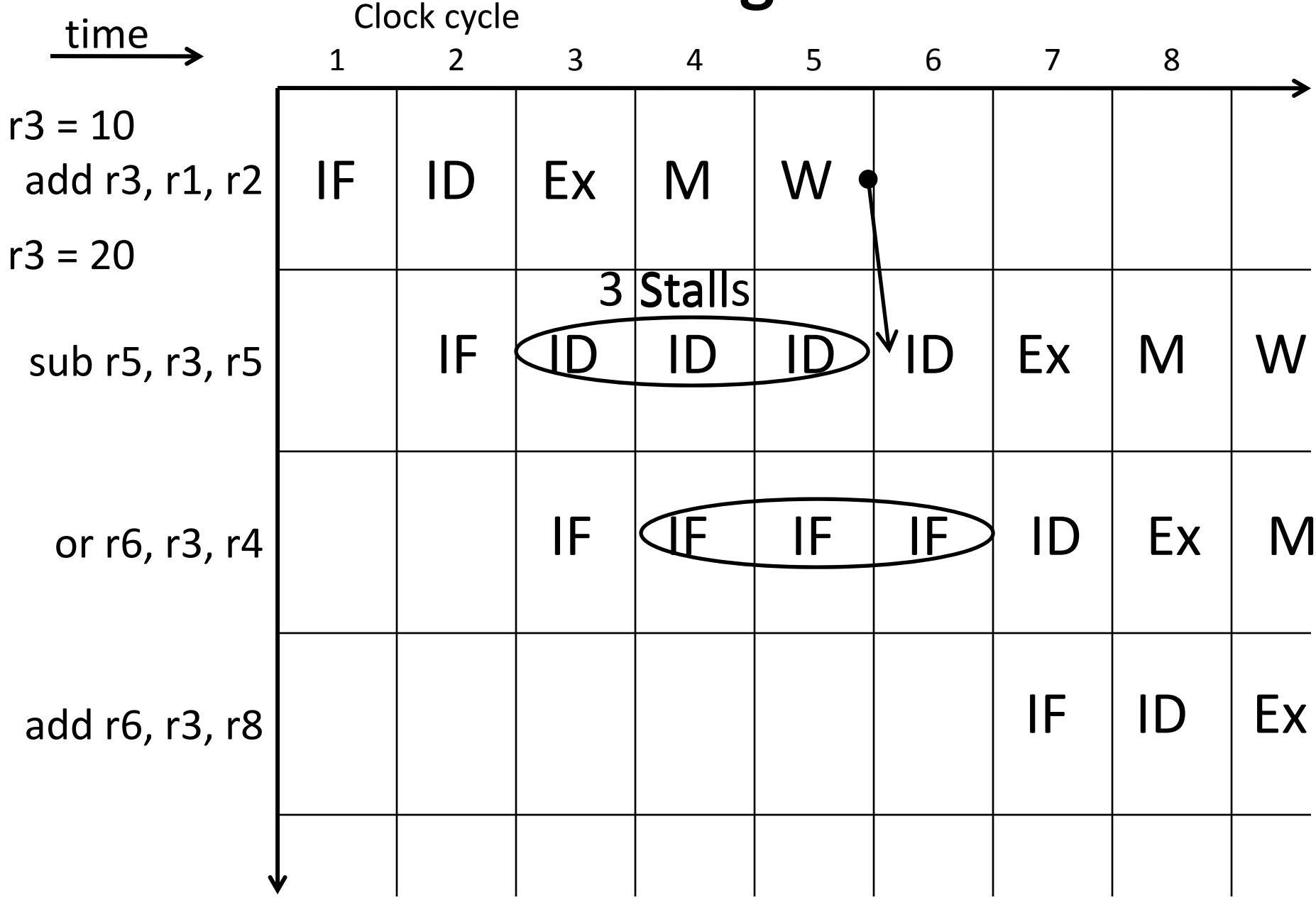
How to stall an instruction in ID stage

- prevent IF/ID pipeline register update
 - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
 - innocuous “bubble” passes through pipeline
- prevent PC update
 - stalls the next (IF stage) instruction

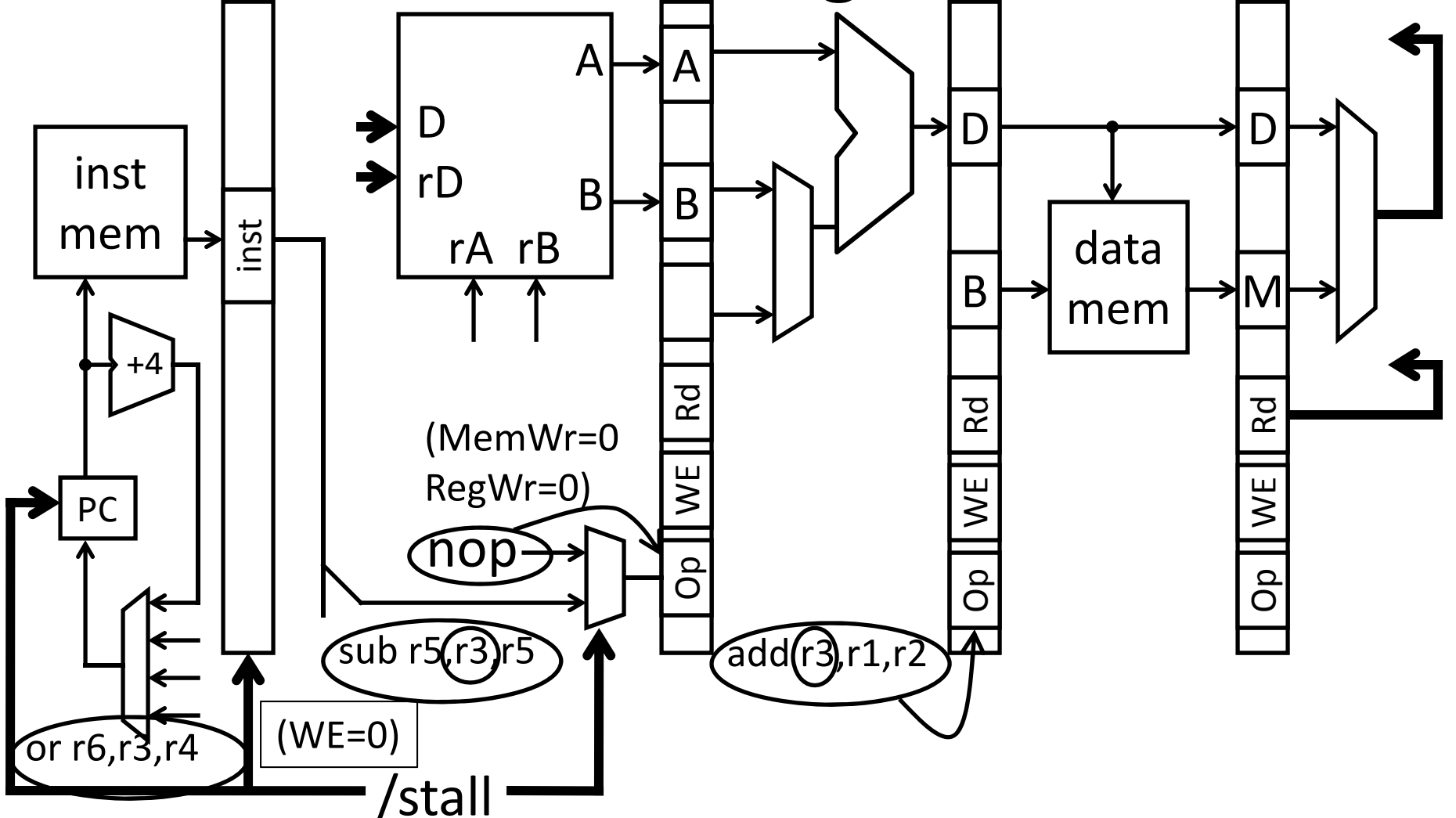
Detecting Data Hazards



Stalling

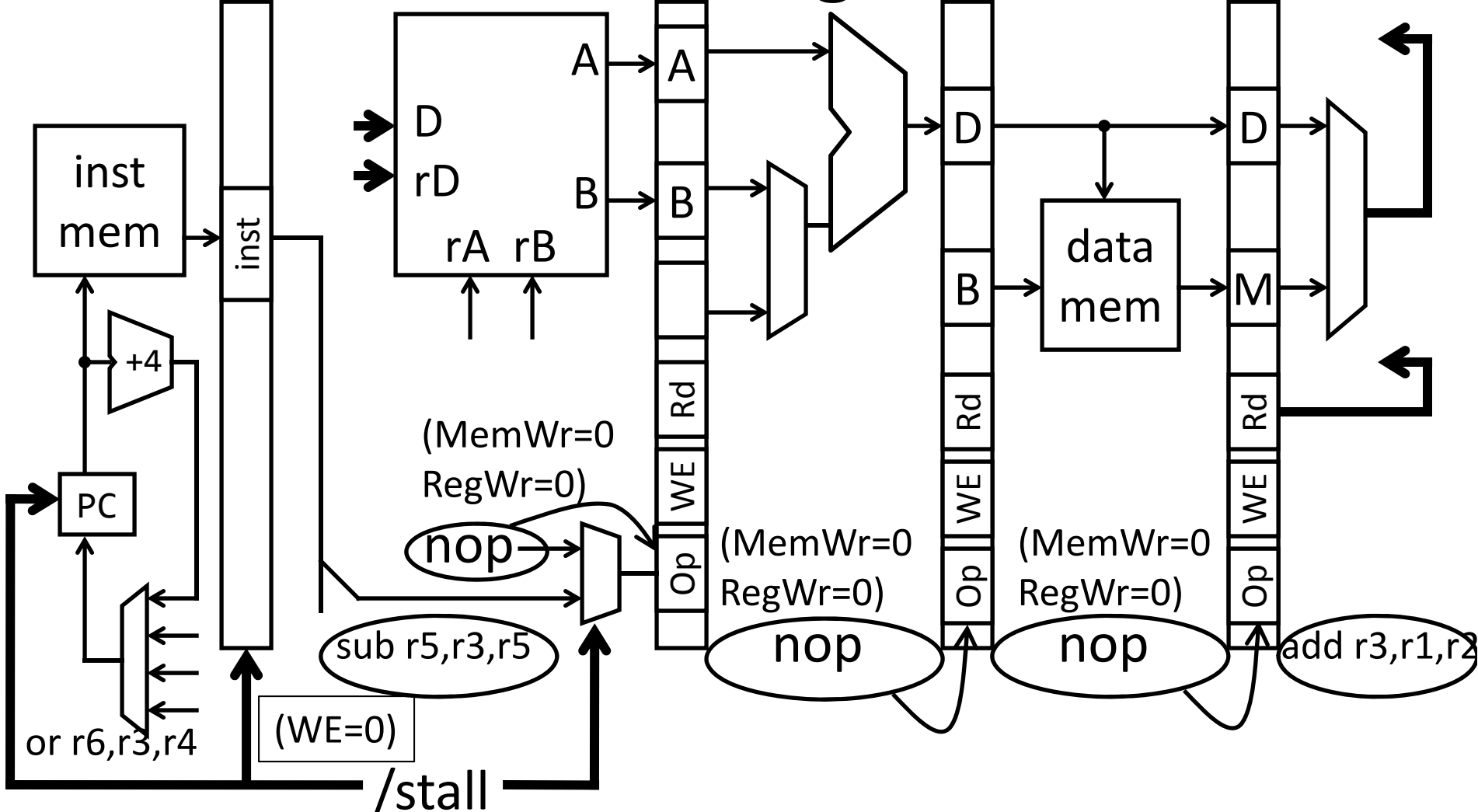


Stalling



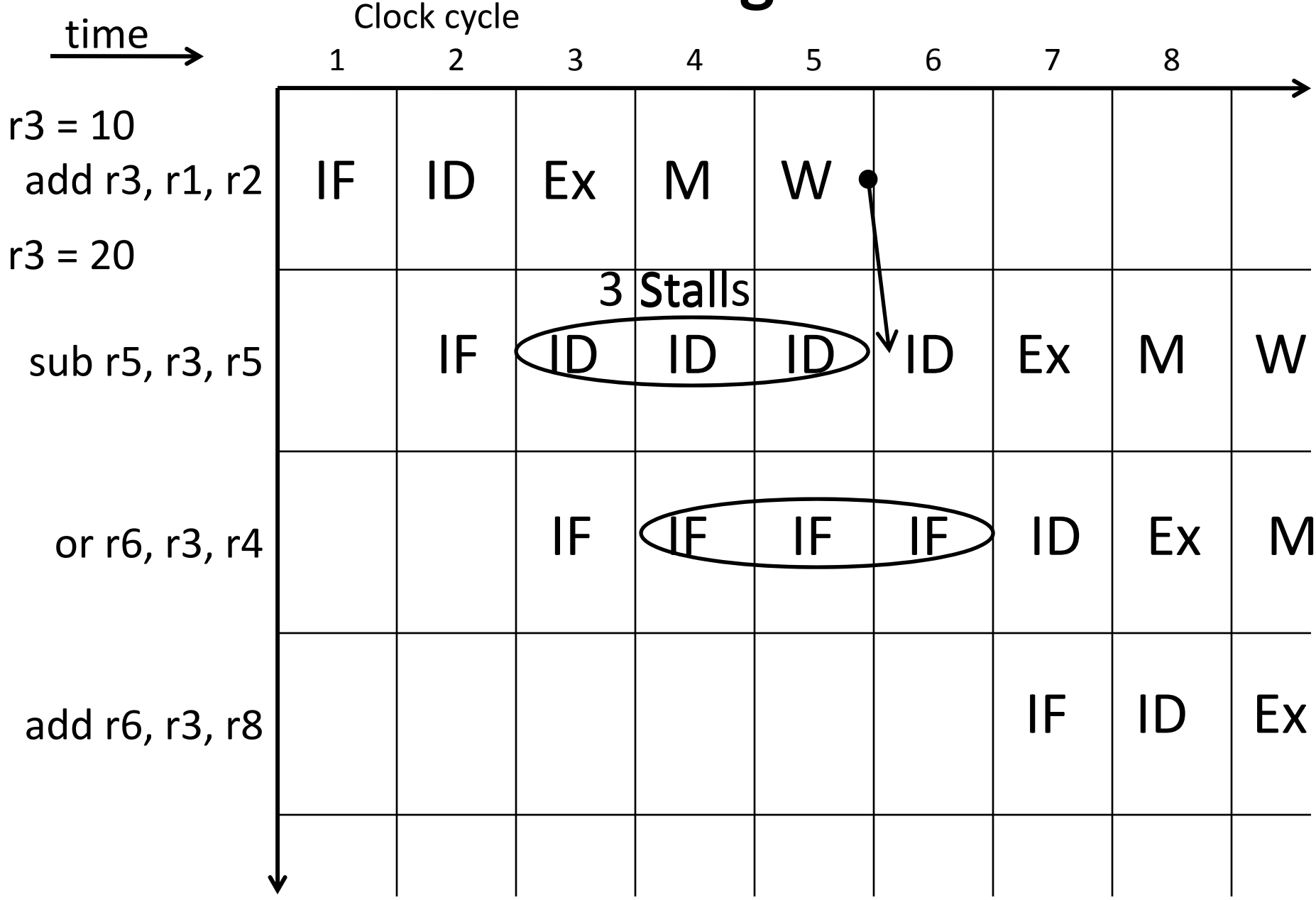
$$\text{NOP} = \text{If}(\text{IF}/\text{ID}.rA \neq 0 \ \&\& \\
 \text{(IF}/\text{ID}.rA == \text{ID}/\text{Ex}.Rd \\
 \text{IF}/\text{ID}.rA == \text{Ex}/\text{M}.Rd \\
 \text{IF}/\text{ID}.rA == \text{M}/\text{W}.Rd))$$

Stalling



$NOP = \text{If}(\text{IF}/\text{ID}.rA \neq 0 \ \&\&$
 $\quad (\text{IF}/\text{ID}.rA == \text{ID}/\text{Ex}.Rd$
 $\quad \text{IF}/\text{ID}.rA == \text{Ex}/\text{M}.Rd$
 $\quad \text{IF}/\text{ID}.rA == \text{M}/\text{W}.Rd))$

Stalling



Stalling

How to stall an instruction in ID stage

- prevent IF/ID pipeline register update
 - stalls the ID stage instruction
- convert ID stage instr into nop for later stages
 - innocuous “bubble” passes through pipeline
- prevent PC update
 - stalls the next (IF stage) instruction

Takeaway

Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards.

Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file.

*Bubbles in pipeline significantly decrease performance.

Next Goal: Resolving Data Hazards via Forwarding

What to do if data hazard detected?

A) Wait/Stall

B) Reorder in Software (SW)

C) Forward/Bypass

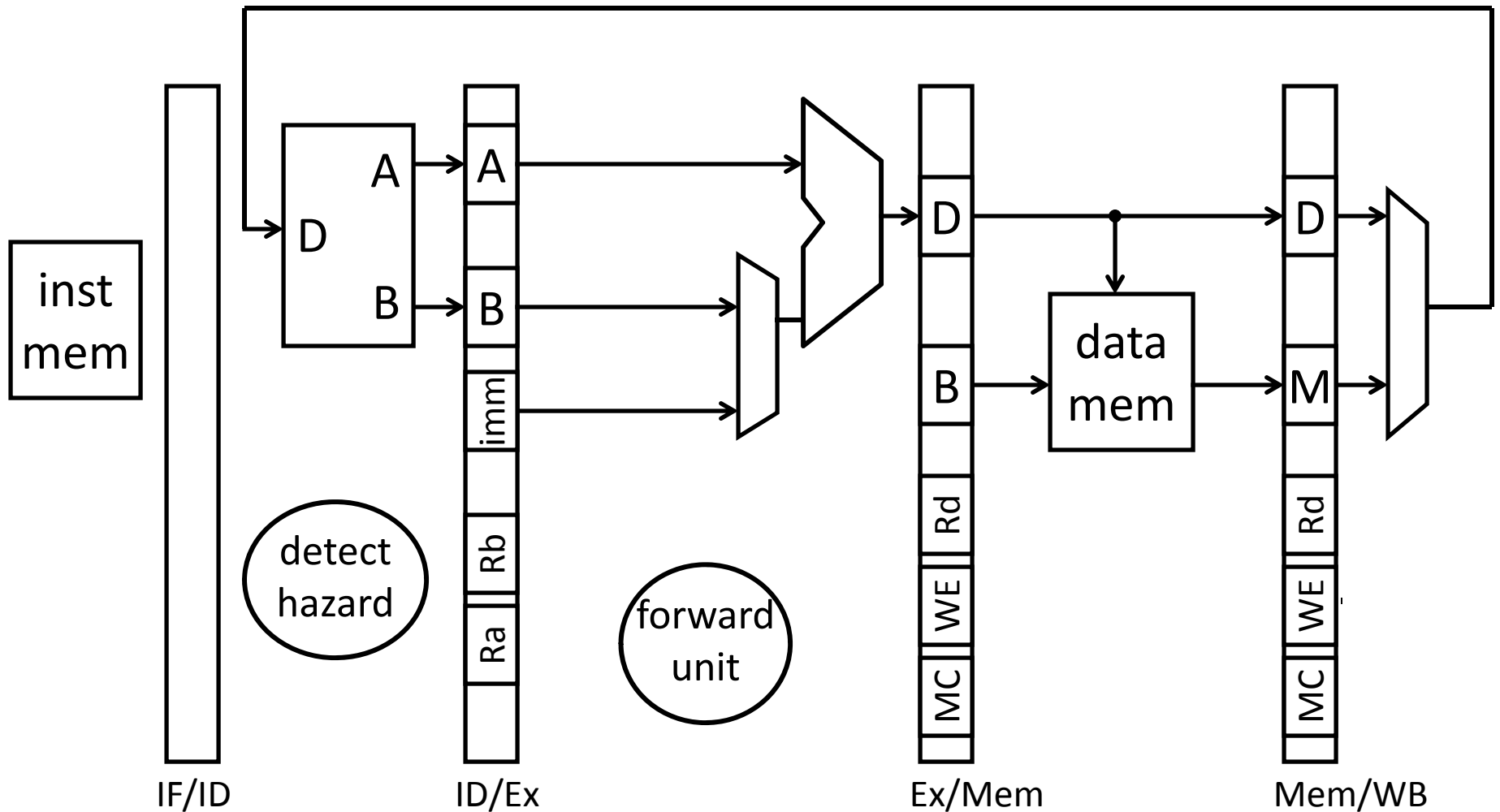
Forwarding

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).

Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W→Ex)
- RegisterFile Bypass

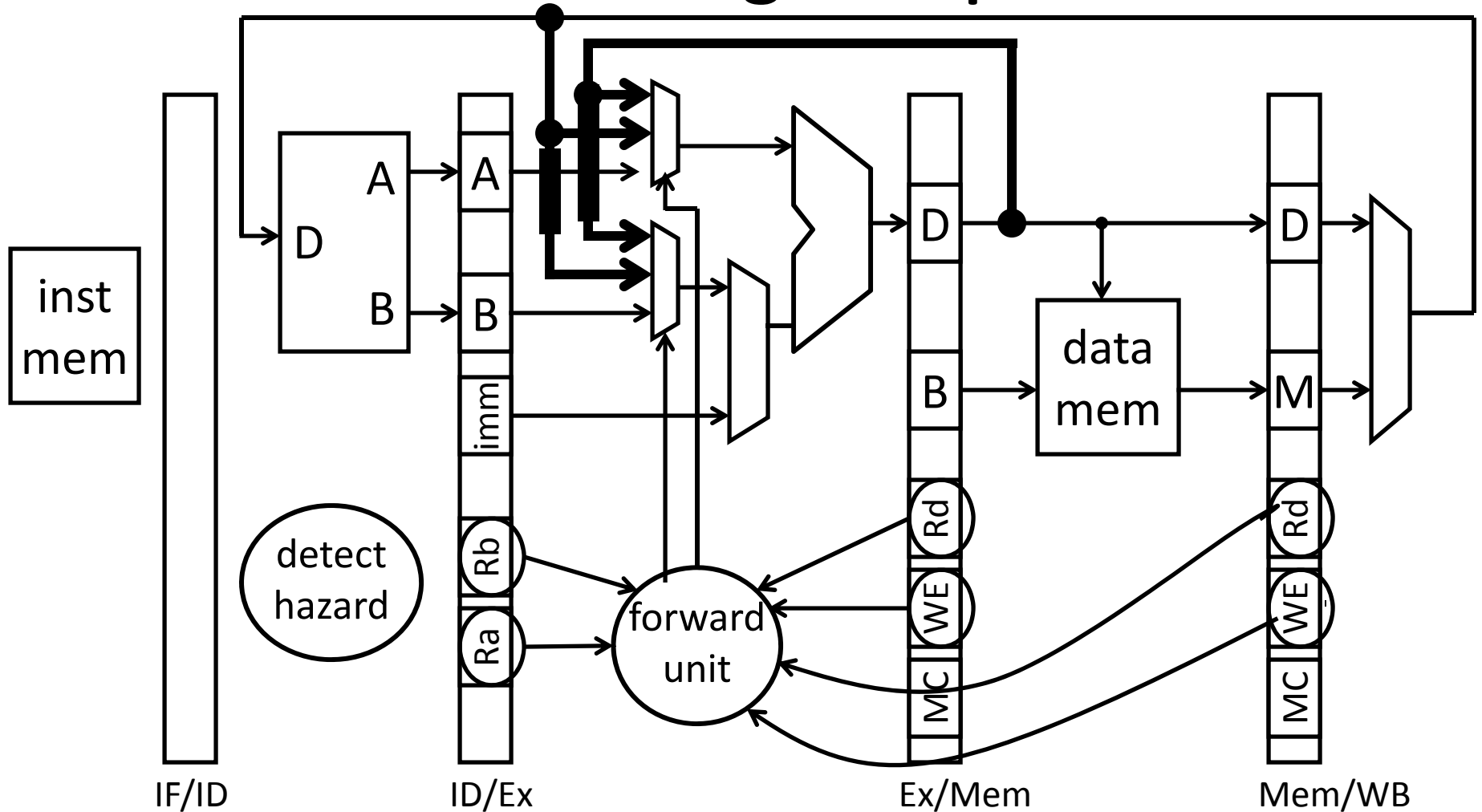
Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage ($M \rightarrow Ex$)
- Forwarding from Mem/WB register to Ex stage ($W \rightarrow Ex$)
- RegisterFile Bypass

Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

Forwarding Datapath 1

Ex/MEM to EX Bypass

- EX needs ALU result that is still in MEM stage
- Resolve:

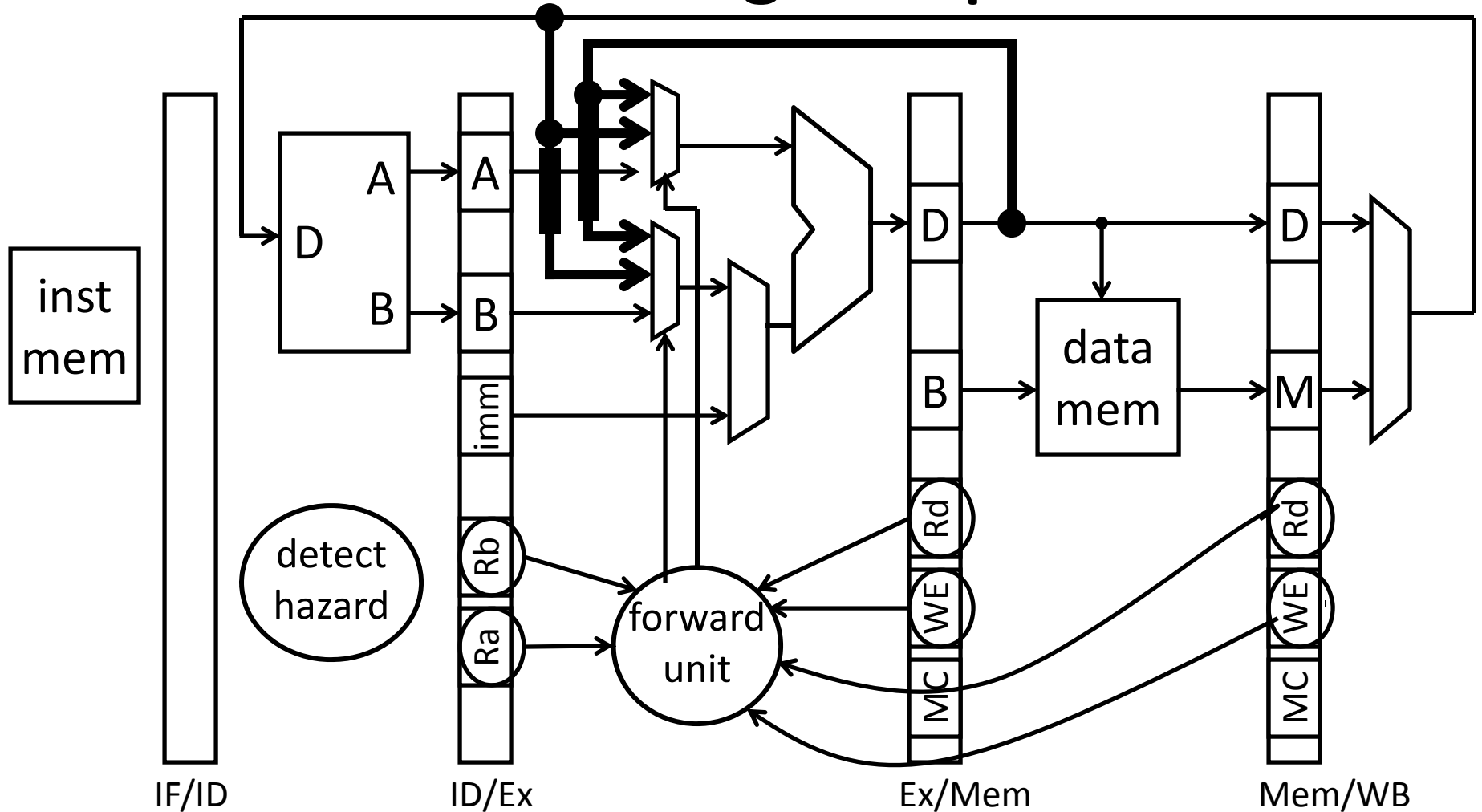
Add a bypass from EX/MEM.D to start of EX

How to detect? Logic in Ex Stage:

$$\text{forward} = (\text{Ex/M.WE} \ \&\& \ \text{EX/M.Rd} \neq 0 \ \&\& \\ \text{ID/Ex.Ra} == \text{Ex/M.Rd})$$

|| (same for Rb)

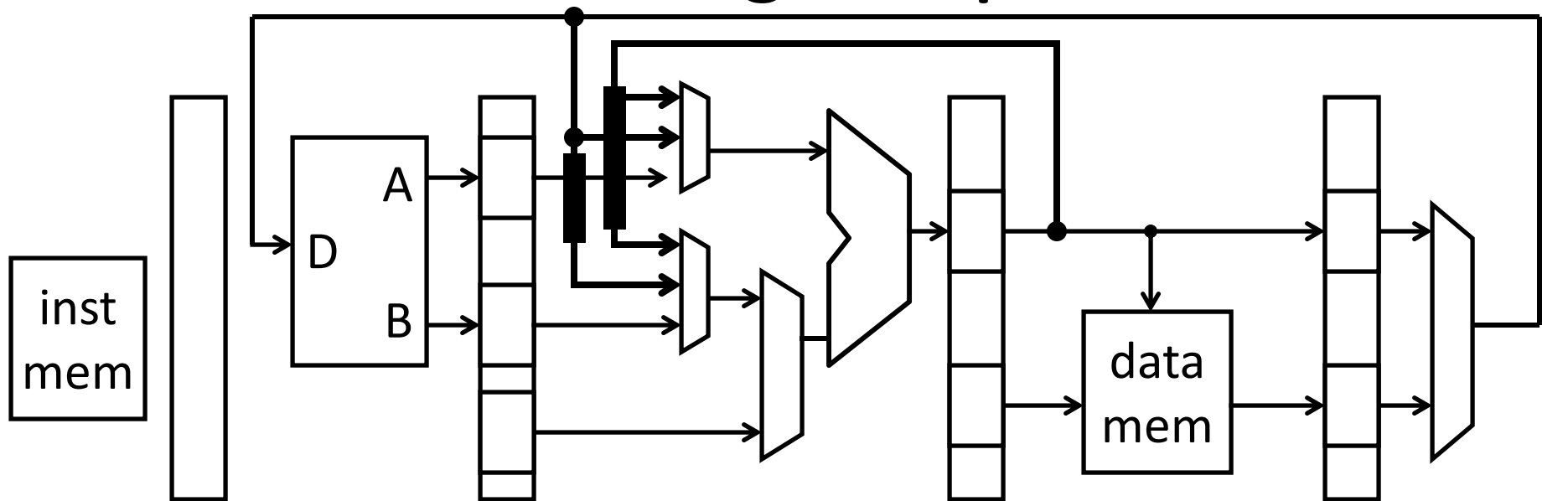
Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

Forwarding Datapath 1



add r3, r1, r2	IF	ID	Ex ●	M	W				
sub r5, r3, r1		IF	ID	Ex	M	W			

An arrow points from the dot in the 'Ex' stage of the first row to the 'Ex' stage of the second row, indicating a data hazard.

Forwarding Datapath 2

Mem/WB to EX Bypass

- EX needs value being written by WB
- Resolve:

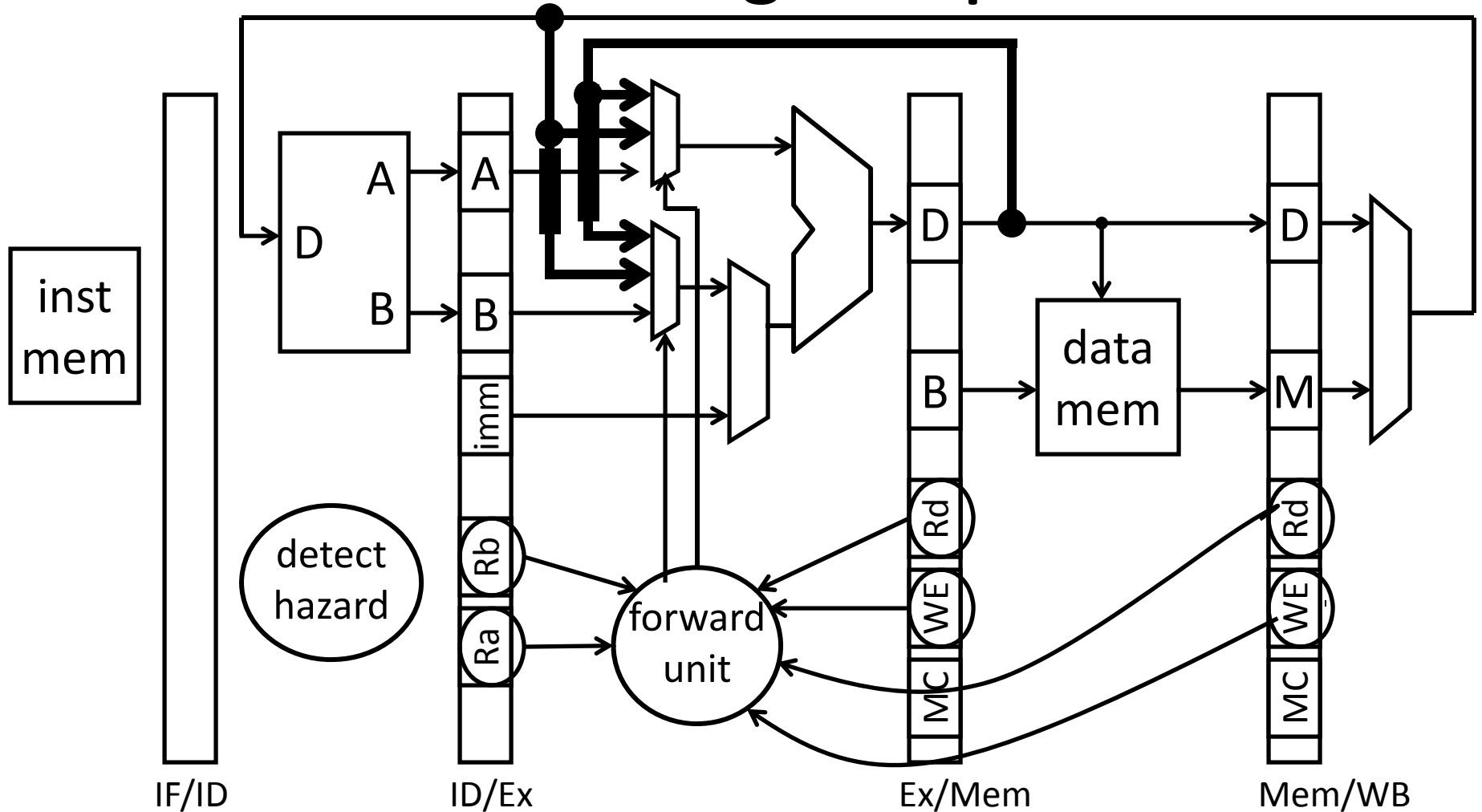
Add bypass from WB final value to start of EX

How to detect? Logic in Ex Stage:

$$\begin{aligned} \text{forward} = & (\text{M/WB.WE} \ \&\& \ \text{M/WB.Rd} \ != \ 0 \ \&\& \\ & \text{ID/Ex.Ra} \ == \ \text{M/WB.Rd} \ \&\& \\ & \text{not} \ (\text{Ex/M.WE} \ \&\& \ \text{Ex/M.Rd} \ != \ 0 \ \&\& \\ & \text{ID/Ex.Ra} \ == \ \text{Ex/M.Rd}) \\ & || \ (\text{same for Rb}) \end{aligned}$$

Check pg. 311

Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

Register File Bypass

Register File Bypass

- Reading a value that is currently being written

Detect:

$((Ra == MEM/WB.Rd) \text{ or } (Rb == MEM/WB.Rd))$
and (WB is writing a register)

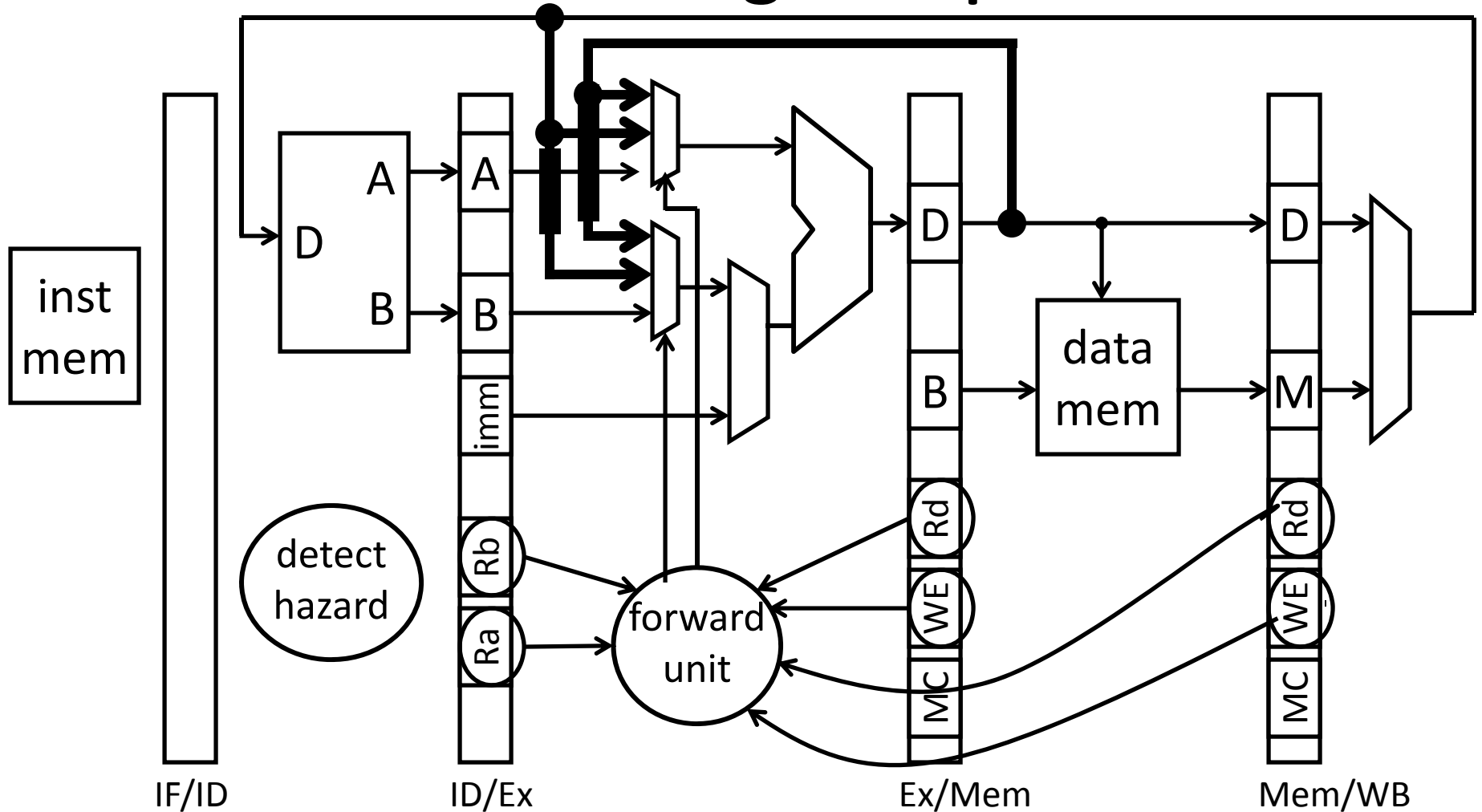
Resolve:

Add a bypass around register file (WB to ID)

Better: (Hack) just negate register file clock

- writes happen at end of first half of each clock cycle
- reads happen during second half of each clock cycle

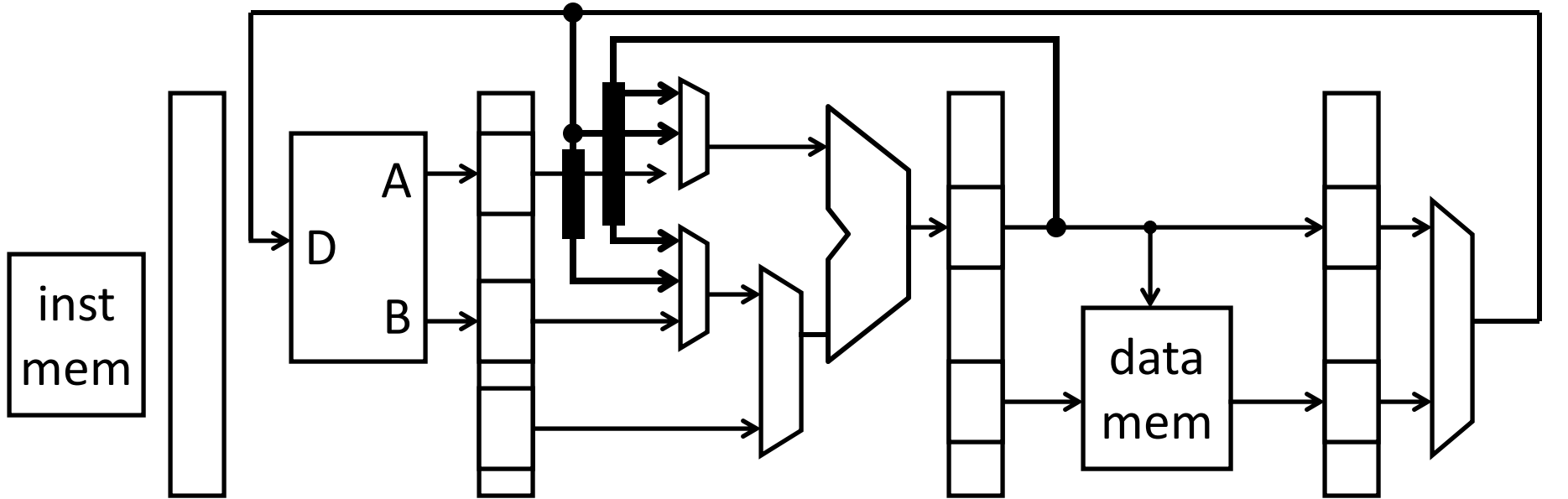
Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M → Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

Register File Bypass

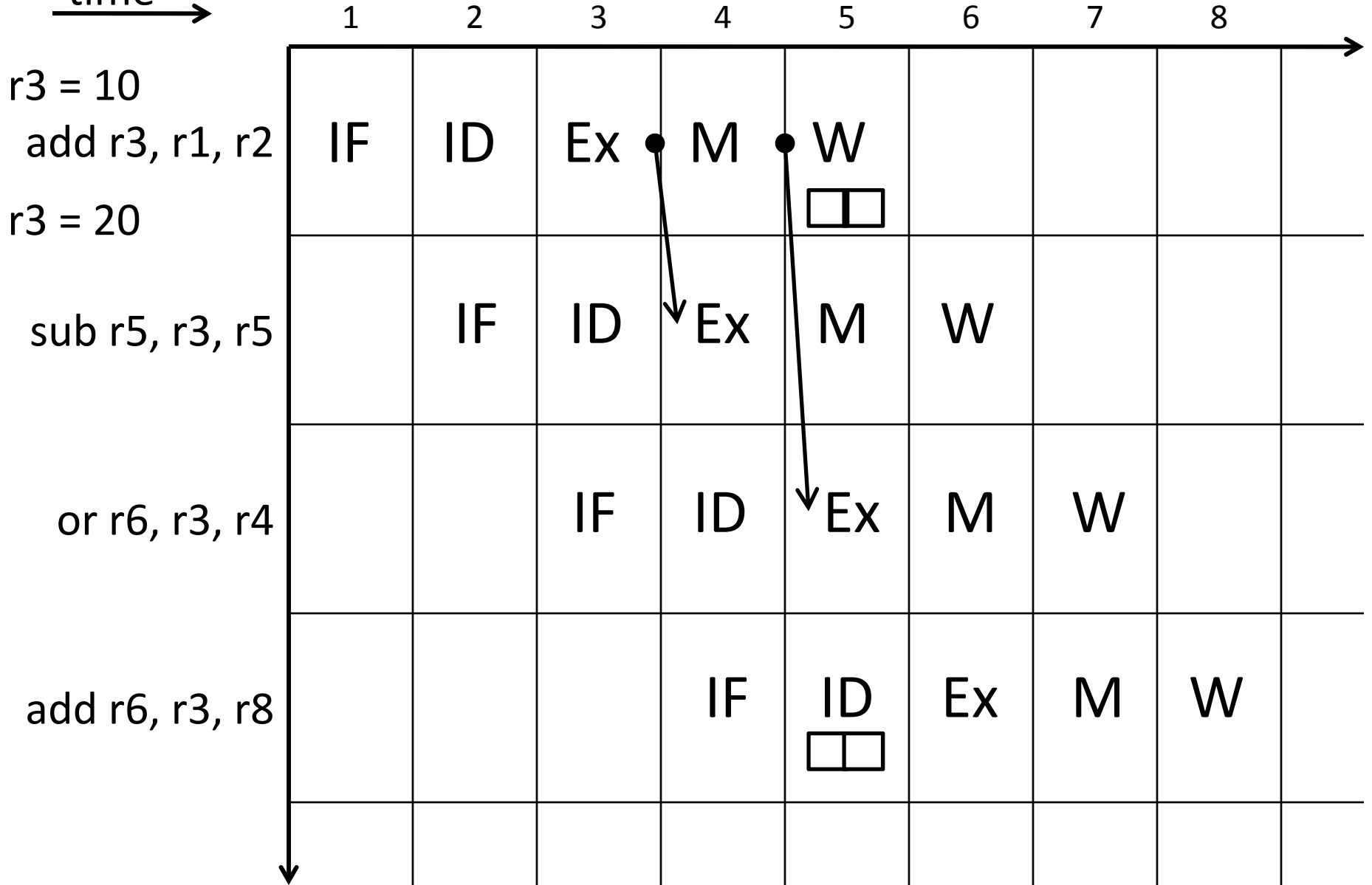


add r3, r1, r2	IF	ID	Ex	M	W			
sub r5, r3, r1		IF	ID	Ex	M	W		
or r6, r3, r4			IF	ID	Ex	M	W	
add r6, r3, r8				IF	ID	Ex	M	W

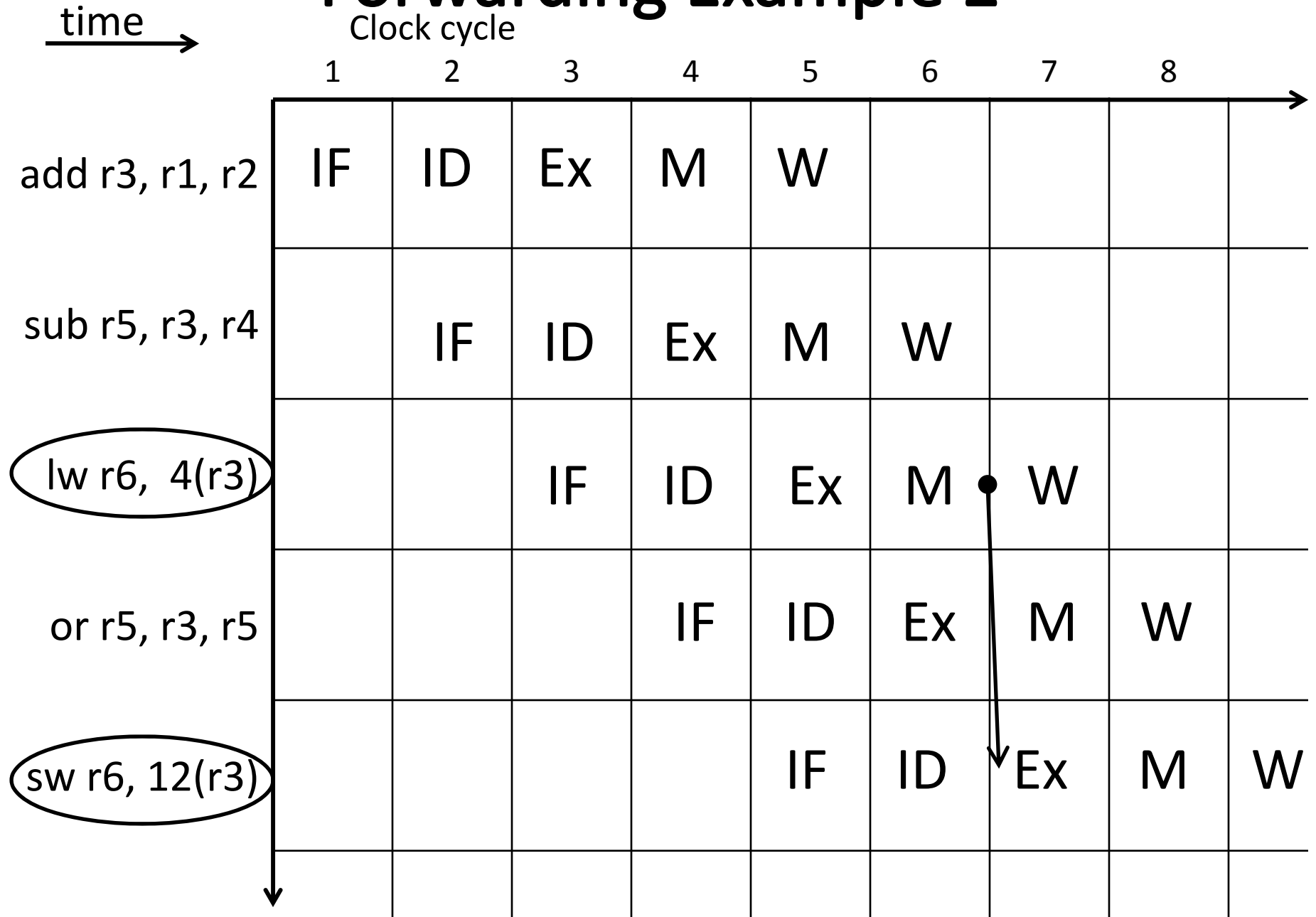
Forwarding Example

Clock cycle

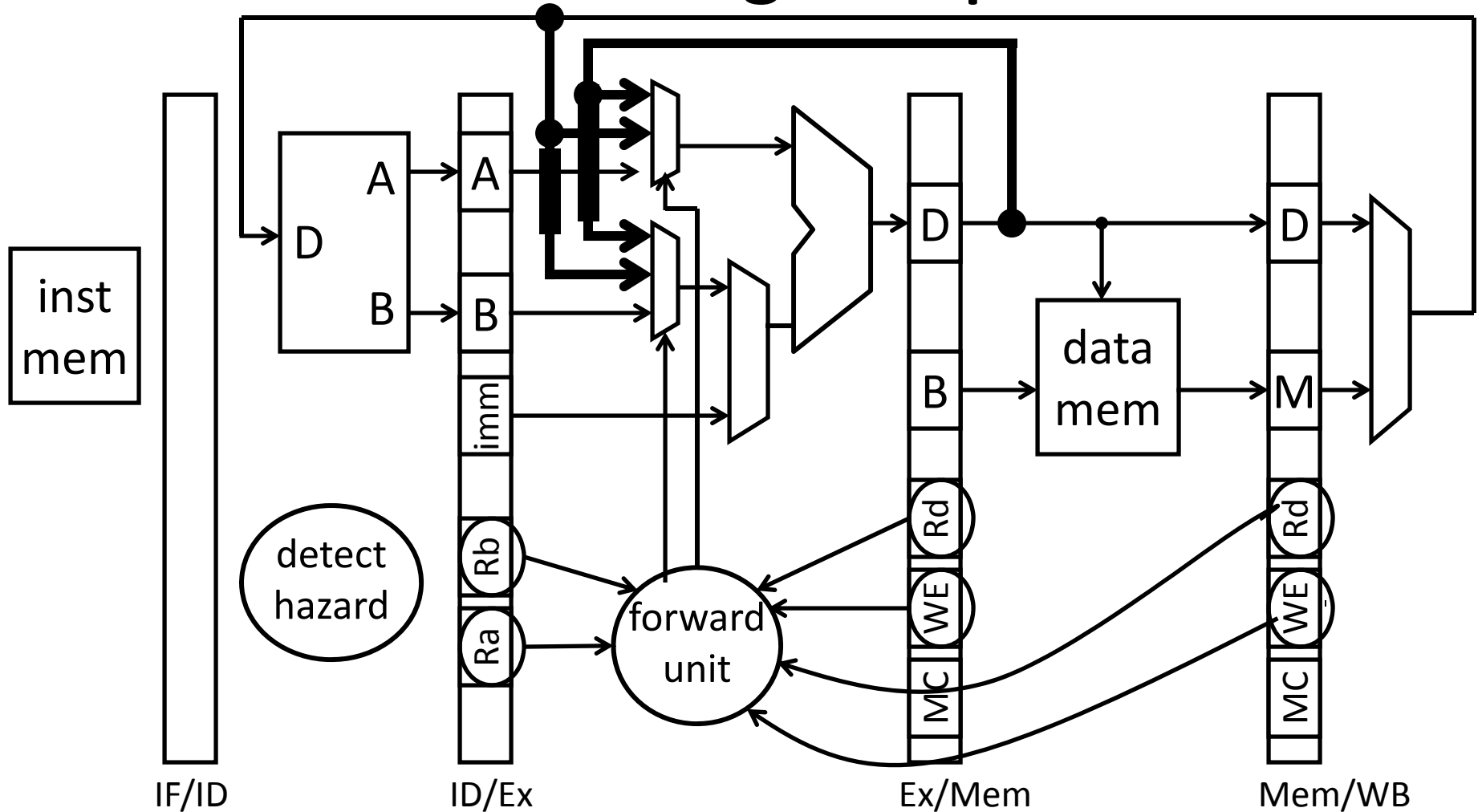
time →



Forwarding Example 2



Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- Register File Bypass

Takeaway

Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. A pipelined processor needs to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Bubbles (nops) in pipeline significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.

Data Hazard Recap

Stall

- Pause current and all subsequent instructions

Forward/Bypass

- Try to steal correct value from elsewhere in pipeline
- Otherwise, fall back to stalling or require a delay slot

Tradeoffs?