

# Pipelining

**Hakim Weatherspoon**

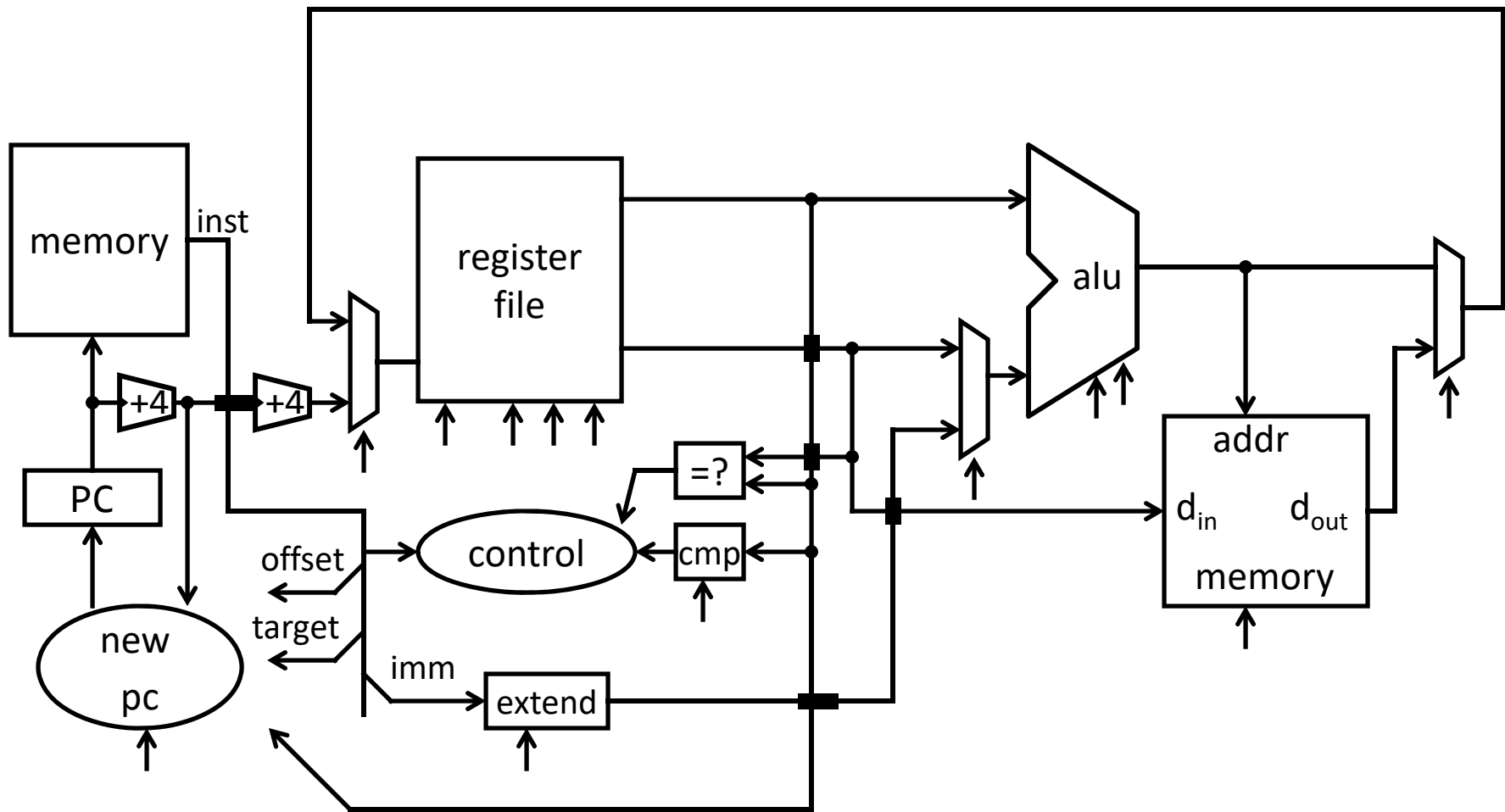
**CS 3410**

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

# Review: Single Cycle Processor



# Review: Single Cycle Processor

## Advantages

- Single cycle per instruction make logic and clock simple

## Disadvantages

- Since instructions take different time to finish, memory and functional unit are not efficiently utilized
- Cycle time is the longest delay
  - Load instruction
- Best possible CPI is 1 (actually  $< 1$  w parallelism)
  - However, lower MIPS and longer clock period (lower clock frequency); hence, lower performance

# Review: Multi Cycle Processor

## Advantages

- Better MIPS and smaller clock period (higher clock frequency)
- Hence, better performance than Single Cycle processor

## Disadvantages

- Higher CPI than single cycle processor

## Pipelining: Want better Performance

- want small CPI (close to 1) with high MIPS and short clock period (high clock frequency)

# Improving Performance

Parallelism

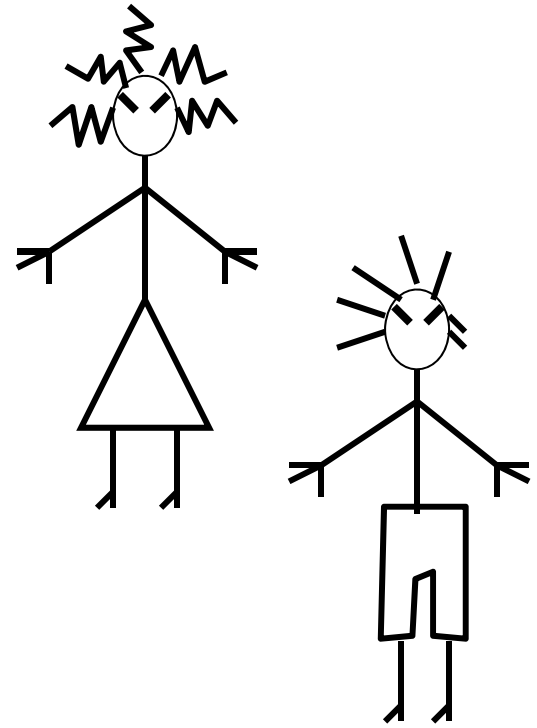
Pipelining

Both!

# The Kids

Alice

Bob



They don't always get along...

# The Bicycle



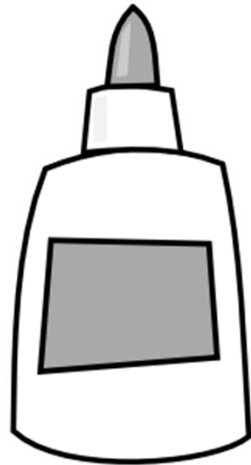
# The Materials



Saw



Drill



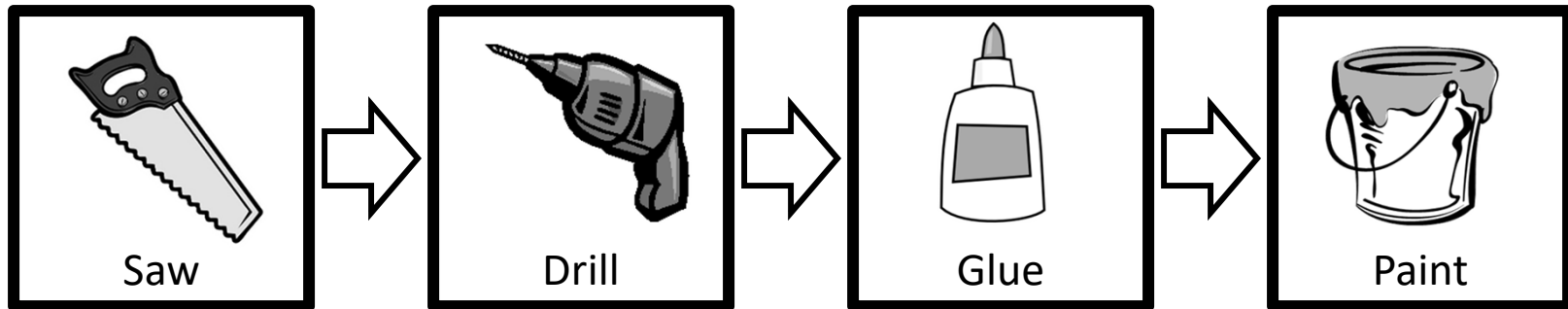
Glue



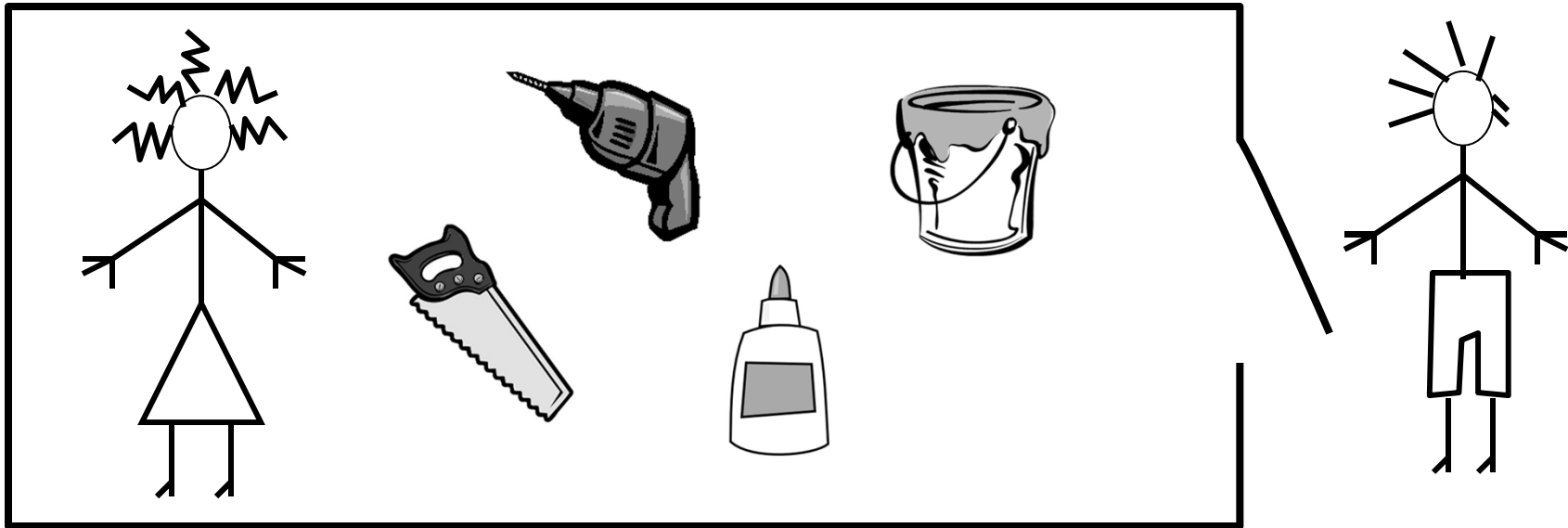
Paint

# The Instructions

N pieces, each built following same sequence:



# Design 1: Sequential Schedule



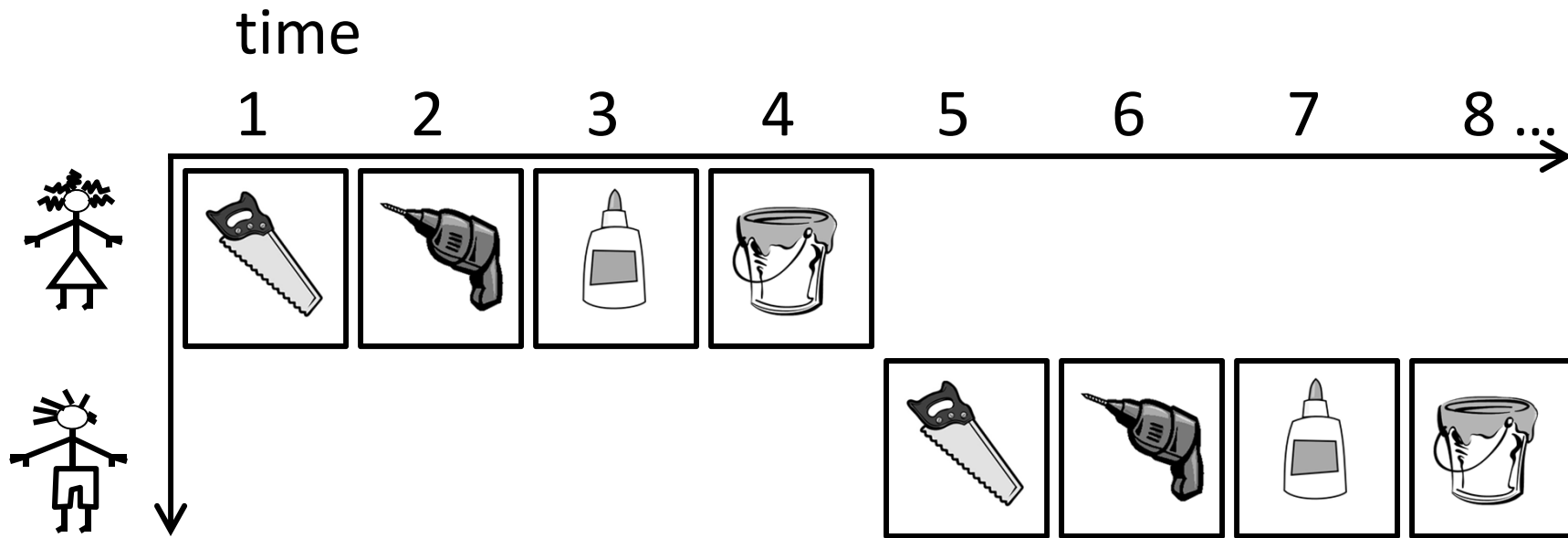
Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

# Sequential Performance



Latency: 4 hours/task

Throughput: 1 task/4 hrs

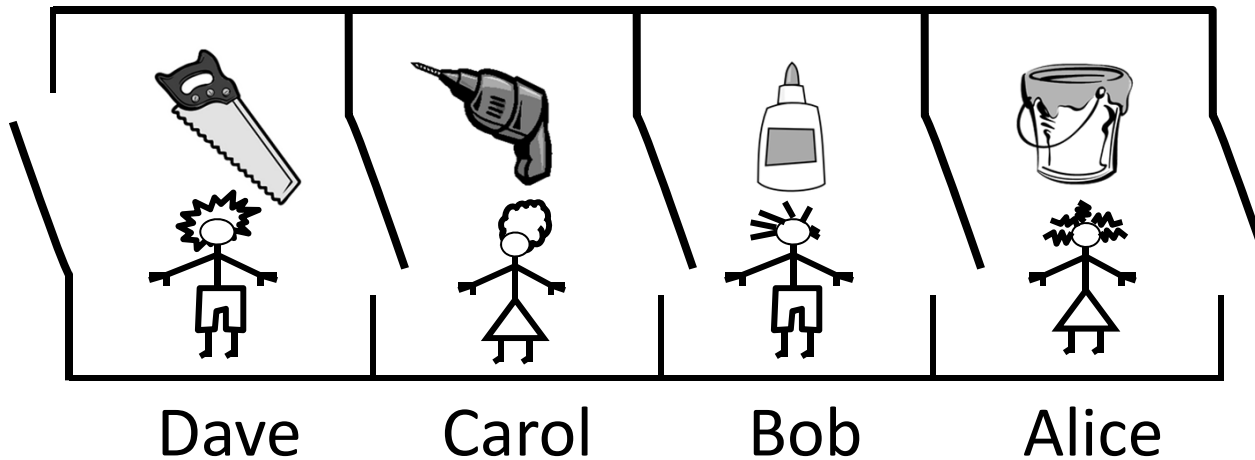
Concurrency: 1

Can we do better?

CPI = 4

# Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



One person owns a stage at a time

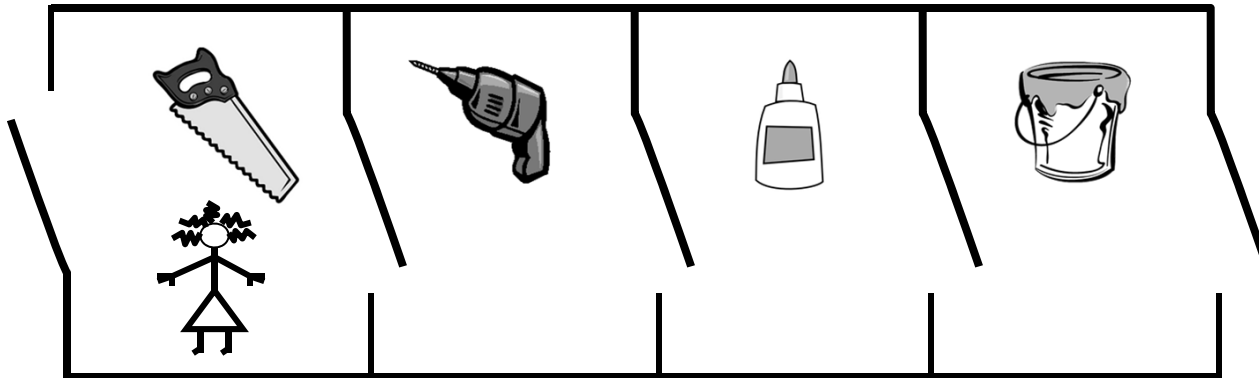
4 stages

4 people working simultaneously

Everyone moves right in lockstep

# Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



Alice

One person owns a stage at a time

4 stages

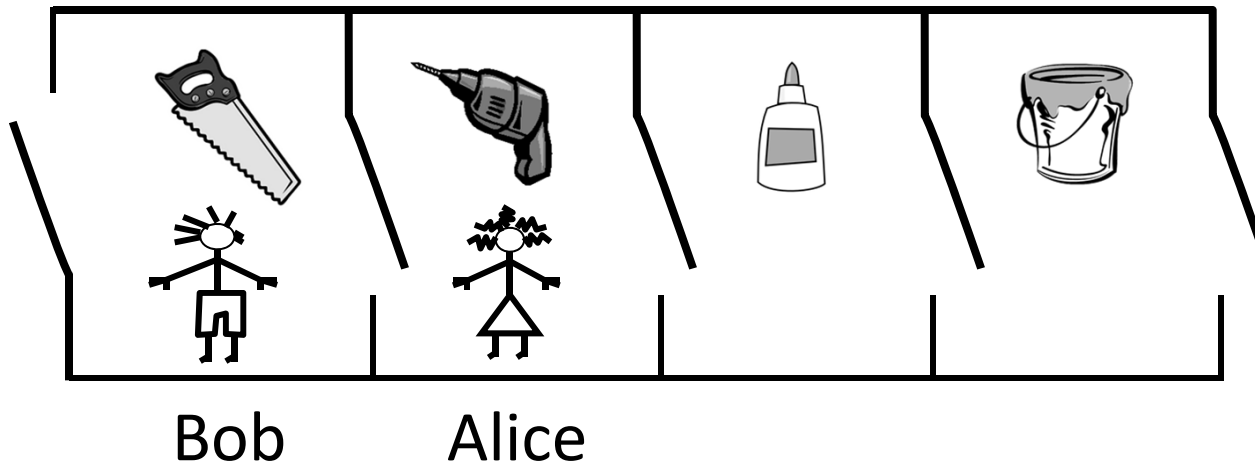
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

# Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



One person owns a stage at a time

4 stages

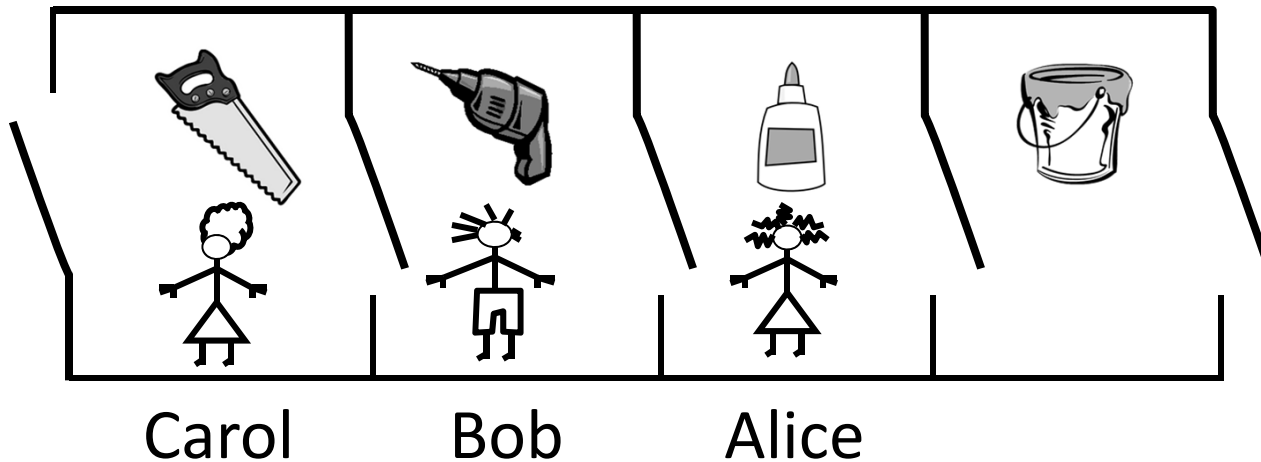
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

# Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



One person owns a stage at a time

4 stages

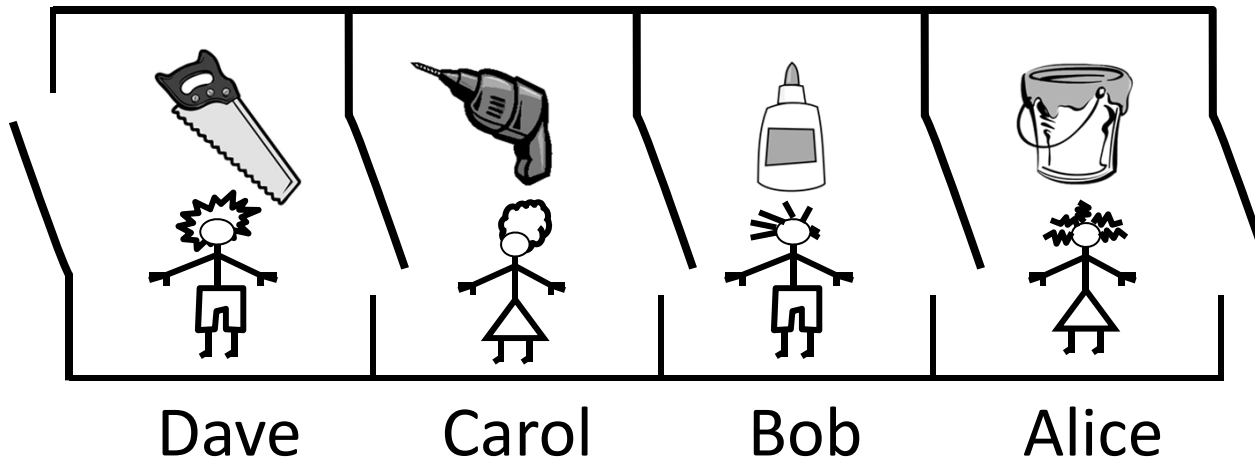
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

# Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



One person owns a stage at a time

4 stages

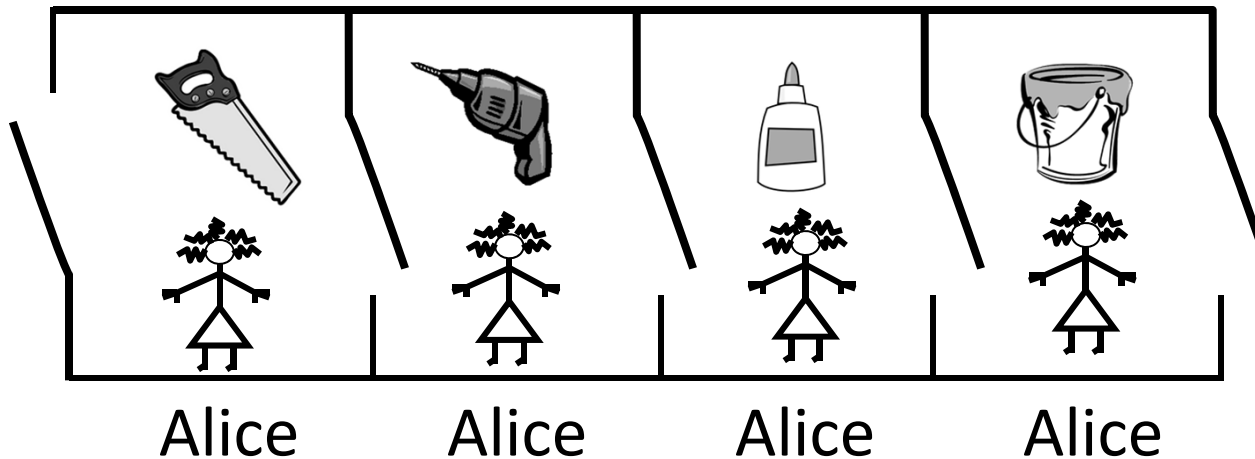
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

# Design 2: Pipelined Design

Partition room into *stages* of a *pipeline*



One person owns a stage at a time

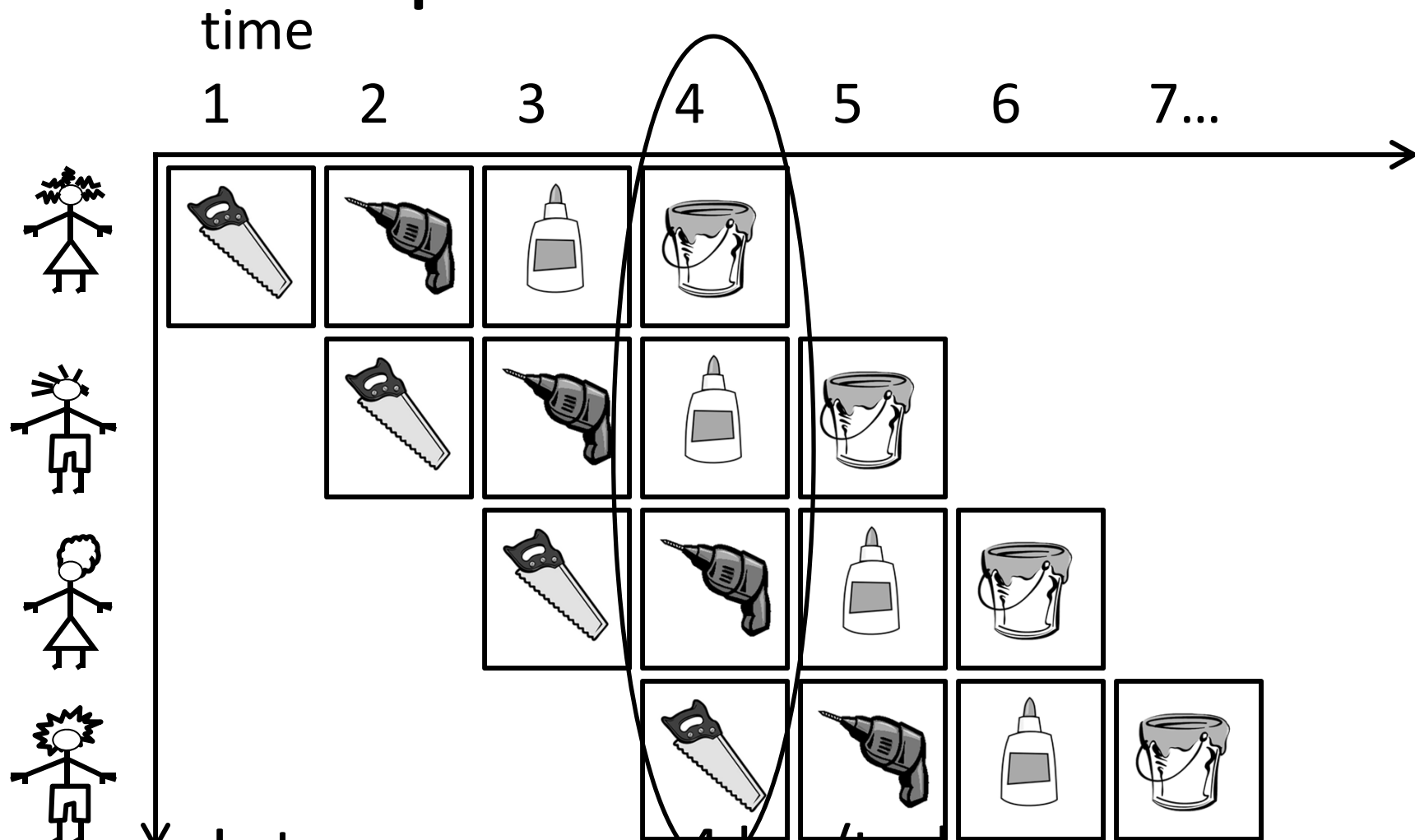
4 stages

4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

# Pipelined Performance



Latency:

4 hrs/task

Throughput:

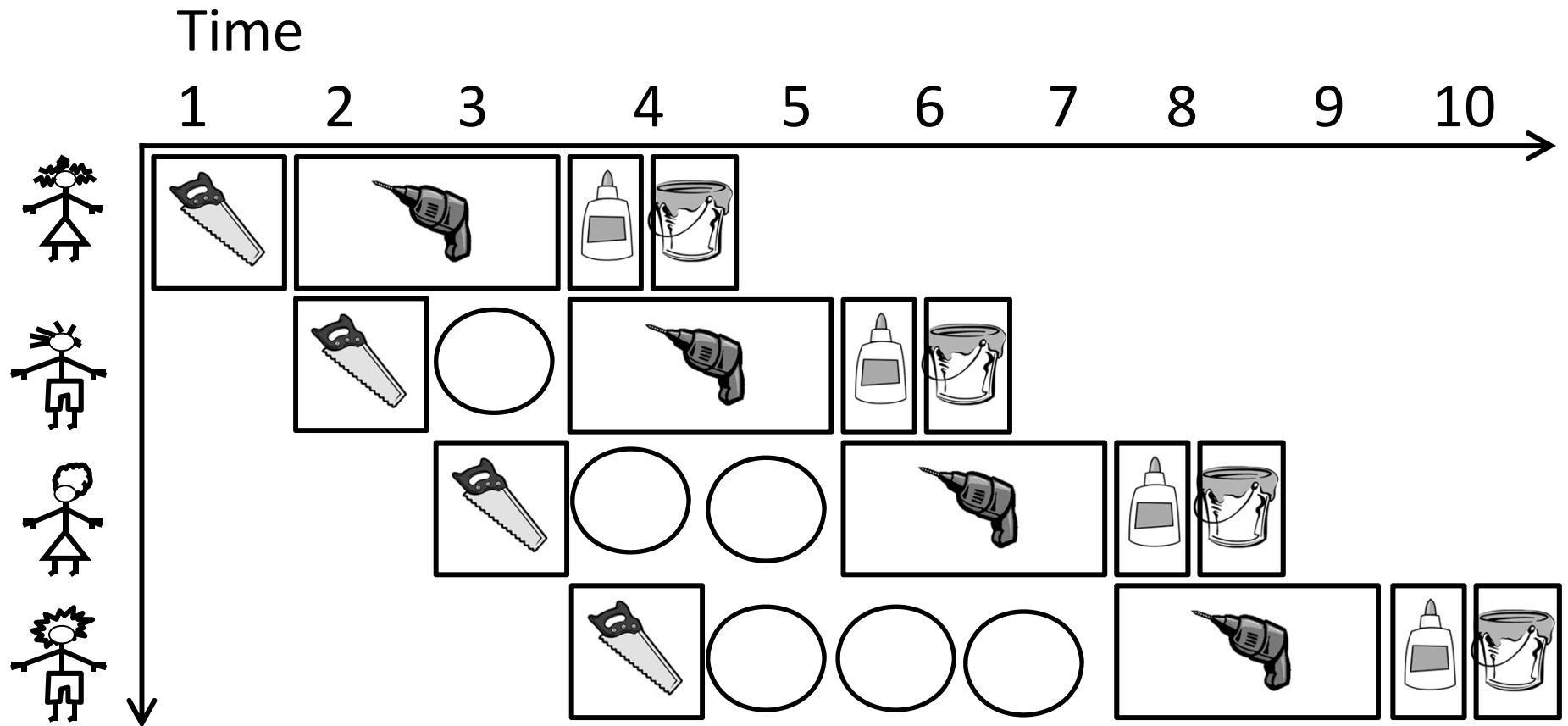
1 task/hr

Concurrency:

4

CPI = 1

# Pipelined Performance



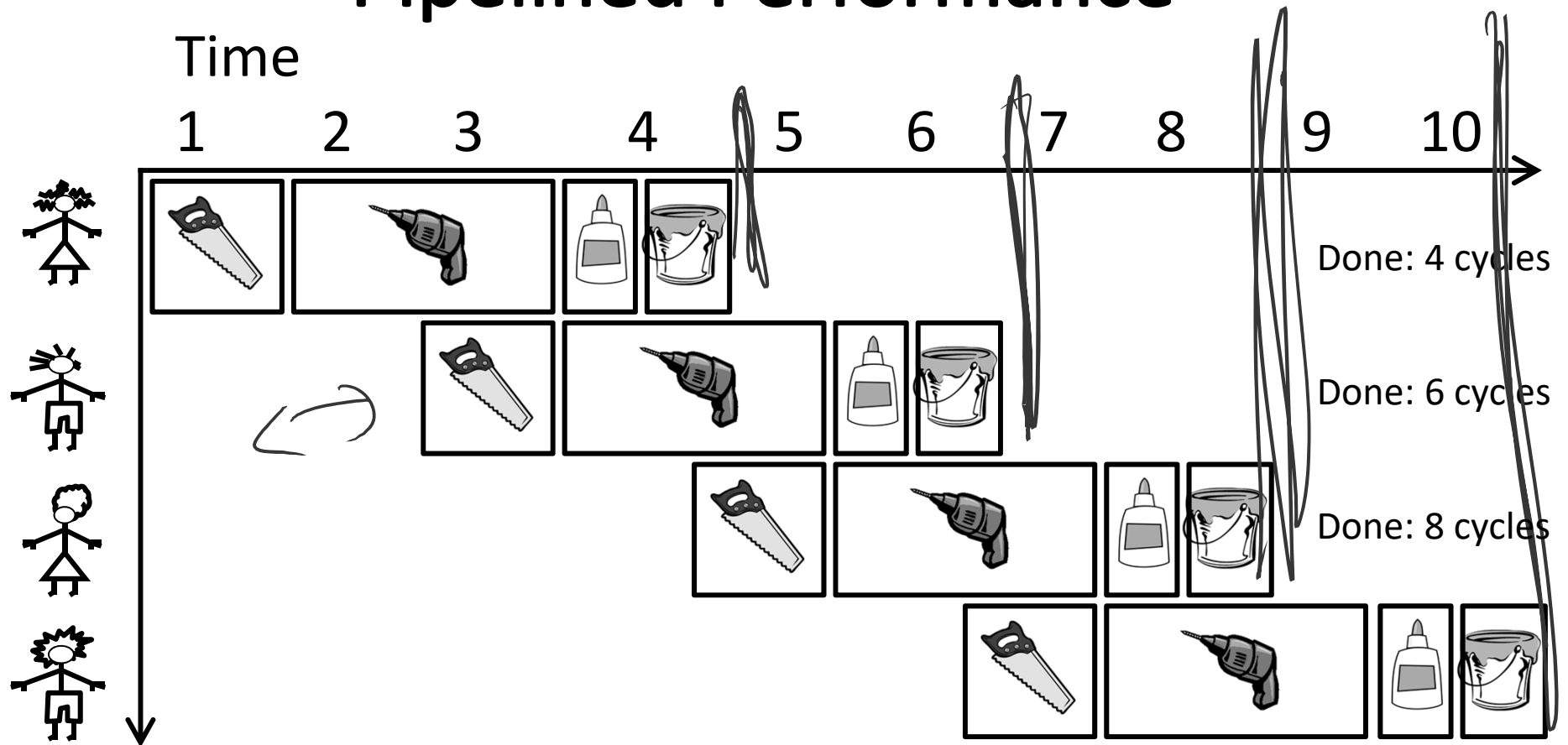
What if drilling takes twice as long, but gluing and paint take  $\frac{1}{2}$  as long?

Latency:

Throughput:

CPI =

# Pipelined Performance



What if drilling takes twice as long, but gluing and paint take  $\frac{1}{2}$  as long?

Latency: 4 cycles/task

Throughput: 1 task/2 cycles      CPI = 2

# Lessons

## Principle:

Throughput increased by parallel execution

Balanced pipeline very important

Else slowest stage dominates performance

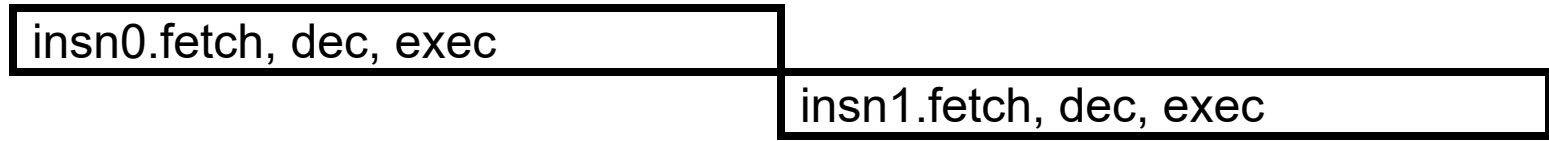
## Pipelining:

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

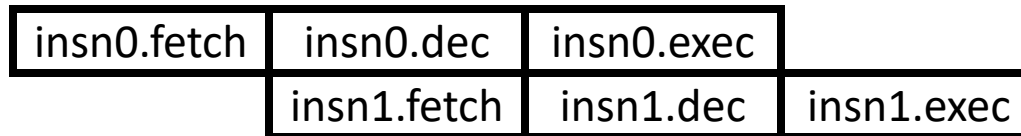
# Single Cycle vs Pipelined Processor

# Single Cycle → Pipelining

## Single-cycle



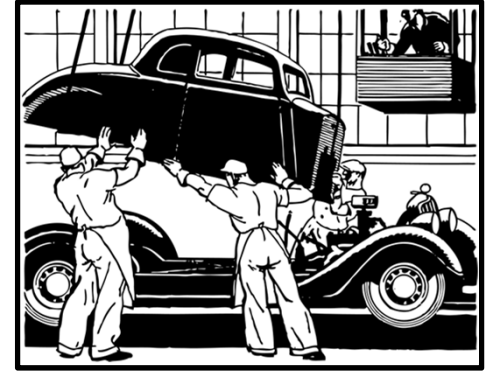
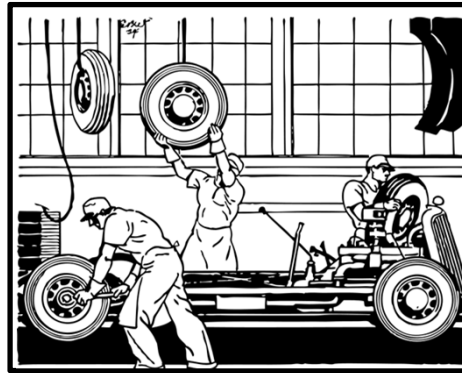
## Pipelined



# Agenda

## 5-stage Pipeline

- Implementation
- Working Example

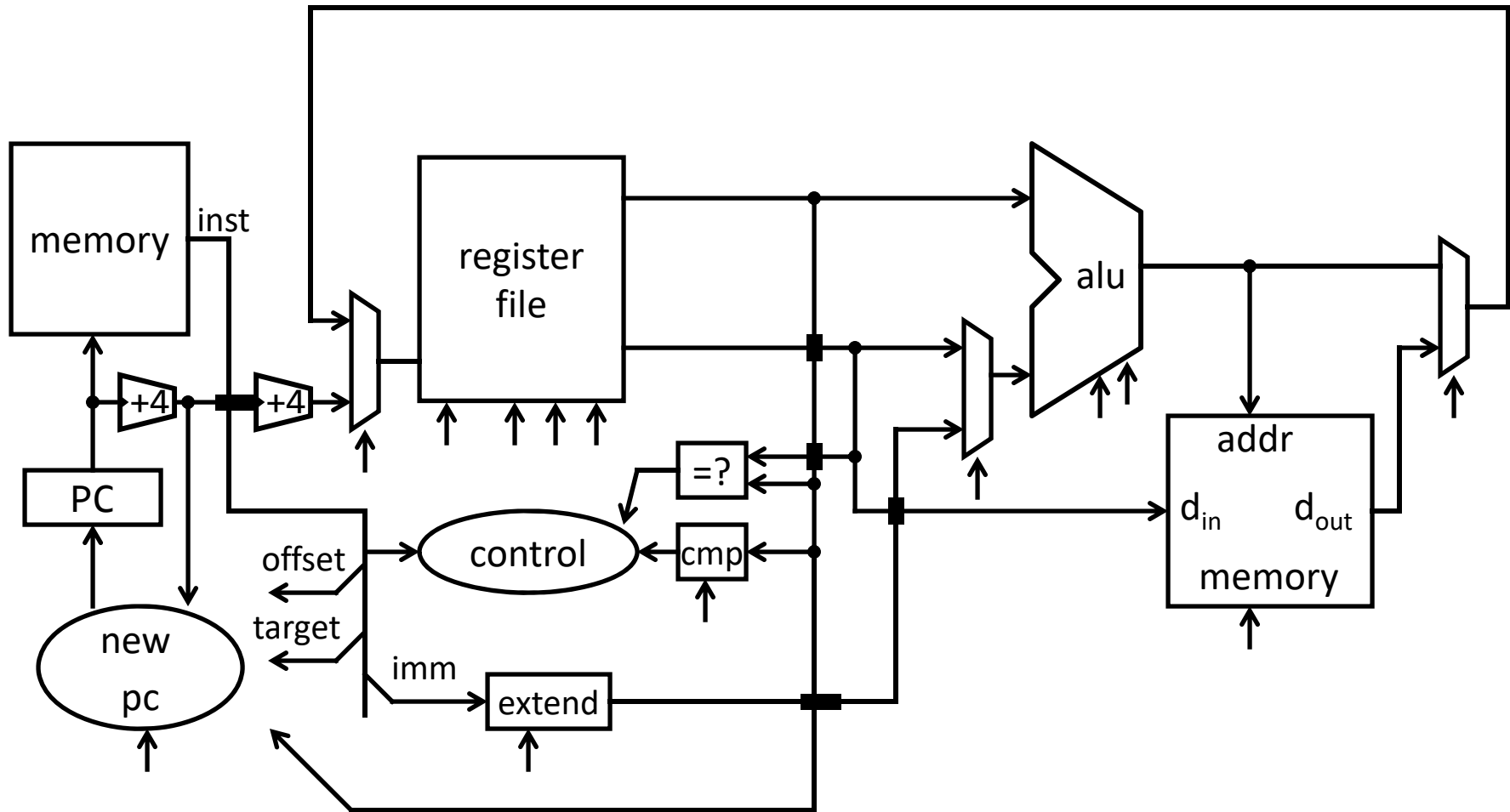


## Hazards

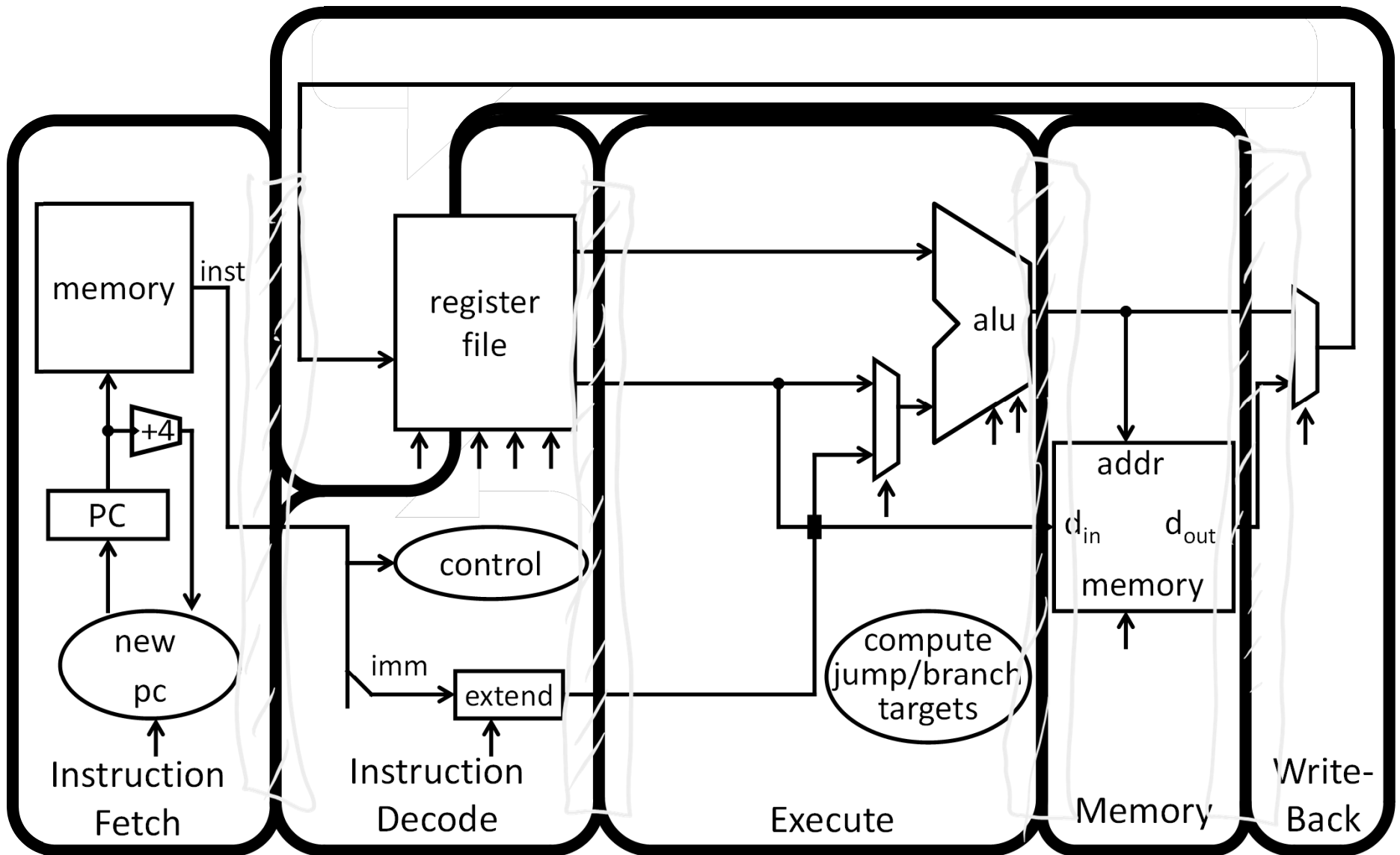
- Structural
- Data Hazards
- Control Hazards

# A Processor

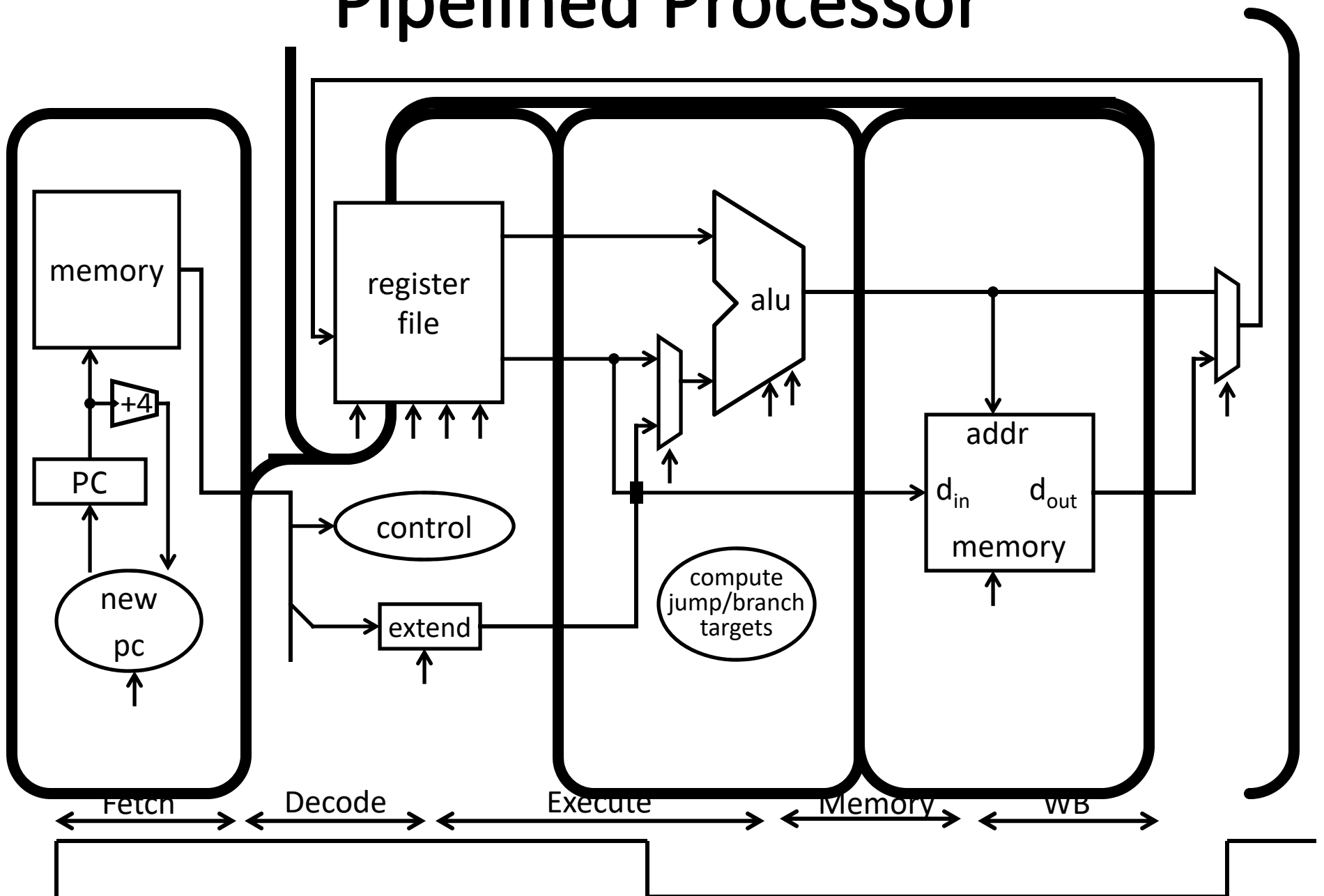
Review: Single cycle processor



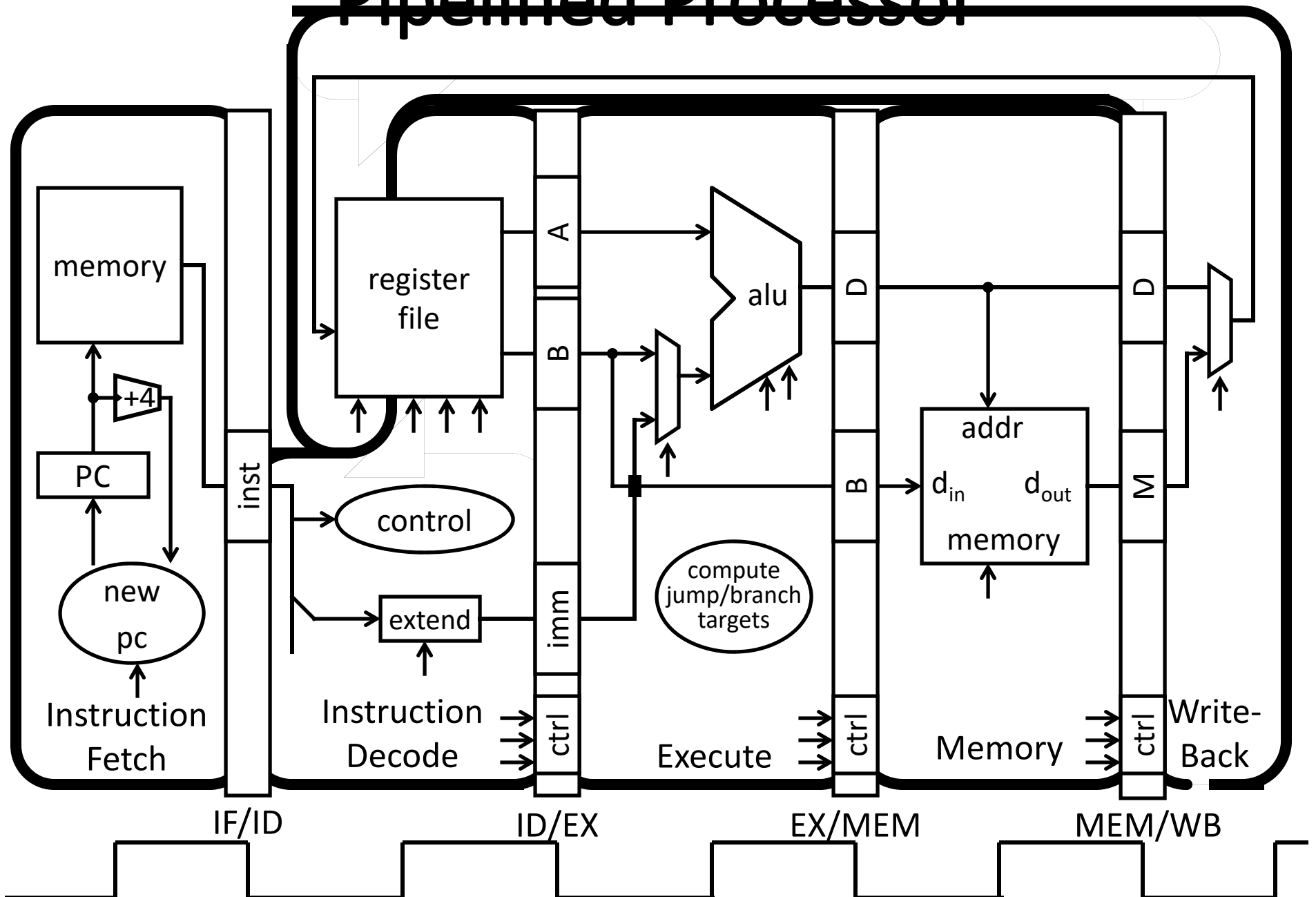
# A Processor



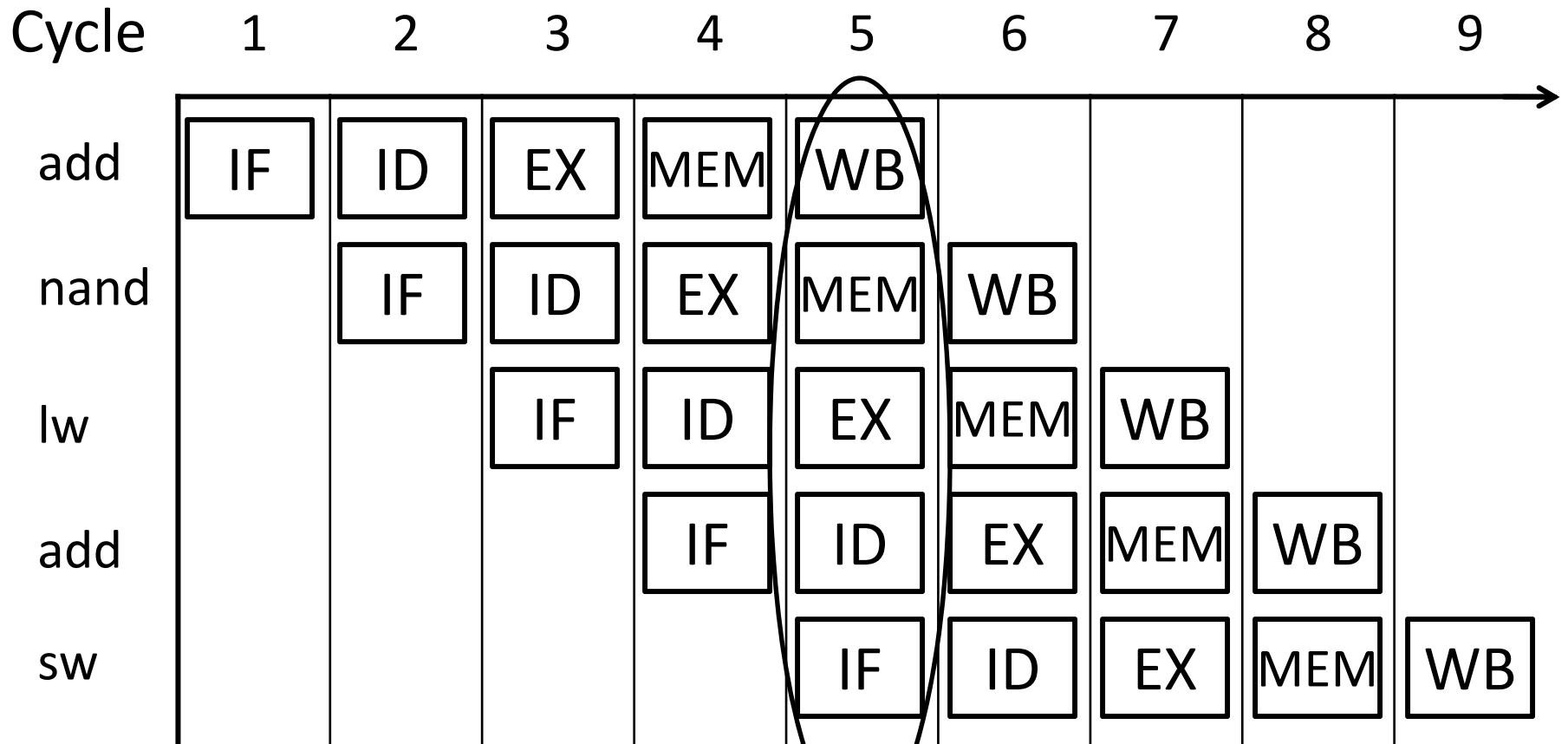
# Pipelined Processor



# Pipelined Processor



# Time Graphs



Latency:

5 cycles

Throughput:

1 insn/cycle

CPI = 1

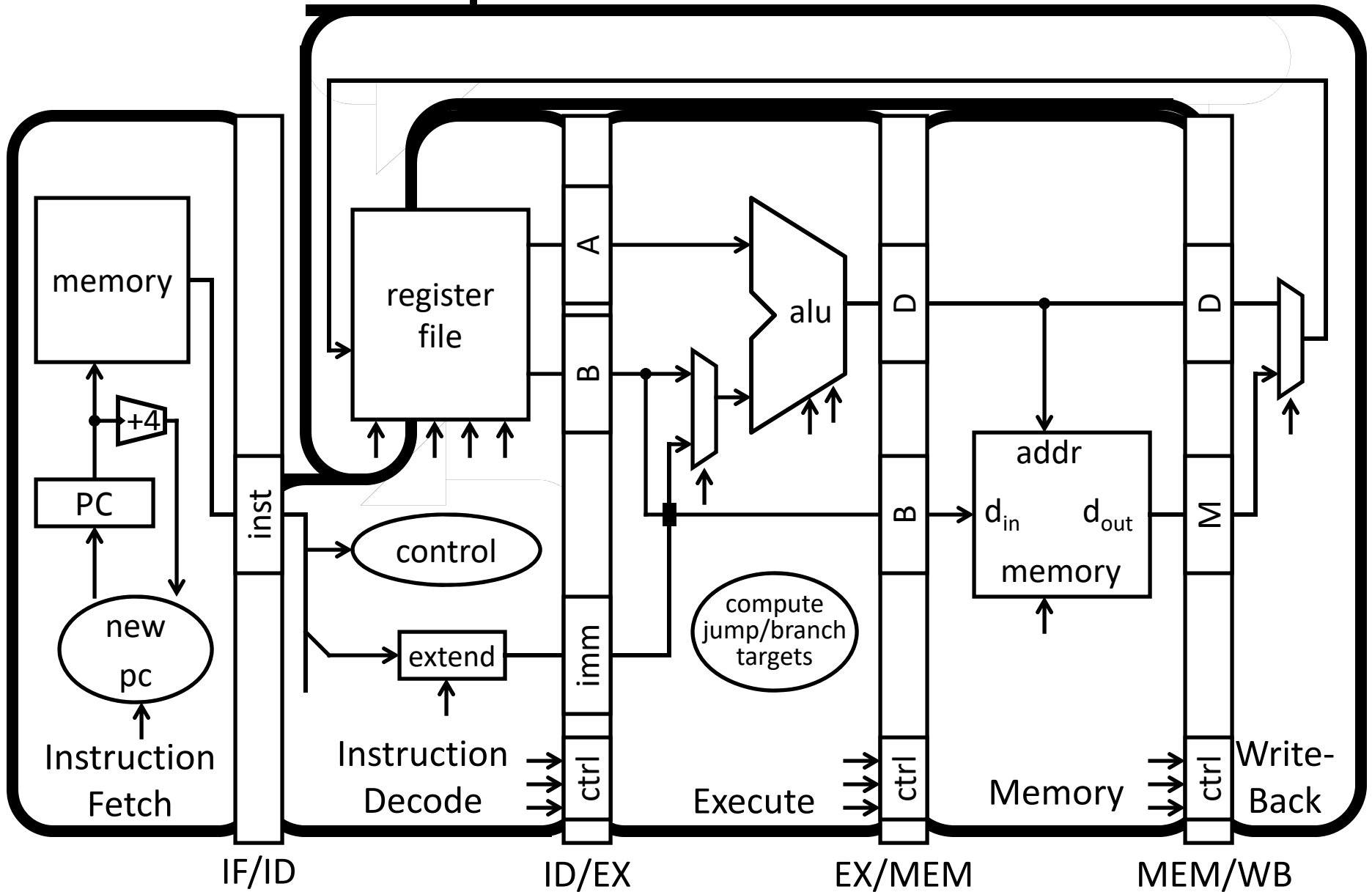
Concurrency:

5

# Principles of Pipelined Implementation

- Break datapath into multiple cycles (here 5)
  - Parallel execution increases throughput
  - Balanced pipeline very important
    - Slowest stage determines clock rate
    - Imbalance kills performance
- Add pipeline registers (flip-flops) for isolation
  - Each stage begins by reading values *from* latch
  - Each stage ends by writing values *to* latch
- Resolve hazards

# Pipelined Processor



# Pipeline Stages

Stage	Perform Functionality	Latch values of interest
<b>Fetch</b>	Use PC to index Program Memory, increment PC	Instruction bits (to be decoded) PC + 4 (to compute branch targets)
<b>Decode</b>	Decode instruction, generate control signals, read register file	Control information, Rd index, immediates, offsets, register values (Ra, Rb), PC+4 (to compute branch targets)
<b>Execute</b>	Perform ALU operation Compute targets (PC+4+offset, etc.) in case this is a branch, decide if branch taken	Control information, Rd index, <i>etc.</i> Result of ALU operation, value in case this is a store instruction
<b>Memory</b>	Perform load/store if needed, address is ALU result	Control information, Rd index, <i>etc.</i> Result of load, pass result from execute
<b>Writeback</b>	Select value, write to register file	

# Instruction Fetch (IF)

## Stage 1: Instruction Fetch

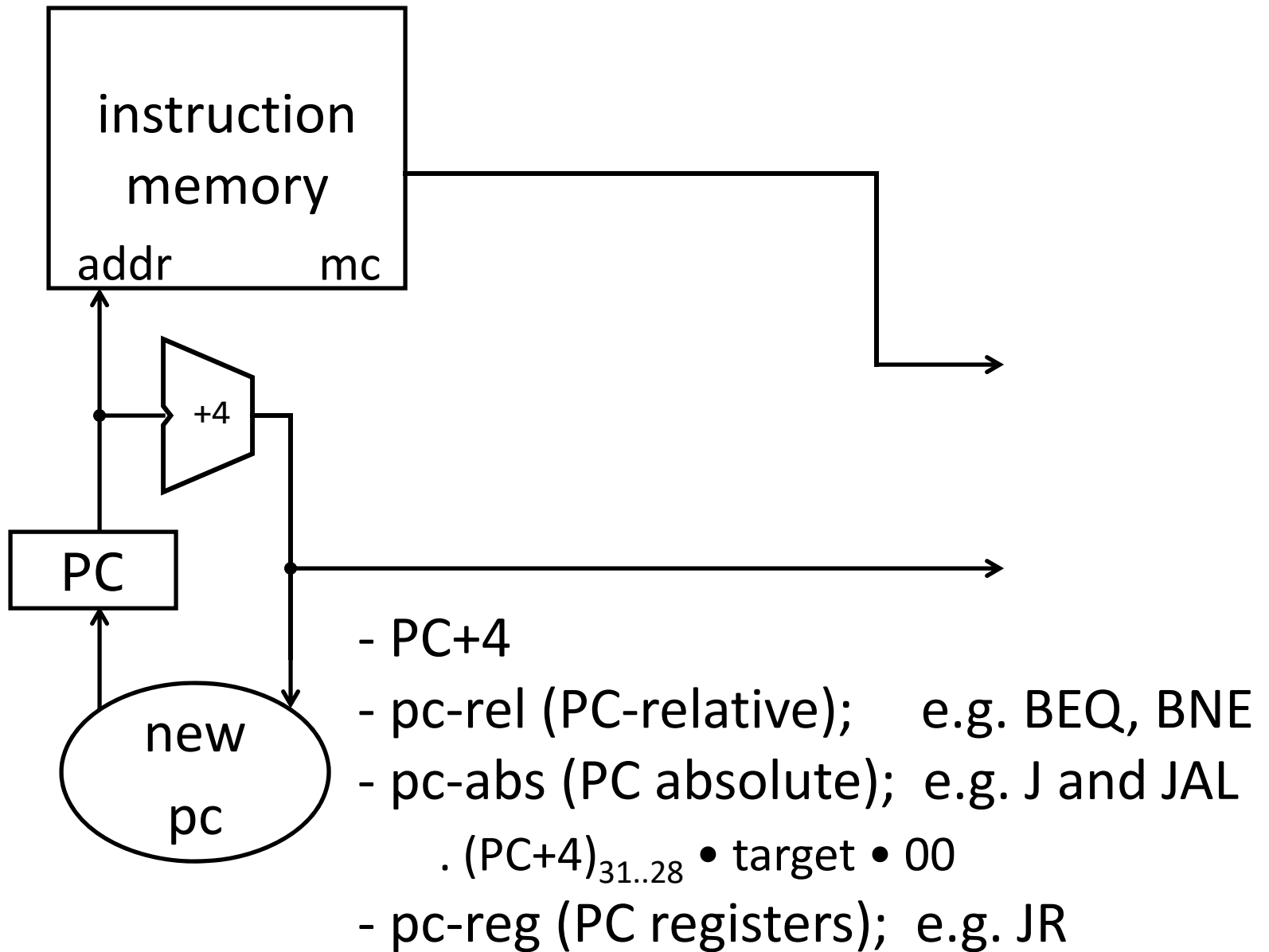
Fetch a new instruction every cycle

- Current PC is index to instruction memory
- Increment the PC at end of cycle (assume no branches for now)

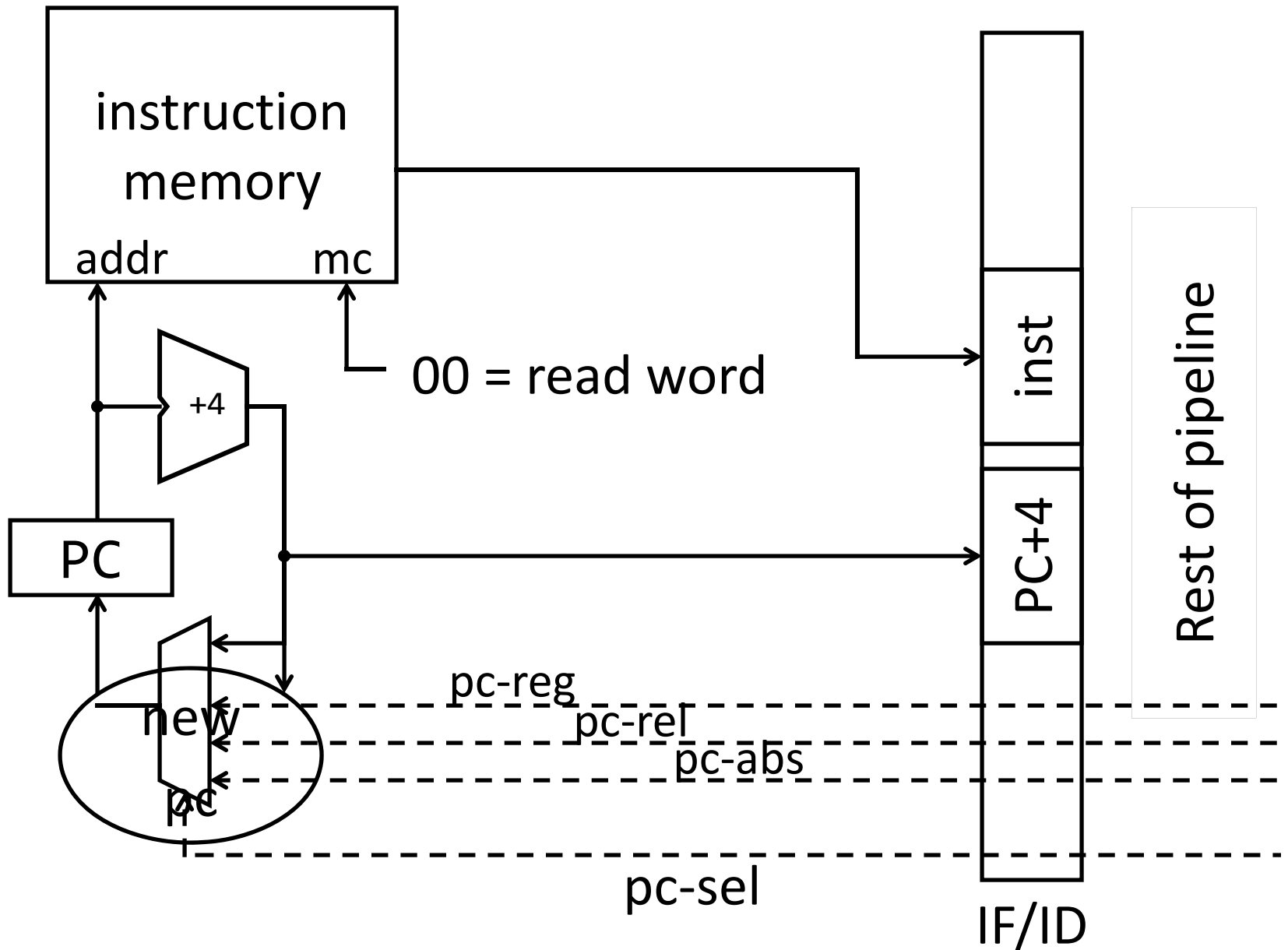
Write values of interest to pipeline register (IF/ID)

- Instruction bits (for later decoding)
- PC+4 (for later computing branch targets)

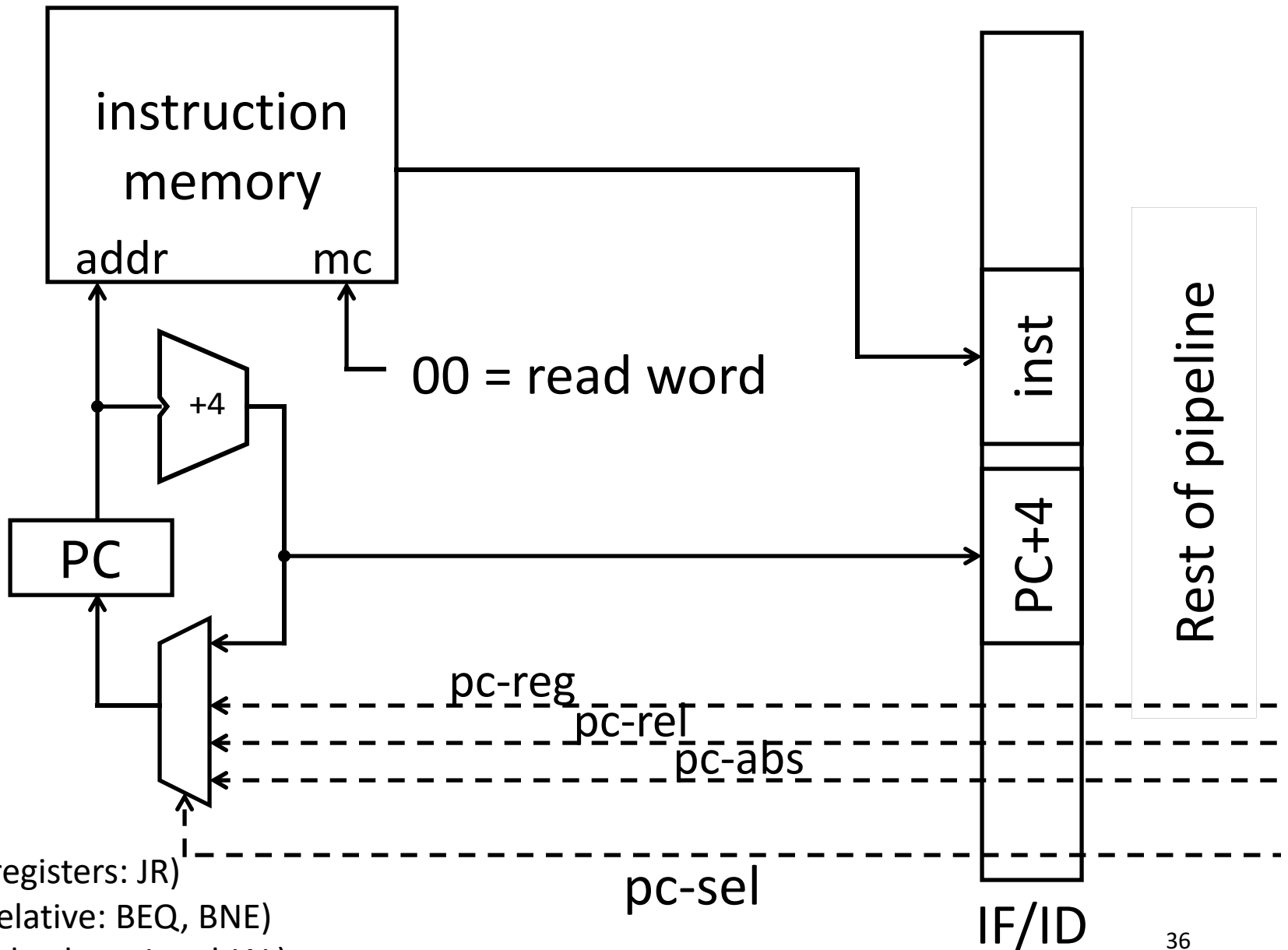
# Instruction Fetch (IF)



# Instruction Fetch (IF)



# Instruction Fetch (IF)



# Decode

## Stage 2: Instruction Decode

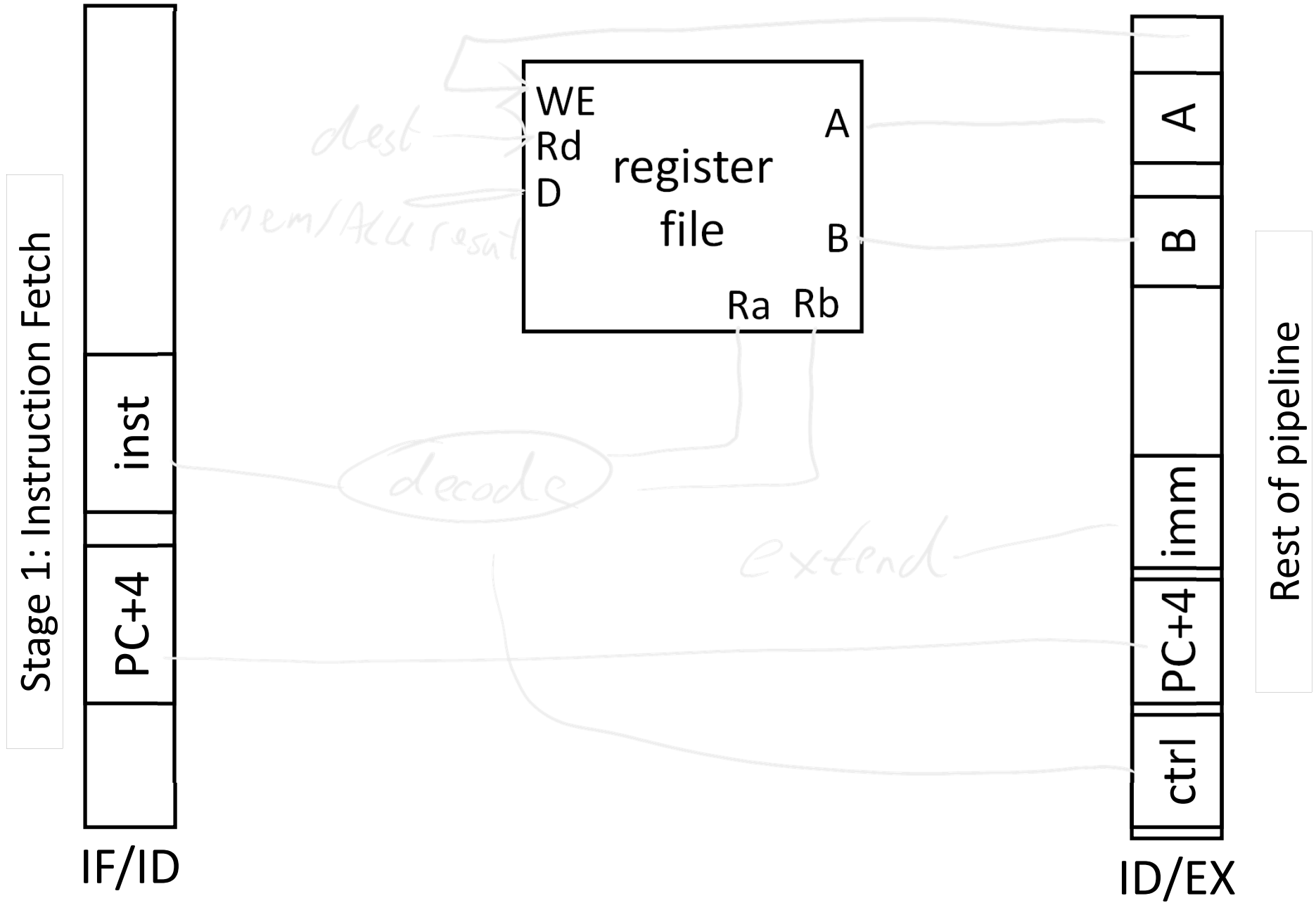
On every cycle:

- Read IF/ID pipeline register to get instruction bits
- Decode instruction, generate control signals
- Read from register file

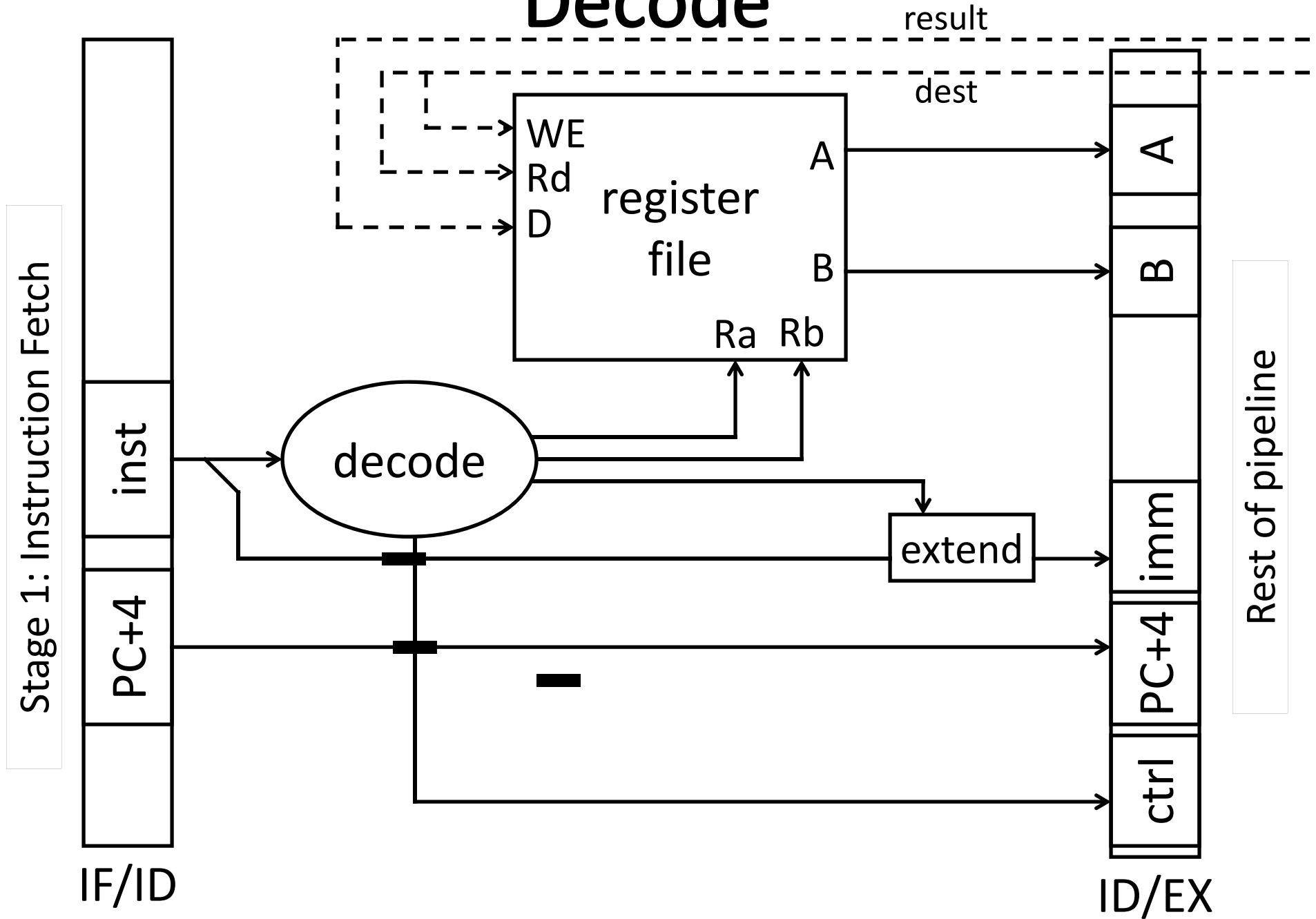
Write values of interest to pipeline register (ID/EX)

- Control information, Rd index, immediates, offsets, ...
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)

# Decode



# Decode



# Execute (EX)

## Stage 3: Execute

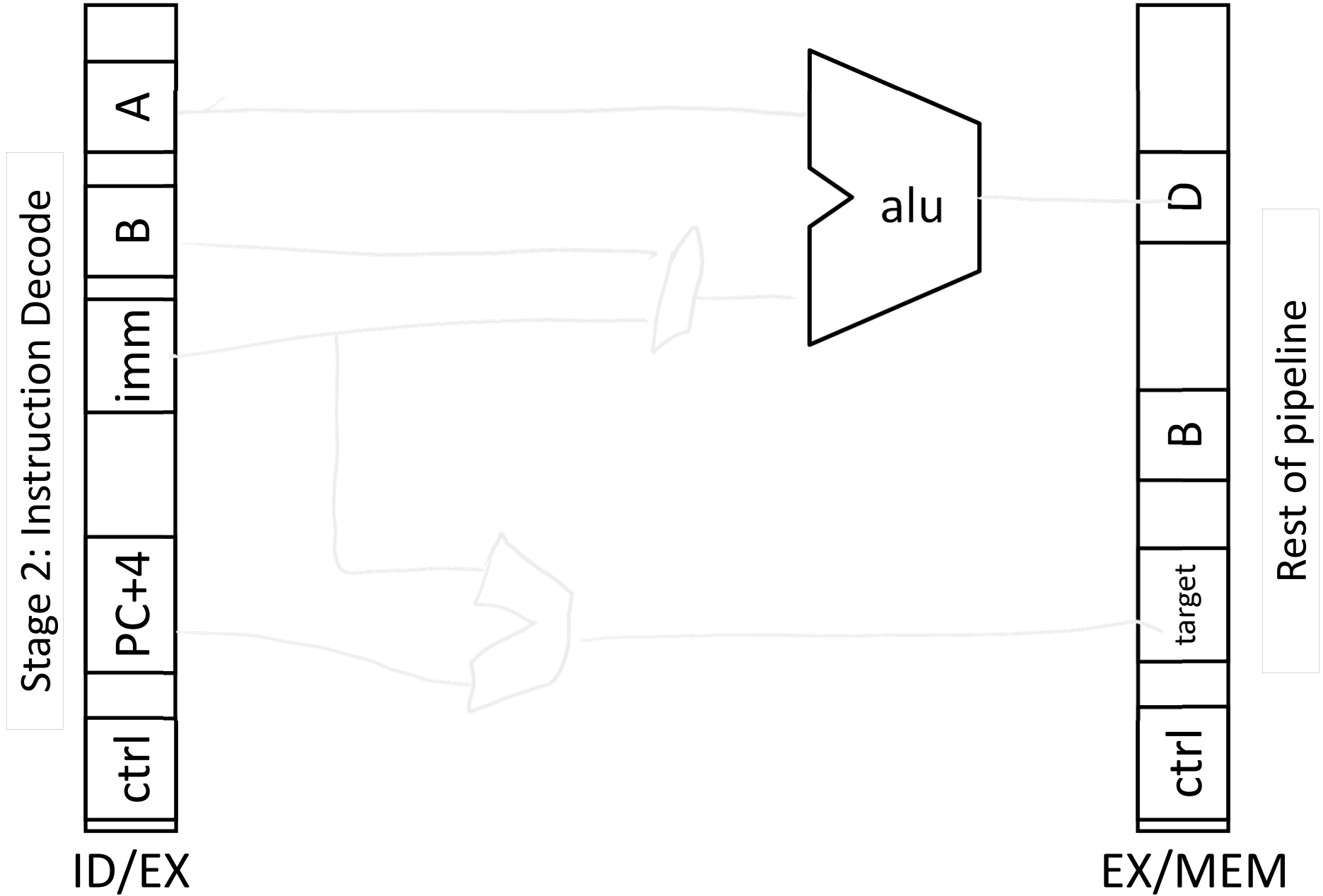
On every cycle:

- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) *in case* this is a branch
- Decide if jump/branch should be taken

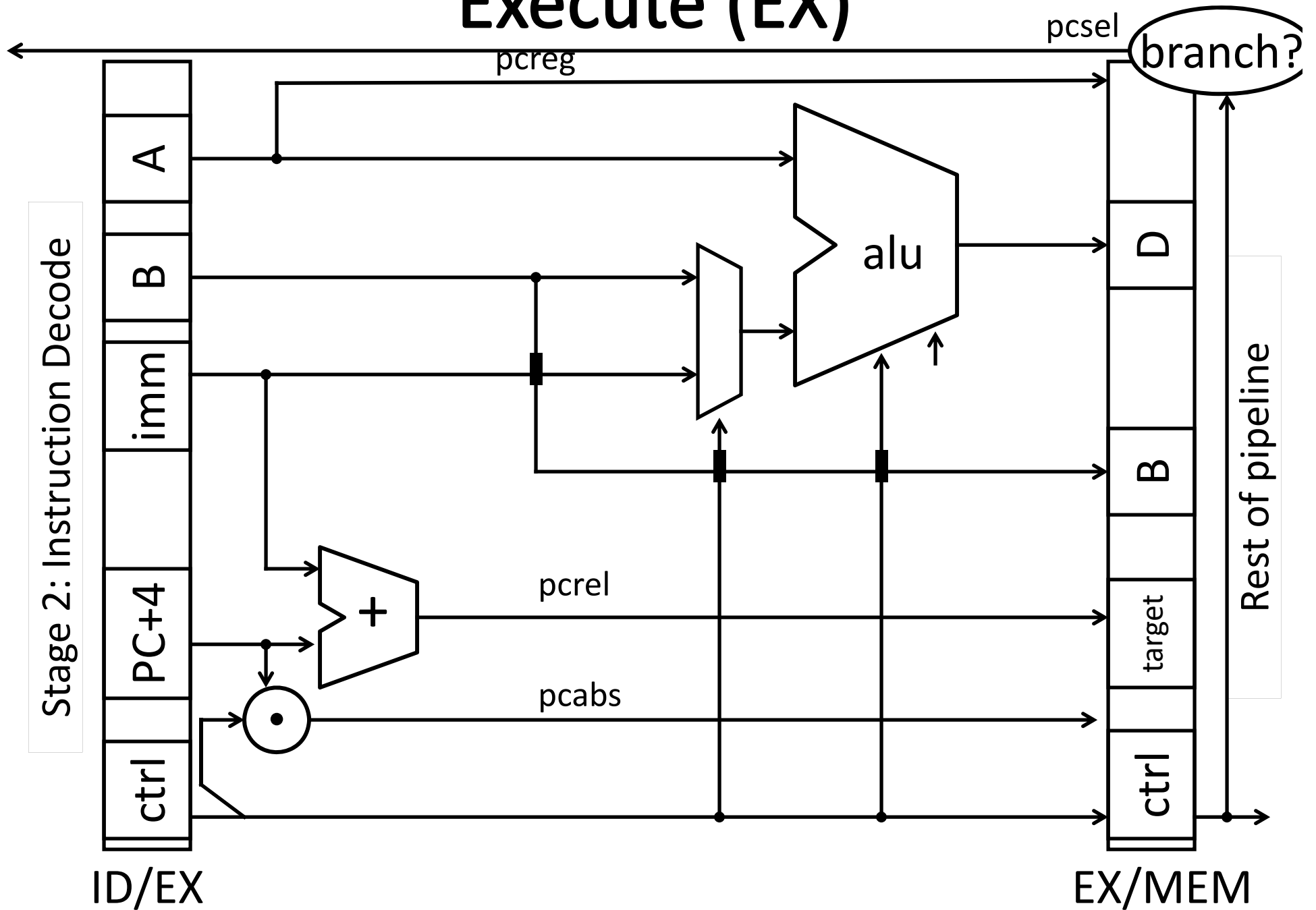
Write values of interest to pipeline register (EX/MEM)

- Control information, Rd index, ...
- Result of ALU operation
- Value *in case* this is a memory store instruction

# Execute (EX)



# Execute (EX)



# MEM

## Stage 4: Memory

On every cycle:

- Read EX/MEM pipeline register to get values and control bits
- Perform memory load/store if needed
  - address is ALU result

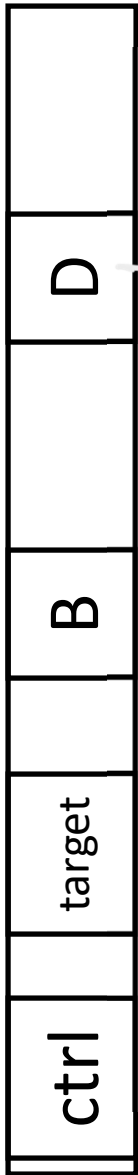
Write values of interest to pipeline register (MEM/WB)

- Control information, Rd index, ...
- Result of memory operation
- Pass result of ALU operation

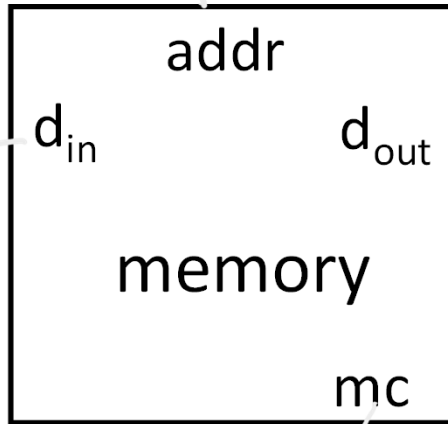
# MEM

ALU Result  
- Ra + Rb  
~ SW/LW

Stage 3: Execute



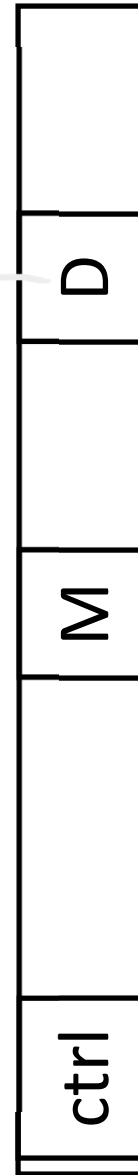
EX/MEM



SW

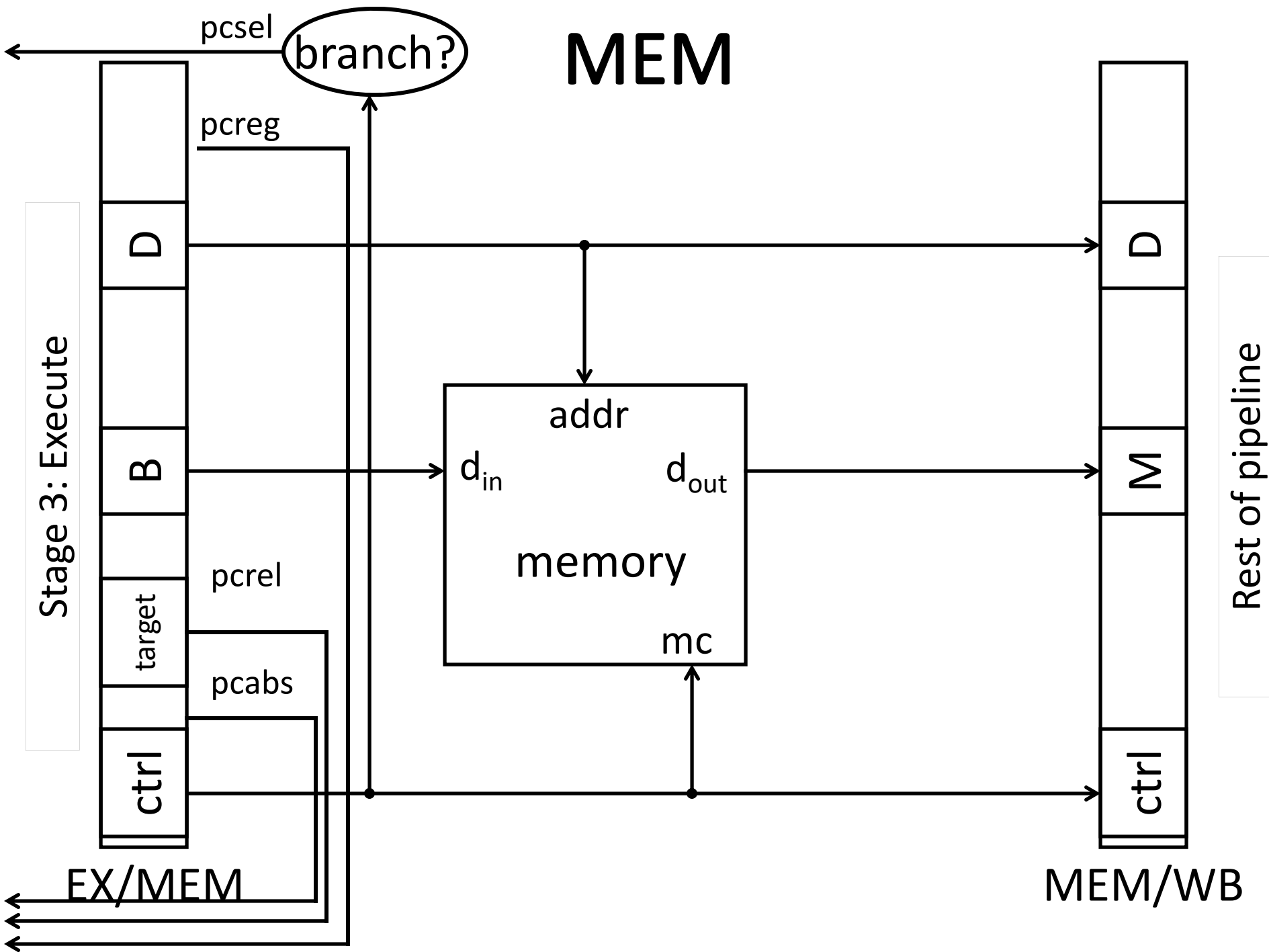
LW

SW = wr  
LW = rd



Rest of pipeline

MEM/WB



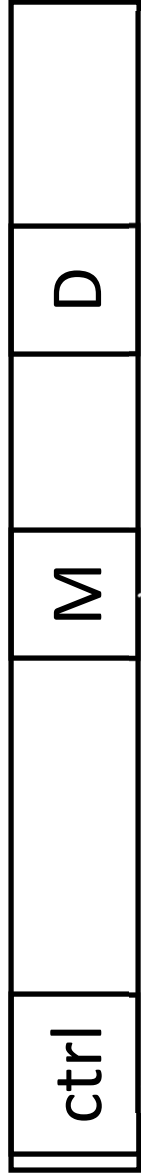
# WB

## Stage 5: Write-back

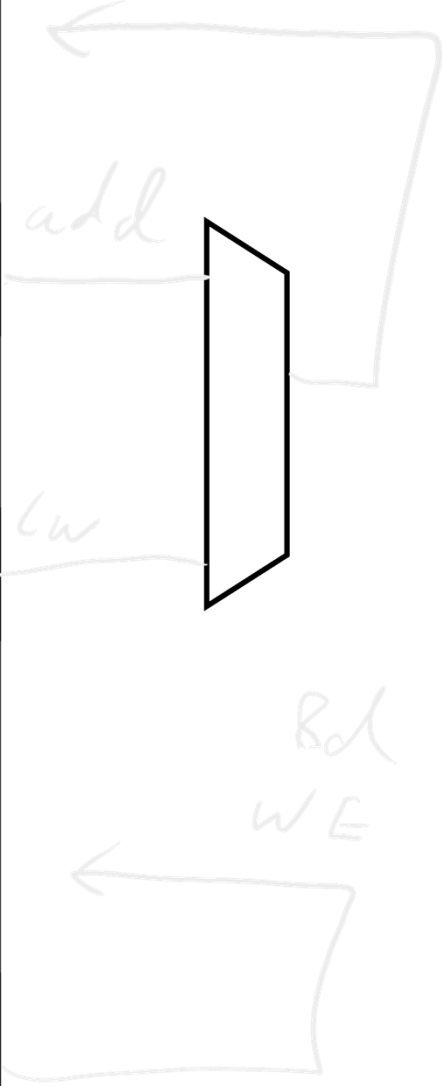
On every cycle:

- Read MEM/WB pipeline register to get values and control bits
- Select value and write to register file

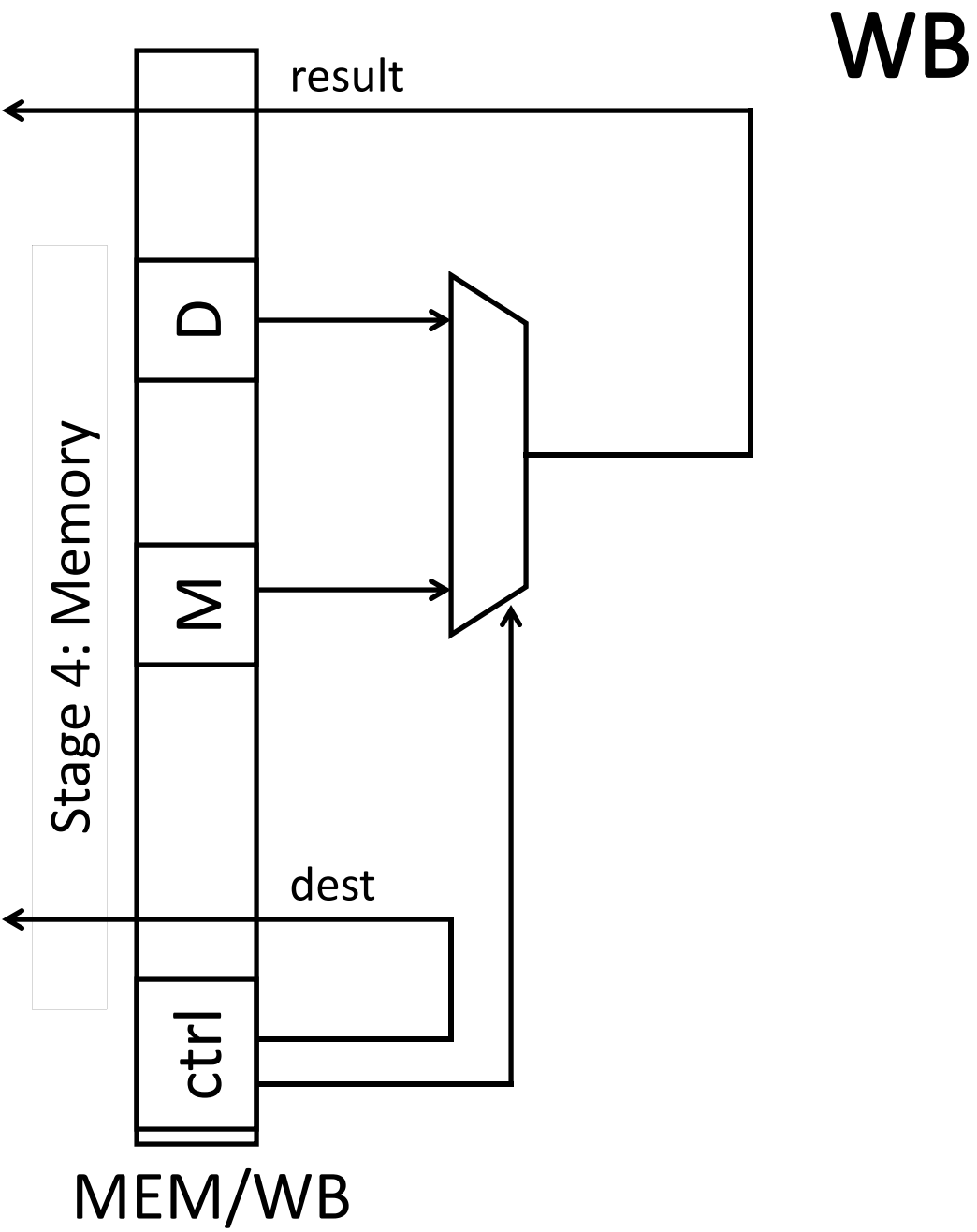
Stage 4: Memory



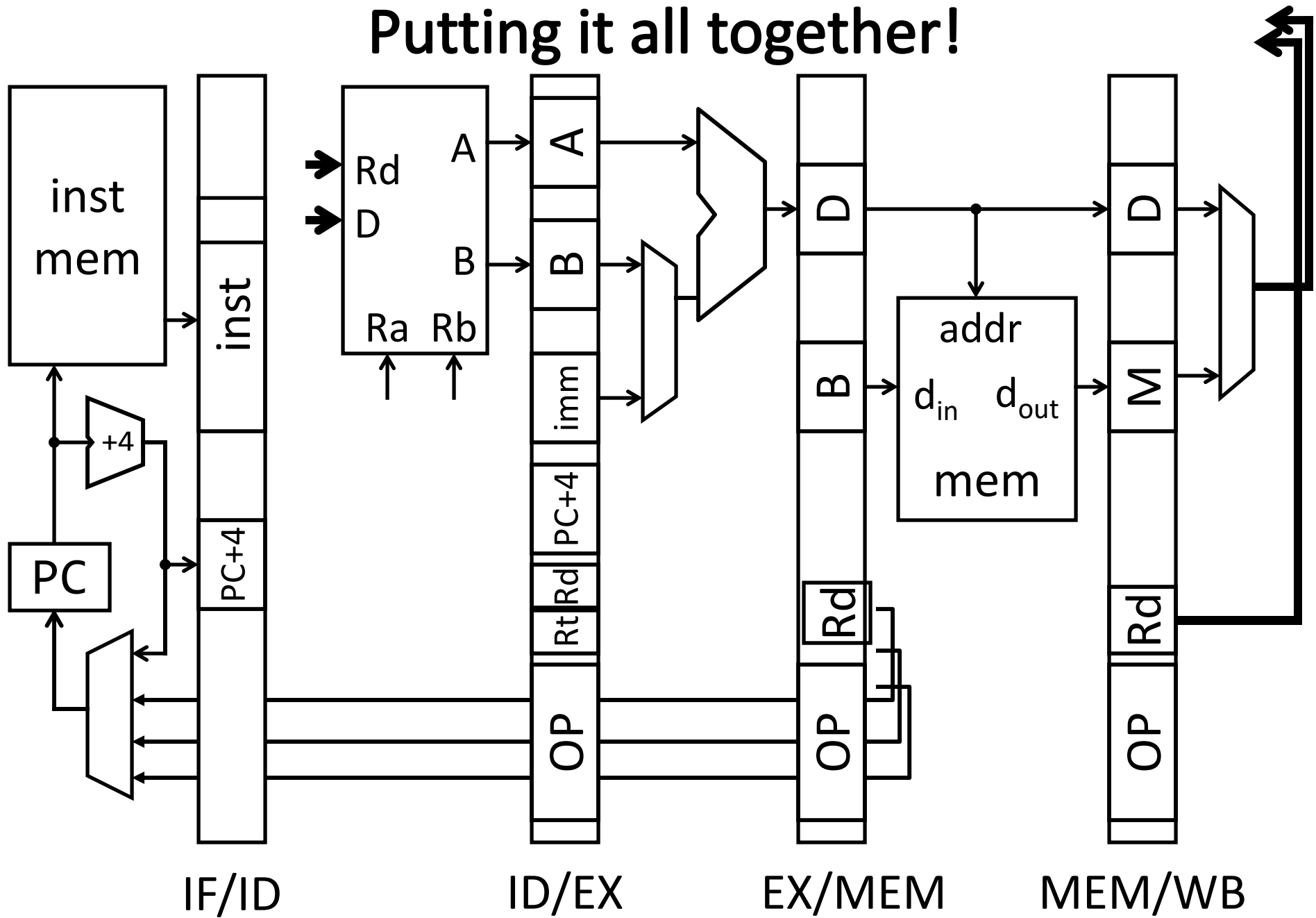
MEM/WB



WB



# Putting it all together!



# iClicker Question

Consider a non-pipelined processor with clock period  $C$  (e.g., 50 ns). If you divide the processor into  $N$  stages (e.g., 5), your new clock period will be:

- A.  $C$
- B.  $N$
- C. less than  $C/N$
- D.  $C/N$
- E. greater than  $C/N$

# iClicker Question

Consider a non-pipelined processor with clock period  $C$  (e.g., 50 ns). If you divide the processor into  $N$  stages (e.g., 5), your new clock period will be:

- A.  $C$
- B.  $N$
- C. less than  $C/N$
- D.  $C/N$
- E. greater than  $C/N$

# 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)

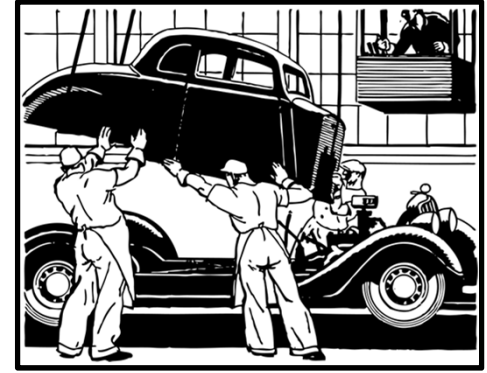
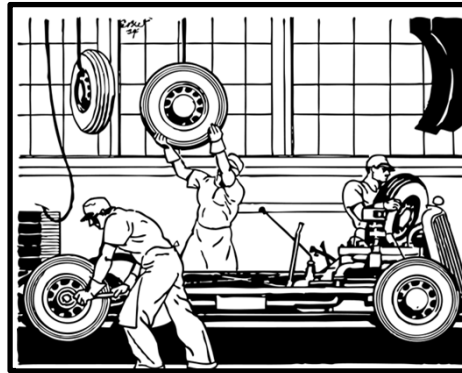
# MIPS is *designed* for pipelining

- Instructions same length
  - 32 bits, easy to fetch and then decode
- 3 types of instruction formats
  - Easy to route bits between stages
  - Can read a register source before even knowing what the instruction is
- Memory access through lw and sw only
  - Access memory after ALU

# Agenda

## 5-stage Pipeline

- Implementation
- Working Example



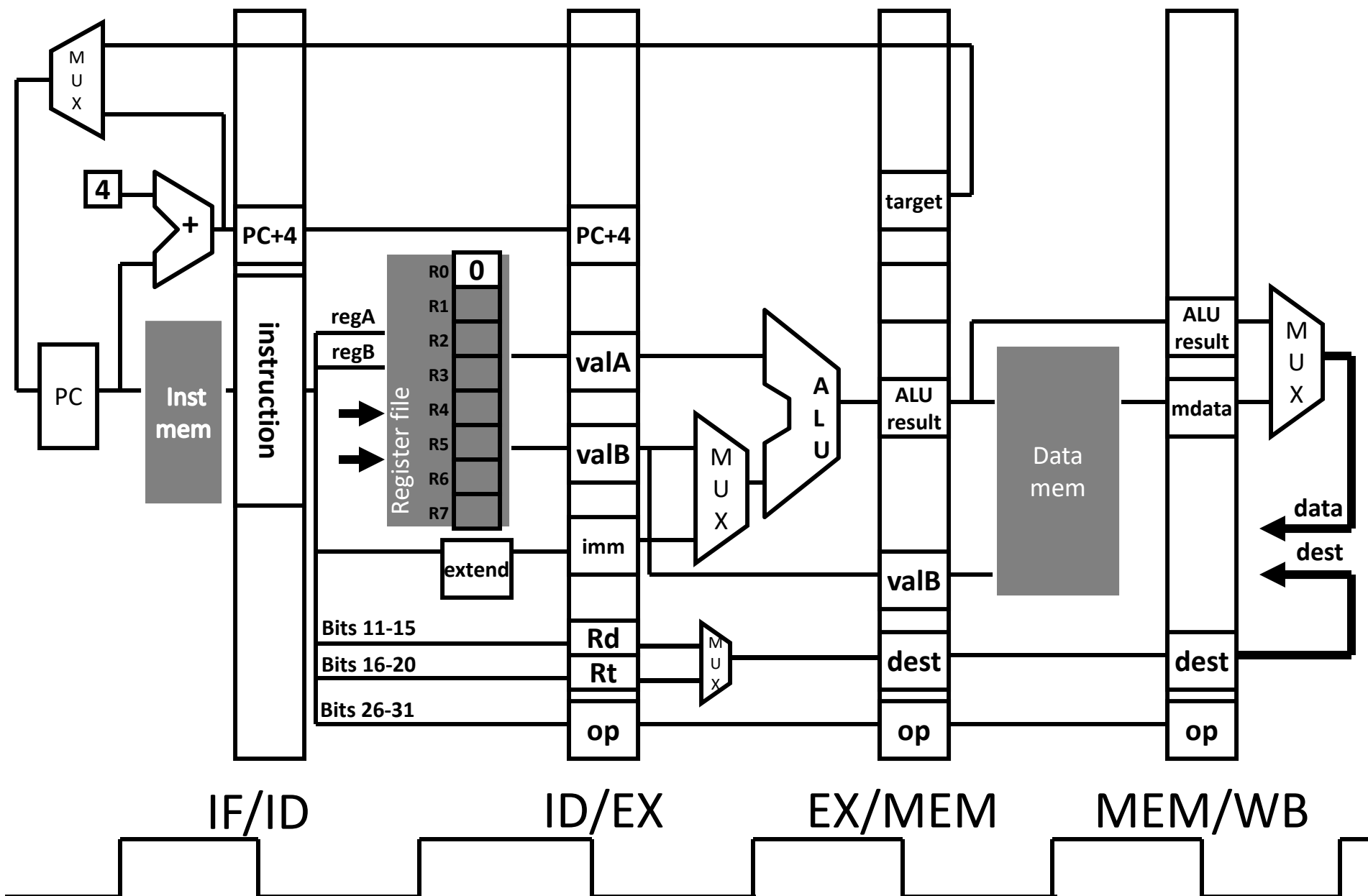
## Hazards

- Structural
- Data Hazards
- Control Hazards

## Example: : Sample Code (Simple)

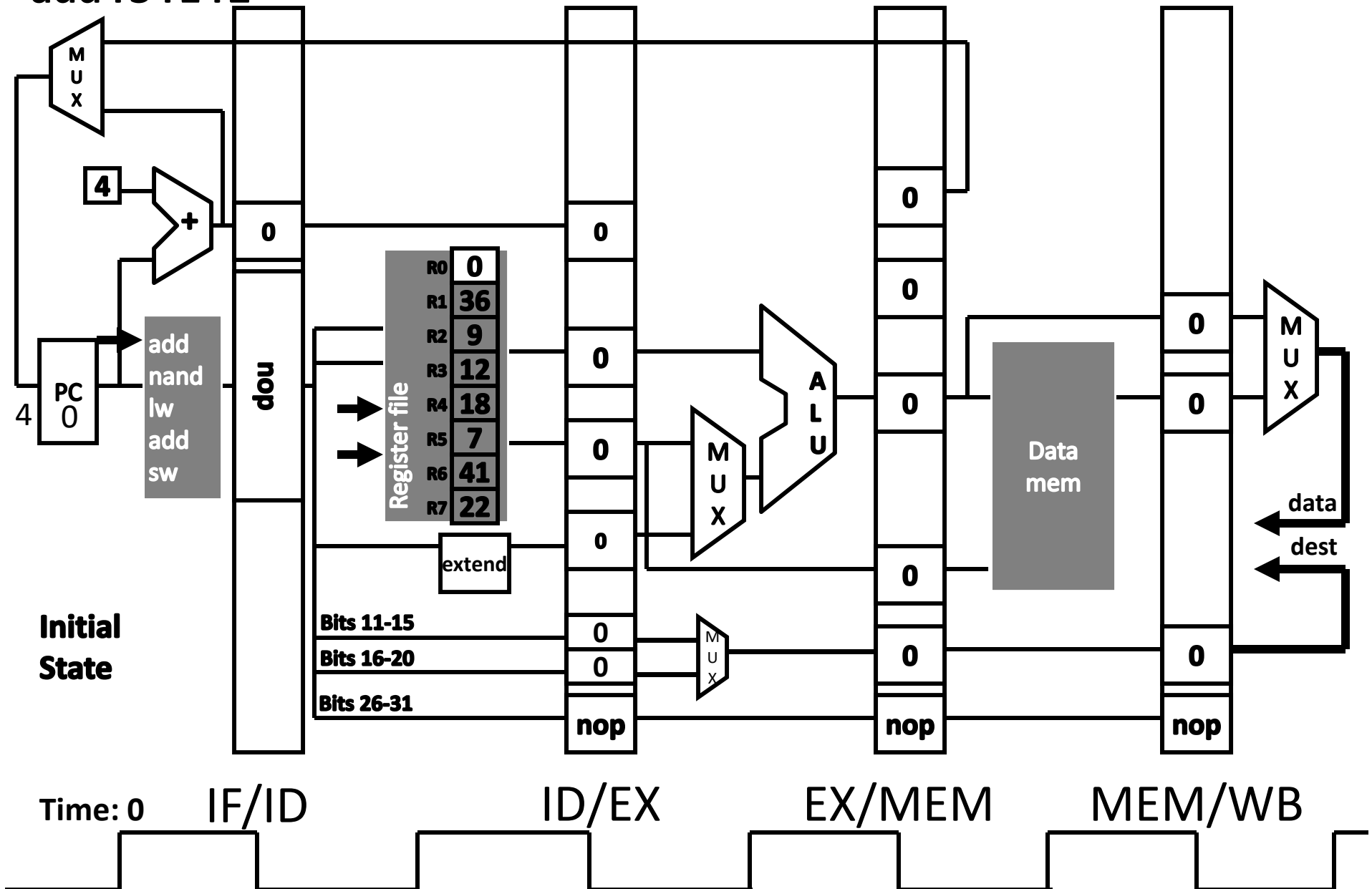
```
add    r3 ← r1, r2
nand   r6 ← r4, r5
lw     r4 ← 20(r2)
add    r5 ← r2, r5
sw     r7 → 12(r3)
```

Assume 8-register machine

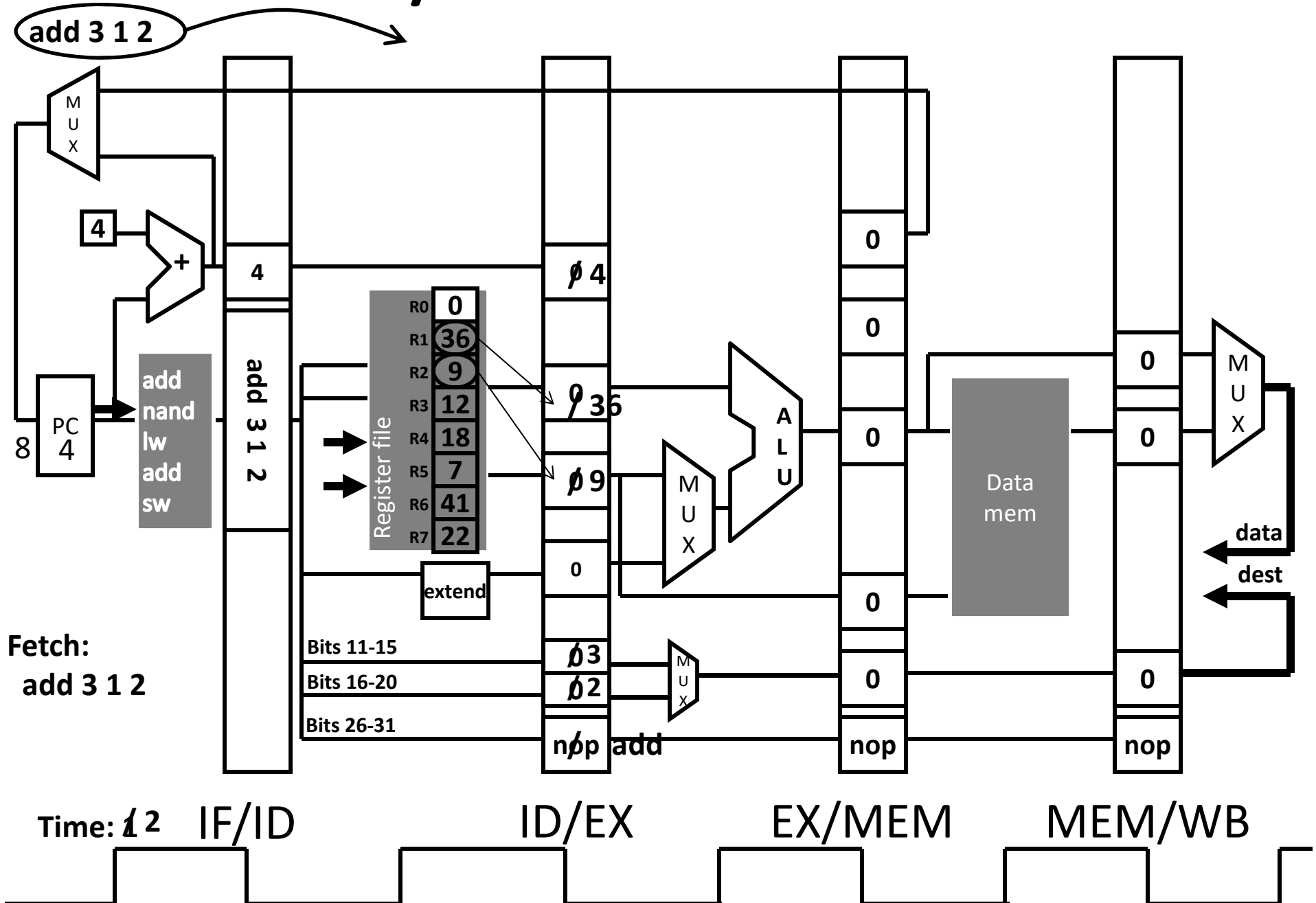


At time 1,  
Fetch  
add r3 r1 r2

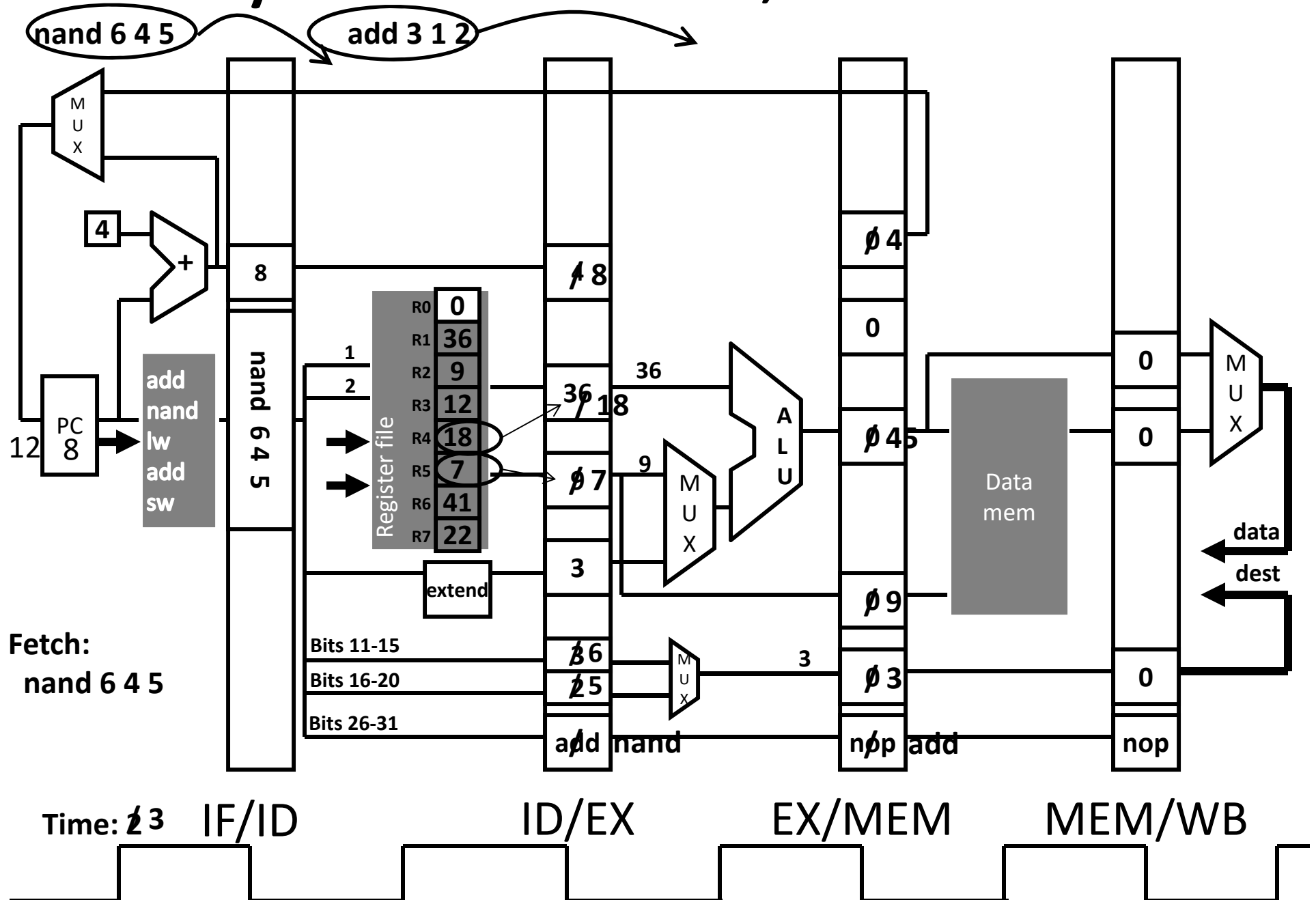
# Example: Start State @ Cycle 0



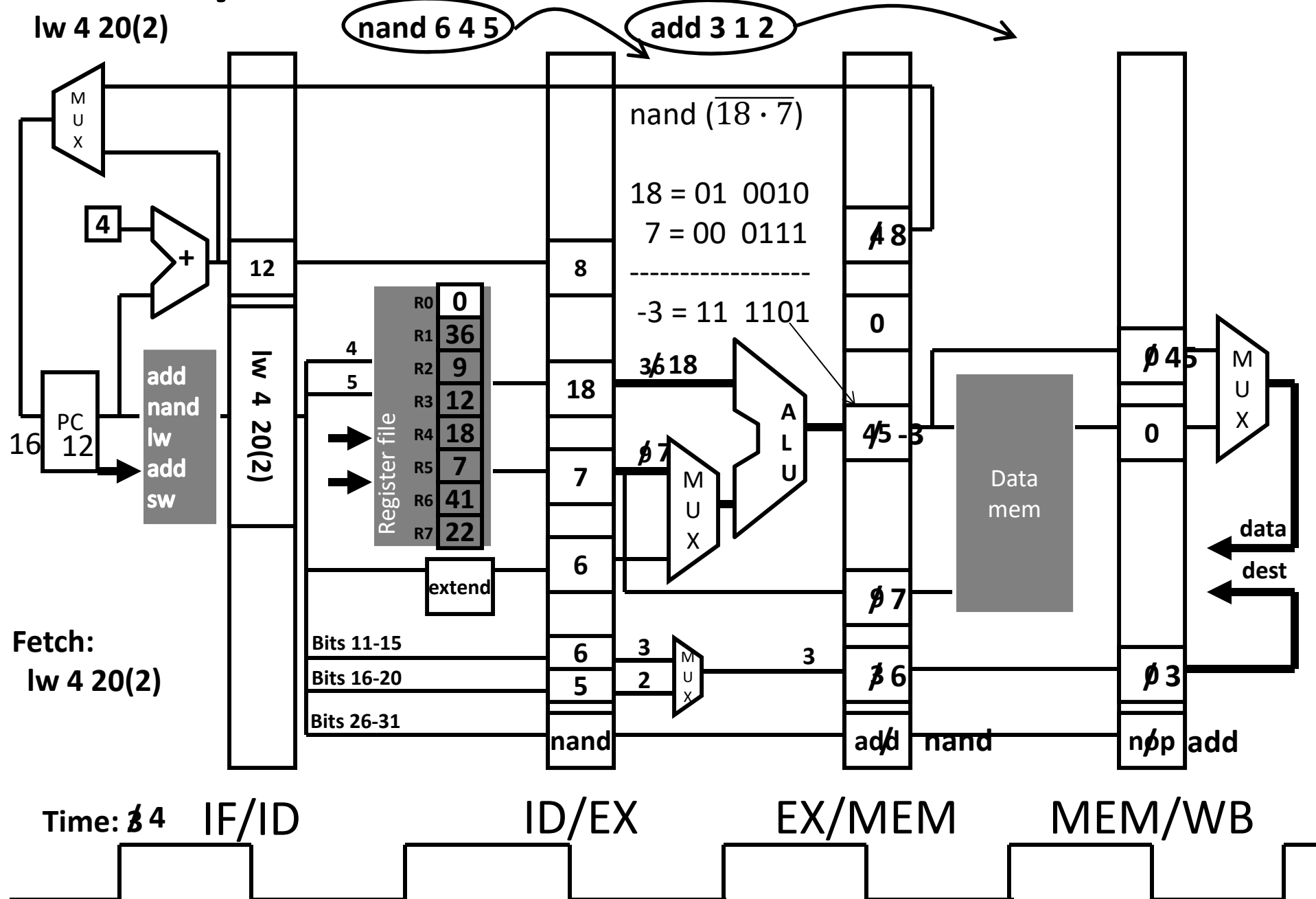
# Cycle 1: Fetch add



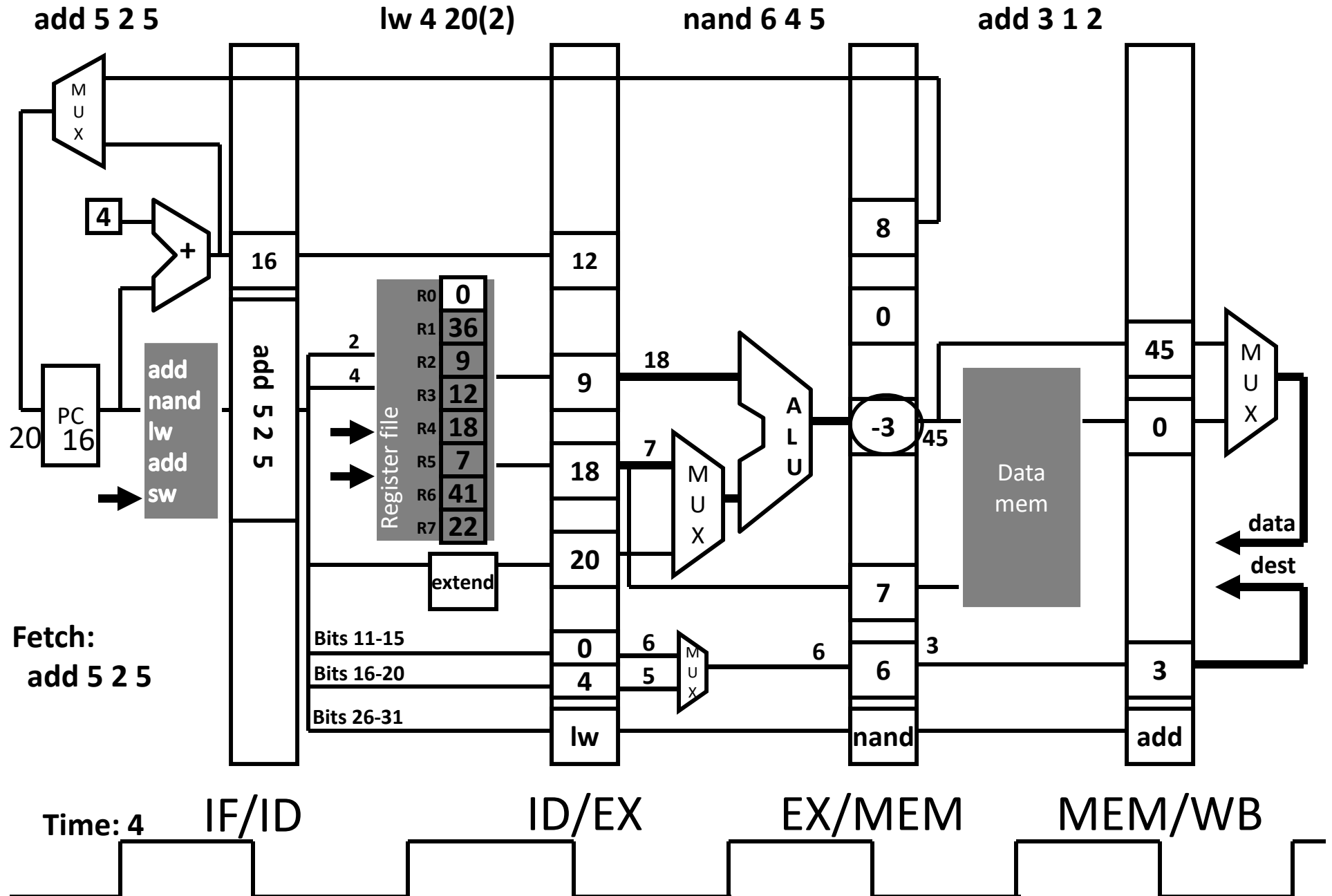
# Cycle 2: Fetch nand, Decode add



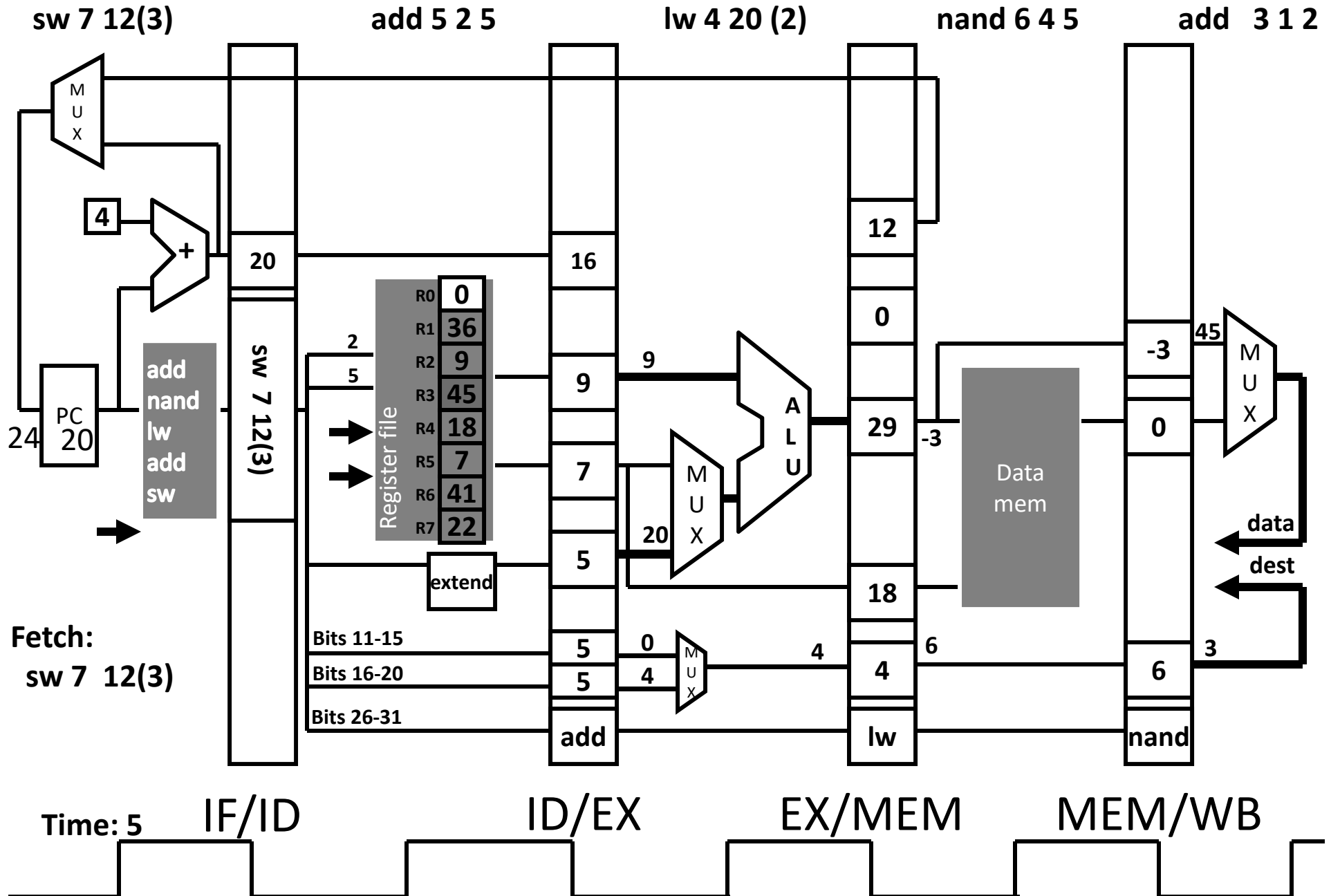
# Cycle 3: Fetch lw, Decode nand, ...



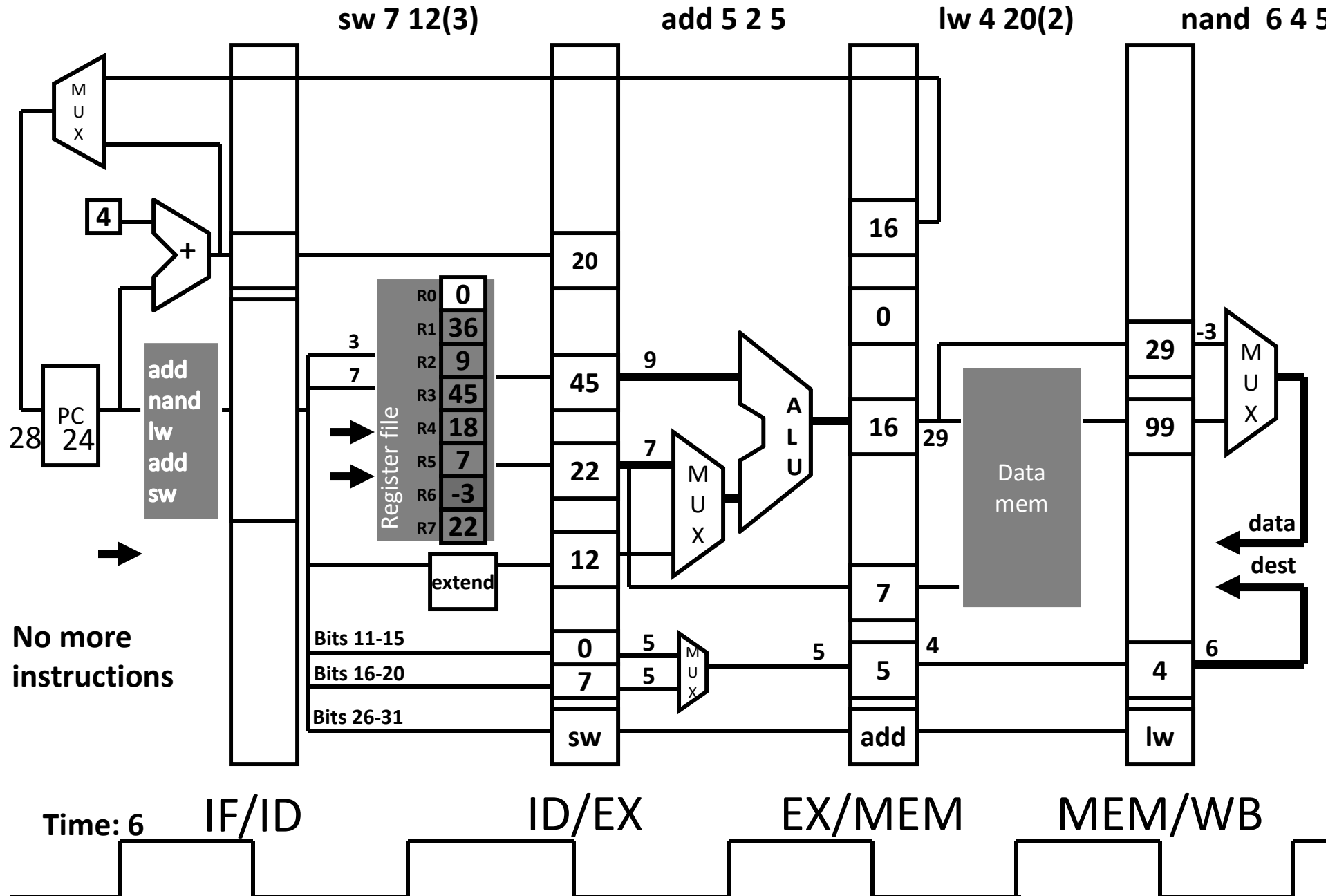
# Cycle 4: Fetch add, Decode lw, ...



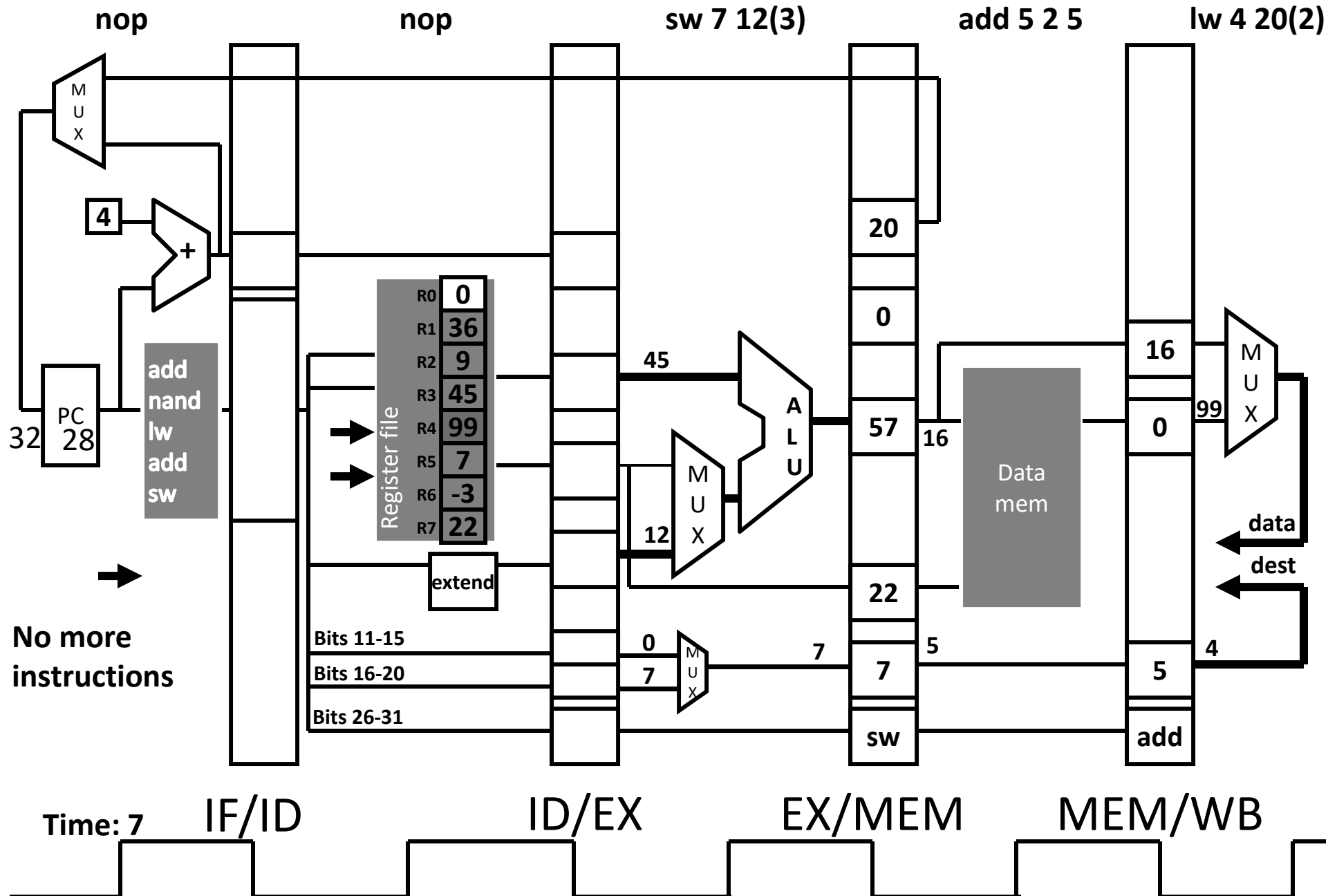
# Cycle 4: Fetch add, Decode lw, ...



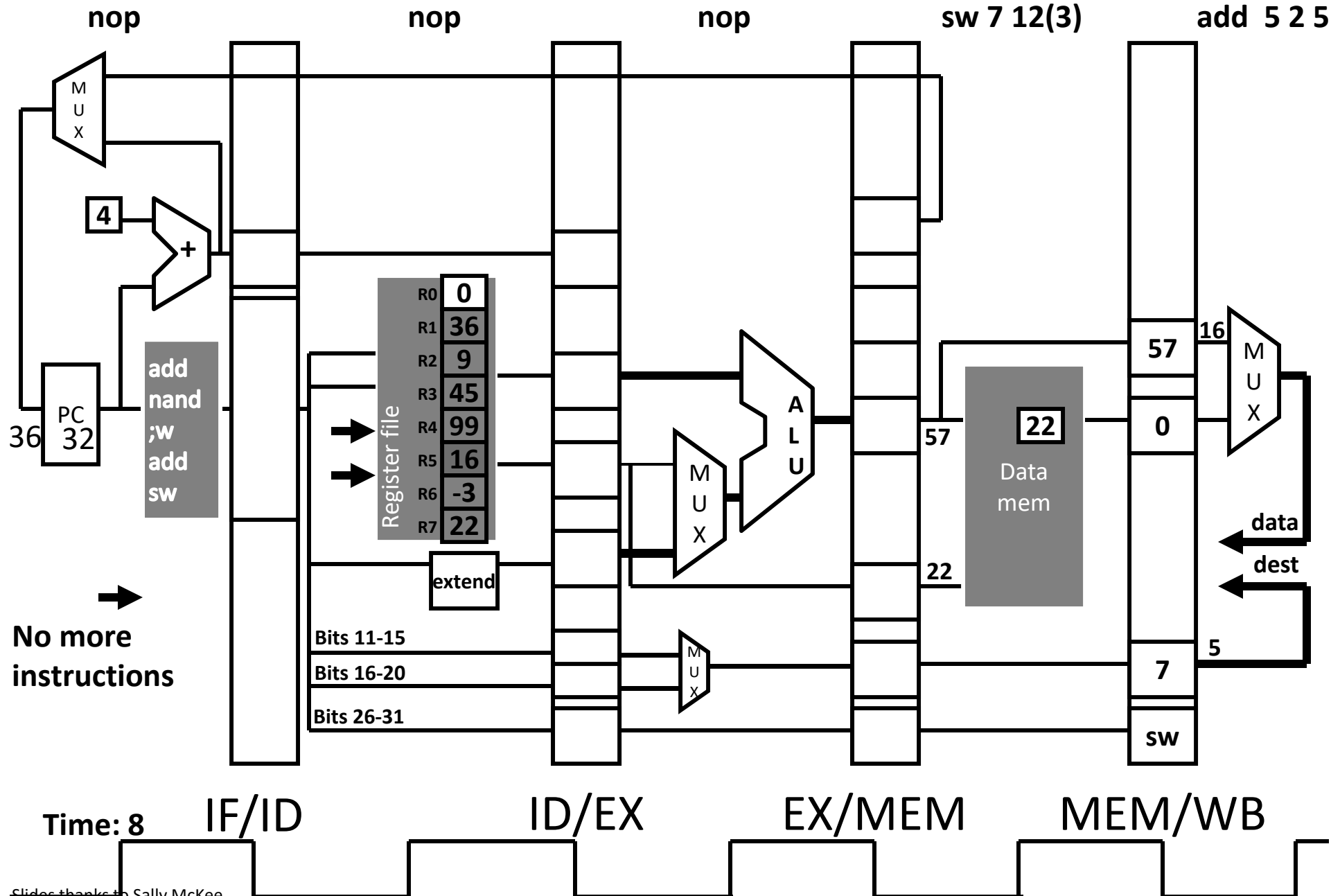
# Cycle 6: Decode sw, ...



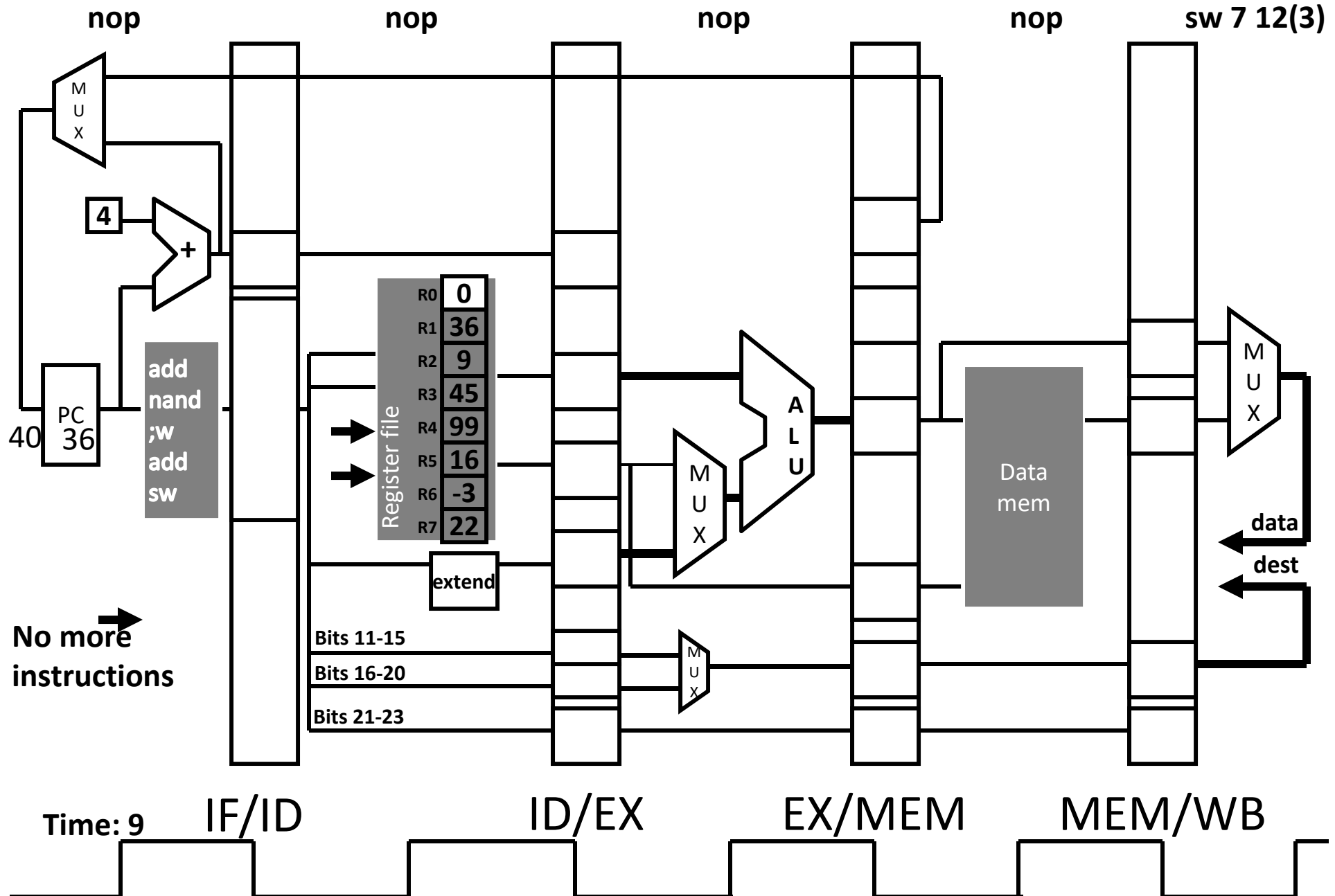
# Cycle 7: Execute sw, ...



# Cycle 8: Memory sw, ...



# Cycle 9: Writeback sw, ...



# iClicker Question

Pipelining is great because:

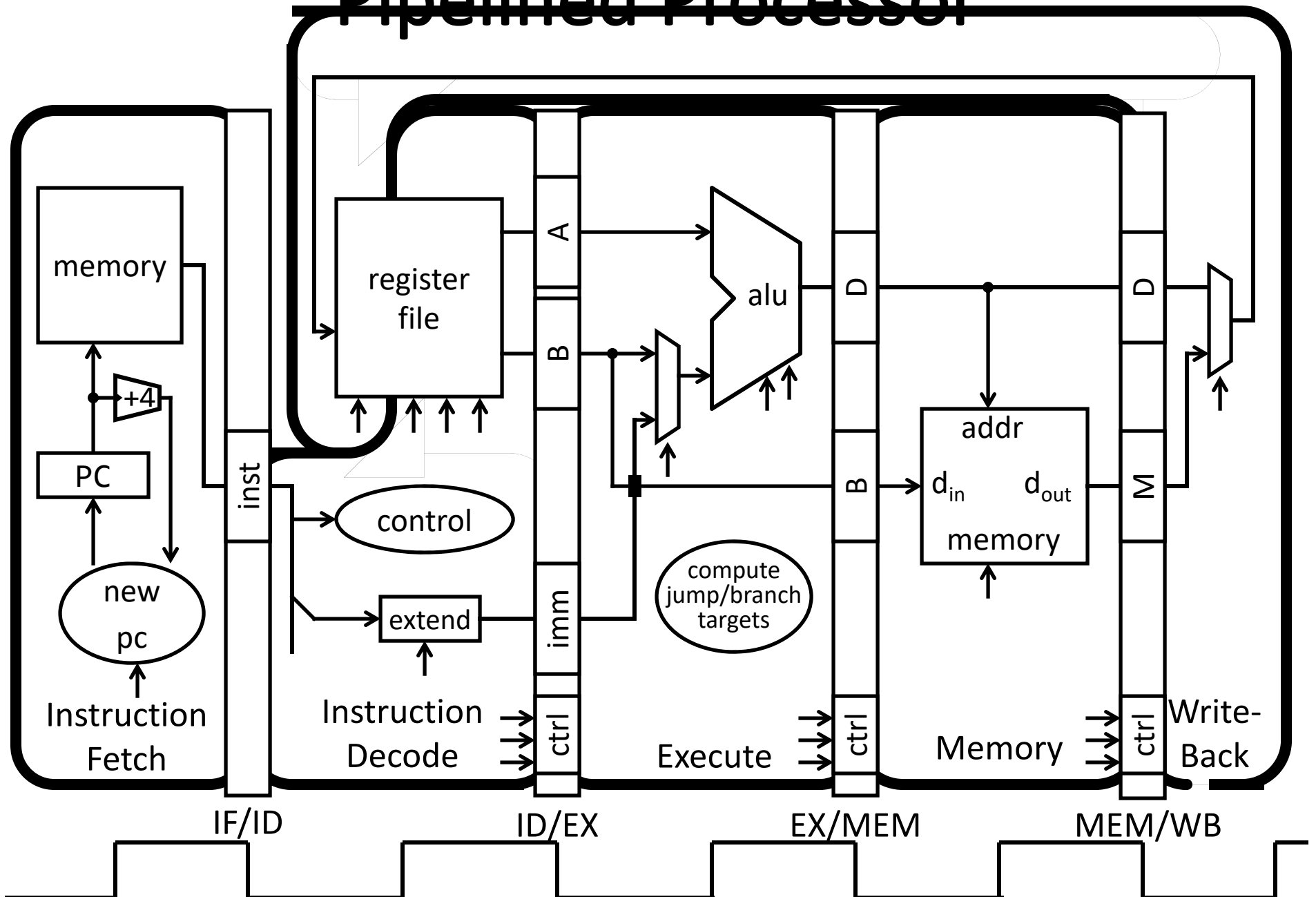
- A. You can fetch and decode the same instruction at the same time.
- B. You can fetch two instructions at the same time.
- C. You can fetch one instruction while decoding another.
- D. Instructions only need to visit the pipeline stages that they require.
- E. C and D

# iClicker Question

Pipelining is great because:

- A. You can fetch and decode the same instruction at the same time.
- B. You can fetch two instructions at the same time.
- C. You can fetch one instruction while decoding another.
- D. Instructions only need to visit the pipeline stages that they require.
- E. C and D

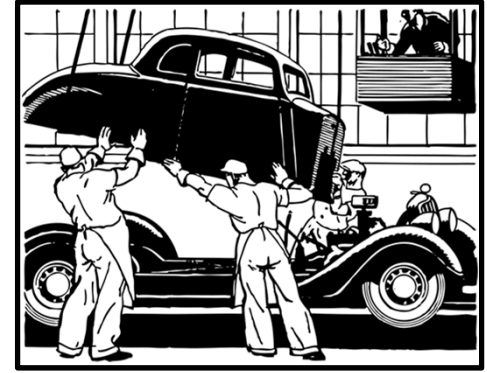
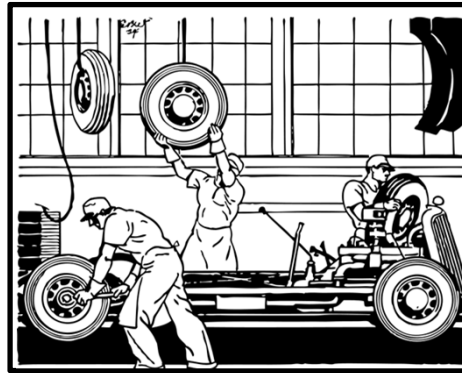
# Pipelined Processor



# Agenda

## 5-stage Pipeline

- Implementation
- Working Example



## Hazards

- Structural
- Data Hazards
- Control Hazards

# Hazards

Correctness problems associated w/processor design

## 1. Structural hazards

Same resource needed for different purposes at the same time (Possible: ALU, Register File, Memory)

## 2. Data hazards

Instruction output needed before it's available

## 3. Control hazards

Next instruction PC unknown at time of Fetch

# Dependences and Hazards

**Dependence:** relationship between two insns

- **Data:** two insns use same storage location
- **Control:** 1 insn affects whether another executes at all
- *Not a bad thing*, programs would be boring otherwise
- Enforced by making older insn go before younger one
  - Happens naturally in single-/multi-cycle designs
  - But not in a pipeline

**Hazard:** dependence & possibility of wrong insn order

- Effects of wrong insn order cannot be externally visible
- *Hazards are a bad thing*: most solutions either complicate the hardware or reduce performance

# iClicker Question

## Data Hazards

- register file (RF) reads occur in stage 2 (ID)
- RF writes occur in stage 5 (WB)
- RF written in ½ half, read in second ½ half of cycle

x10:      add r3  $\leftarrow$  r1, r2

x14:      sub r5  $\leftarrow$  r3, r4

1. Is there a dependence?
2. Is there a hazard?

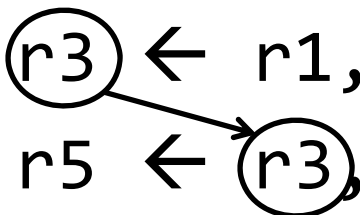
- A) Yes
  - B) No
  - C) Cannot tell with the information given.

# iClicker Question

## Data Hazards

- register file (RF) reads occur in stage 2 (ID)
- RF writes occur in stage 5 (WB)
- RF written in ½ half, read in second ½ half of cycle

x10:      add (r3) ← r1, r2  
x14:      sub r5 ← (r3), r4



1. Is there a dependence?
2. Is there a hazard?

- ☒ A) Yes for both  
☐ B) No  
☐ C) Cannot tell with the information given.

## iClicker Follow-up

Which of the following statements is true?

- A. Whether there is a data dependence between two instructions depends on the machine the program is running on.
- B. Whether there is a data hazard between two instructions depends on the machine the program is running on.
- C. Both A & B
- D. Neither A nor B

## iClicker Follow-up

Which of the following statements is true?

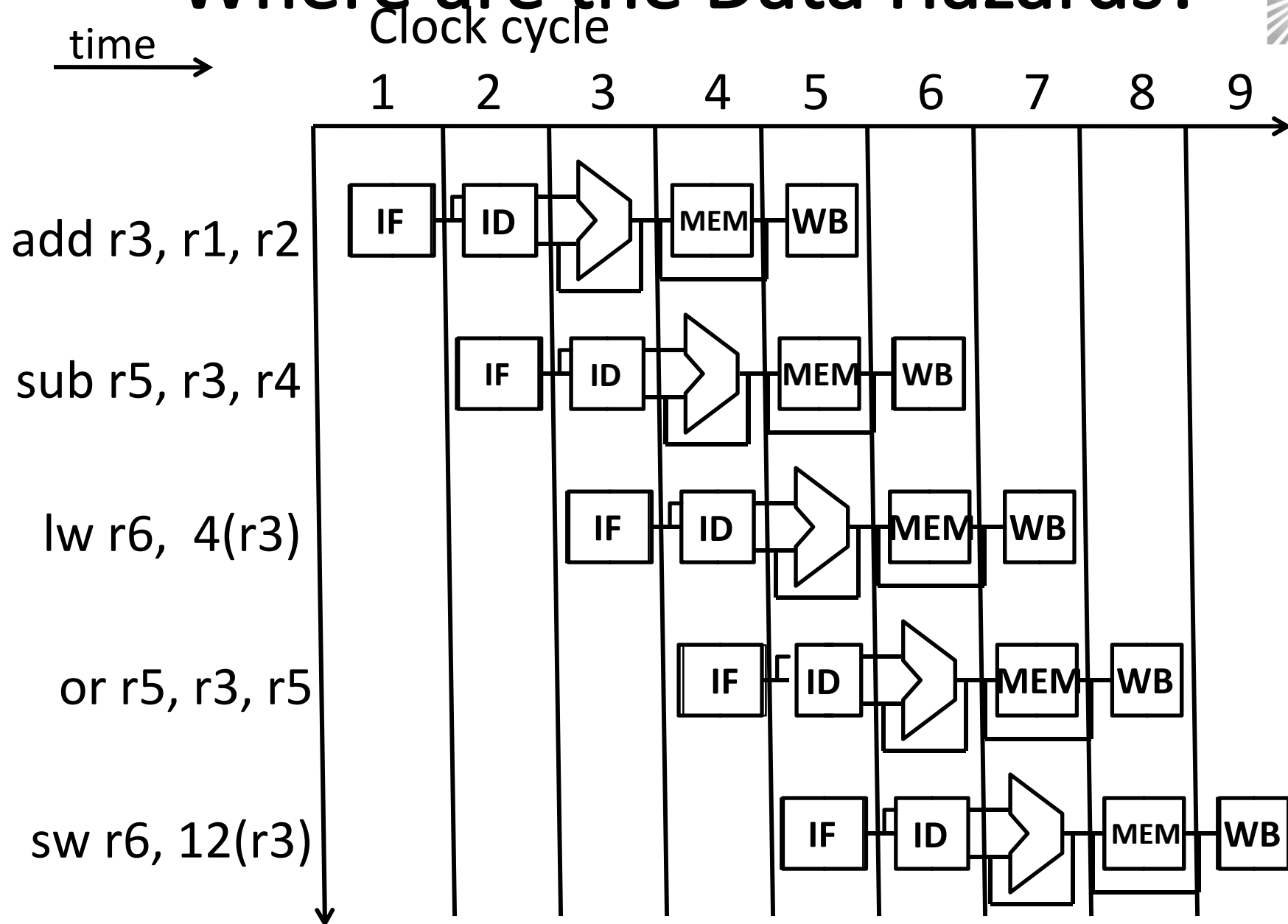
A. Whether there is a data dependence between two instructions depends on the machine the program is running on.

B. Whether there is a data hazard between two instructions depends on the machine the program is running on.

C. Both A & B

D. Neither A nor B

# Where are the 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

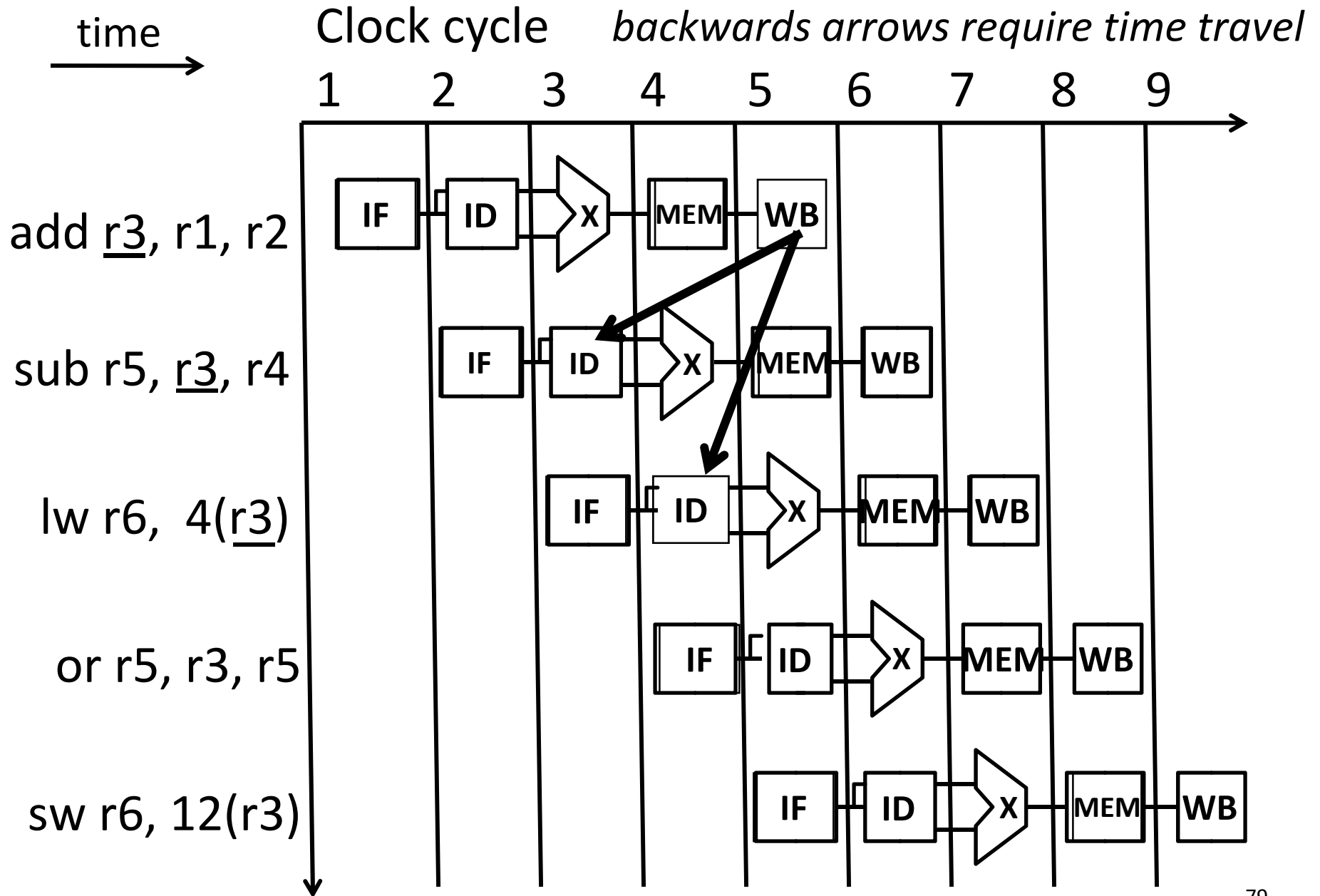
B) 2

C) 3

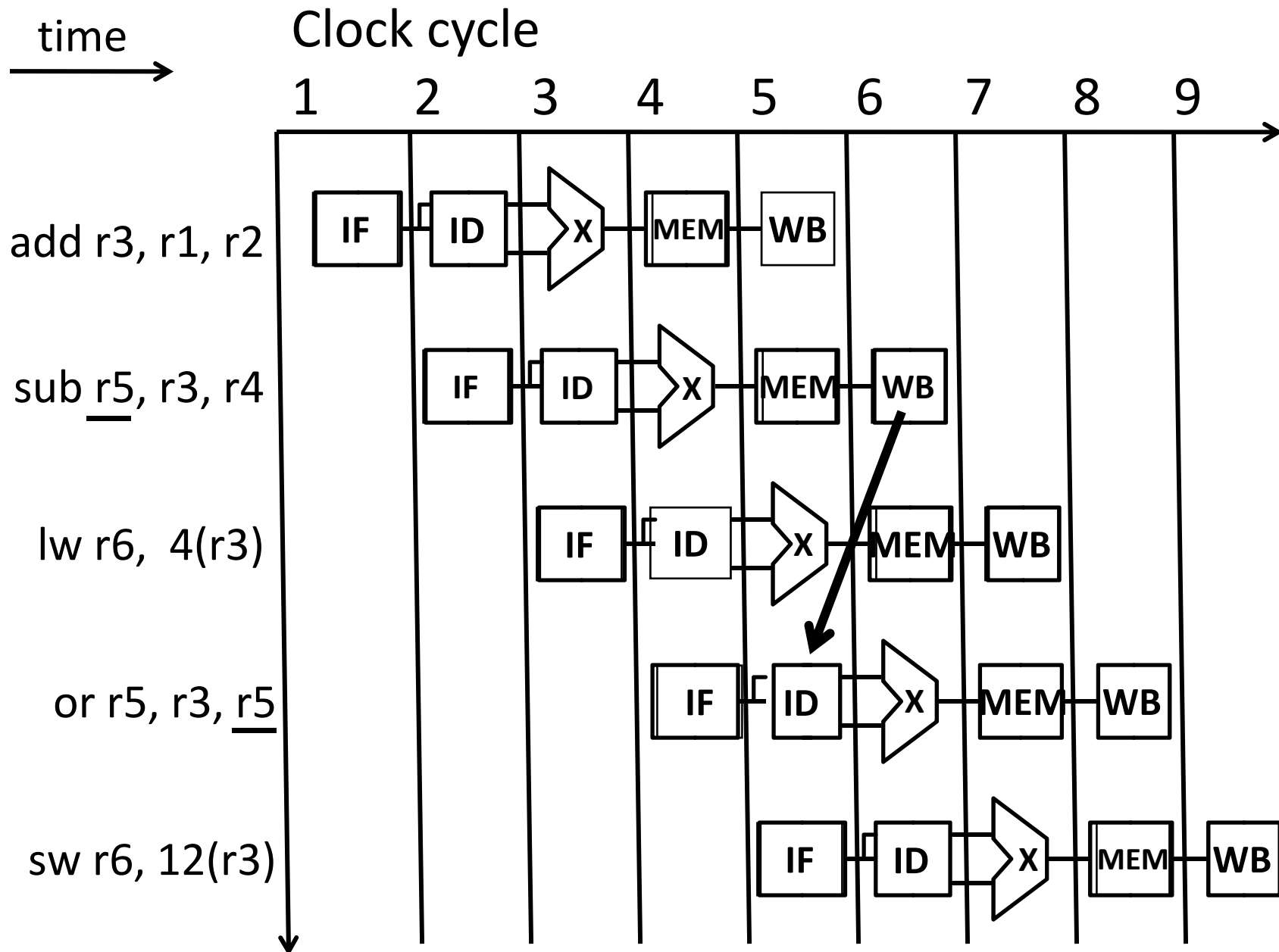
D) 4

E) 5

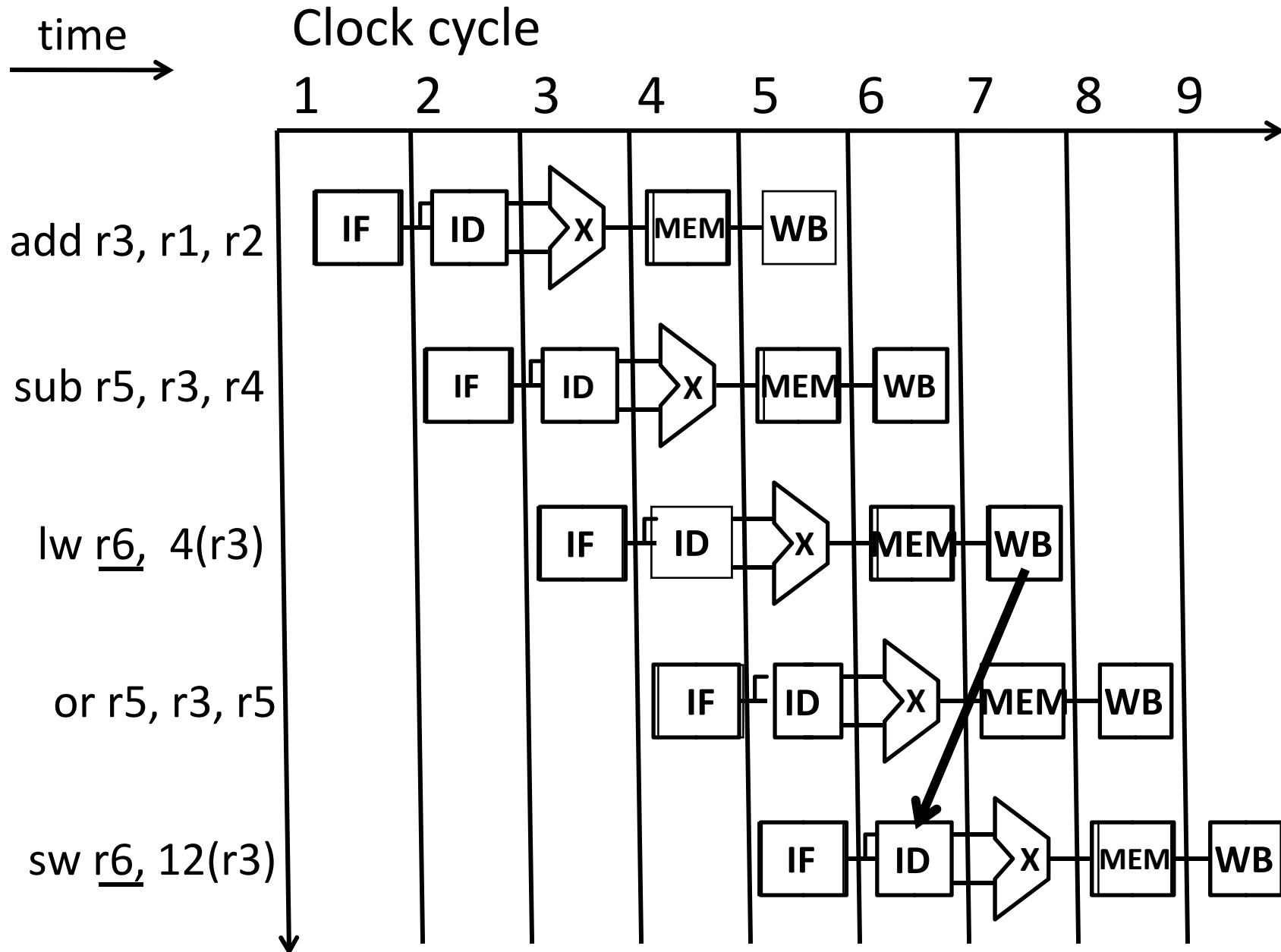
# Visualizing Data Hazards (1)



# Visualizing Data Hazards (2)



# Visualizing Data Hazards (3)

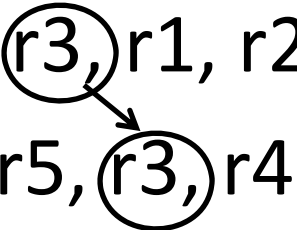


# Data Hazards

## Data Hazards

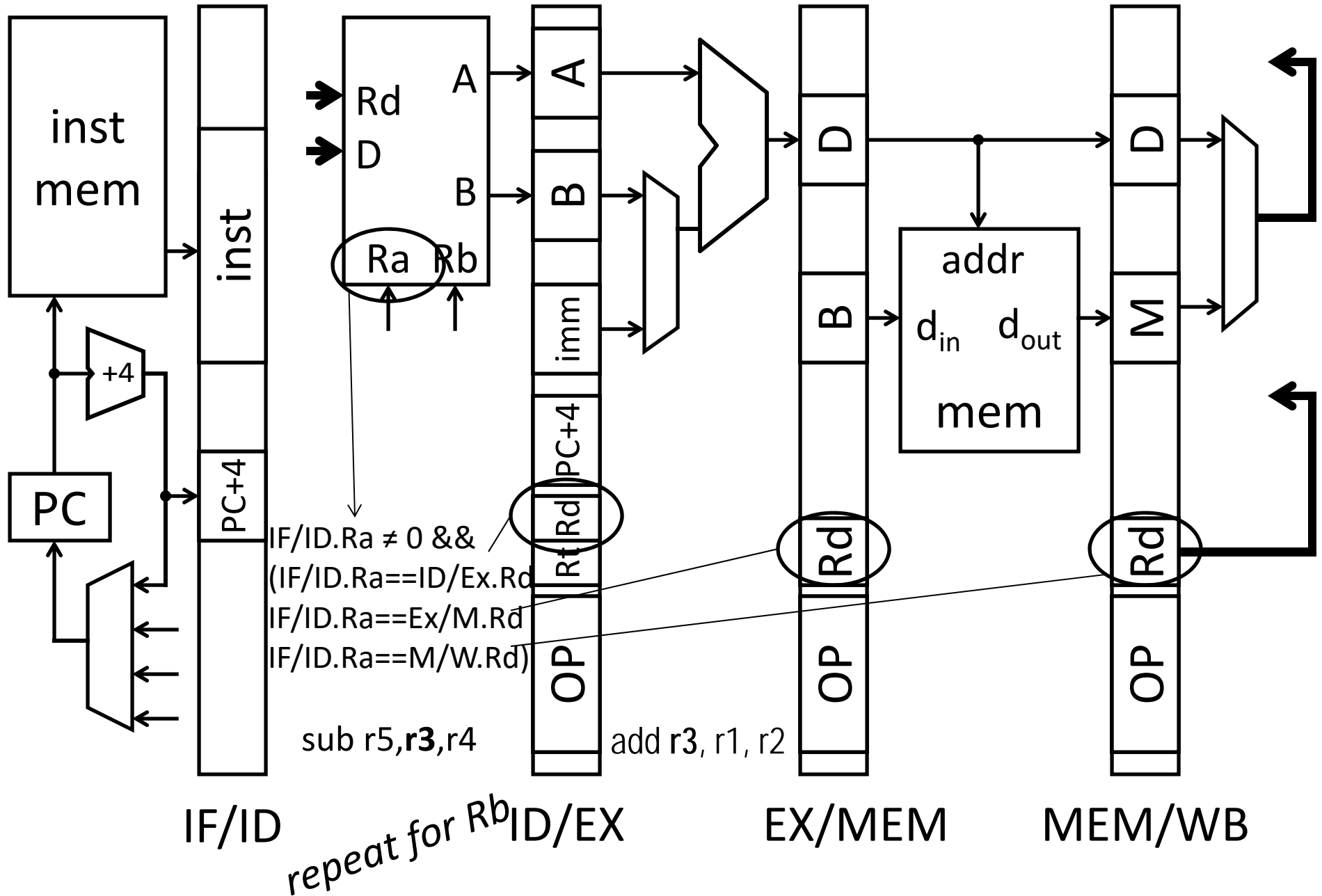
- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

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 &&

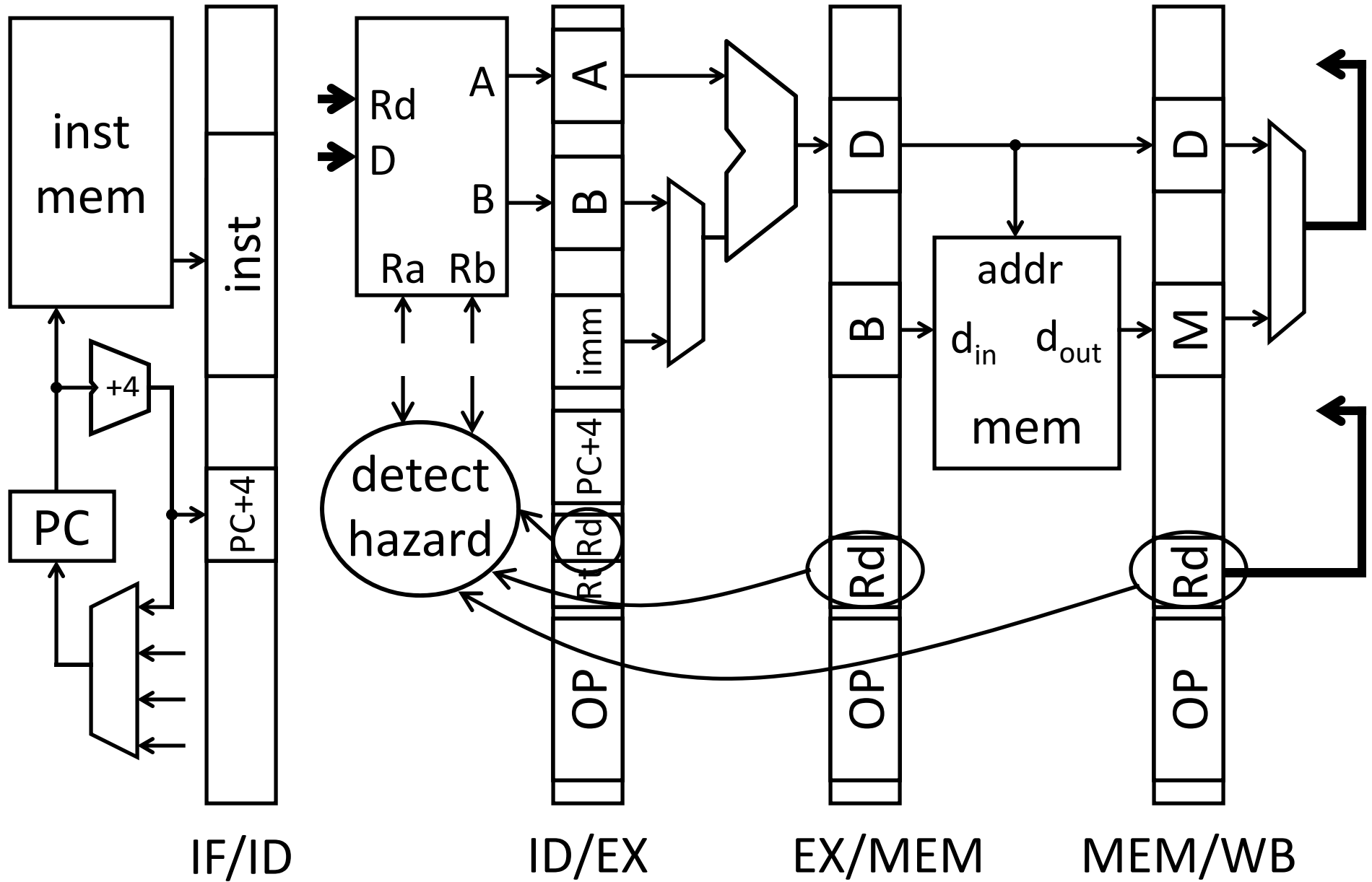
(IF/ID.Ra == ID/EX.Rd ||

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?

# iClicker

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

# Possible Responses to Data Hazards

## 1. Do Nothing

- Change the ISA to match implementation
- “Hey compiler: don’t create code w/data hazards!”  
*(We can do better than this)*

## 2. Stall

- Pause current and subsequent instructions till safe

## 3. Forward/bypass

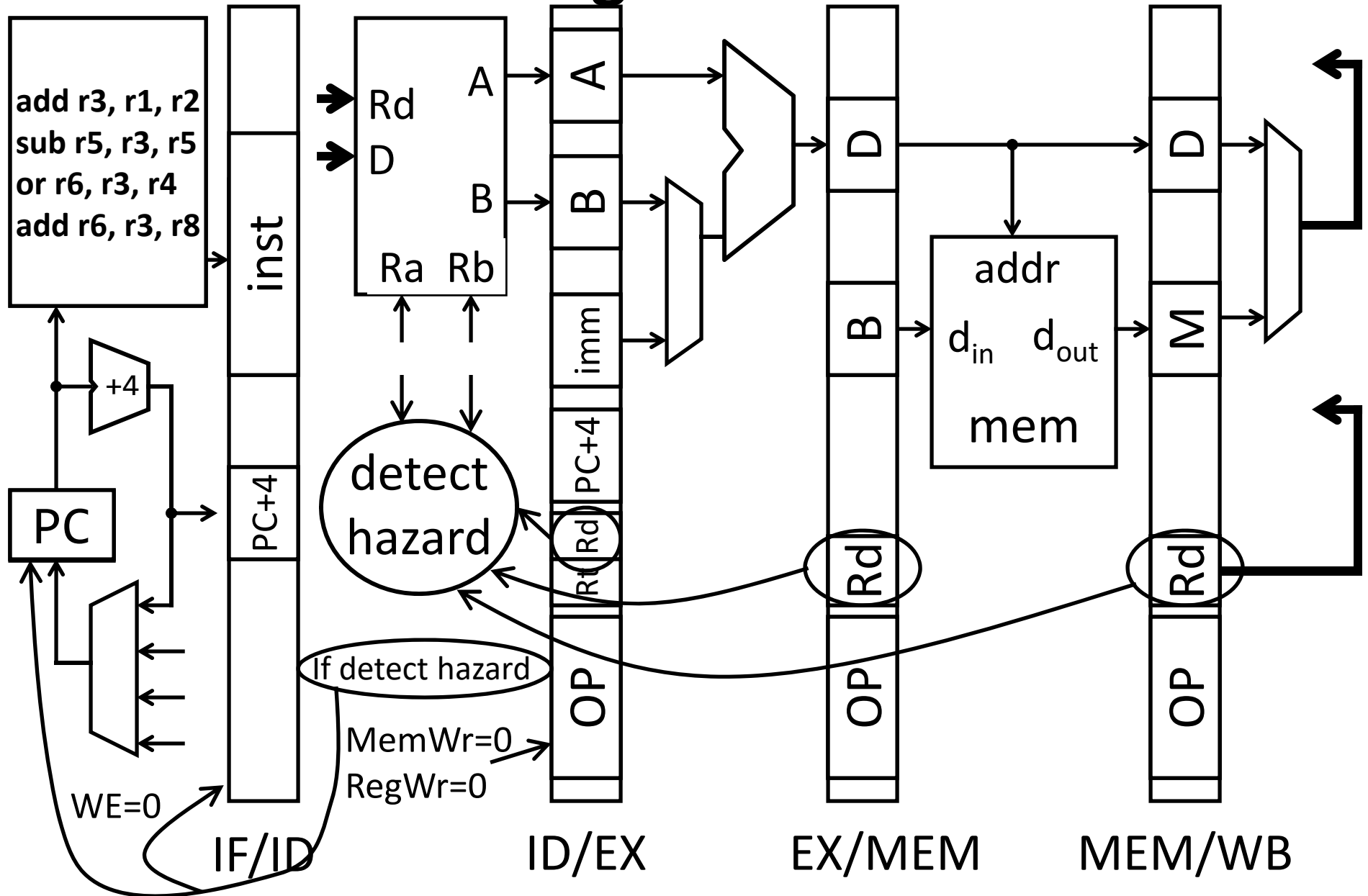
- Forward data value to where it is needed  
*(Only works if value actually exists already)*

# Stalling

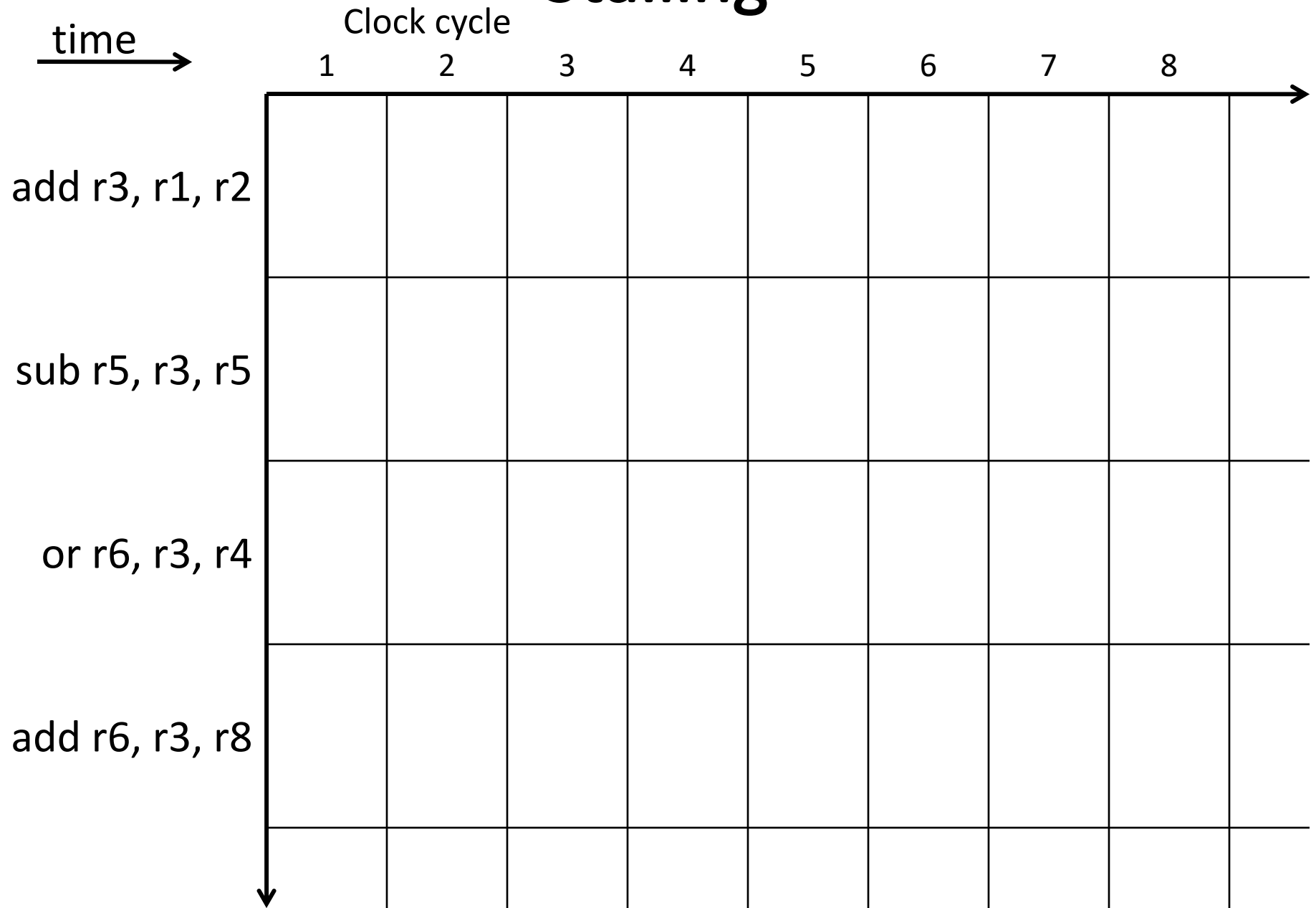
## How to stall an instruction in ID stage

- 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

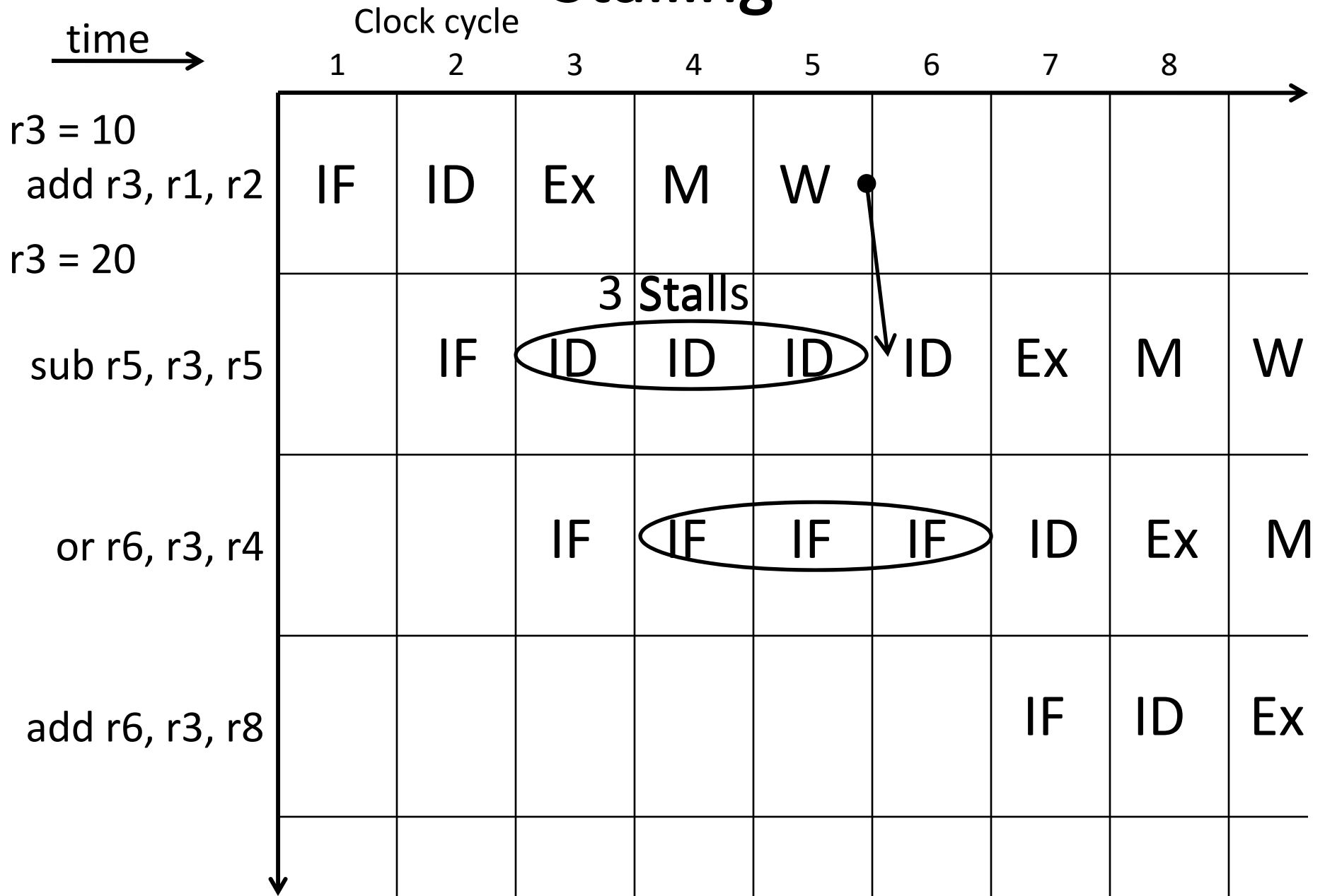
# Detecting Data Hazards



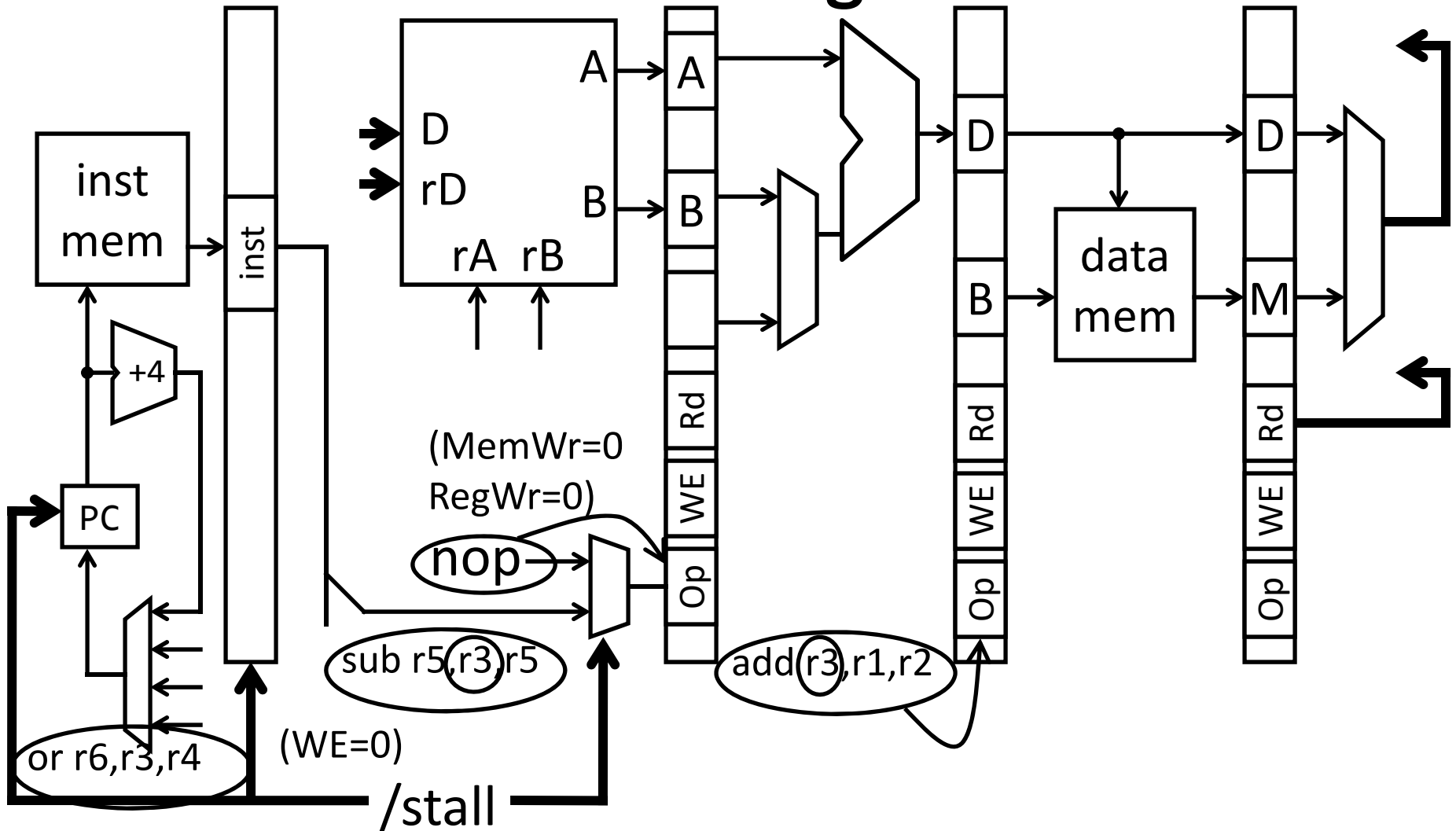
# Stalling



# Stalling



# Stalling



NOP = If(IF/ID.rA  $\neq$  0 &&

(IF/ID.rA==ID/Ex.Rd

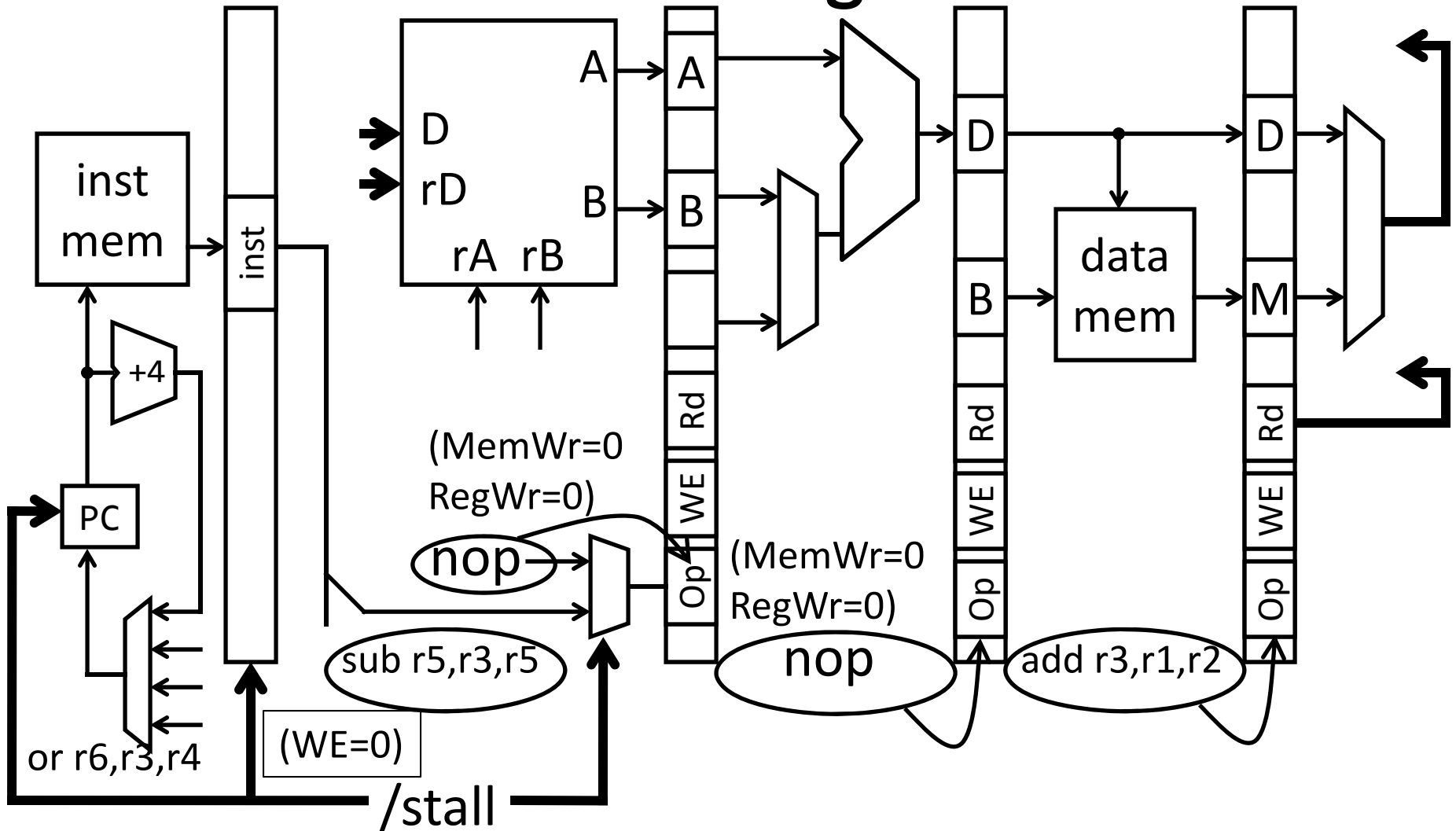
IF/ID.rA==Ex/M.Rd

IF/ID.rA==M/W.Rd))



STALL CONDITION MET

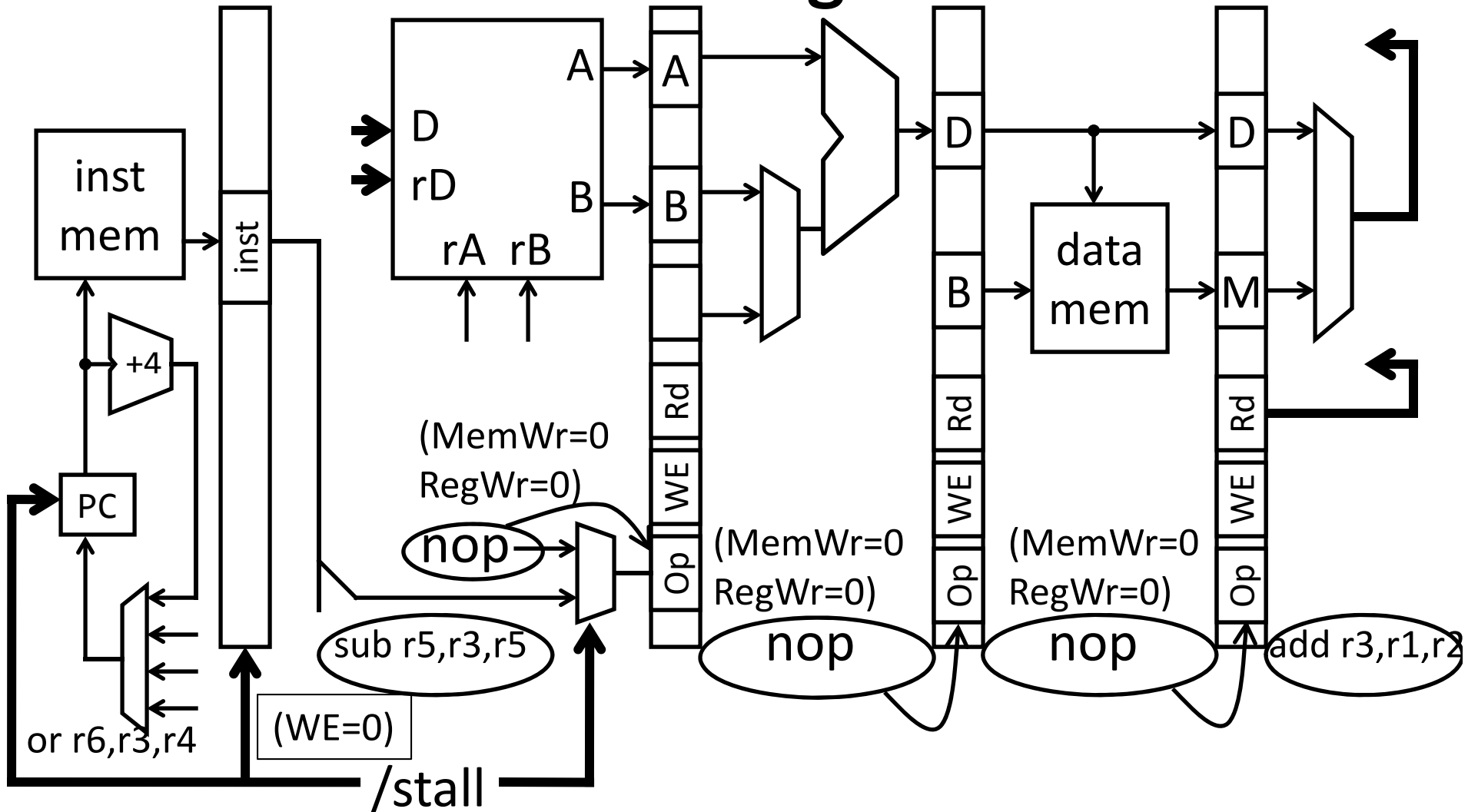
# Stalling



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

← STALL CONDITION MET

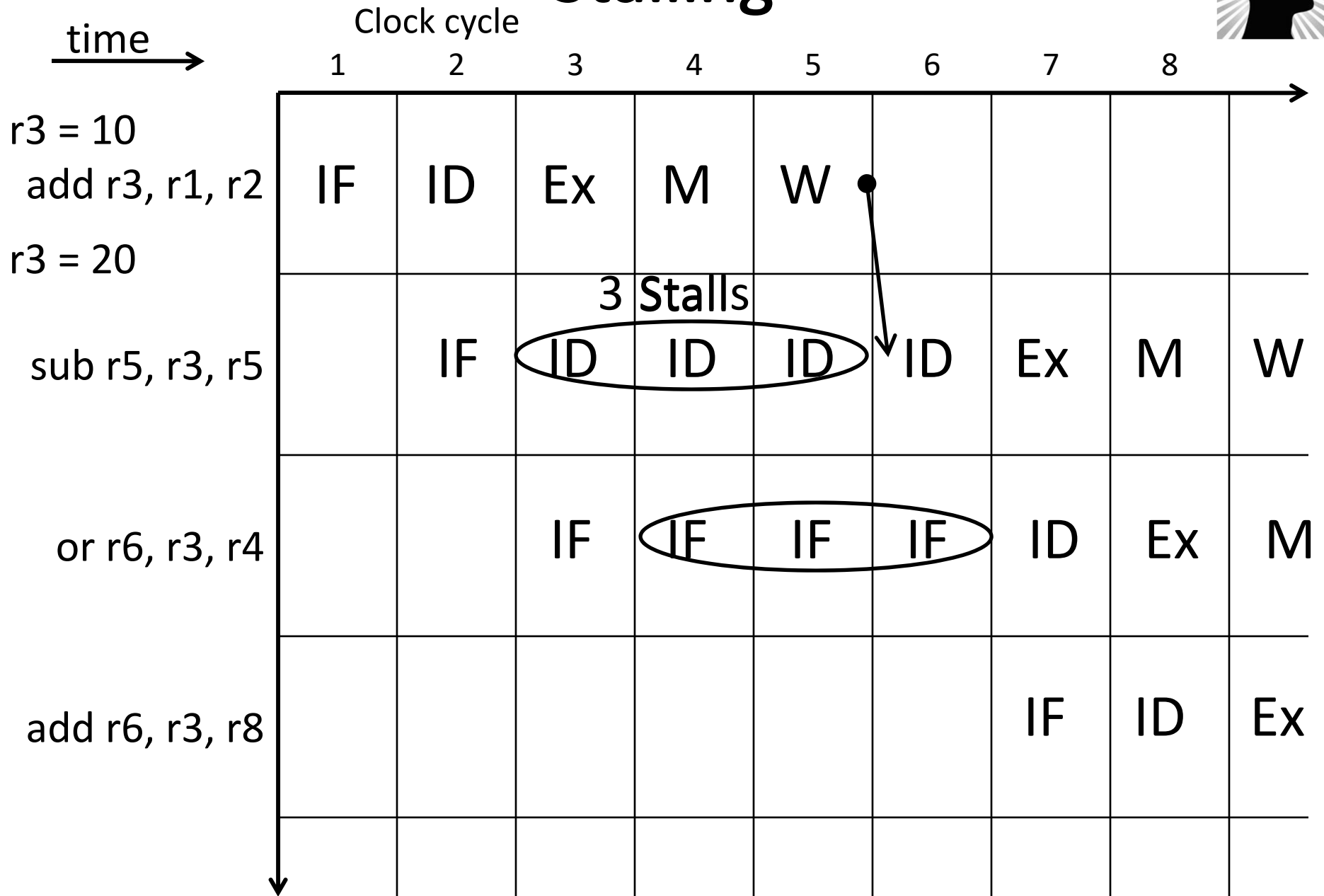
# Stalling



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

STALL CONDITION MET

# 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.

# Possible Responses to Data Hazards

## 1. Do Nothing

- Change the ISA to match implementation
- “Compiler: don’t create code with data hazards!”  
*(Nice try, we can do better than this)*

## 2. Stall

- Pause current and subsequent instructions till safe

## 3. Forward/bypass

- Forward data value to where it is needed  
*(Only works if value actually exists already)*

# Forwarding

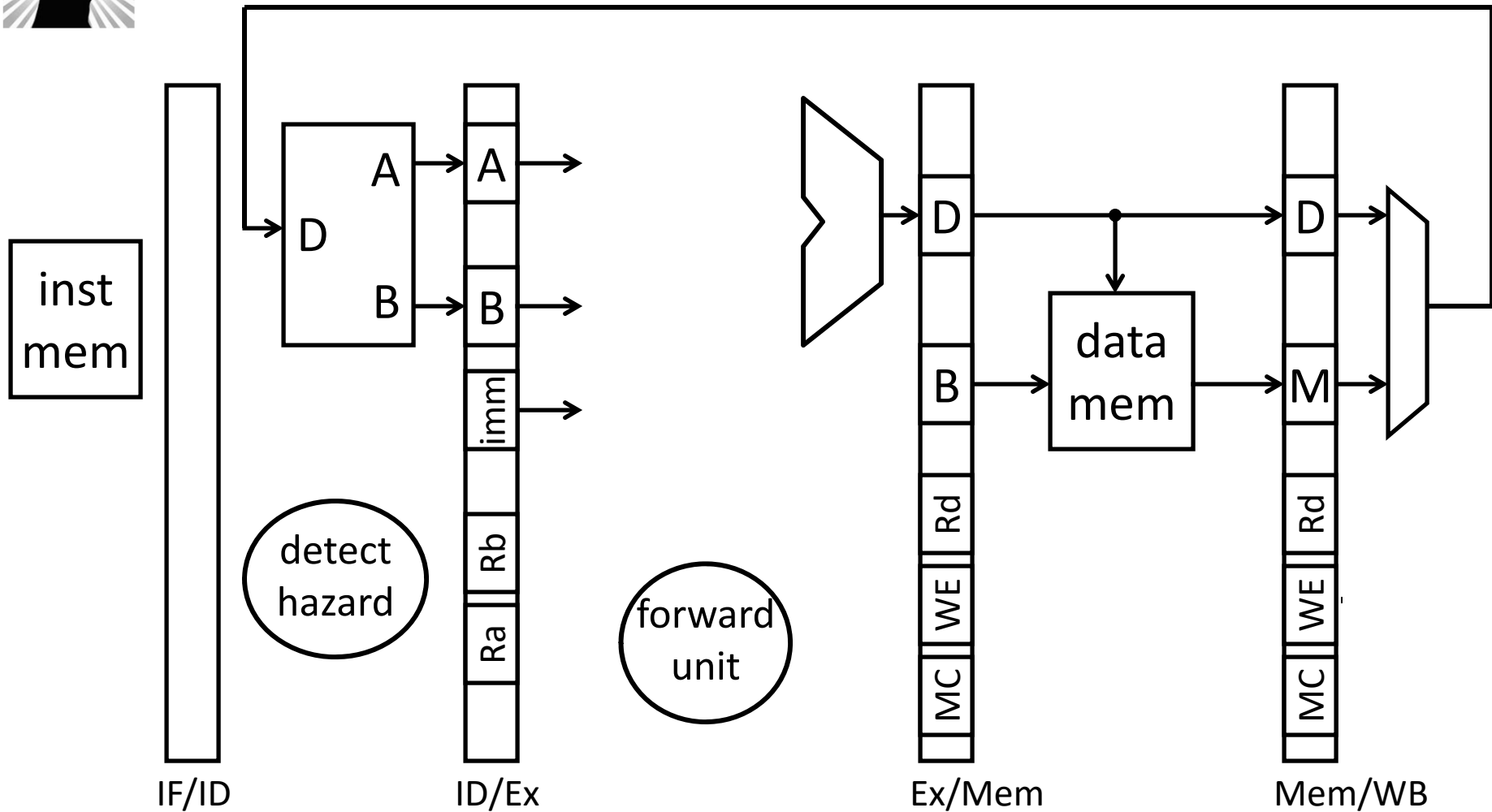
Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).

Three types of forwarding/bypass

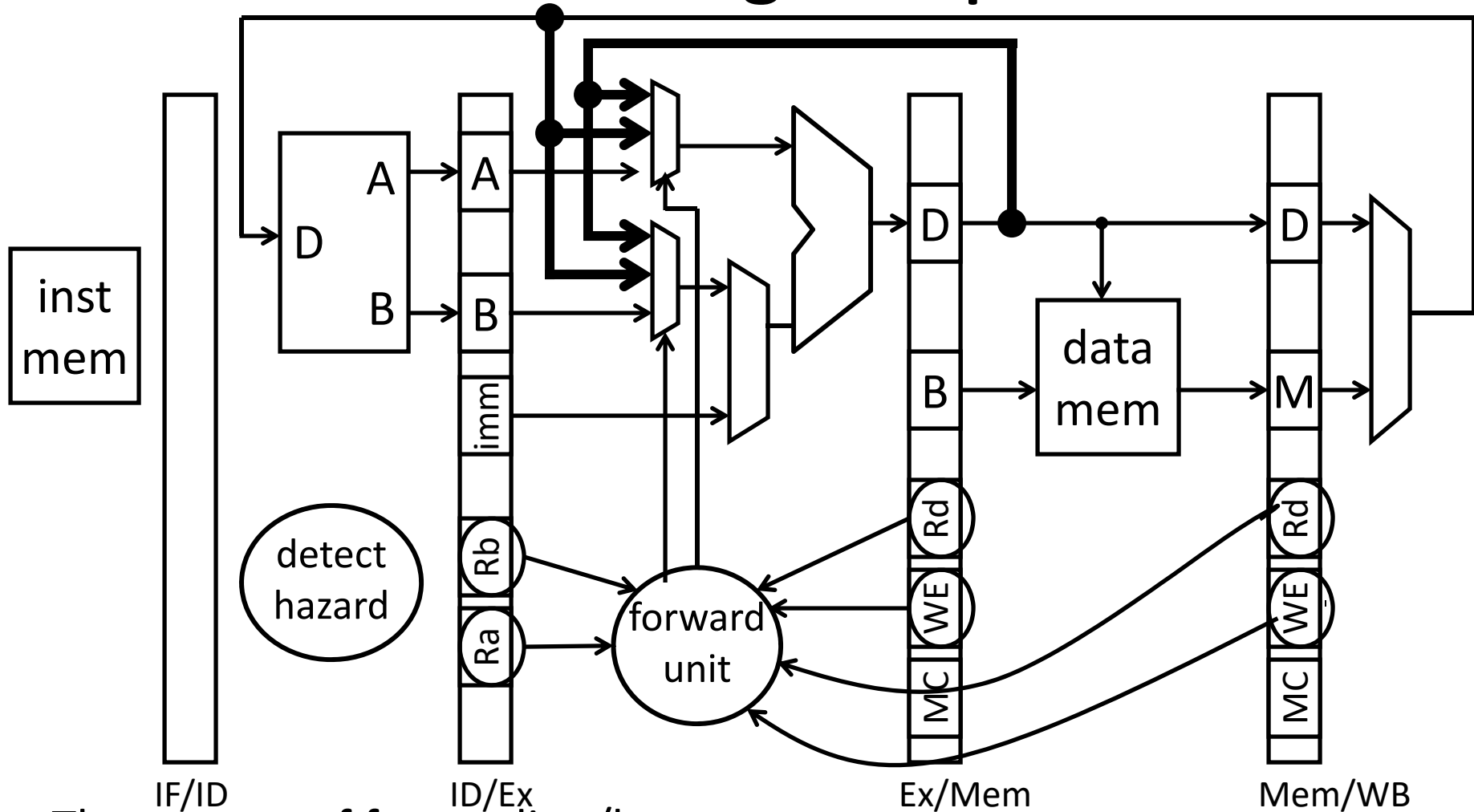
- Forwarding from Ex/Mem registers to Ex stage ( $M \rightarrow Ex$ )
- Forwarding from Mem/WB register to Ex stage ( $W \rightarrow Ex$ )
- RegisterFile Bypass



# Add the Forwarding Datapath



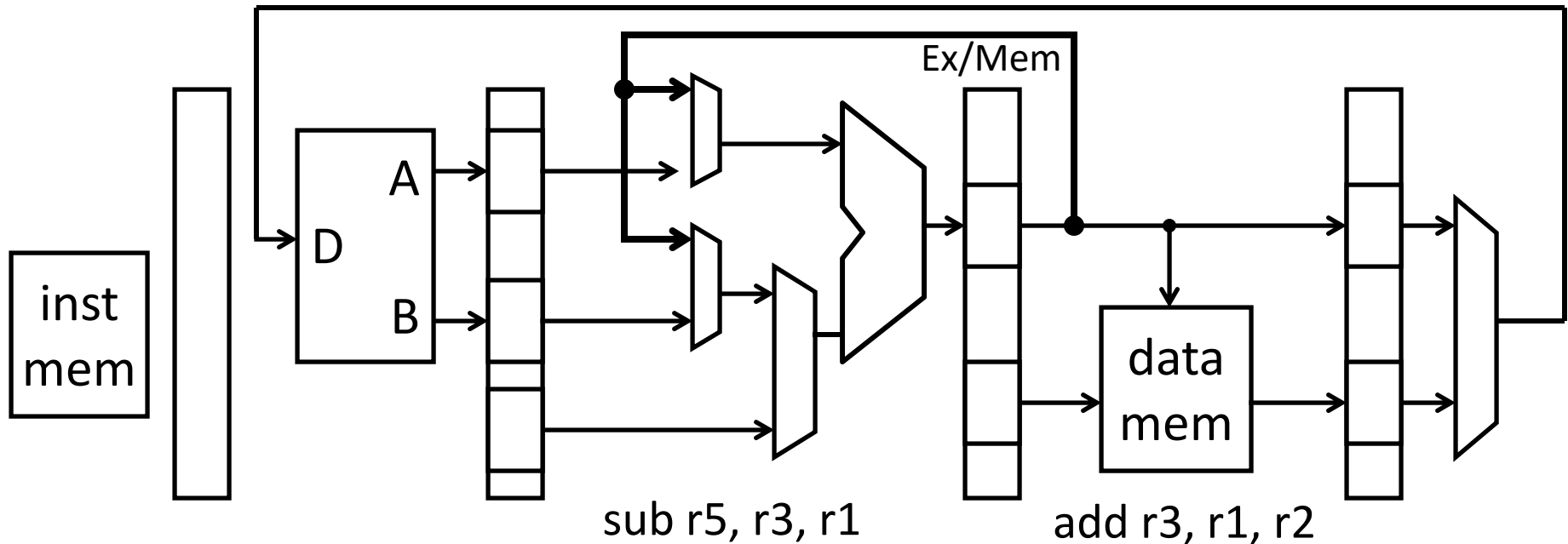
# Forwarding Datapath



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 1: Ex/MEM → EX

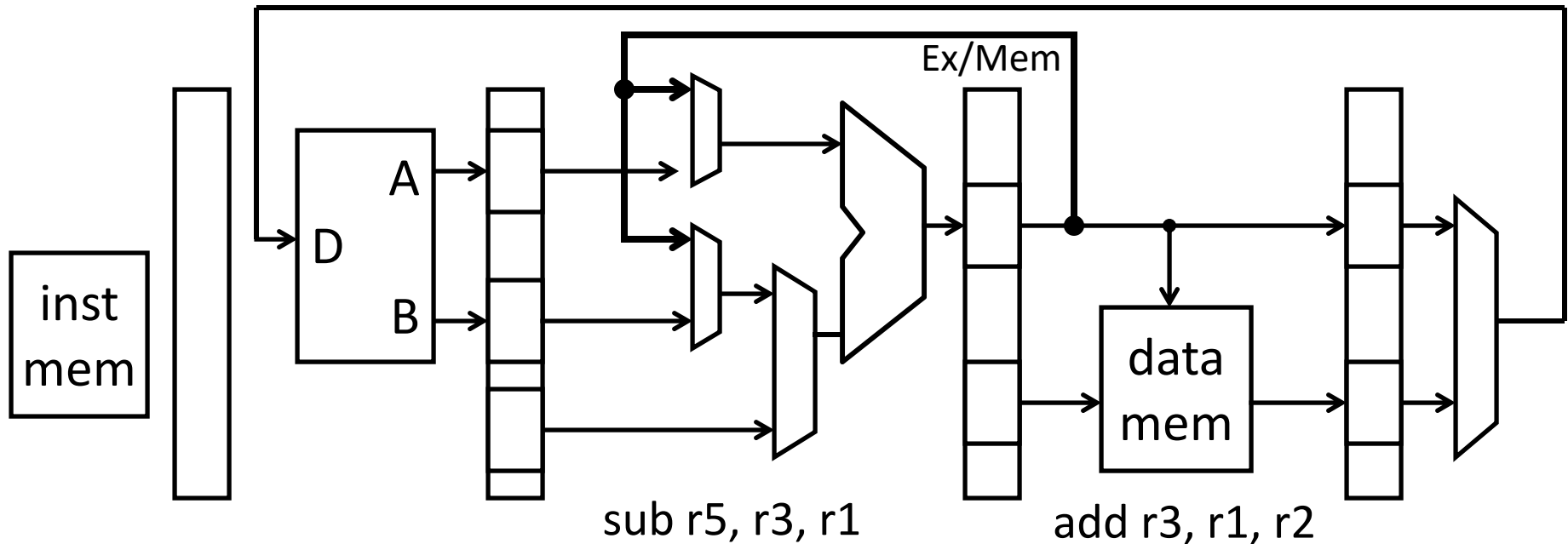


add r3, r1, r2	IF	ID	Ex •	M	W	
sub r5, r3, r1		IF	ID	Ex	M	W

Problem: EX needs ALU result that is in MEM stage

Solution: add a bypass from EX/MEM.D to start of EX

# Forwarding Datapath 1: Ex/MEM $\rightarrow$ EX



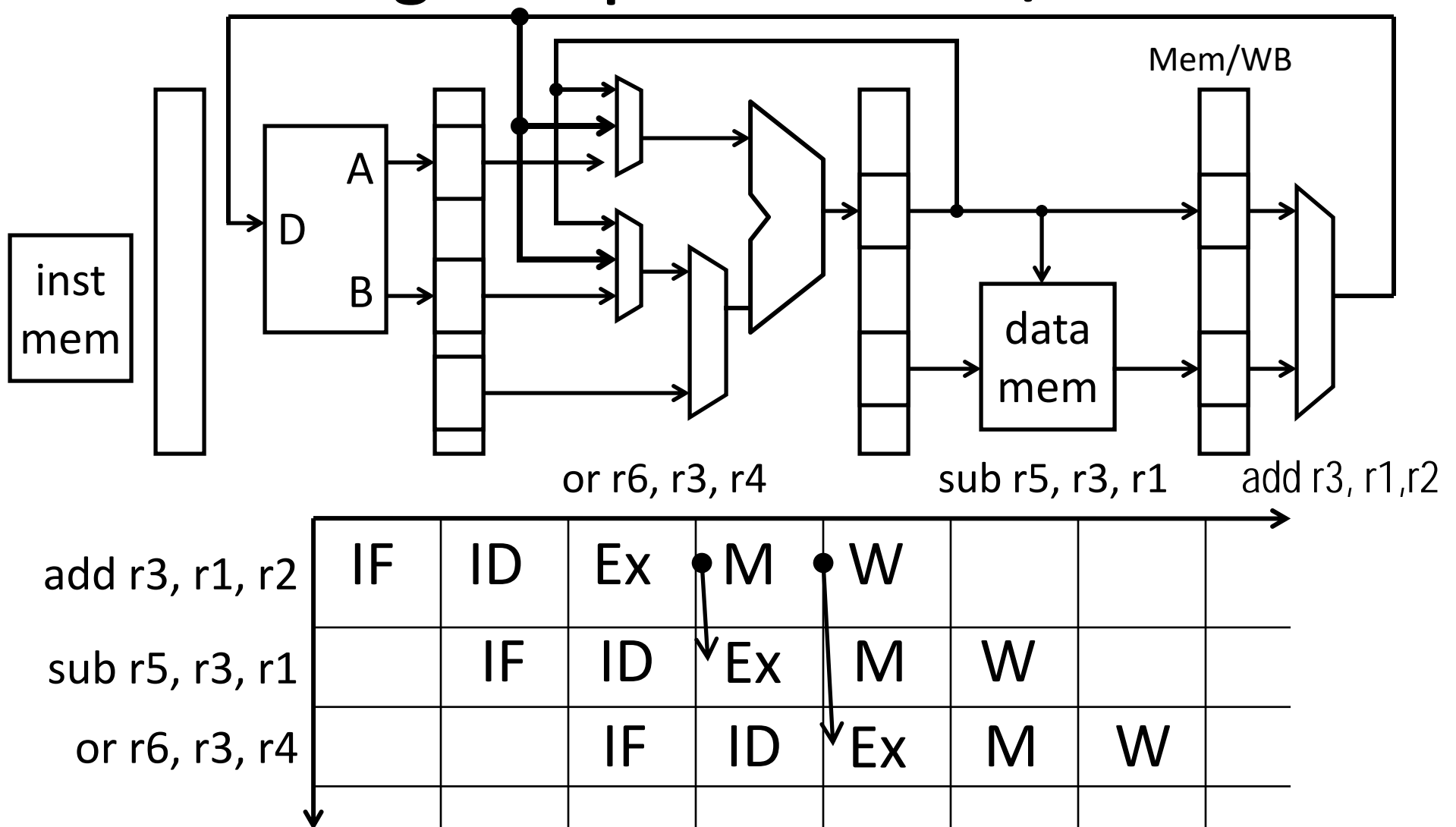
Detection Logic in Ex Stage:

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

$\text{ID/Ex.Ra} == \text{Ex/M.Rd})$

$||$  (same for Rb)

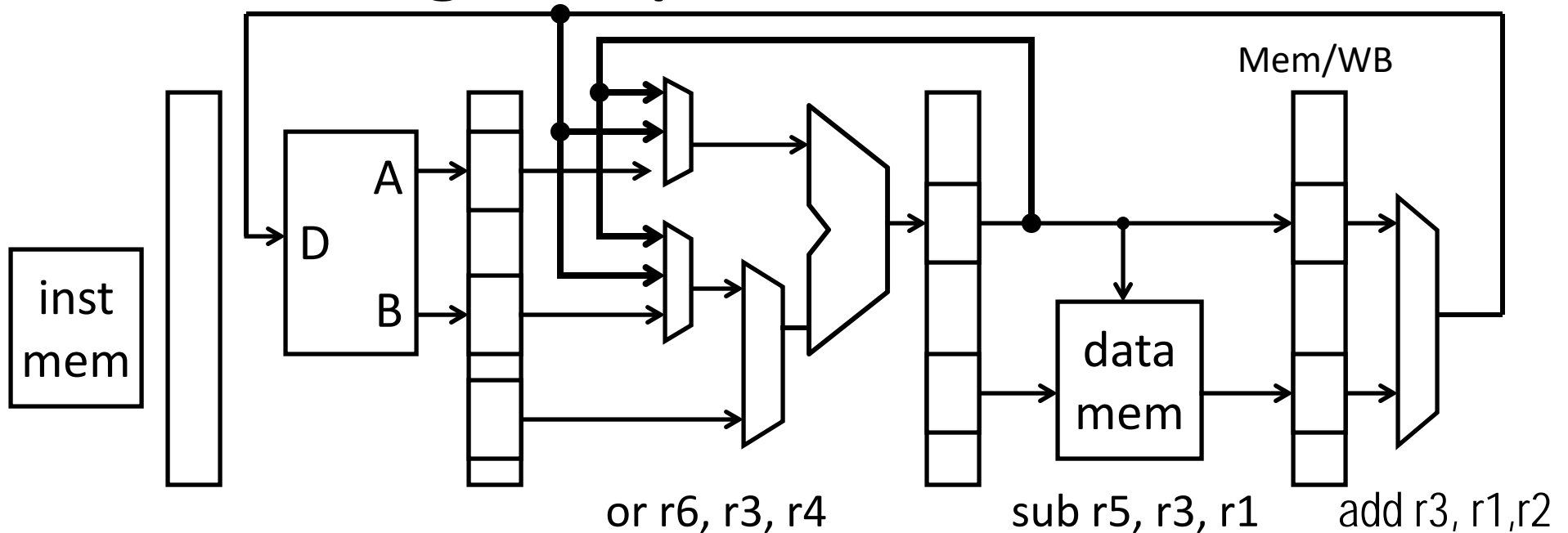
# Forwarding Datapath 2: Mem/WB → EX



Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

# Forwarding Datapath 2: Mem/WB → EX



Detection Logic:

forward = (M/WB.WE && M/WB.Rd != 0 &&

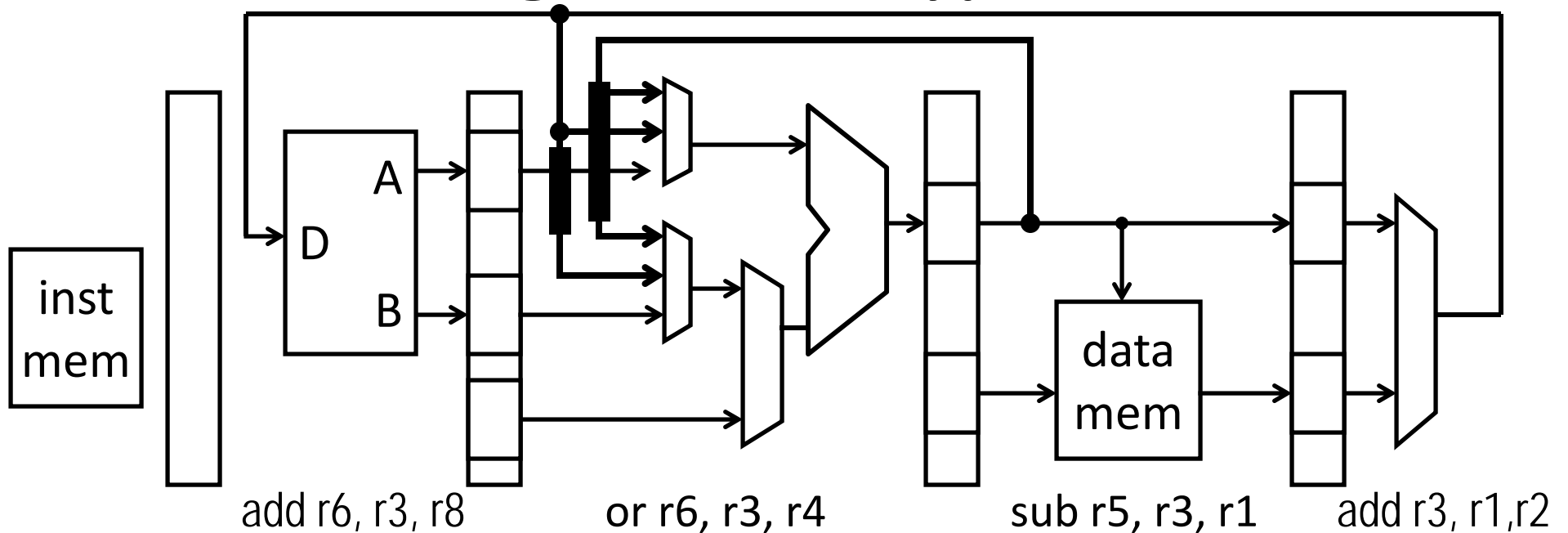
ID/Ex.Ra == M/WB.Rd &&

not (ID/Ex.WE && Ex/M.Rd != 0 &&

ID/Ex.Ra == Ex/M.Rd)

|| (same for Rb)

# Register File Bypass

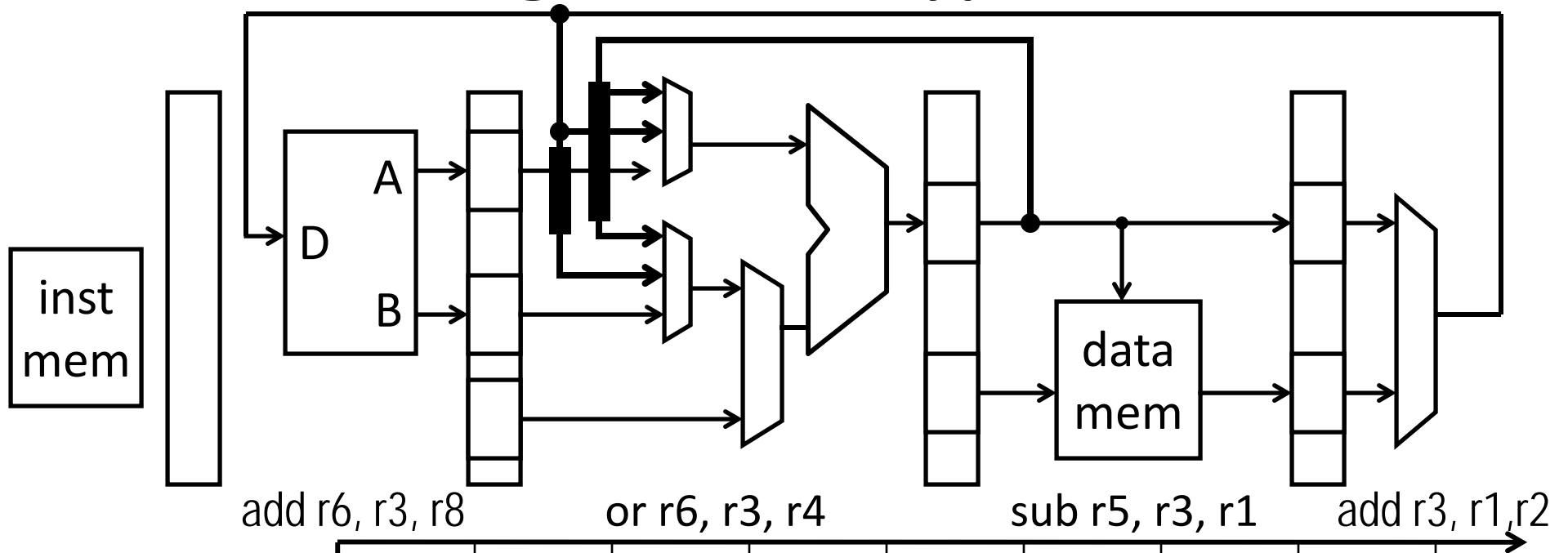


Problem: Reading a value that is currently being written

Solution: 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

# 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

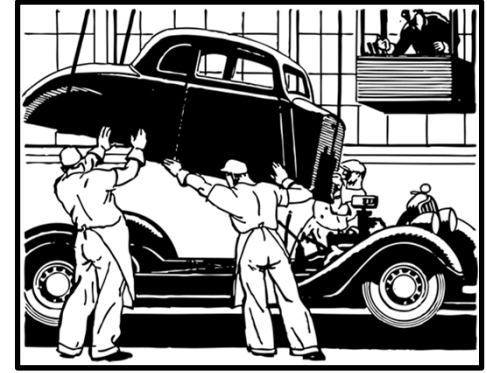
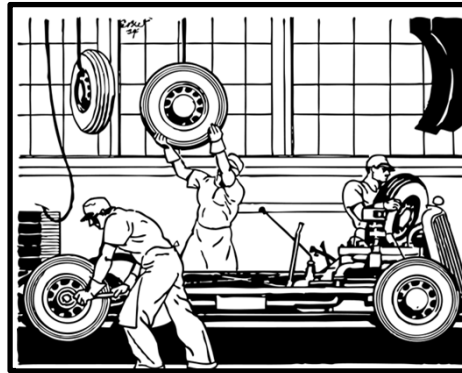
M

W

# Agenda

## 5-stage Pipeline

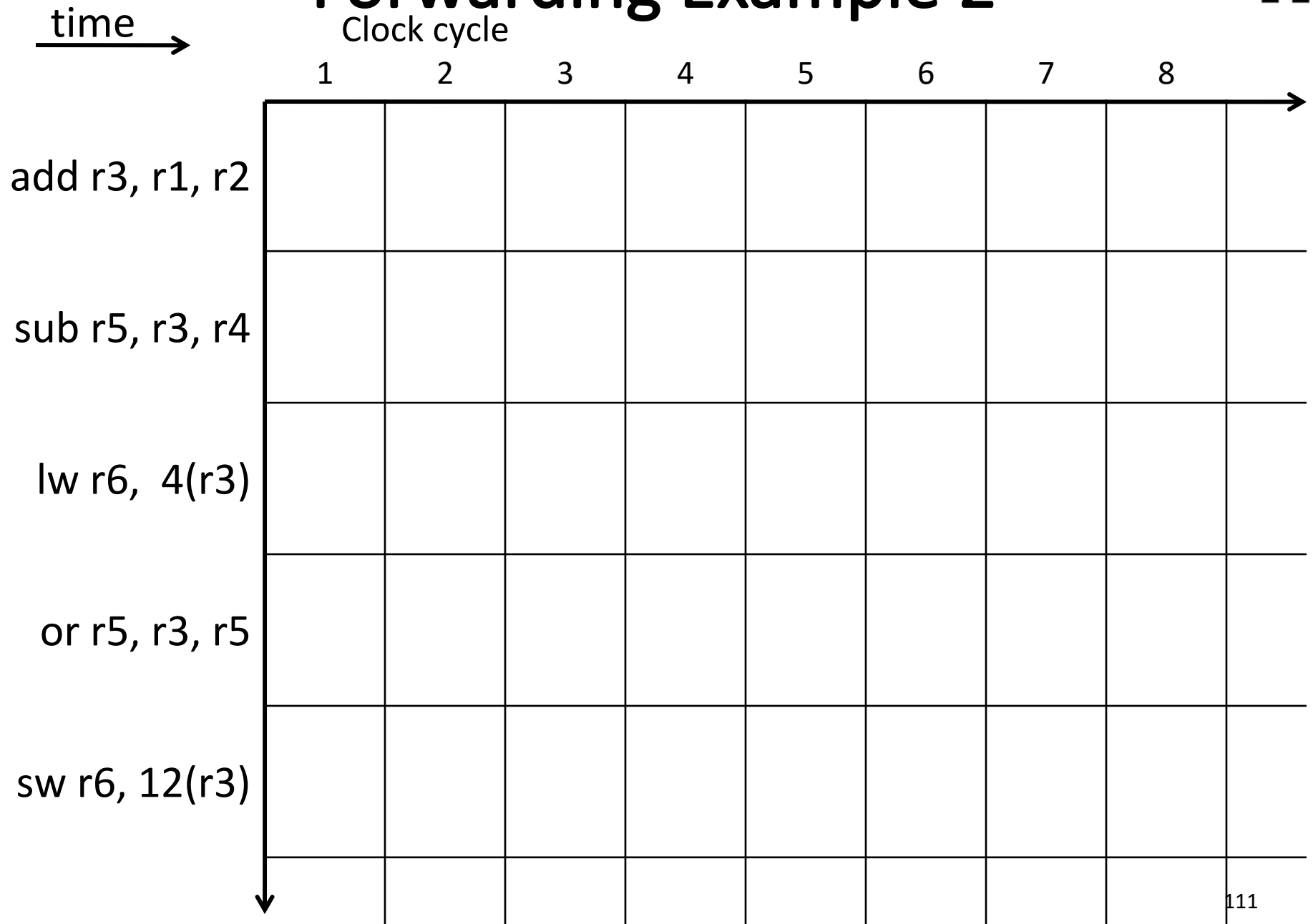
- Implementation
- Working Example



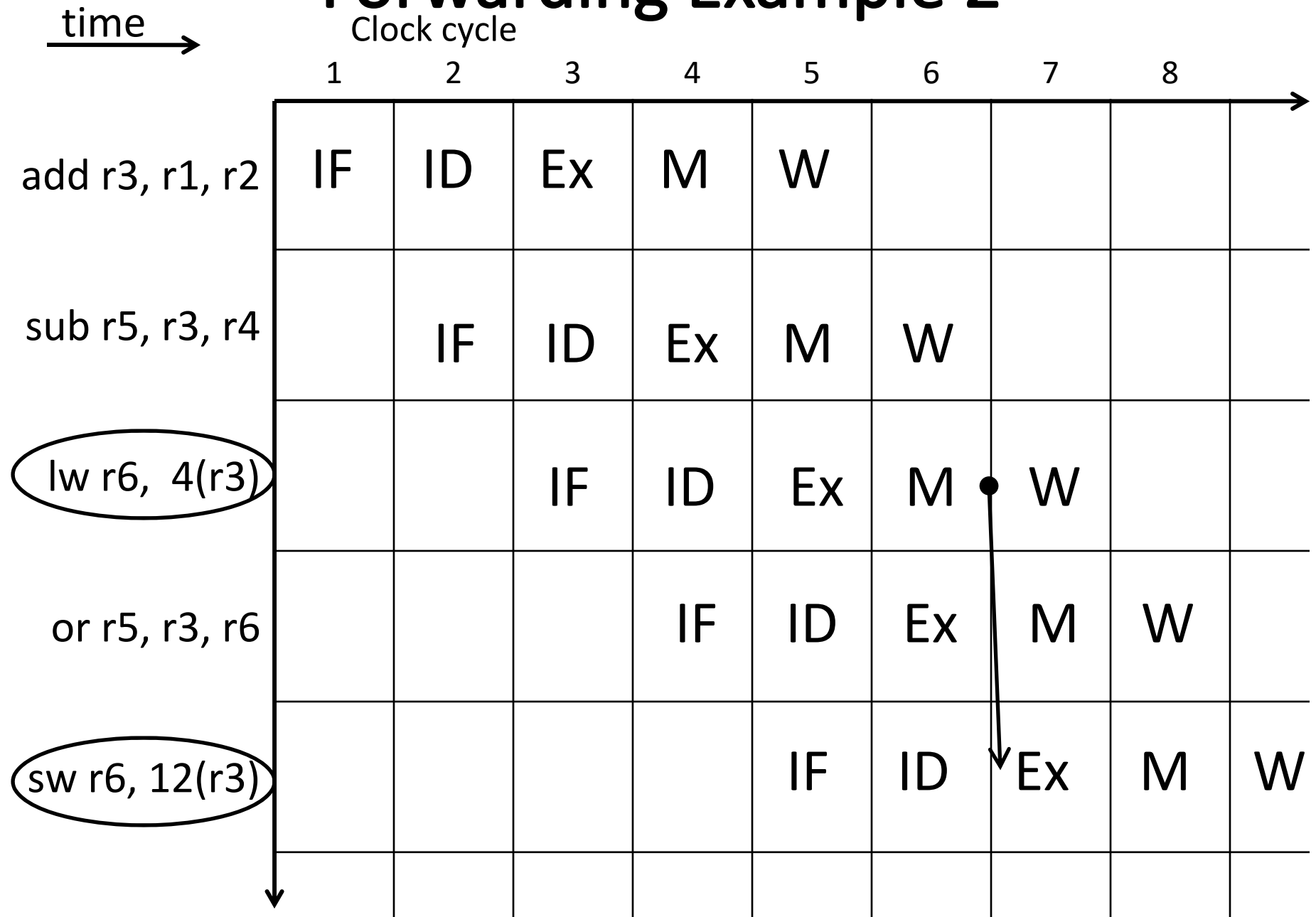
## Hazards

- Structural
- Data Hazards
- Control Hazards

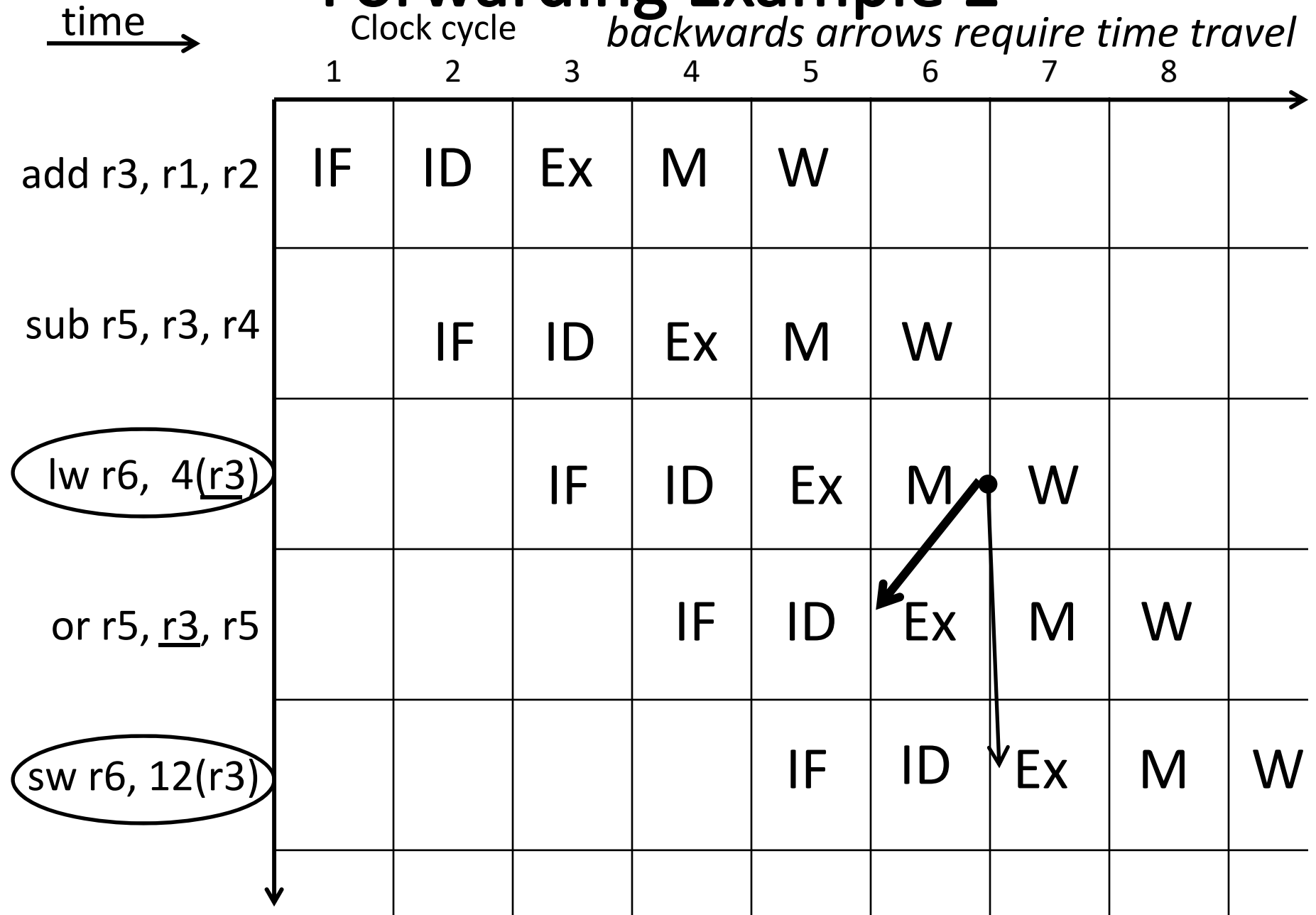
# Forwarding Example 2



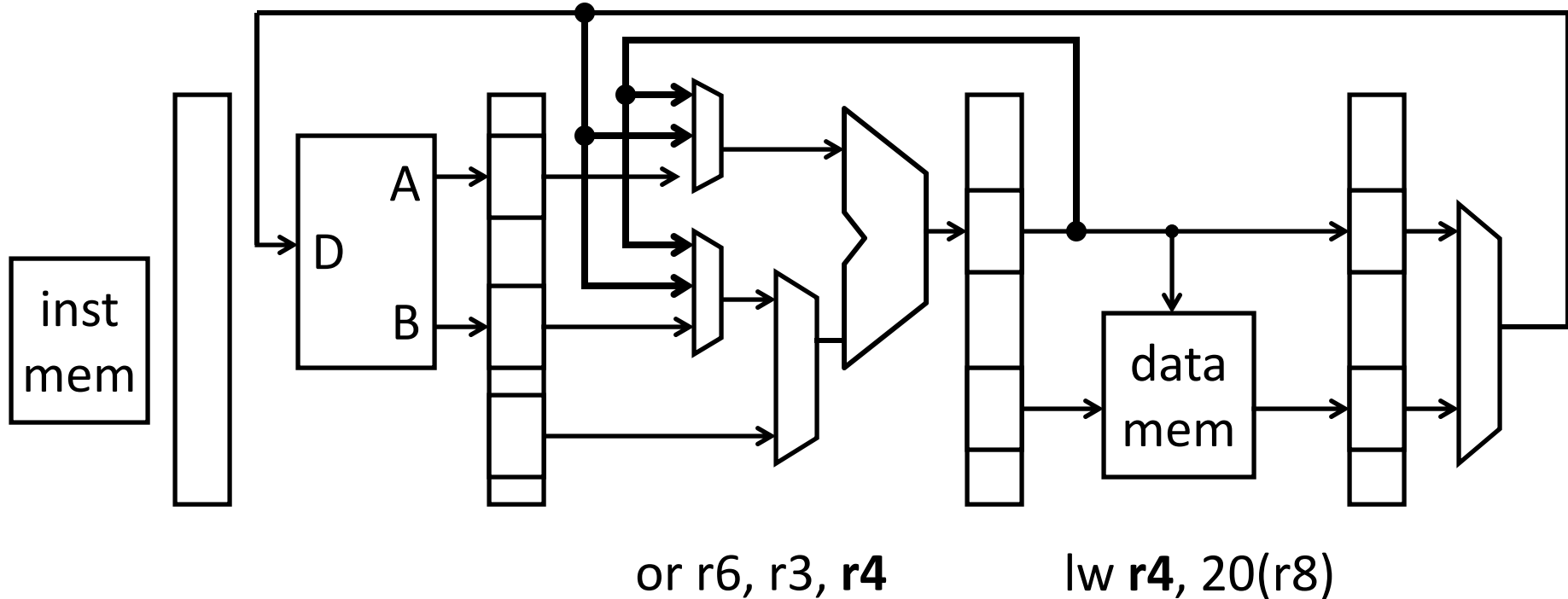
# Forwarding Example 2



# Forwarding Example 2



# Load-Use Hazard Explained



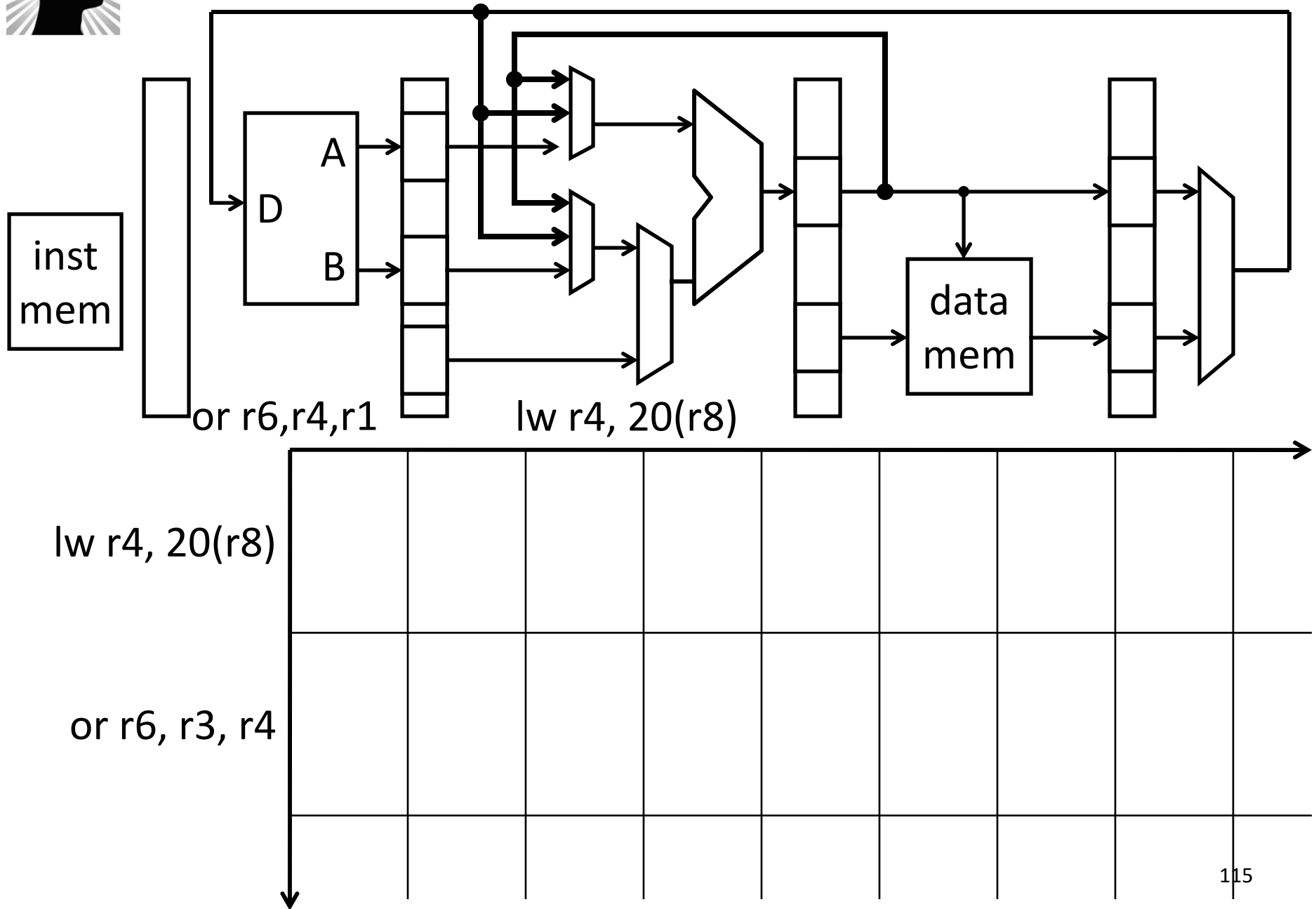
Data dependency after a load instruction:

- Value not available until ***after*** the M stage  
→ Next instruction cannot proceed if dependent

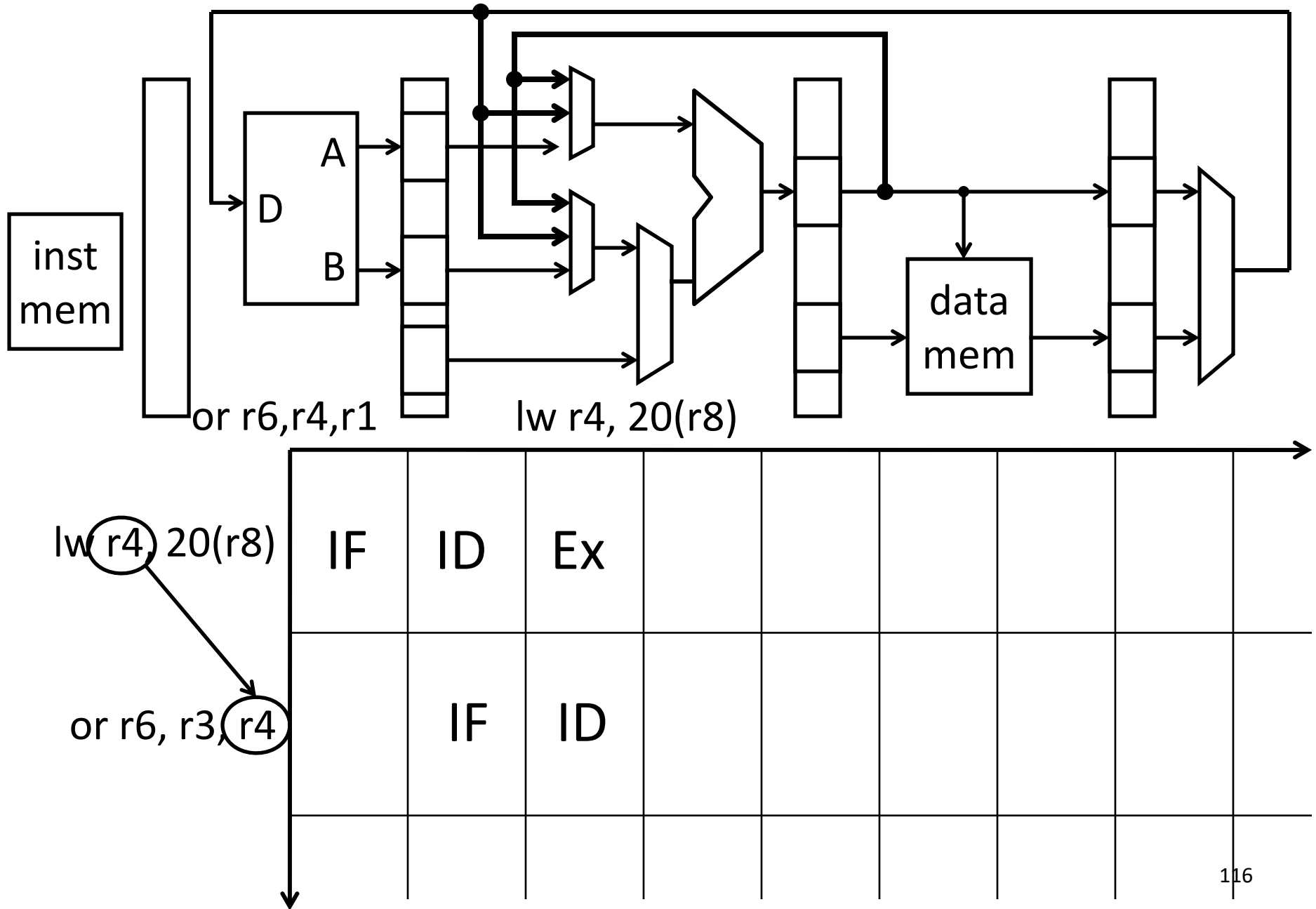
***THE KILLER HAZARD***



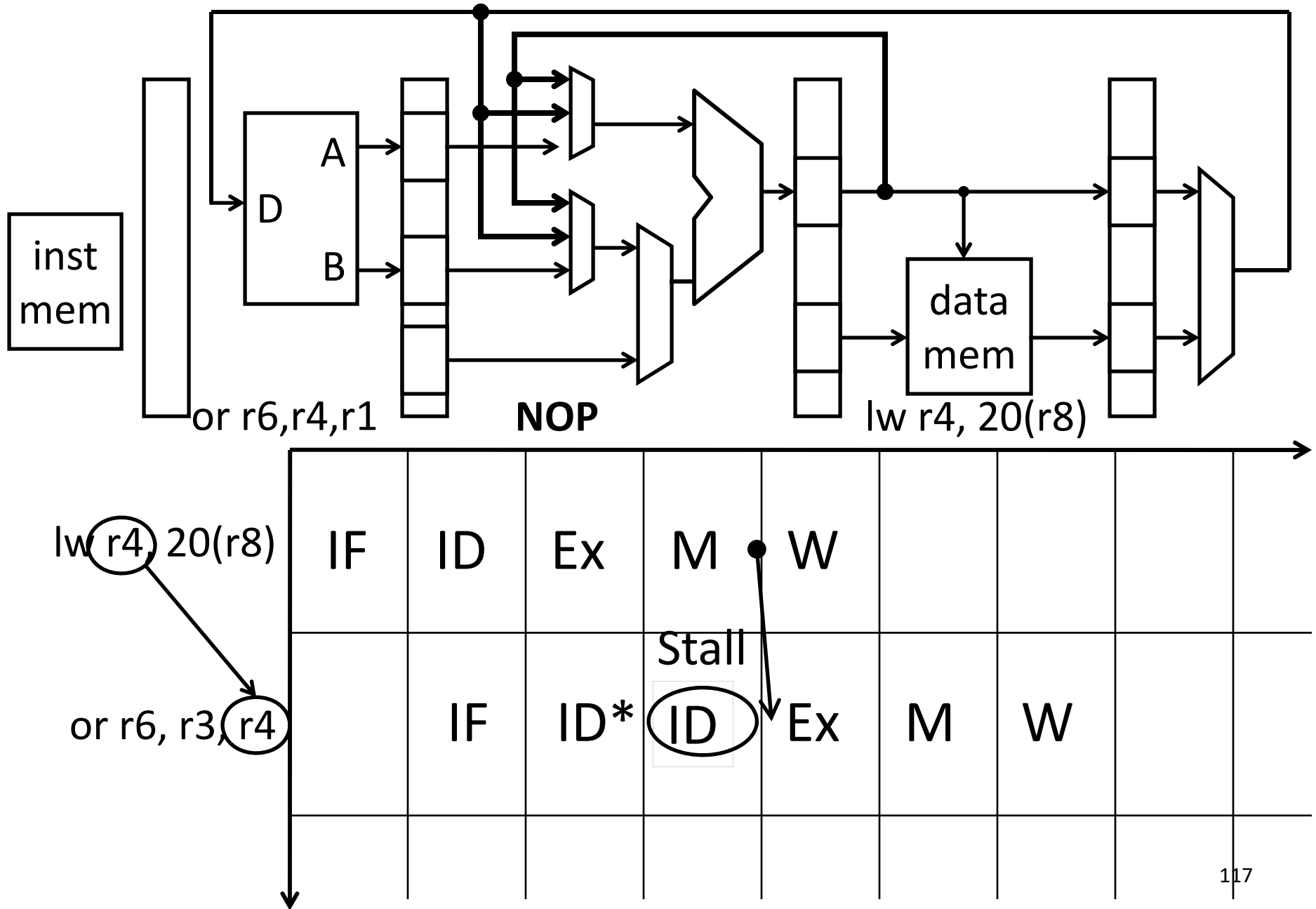
# Load-Use Stall



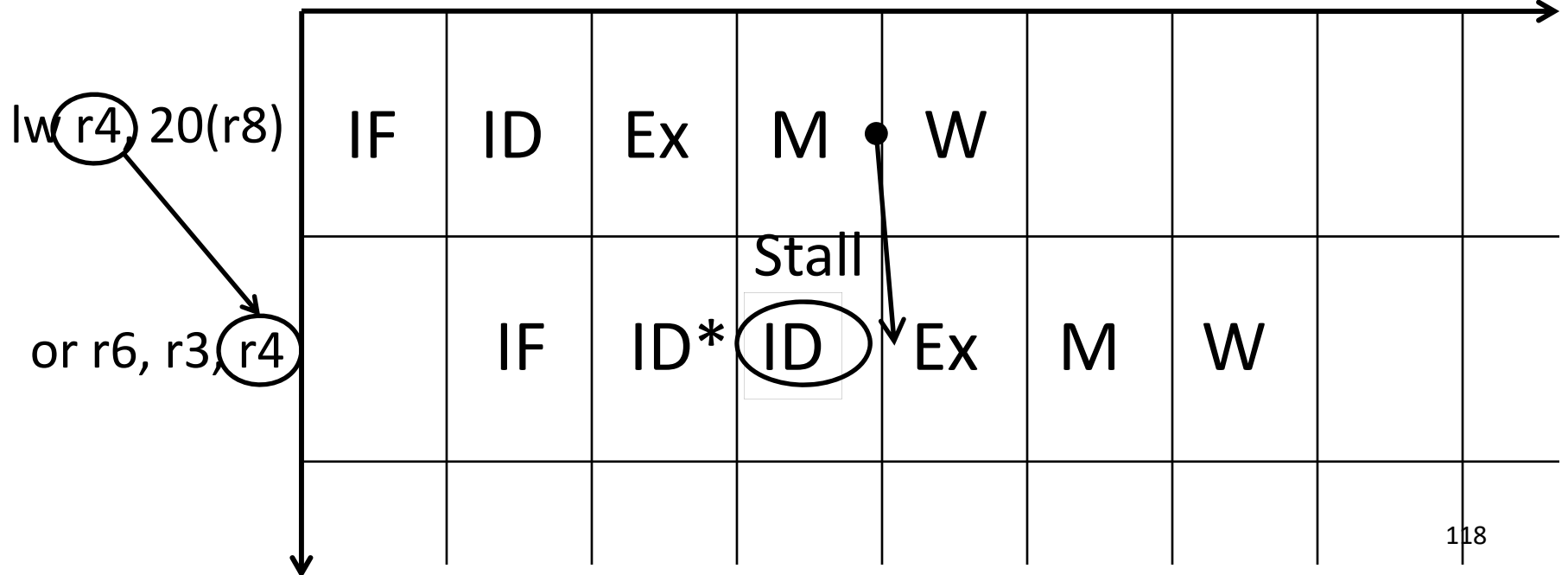
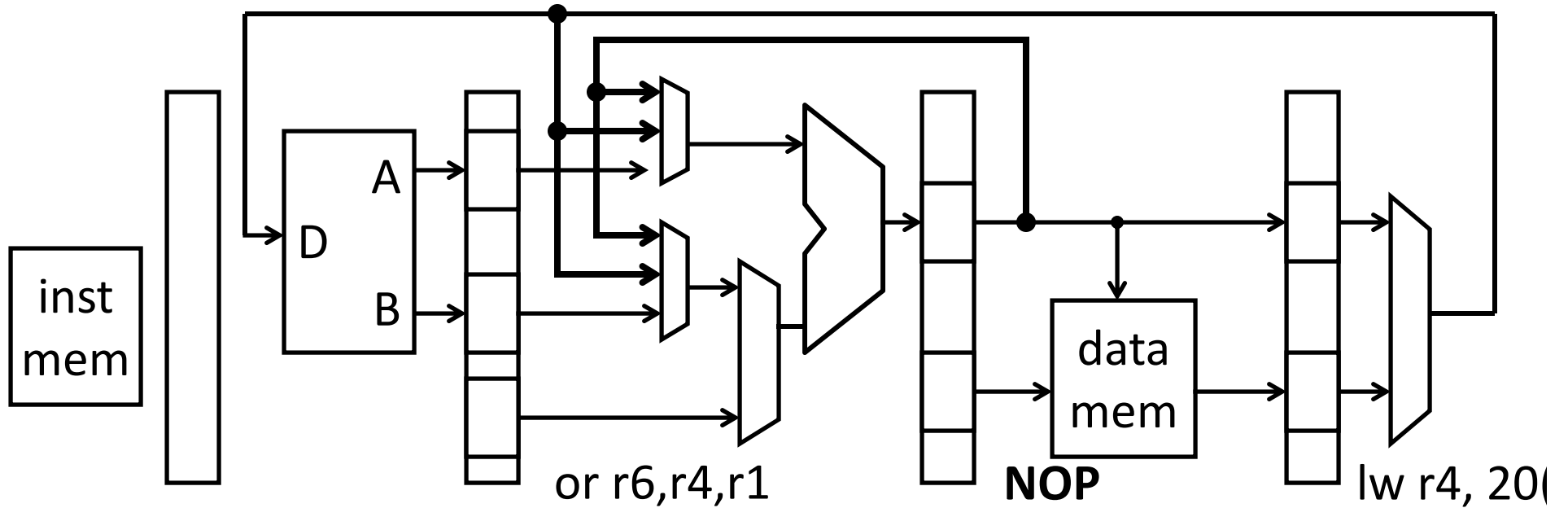
# Load-Use Stall (1)



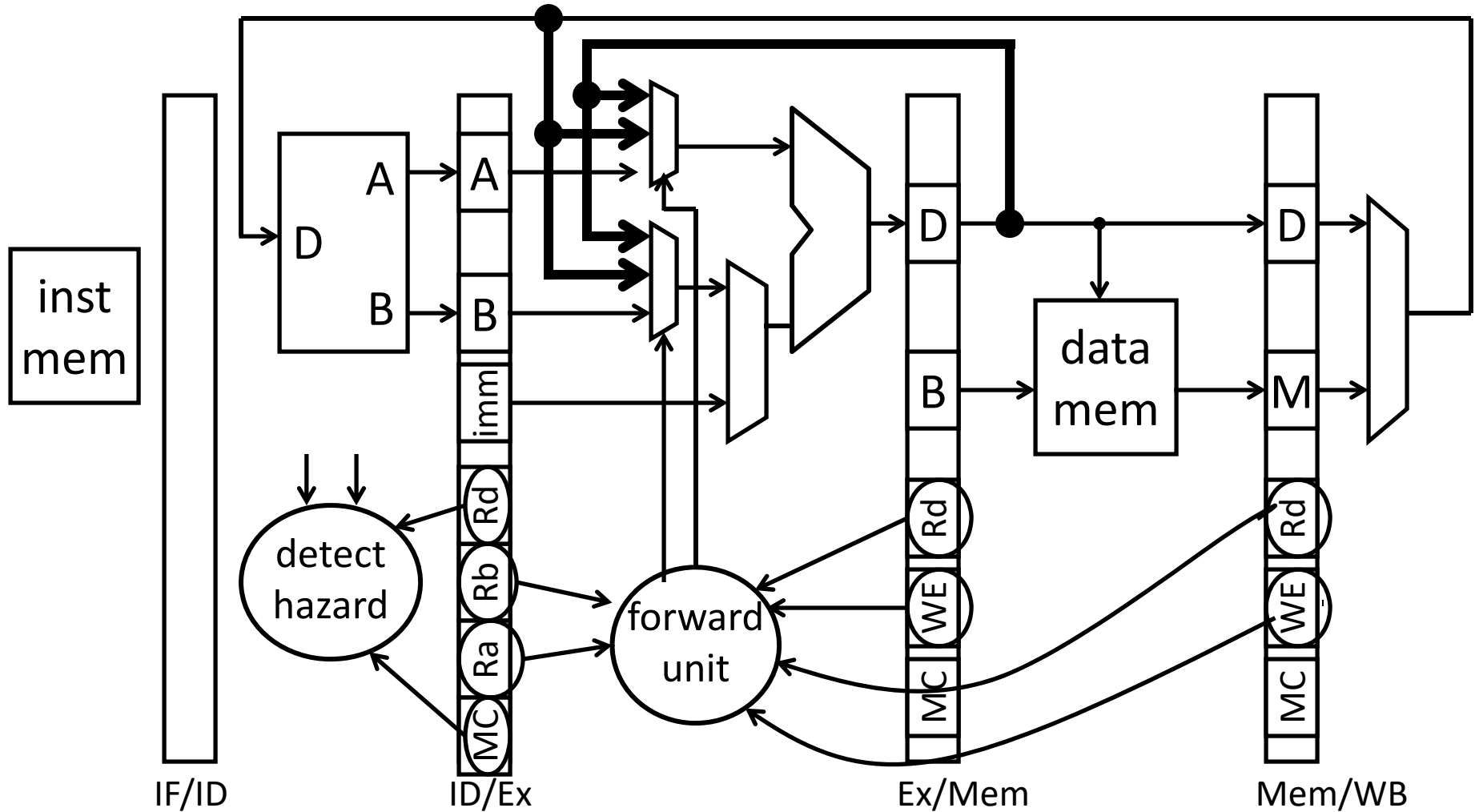
# Load-Use Stall (2)



# Load-Use Stall (3)

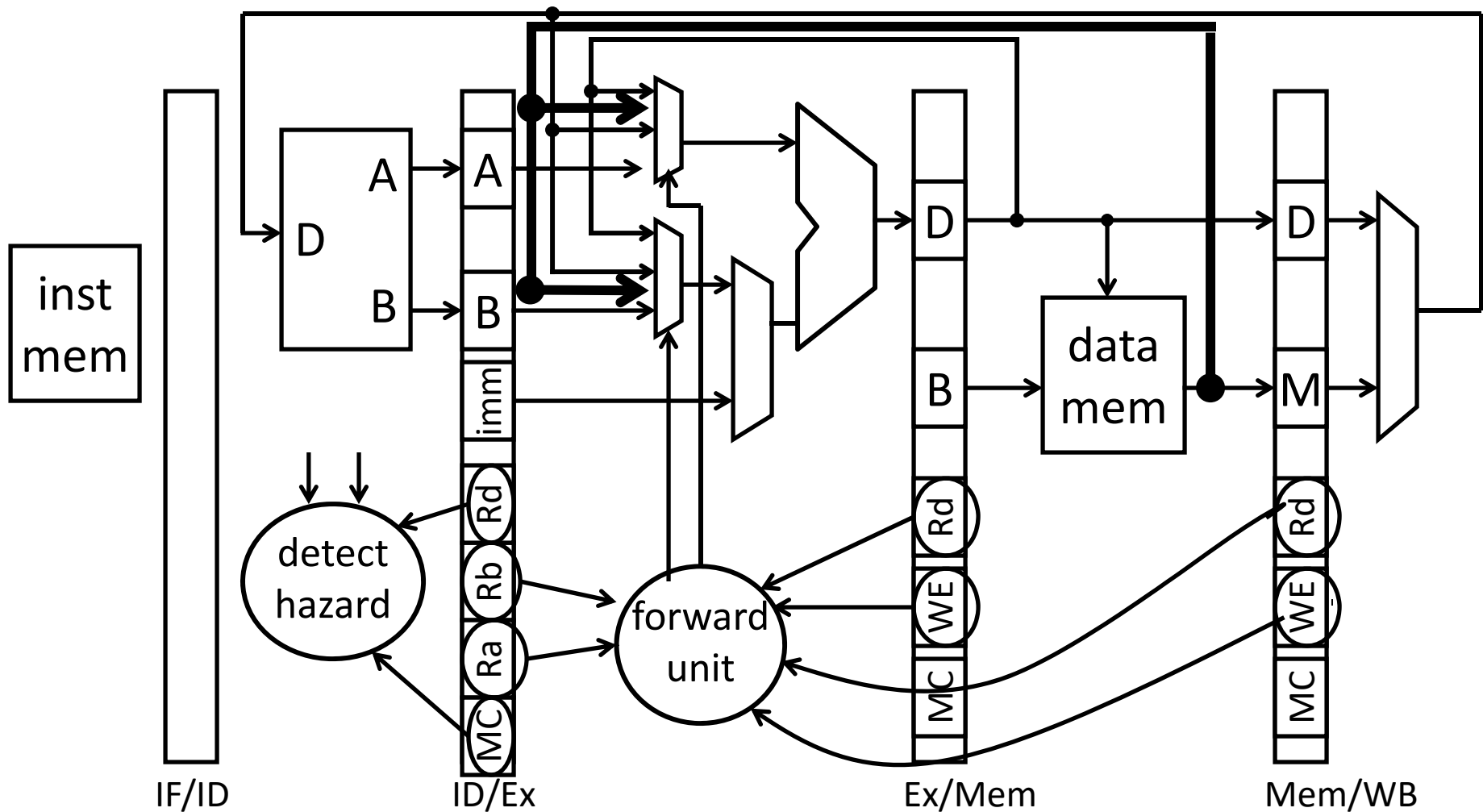


# Load-Use Detection



$$\text{Stall} = \text{If}(\text{ID/Ex.MemRead} \ \&\& \\
\text{IF/ID.Ra} == \text{ID/Ex.Rd})$$

# Incorrectly Resolving Load-Use Hazards



Most frequent 3410 **non-solution** to load-use hazards

**Why is this “solution” so so so so so so awful?**

# iClicker Question

Forwarding values directly from Memory to the Execute stage without storing them in a register first:

- A. Does not remove the need to stall.
- B. Adds one too many possible inputs to the ALU.
- C. Will cause the pipeline register to have the wrong value.
- D. Halves the frequency of the processor.
- E. Both A & D

# iClicker Question

Forwarding values directly from Memory to the Execute stage without storing them in a register first:

- A. Does not remove the need to stall.
- B. Adds one too many possible inputs to the ALU.
- C. Will cause the pipeline register to have the wrong value.
- D. Halves the frequency of the processor.
- E. Both A & D

# Resolving Load-Use Hazards

Two MIPS Solutions:

- MIPS 2000/3000: delay slot
  - ISA says results of loads are not available until one cycle later
  - Assembler inserts nop, or reorders to fill delay slot
- MIPS 4000 onwards: stall
  - But really, programmer/compiler reorders to avoid stalling in the load delay slot

# 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.

# Quiz

Find all hazards, and say how they are resolved:

add r3, r1, r2

nand r5, r3, r4

add r2, r6, r3

lw r6, 24(r3)

sw r6, 12(r2)

# Quiz

Find all hazards, and say how they are resolved:

Diagram illustrating five MIPS instructions with data dependencies highlighted by arrows and circled register names:

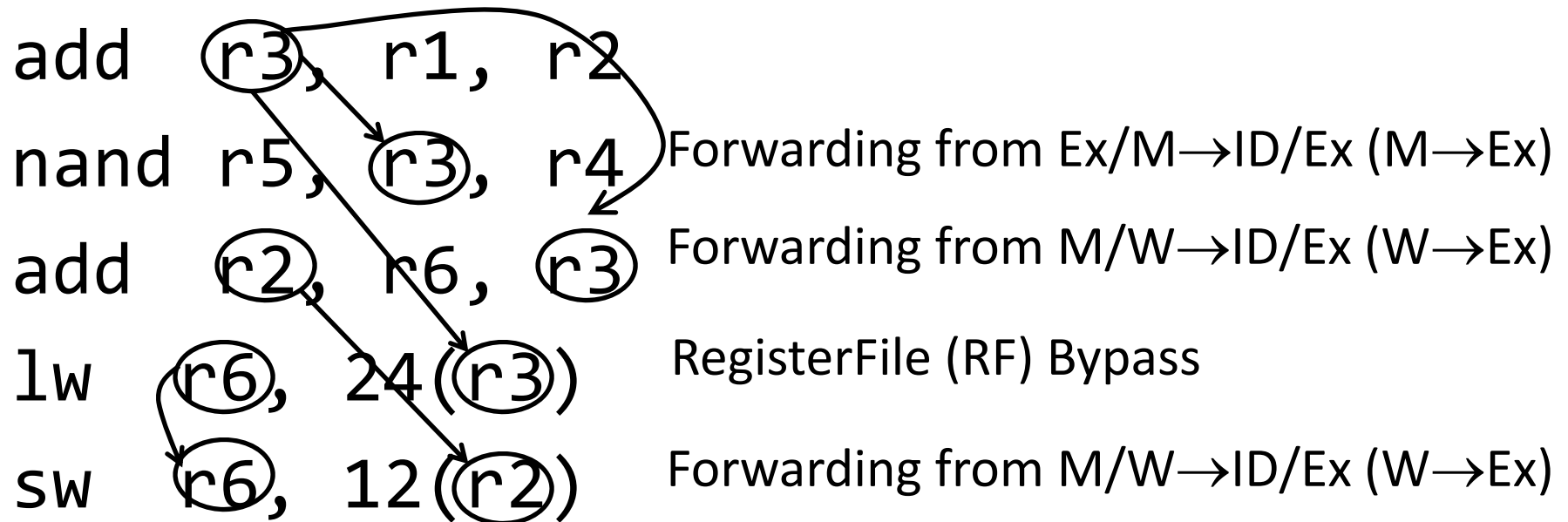
- add r3, r1, r2
- nand r5, r3, r4
- add r2, r6, r3
- lw r6, 24(r3)
- sw r6, 12(r2)

Arrows indicate dependencies: from r3 in the first instruction to r3 in the second; from r3 in the second instruction to r3 in the third; from r2 in the third instruction to r2 in the fifth; from r6 in the third instruction to r6 in the fourth; from r6 in the fourth instruction to r6 in the fifth; and from r3 in the first instruction to r2 in the fifth (via a curved arrow).

5 Hazards

# Quiz

Find all hazards, and say how they are resolved:



***Stall***

+ Forwarding from M/W→ID/Ex (W→Ex)

5 Hazards

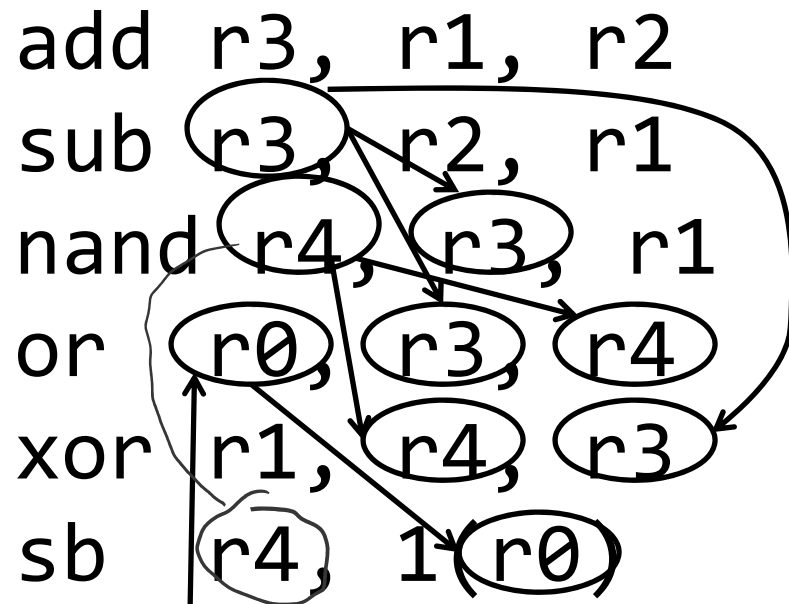
# Quiz

Find all hazards, and say how they are resolved:

```
add r3, r1, r2
sub r3, r2, r1
nand r4, r3, r1
or   r0, r3, r4
xor  r1, r4, r3
sb   r4, 1(r0)
```

# Quiz 2

Find all hazards, and say how they are resolved:



Hours and hours of debugging!

# Data Hazard Recap

## Delay Slot(s)

- Modify ISA to match implementation

## Stall

- Pause current and all subsequent instructions

## Forward/Bypass

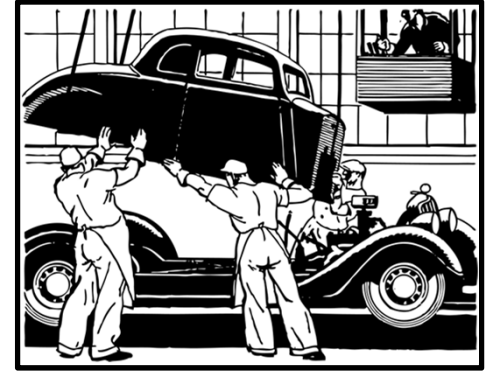
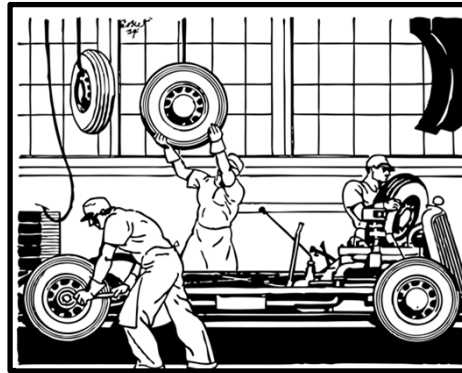
- Try to steal correct value from elsewhere in pipeline
- Otherwise, fall back to stalling or require a delay slot

## Tradeoffs?

# Agenda

## 5-stage Pipeline

- Implementation
- Working Example



## Hazards

- Structural
- Data Hazards
- Control Hazards

## A bit of Context

```
i = 0;
do {
    n += 2;
    i++;
} while(i < max)
i = 7;
n--;
```

$i \rightarrow r1$   
*Assume:*  
 $n \rightarrow r2$   
 $max \rightarrow r3$

x10	addiu r1, r0, 0	# i=0
x14 Loop:	addiu r2, r2, 2	# n += 2
x18	addiu r1, r1, 1	# i++
x1C	blt r1, r3, Loop	# i<max?
x20	addiu r1, r0, 7	# i = 7
x24	subi r2, r2, 1	# n--

# Control Hazards

## Control Hazards

- instructions are fetched in stage 1 (IF)
  - branch and jump decisions occur in stage 3 (EX)
- next PC not known until **2 *cycles* after** branch/jump

```
x1C    blt    r1, r3, Loop
x20    addiu  r1, r0, 7
x24    subi  r2, r2, 1
```

Branch **not** taken?

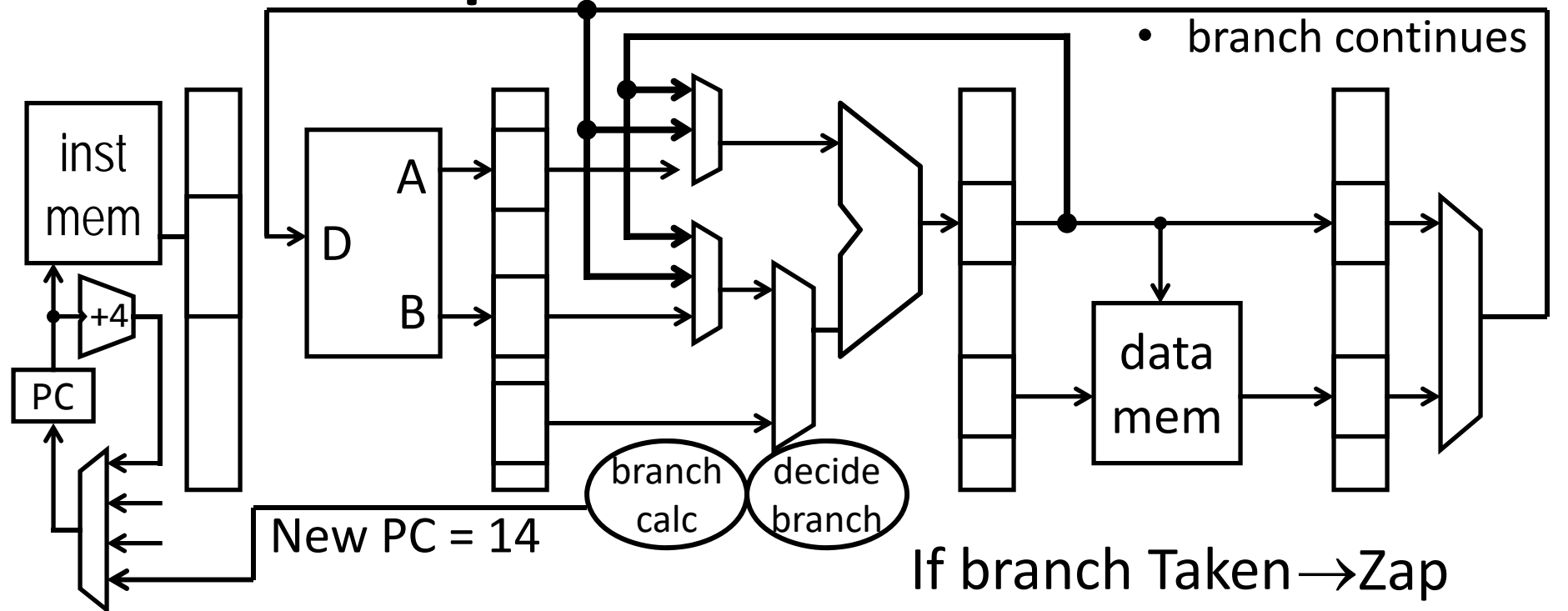
No Problem!

Branch taken?

Just fetched 2 addi's

→ **Zap & Flush**

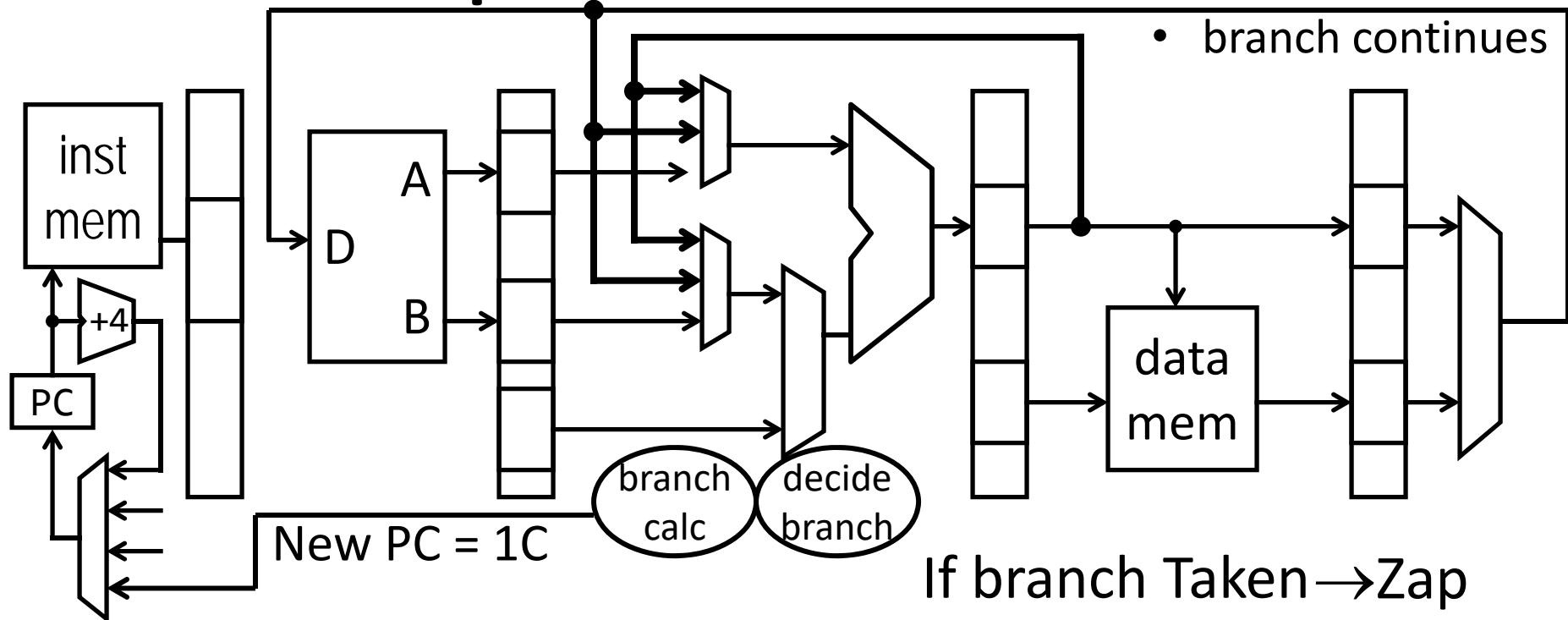
# Zap & Flush



1C blt r1,r3,L	IF	ID	Ex	M	W			
20 addiu r1,r0,7		IF	ID	NOP	NOP	NOP		
24 subi r2,r2,1			IF	NOP	NOP	NOP	NOP	
14 L:addi r2,r2,2				IF	ID	Ex	M	W

# Zap & Flush

- prevent PC update
- clear IF/ID latch
- branch continues



1C blt r1,r3,L	IF	ID	Ex	M	W			
20 addiu r1,r0,7		IF	ID	NOP	NOP	NOP		
24 subi r2,r2,1			IF	NOP	NOP	NOP	NOP	
14 L: addi r2,r2,2				IF	ID	Ex	M	W

*For every taken branch? OUCH!!!*

# Reducing the cost of control hazard

## 1. Delay Slot

- You MUST do this
- MIPS ISA: 1 insn after ctrl insn *always* executed
  - Whether branch taken or not

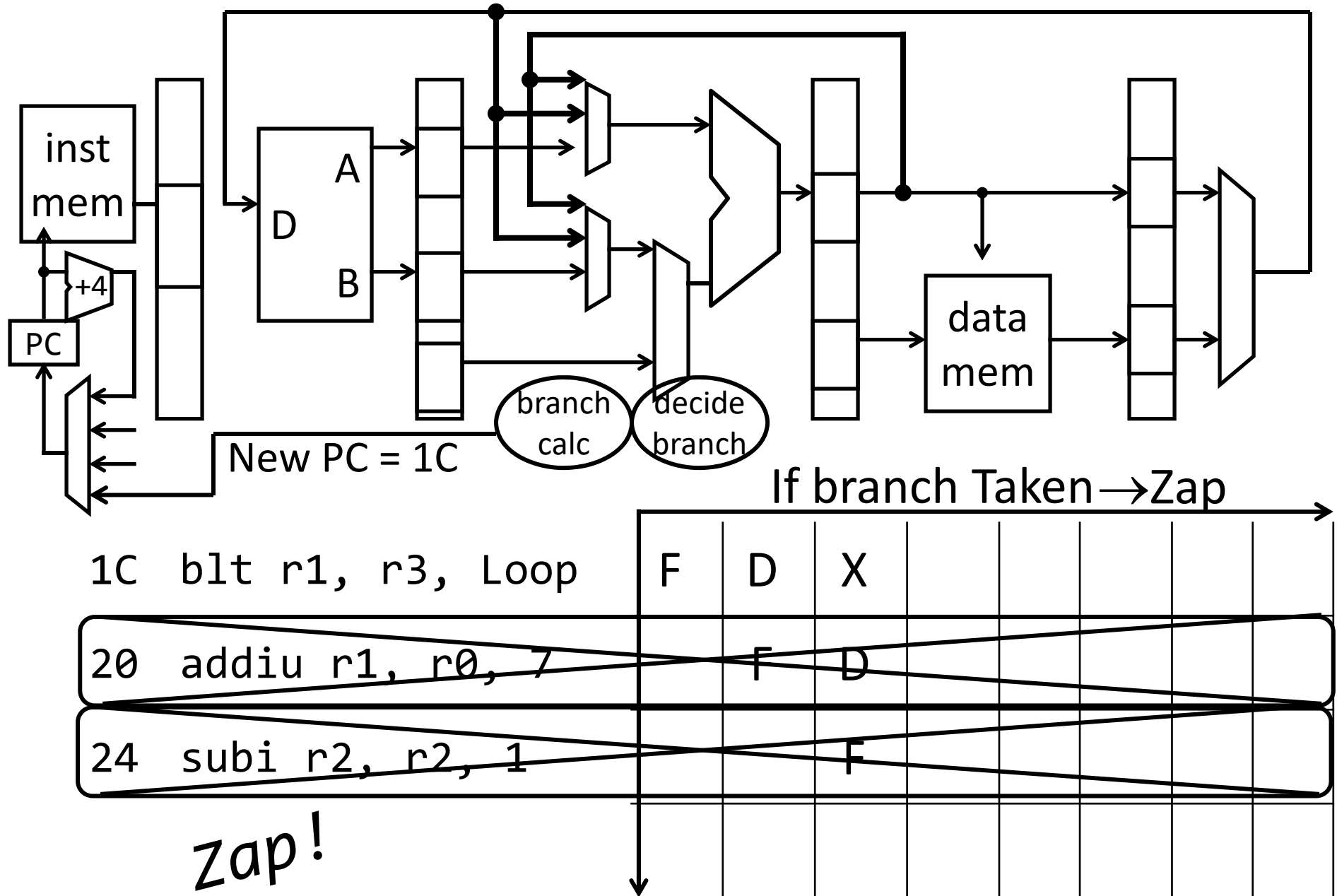
## 2. Resolve Branch at Decode

- Some groups do this for Project 3, your choice
- Move branch calc from EX to ID
- Alternative: just zap 2<sup>nd</sup> instruction when branch taken

## 3. Branch Prediction

- Not in 3410, but every processor worth *anything* does this (*no offense!*)

# Problem: Zapping 2 insns/branch



# `i = 0;`    **Solution #1: Delay Slot**

`do {`

`n += 2;`

`i++;`

`} while(i < max)`

`i = 7;`

`n--;`

x10            `addiu r1, r0, 0`        # `i=0`

x14 Loop: `addiu r2, r2, 2`        # `n += 2`

x18            `addiu r1, r1, 1`        # `i++`

x1C            `blt r1, r3, Loop`    # `i<max?`

x20            `nop`

x24            `addiu r1, r0, 7`        # `i = 7`

x28            `subi r2, r2, 1`        # `n++`

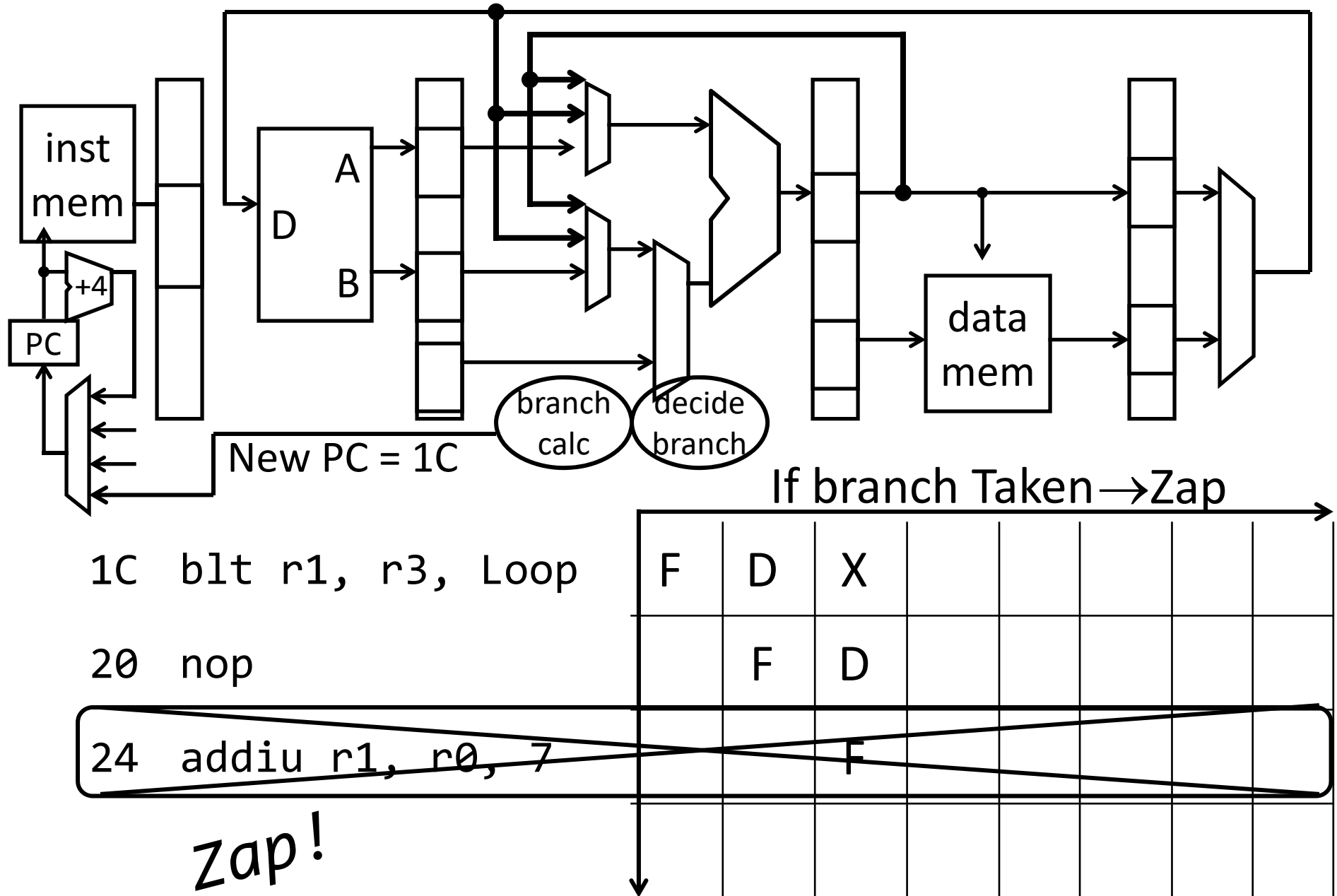
*i* → *r1*

*Assume:*

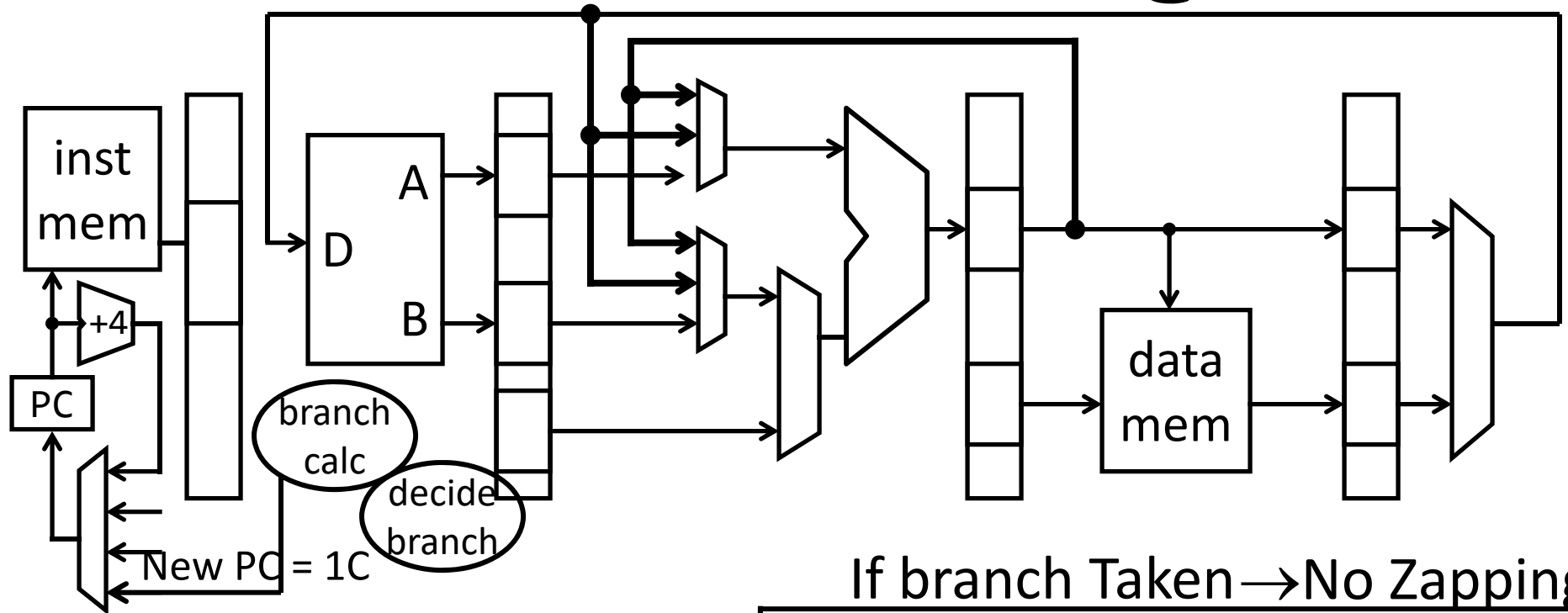
*n* → *r2*

*max* → *r3*

# Delay Slot in Action



# Soln #2: Resolve Branches @ Decode



1C blt r1, r3, Loop

20 nop

14 Loop: addiu r2, r2, 2

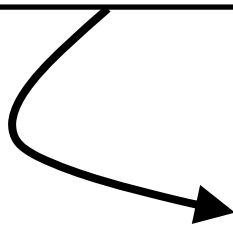
*No Zapping!*

If branch Taken → No Zapping

F	D	X					
	F	D					
		F					
							140

# Optimization: Fill the Delay Slot

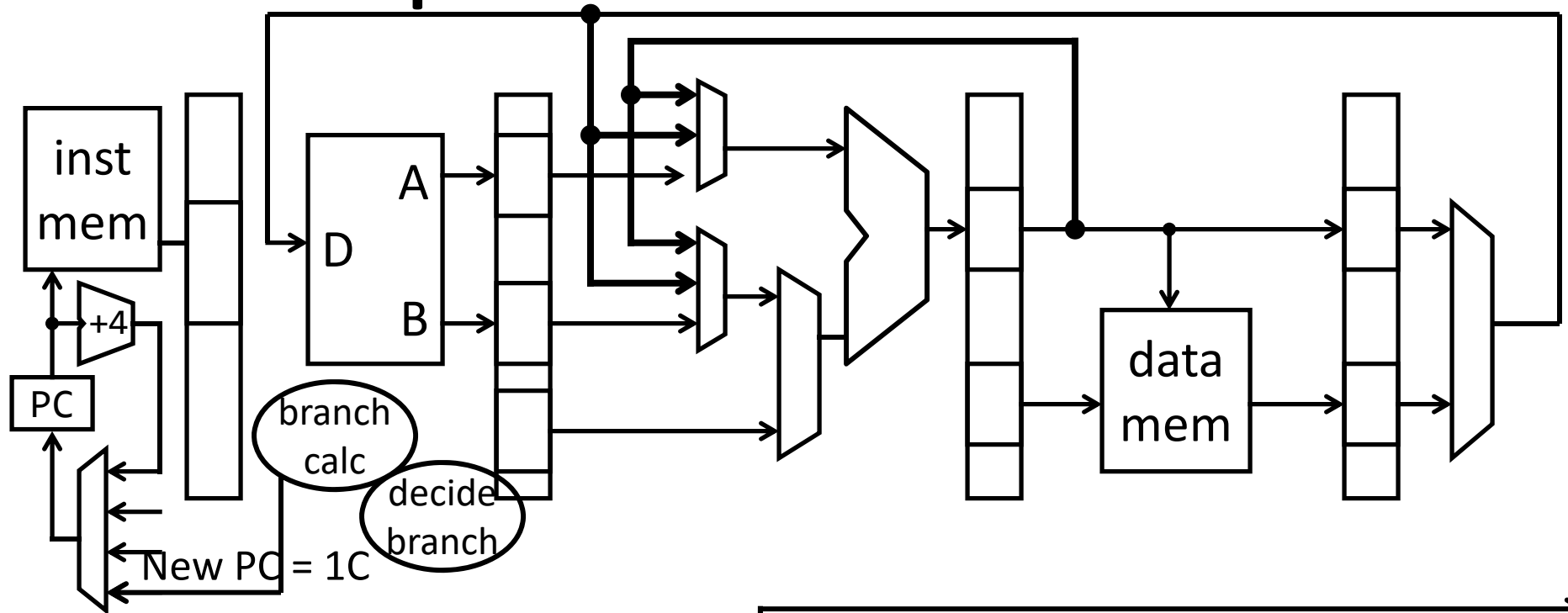
```
x10          addiu r1, r0, 0      # i=0
x14 Loop:    addiu r2, r2, 2      # n += 2
x18          addiu r1, r1, 1      # i++
x1C          blt  r1, r3, Loop    # i<max?
x20          nop
```



↓ *Compiler transforms code*

```
x10          addiu r1, r0, 0      # i=0
x14 Loop:    addiu r1, r1, 1      # i++
x18          blt  r1, r3, Loop    # i<max?
x1C          addiu r2, r2, 2      # n += 2
```

# Optimization In Action!



1C blt r1, r3, Loop

20 addi r2,r2,2

14 Loop:addi r1,r1,1

**No Nop or Zapping!**

Note: Insn in delay slot will *always* be executed whether branch take or not

F	D	X					
	F	D					
		F					

# Branch Prediction

Most processor support **Speculative Execution**

- *Guess* direction of the branch
  - Allow instructions to move through pipeline
  - Zap them later if guess turns out to be wrong
- *A must* for long pipelines

# Speculative Execution: Loops

Pipeline so far

- “Guess” (predict) that the branch will not be taken

We can do better!

- Make prediction based on last branch
- Predict “take branch” if last branch “taken”
- Or Predict “do not take branch” if last branch “not taken”
- Need one bit to keep track of last branch


# Speculative Execution: Loops

What is accuracy of branch predictor?

Wrong twice per loop!

Once on loop enter and exit


We can do better with 2 bits

While ( $r3 \neq 0$ ) {...  $r3--$ ;}  


Top: BEQZ r3, End

J Top

End:

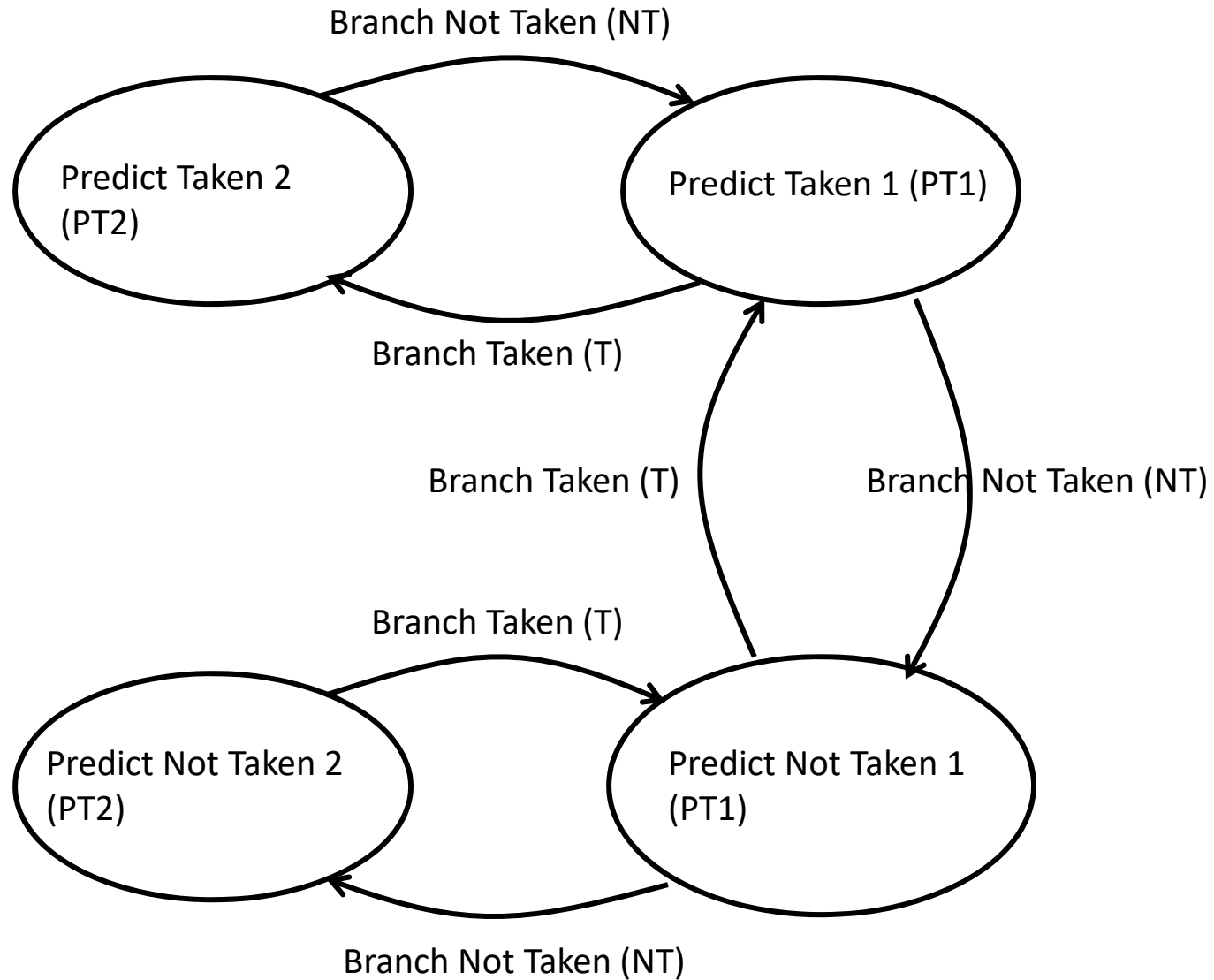
While ( $r3 \neq 0$ ) {...  $r3--$ ;}  


Top2: BEQZ r3, End2

J Top

End2:

# Speculative Execution: Branch Execution



# Summary

## Control hazards

- Is branch taken or not?
- Performance penalty: stall and flush

## Reduce cost of control hazards

- Move branch decision from Ex to ID
  - 2 nops to 1 nop
- Delay slot
  - Compiler puts useful work in delay slot. ISA level.
- Branch prediction
  - Correct. Great!
  - Wrong. Flush pipeline. Performance penalty

# Hazards Summary

Data hazards

Control hazards

Structural hazards

- resource contention
- so far: impossible because of ISA and pipeline design

# Hazards Summary

## Data hazards

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

## Control hazards

- branch instruction may change the PC in stage 3 (EX)
- next instructions have already started executing

## Structural hazards

- resource contention
- so far: impossible because of ISA and pipeline design

# Data Hazard Takeaways

Data hazards occur when an operand (register) depends on the result of a previous instruction that may not be computed yet. Pipelined processors need 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. Nops significantly decrease performance.

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

# Control Hazard Takeaways

Control hazards occur because the PC following a control instruction is not known until control instruction is executed. If branch is taken → need to zap instructions. 1 cycle performance penalty.

Delay Slots can potentially increase performance due to control hazards. The instruction in the delay slot will ***always*** be executed. Requires software (compiler) to make use of delay slot. Put nop in delay slot if not able to put useful instruction in delay slot.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. With a delay slot, this removes the need to flush instructions on taken branches.