Pipelining

Hakim Weatherspoon
CS 3410
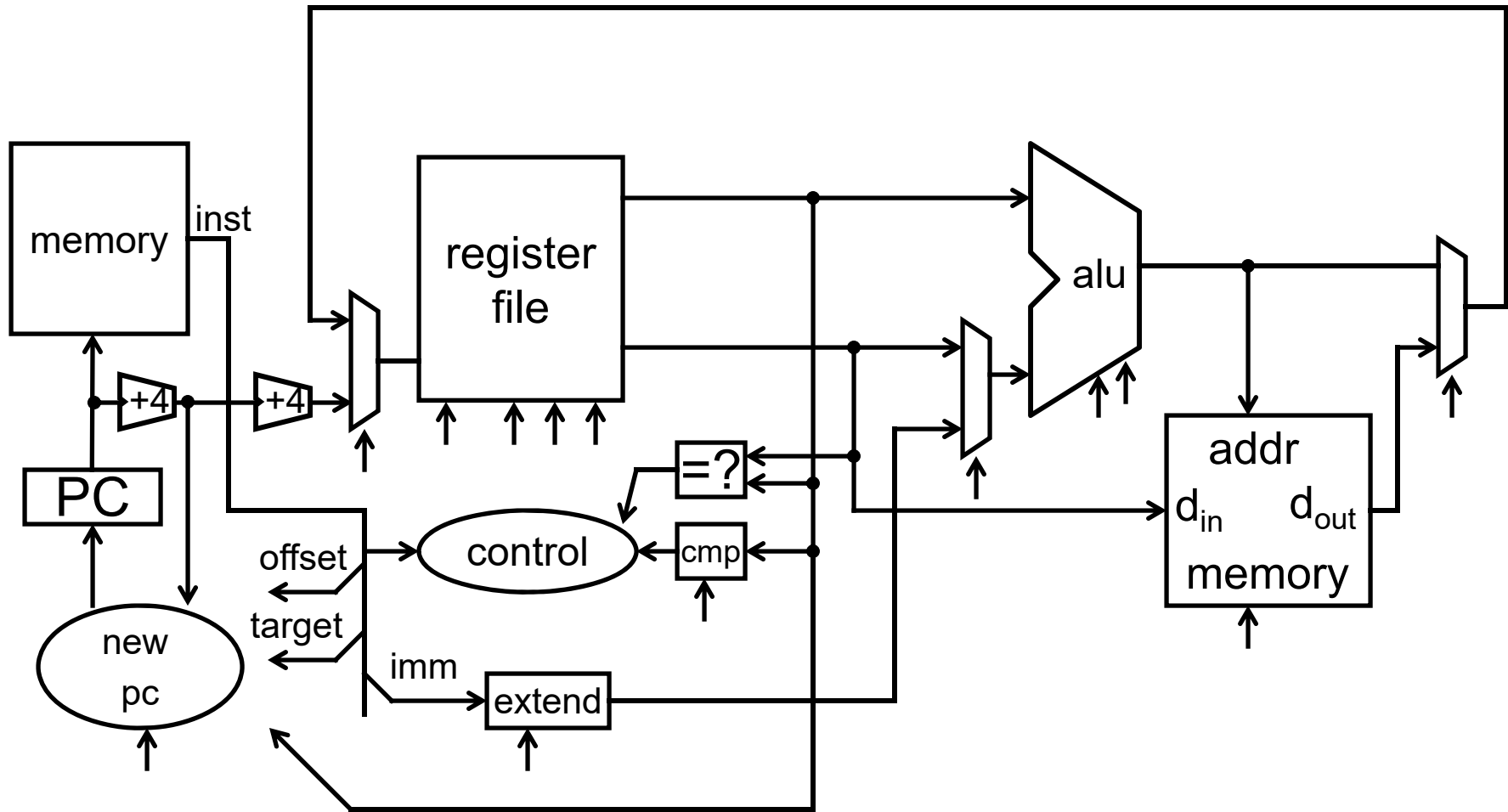Computer Science
Cornell University
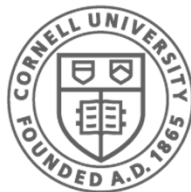
[Weatherspoon, Bala, Bracy, McKee, and Sirer]

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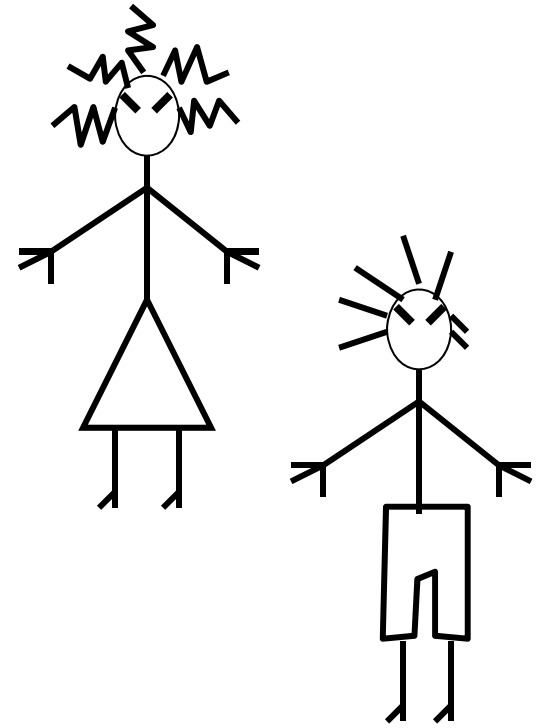 - want small CPI (close to 1) with high MIPS and short clock period (high clock frequency)

Improving Performance

- Parallelism

- Pipelining

- Both!

The Kids

Alice

Bob

They don't always get along…

The Bicycle

The Materials

Saw

Drill
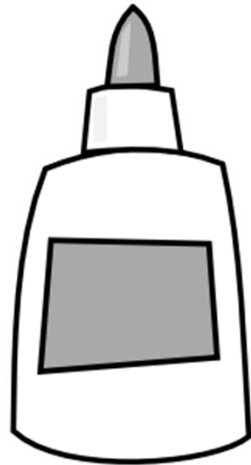
Glue

Paint

The Instructions

N pieces, each built following same sequence:

Saw → Drill → Glue → Paint
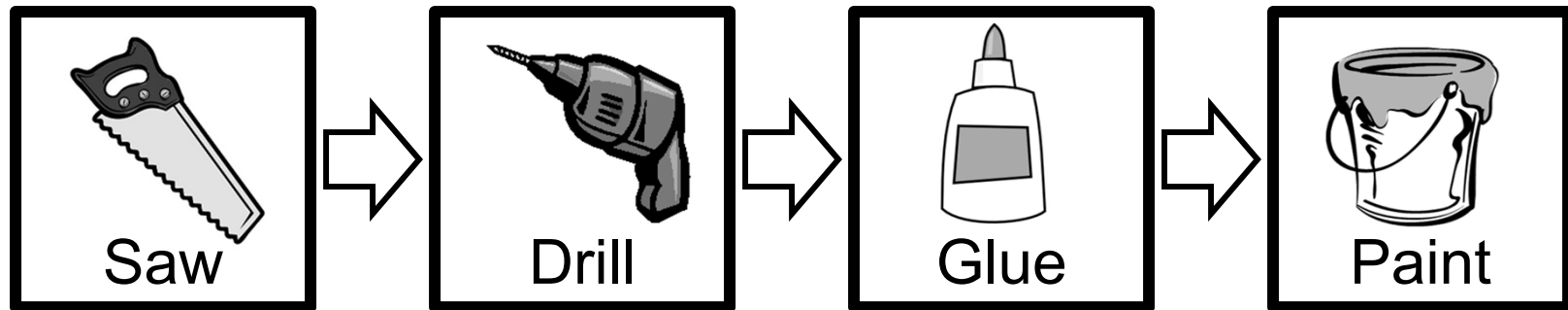
Design 1: Sequential Schedule
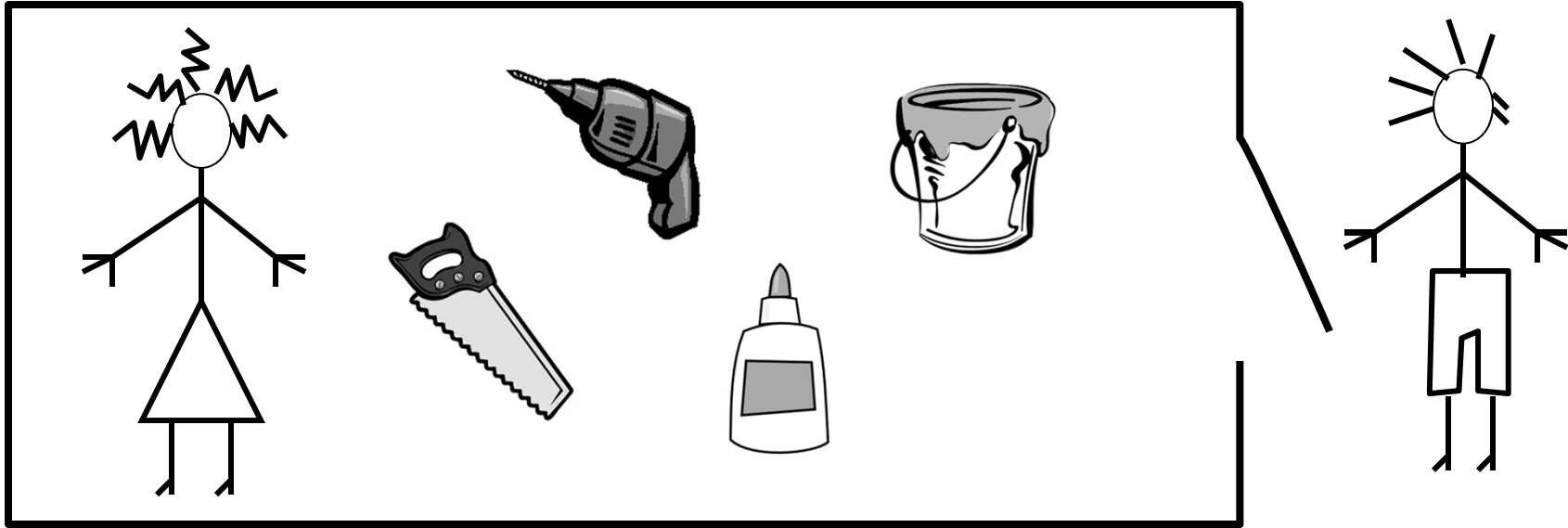
Alice owns the room

Bob can enter when Alice is finished

Repeat for remaining tasks

No possibility for conflicts

Sequential Performance

time

1   2   3   4   5   6   7   8 ...
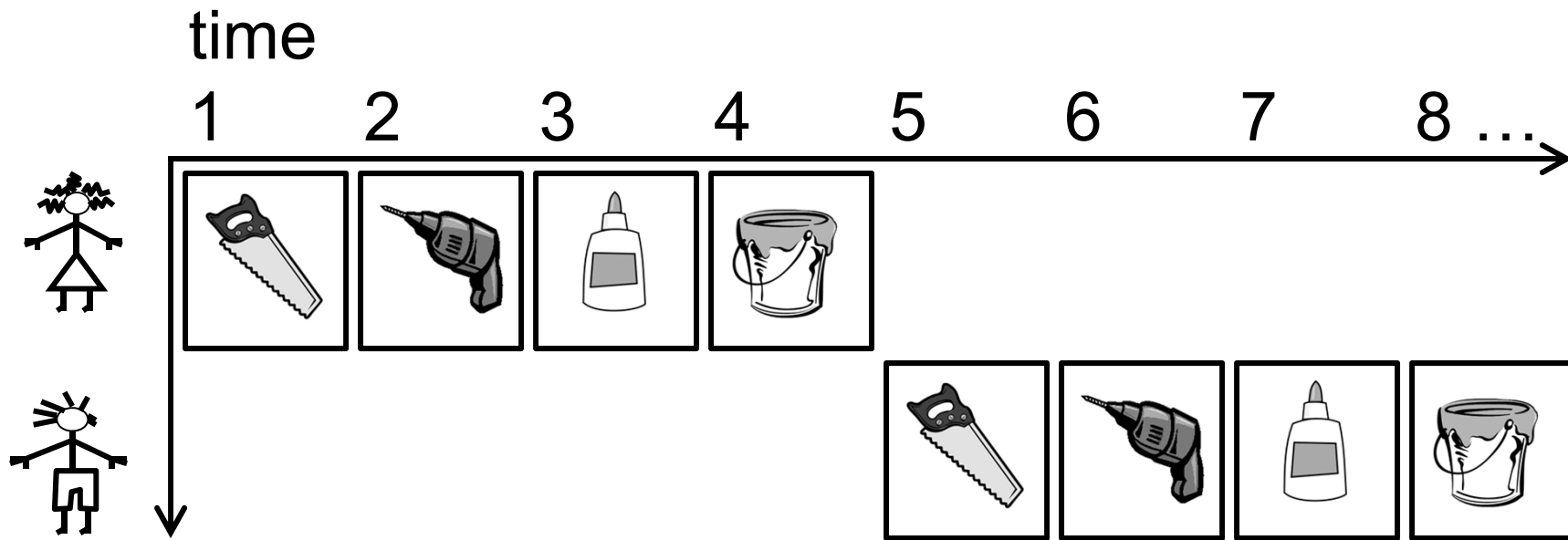
Latency: 4 hours/task
Throughput: 1 task/4 hrs
Concurrency: 1

CPI = 4

Design 2: Pipelined Design
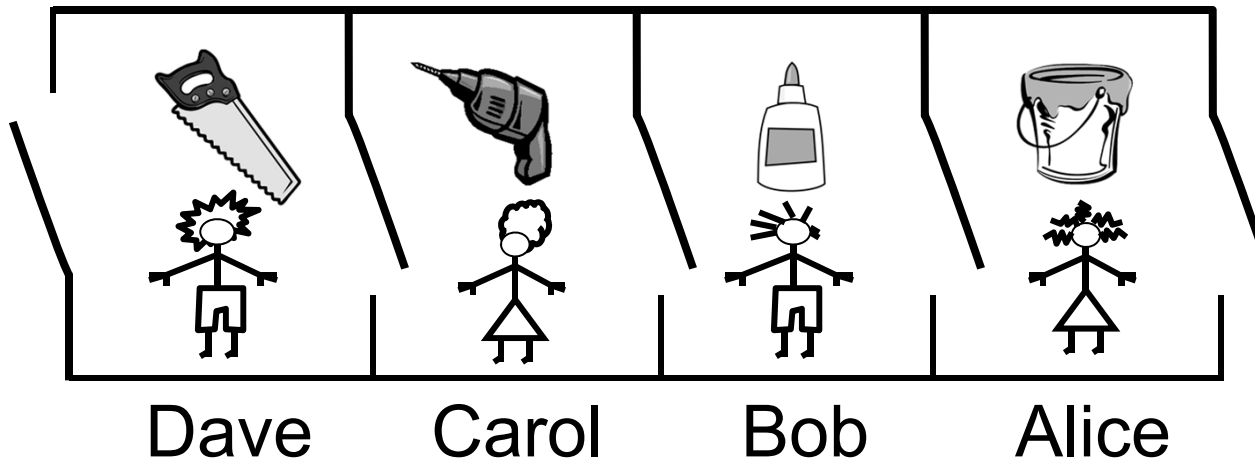
Partition room into *stages* of a *pipeline*

Dave   Carol   Bob   Alice

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*

Bob   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*

Dave Carol Bob 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*

Alice   Alice   Alice   Alice

One person owns a stage at a time

4 stages
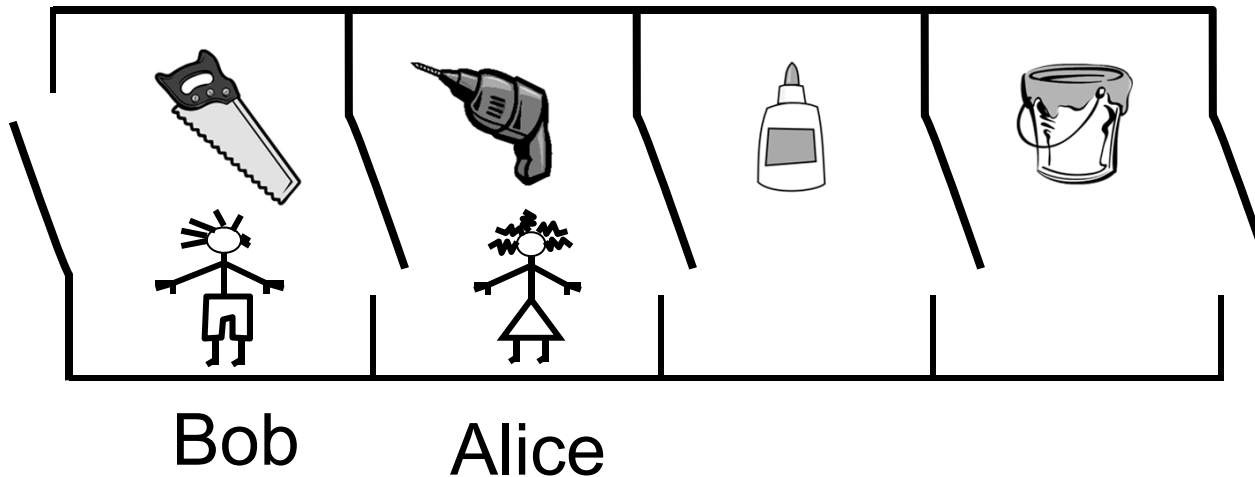
4 people working simultaneously

Everyone moves right in lockstep

It still takes all four stages for one job to complete

16

Pipelined Performance

time

Latency: 4 hrs/task
Throughput: 1 task/hr
Concurrency: 4   CPI = 1

Pipelined Performance

Time
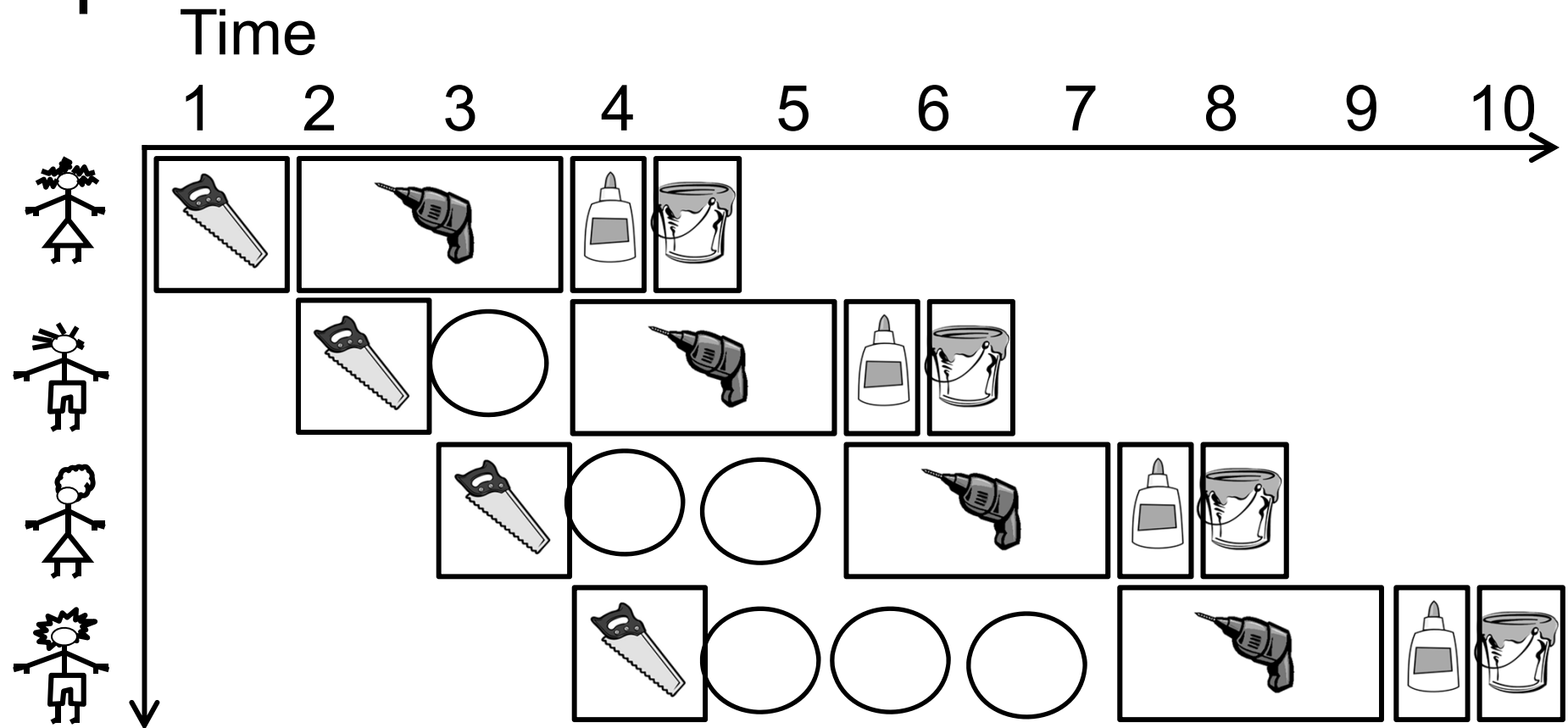
What if drilling takes twice as long, but gluing and paint take ½ as long?

Latency:

Throughput: CPI =

Pipelined Performance

Time

1   2   3   4   5   6   7   8   9   10

Done: 4 cycles

Done: 6 cycles

Done: 8 cycles
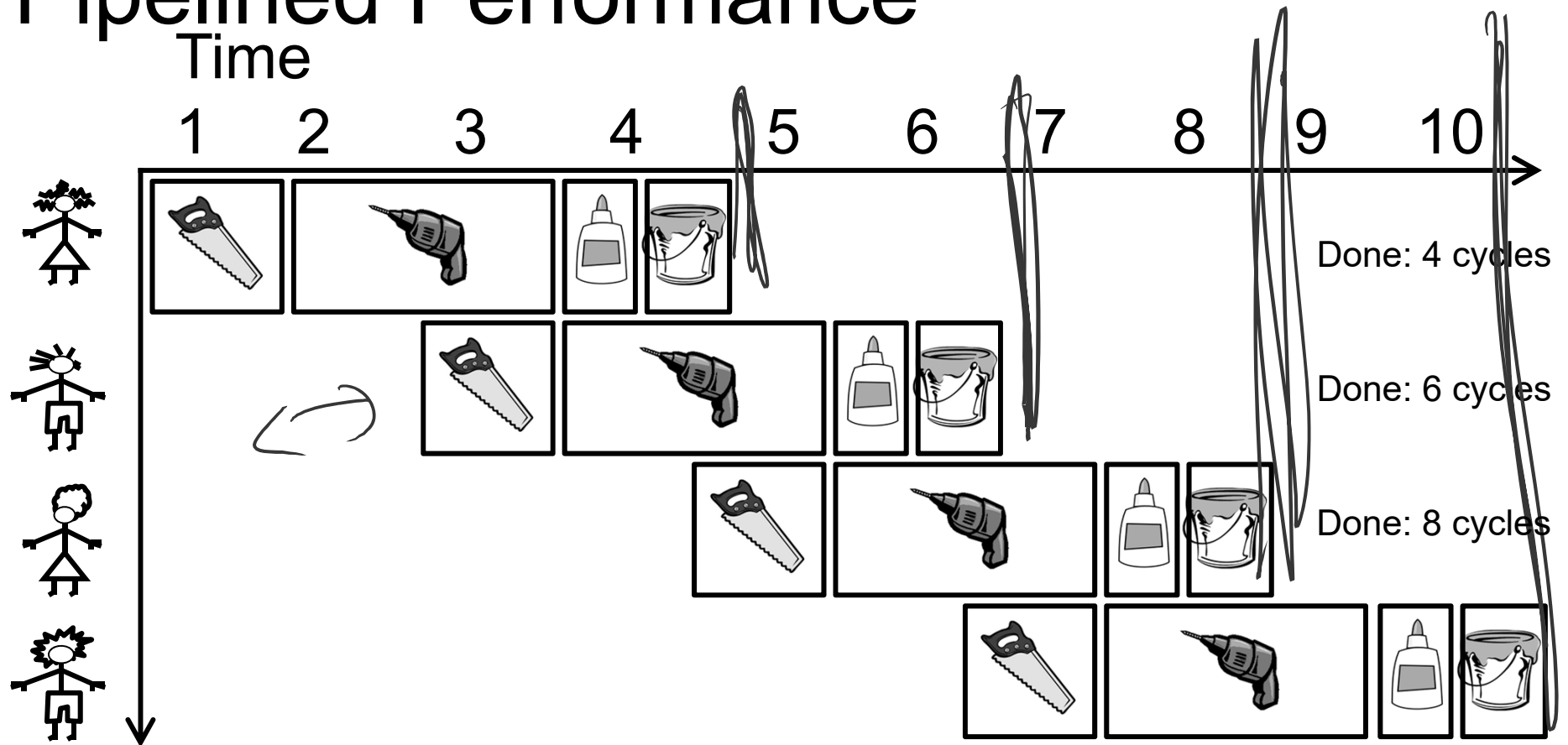
What if drilling takes twice as long, but gluing and paint take ½ as l

Latency: 4 cycles/task

Throughput: 1 task/2 cycles   CPI = 2

19

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
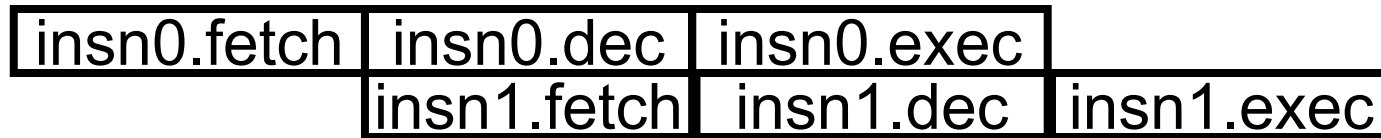
Single-cycle

insn0.fetch, dec, exec

insn1.fetch, dec, exec

Pipelined

insn0.fetch	insn0.dec	insn0.exec

insn1.fetch	insn1.dec	insn1.exec

Agenda

- 5-stage Pipeline
- Implementation
- Working Example
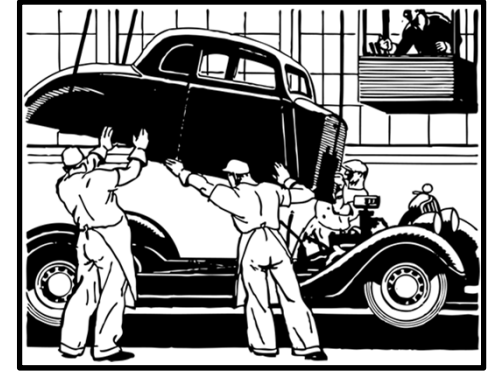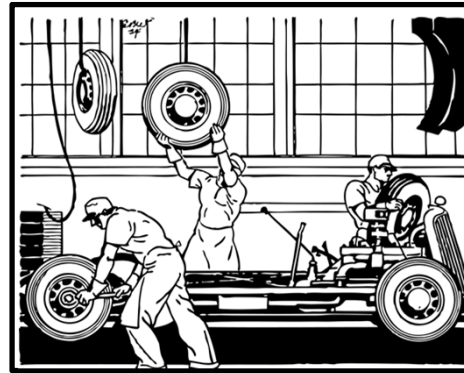
Hazards
- Structural
- Data Hazards
- Control Hazards
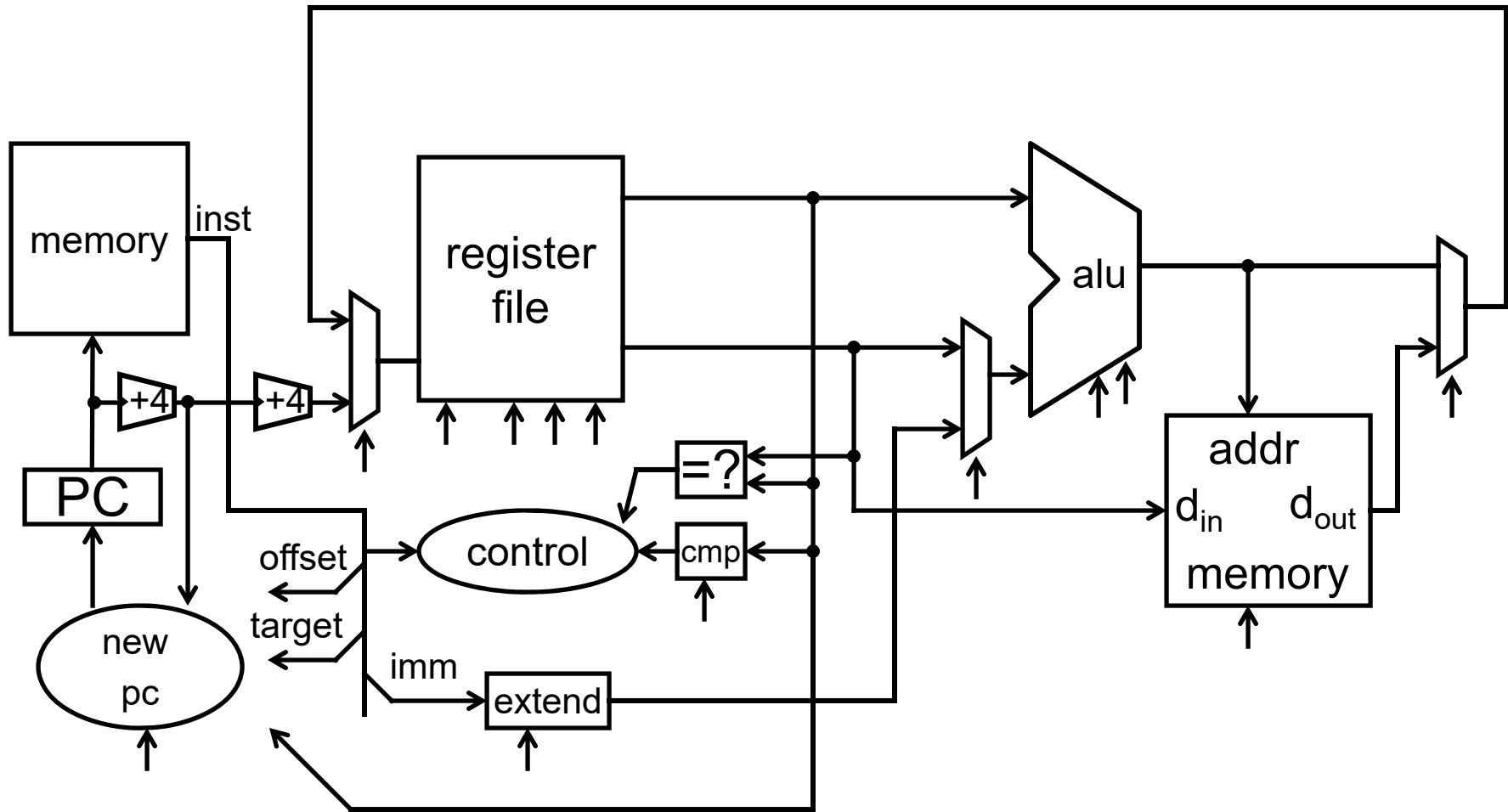
Review: Single Cycle Processor

Pipelined Processor

memory | inst

register file

alu

addr
d_{in} d_{out}
memory

+4

PC

control

new
pc

imm

extend

compute
jump/branch
targets

Fetch | Decode | Execute | Memory | WB

25

Pipelined Processor

memory

+4

PC

new pc

Instruction Fetch

inst

register file

control

extend

Instruction Decode

A

B

imm

ctrl

alu

compute jump/branch targets

Execute

D

B

ctrl

addr

d_{in} d_{out}

memory

Memory

D

M

ctrl

Write-Back

IF/ID   ID/EX   EX/MEM   MEM/WB

Time Graphs

Cycle   1   2   3   4   5   6   7   8   9

	1	2	3	4	5	6	7	8	9
add	IF	ID	EX	MEM	WB				
nand		IF	ID	EX	MEM	WB			
lw			IF	ID	EX	MEM	WB		
add				IF	ID	EX	MEM	WB	
sw					IF	ID	EX	MEM	WB

Latency: 5 cycles
Throughput: 1 insn/cycle   CPI = 1
Concurrency: 5

27

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

IF/ID   ID/EX   EX/MEM   MEM/WB

29
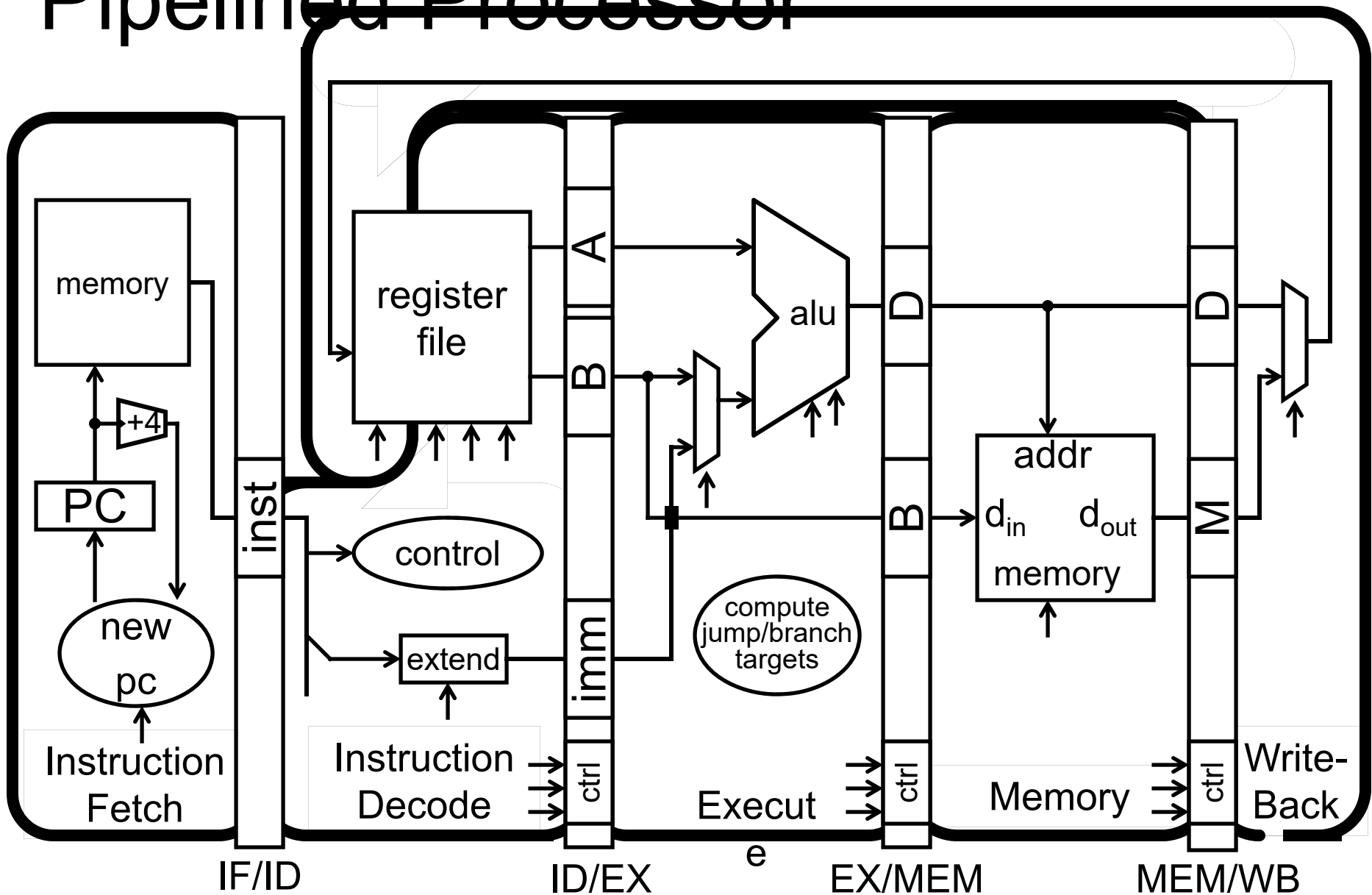
Pipeline Stages

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

addr   mc

+4

PC

new
pc

- PC+4
- pc-rel (PC-relative); e.g. JAL, BEQ, BNE
- pc-reg (PC registers); e.g. JALR

Instruction Fetch (IF)

instruction
memory

addr   mc

+4

00 = read word

PC

inst

PC+4

Rest of pipeline

pc-reg

pc-rel

pc-sel

IF/ID

33
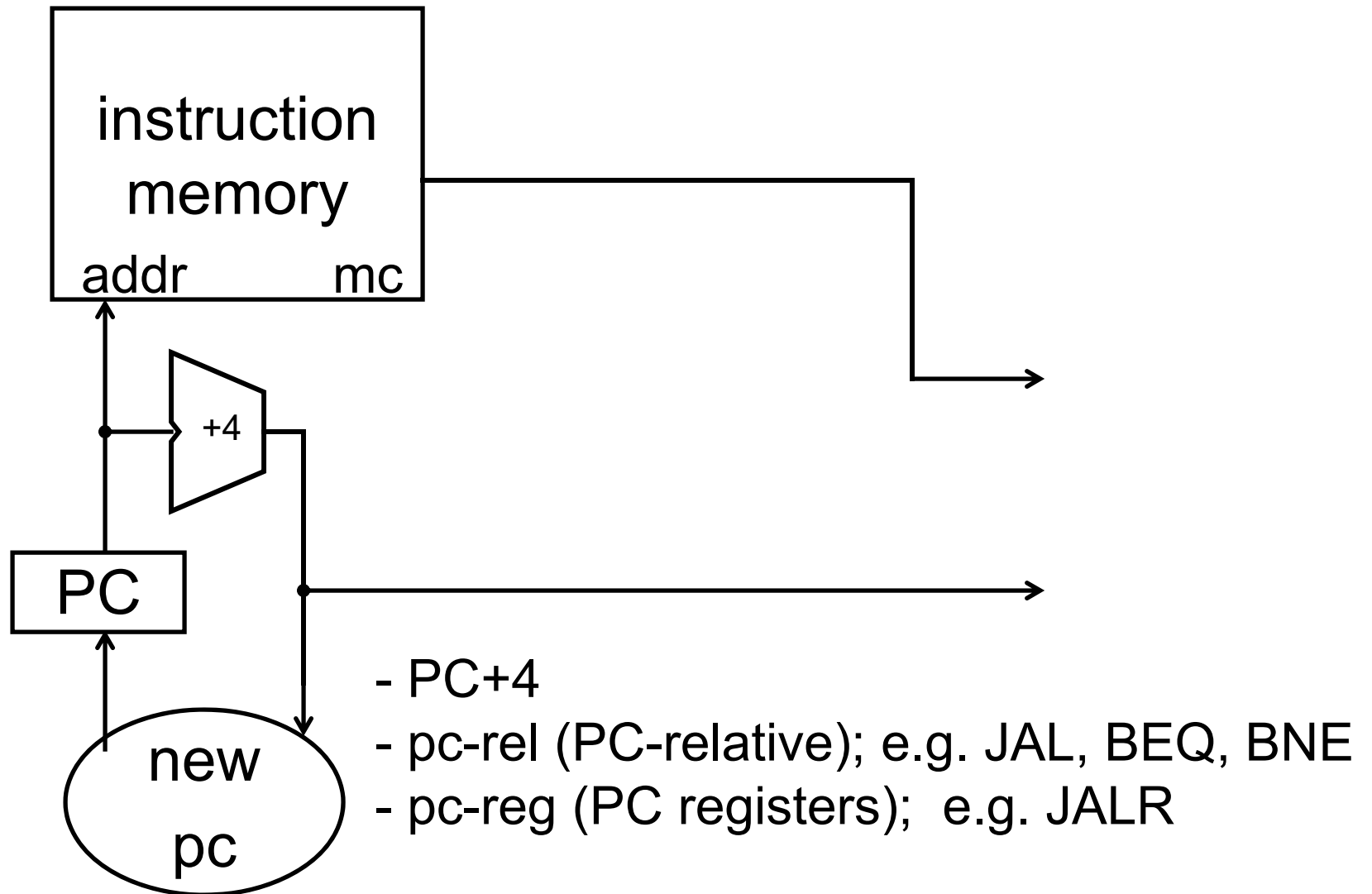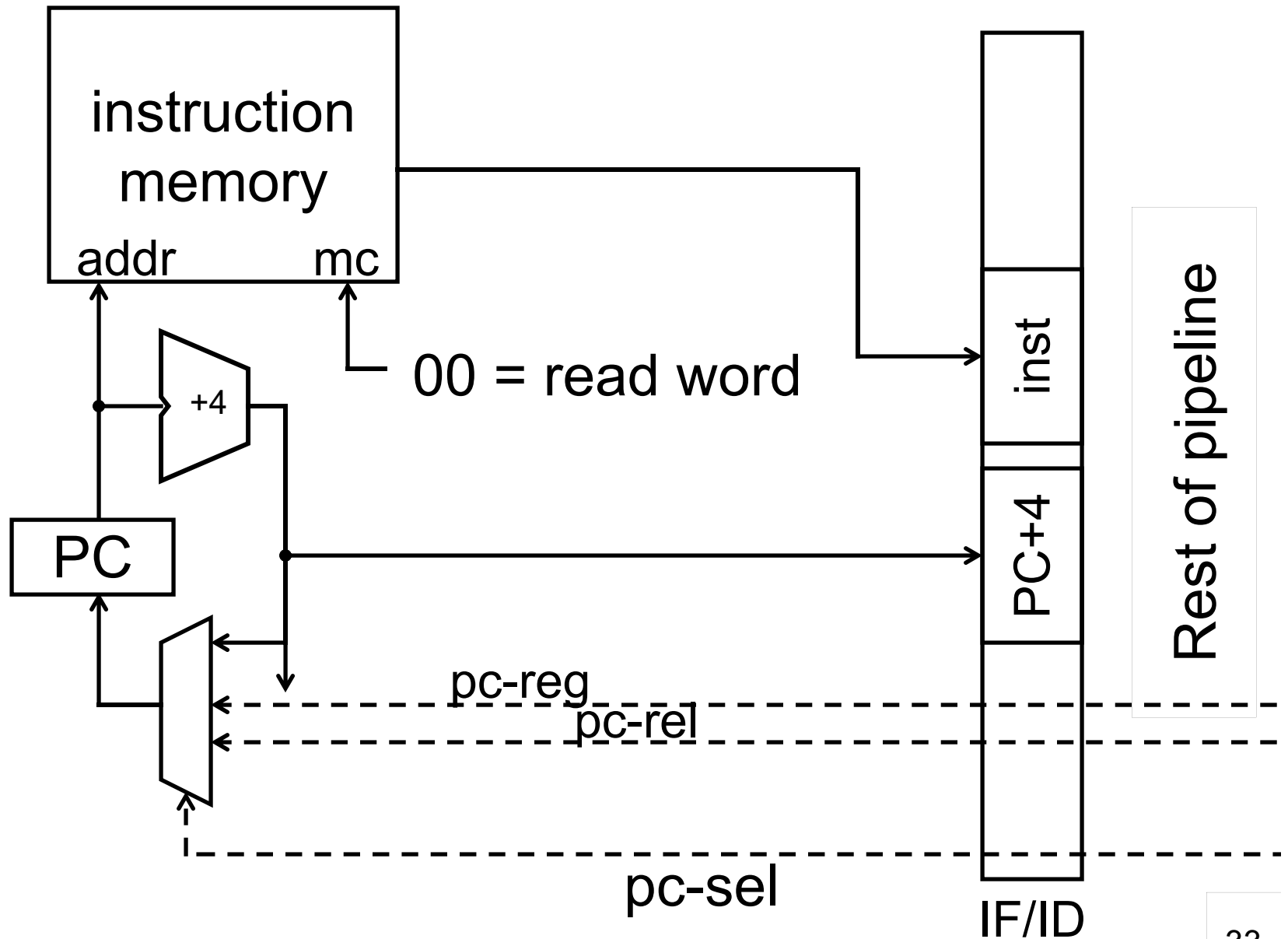
Decode

- 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

result

dest

Stage 1: Instruction Fetch

IF/ID

inst

PC+4

decode

WE
Rd
D

register
file

Ra Rb

A

B

extend

ID/EX

A

B

imm

PC+4

ctrl

Rest of pipeline

35
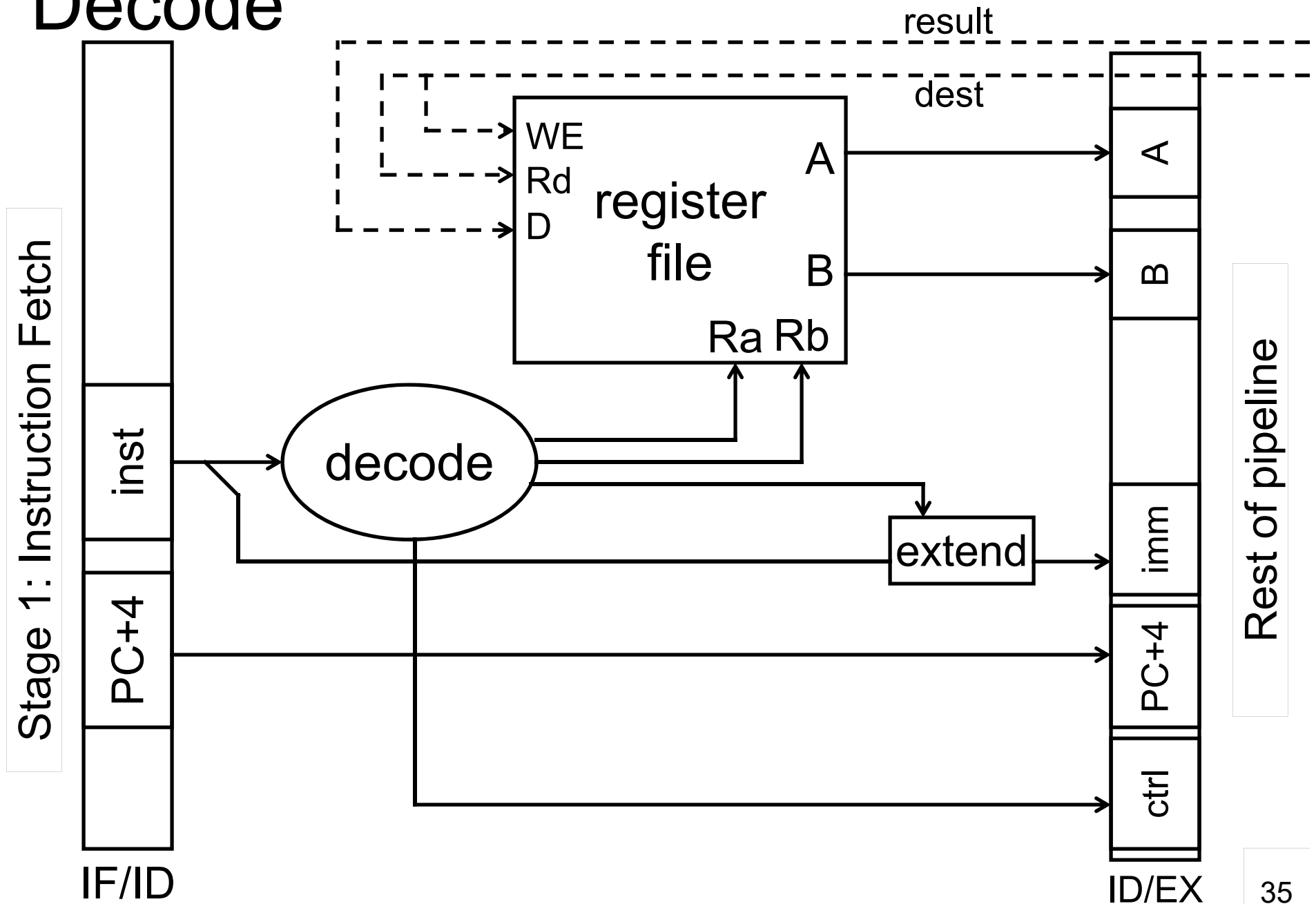
Execute (EX)

- 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)
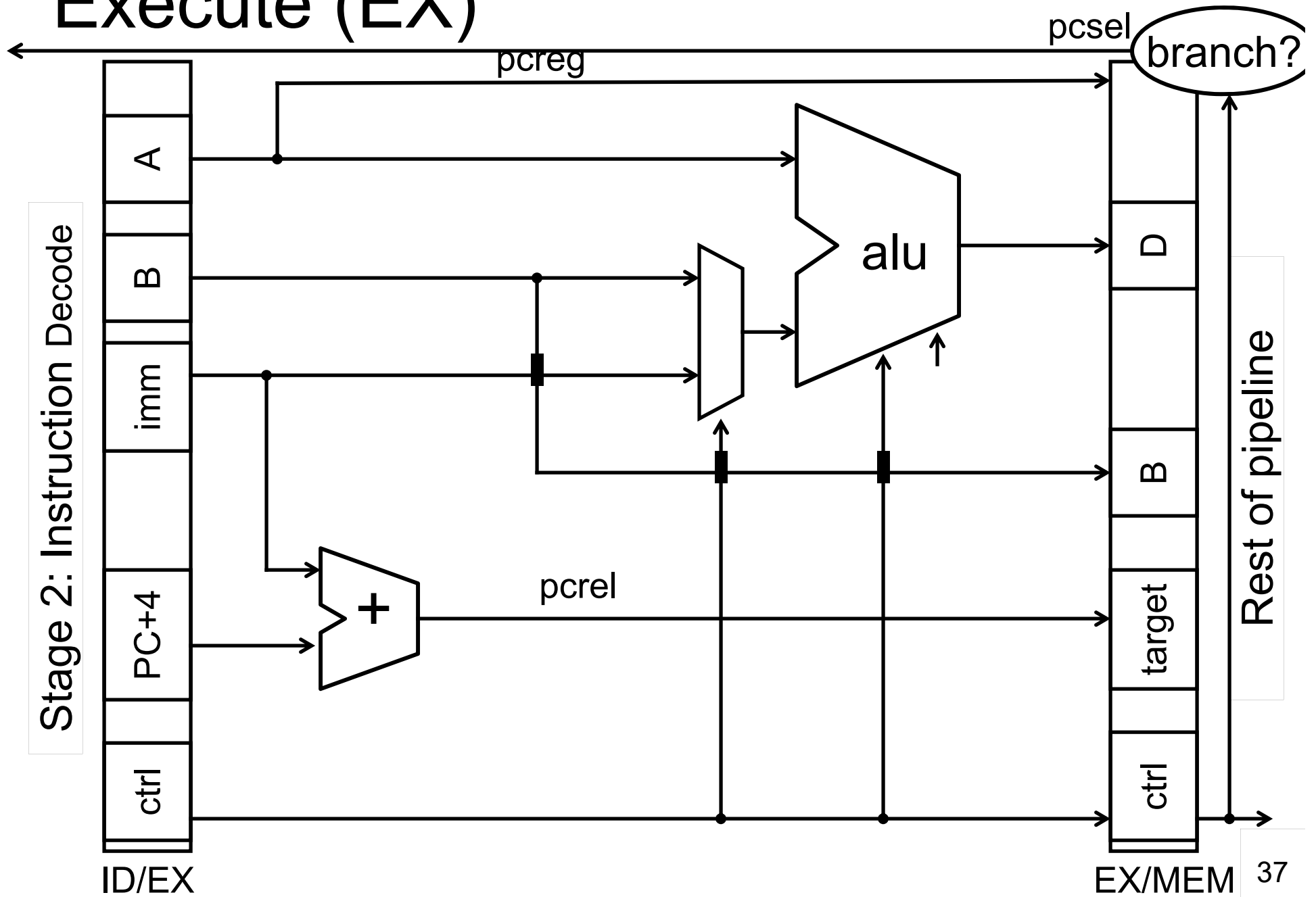
MEM

- 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

Stage 3: Execute

EX/MEM

pcsel

branch?

pcreg

pcrel

D

B

target

ctrl

addr

d_{in} memory d_{out}

mc

Rest of pipeline

D

M

ctrl

MEM/WB

39
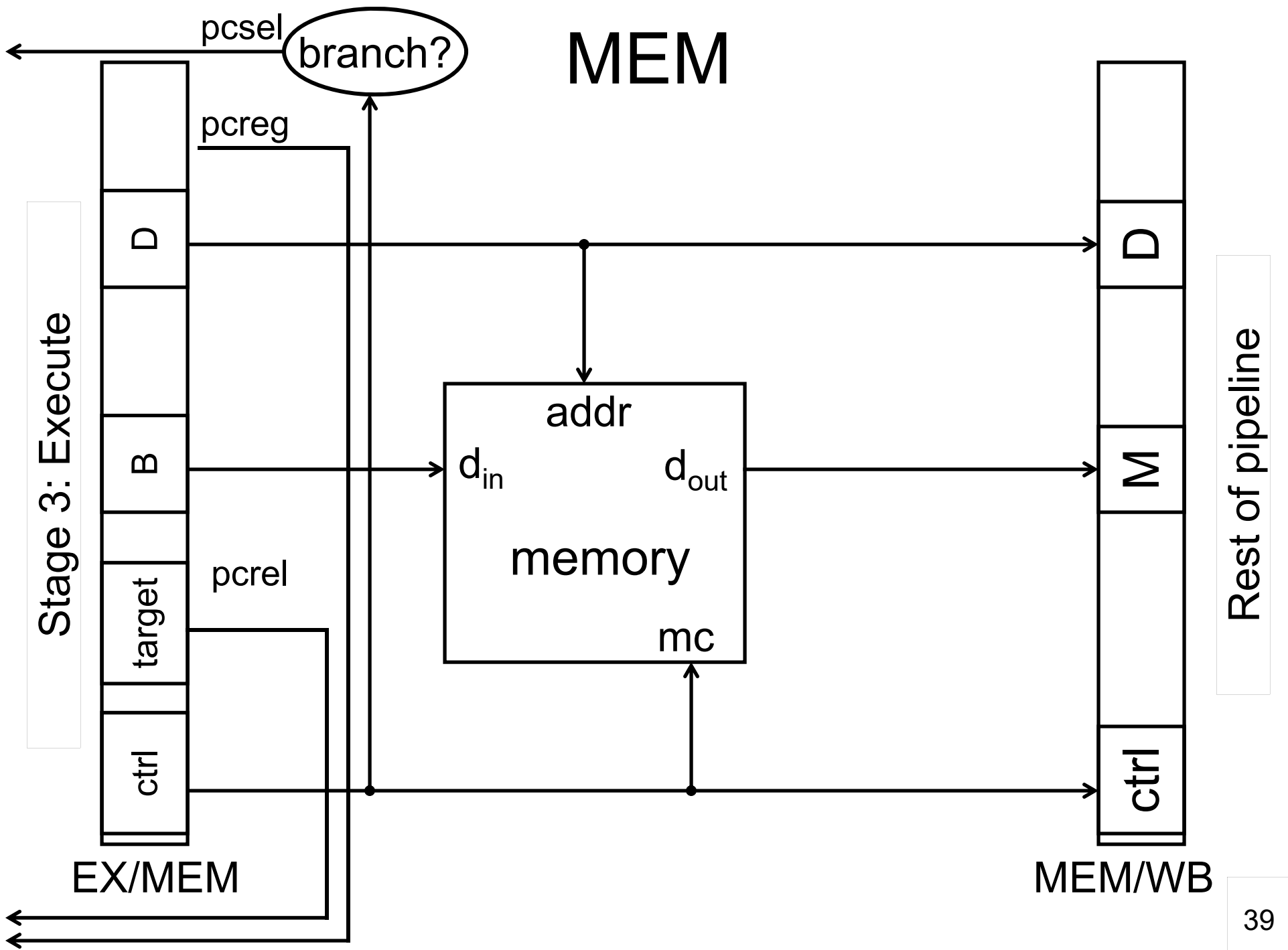
WB

- 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
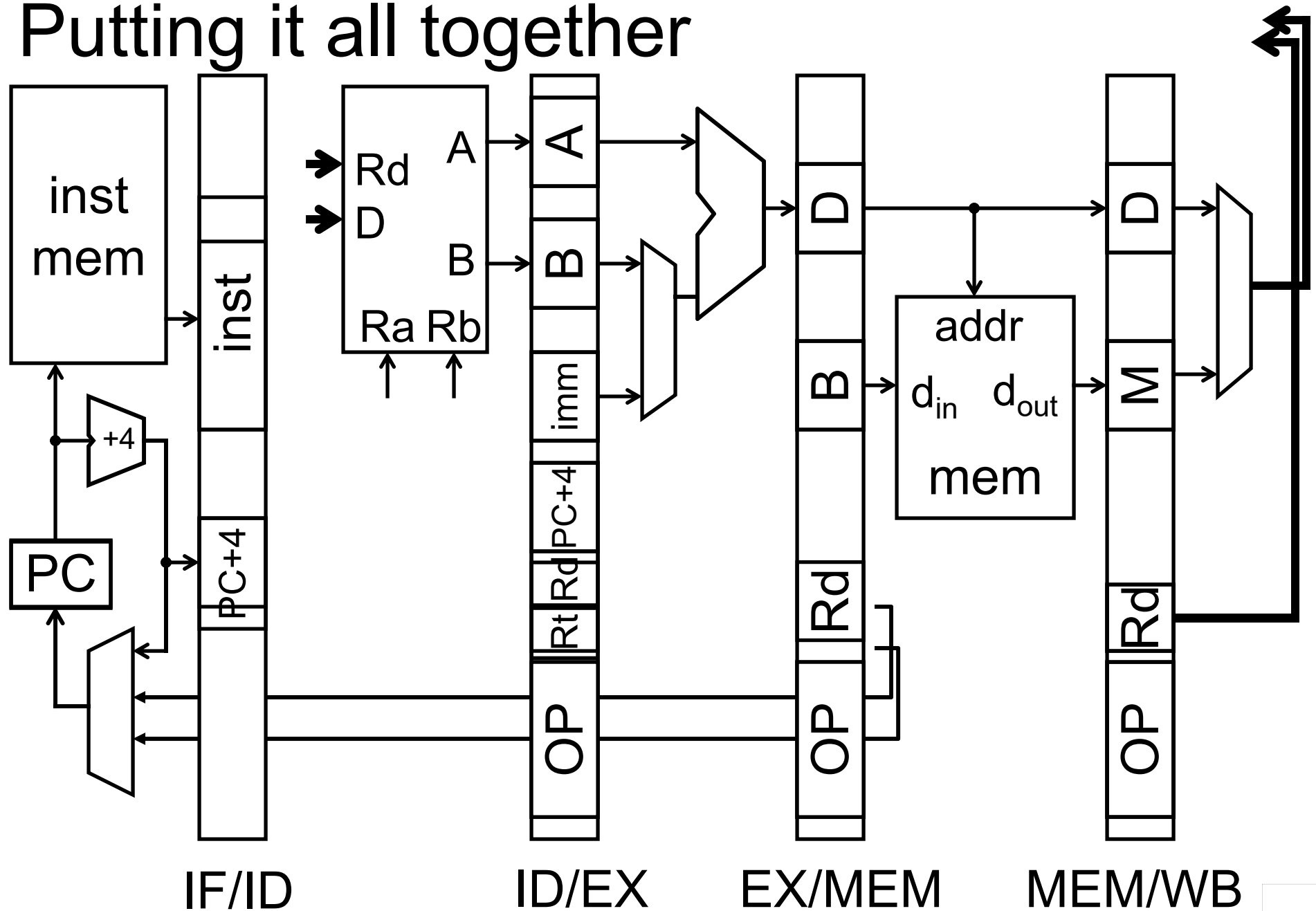
WB

result

D

M

Stage 4: Memory

dest

ctrl

MEM/WB

Putting it all together

inst mem

inst

+4

PC

PC+4

Rd
D
A
B
Ra Rb

A
B
imm
PC+4
Rt Rd
OP

D

B

Rd
OP

addr

d_{in} d_{out}

mem

D

M

Rd
OP

IF/ID   ID/EX   EX/MEM   MEM/WB

42

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)

RISC-V is *designed* for pipelining

- Instructions same length
  - 32 bits, easy to fetch and then decode

- 4 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
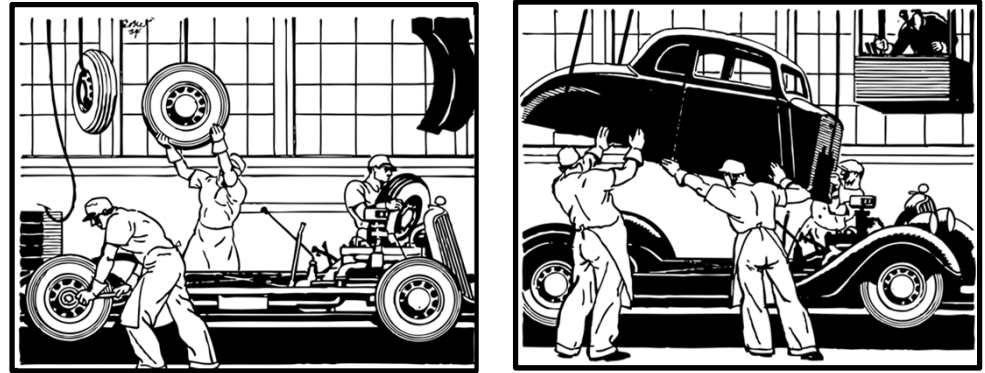
Hazards
- Structural
- Data Hazards
- Control Hazards

Example: Sample Code (Simple)

```
add     x3 ← x1, x2
nand    x6 ← x4, x5
lw      x4 ← x2, 20
add     x5 ← x2, x5
sw      x7 → x3, 12
```

Assume 8-register machine

MUX

4

+

PC+4

PC

Inst mem

instruction

regA
regB

Register file

x0 **0**
x1
x2
x3
x4
x5
x6
x7

extend

Bits 7-11

Bits 0-6

PC+4

valA

valB

imm

Rd

op

MUX

ALU

target

ALU result

valB

dest

op

Data mem

ALU result

mdata

dest

op

MUX

data

dest

IF/ID   ID/EX   EX/MEM   MEM/WB

49
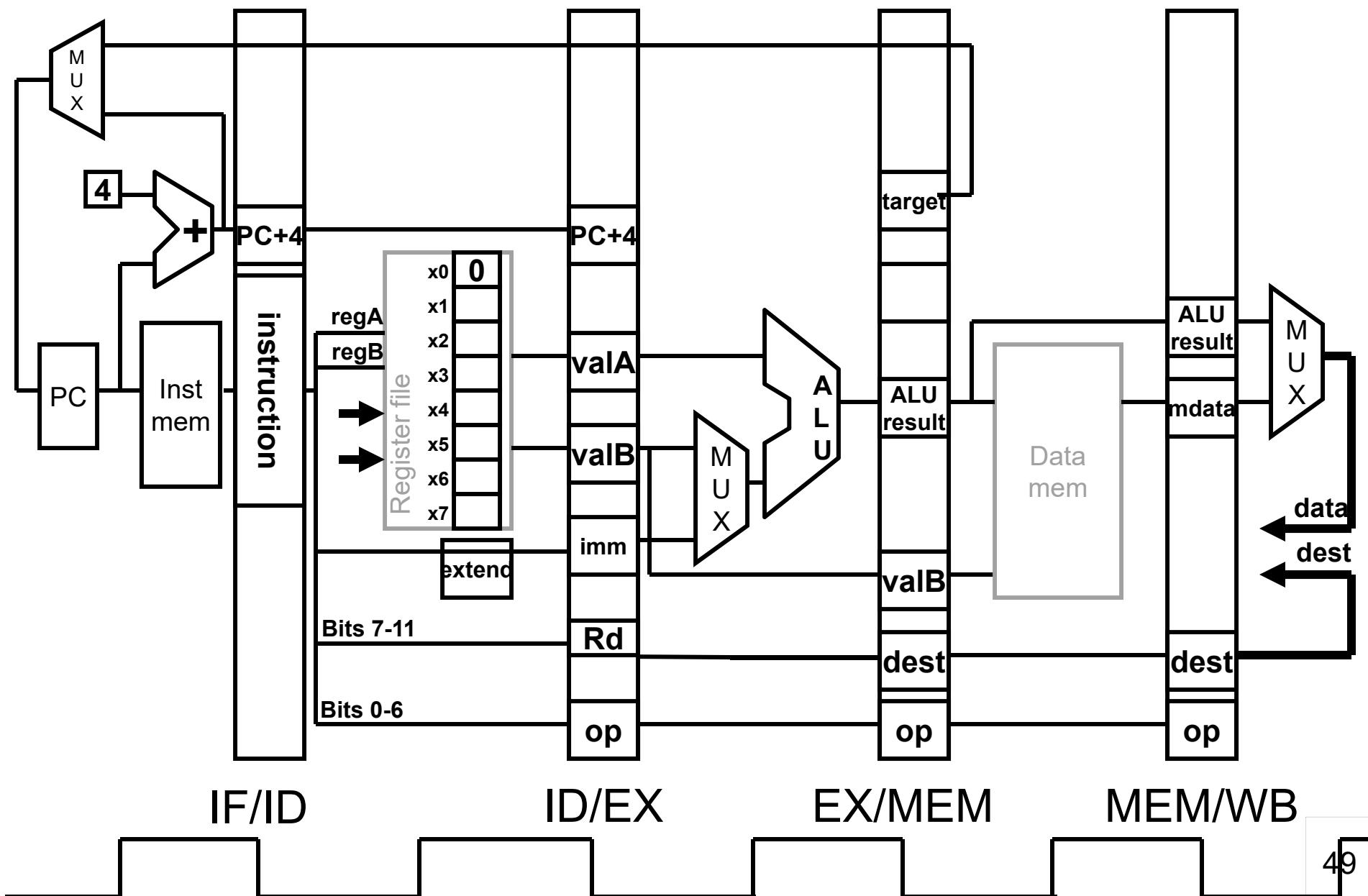
At time 1,
Fetch
add x3 x1 x2

Example: Start State @ Cycle 0

Initial
State

IF/ID   ID/EX   EX/MEM   MEM/WB

50

Cycle 1: Fetch add

add 3 1 2

Fetch:
 add 3 1 2

MUX

4

+

4

PC 4

8

Add Nand Lw Add sw

add 3 1 2

regA
regB

Register file

x0 | 0
x1 | 36
x2 | 9
x3 | 12
x4 | 18
x5 | 7
x6 | 41
x7 | 22

extend

Bits 7-11

Bits 0-6

0 4

0

0 36

0 9

0

0 3

nop add

MUX

A L U

MUX

0

0

0

0

0

0

nop

Data mem

0

0

0

0

nop

MUX

data

dest

Time: 1 2 IF/ID   ID/EX   EX/MEM   MEM/WB

51
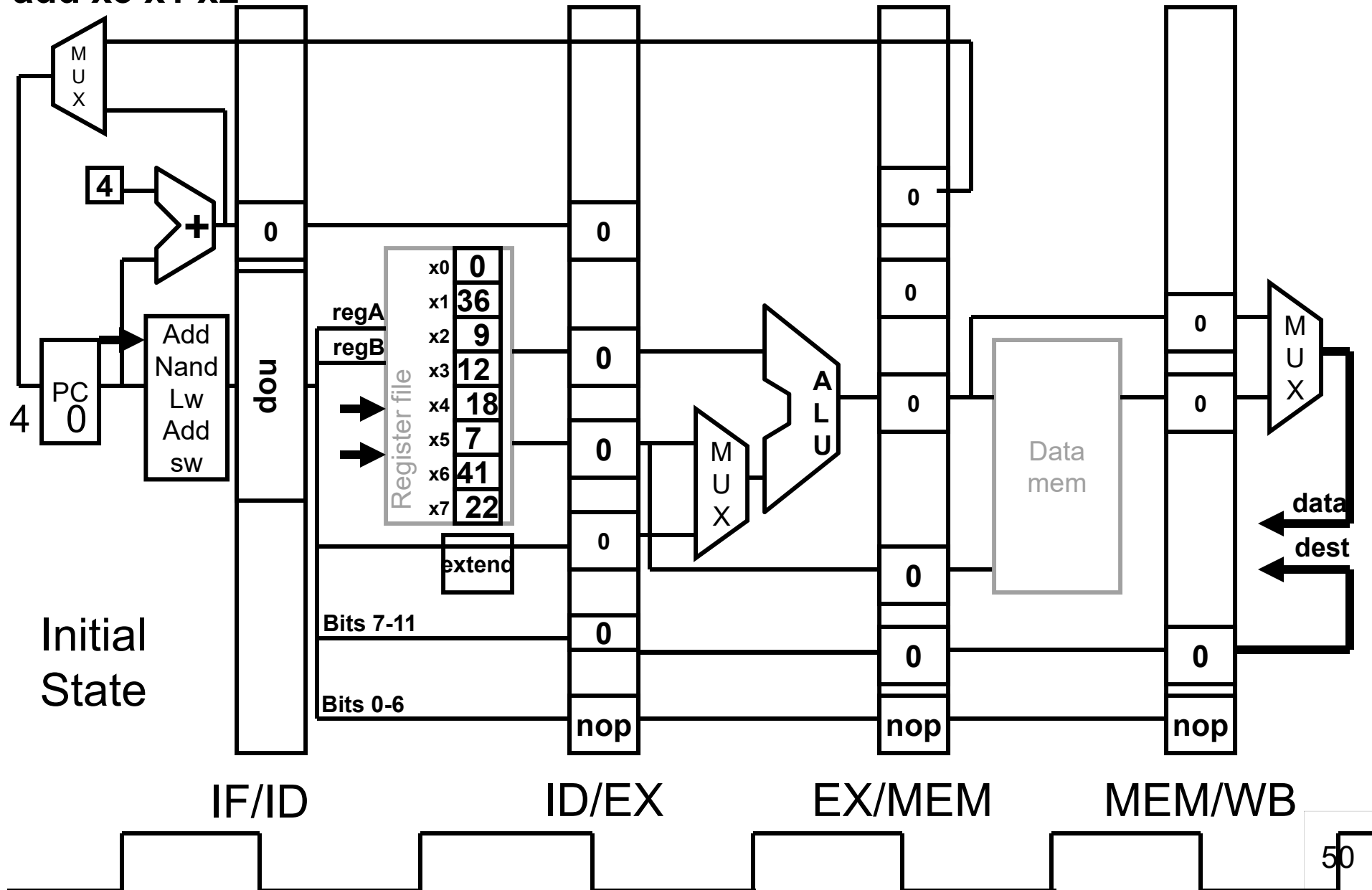
Cycle 2: Fetch nand, Decode add

nand 6 4 5

add 3 1 2

MUX

4

+

8

PC 8

12

Add
Nand
Lw
Add
sw

nand 6 4 5

Register file

x0	0
x1	36
x2	9
x3	12
x4	18
x5	7
x6	41
x7	22

1

2

extend

Bits 7-11

Bits 0-6

0 4

1 8

36 18

0 7

3

3 6

Fetch:
 nand 6 4 5

add nand

36

9

MUX

ALU

0 4

0

0 45

0 9

nop add

3

0 3

Data
mem

0

0

MUX

data

dest

0

nop

Time: 2 3 IF/ID   ID/EX   EX/MEM   MEM/WB

52

Cycle 3: Fetch lw, Decode nand, …

lw 4 2 20

nand 6 4 5

add 3 1 2

nand $(\overline{18 \cdot 7})$

$18 = 01\ 0010$
$7 = 00\ 0111$

$-3 = 11\ 1101$

Register file	
x0	**0**
x1	**36**
x2	**9**
x3	**12**
x4	**18**
x5	**7**
x6	**41**
x7	**22**

MUX

4

+

8

PC
12

16

Add
Nand
Lw
Add
sw

lw 4 2 20

4

5

extend

Bits 7-11

Bits 0-6

Fetch:
lw 4 2 20

8

18

7

3

6

nand

48

0

3/618

45-3

97

36

add

ALU

9/7

MUX

Data
mem

45-3

0

97

36

3

nand

p 45

0

33

nop

add

data

dest

M
U
X

Time: 34 IF/ID

ID/EX

EX/MEM

MEM/WB

53

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

add 5 2 5   **lw 4 2 20**   **nand 6 4 5**   **add 3 1 2**

Fetch:
  add 5 2 5

Time: 4   IF/ID   ID/EX   EX/MEM   MEM/WB

54
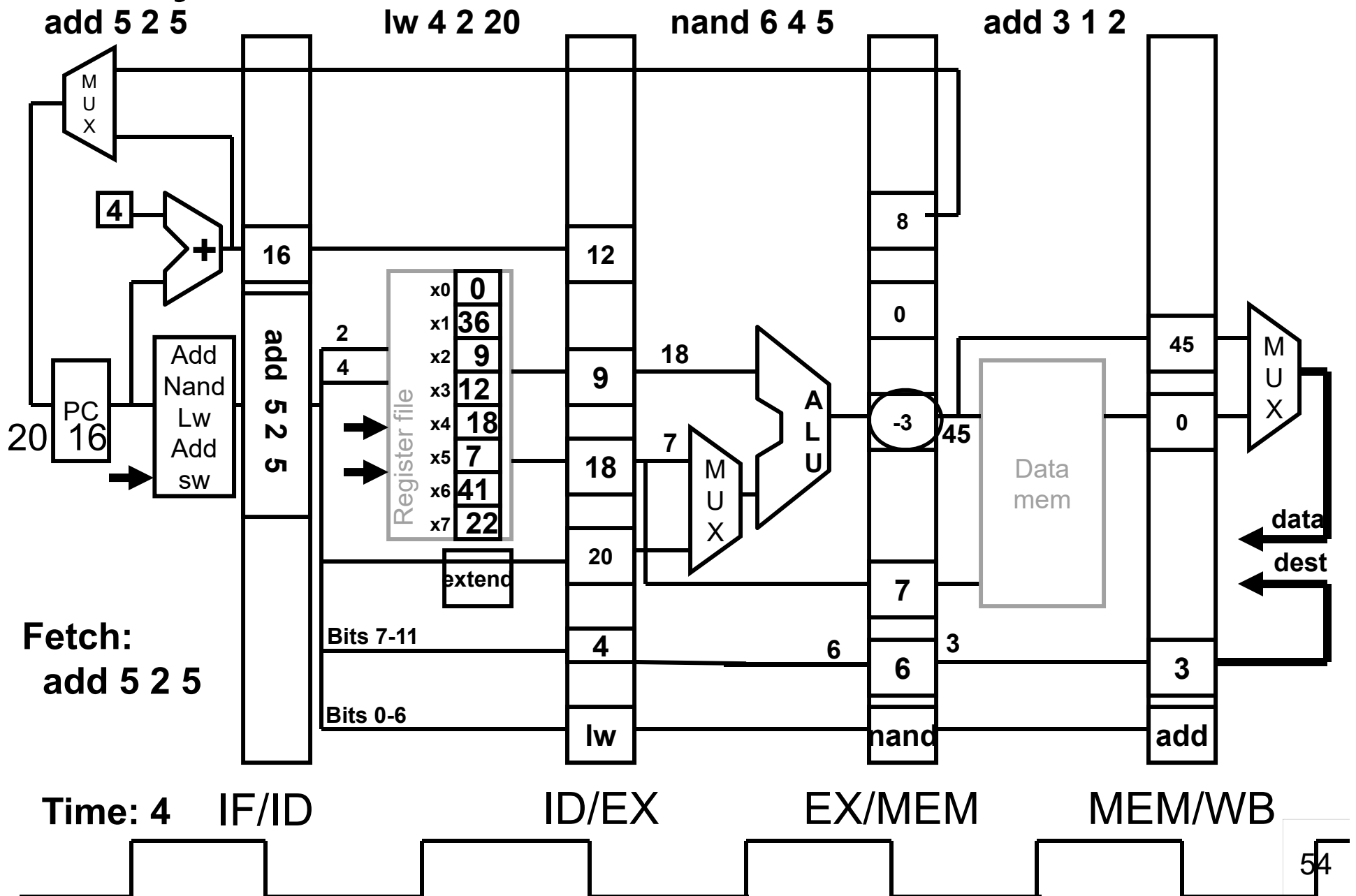
Cycle 5: Fetch sw, Decode add, ...

sw 7 3 12   **add 5 2 5**   **lw 4 20 (2)**   **nand 6 4 5**   **add 3 1 2**

Fetch:
sw 7 3 12

Time: 5   IF/ID   ID/EX   EX/MEM   MEM/WB

55

Cycle 6: Decode sw, …

sw 7 3 12   add 5 2 5   lw 4 2 20   nand 6 4 5

Register file	
x0	0
x1	36
x2	9
x3	45
x4	18
x5	7
x6	-3
x7	22

No more instructions

Time: 6   IF/ID   ID/EX   EX/MEM   MEM/WB

56

Cycle 7: Execute sw, ...

nop **nop** **sw 7 3 12** **add 5 2 5** **lw 4 2 20**

M
U
X

4

+

Add
Nand
Lw
Add
sw

PC
32 **28**

Register file

x0	**0**
x1	**36**
x2	**9**
x3	**45**
x4	**99**
x5	**7**
x6	**-3**
x7	**22**

extend

Bits 7-11

Bits 0-6

No more instructions

45

M
U
X

12

A
L
U

20

0

57 **16**

22

7 **5**

7

sw

Data
mem

16

0

5

add

M
U
X

99

data

dest

4

5

Time: 7 IF/ID ID/EX EX/MEM MEM/WB

57

Cycle 8: Memory sw, ...

nop　　　　　　**nop**　　　　　　　**nop**　　　　　**sw 7 3 12**　　　**add 5 2 5**

MUX

4

+

36　PC 32

Add
Nand
Lw
Add
sw

**No more
instructions**

Register file

x0	0
x1	36
x2	9
x3	45
x4	99
x5	16
x6	-3
x7	22

extend

Bits 7-11

Bits 0-6

MUX

A
L
U

57

22

Data
mem

57

0

16

MUX

data

dest

5

7

sw

Time: 8　　IF/ID　　　　　　ID/EX　　　　　EX/MEM　　　　MEM/WB

Cycle 9: Writeback sw, ...

M U X

4

+

Add
Nand
Lw
Add
sw

PC
40 36

x0 **0**
x1 **36**
x2 **9**
x3 **45**
x4 **99**
x5 **16**
x6 **-3**
x7 **22**

Register file

extend

Bits 7-11

Bits 0-6

A L U

M U X

M U X

Data mem

data

dest

No more instructions

Time: 9   IF/ID   ID/EX   EX/MEM   MEM/WB

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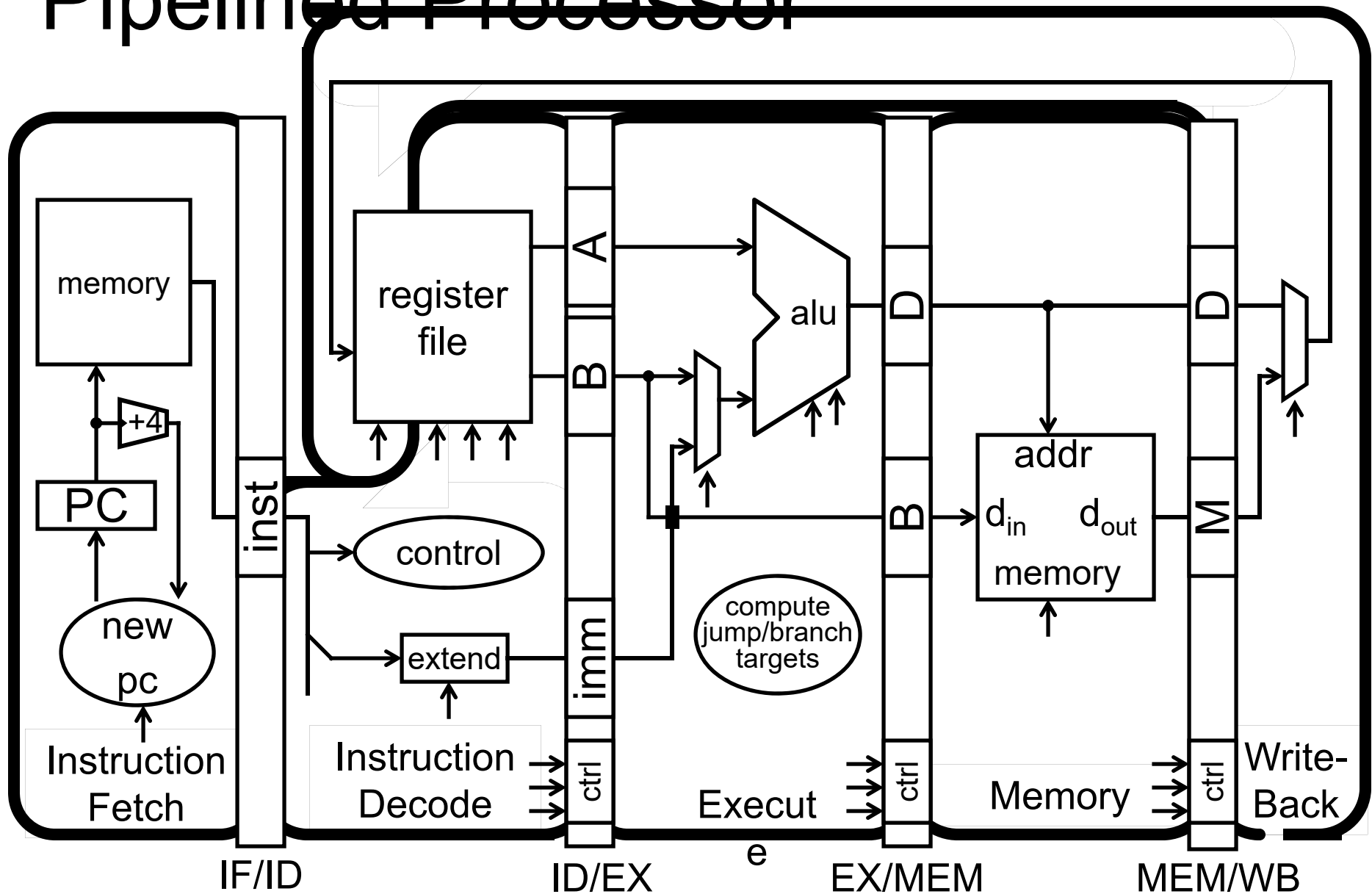
A. You can fetch and decode the same instruction at the same time.
B. You can fetch two instructions at the same time.
C. You can fetch one instruction while decoding another.
D. Instructions only need to visit the pipeline stages that they require.
E. C and D

Pipelined Processor

memory

register file

+4

PC

A

B

alu

D

addr

d_{in} d_{out}

memory

D

M

inst

control

imm

compute jump/branch targets

B

new pc

extend

Instruction Fetch

Instruction Decode

ctrl

Execute

ctrl

Memory

ctrl

Write-Back

IF/ID

ID/EX

EX/MEM

MEM/WB

62
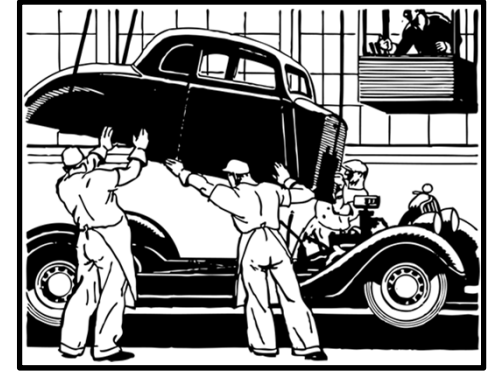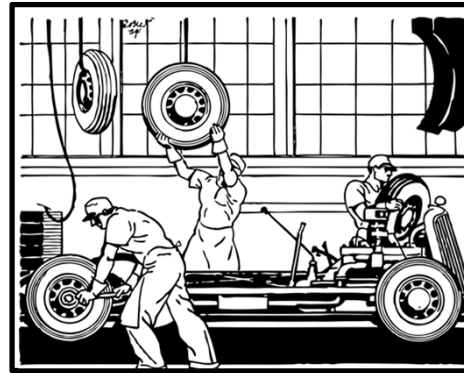
Agenda

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

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 x3 ← x1, x2
x14:    sub x5 ← x3, x4
```

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

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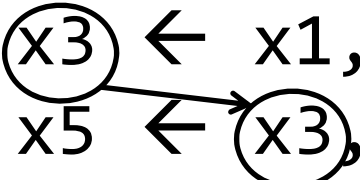C) Cannot tell with the information given.

66

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 (x3) ← x1, x2
x14:     sub x5 ← (x3), x4
```

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

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

67

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

69

Where are the Data Hazards?

time →

Clock cycle

	1	2	3	4	5	6	7	8	9

add x3, x1, x2   IF   ID   ⟩   MEM   WB

sub x5, x3, x4   IF   ID   ⟩   MEM   WB

lw x6, x3, 4   IF   ID   ⟩   MEM   WB

or x5, x3, x5   IF   ID   ⟩   MEM   WB

sw x6, x3, 12   IF   ID   ⟩   MEM   WB

iClicker

add x3, x1, x2

sub x5, x3, x4
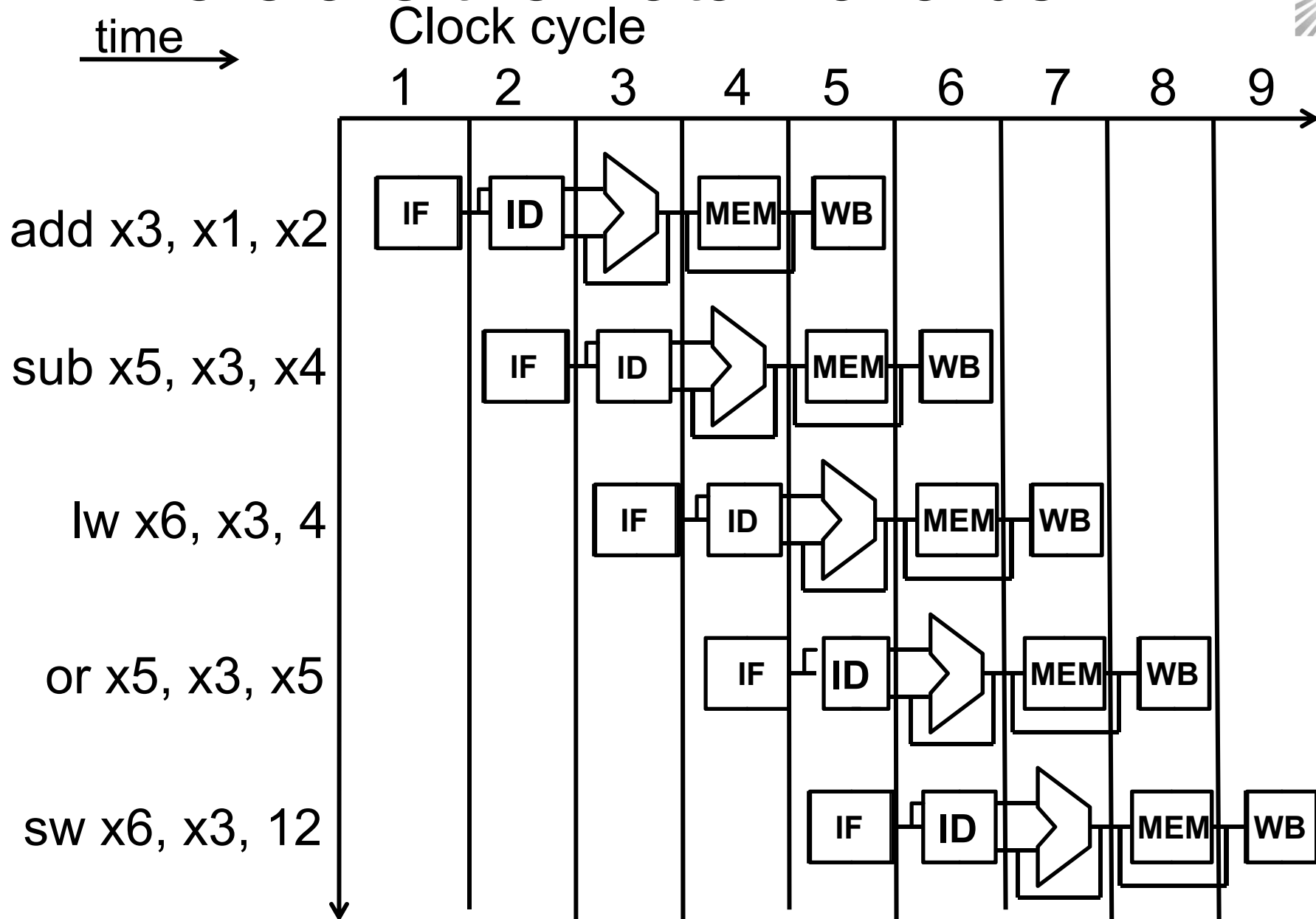
   lw x6, x3, 4

  or x5, x3, x5

sw x6, x3, 12

How many data hazards due to x3 only

A) 1

B) 2

C) 3

D) 4

E) 5

Visualizing Data Hazards (1)
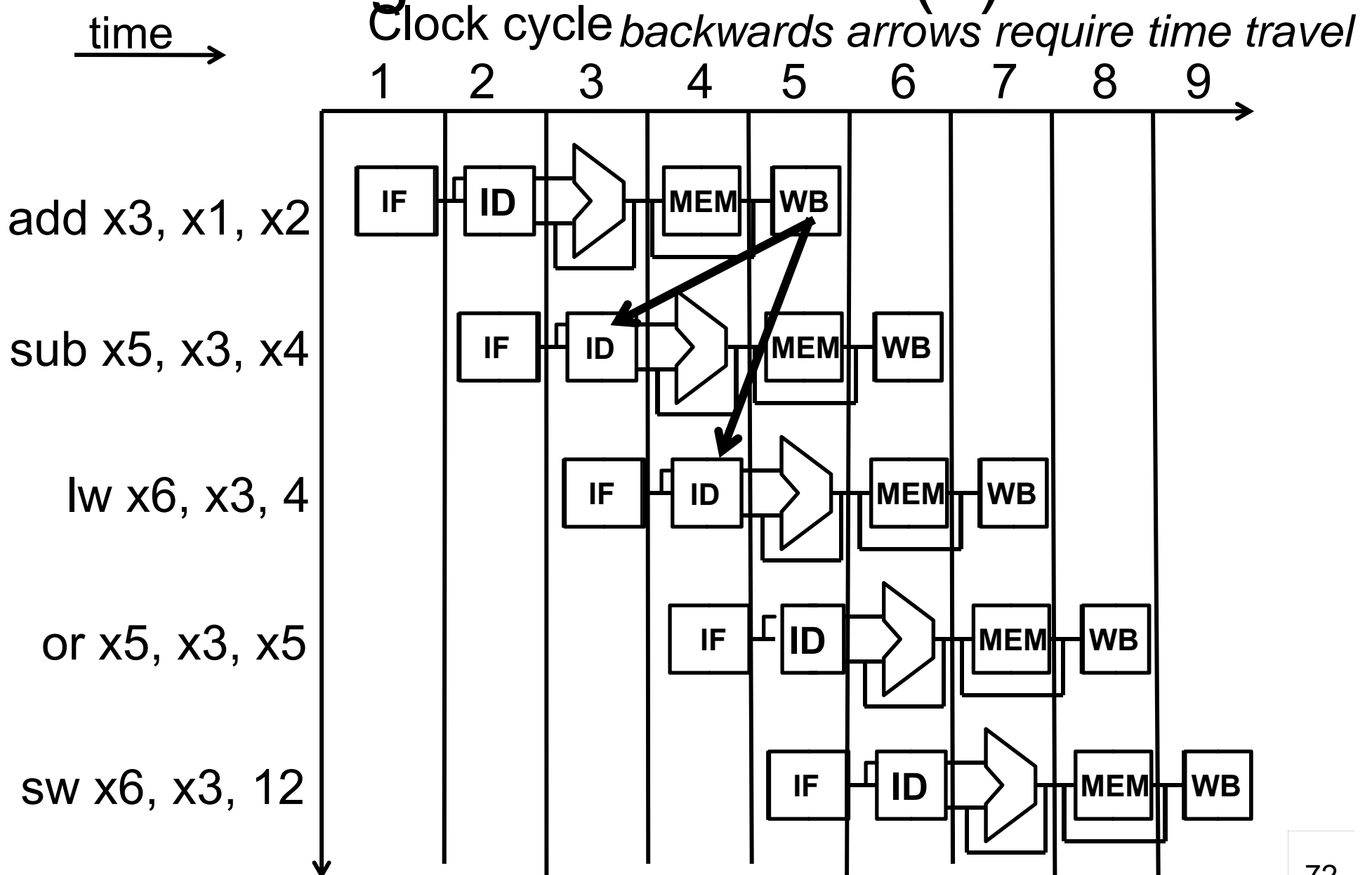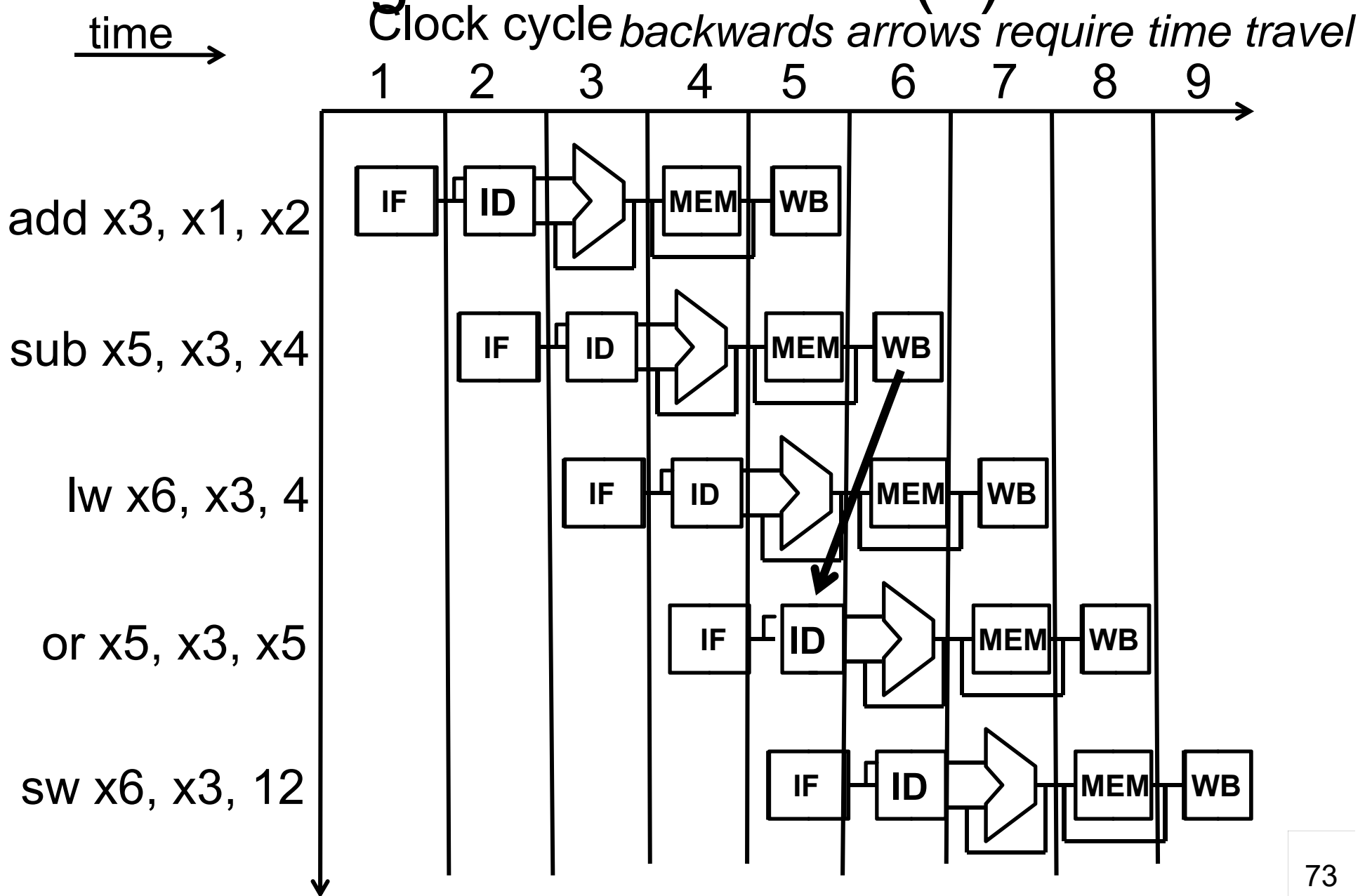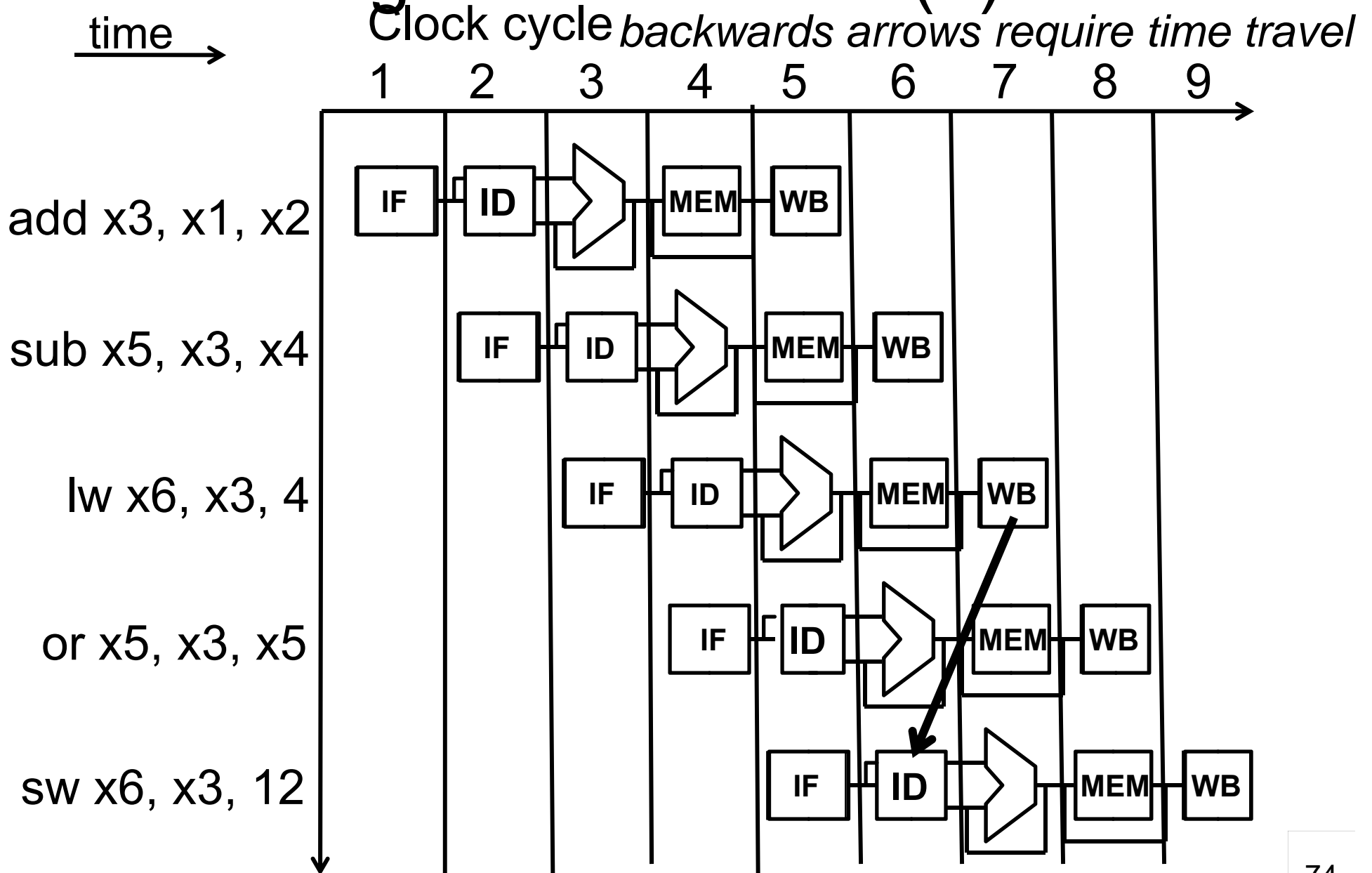
time →

Clock cycle *backwards arrows require time travel*

1   2   3   4   5   6   7   8   9

add x3, x1, x2 — IF ID MEM WB

sub x5, x3, x4 — IF ID MEM WB

lw x6, x3, 4 — IF ID MEM WB

or x5, x3, x5 — IF ID MEM WB

sw x6, x3, 12 — IF ID MEM WB

72

Visualizing Data Hazards (2)

time →

Clock cycle *backwards arrows require time travel*

	1	2	3	4	5	6	7	8	9

add x3, x1, x2   IF   ID   ▷   MEM   WB

sub x5, x3, x4   IF   ID   ▷   MEM   WB

lw x6, x3, 4   IF   ID   ▷   MEM   WB

or x5, x3, x5   IF   ID   ▷   MEM   WB

sw x6, x3, 12   IF   ID   ▷   MEM   WB

Visualizing Data Hazards (3)

time →

Clock cycle *backwards arrows require time travel*

1   2   3   4   5   6   7   8   9

add x3, x1, x2   IF   ID   ▷   MEM   WB

sub x5, x3, x4   IF   ID   ▷   MEM   WB

lw x6, x3, 4   IF   ID   ▷   MEM   WB

or x5, x3, x5   IF   ID   ▷   MEM   WB

sw x6, x3, 12   IF   ID   ▷   MEM   WB

74
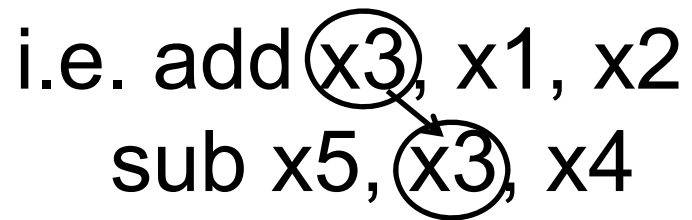
Data Hazards

- 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 x3, x1, x2
   sub x5, x3, x4

How to detect?

Detecting Data Hazards

inst mem

inst

+4

PC

PC+4

Rd

D

A

B

Ra Rb

A

B

imm

PC+4

Rd

OP

D

B

D

d$_{in}$ d$_{out}$

addr

mem

Rd

OP

D

M

Rd

OP

IF/ID.Rs1 ≠ 0 &&
(IF/ID.Rs1==ID/Ex.Rd
IF/ID.Rs1==Ex/M.Rd
IF/ID.Rs1==M/W.Rd)

sub x5,x**3**,x4

add **x3**, x1, x2

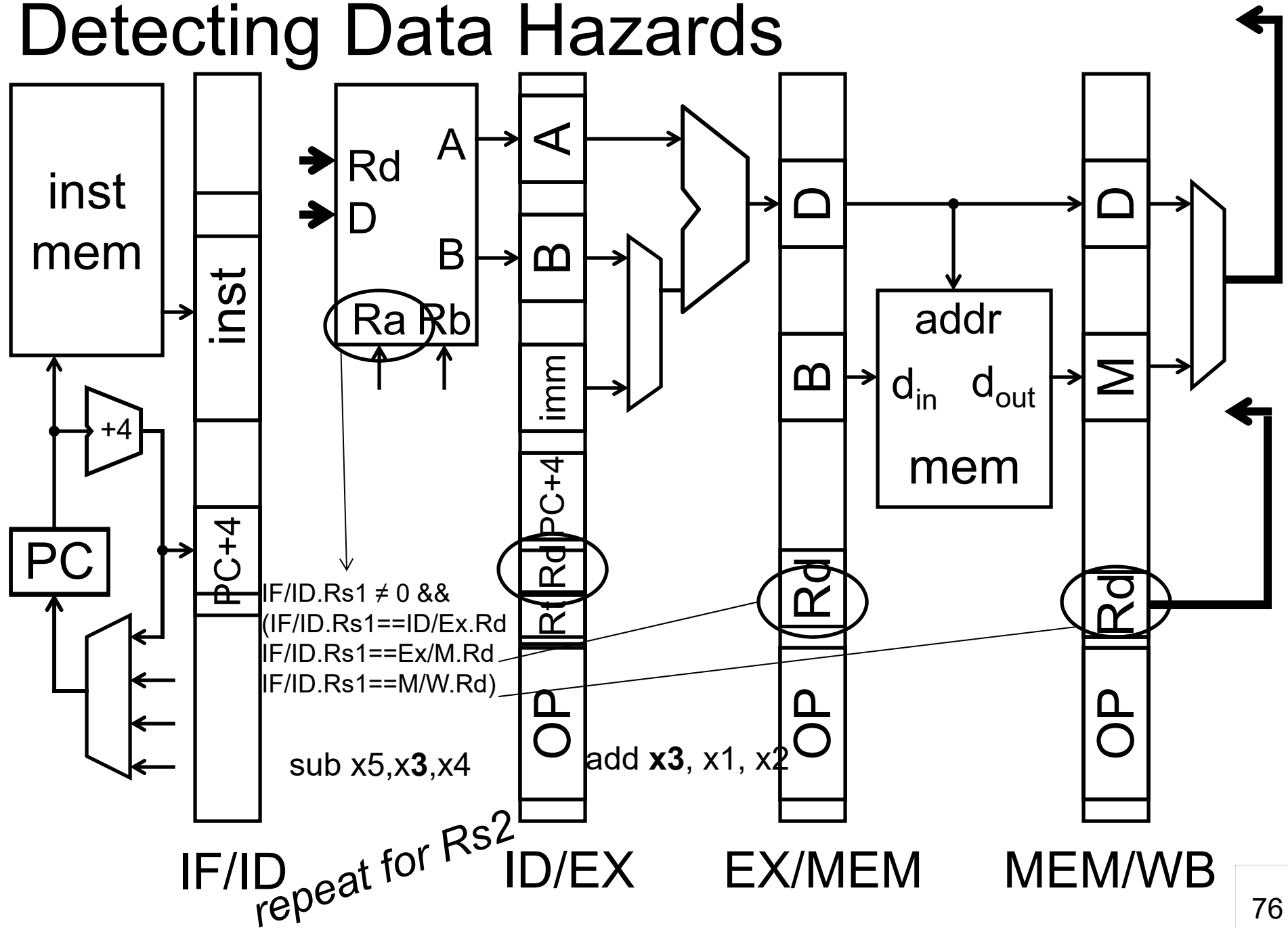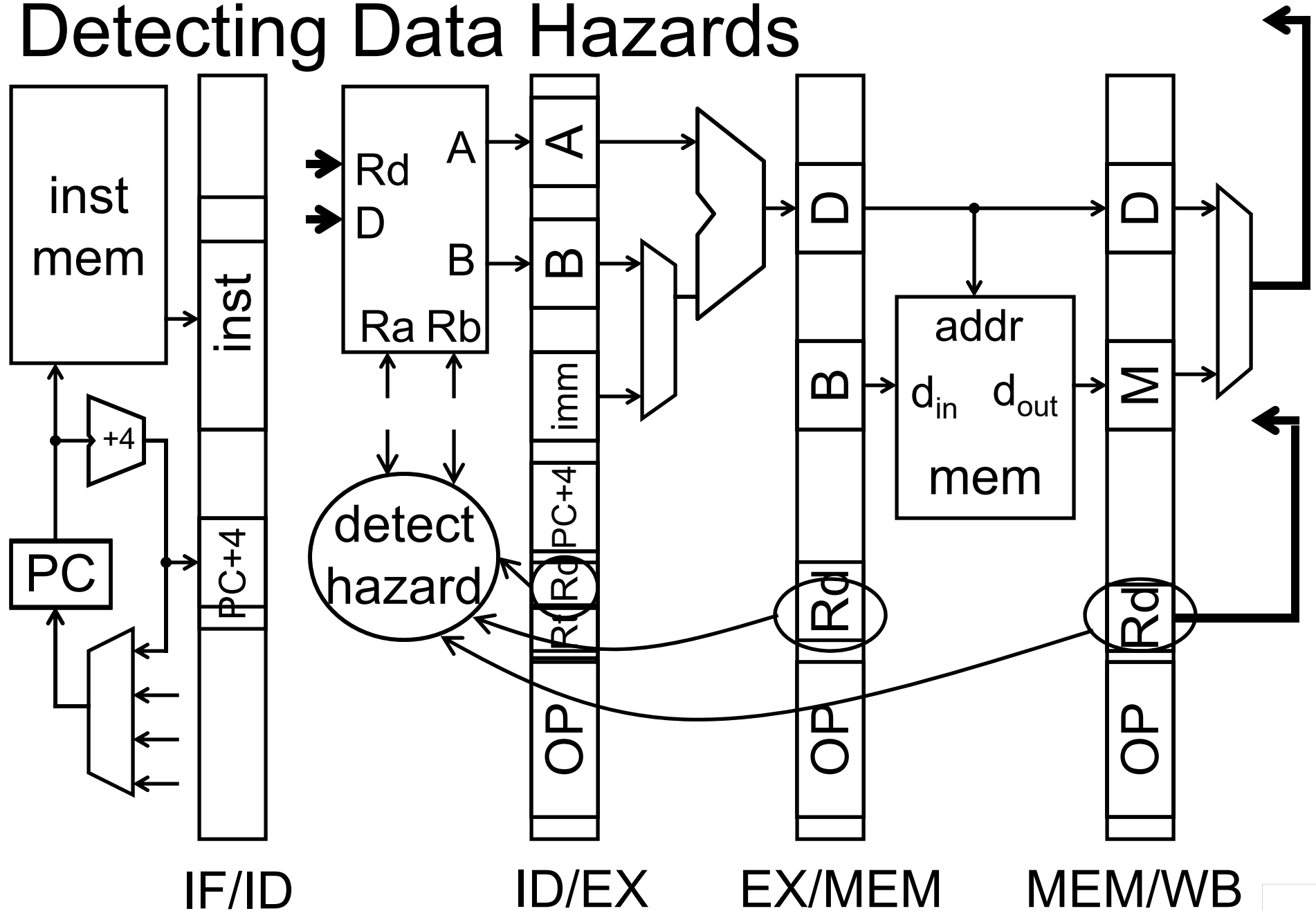IF/ID   repeat for Rs2   ID/EX   EX/MEM   MEM/WB

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.Rs1 != 0 &&
            (IF/ID.Rs1 == ID/EX.Rd ||
             IF/ID.Rs1 == EX/M.Rd ||
             IF/ID.Rs1 == M/WB.Rd))
            || (same for Rs2)
```

Detecting Data Hazards

inst mem

IF/ID

inst

PC+4

PC

+4

Rd
D
A
B
Ra Rb

detect hazard

ID/EX

A
B
imm
PC+4
Rd(Rd)
OP

D
B
Rd
OP

EX/MEM

addr
d_{in} d_{out}
mem

Rd
OP

MEM/WB

D
M
Rd
OP

Takeaway

Data hazards occur when a 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
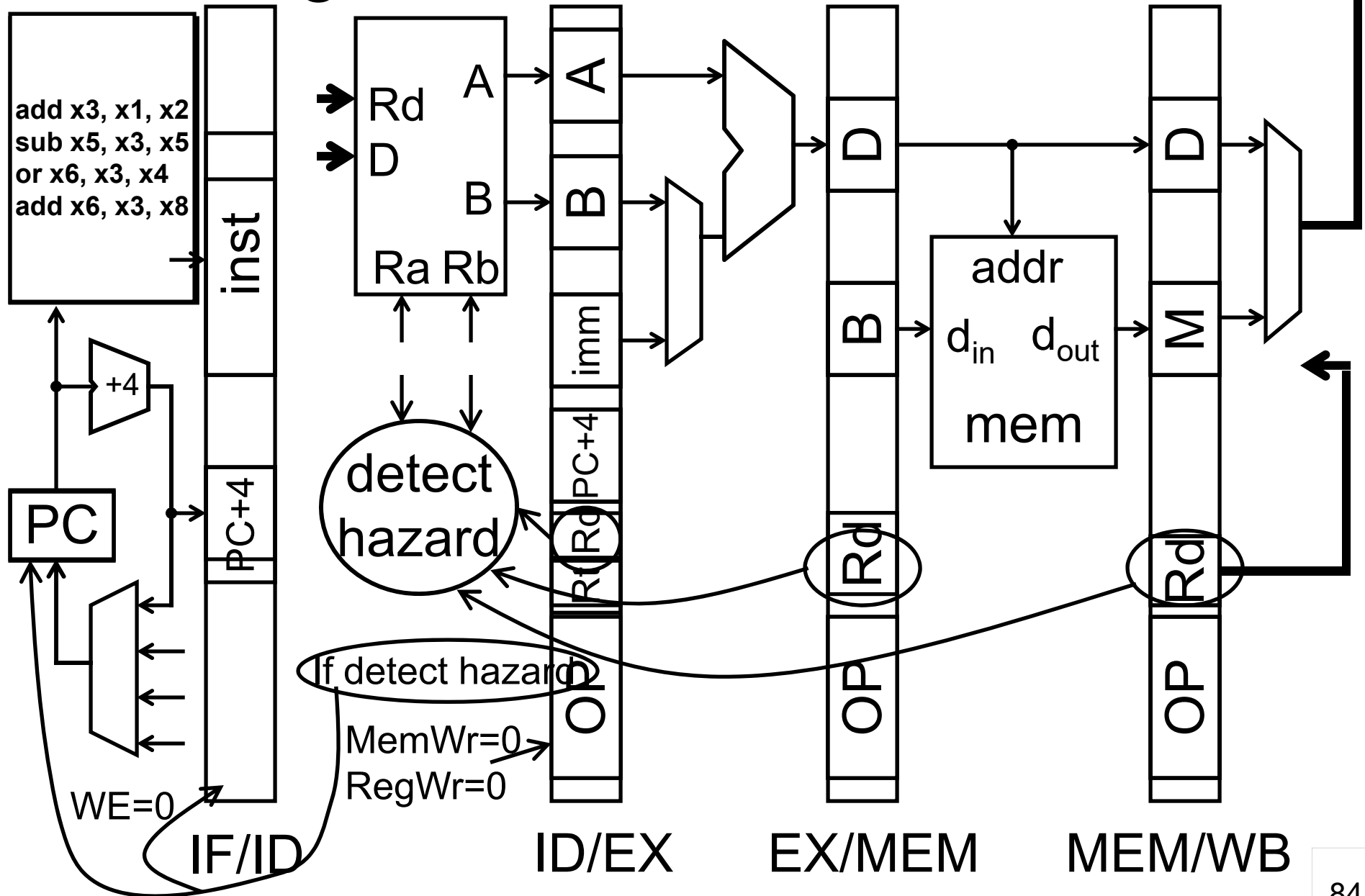   - Pause current and subsequent instructions till safe
3. Forward/bypass
   - Forward data value to where it is needed
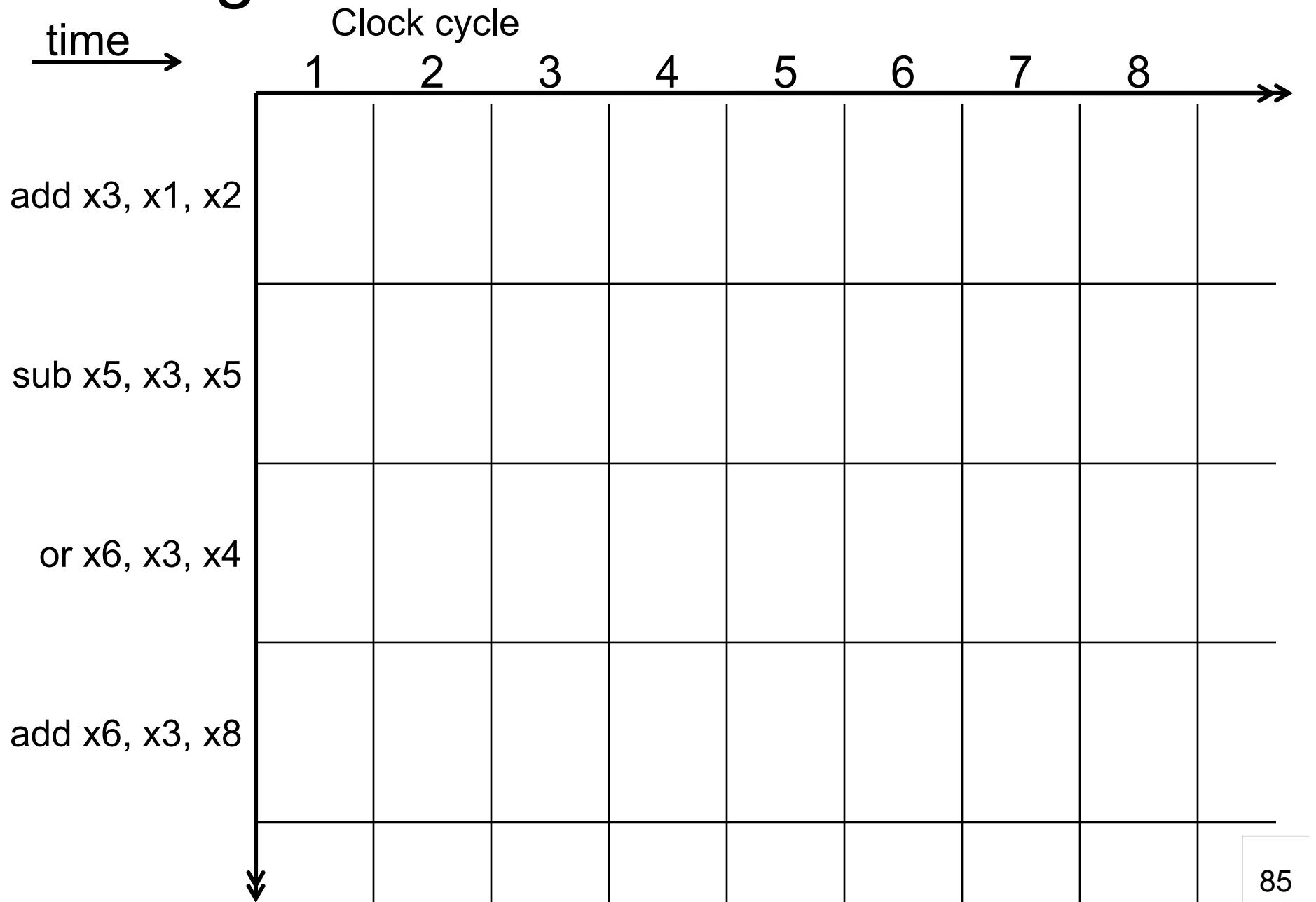   (*Only works if value actually exists already*)

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

add x3, x1, x2
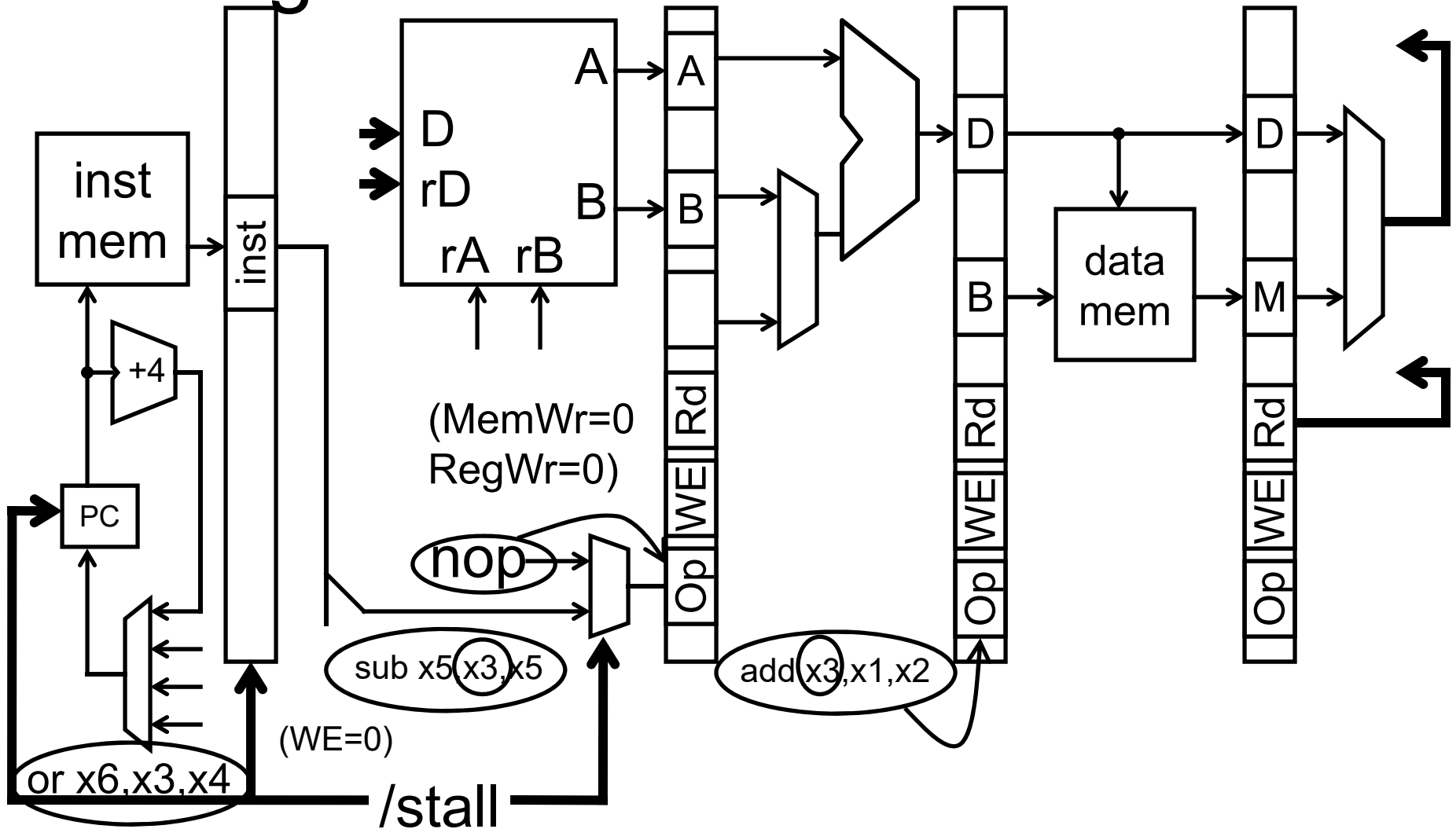sub x5, x3, x5
or x6, x3, x4
add x6, x3, x8

inst

Rd
D
A
B
Ra Rb

A
B
imm
PC+4
R(Rd)
OP

+4

PC

PC+4

detect hazard

If detect hazard

MemWr=0
RegWr=0

WE=0

IF/ID

D
B
Rd
OP

addr
d_{in} d_{out}
mem

ID/EX

EX/MEM

D
M
Rd
OP

MEM/WB

Stalling

time →

Clock cycle

	1	2	3	4	5	6	7	8
add x3, x1, x2								
sub x5, x3, x5								
or x6, x3, x4								
add x6, x3, x8								

Stalling

time →

Clock cycle

	1	2	3	4	5	6	7	8
x3 = 10 add x3, x1, x2	IF	ID	Ex	M	W			
x3 = 20 sub x5, x3, x5		IF	ID	ID	ID	ID	Ex	M	W
or x6, x3, x4			IF	IF	IF	IF	ID	Ex	M
add x6, x3, x8							IF	ID	Ex

3 Stalls

Stalling

inst mem

+4

PC

inst

D
rD

A

B

rA rB

A

B

(MemWr=0
RegWr=0)

Op|WE|Rd

D

B

D

data
mem

Op|WE|Rd

Op|WE|Rd

M

nop

sub x5,x3,x5

add x3,x1,x2

(WE=0)

or x6,x3,x4

/stall

NOP = If(IF/ID.Rs1 ≠ 0 &&

(IF/ID.Rs1==ID/Ex.Rd   ← STALL CONDITION MET

IF/ID.Rs1==Ex/M.Rd

IF/ID.Rs1==M/W.Rd))

87

Stalling

inst mem

+4

PC

inst

D
rD
rA rB
A
B

A
B

D
B

data mem

D
M

Op|WE|Rd

Op|WE|Rd

Op|WE|Rd

(MemWr=0
RegWr=0)

nop

(MemWr=0
RegWr=0)

sub x5,x3,x5

nop

add x3,x1,x2

or x6,x3,x4

(WE=0)

/stall
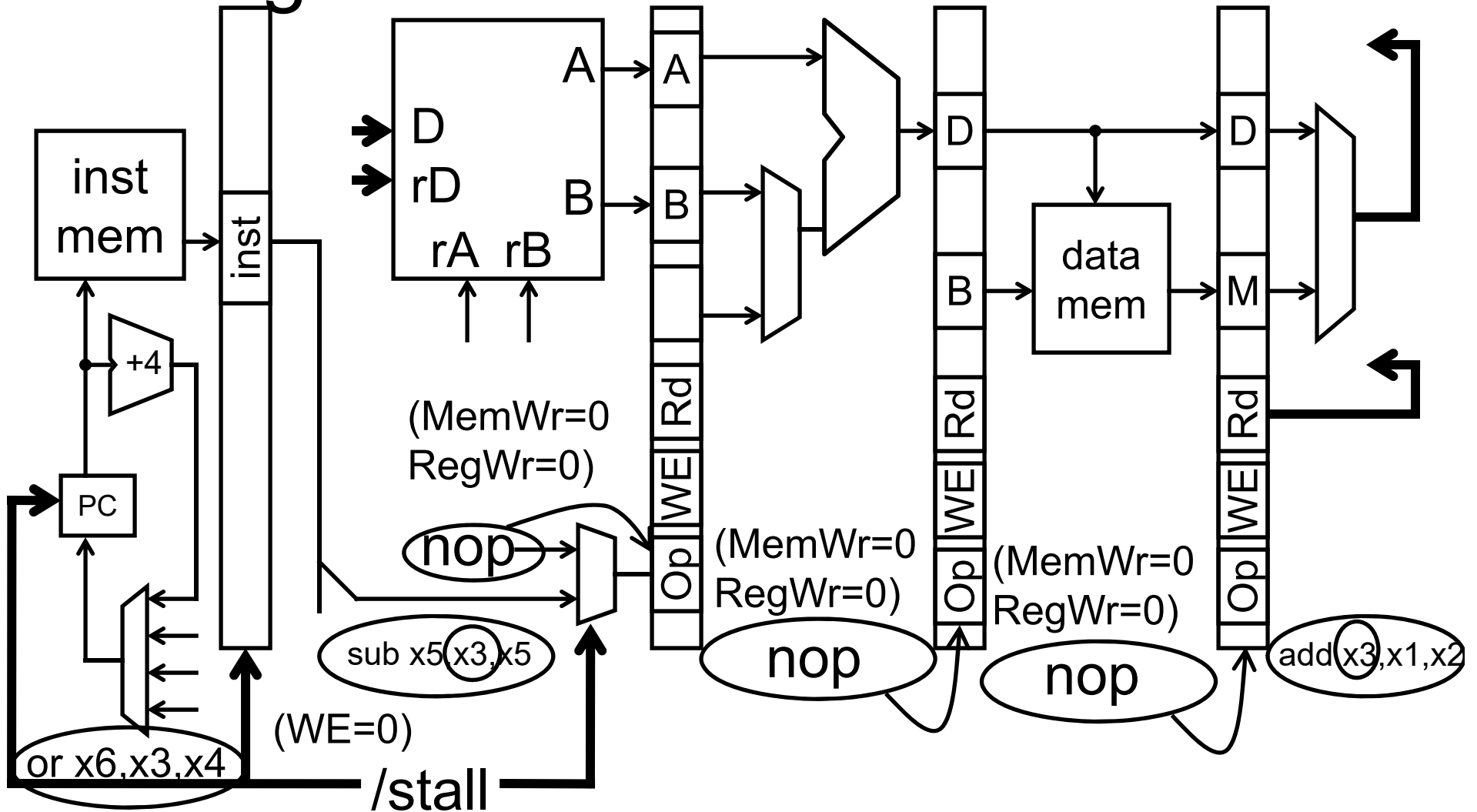
NOP = If(IF/ID.Rs1 ≠ 0 &&
(IF/ID.Rs1==ID/Ex.Rd
IF/ID.Rs1==Ex/M.Rd ← STALL CONDITION MET
IF/ID.Rs1==M/W.Rd))

88

Stalling

inst mem

+4

PC

inst

D
rD
rA rB
A
B

(MemWr=0
RegWr=0)

nop

sub x5,x3,x5

A
B
Op WE Rd

D
B
Op WE Rd

(MemWr=0
RegWr=0)

nop

data mem

D
M
Op WE Rd

(MemWr=0
RegWr=0)

nop

add x3,x1,x2

(WE=0)

or x6,x3,x4

/stall

NOP = If(IF/ID.Rs1 ≠ 0 &&
      (IF/ID.Rs1==ID/Ex.Rd
       IF/ID.Rs1==Ex/M.Rd
       IF/ID.Rs1==M/W.Rd) ←STALL CONDITION MET

Stalling
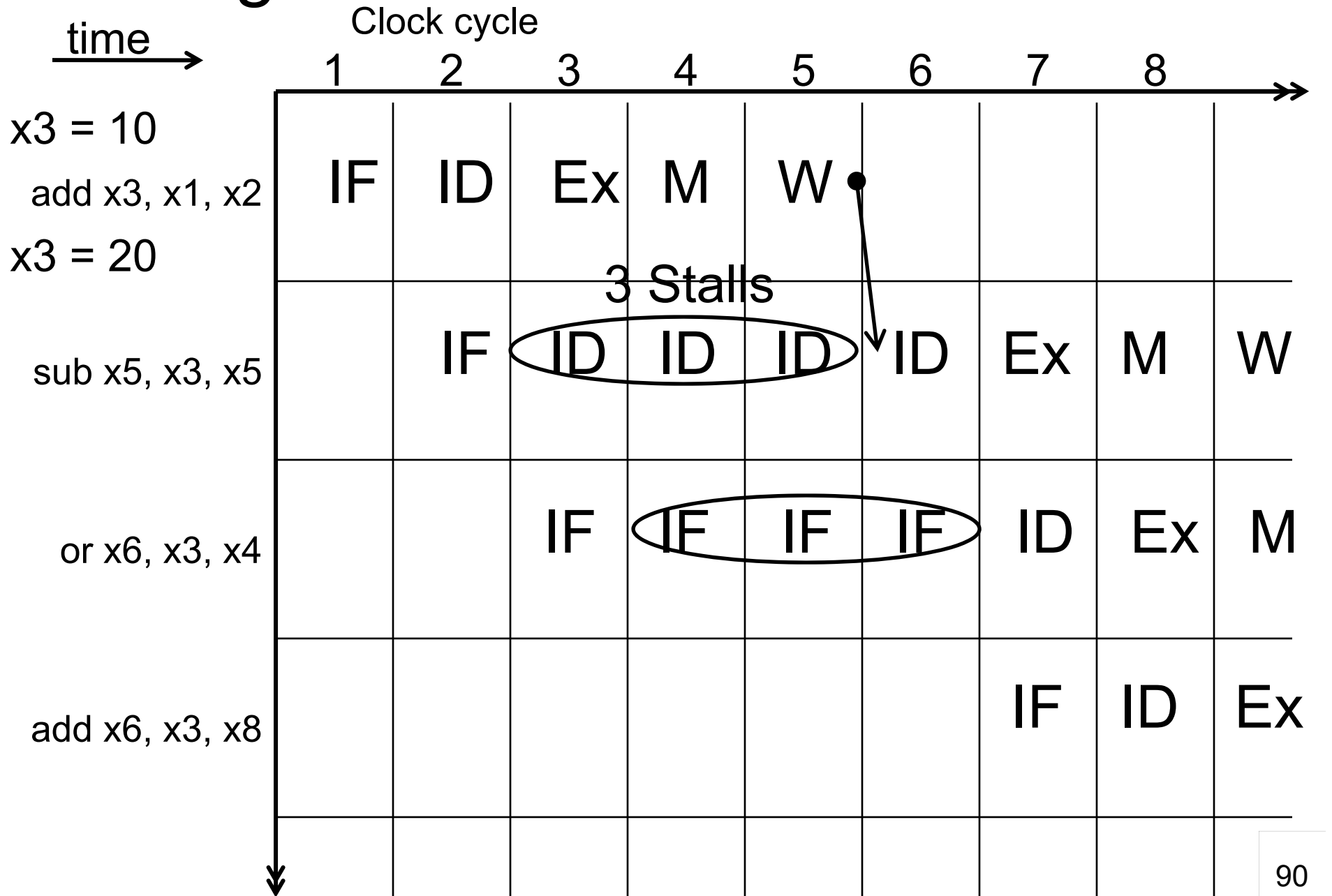
time →

Clock cycle

	1	2	3	4	5	6	7	8

x3 = 10

add x3, x1, x2   IF   ID   Ex   M   W •

x3 = 20

3 Stalls

sub x5, x3, x5   IF (ID   ID   ID) ID   Ex   M   W

or x6, x3, x4   IF (IF   IF   IF) ID   Ex   M

add x6, x3, x8   IF   ID   Ex

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 a 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
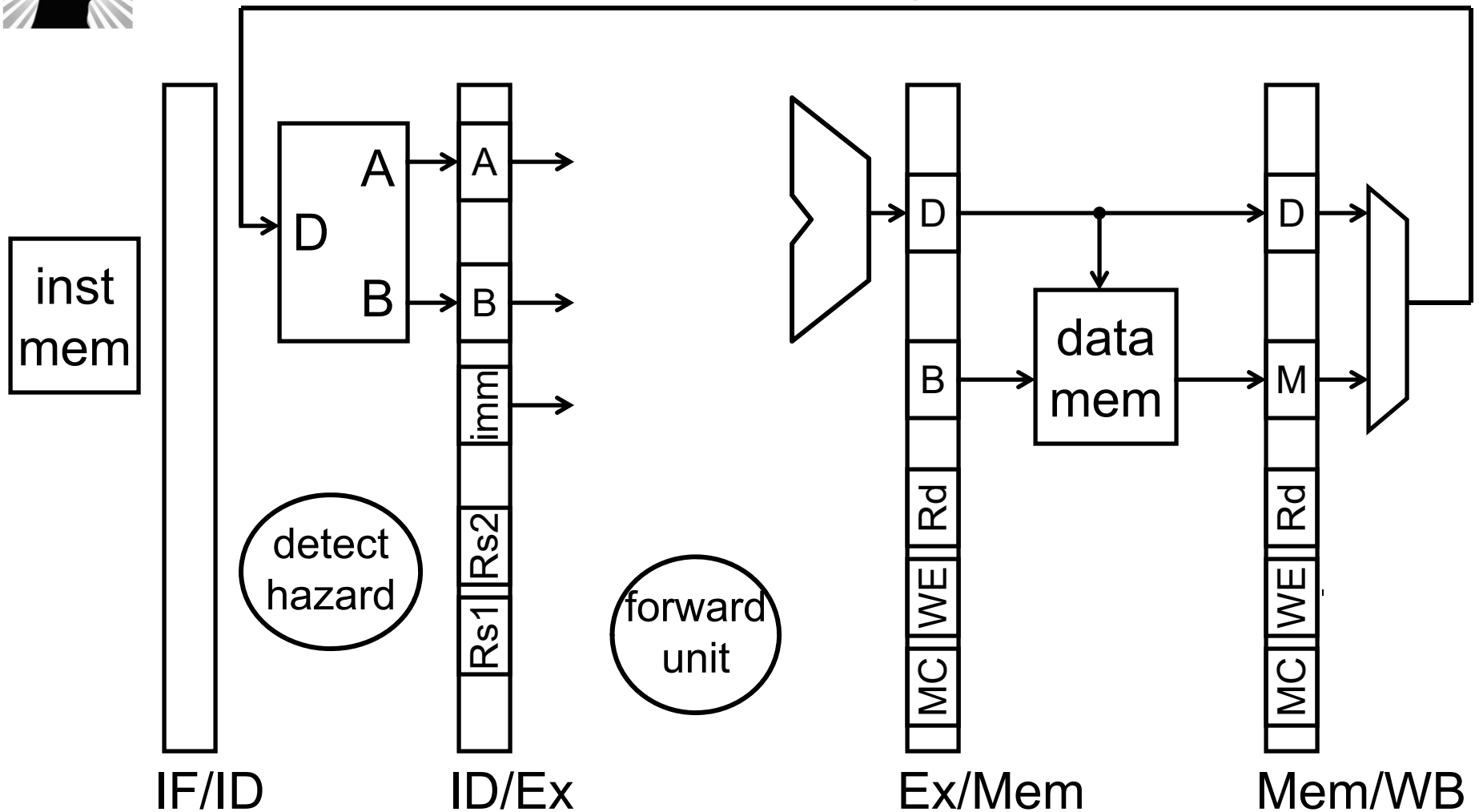3. Forward/bypass
   - Forward data value to where it is needed
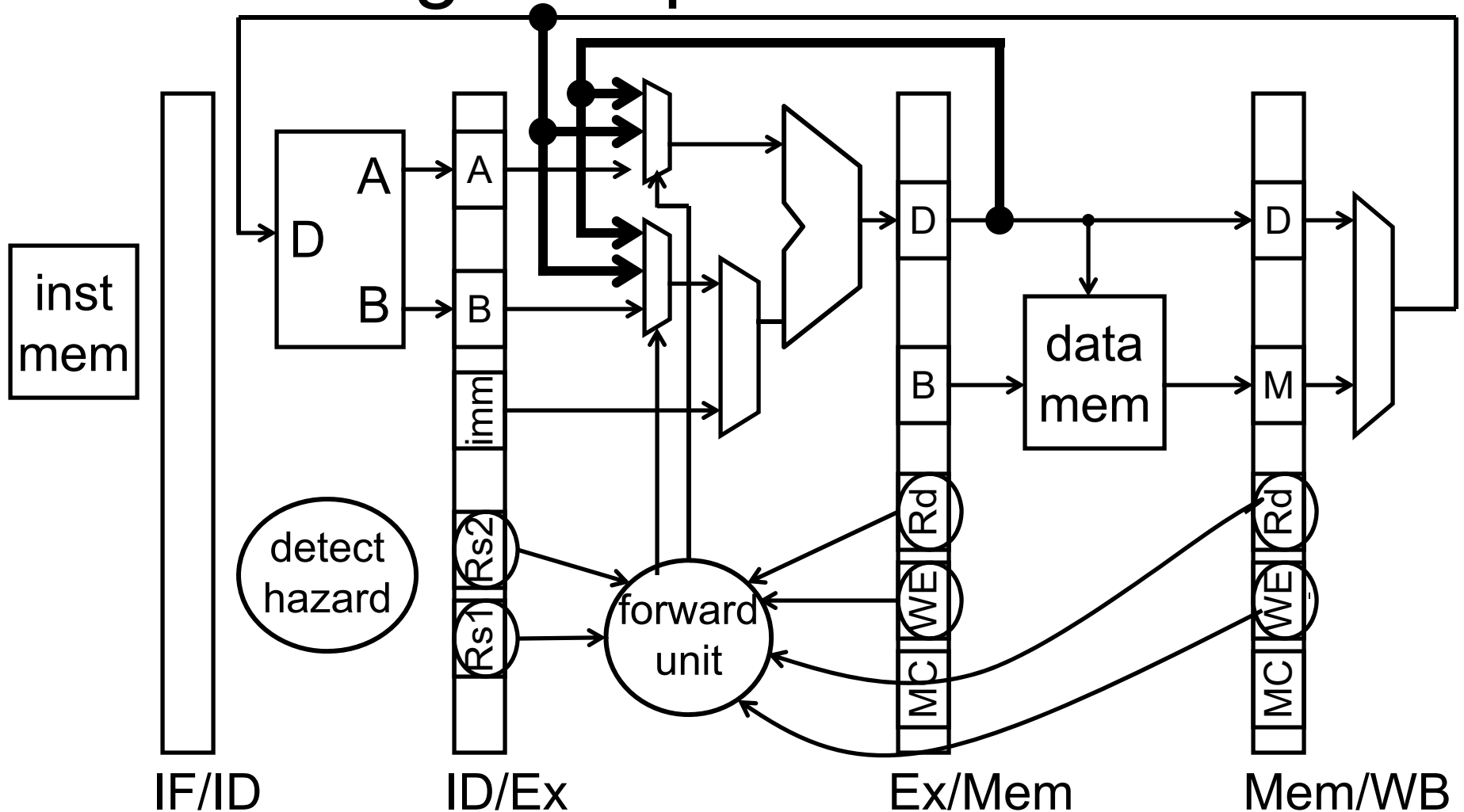   (*Only works if value actually exists already*)

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

inst mem

A A

D

B B

imm

detect hazard

Rs1 Rs2

forward unit

D D

B data mem M

MC WE Rd

MC WE Rd

IF/ID   ID/Ex   Ex/Mem   Mem/WB

95

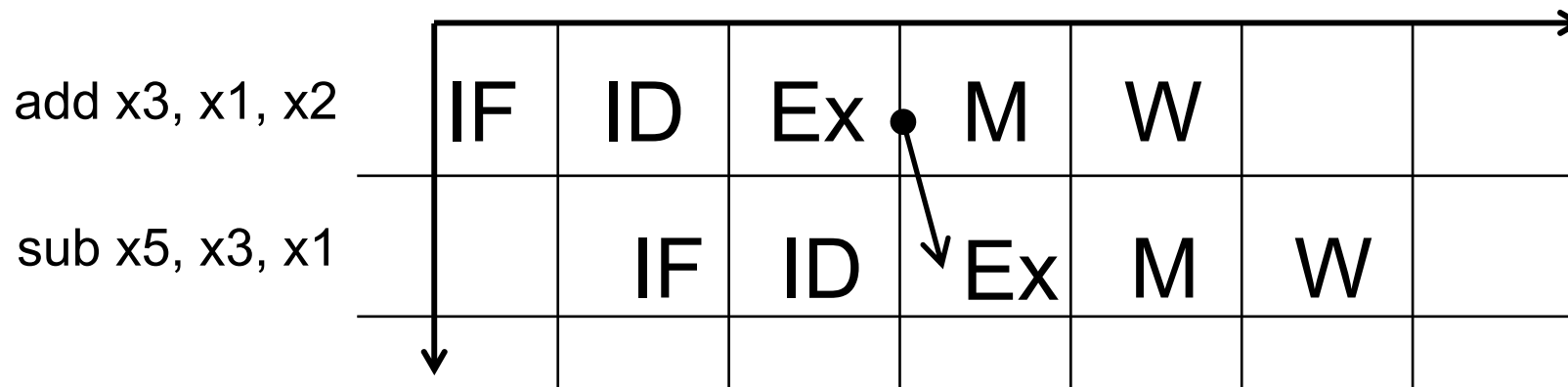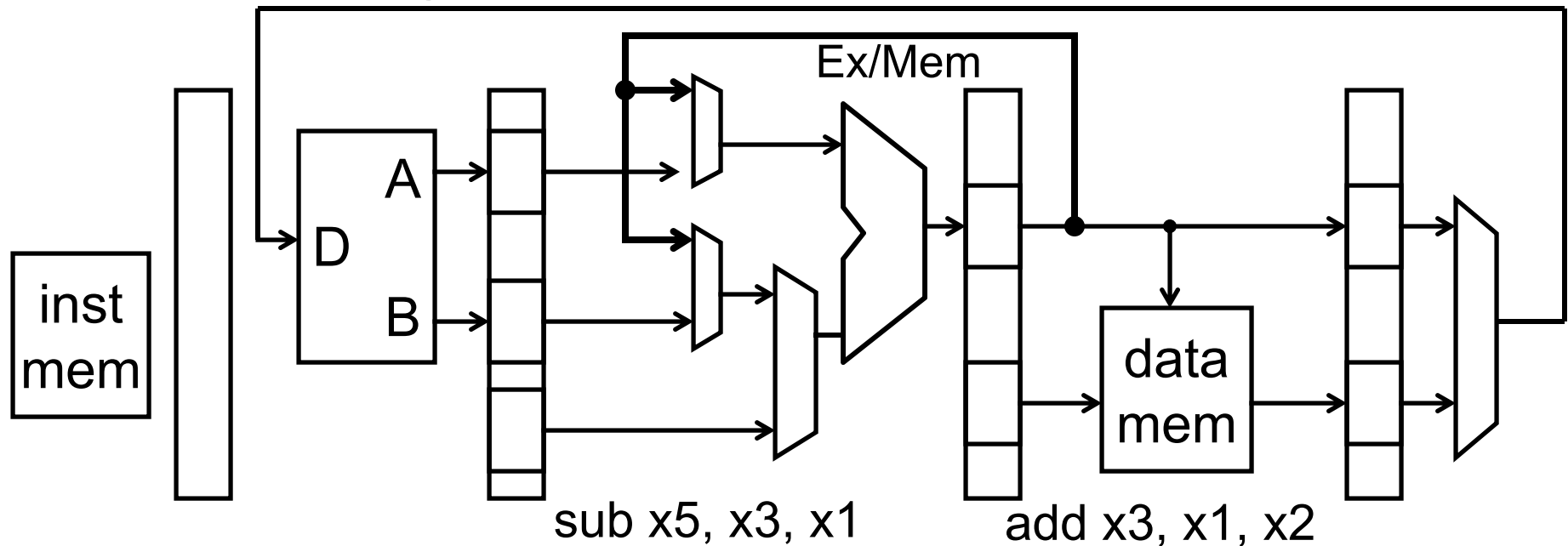# Forwarding Datapath

Three types of forwarding/bypass
- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
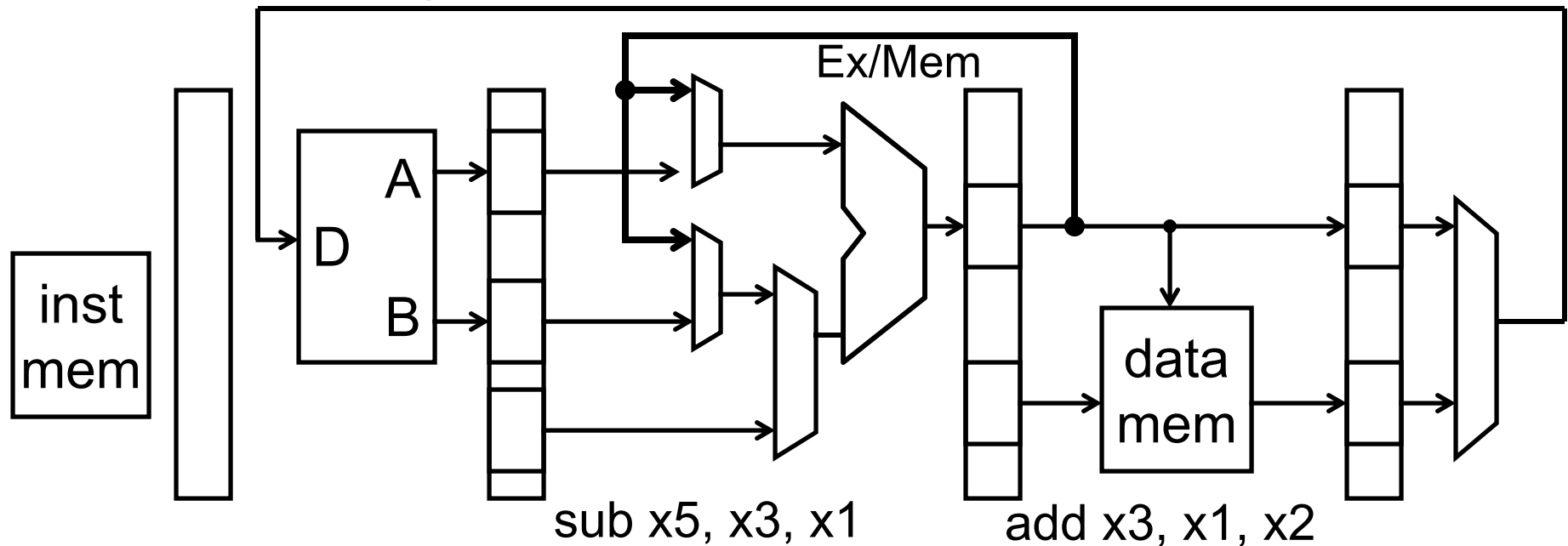- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

96

Forwarding Datapath 1: Ex/MEM → EX

sub x5, x3, x1

add x3, x1, x2

Ex/Mem

IF	ID	Ex	M	W		
	IF	ID	Ex	M	W	

add x3, x1, x2

sub x5, x3, x1

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 → EX

sub x5, x3, x1   add x3, x1, x2

Detection Logic in Ex Stage:

forward = (Ex/M.WE && EX/M.Rd != 0 &&

ID/Ex.Rs1 == Ex/M.Rd)

|| (same for Rs2)

Forwarding Datapath 2: Mem/WB→ EX
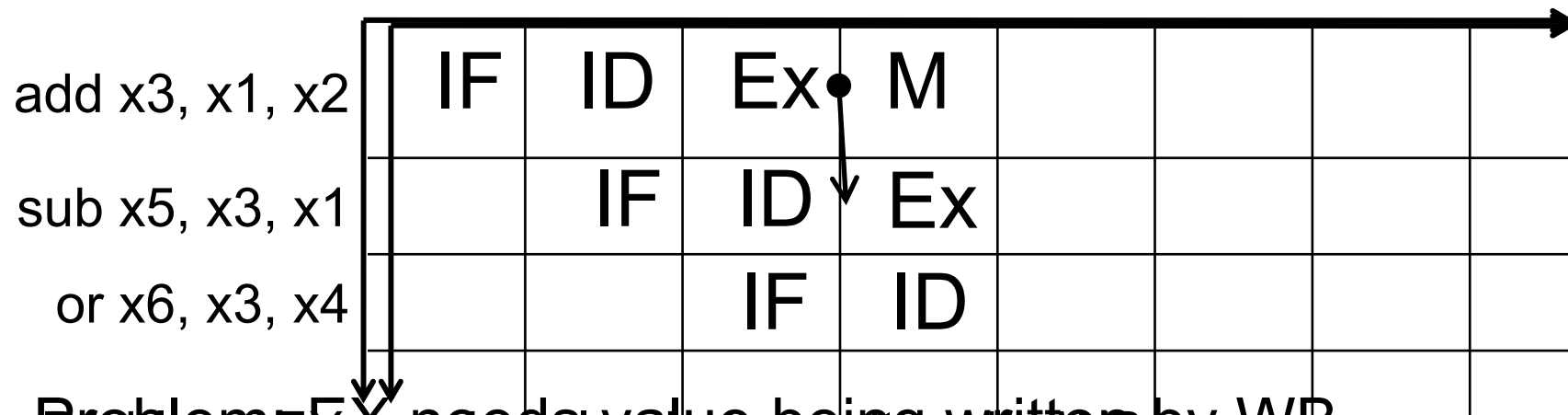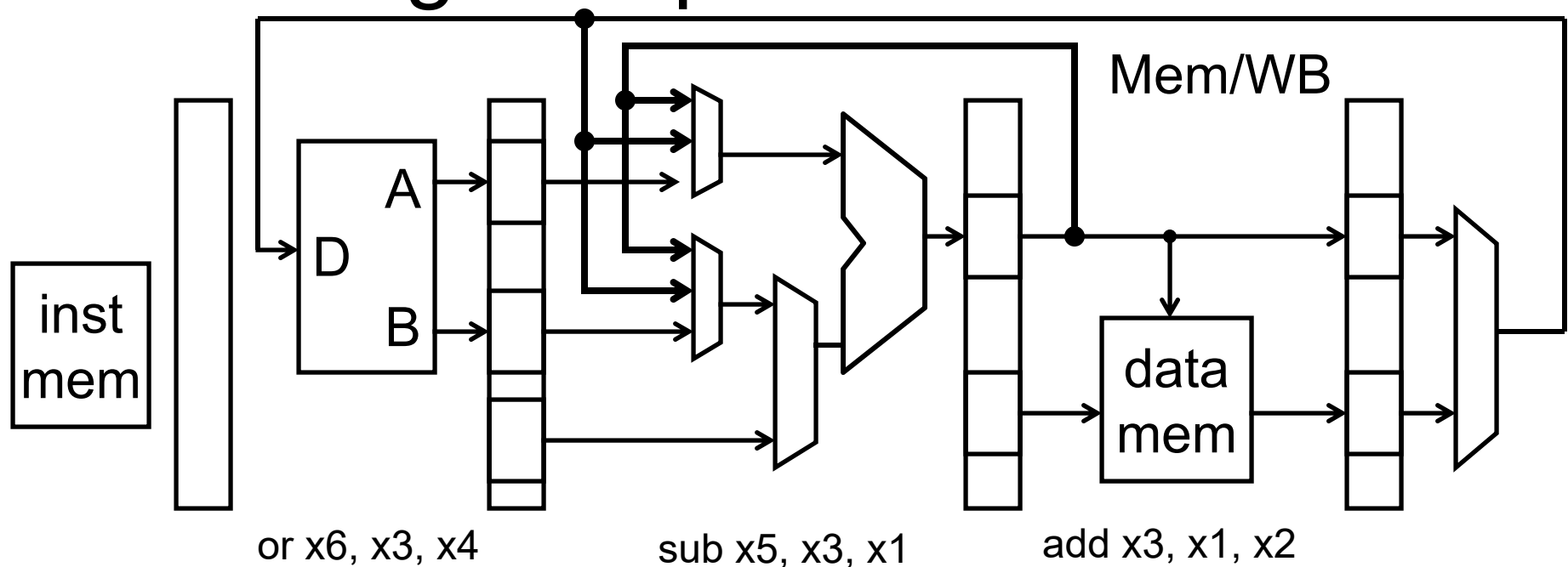
Mem/WB

or x6, x3, x4   sub x5, x3, x1   add x3, x1, x2

	IF	ID	Ex	M			
add x3, x1, x2	IF	ID	Ex	M			
sub x5, x3, x1		IF	ID	Ex			
or x6, x3, x4			IF	ID			

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

Mem/WB

or x6, x3, x4   sub x5, x3, x1   add x3, x1, x2

add x3, x1, x2	IF	ID	Ex	M	W		
sub x5, x3, x1		IF	ID	Ex	M	W	
or x6, x3, x4			IF	ID	Ex	M	W

Problem: EX needs value being written by WB

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

100

Forwarding Datapath 2: Mem/WB→ EX

Mem/WB

or x6, x3, x4   sub x5, x3, x1   add x3, x1, x2
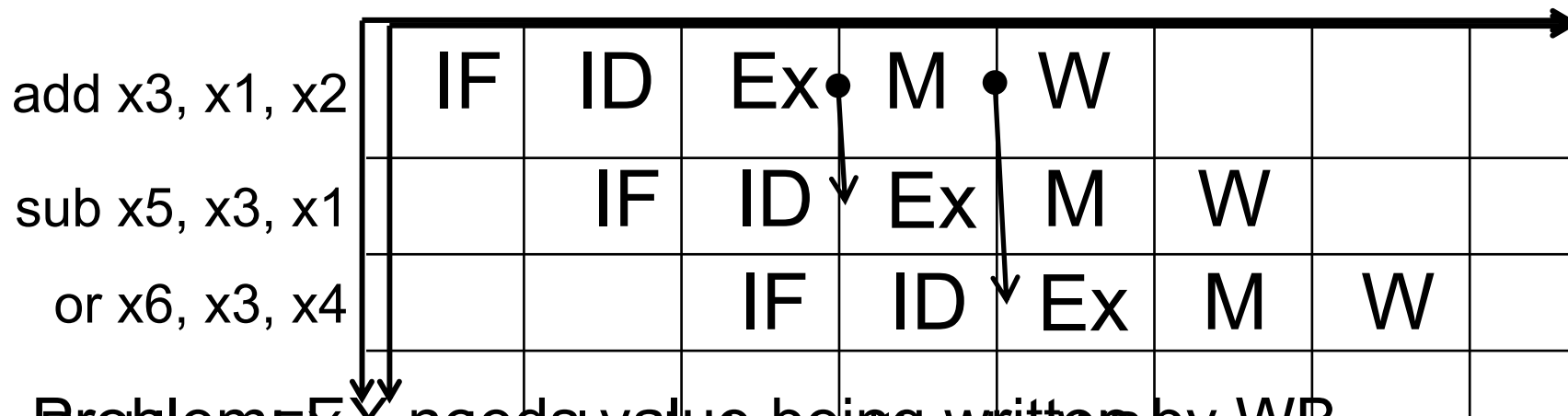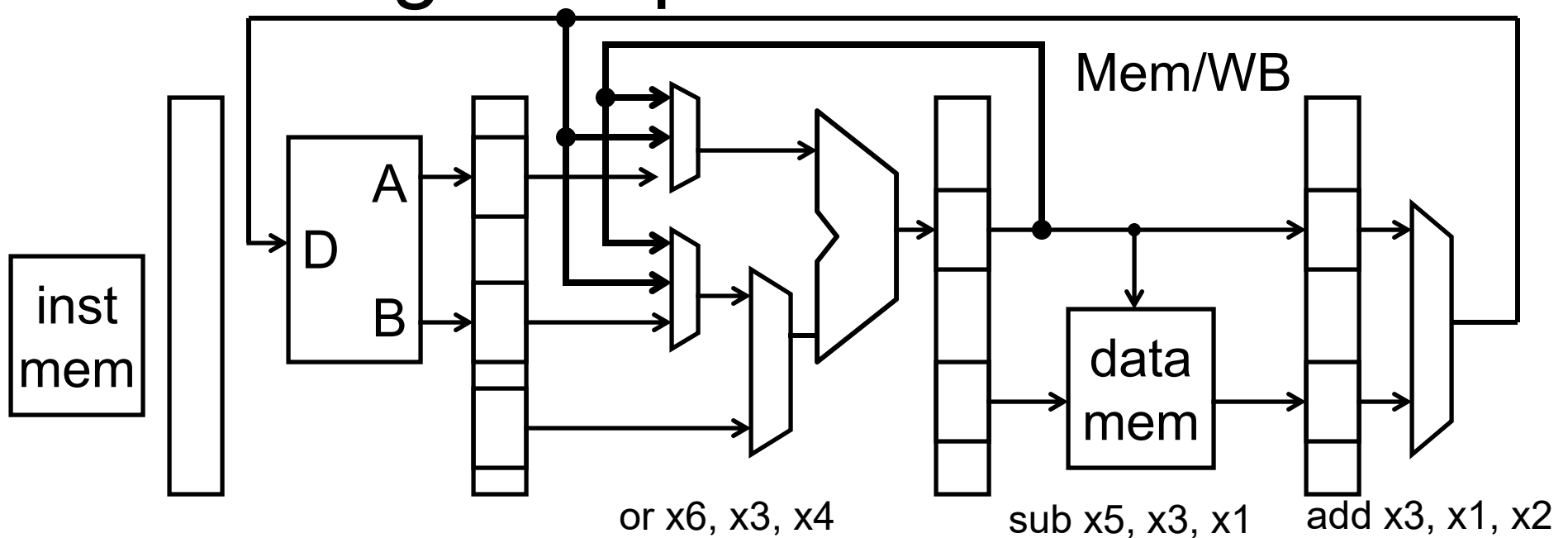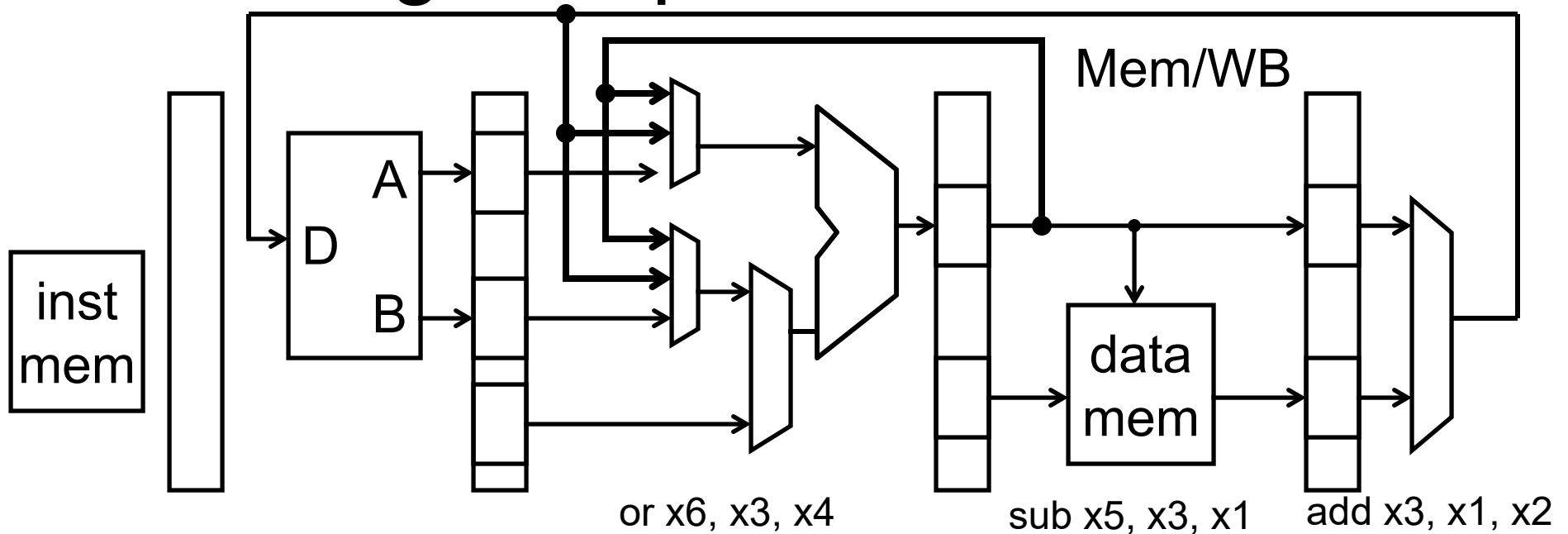
Detection Logic:

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

ID/Ex.Rs1 == M/WB.Rd &&

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

ID/Ex.Rs1 == Ex/M.Rd)

|| (same for Rs2)

Register File Bypass

add x6, x3, x8   or x6, x3, x4   sub x5, x3, x1 add x3, x1,x2
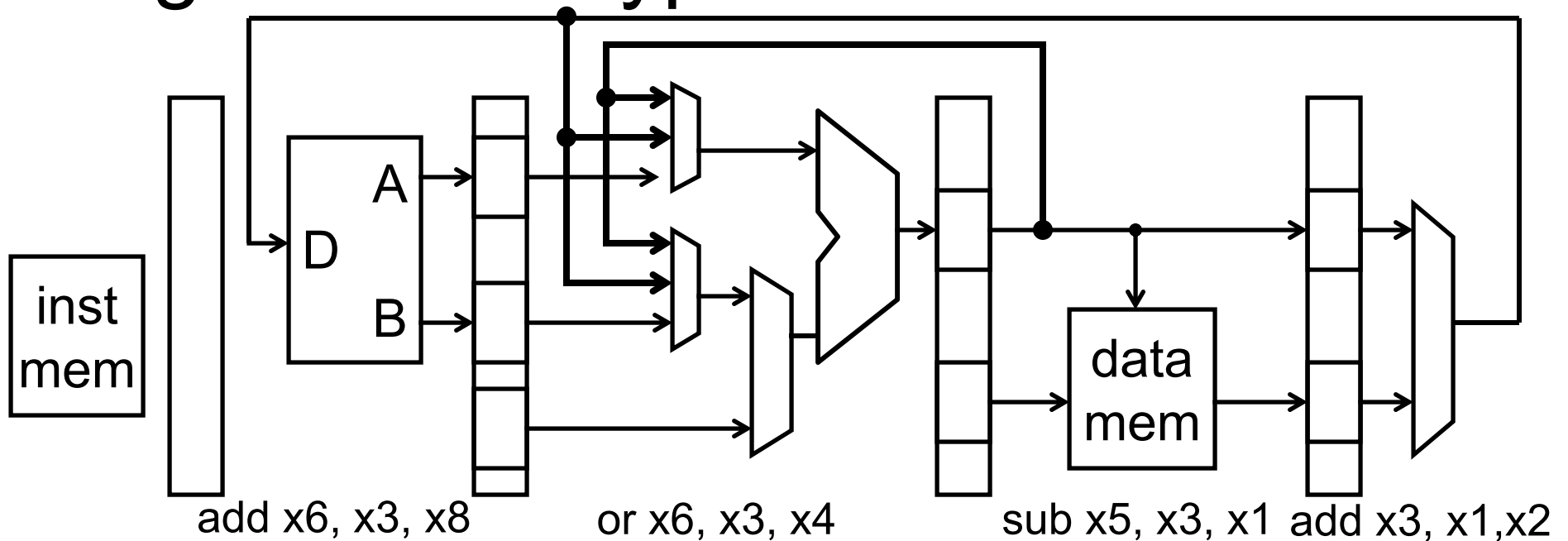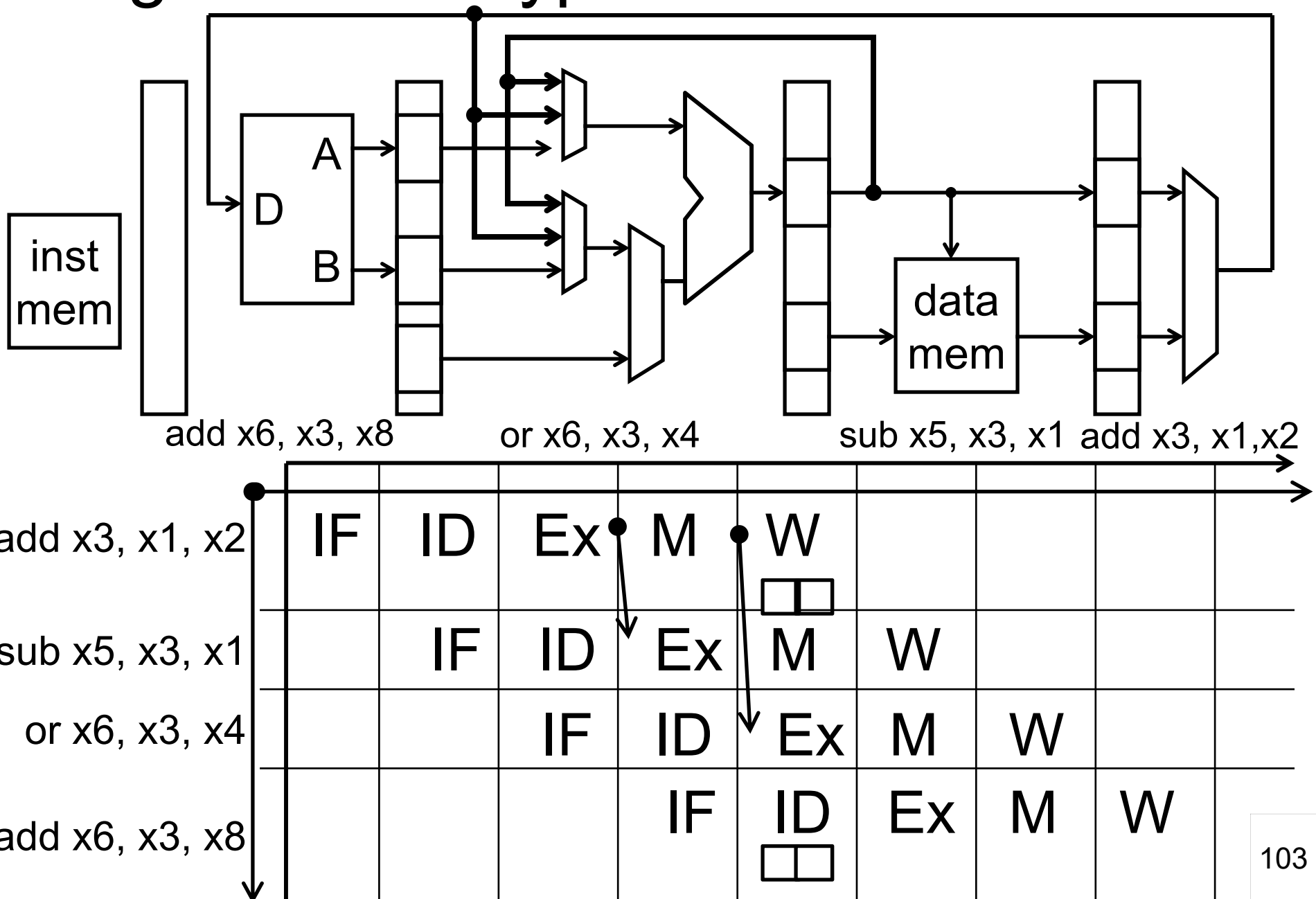
Problem: Reading a value that is currently being written

Solution: just negate register file clock

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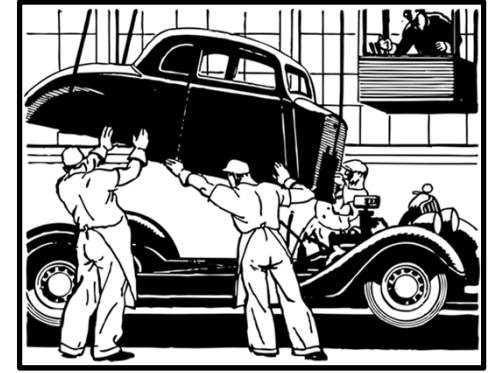- reads happen during second half of each clock cycle

Register File Bypass

add x6, x3, x8   or x6, x3, x4   sub x5, x3, x1   add x3, x1,x2

	IF	ID	Ex	M	W			
add x3, x1, x2	IF	ID	Ex	M	W			
sub x5, x3, x1		IF	ID	Ex	M	W		
or x6, x3, x4			IF	ID	Ex	M	W	
add x6, x3, x8				IF	ID	Ex	M	W

103
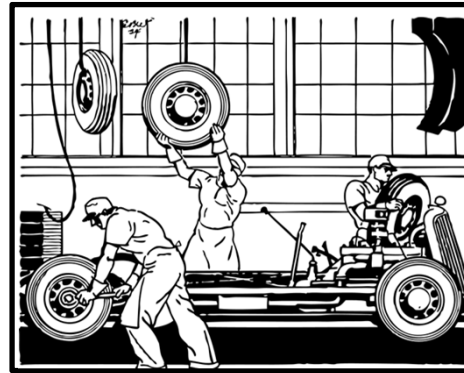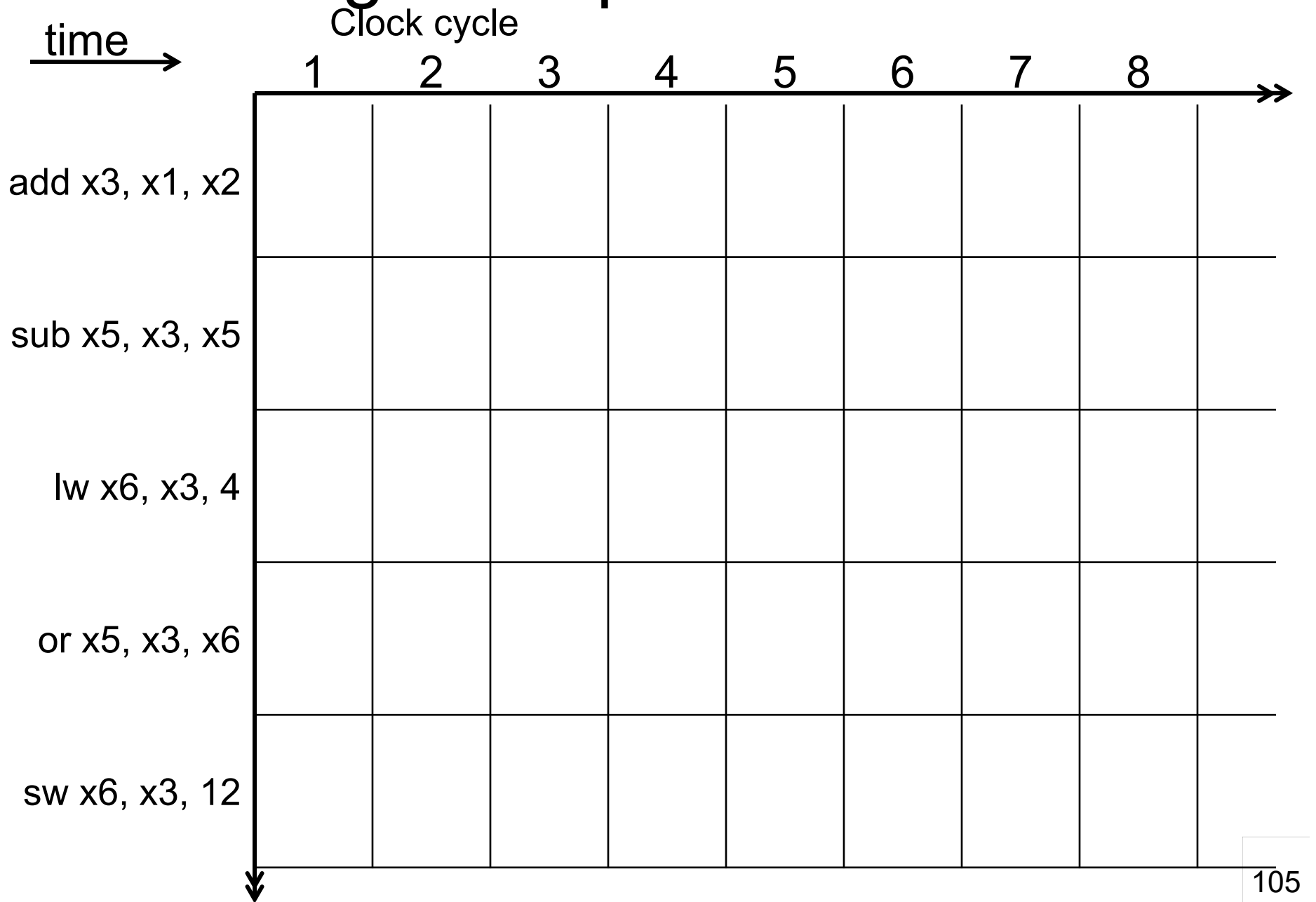
Agenda

5-stage Pipeline
- Implementation
- Working Example

Hazards
- Structural
- Data Hazards
- Control Hazards

Forwarding Example 2

time →

Clock cycle

	1	2	3	4	5	6	7	8
add x3, x1, x2								
sub x5, x3, x5								
lw x6, x3, 4								
or x5, x3, x6								
sw x6, x3, 12								

105

Forwarding Example 2

Clock cycle

	1	2	3	4	5	6	7	8
add x3, x1, x2	IF	ID	Ex	M	W			
sub x5, x3, x5		IF	ID	Ex	M	W		
lw x6, x3, 4			IF	ID	Ex	M	W	
or x5, x3, x6				IF	ID	Ex	M	W
sw x6, x3, 12					IF	ID	Ex	M	W

106

Forwarding Example 2

Clock cycle   *backwards arrows require time trave*

	1	2	3	4	5	6	7	8
add x3, x1, x2	IF	ID	Ex	M	W			
sub x5, x3, x5		IF	ID	Ex	M	W		
lw x6, x3, 4			IF	ID	Ex	M	W	
or x5, x3, x6				IF	ID	Ex	M	W
sw x6, x3, 12					IF	ID	Ex	M	W
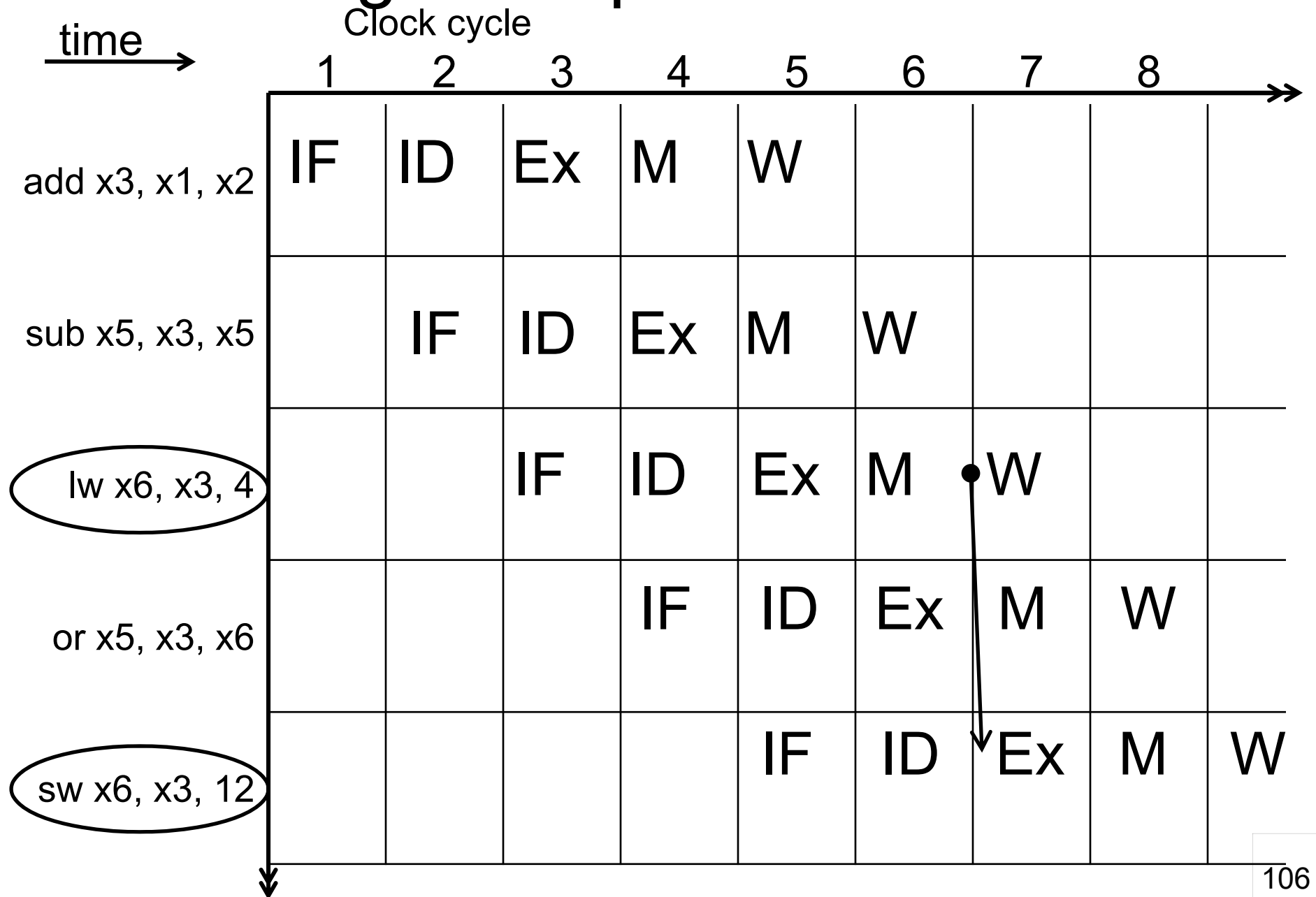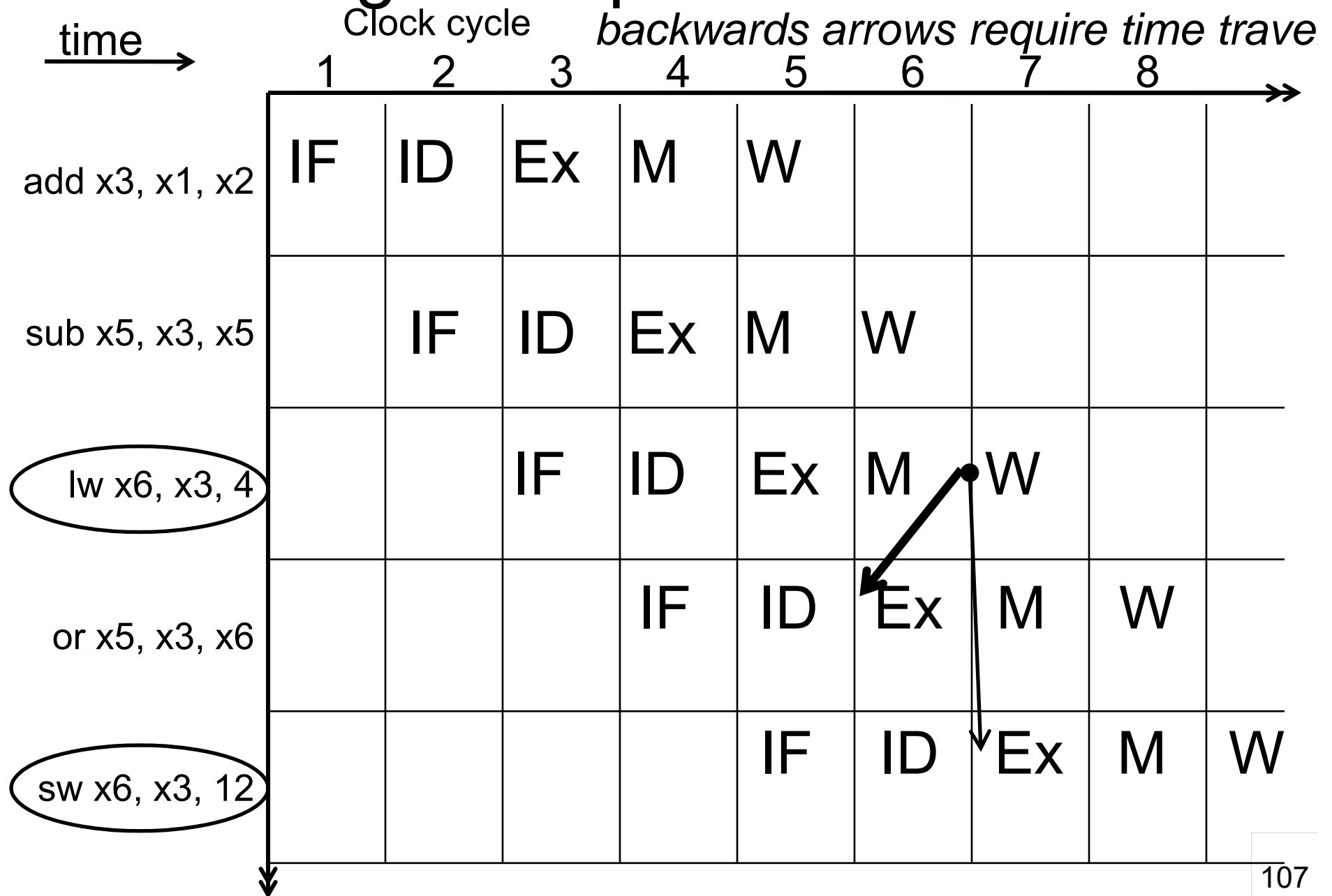
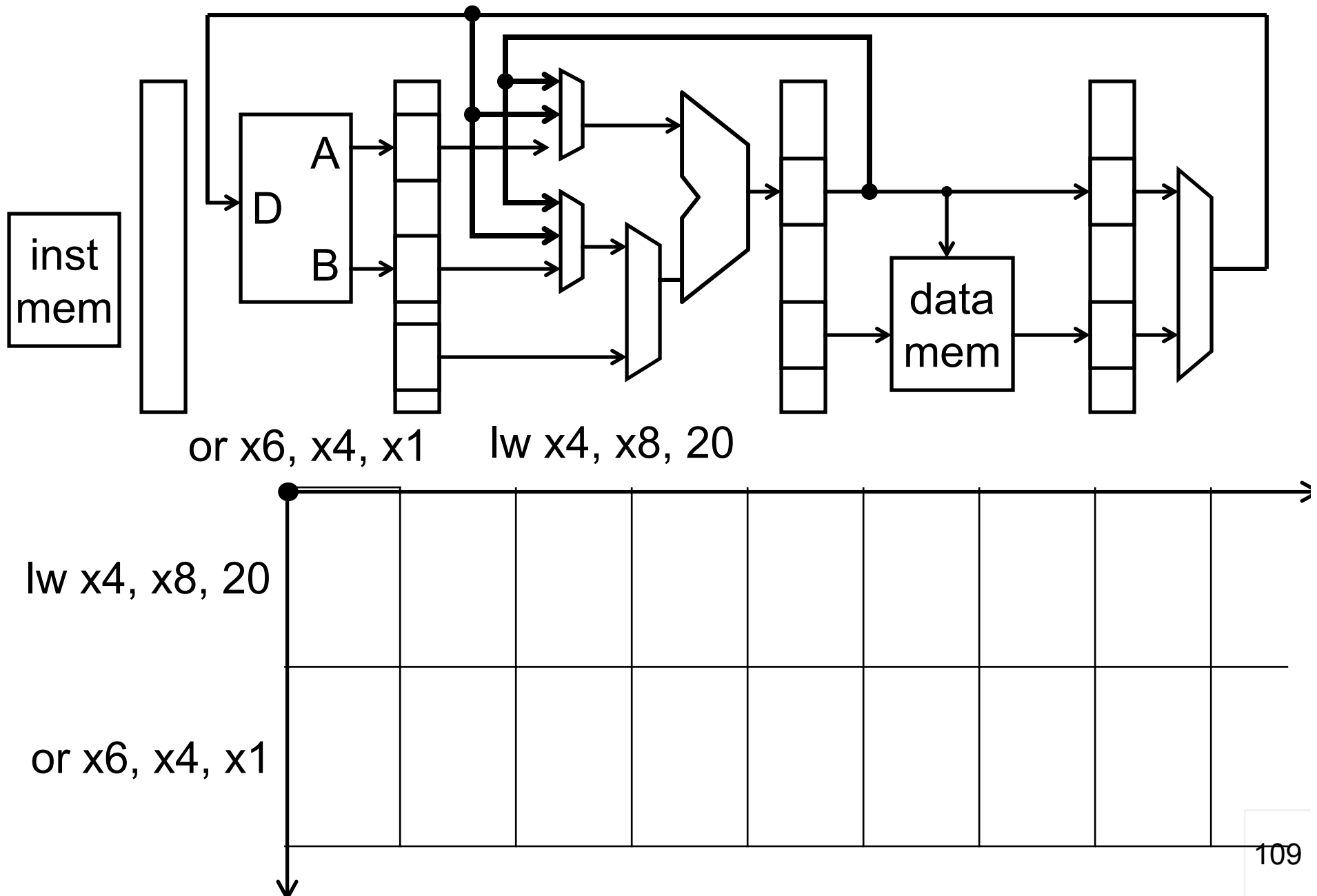# Load-Use Hazard Explained

or x5, x3, **x4**   lw **x4**, x8, 20
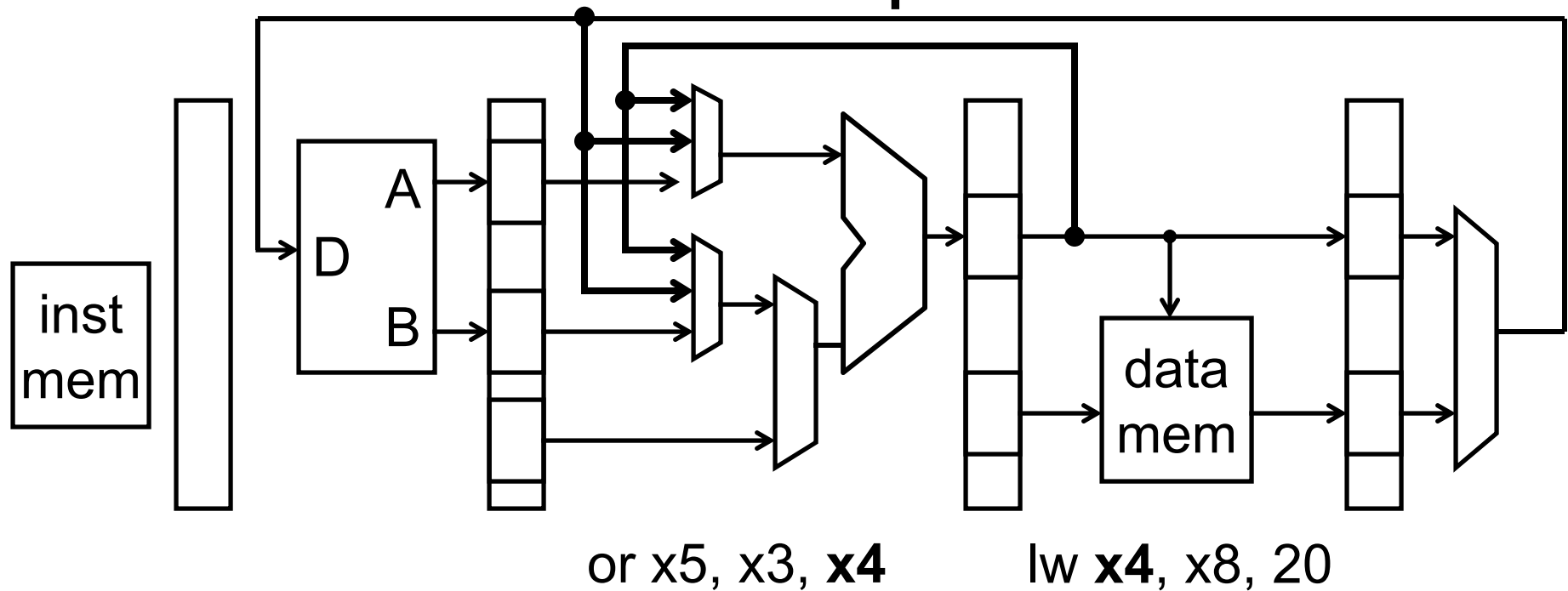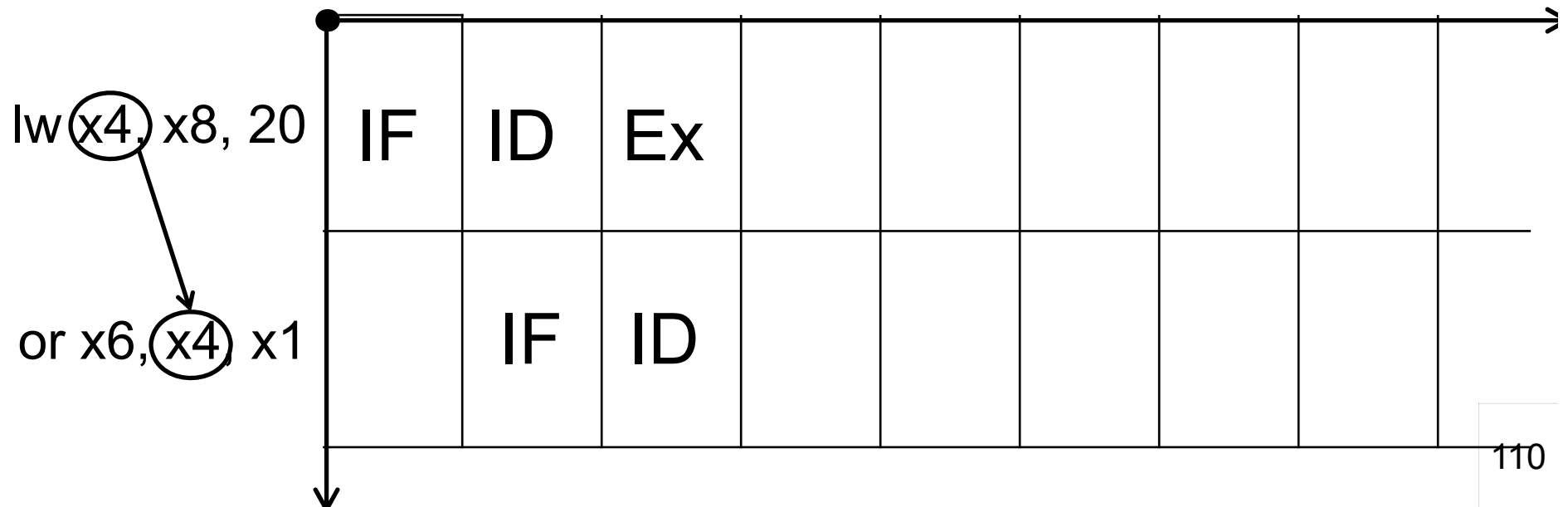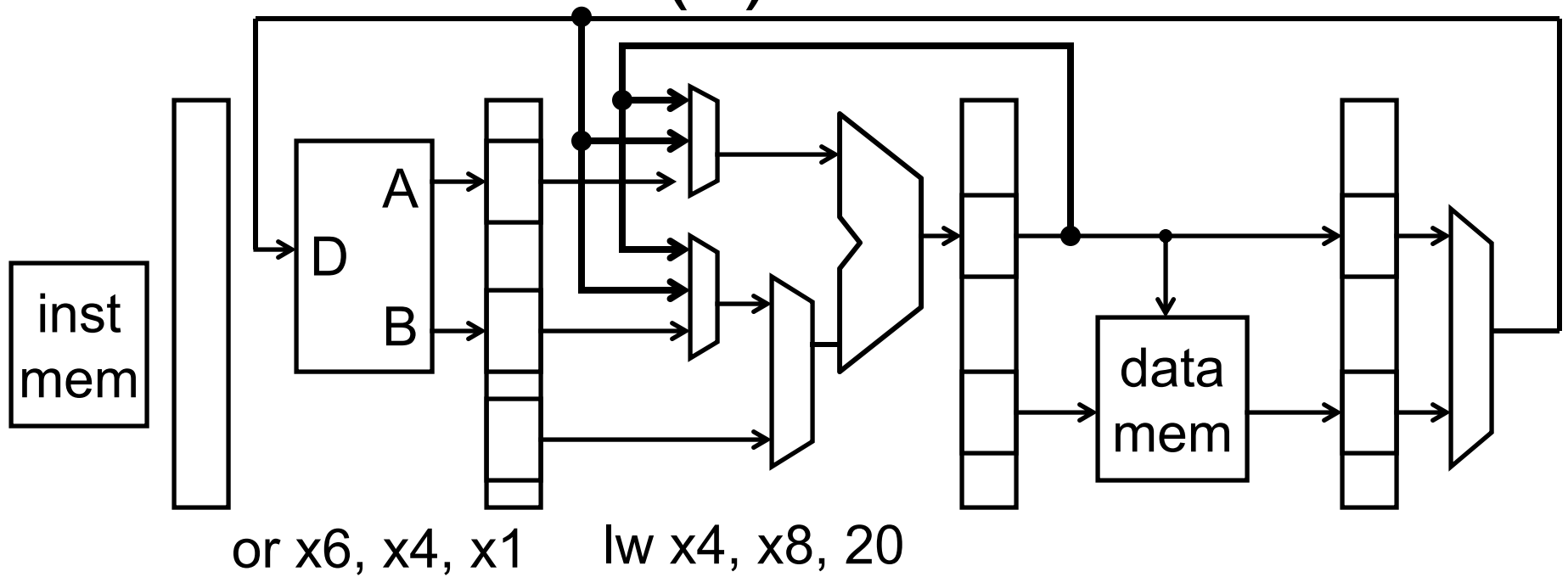
Data dependency after a load instruction:
- Value not available until *after* the M stage

→Next instruction cannot proceed if dependent

THE KILLER HAZARD

Load-Use Stall

inst mem

D A B

or x6, x4, x1   lw x4, x8, 20

data mem

lw x4, x8, 20

or x6, x4, x1

Load-Use Stall (1)

or x6, x4, x1   lw x4, x8, 20

IF	ID	Ex						
	IF	ID						

lw (x4) x8, 20

or x6,(x4) x1

Load-Use Stall (2)

or x6, x4, x1   **NOP**   lw x4, x8, 20

	IF	ID	Ex	M	W				
lw (x4) x8, 20	IF	ID	Ex	M	W				
or x6,(x4) x1		IF	ID*	Stall (ID)	Ex	M	W		

111

Load-Use Stall (3)

or x6, x4, x1   **NOP**   lw x4, x8,

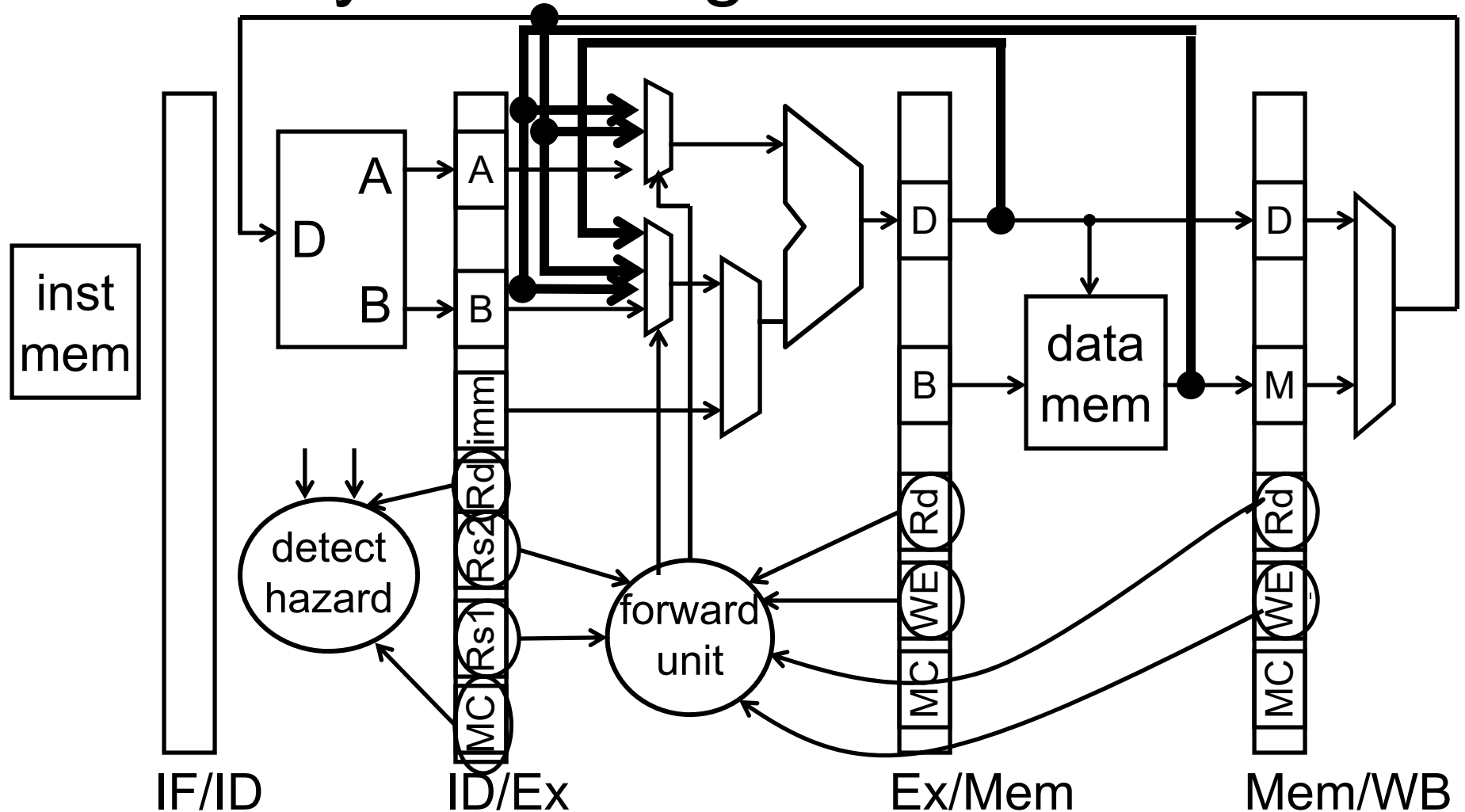| lw x4, x8, 20	IF	ID	Ex	M	W			
or x6, x4, x1		IF	ID*	Stall ID	Ex	M	W	
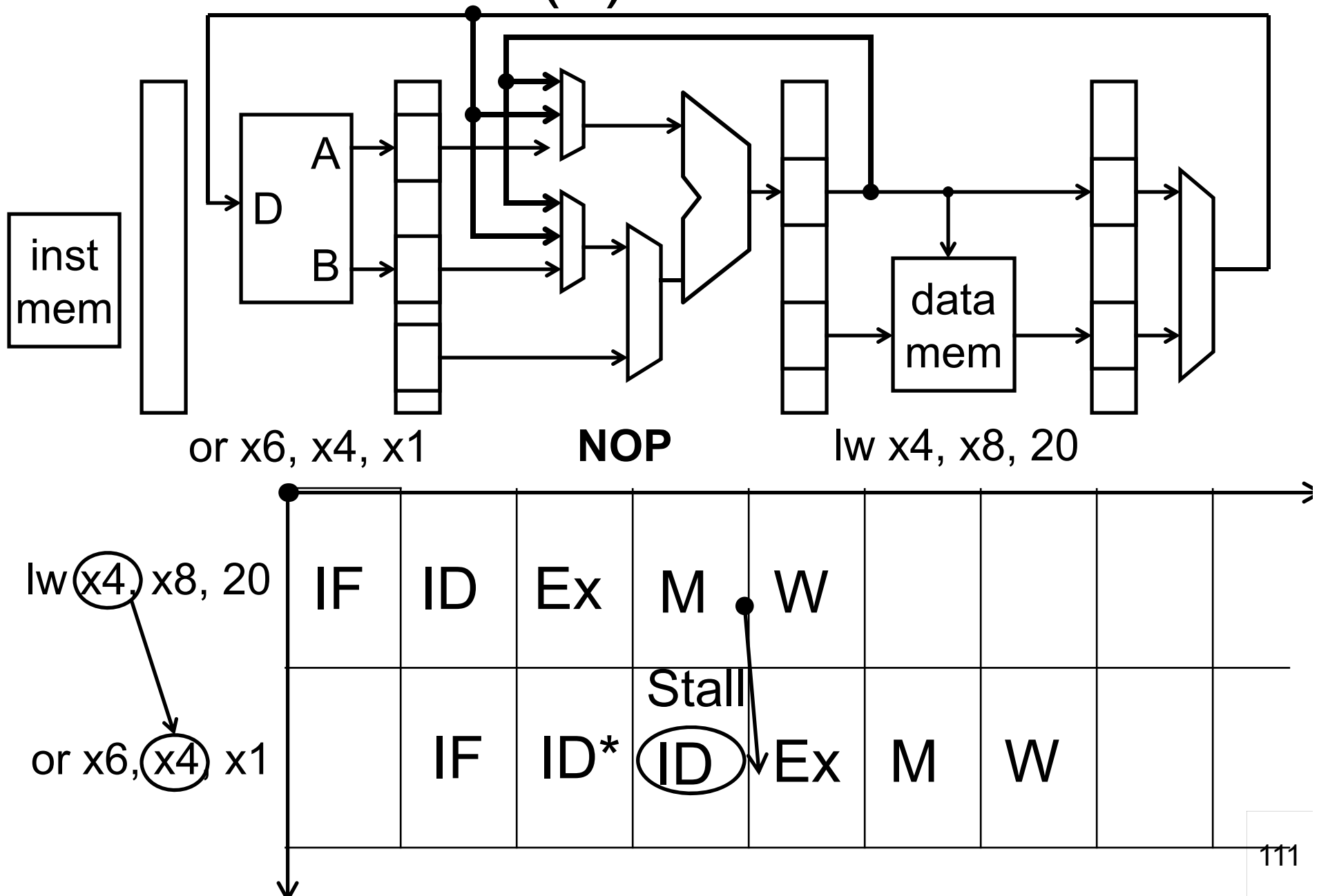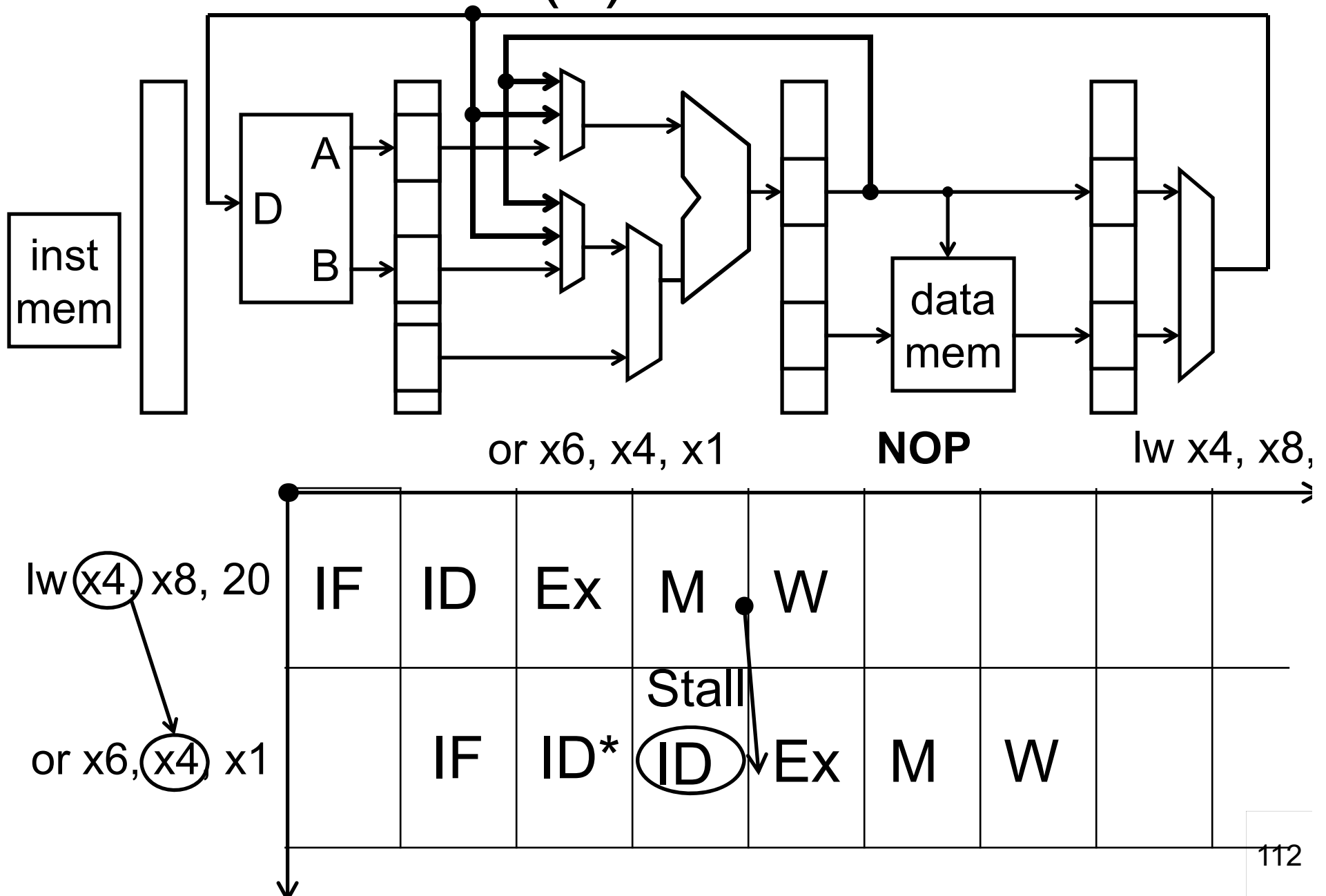
Load-Use Detection
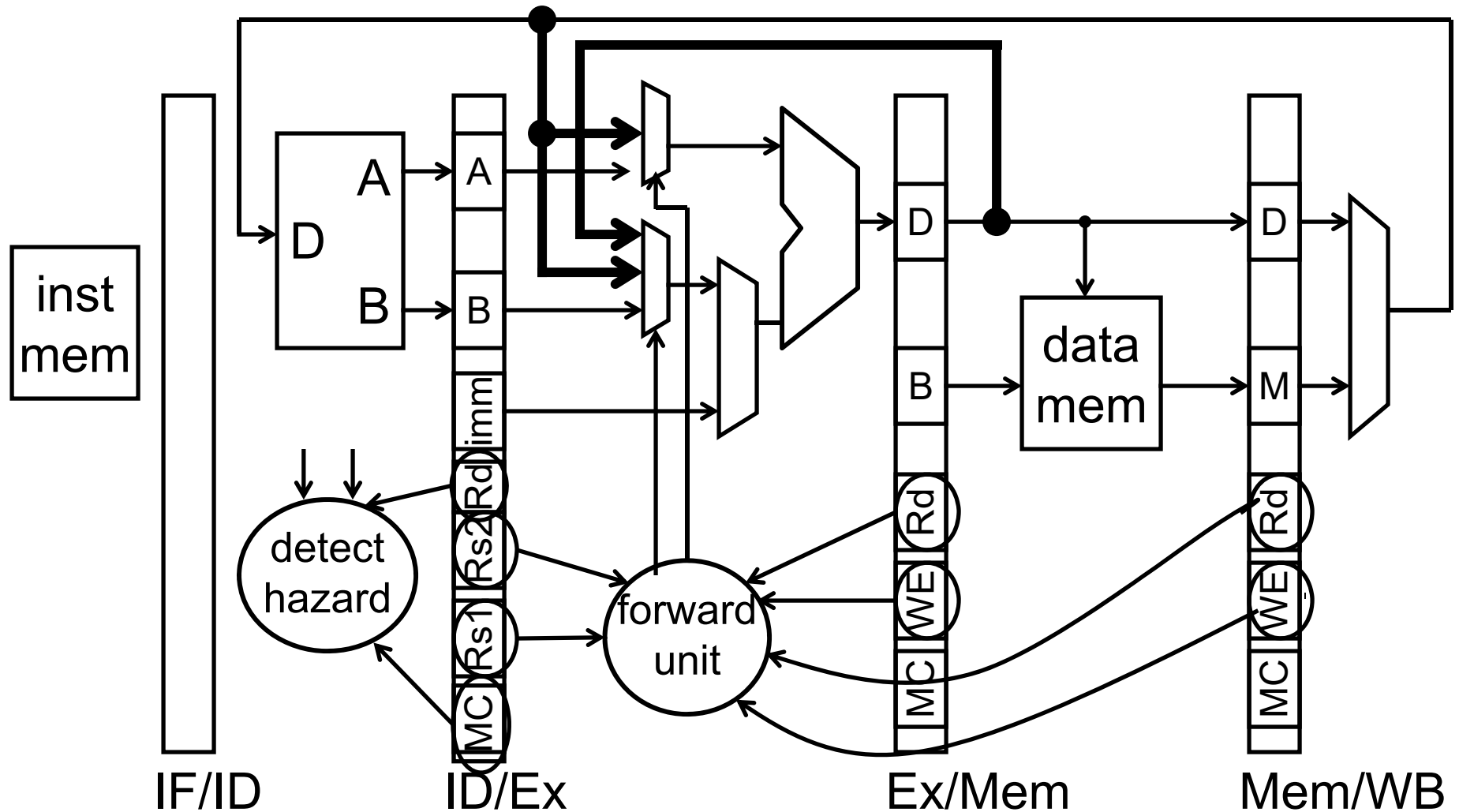
Stall =   If(ID/Ex.MemRead &&
          IF/ID.Rs1 == ID/Ex.Rd

113

Incorrectly Resolving Load-Use Hazards

inst mem

IF/ID   ID/Ex   Ex/Mem   Mem/WB

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

RISC-V Solution : Load-Use Stall

- Stall must be inserted so that load instruction can go through and update the register file.
- Forwarding from RAM is not an option.
- In some cases, real world compilers can optimize to avoid these situations.

Takeaway

Data hazards occur when a 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.
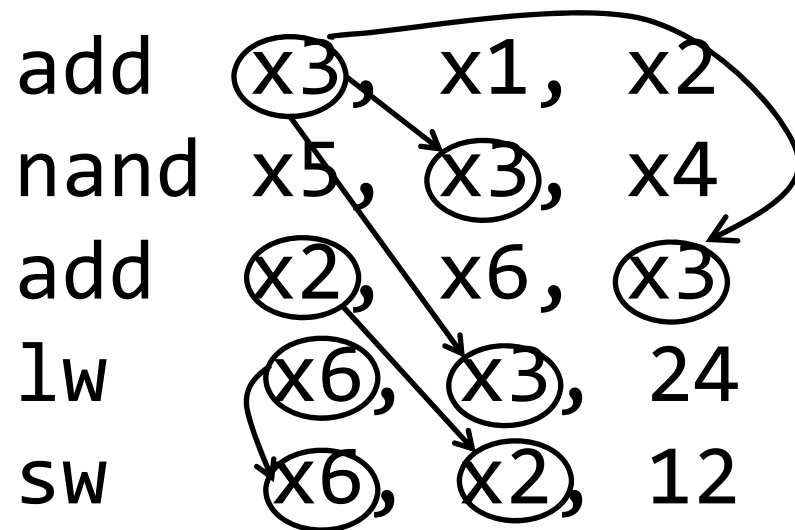
Quiz

Find all hazards, and say how they are resolved:

```
add   x3, x1, x2
nand  x5, x3, x4
add   x2, x6, x3
lw    x6, x3, 24
sw    x6, x2, 12
```
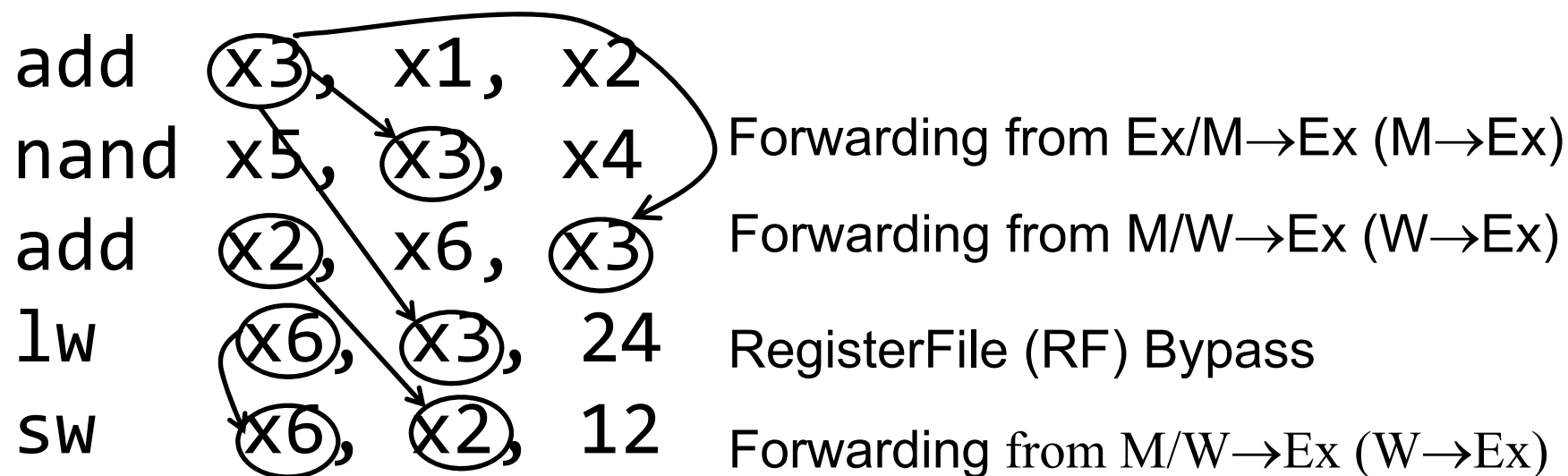
Quiz

Find all hazards, and say how they are resolved:

```
add   x3,  x1,  x2
nand  x5,  x3,  x4
add   x2,  x6,  x3
lw    x6,  x3,  24
sw    x6,  x2,  12
```

5 Hazards

Quiz

Find all hazards, and say how they are resolved:

```
add   x3,  x1,  x2
nand  x5,  x3,  x4        Forwarding from Ex/M→Ex (M→Ex)
add   x2,  x6,  x3        Forwarding from M/W→Ex (W→Ex)
lw    x6,  x3,  24        RegisterFile (RF) Bypass
sw    x6,  x2,  12        Forwarding from M/W→Ex (W→Ex)
```

Stall
+ Forwarding from M/W→Ex (W→Ex)

5 Hazards

Quiz

Find all hazards, and say how they are resolved:

```
add    x3, x1, x2
sub    x3, x2, x1
nand   x4, x3, x1
or     x0, x3, x4
xor    x1, x4, x3
sb     x4, x0, 1
```
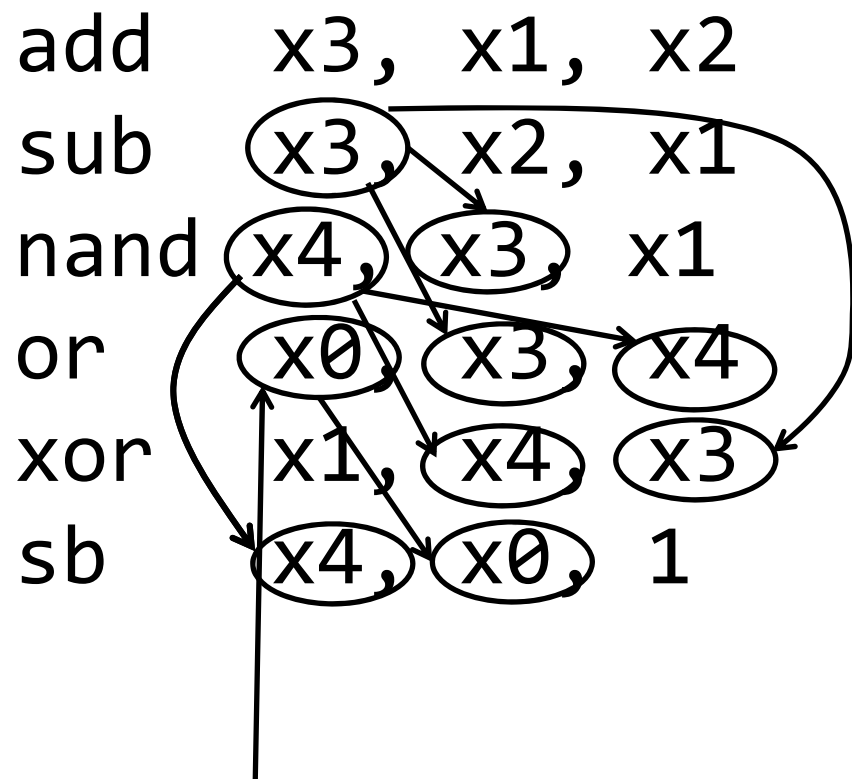
Hours and hours of debugging!

Data Hazard Recap

Delay Slot(s)

- Modify ISA to match implementation

Stall

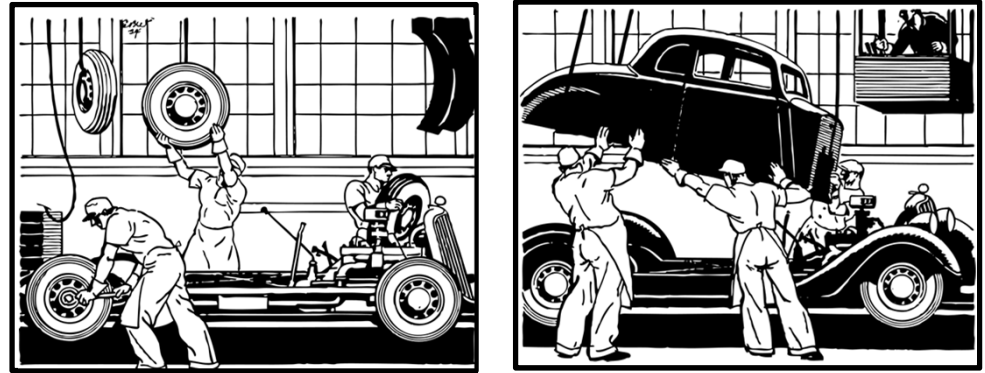- Pause current and all subsequent instructions

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 → x1
Assume:
n → x2
max → x3

x10		addi x1, x0, 0	# i=0
x14	Loop:	addi x2, x2, 2	# n += 2
x18		addi x1, x1, 1	# i++
x1C		blt x1, x3, Loop	# i<max?
x20		addi x1, x0, 7	# i = 7
x24		subi x2, x2, 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  x1, x3, Loop
x20     addi x1, x0, 7
x24     subi x2, x2, 1
```

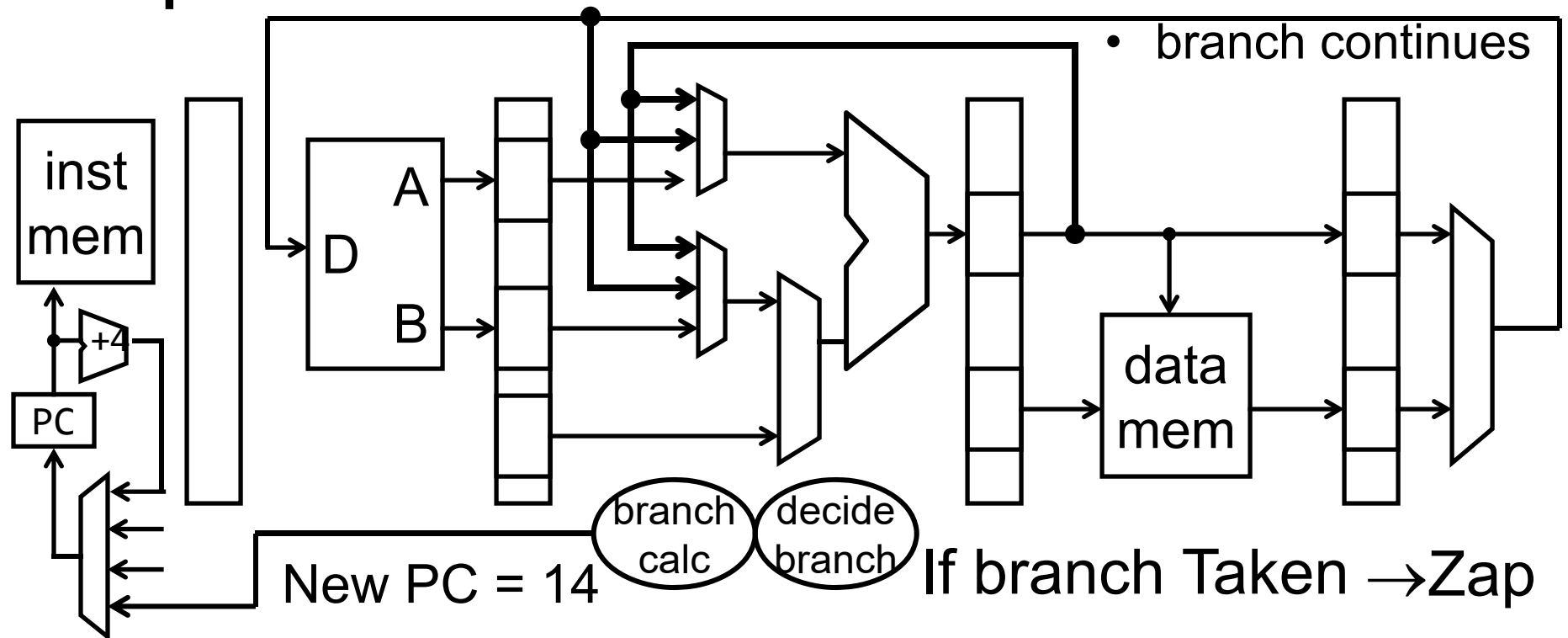Branch **<u>not</u>** taken?
  No Problem!
Branch taken?
 Just fetched 2 insns
  → **Zap & Flush**

Zap & Flash

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

inst mem

+4

PC

D A B

branch calc

decide branch

New PC = 14

If branch Taken →Zap

data mem

	IF	ID	Ex	M	W		
1C blt x1,x3,L	IF	ID	Ex	M	W		
20 addi x1,x0,7		IF	ID	NOP	NOP	NOP	
24 subi x2,x2,1			IF	NOP	NOP	NOP	NOP
14 L:addi x2,x2,2				IF	ID	Ex	M	W

127

Zap & Flash

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

inst mem

+4

PC

D

A

B

branch calc

decide branch

New PC = 14

If branch Taken →Zap

data mem

	IF	ID	Ex	M	W					
1C blt x1,x3,L	IF	ID	Ex	M	W					
20 addi x1,x0,7		IF	ID	NOP	NOP	NOP				
24 subi x2,x2,1			IF	NOP	NOP	NOP	NOP			
14 L:addi x2,x2,2				IF	ID	Ex	M	W		

For every taken branch? OUCH!!!
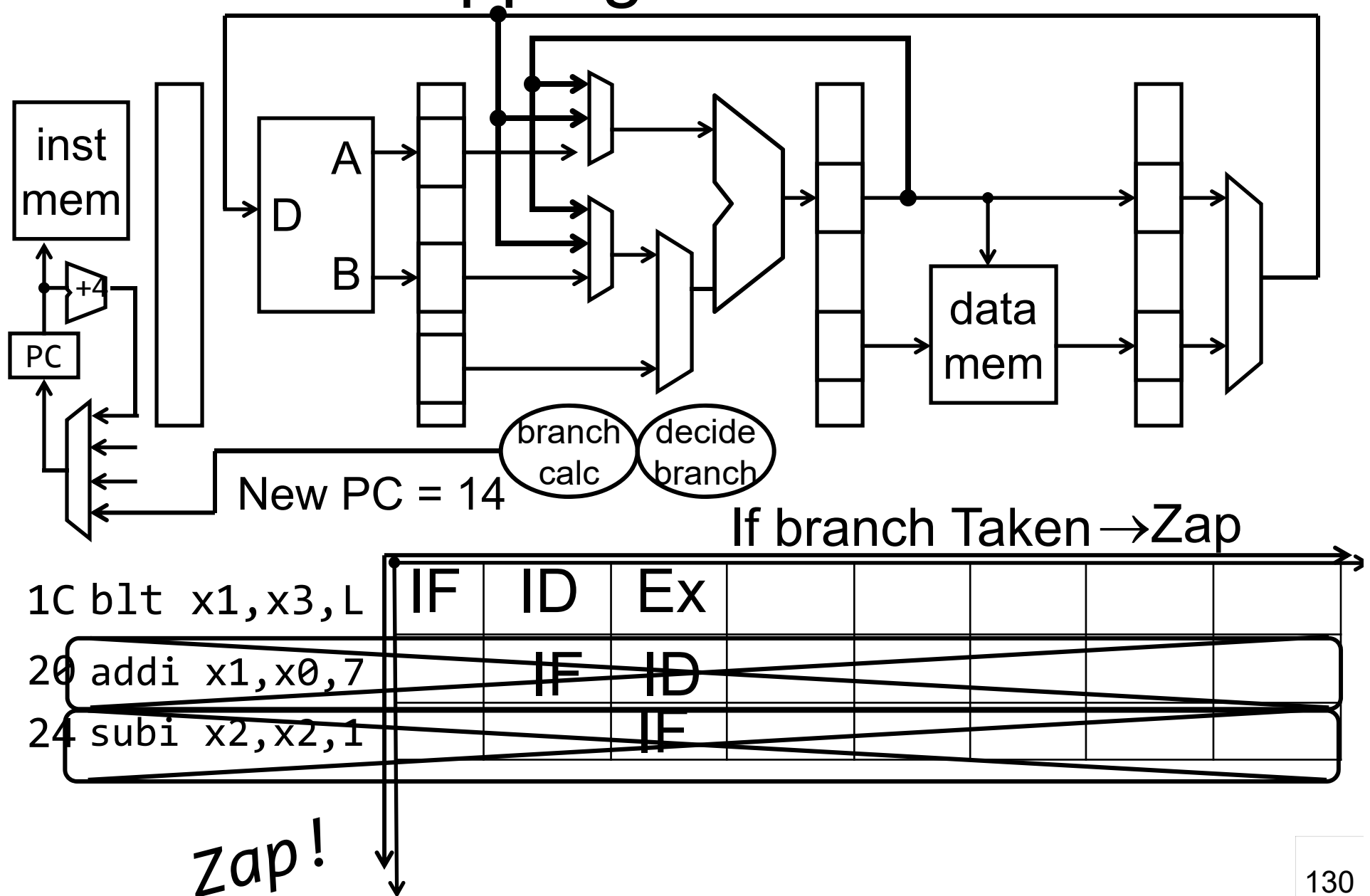
128
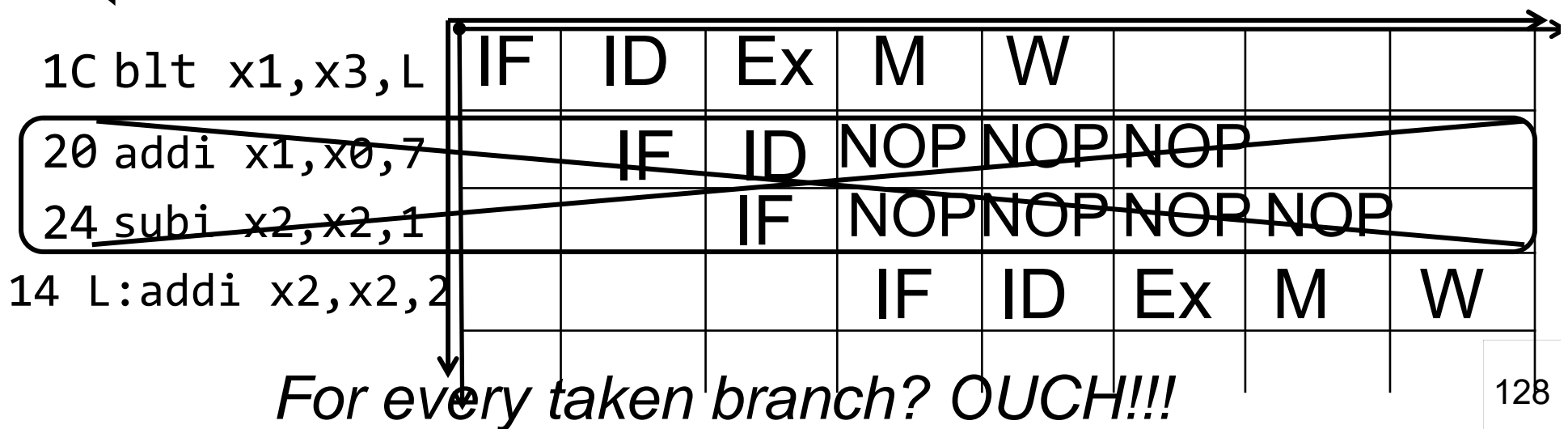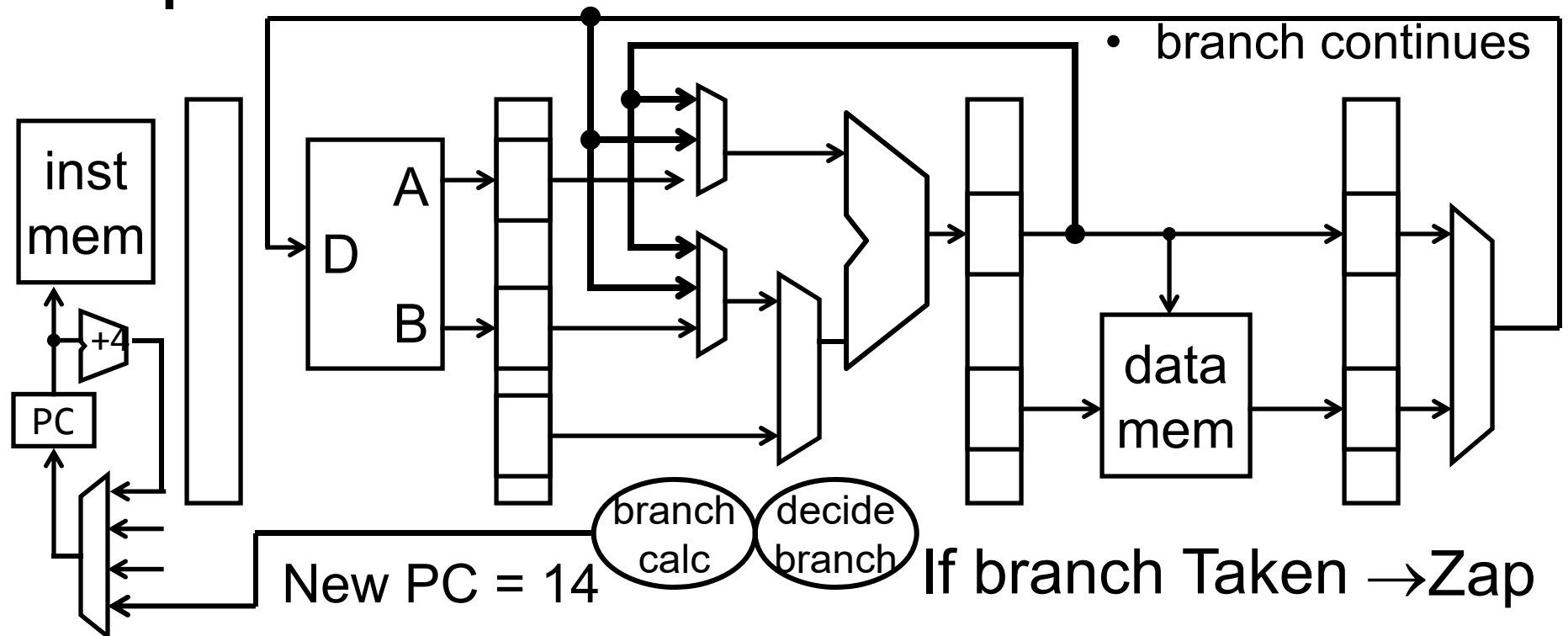
Reducing the cost of control hazard

1. **Resolve Branch at Decode**
   - Some groups do this for Project 3, your choice
   - Move branch calc from EX to ID
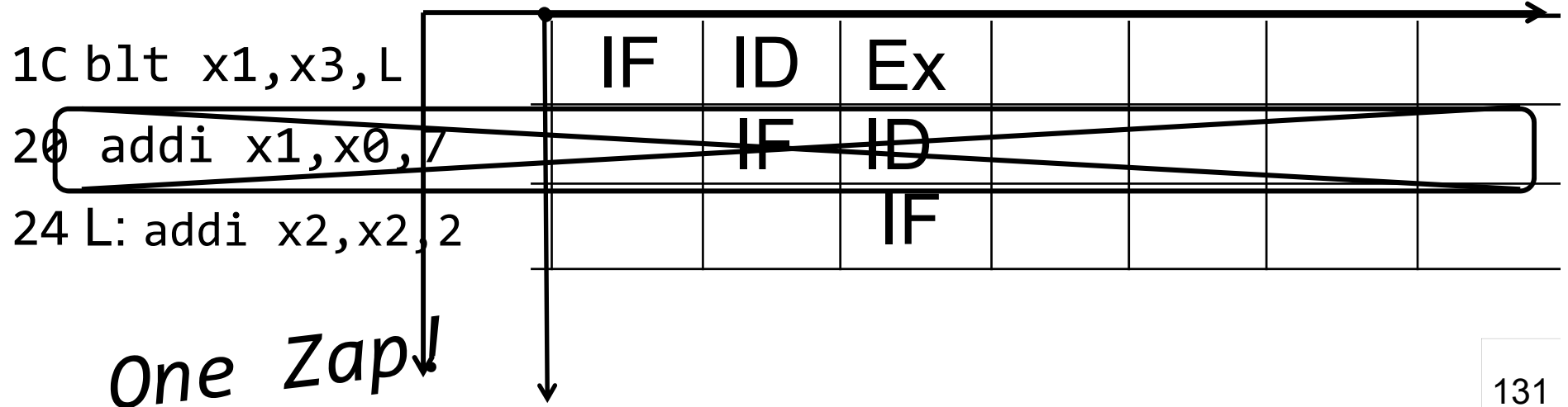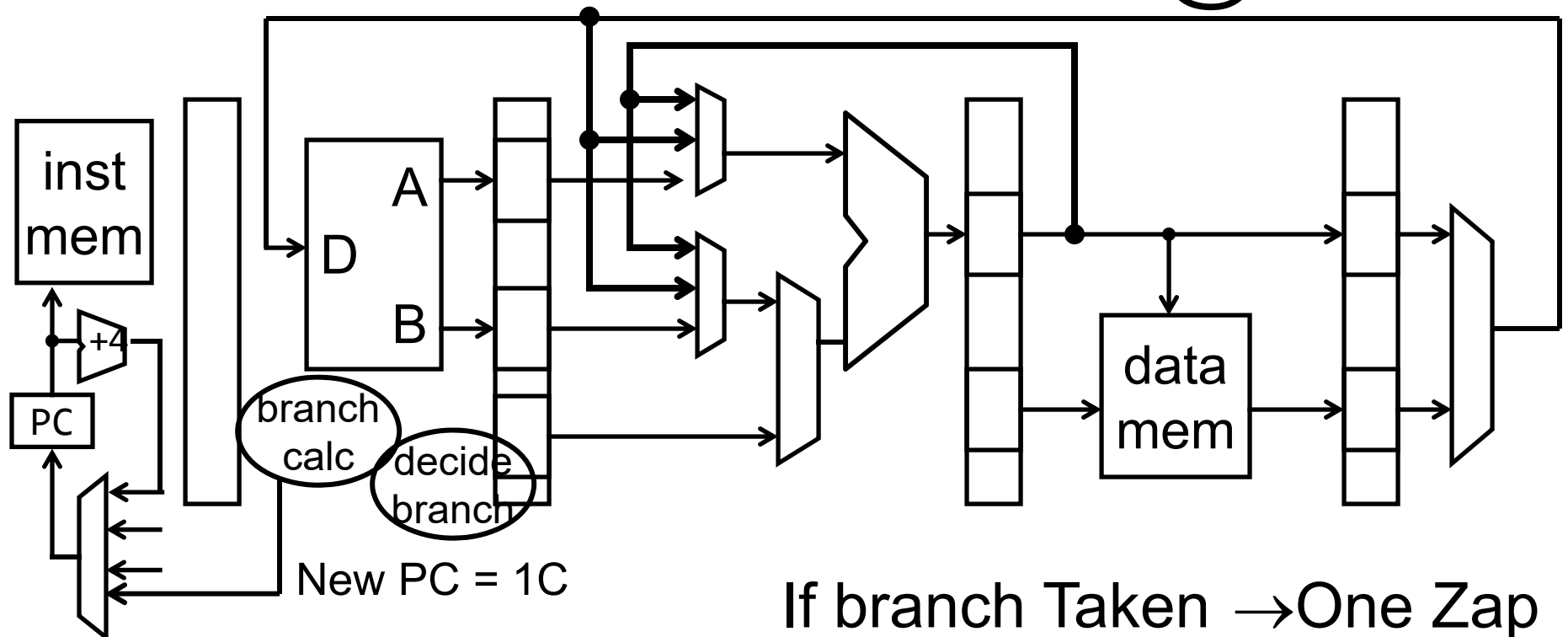   - Alternative: just zap 2nd instruction when branch taken

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

Problem: Zapping 2 insns/branch

New PC = 14

branch calc | decide branch

If branch Taken→Zap

	IF	ID	Ex					
1C blt x1,x3,L	IF	ID	Ex					
20 addi x1,x0,7		IF	ID					
24 subi x2,x2,1			IF					

Zap!

130

Soln #1: Resolve Branches @ Decode

inst mem

+4

PC

D

A

B

branch calc

decide branch

New PC = 1C

data mem

If branch Taken →One Zap

	IF	ID	Ex					
1C blt x1,x3,L								
20 addi x1,x0,7		IF	ID					
24 L: addi x2,x2,2			IF					

One Zap!

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 (x3 ≠ 0) {…. x3--;}
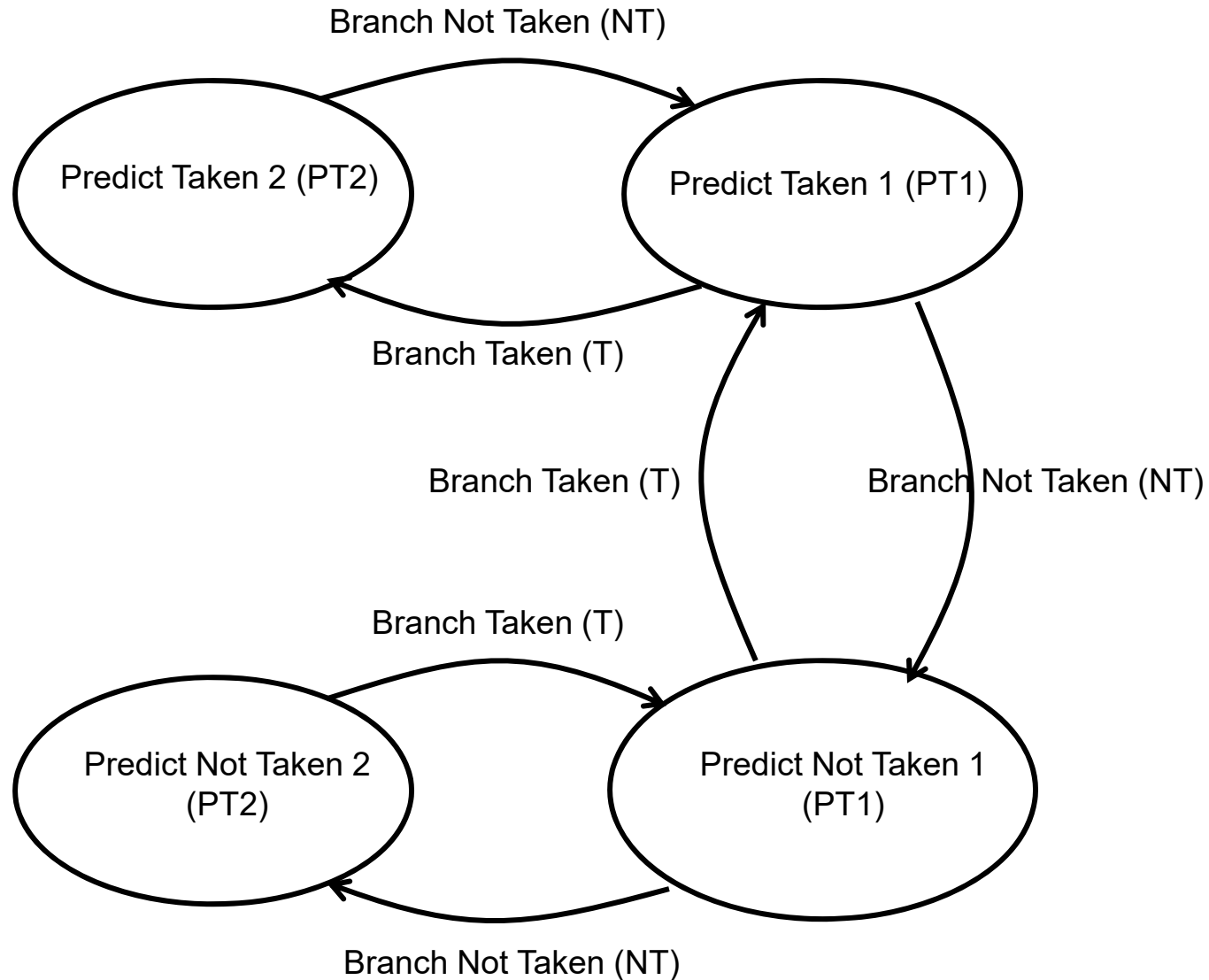Top: BEQ x3, x0, End

          J Top

End:

While (r3 ≠ 0) {…. r3--;}
Top2: BEQ x3, x0, End2

          J Top

End2:

Speculative Execution: Branch Execution

Branch Not Taken (NT)

Predict Taken 2 (PT2)

Predict Taken 1 (PT1)

Branch Taken (T)

Branch Taken (T)

Branch Not Taken (NT)

Branch Taken (T)

Predict Not Taken 2 (PT2)

Predict Not Taken 1 (PT1)

Branch Not Taken (NT)

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

- 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 a 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.
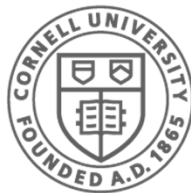
Control Hazard Takeaways

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.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage.

Have a great February Break!!