Pipelining and Hazards

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 4.6-4.8

Announcements

Prelim next week

- Tuesday at 7:30.

- Go based on netid

Prelim reviews

- Friday and Sunday evening. 7:30 again.
- Location: TBA on piazza

Prelim conflicts

- Contact KB , Prof. Weatherspoon, Andrew Hirsch

Survey

- Constructive feedback is very welcome

Administrivia

Prelim1:

- Time: We will start at 7:30pm sharp, so come early
- Loc: Upson B17 [a-e]*, Olin 255[f-m]*, Philips 101 [n-z]*
- Closed Book
  - Cannot use electronic device or outside material
- Practice prelims are online in CMS

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

Pipelining

Principle:

Throughput increased by parallel execution

Balanced pipeline very important

Else slowest stage dominates performance

Pipelining:

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

Basic Pipeline

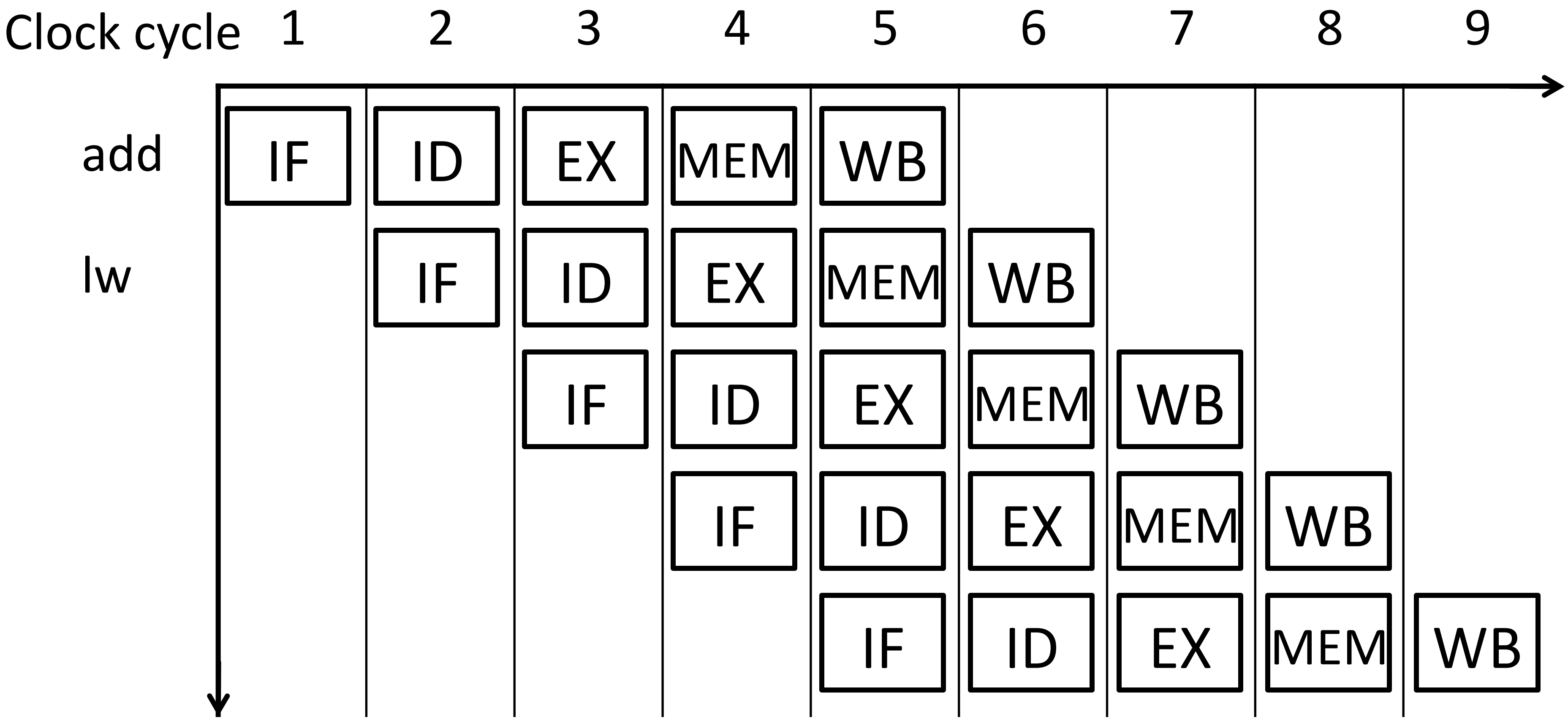
Five stage "RISC" load-store architecture

1. Instruction fetch (IF)
   - get instruction from memory, increment PC
2. Instruction Decode (ID)
   - translate opcode into control signals and read registers
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   - update register file

Pipelined Implementation

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

Time Graphs

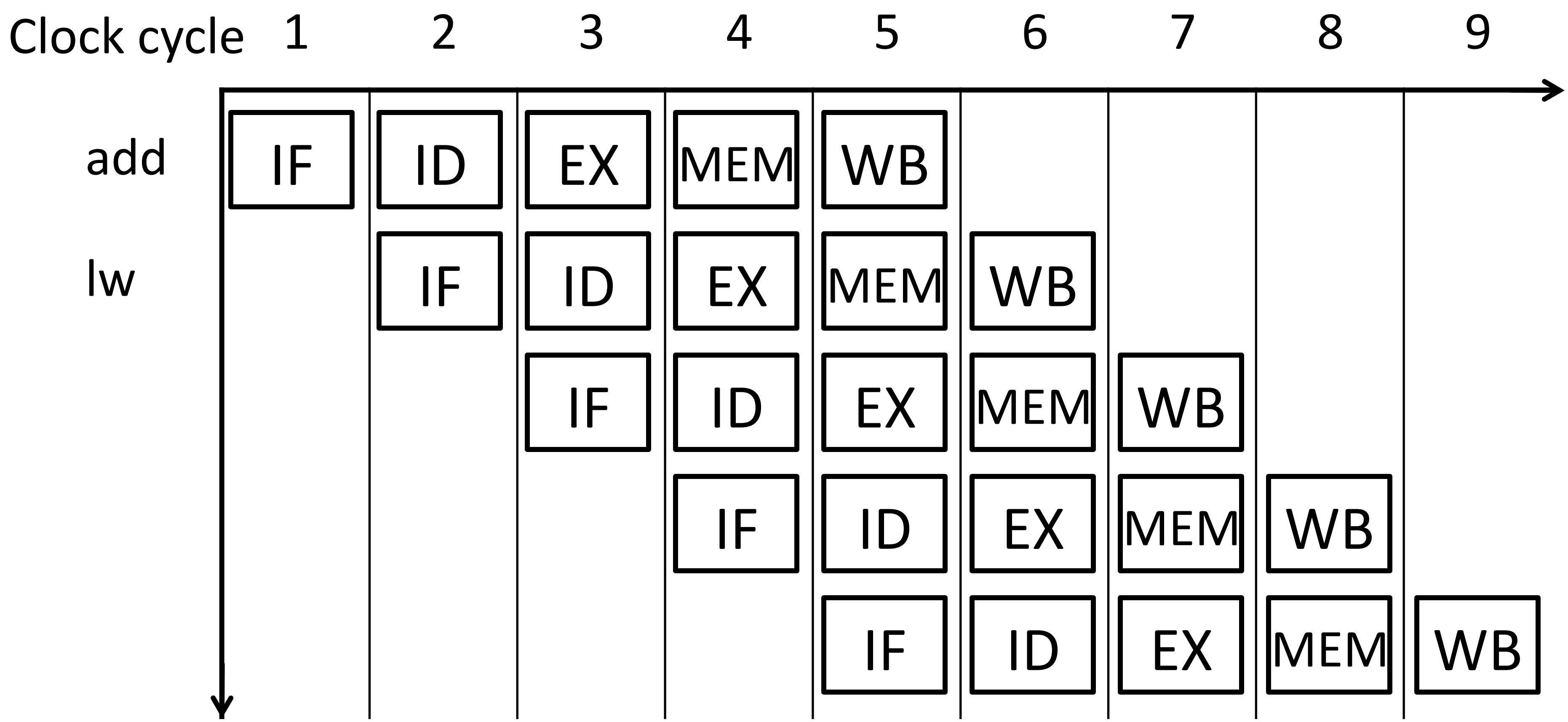


iClicker

The pipeline achieves

A) Latency: 1, throughput: 1 instr/cycle
B) Latency: 5, throughput: 1 instr/cycle
C) Latency: 1, throughput: 1/5 instr/cycle
D) Latency: 5, throughput: 5 instr/cycle
E) None of the above

Time Graphs



Latency: 5

Throughput: 1 instruction/cycle

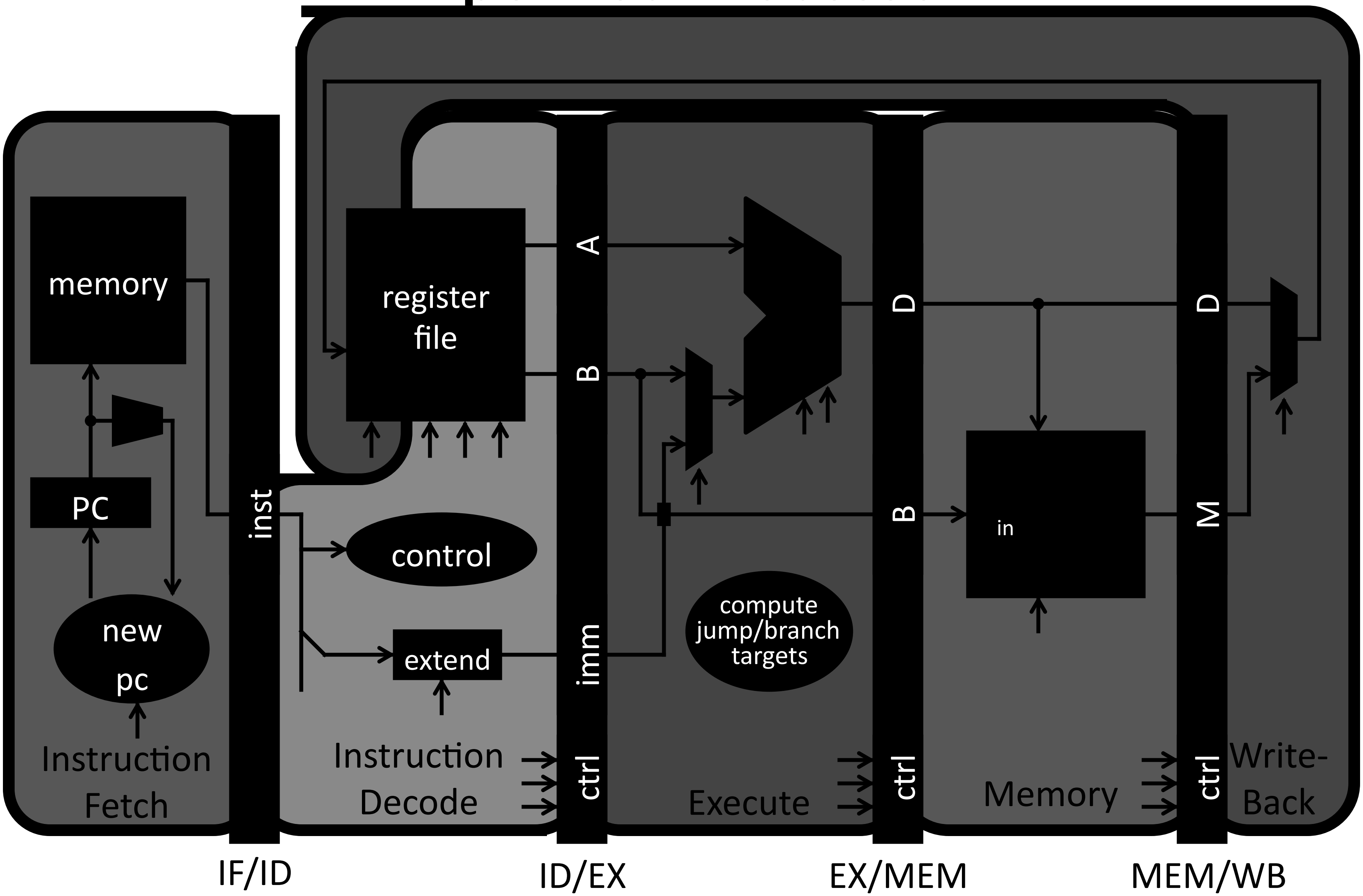
Concurrency: 5

Pipelined Implementation

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

- Stages must share information. How?
  - Add pipeline registers (flip-flops) to pass results between different stages

Pipelined Processor



Pipelined Implementation

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

- Stages must share information. How?
  - Add pipeline registers (flip-flops) to pass results between different stages

And is this it? Not quite….

Hazards

3 kinds

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

Will get back to this

First, how to pipeline when no hazards

IF

Stage 1: Instruction Fetch

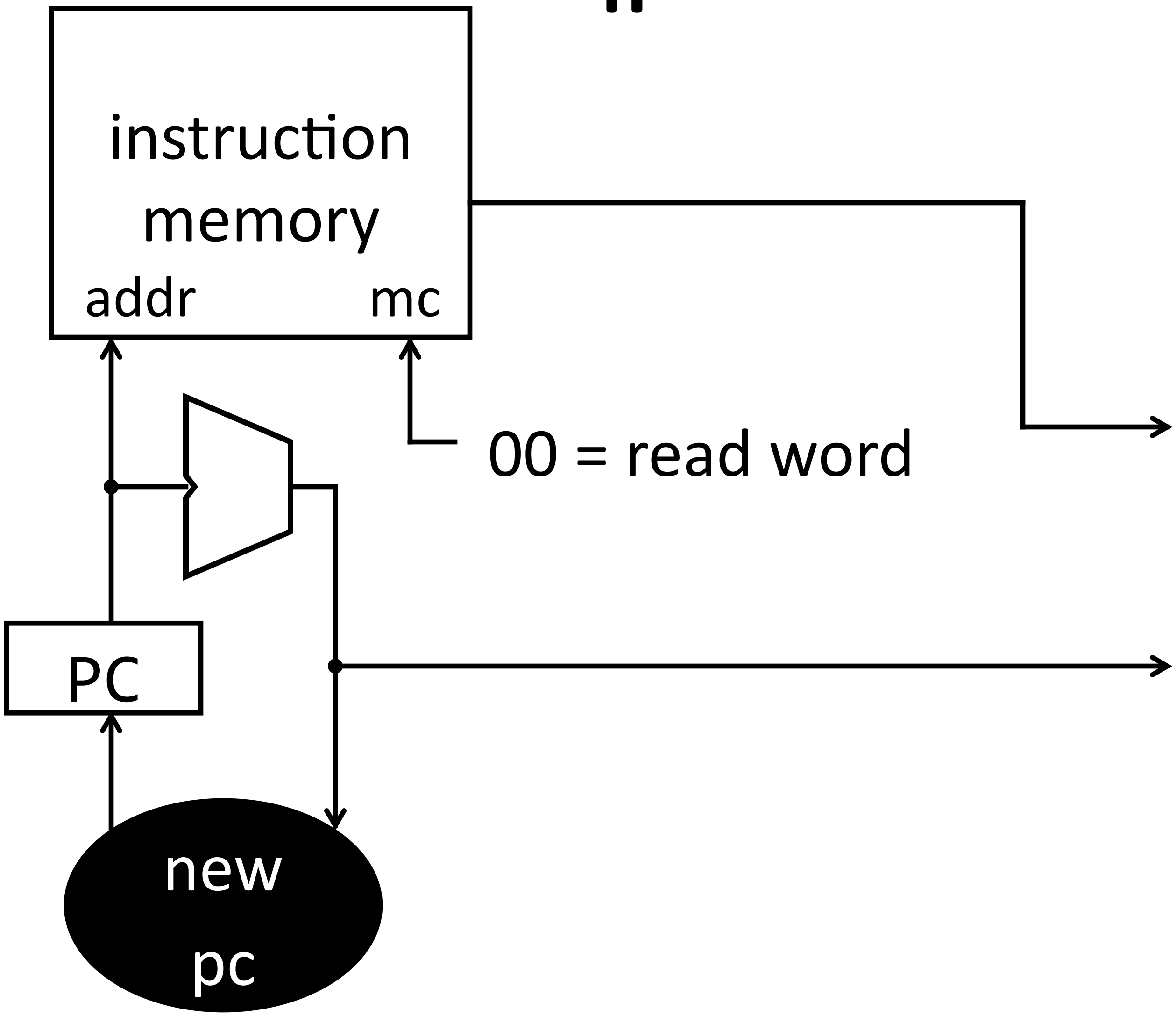
Fetch a new instruction every cycle

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

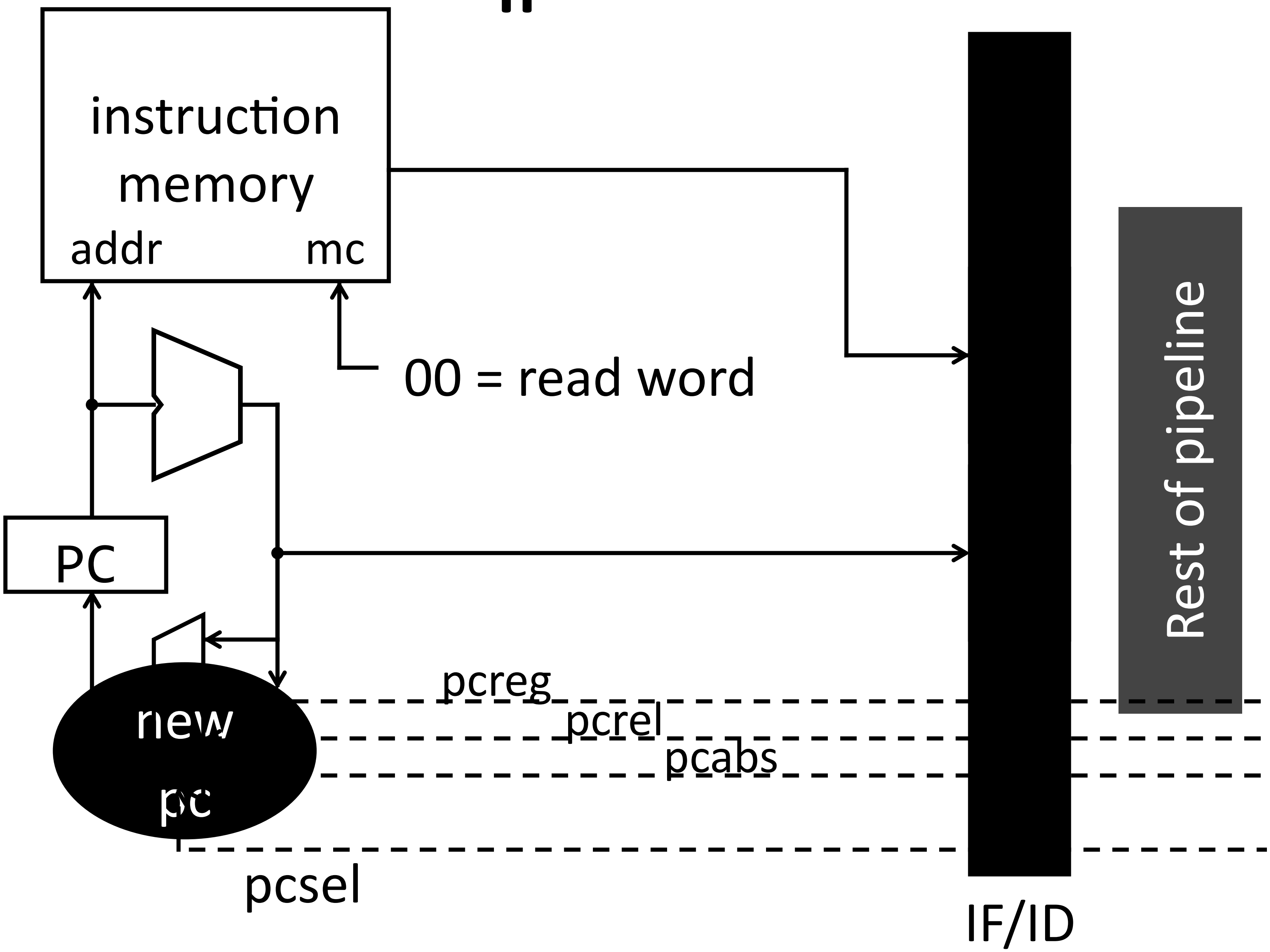
Write values of interest to pipeline register (IF/ID)

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

IF



IF


instruction
memory
addr
mc
00 = read word
PC
new
pc
pcreg
pcrel
pcabs
pcsel
IF/ID
Rest of pipeline

ID

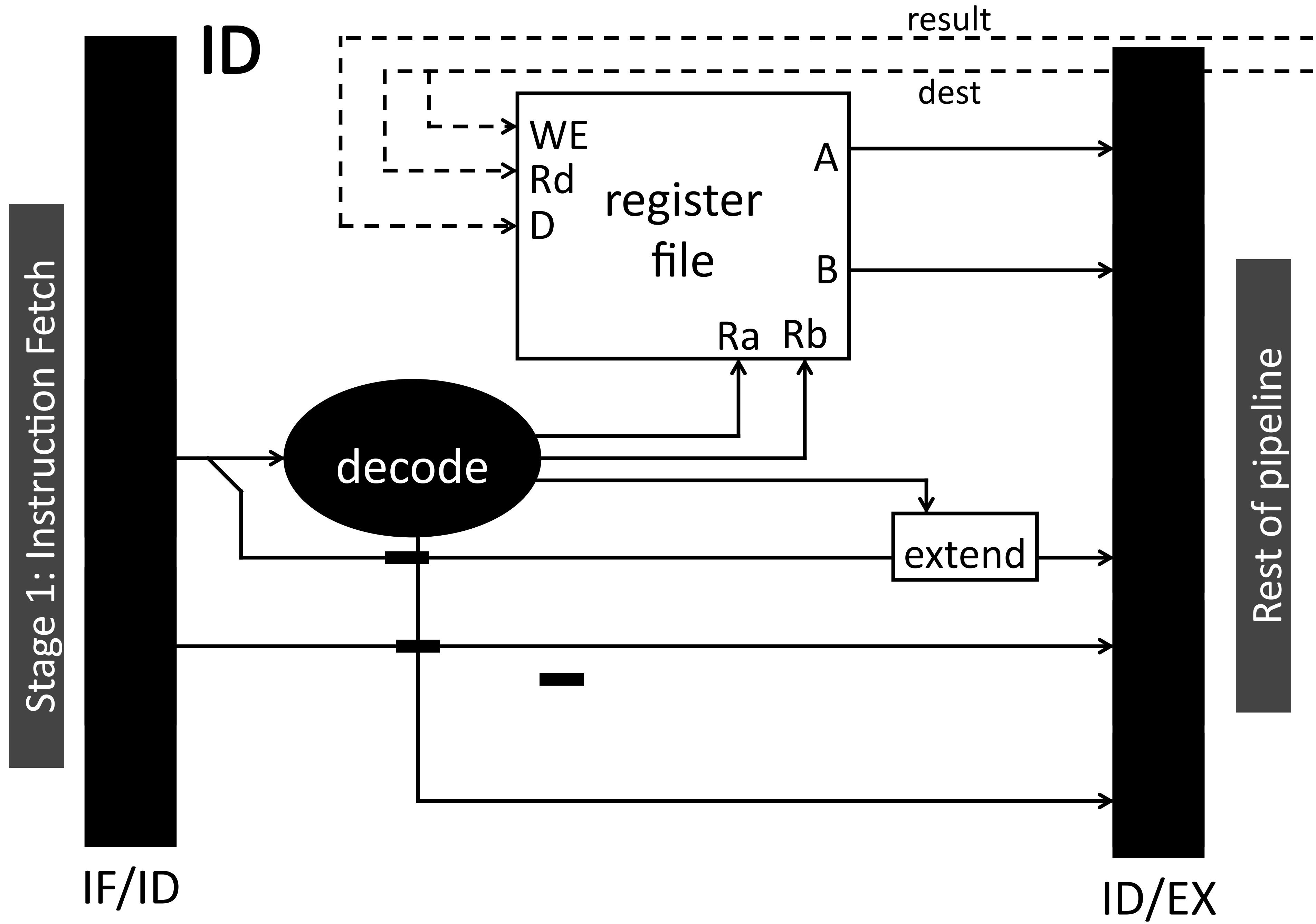
Stage 2: Instruction Decode

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)



ID
result
dest
WE
Rd
D
register
file
A
B
Ra
Rb
decode
extend
Stage 1: Instruction Fetch
Rest of pipeline
IF/ID
ID/EX

EX

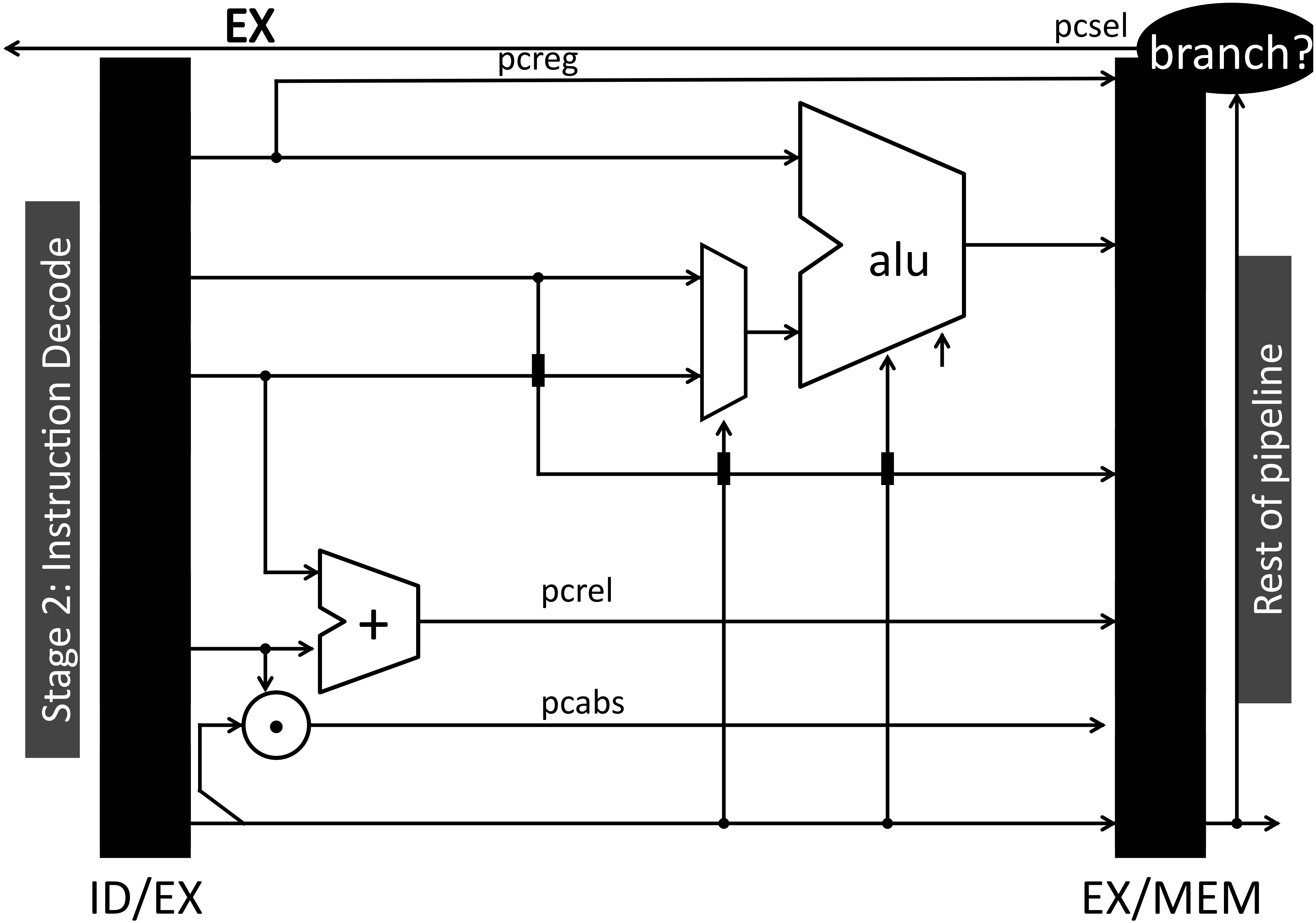
Stage 3: Execute

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


EX
pcsel
branch?
pcreg
alu
Stage 2: Instruction Decode
Rest of pipeline
pcrel
+
pcabs
ID/EX
EX/MEM

MEM

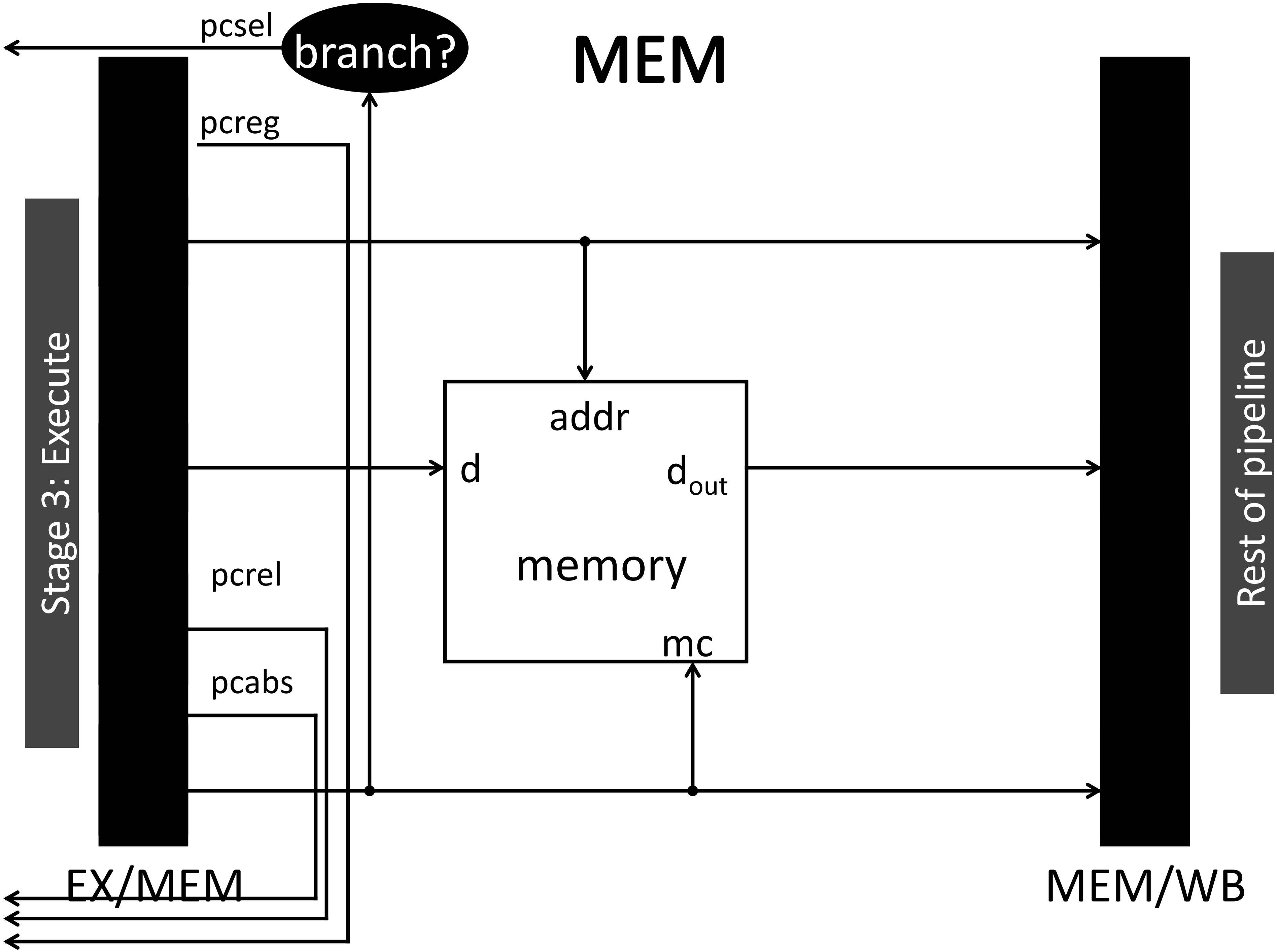
Stage 4: Memory

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


pcsel
branch?
MEM
pcreg
Stage 3: Execute
addr
d
d_out
memory
mc
pcrel
pcabs
Rest of pipeline
EX/MEM
MEM/WB

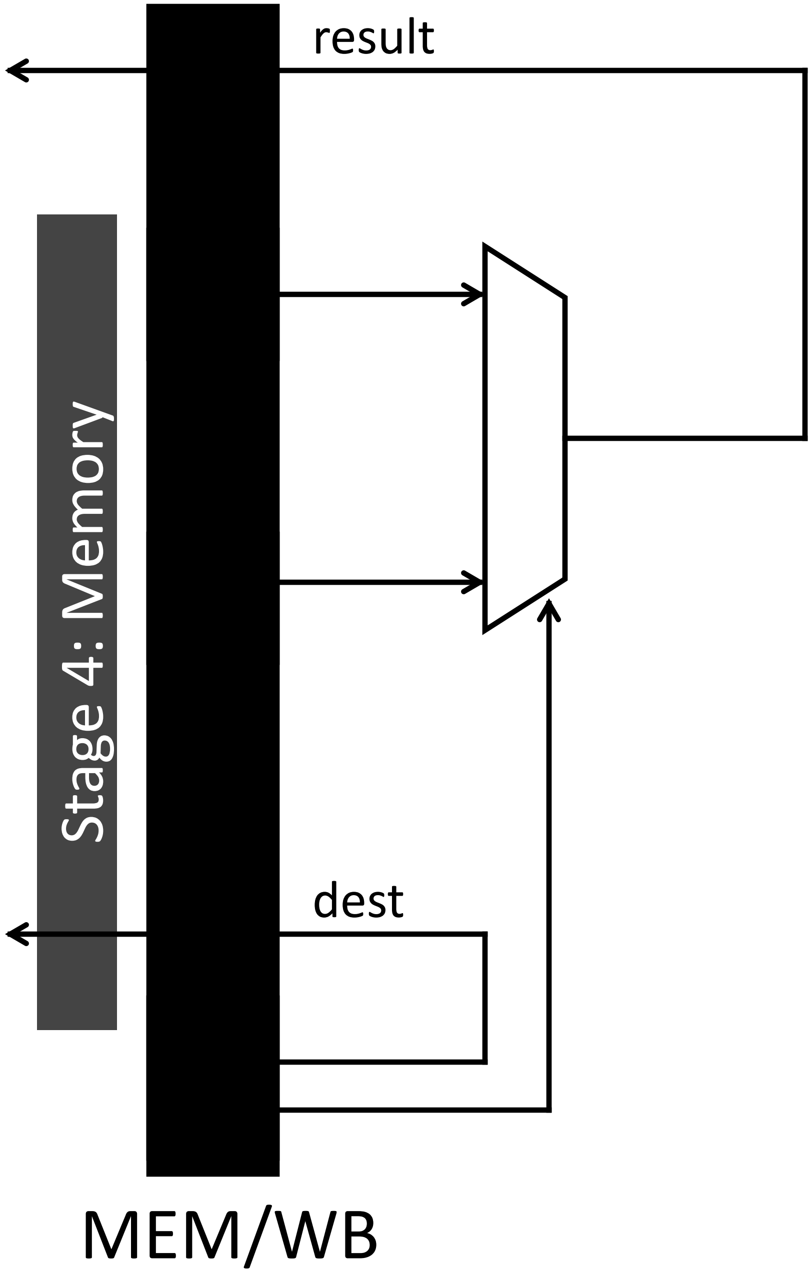
WB

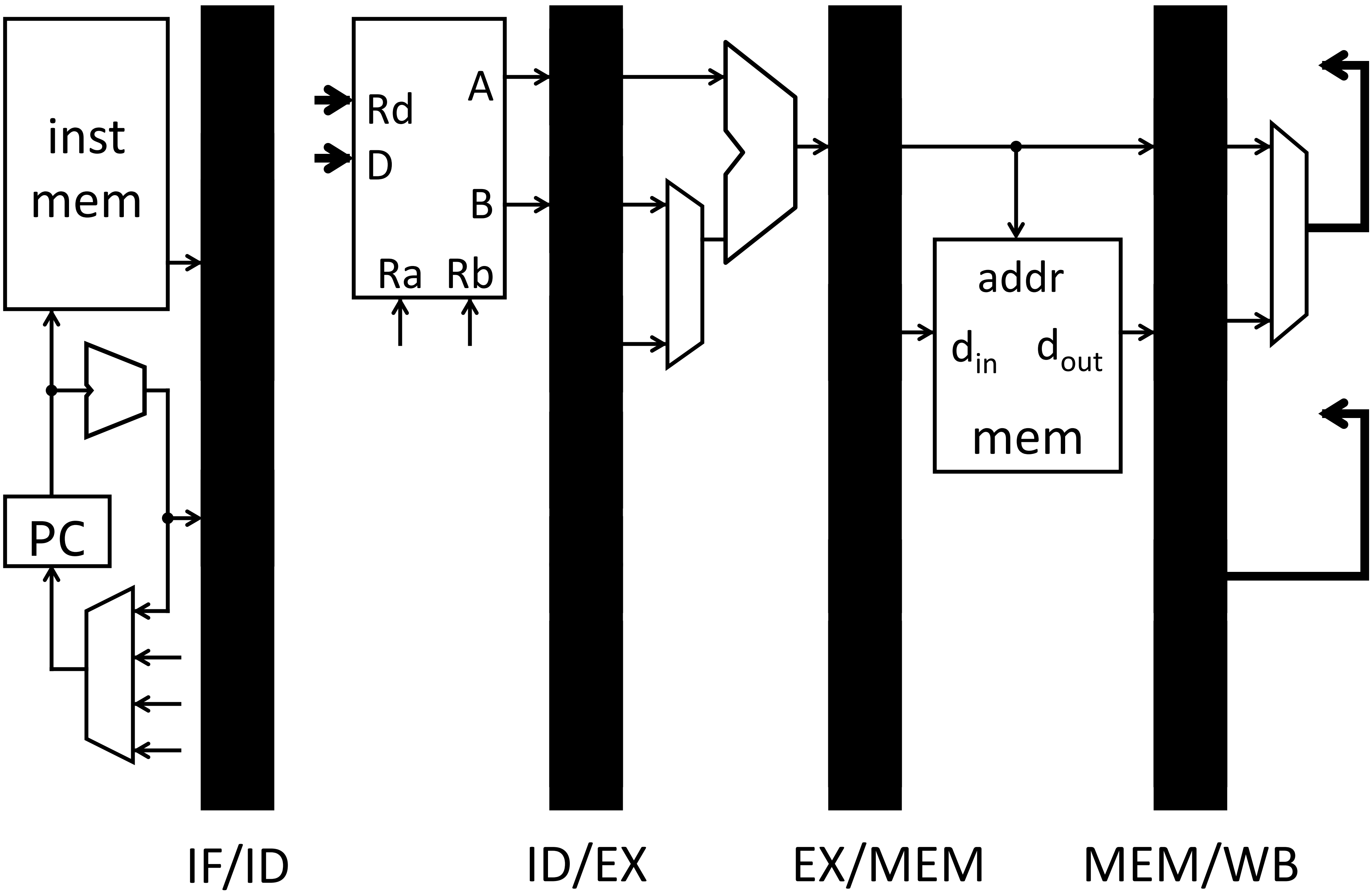
Stage 5: Write-back

On every cycle:

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

WB




inst
mem
PC
Rd
D
A
B
Ra
Rb
addr
d in
d out
mem
IF/ID
ID/EX
EX/MEM
MEM/WB

Example: : Sample Code (Simple)

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

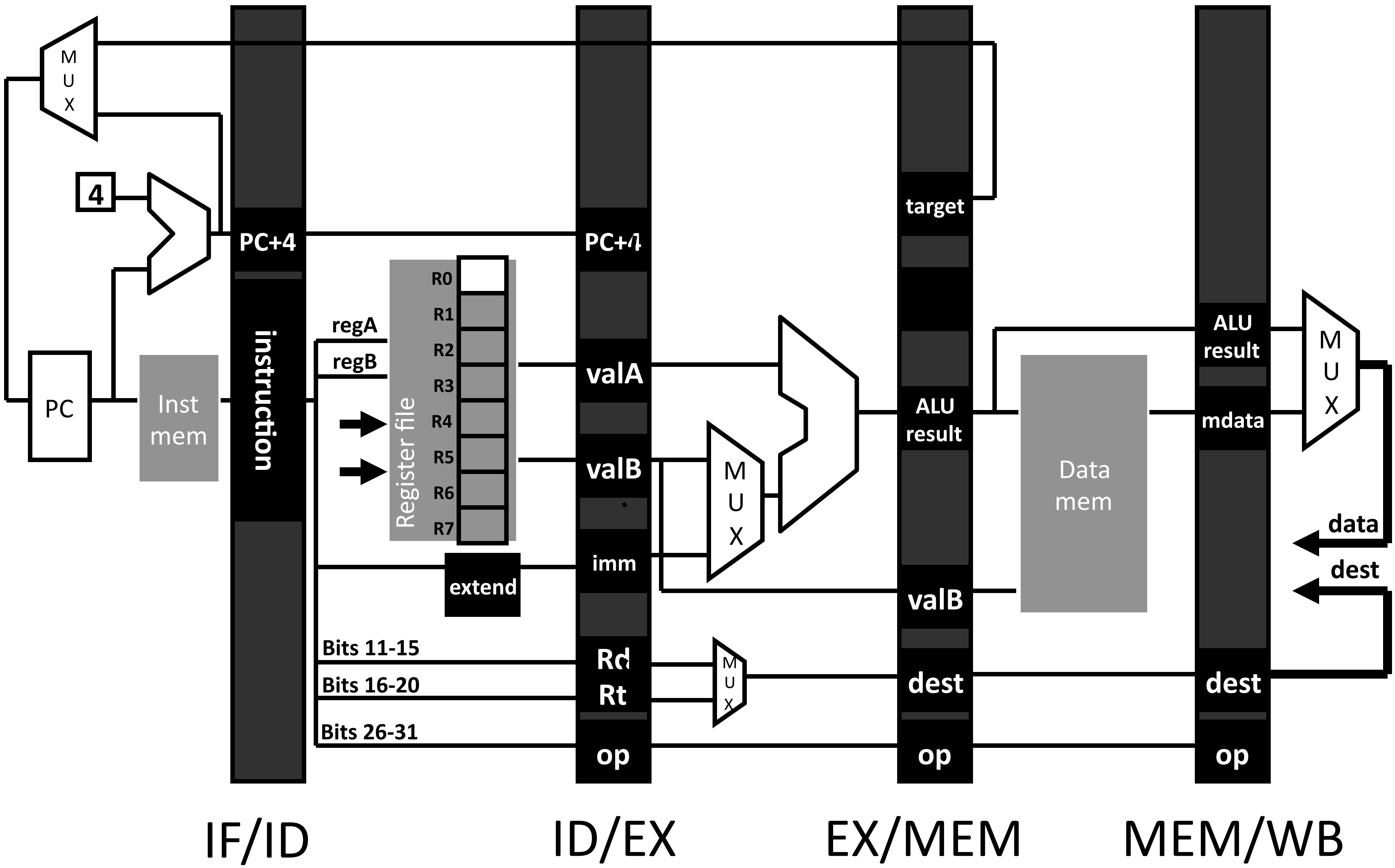
Example: Sample Code (Simple)

Assume eight-register machine

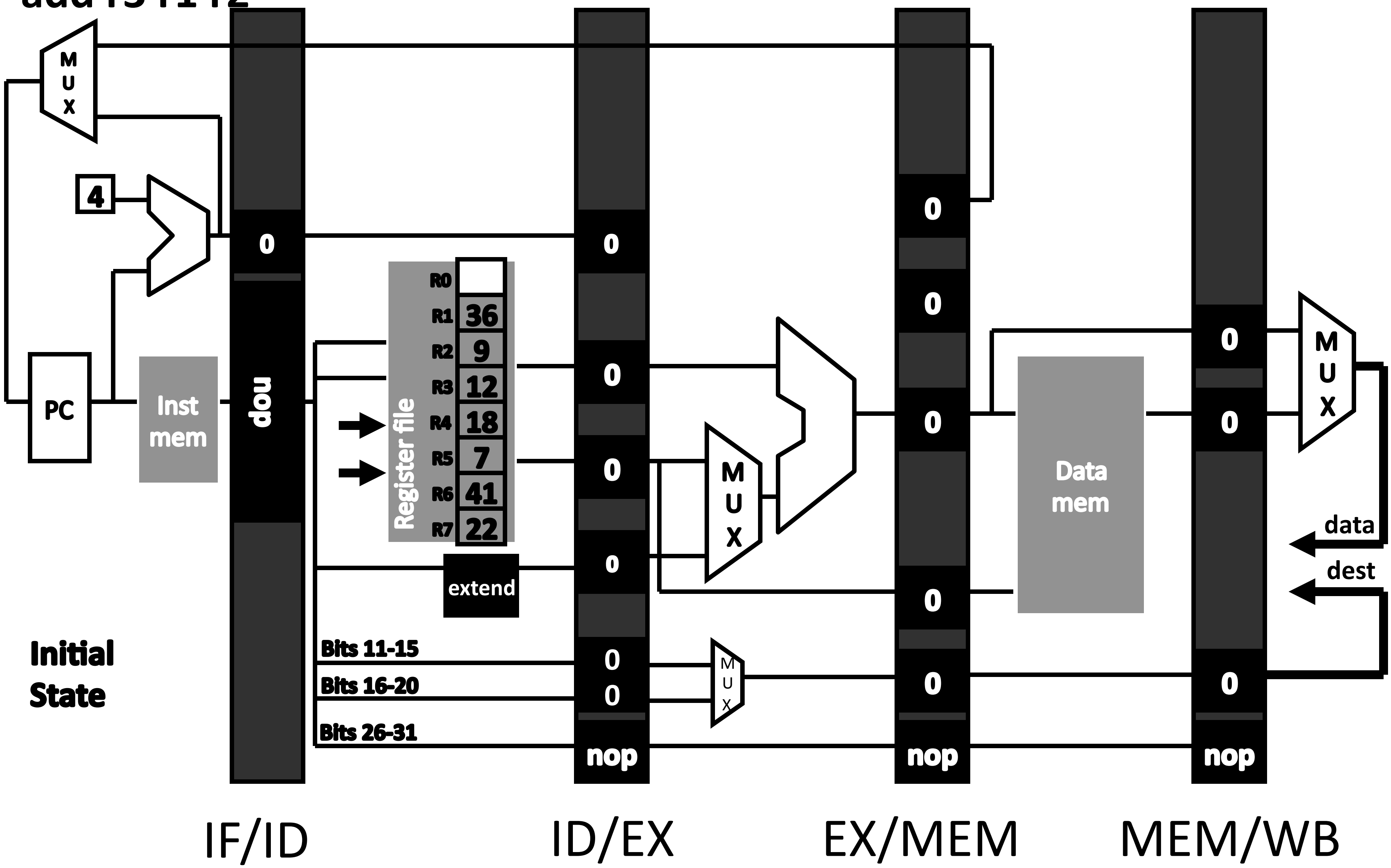
Run the following code on a pipelined datapath

```
add   r3  r1   r2    ;  reg 3 = reg 1 + reg 2
nand  r6  r4   r5    ;  reg 6 = ~(reg 4 & reg 5)
lw    r4  20 (r2)    ;  reg 4 =  Mem[reg2+20]
add   r5  r2   r5    ;  reg 5 = reg 2 + reg 5
sw    r7  12(r3)     ;  Mem[reg3+12] = reg 7
```

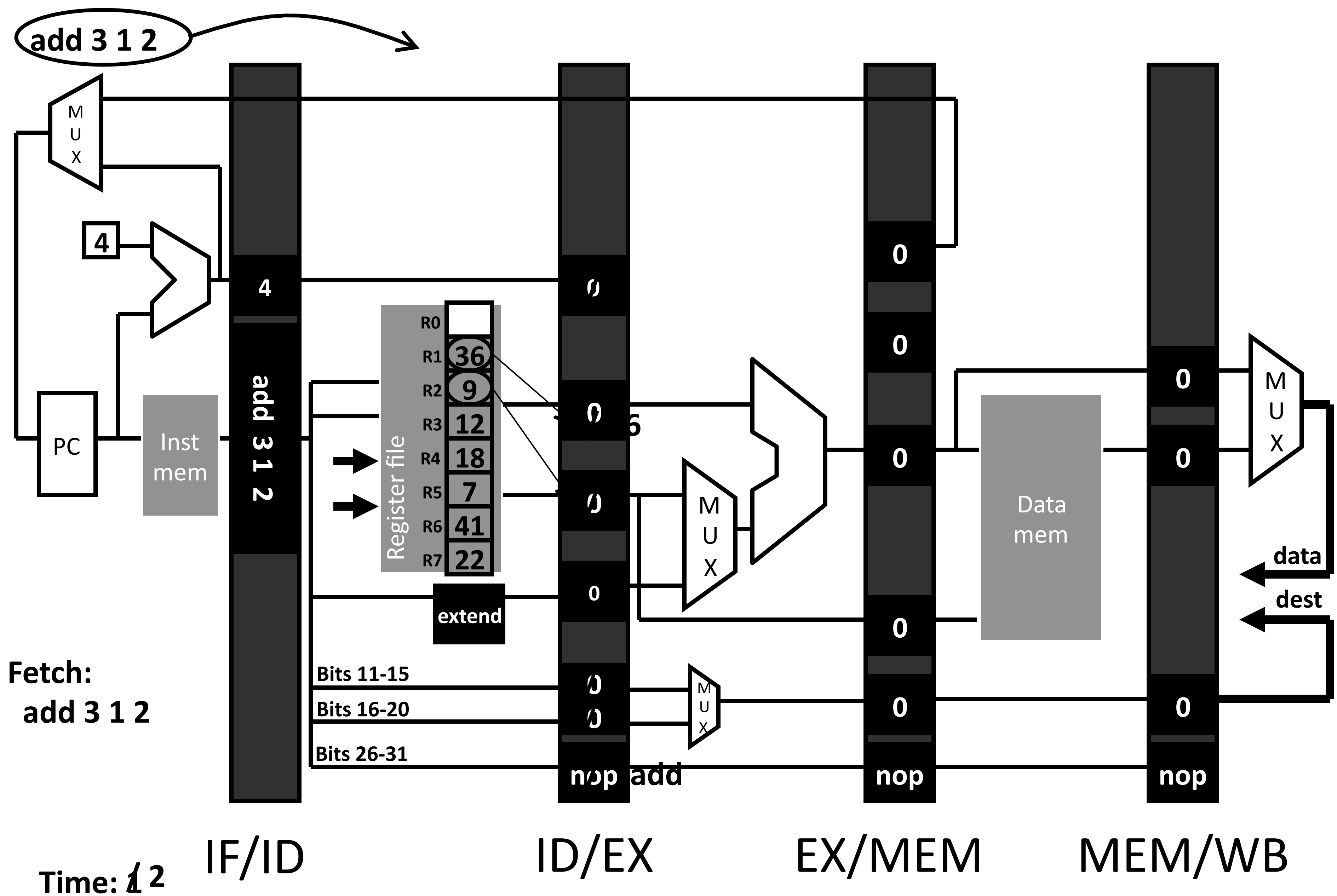
Slides thanks to Sally McKee


MUX
4
PC
Inst mem
PC+4
instruction
regA
regB
Register file
R0
R1
R2
R3
R4
R5
R6
R7
extend
Bits 11-15
Bits 16-20
Bits 26-31
PC+4
valA
valB
imm
Rd
Rt
op
MUX
MUX
target
ALU result
valB
dest
op
Data mem
ALU result
mdata
dest
op
MUX
data
dest
IF/ID
ID/EX
EX/MEM
MEM/WB

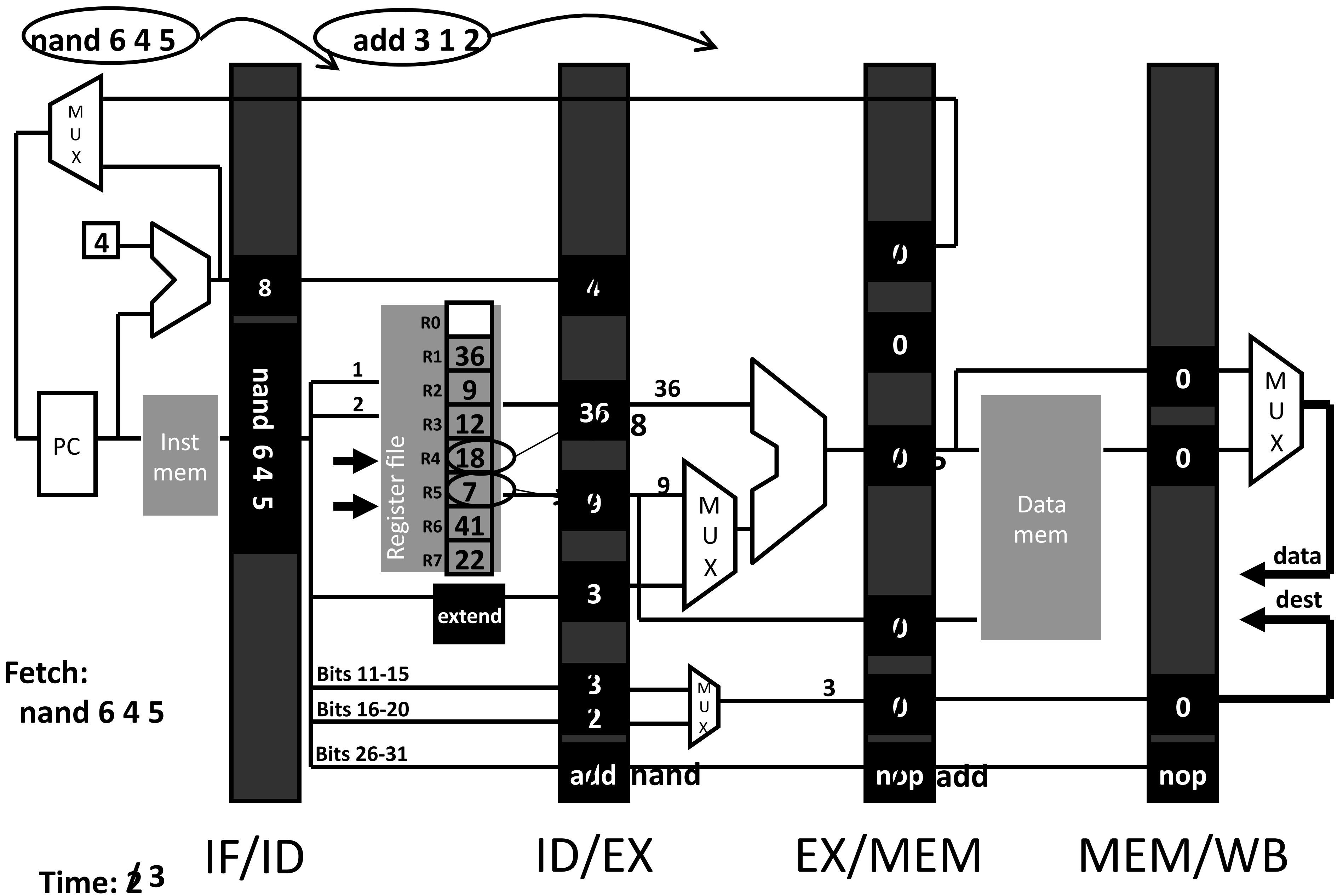
At time 1,
Fetch
add r3 r1 r2

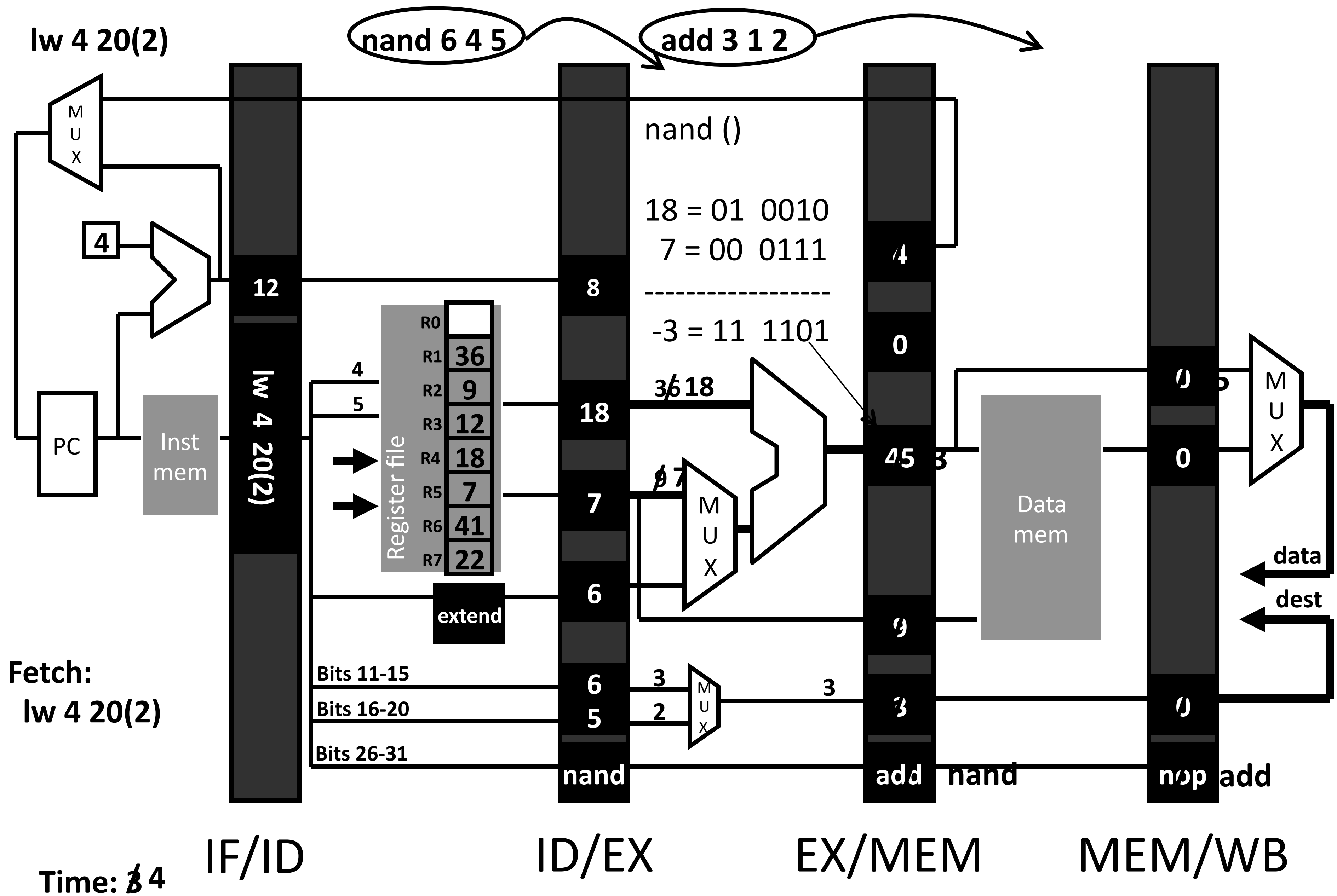


Initial State



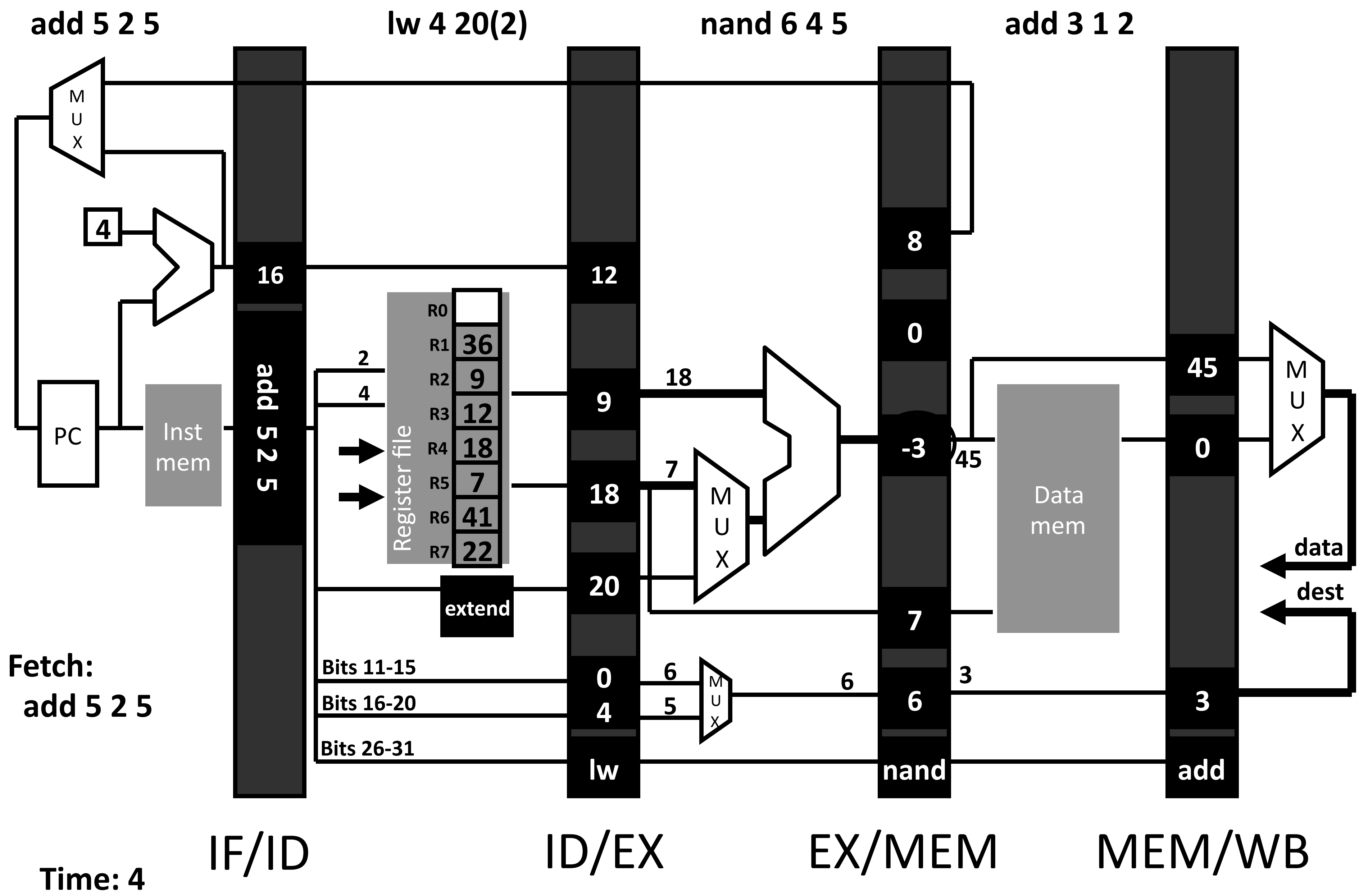
add 3 1 2
MUX
4
PC
Inst mem
add 3 1 2
Register file
R0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
extend
Bits 11-15
Bits 16-20
Bits 26-31
0
nop
add
MUX
Data mem
data
dest
Fetch:
add 3 1 2
Time: 1 2
IF/ID
ID/EX
EX/MEM
MEM/WB


nand 6 4 5
add 3 1 2
M U X
4
8
nand 6 4 5
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
1
2
extend
Bits 11-15
Bits 16-20
Bits 26-31
4
36
8
9
3
3
2
add
nand
36
9
M U X
3
0
0
0
0
0
nop
add
Data mem
0
0
0
nop
data
dest
Fetch:
nand 6 4 5
Time: 2 3
IF/ID
ID/EX
EX/MEM
MEM/WB

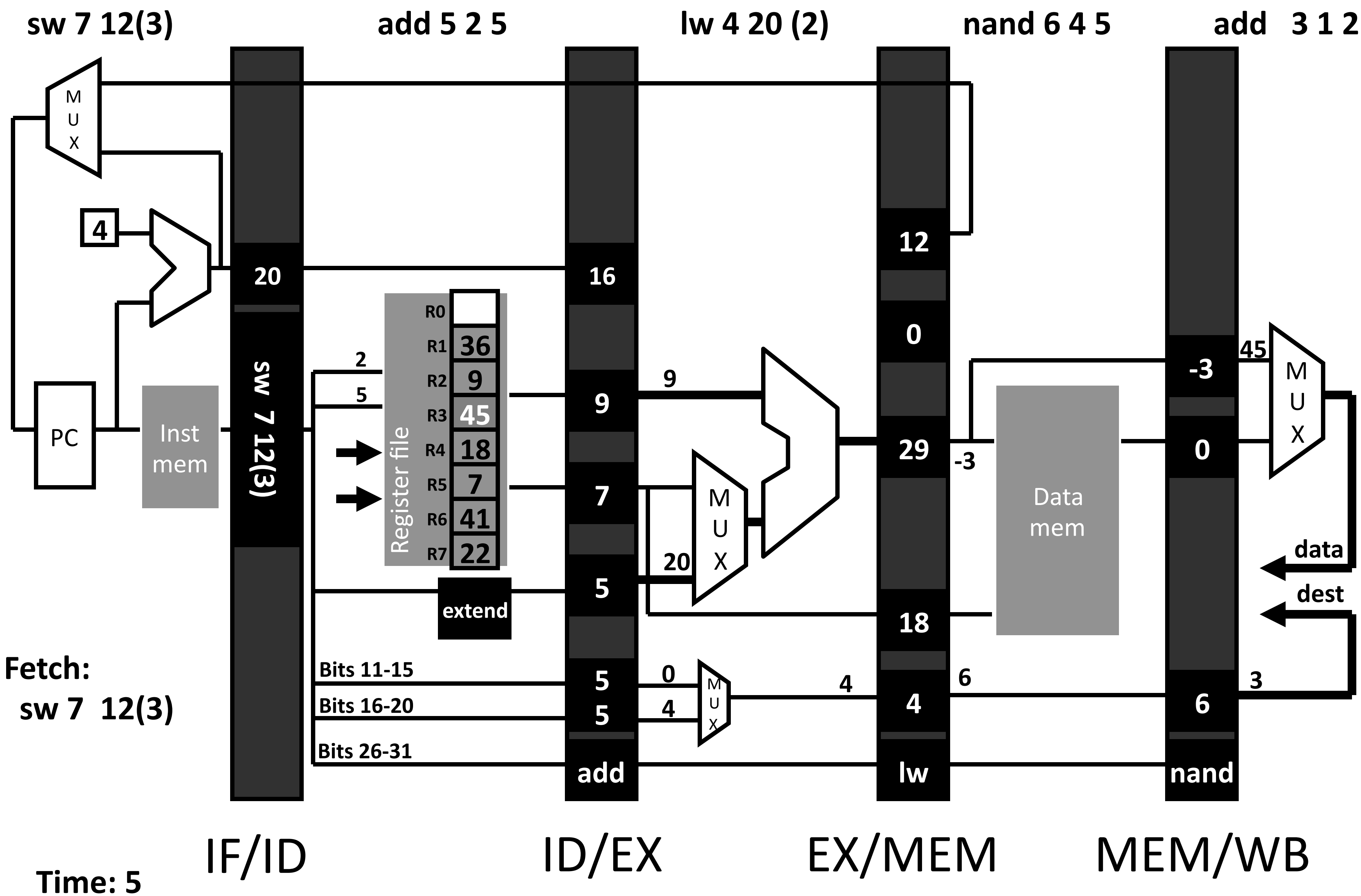


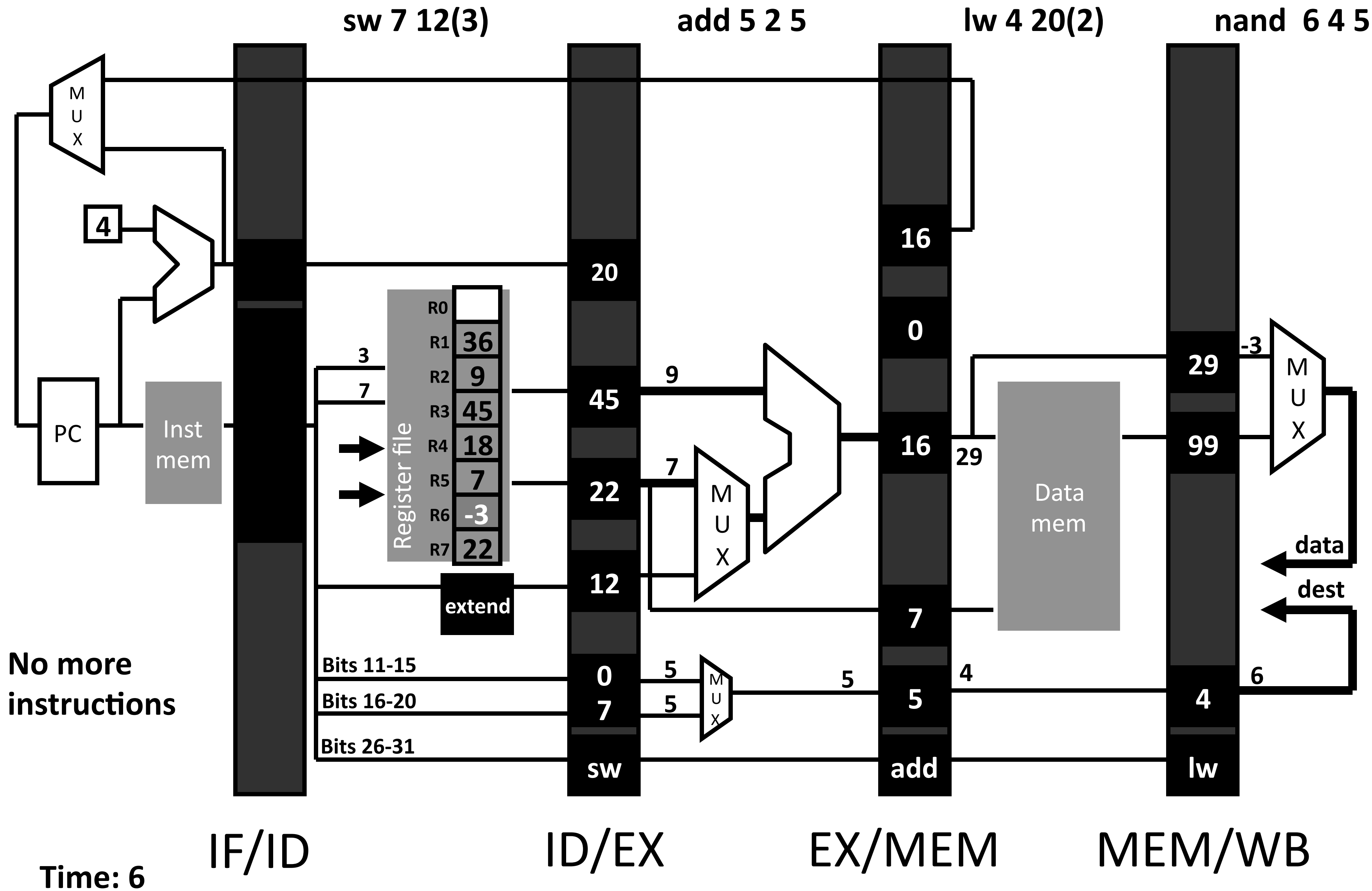
lw 4 20(2)
nand 6 4 5
add 3 1 2
M U X
4
12
8
PC
Inst mem
lw 4 20(2)
nand ()
18 = 01 0010
7 = 00 0111

-3 = 11 1101
4
0
45
R0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
Register file
4
5
18
7
6
extend
36 18
9 7
M U X
Data mem
data
dest
9
Bits 11-15
Bits 16-20
Bits 26-31
6
5
3
2
3
nand
add
nand
nop
add
Fetch:
lw 4 20(2)
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 3 4

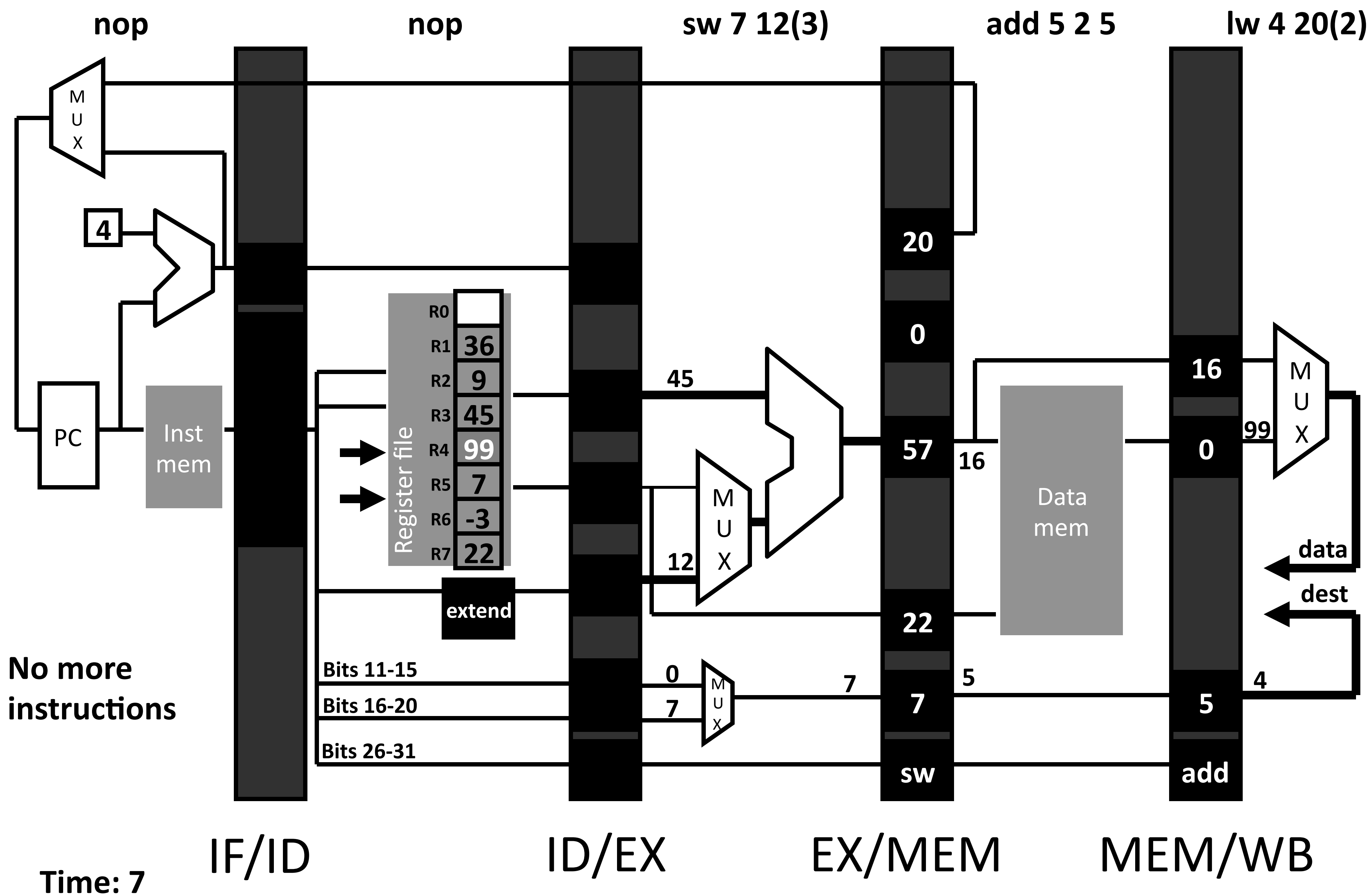


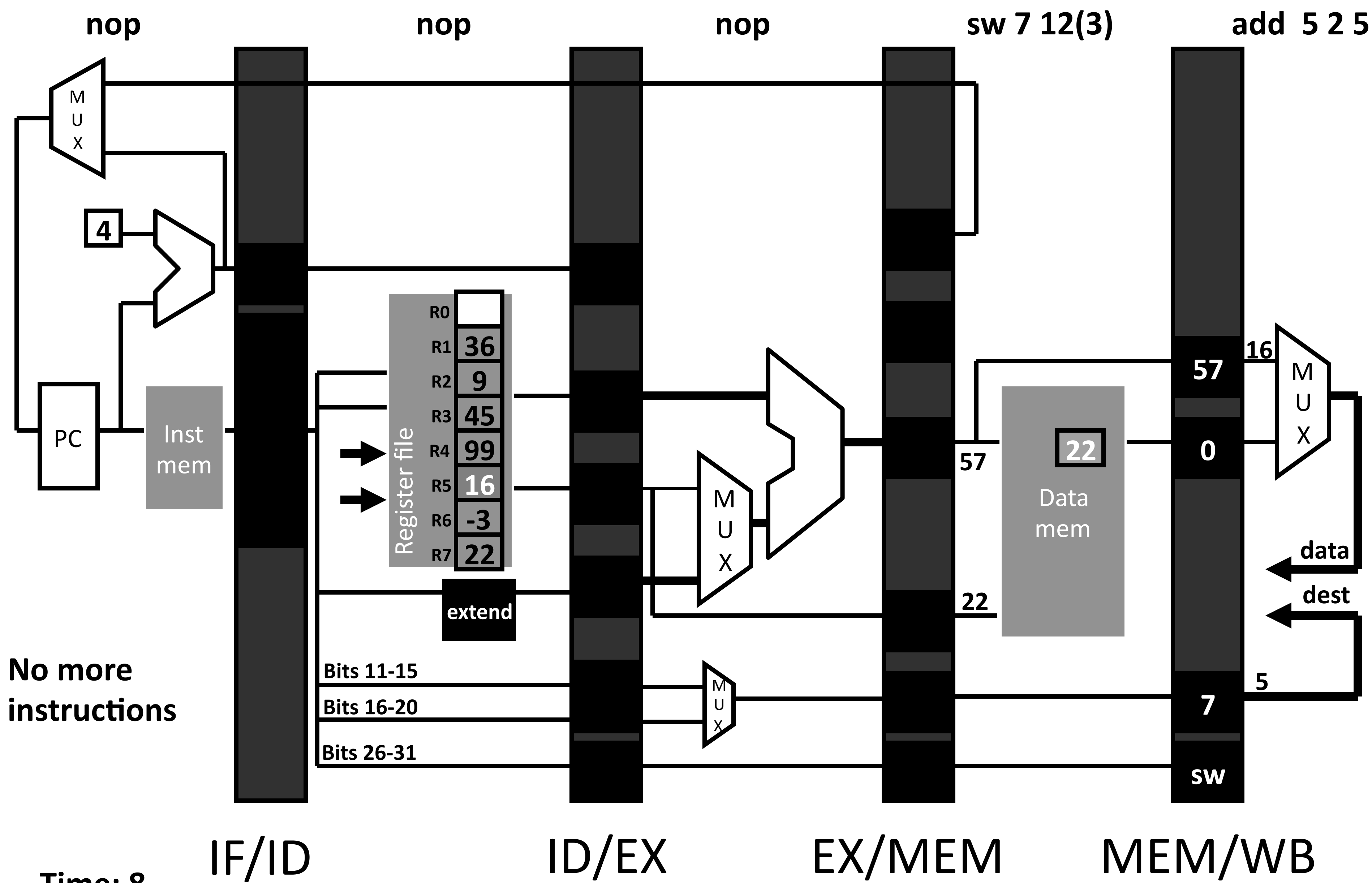
add 5 2 5
lw 4 20(2)
nand 6 4 5
add 3 1 2
MUX
4
16
add 5 2 5
12
8
0
R0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
Register file
2
4
9
18
45
0
-3
45
PC
Inst mem
18
7
MUX
20
extend
Data mem
MUX
data
dest
7
Bits 11-15
Bits 16-20
Bits 26-31
0
4
6
5
MUX
6
6
3
3
lw
nand
add
Fetch:
add 5 2 5
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 4


sw 7 12(3)
add 5 2 5
lw 4 20 (2)
nand 6 4 5
add 3 1 2
M U X
4
20
16
12
0
PC
Inst mem
sw 7 12(3)
R0
R1 36
R2 9
R3 45
R4 18
R5 7
R6 41
R7 22
Register file
2
5
9
9
7
5
20
M U X
29
-3
Data mem
18
-3
0
45
M U X
data
dest
extend
Fetch:
sw 7 12(3)
Bits 11-15
Bits 16-20
Bits 26-31
5
5
0
4
M U X
4
4
6
6
3
add
lw
nand
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 5



sw 7 12(3)
add 5 2 5
lw 4 20(2)
nand 6 4 5
MUX
4
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 45
R4 18
R5 7
R6 -3
R7 22
3
7
extend
20
45
22
12
0
7
sw
9
7
MUX
5
5
MUX
16
0
16
29
7
5
add
Data mem
4
29
-3
99
MUX
4
lw
6
data
dest
Bits 11-15
Bits 16-20
Bits 26-31
No more instructions
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 6

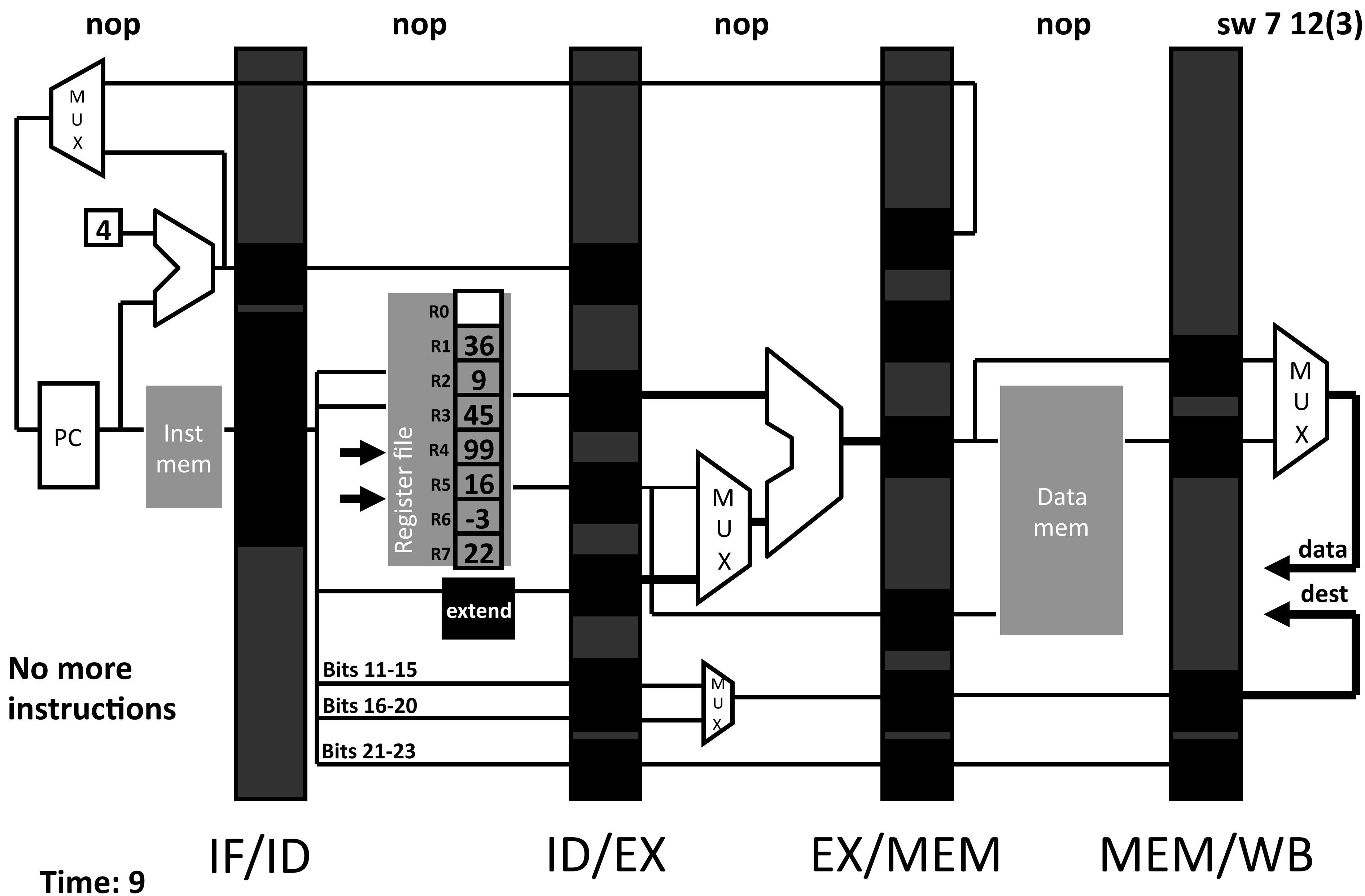

nop
nop
sw 7 12(3)
add 5 2 5
lw 4 20(2)
MUX
4
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 45
R4 99
R5 7
R6 -3
R7 22
extend
45
12
MUX
0
7
MUX
Bits 11-15
Bits 16-20
Bits 26-31
20
0
57
22
7
sw
16
5
Data mem
16
0
5
add
99
4
MUX
data
dest
No more instructions
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 7



No more instructions

Time: 8

Slides thanks to Sally McKee



nop
nop
nop
nop
sw 7 12(3)
M U X
4
PC
Inst mem
Register file
R0
R1 36
R2 9
R3 45
R4 99
R5 16
R6 -3
R7 22
extend
M U X
Data mem
M U X
data
dest
Bits 11-15
Bits 16-20
Bits 21-23
M U X
No more instructions
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 9

Takeaway

Pipelining is a powerful technique to mask latencies and increase throughput

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

Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)

Hazards

See P&H Chapter: 4.7-4.8

Hazards

3 kinds

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

Data Hazards

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

i.e. add r3, r1, r2

sub r5, r3, r4

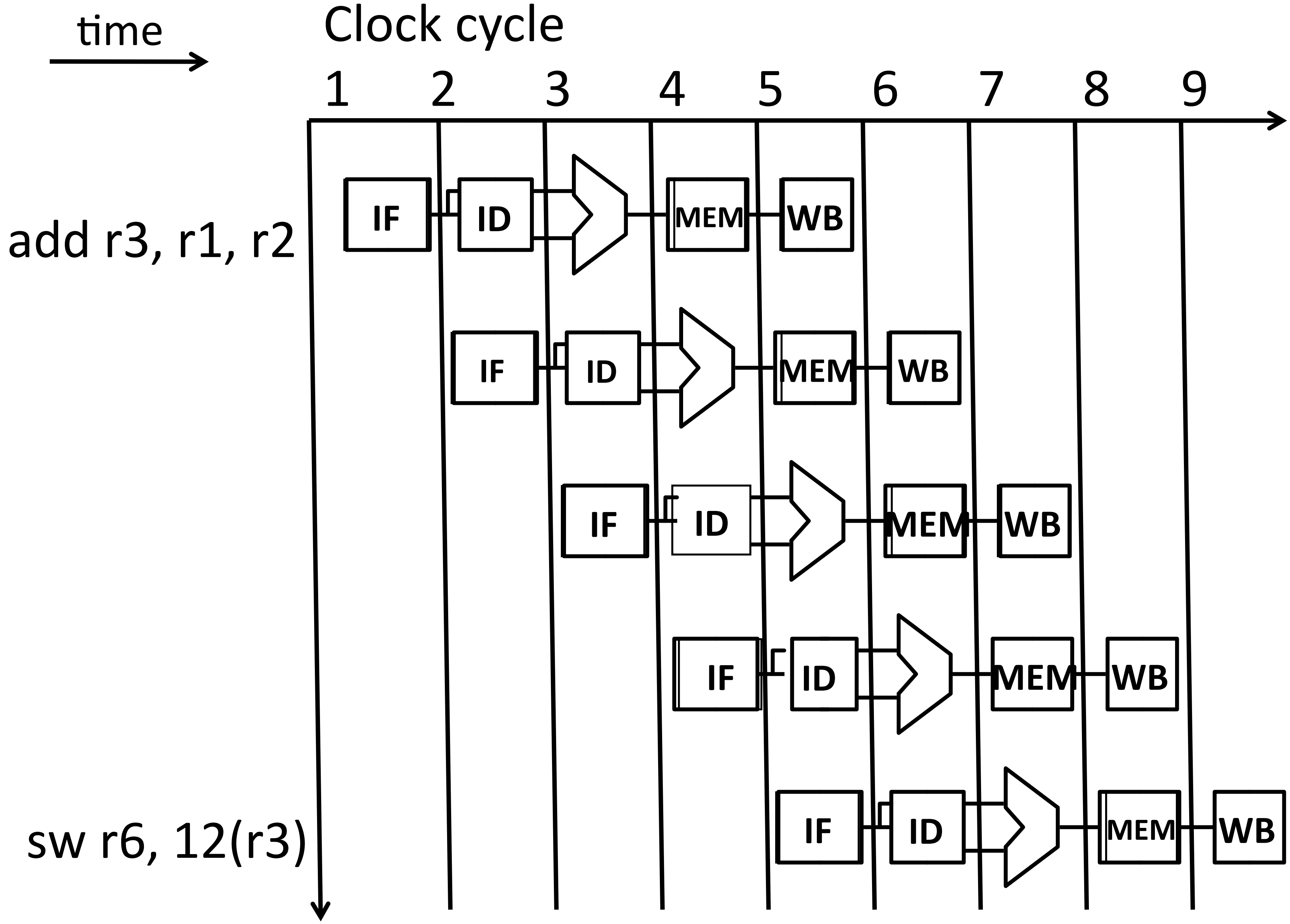
Need to detect and then fix such hazards

Why do data hazards occur?

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- instruction may read (need) values that are being computed further down the pipeline
  - In fact this is quite common

Data Hazards



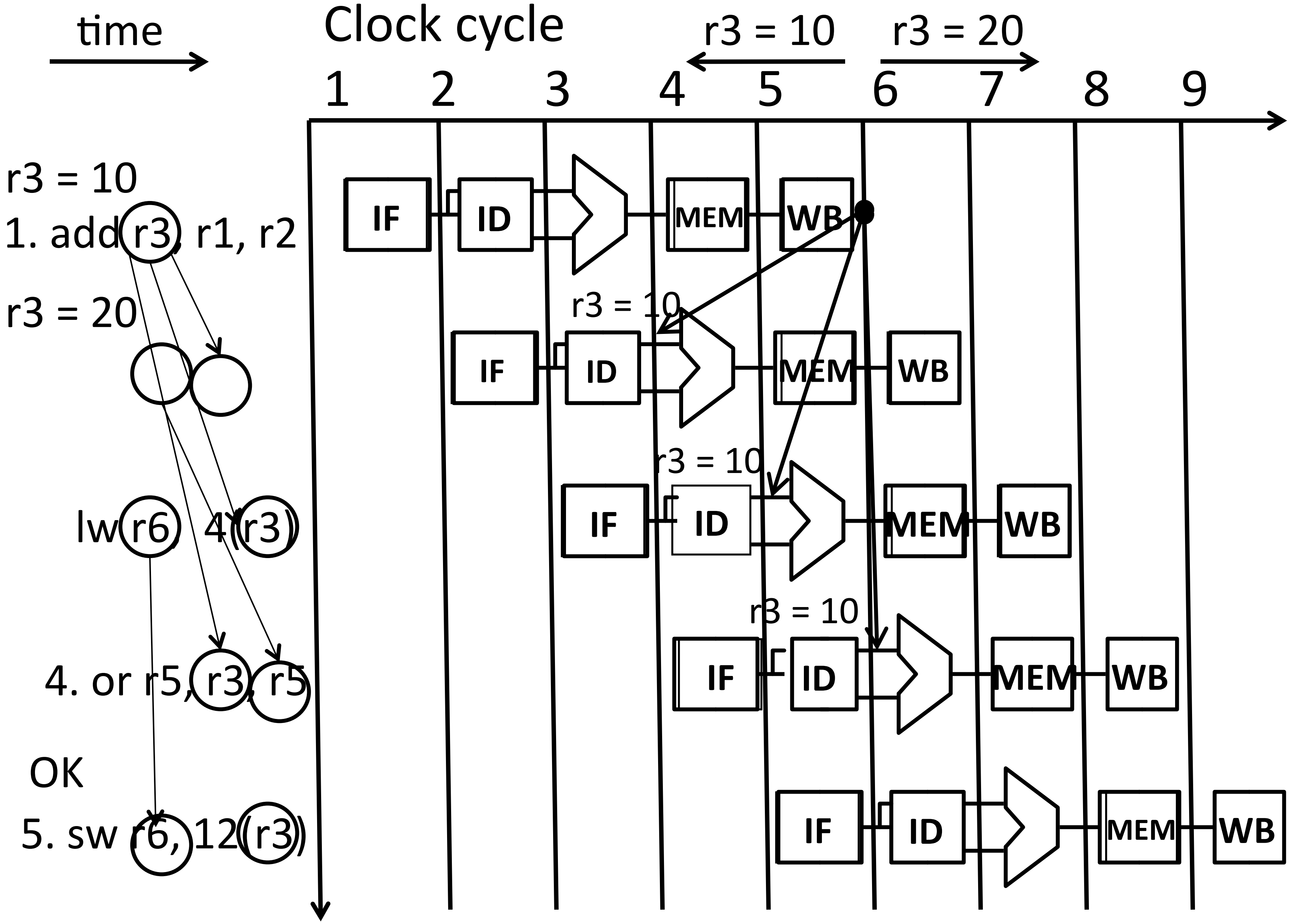
iClicker

```
add r3, r1, r2

sub r5, r3, r4

lw r6,  4(r3)

or r5, r3, r5

sw r6, 12(r3)
```

How many data hazards due to r3 only

A) 1
B) 2
C) 3
D) 4
E) 5

Data Hazards



Data Hazards

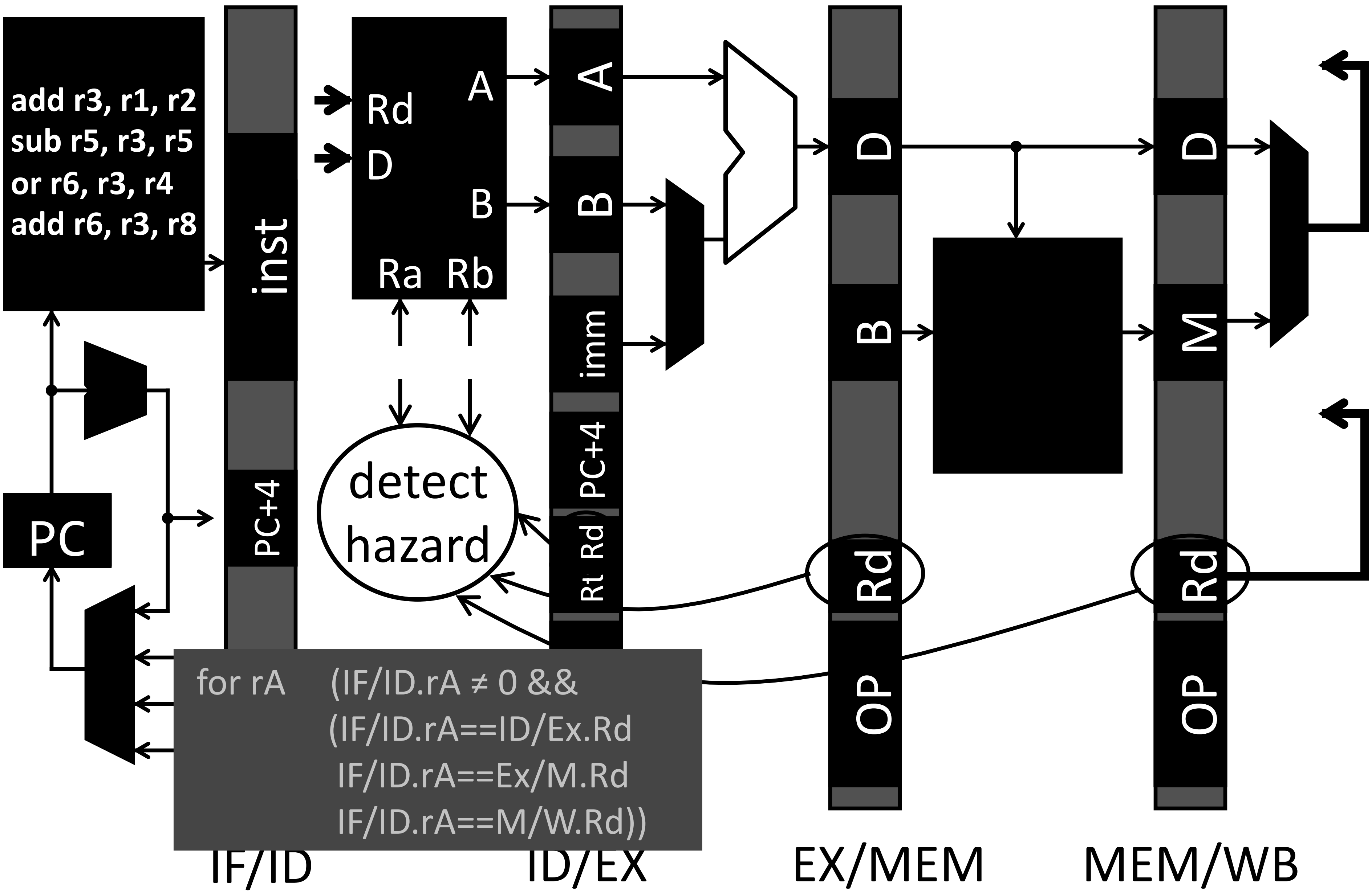
What about data dependencies (also known as a data hazard in a pipelined processor)?

i.e. add r3, r1, r2

sub r5, r3, r4

How to detect?

Detecting Data Hazards



Detecting 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
  - In fact this is quite common

How to detect? (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)
```

Next Goal

- What to do if data hazard detected?

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

Stalling

How to stall an instruction in ID stage

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

Stalling

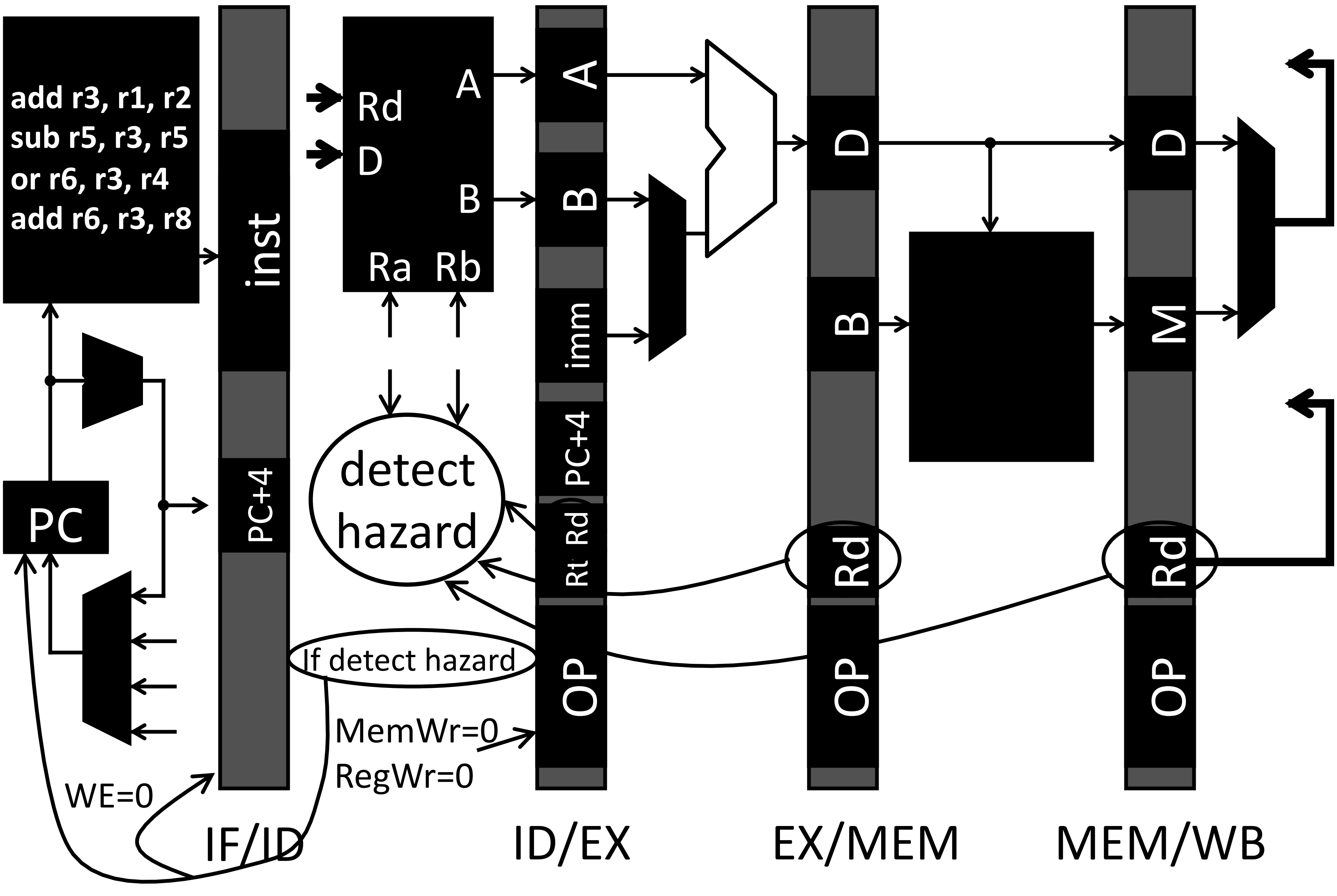
time →

Instruction	1	2	3	4	5	6	7	8	
r3 = 10 add r3, r1, r2 r3 = 20	IF	ID	Ex	M	W				
sub r5, r3, r5		IF	ID	ID	ID	ID	Ex	M	W
or r6, r3, r4			IF	IF	IF	IF	ID	Ex	M
add r6, r3, r8							IF	ID	Ex

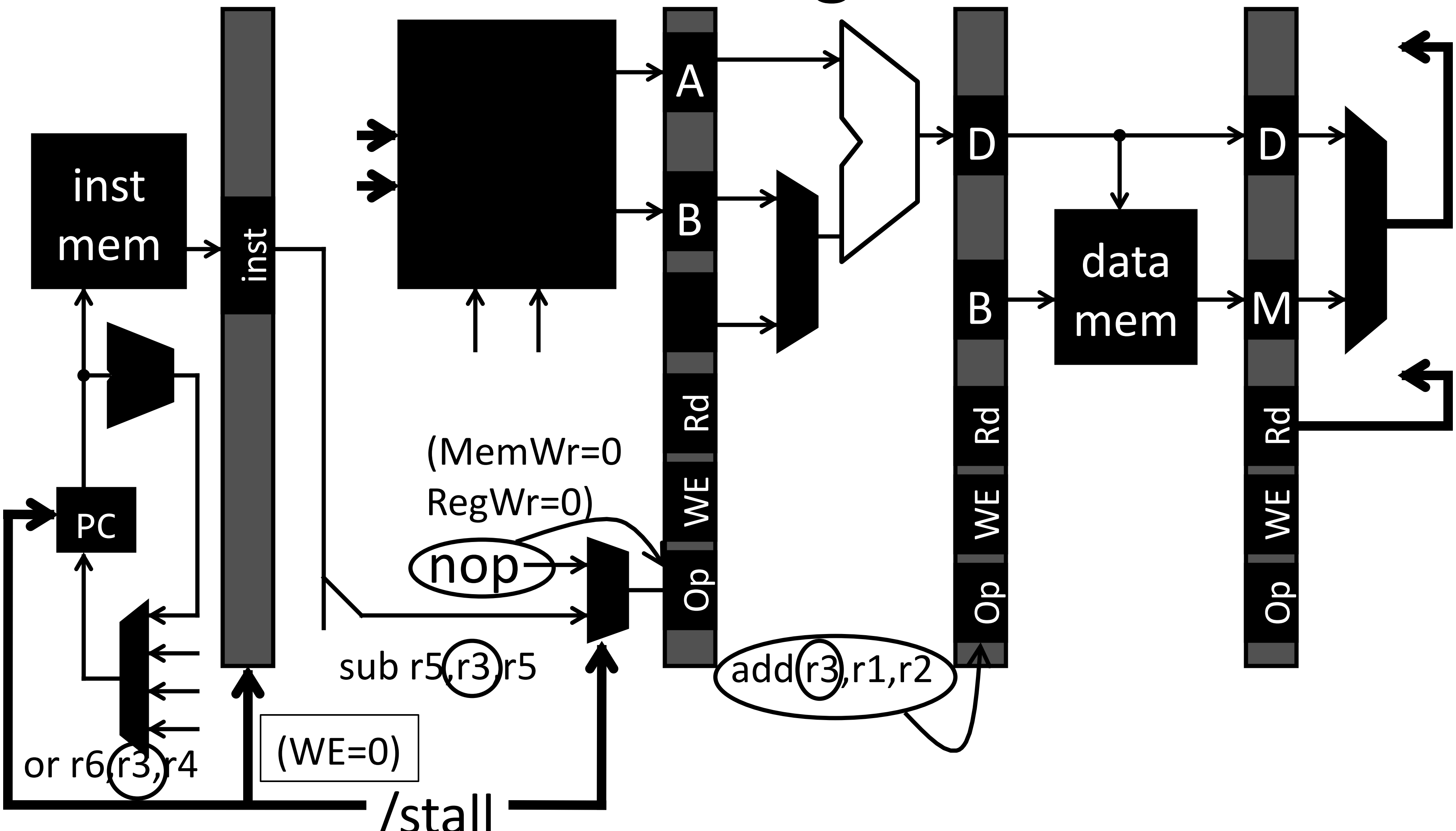
Clock cycle

3 Stalls

Detecting Data Hazards

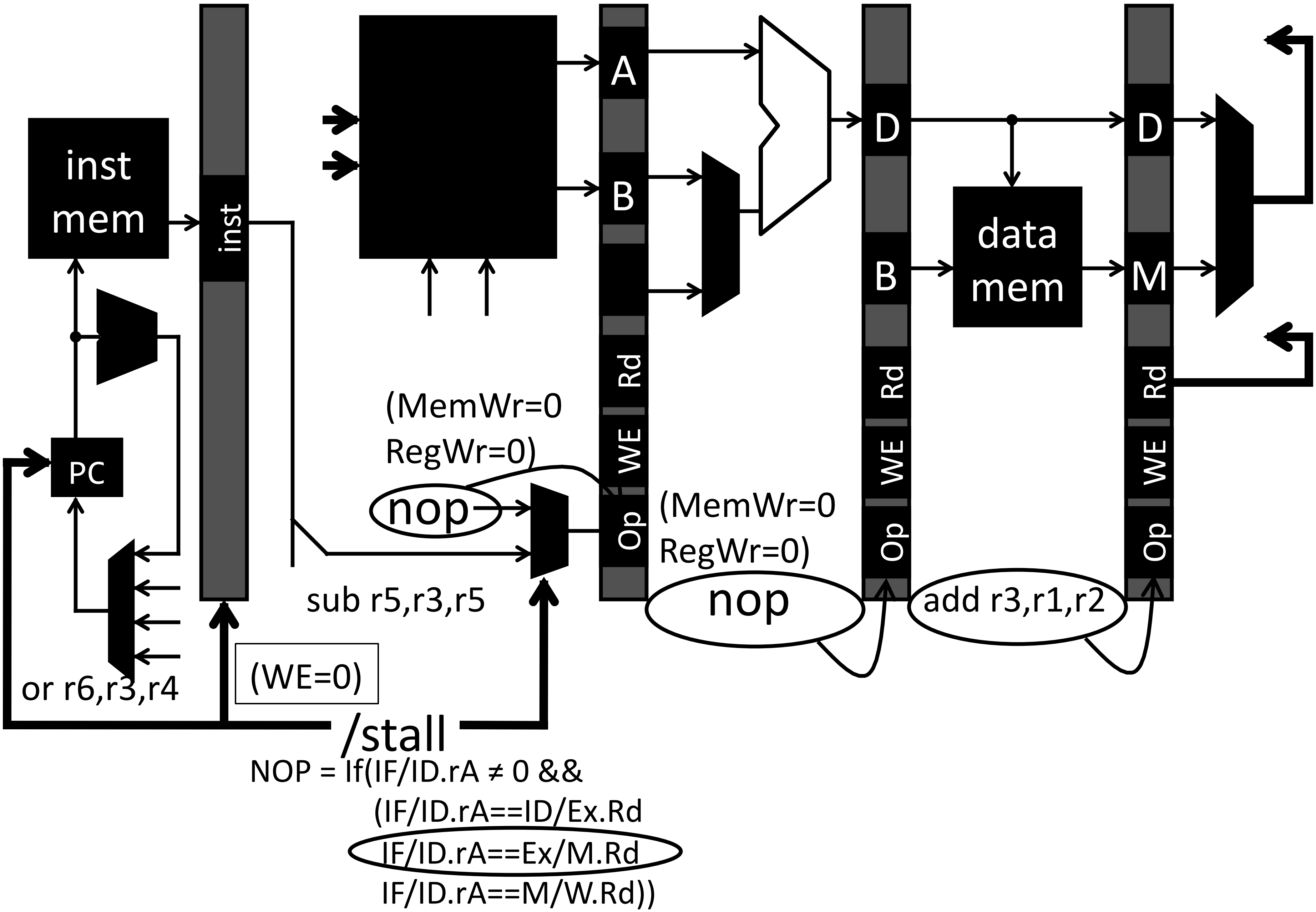


Stalling

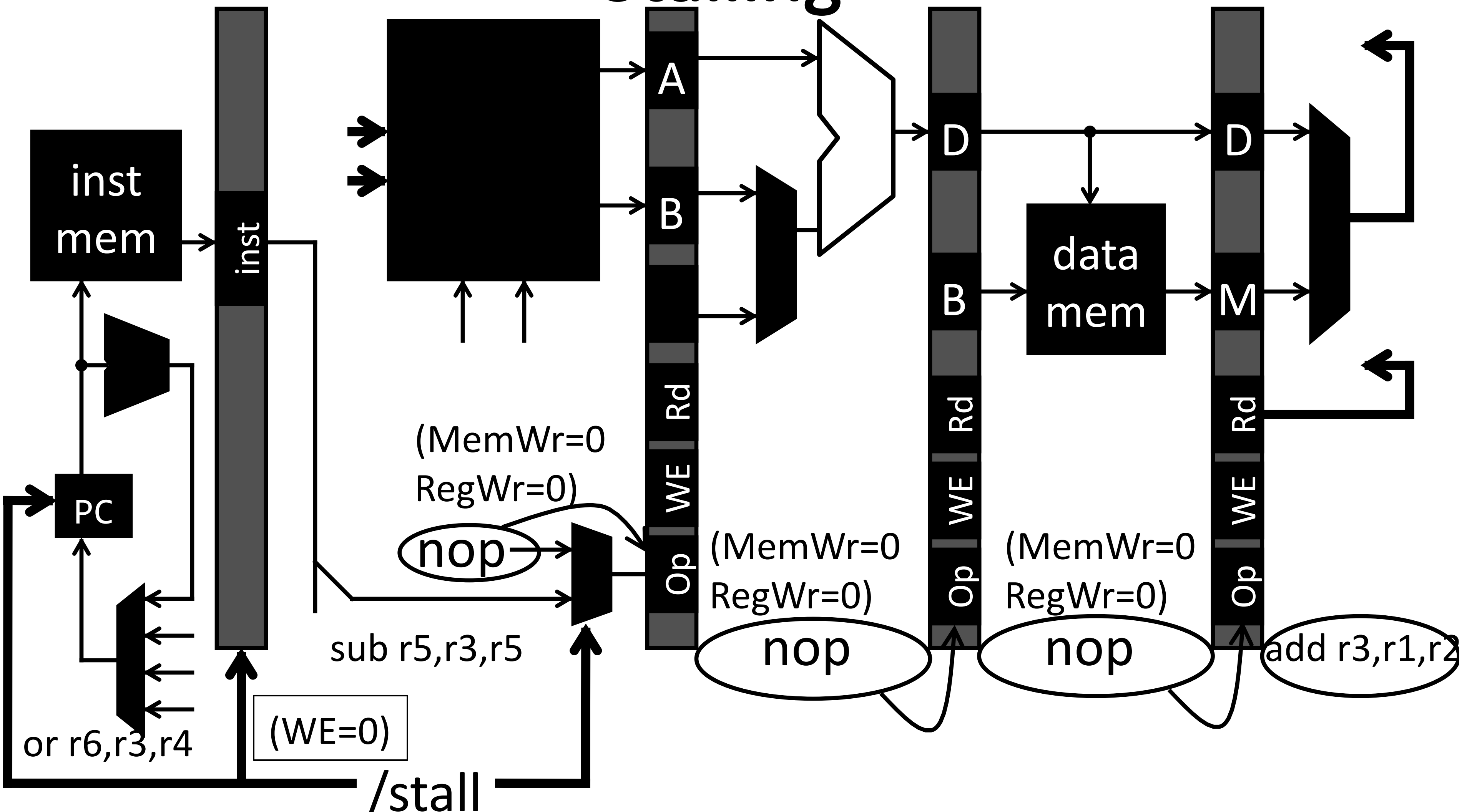


NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))

Stalling

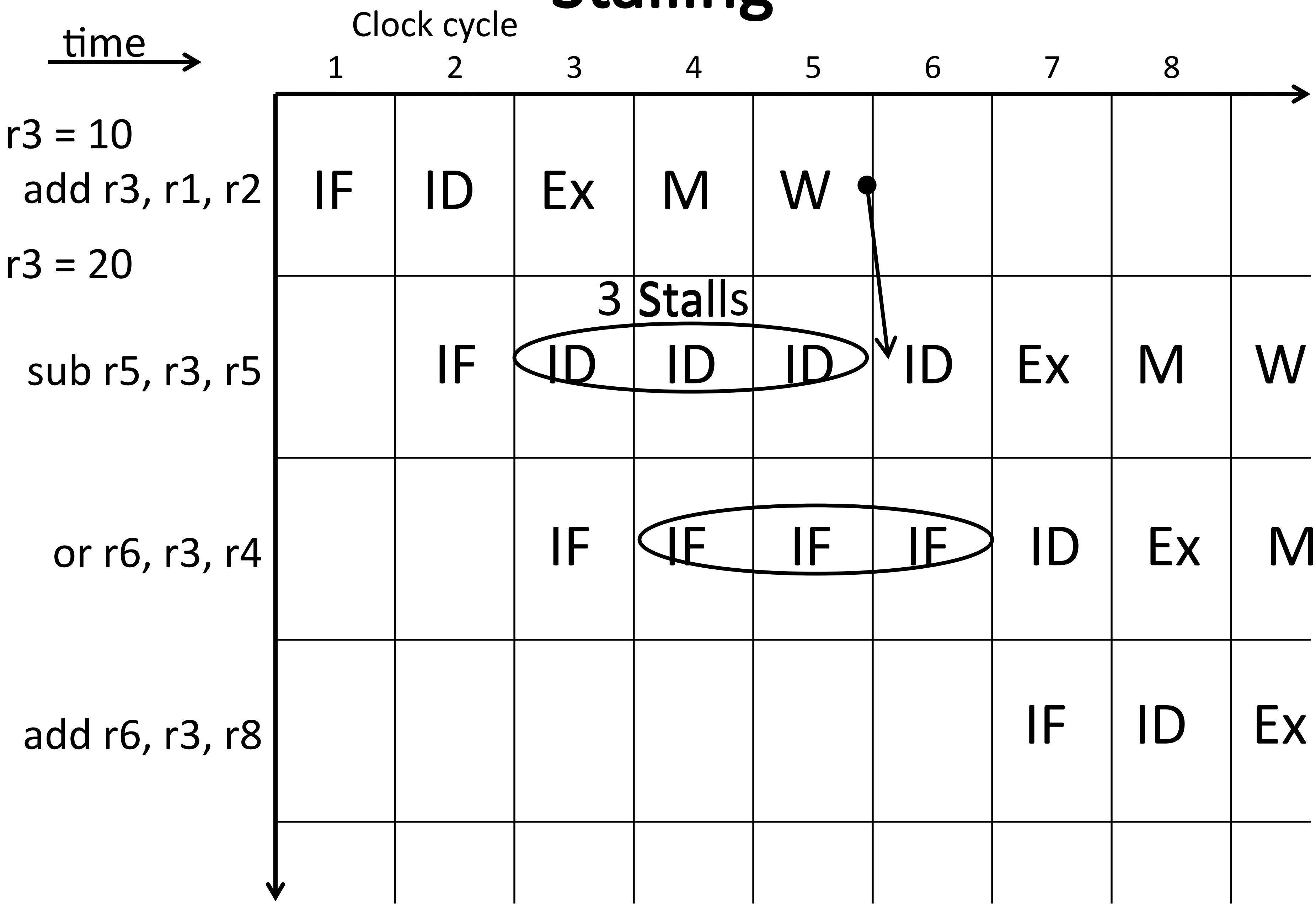

inst mem
inst
A
B
D
B
data mem
D
M
Rd
WE
Op
Rd
WE
Op
Rd
WE
Op
PC
(MemWr=0
RegWr=0)
nop
(MemWr=0
RegWr=0)
nop
add r3,r1,r2
sub r5,r3,r5
or r6,r3,r4
(WE=0)
/stall
NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd
IF/ID.rA==M/W.Rd))

Stalling



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

Stalling


Clock cycle
time
1
2
3
4
5
6
7
8
r3 = 10
add r3, r1, r2
IF ID Ex M W
r3 = 20
3 Stalls
sub r5, r3, r5
IF ID ID ID ID Ex M W
or r6, r3, r4
IF IF IF IF ID Ex M
add r6, r3, r8
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.