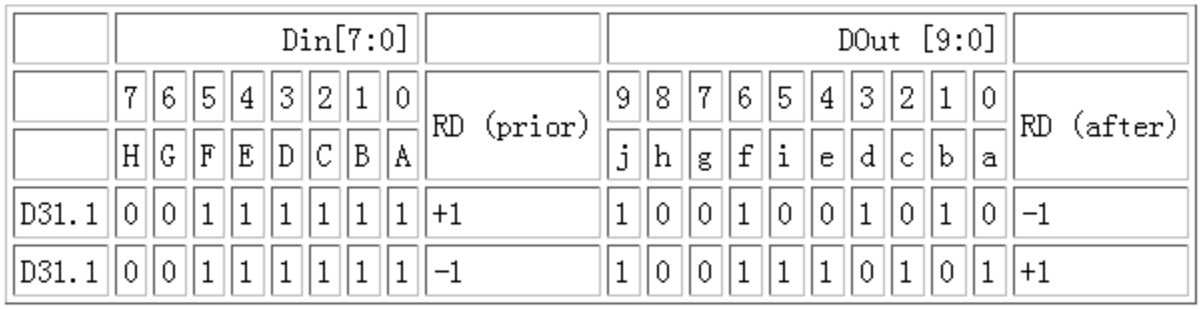
RISC Pipeline

Han Wang
CS3410, Spring 2010
Computer Science
Cornell University

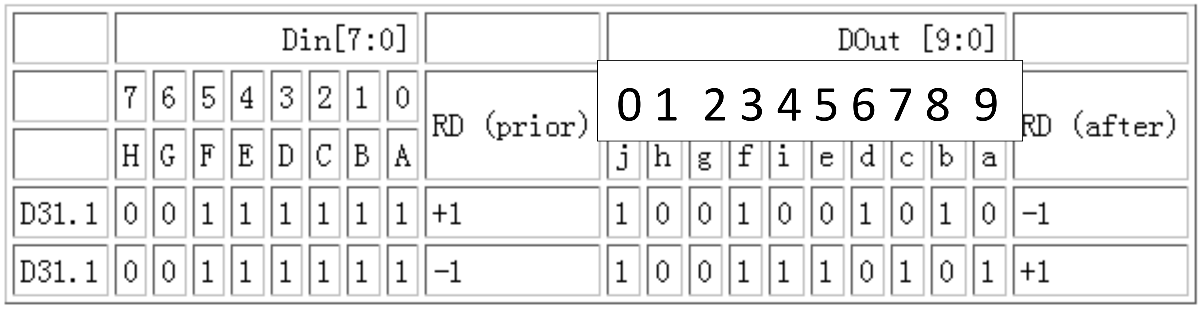
See: P&H Chapter 4.6

Homework 2



	Din[7:0]									DOut [9:0]										
	7	6	5	4	3	2	1	0	RD (prior)	9	8	7	6	5	4	3	2	1	0	RD (after)
	H	G	F	E	D	C	B	A		j	h	g	f	i	e	d	c	b	a	
D31.1	0	0	1	1	1	1	1	1	+1	1	0	0	1	0	0	1	0	1	0	-1
D31.1	0	0	1	1	1	1	1	1	-1	1	0	0	1	1	1	0	1	0	1	+1

✗

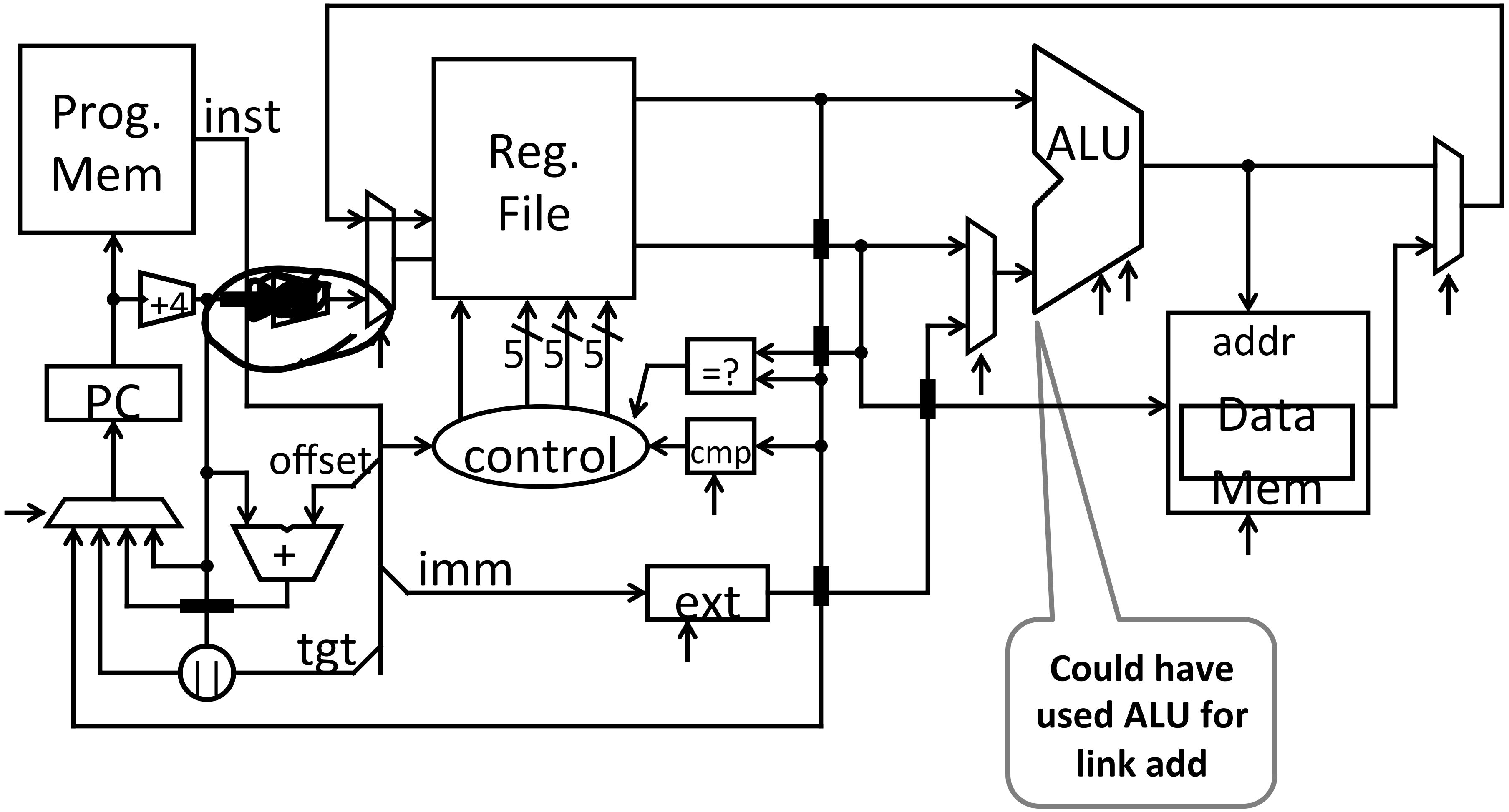


	Din[7:0]									DOut [9:0]										
	7	6	5	4	3	2	1	0	RD (prior)	0	1	2	3	4	5	6	7	8	9	RD (after)
	H	G	F	E	D	C	B	A		j	h	g	f	i	e	d	c	b	a	
D31.1	0	0	1	1	1	1	1	1	+1	1	0	0	1	0	0	1	0	1	0	-1
D31.1	0	0	1	1	1	1	1	1	-1	1	0	0	1	1	1	0	1	0	1	+1

☺

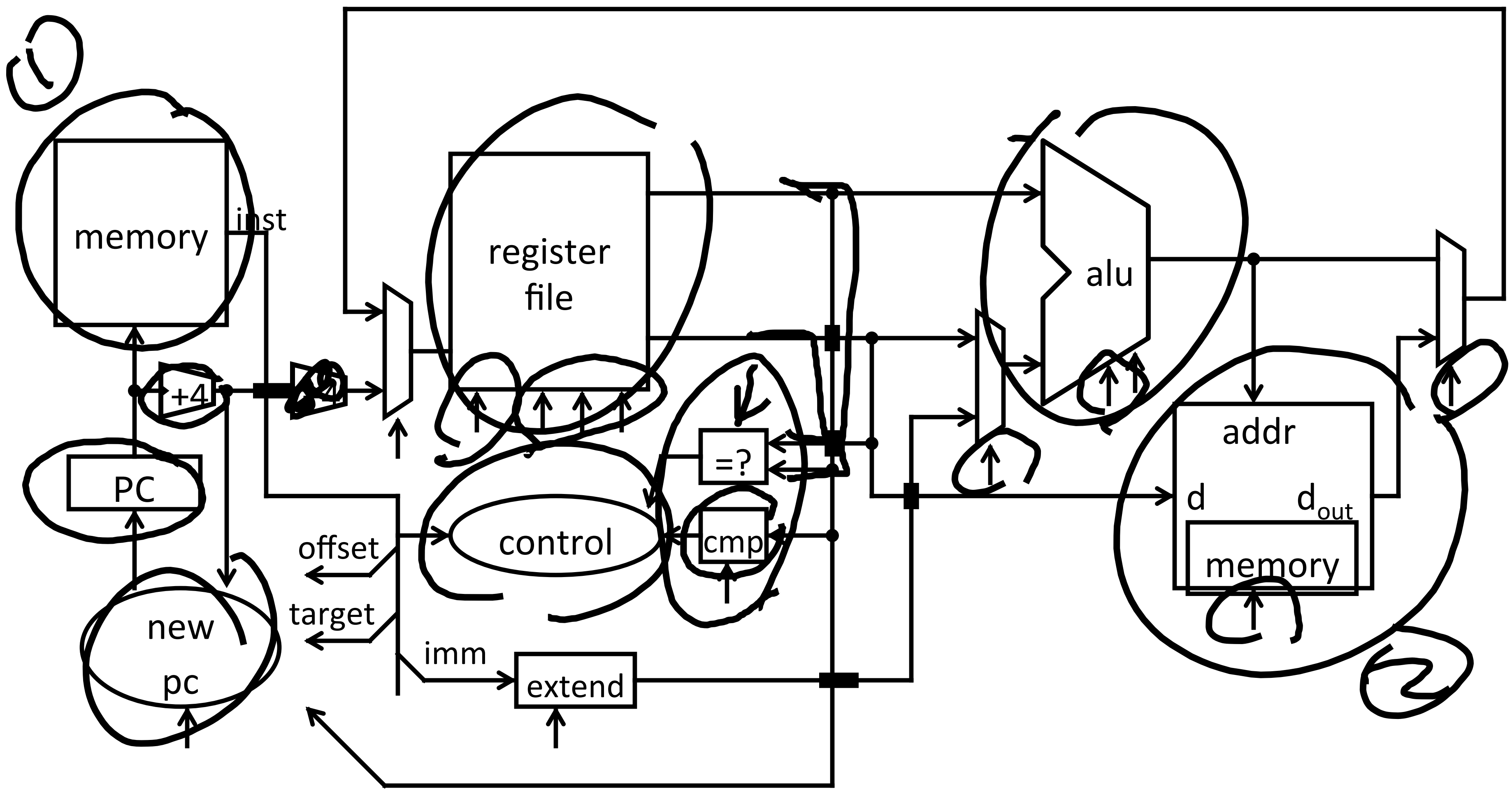
Announcements

- Homework 2 due tomorrow midnight
- Programming Assignment 1 release tomorrow
  - Pipelined MIPS processor (topic of today)
  - Subset of MIPS ISA
- Feedback
  - We want to hear from you!
  - Content?



op	mnemonic	description
0x3	JAL target	r31 = PC+8 (+8 due to branch delay slot)
		PC = (PC+4) || (target << 2)

Review: Single cycle processor



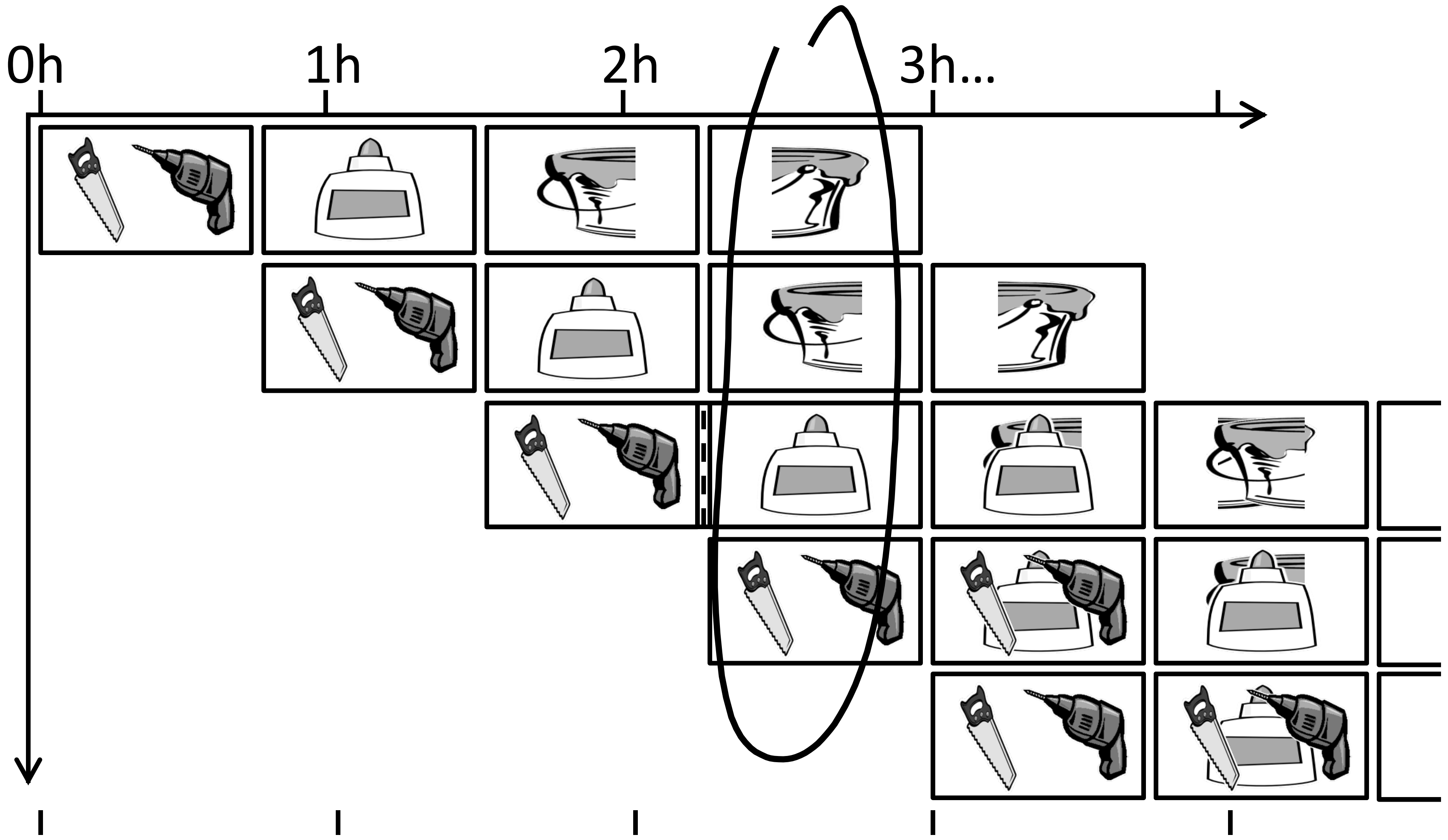
Single Cycle Processor

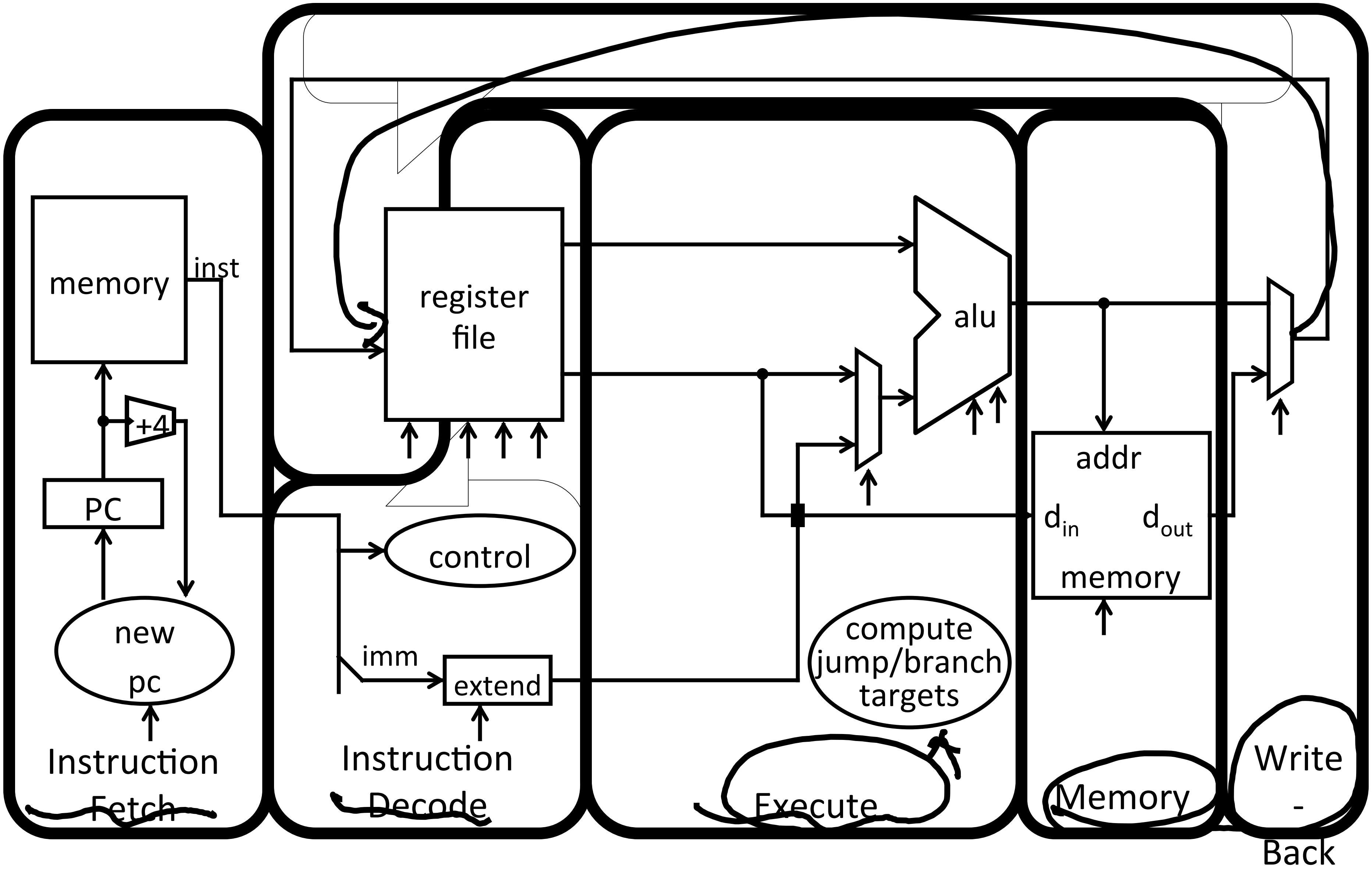
Advantages

- 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


0h
1h
2h
3h...


memory
inst
register file
alu
PC
+4
new pc
control
imm
extend
compute jump/branch targets
addr
d_{in}
d_{out}
memory
Instruction Fetch
Instruction Decode
Execute
Memory
Write - Back

Five stage “RISC” load-store architecture

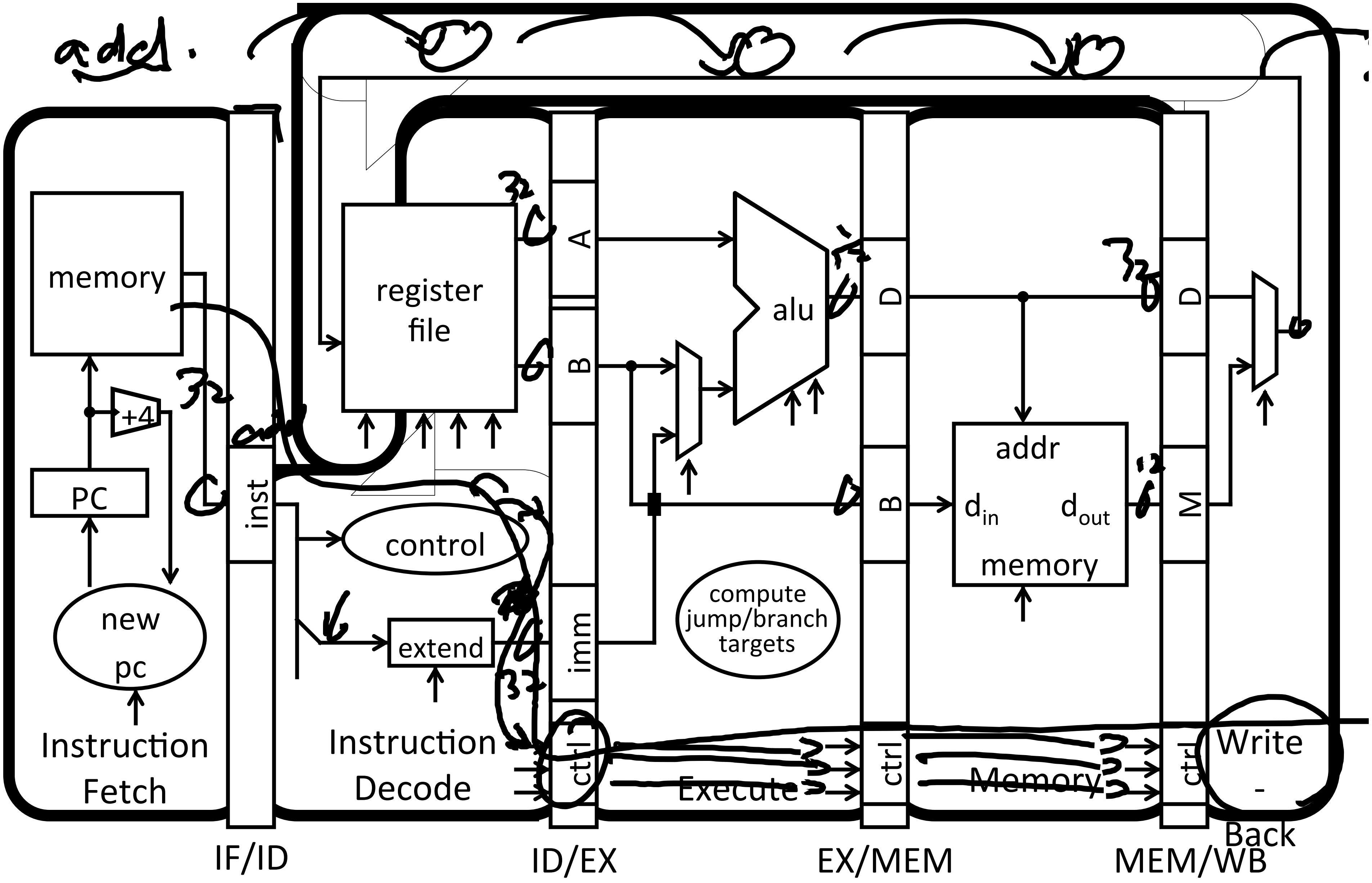
1. Instruction fetch (IF)
   - get instruction from memory, increment PC
2. Instruction Decode (ID)
   - translate opcode into control signals and read registers
3. Execute (EX)
   - perform ALU operation, compute jump/branch targets
4. Memory (MEM)
   - access memory if needed
5. Writeback (WB)
   -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

Slides thanks to Sally McKee & Kavita Bala

Break instructions across multiple clock cycles (five, in this case)

Design a separate stage for the execution performed during each clock cycle

Add pipeline registers to isolate signals between different stages


add.
memory
register file
A
B
alu
D
B
addr
d_{in}
d_{out}
memory
D
M
PC
+4
new pc
inst
control
extend
imm
compute jump/branch targets
ctrl
ctrl
ctrl
Instruction Fetch
Instruction Decode
Execute
Memory
Write - Back
IF/ID
ID/EX
EX/MEM
MEM/WB
32
32
32
32
32

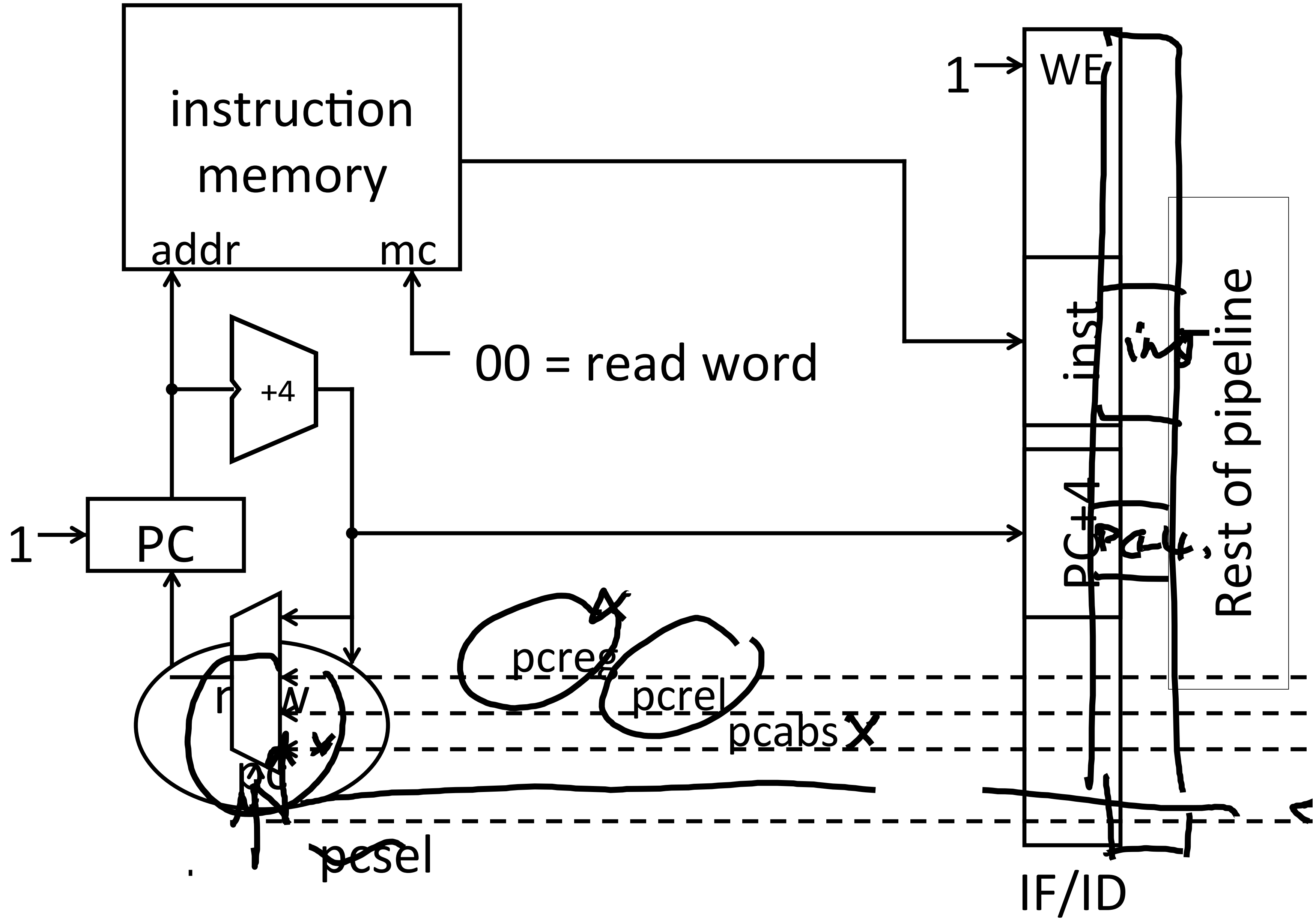
Stage 1: Instruction Fetch

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 memory
addr
mc
00 = read word
+4
1
PC
new
pc
pcsel
pcreg
pcrel
pcabs
1
WE
inst
PC+4
ins
PC+4
Rest of pipeline
IF/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

add r2, r1 + , 100

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

- Control information, Rd index, immediates, offsets, …

  r2 100.

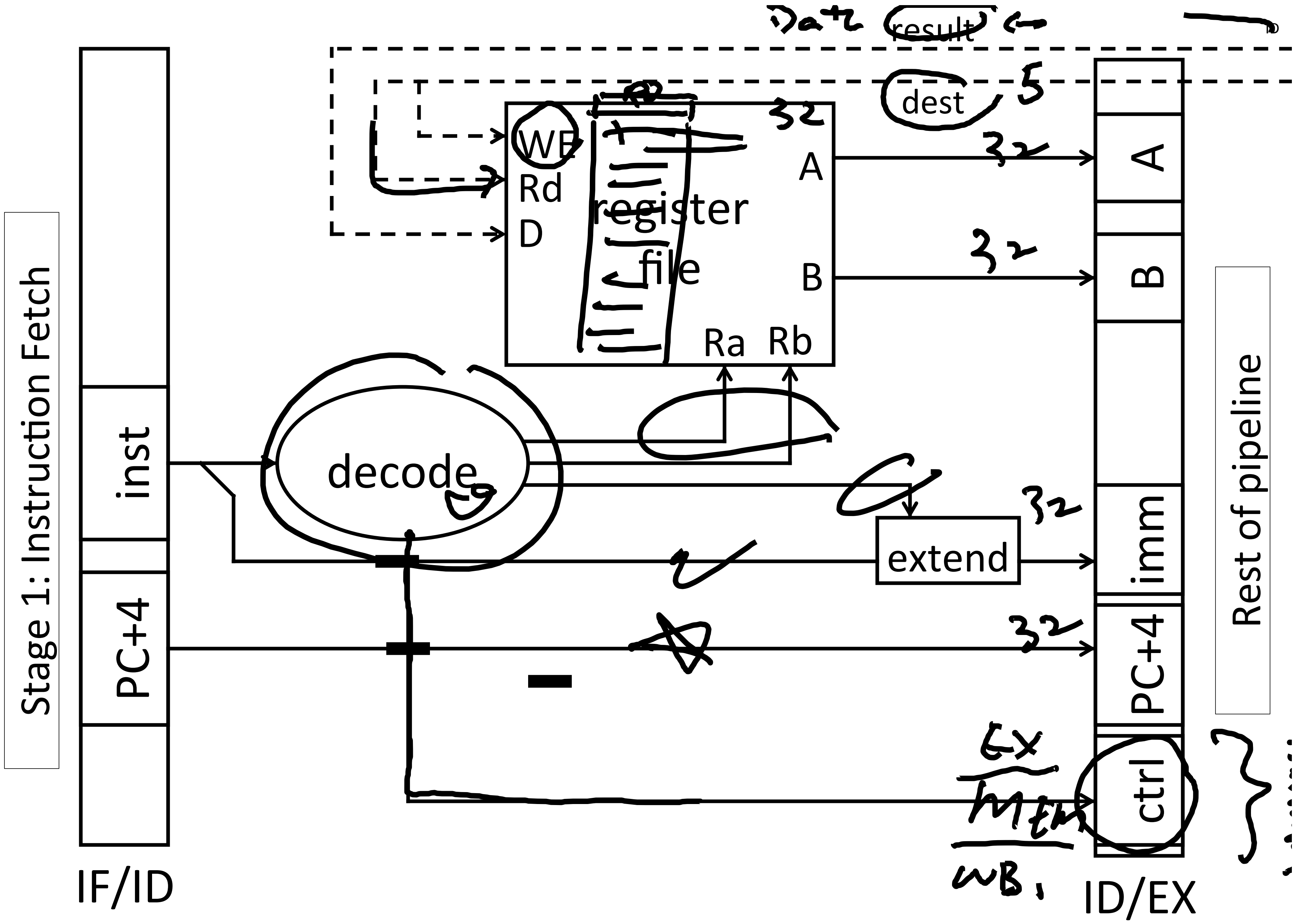
- Contents of Ra, Rb
- PC+4 (for computing branch targets later)

addU R3 , R2 , R1

Ra Rb

PC+4

IF/ID



Data
result
dest
5
32
WE
Rd
D
register file
A
B
Ra
Rb
decode
extend
inst
PC+4
IF/ID
Stage 1: Instruction Fetch
A
B
imm
PC+4
ctrl
ID/EX
Rest of pipeline
EX
MEM
WB.

Stage 3: Execute

Jump. R1, 0x10000100

=
0

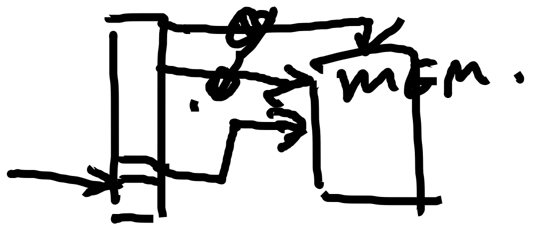
On every cycle:

- 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

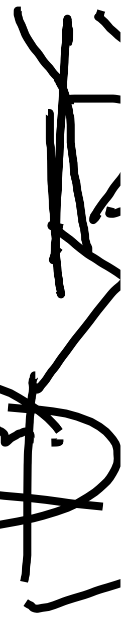
Branch Prediction:

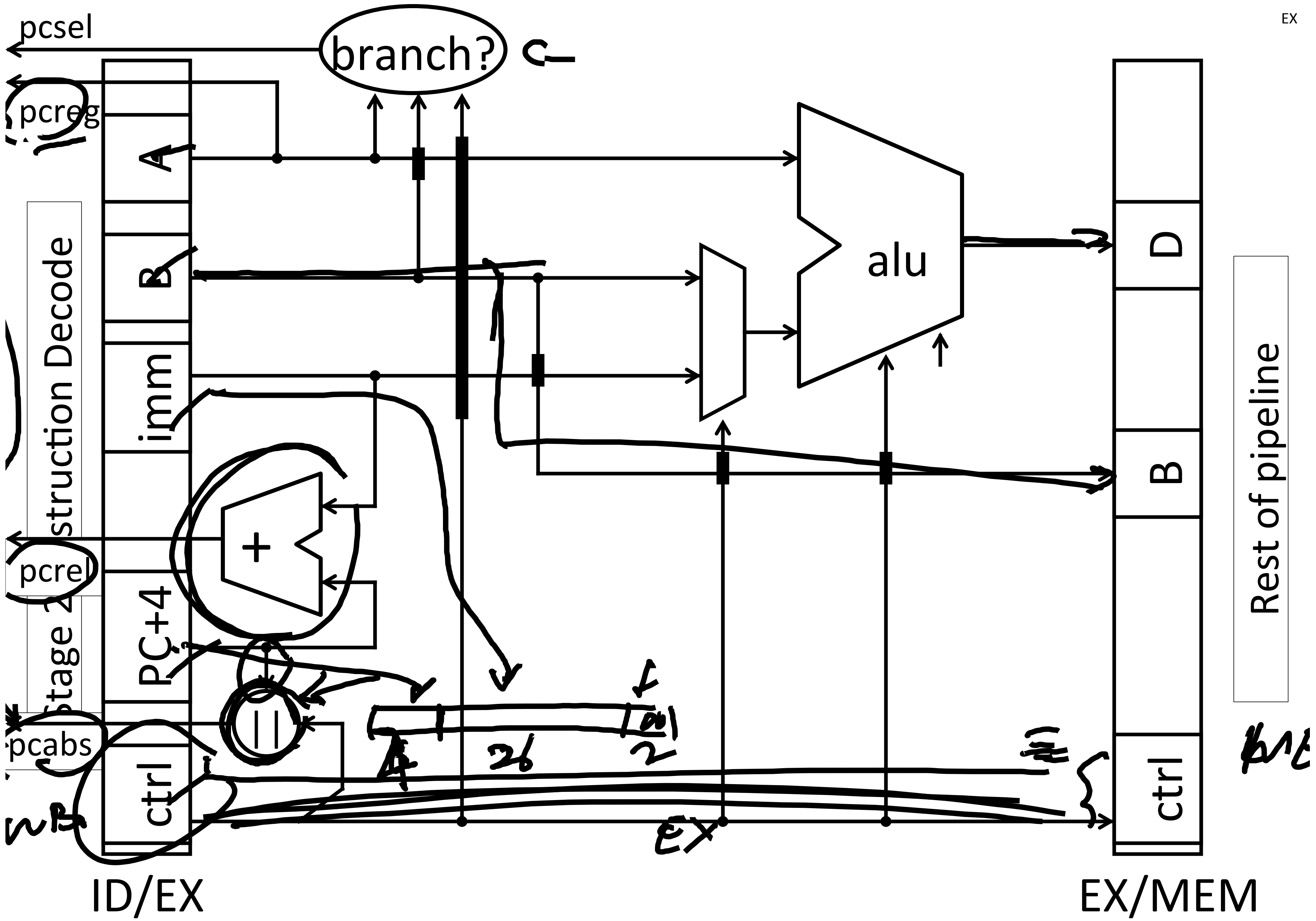
Write values of interest to pipeline register (EX/MEM)

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



SW R2, R1 MEM[




pcsel
branch?
pcreg
A
B
imm
PC+4
ctrl
pcrel
pcabs
+
||
alu
D
B
ctrl
Stage 2: Instruction Decode
Rest of pipeline
26
2
EX
ID/EX
EX/MEM

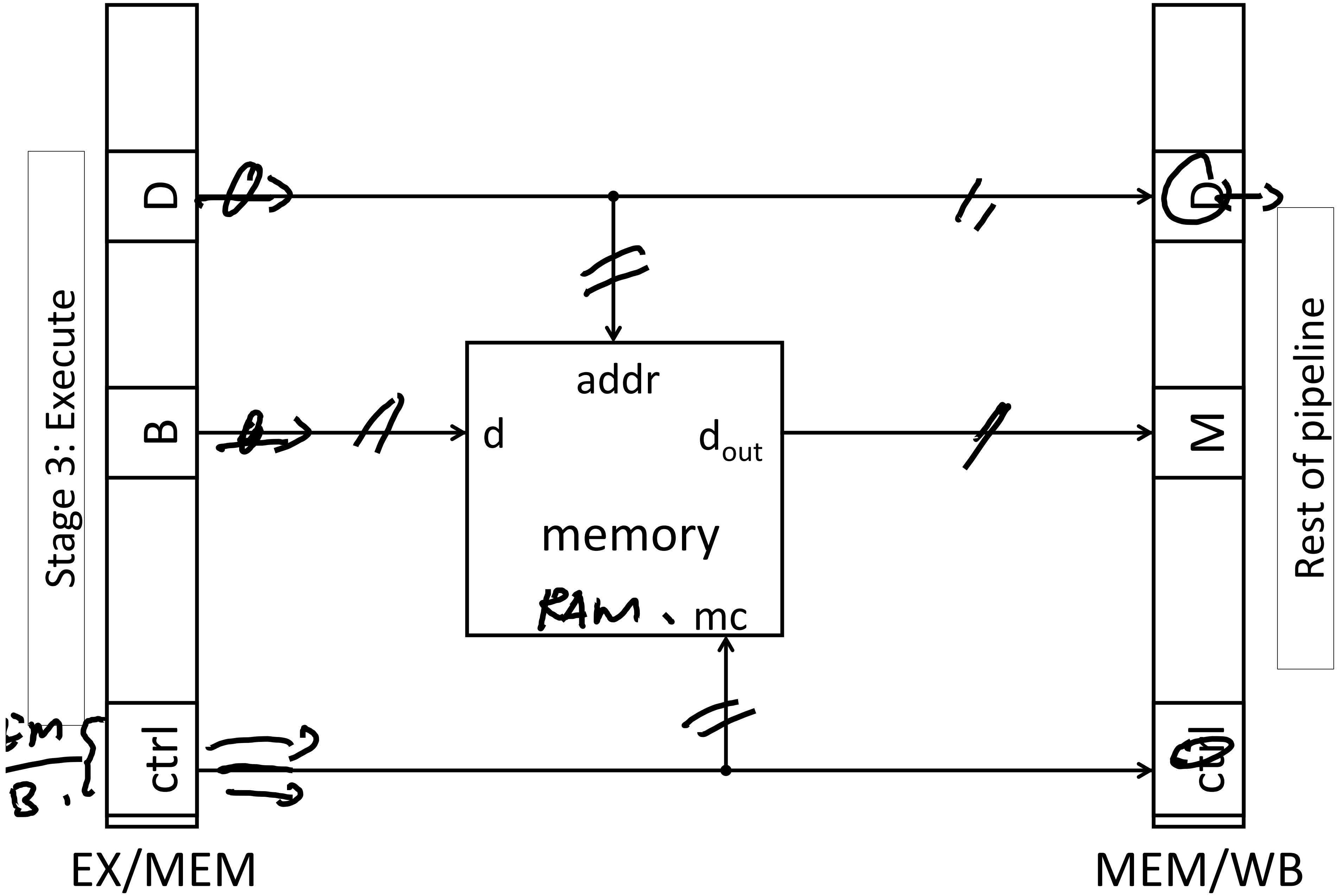
Stage 4: Memory

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)

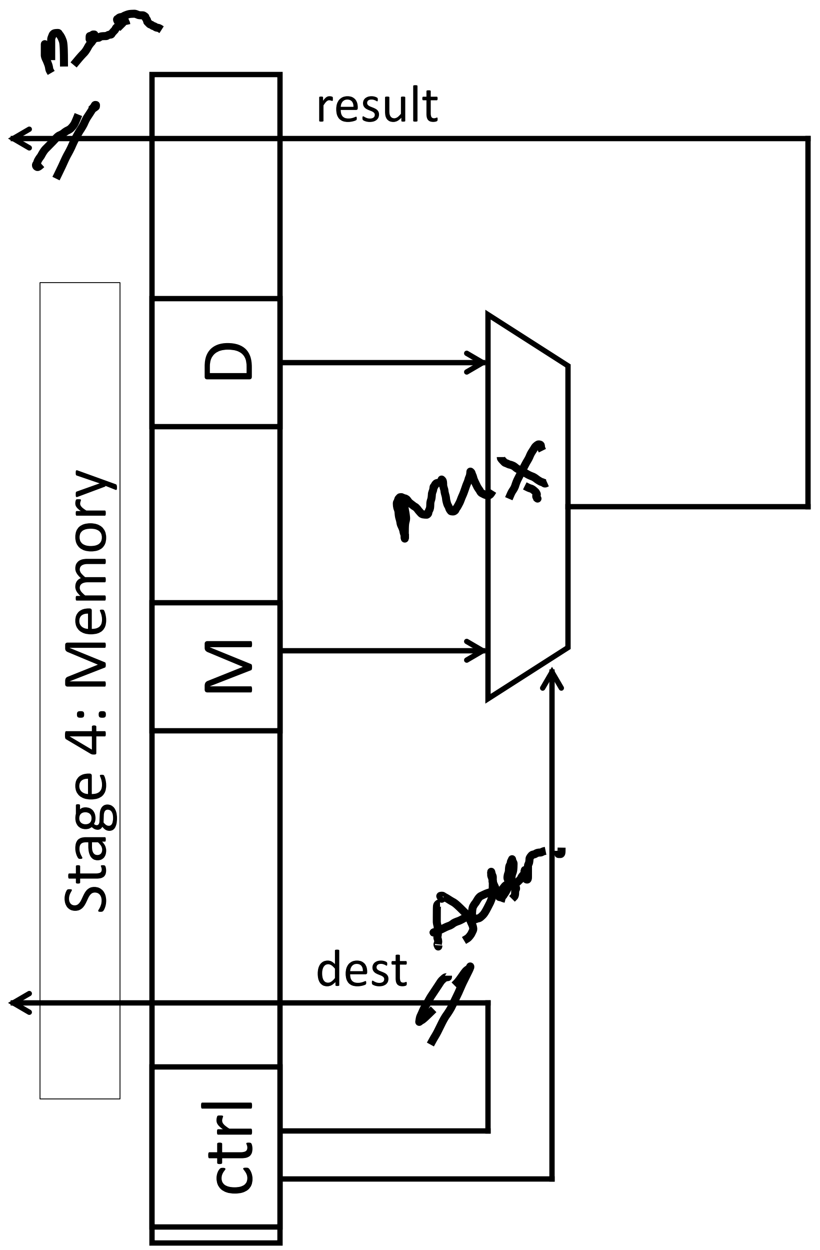
- Control information, Rd index, …
- Result of memory operation ←
- Pass result of ALU operation ←


Stage 3: Execute
D
B
ctrl
EX/MEM
addr
d
d_{out}
memory
RAM.
mc
D
M
ctrl
MEM/WB
Rest of pipeline

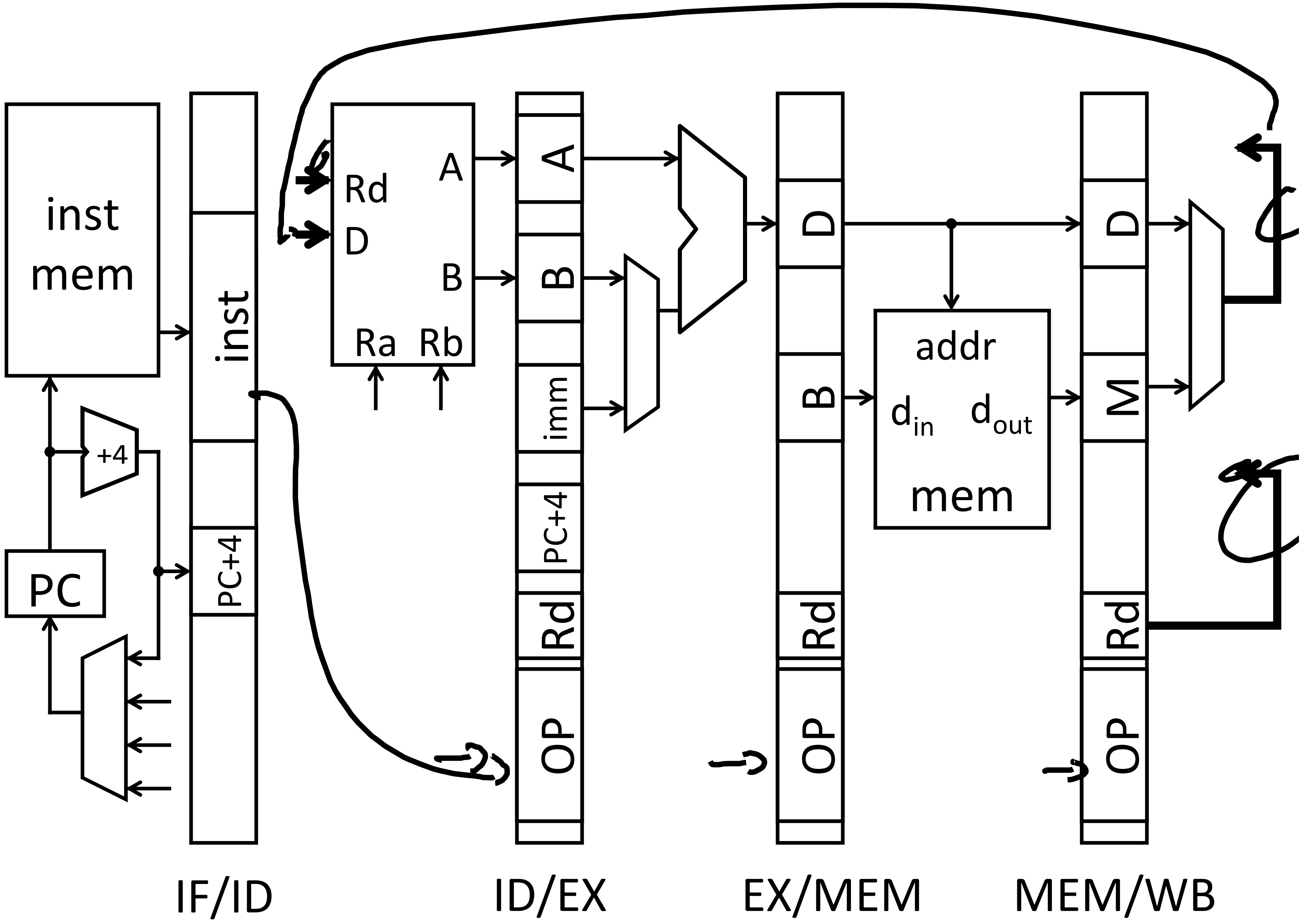
Stage 5: Write-back

On every cycle:

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



MEM/WB



inst mem
inst
PC+4
PC
+4
Rd
D
A
B
Ra
Rb
A
B
imm
PC+4
Rd
OP
D
B
Rd
OP
addr
d_{in}
d_{out}
mem
D
M
Rd
OP
IF/ID
ID/EX
EX/MEM
MEM/WB

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

1
2
5

$r3 = r1 + r2 = 45$

$r6 = \sim(\underbrace{r4}_{18} \,\&\, \underbrace{r5}_{7}) = 111$

3

$= -$

…0010010

…00111

~~0000000~~

~~000~~

~~00010~~

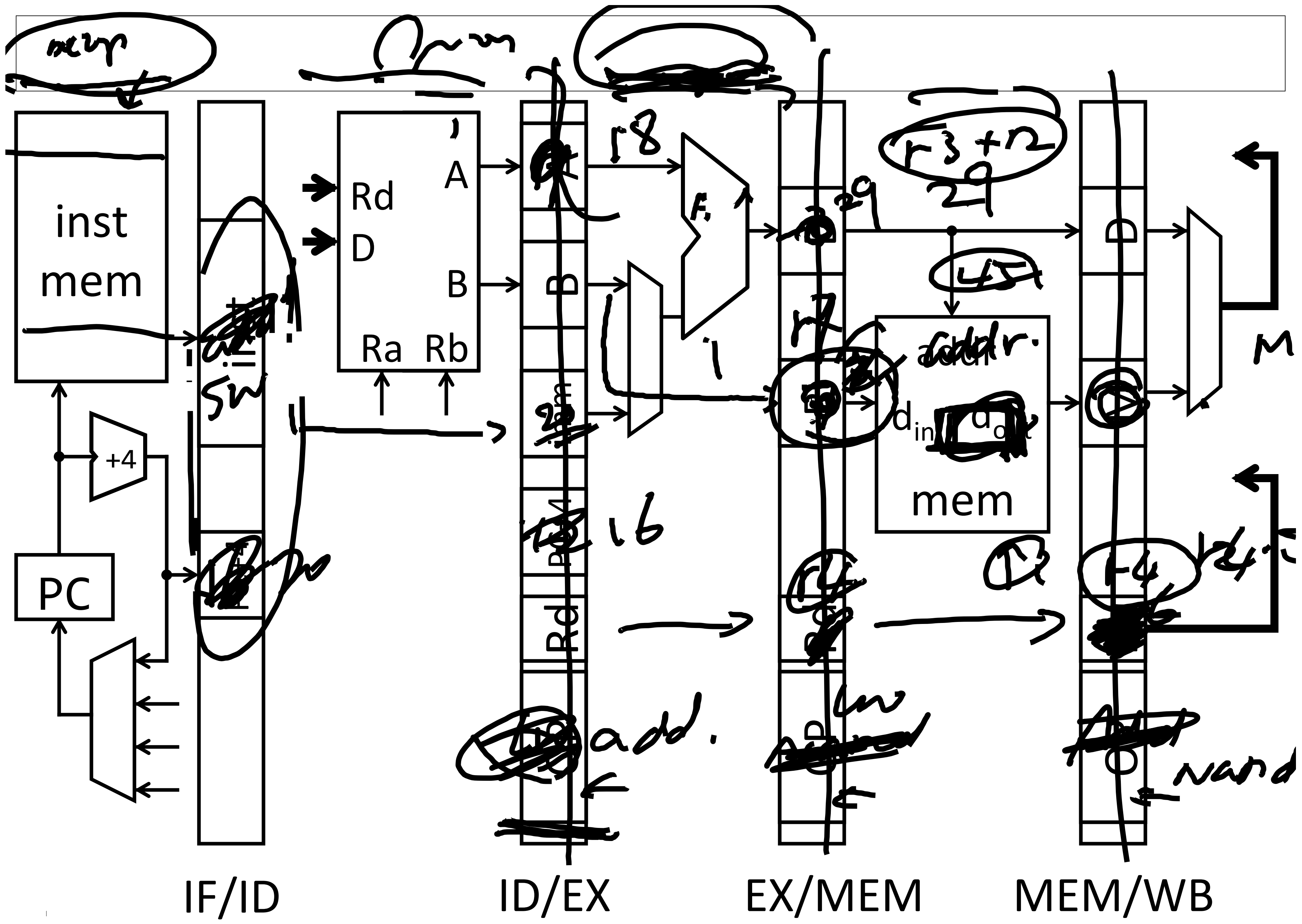
$\sim$ 00010

…111101

$r4 = MEM[r2 + 20]$

$r5 = \underbrace{r2}_{9} + \underbrace{r5}_{7} = 16$

$MEM[r3 + r2] = \underbrace{r7}_{77}$


inst
mem
PC
+4
Rd
D
A
B
Ra
Rb
imm
PC+4
Rd
OP
addr
d_{in}
d_{out}
mem
IF/ID
ID/EX
EX/MEM
MEM/WB
r8
r3+r2
29
45
r7
r4
16
add.
lw
nand
sw
M
addr.

Clock cycle

	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

Throughput: 1

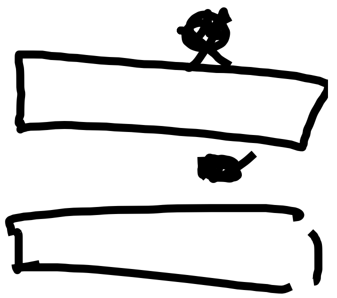
Concurrency: 5

CPI = 1

Powerful technique for masking latencies

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

assumption



Abstraction promotes decoupling

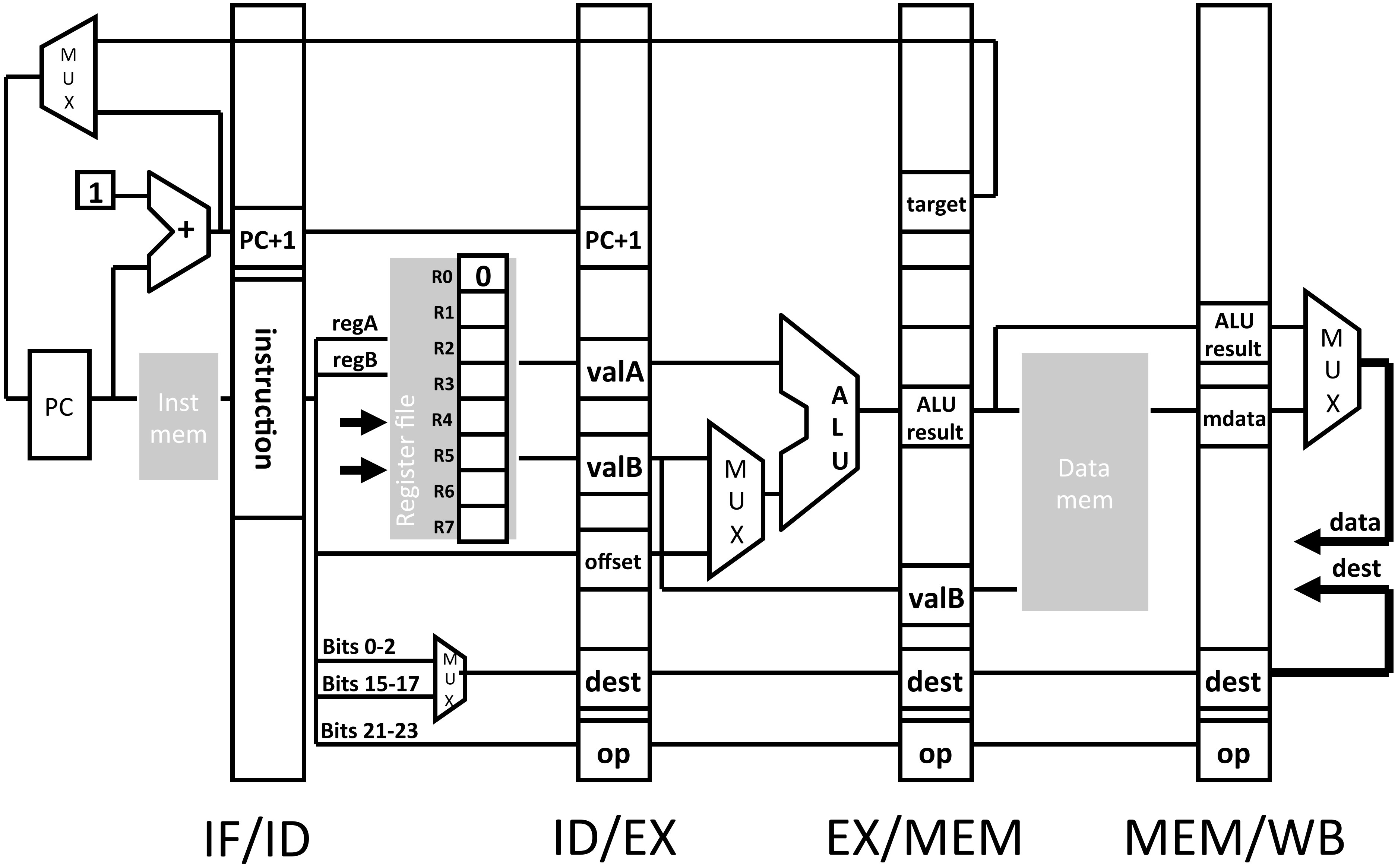
- Interface (ISA) vs. implementation (Pipeline)

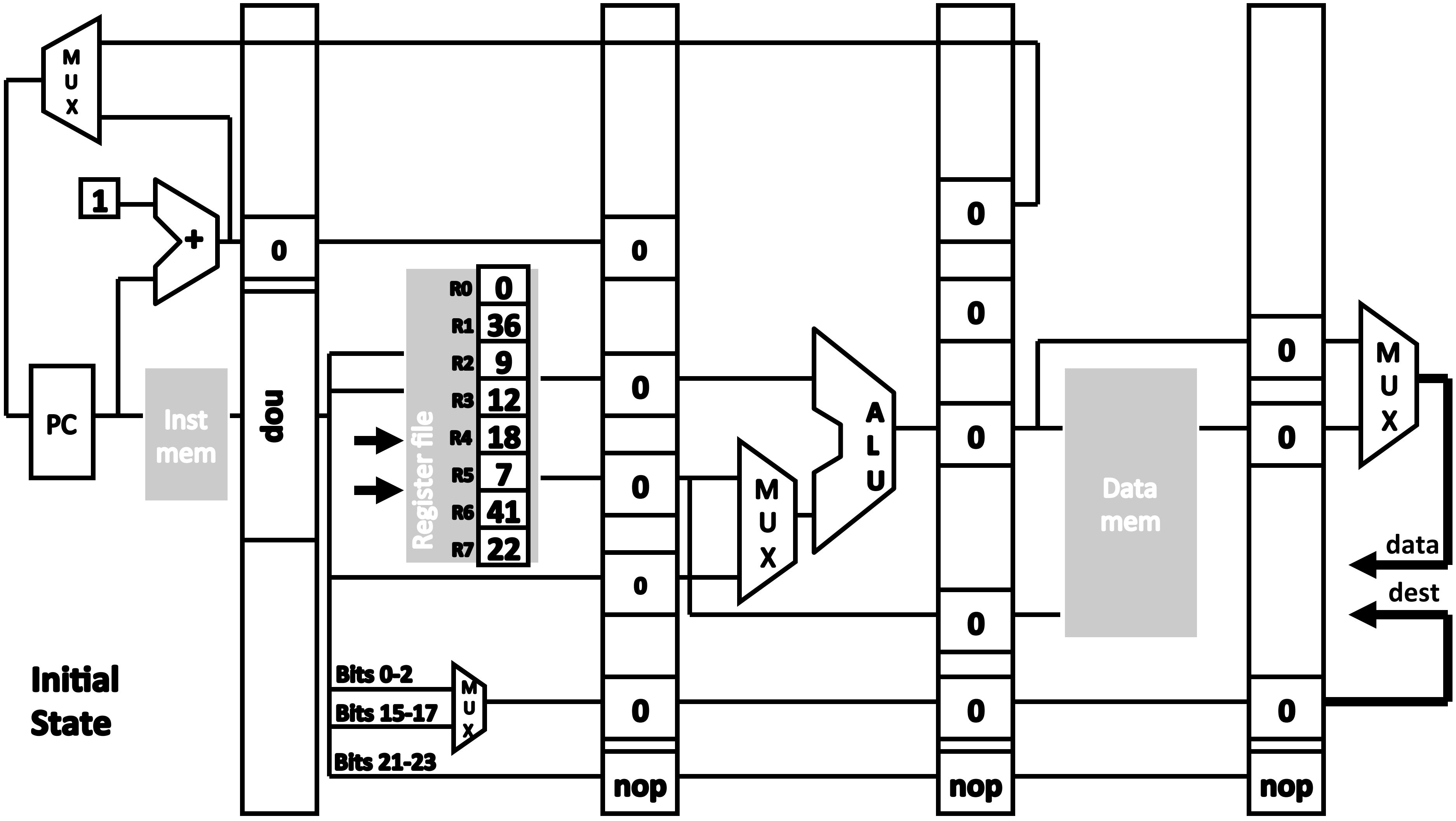
The end

Assume eight-register machine

Run the following code on a pipelined datapath

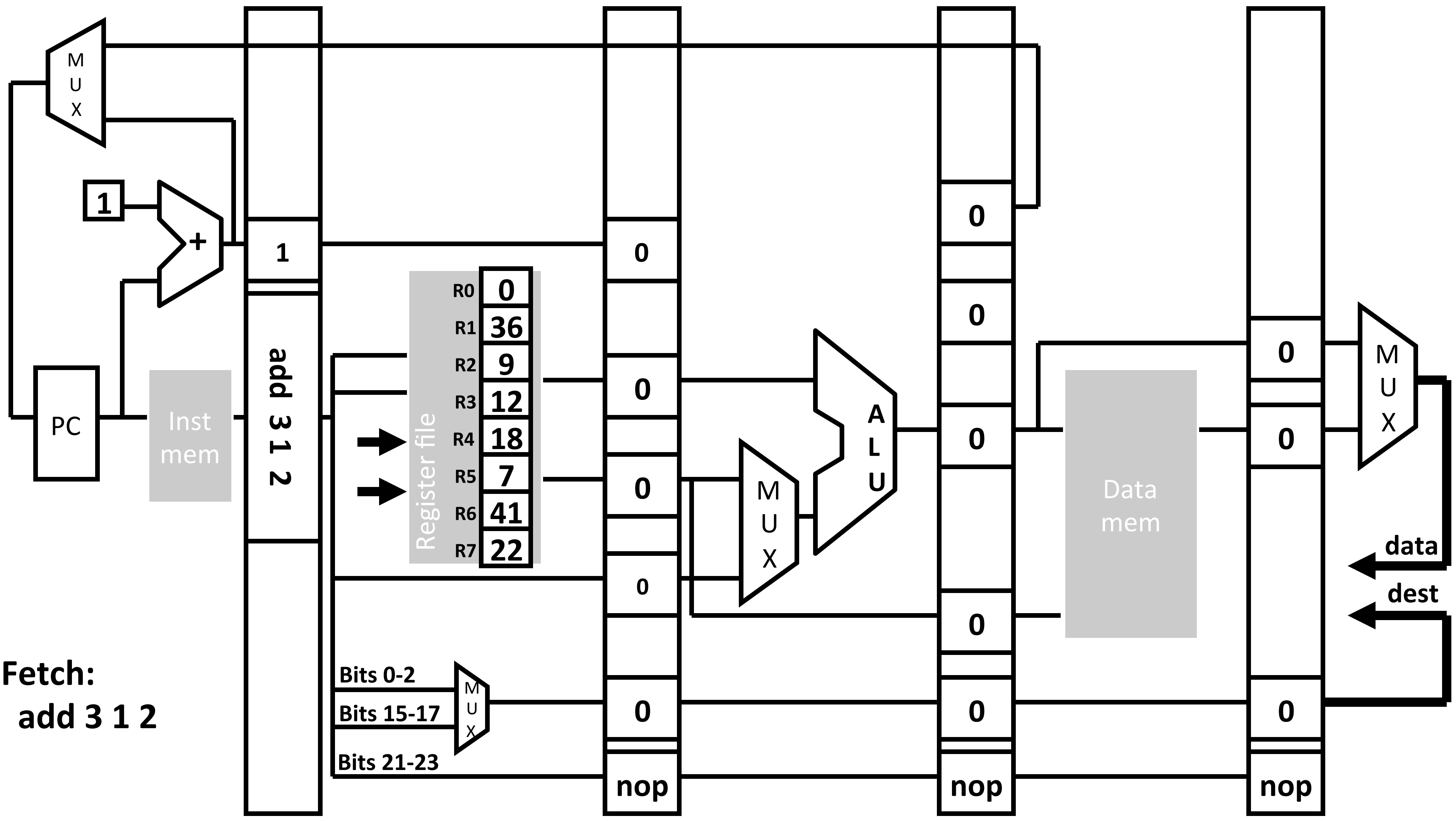
```
add       3  1   2   ;  reg 3 = reg 1 + reg 2
nand      6  4   5   ;  reg 6 = ~(reg 4 & reg 5)
lw  4  20 (2)  ;  reg 4 =  Mem[reg2+20]
add       5  2   5   ;  reg 5 = reg 2 + reg 5
sw       7   12(3)   ;  Mem[reg3+12] = reg 7
```


M U X
1
+
PC+1
PC
Inst mem
instruction
regA
regB
Register file
R0
0
R1
R2
R3
R4
R5
R6
R7
PC+1
valA
valB
offset
M U X
A L U
target
ALU result
valB
Data mem
ALU result
mdata
M U X
data
dest
Bits 0-2
Bits 15-17
Bits 21-23
M U X
dest
dest
dest
op
op
op
IF/ID
ID/EX
EX/MEM
MEM/WB



MUX
1
+
0
PC
Inst mem
nop
Register file
R0 0
R1 36
R2 9
R3 12
R4 18
R5 7
R6 41
R7 22
0
0
0
0
0
nop
ALU
MUX
0
0
0
0
0
nop
Data mem
0
0
0
nop
MUX
data
dest
Bits 0-2
Bits 15-17
Bits 21-23
MUX
Initial State
IF/ID
ID/EX
EX/MEM
MEM/WB

add 3 1 2

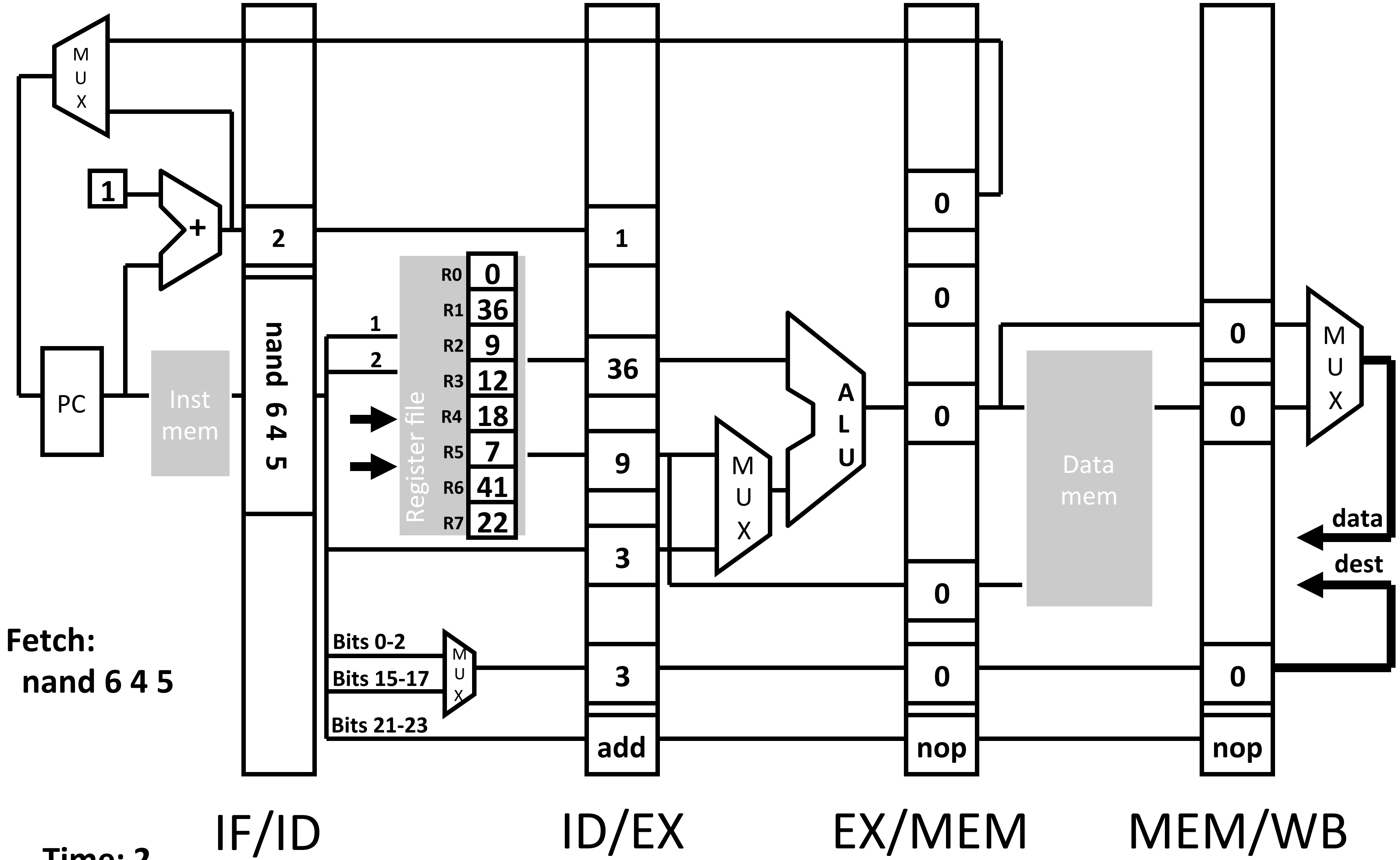


Fetch:
add 3 1 2

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

Time: 1

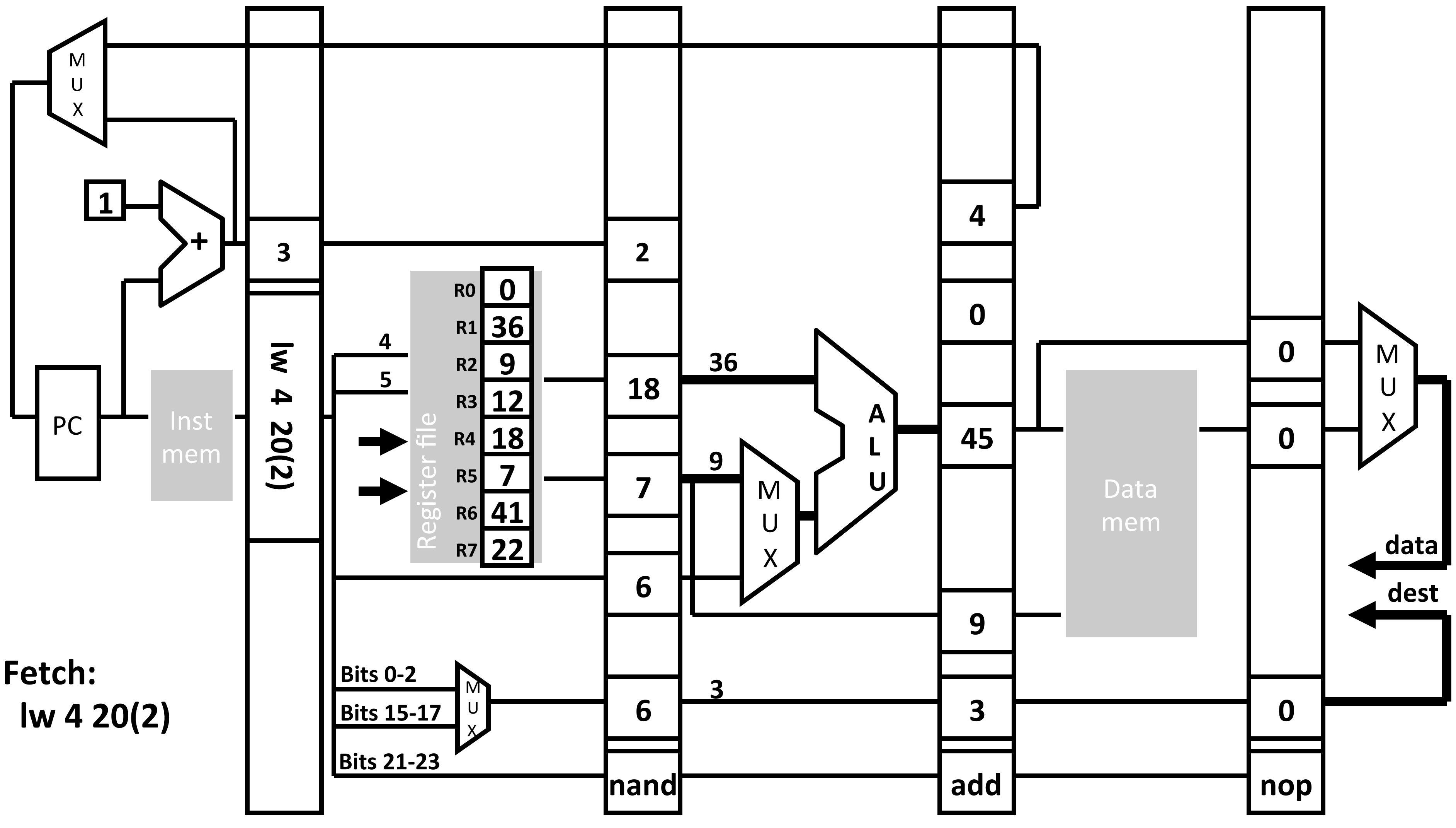
nand 6 4 5 **add 3 1 2**



Fetch:
nand 6 4 5

Time: 2

lw 4 20(2)　　nand 6 4 5　　add 3 1 2

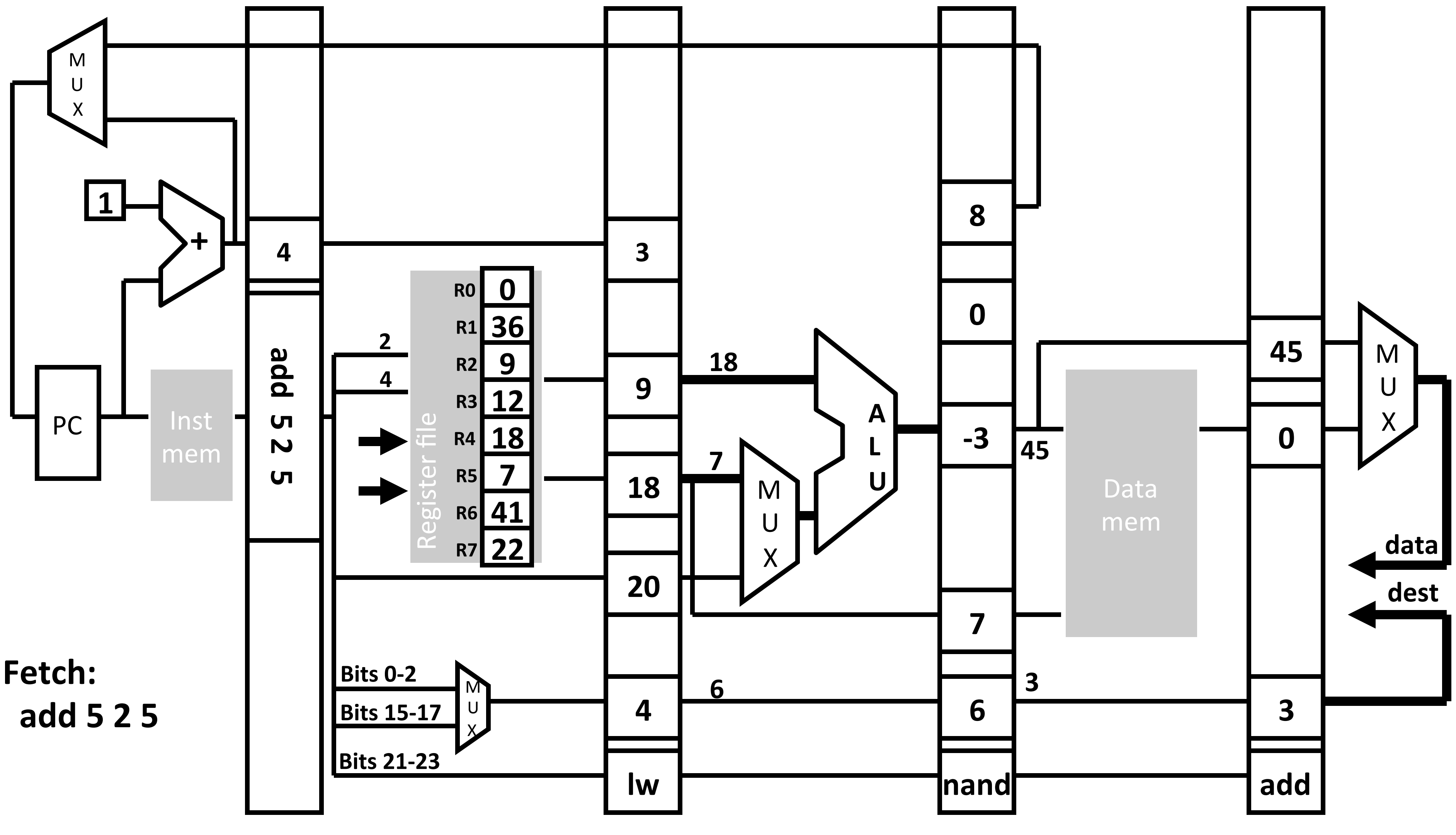


Fetch:
lw 4 20(2)

IF/ID　　ID/EX　　EX/MEM　　MEM/WB

Time: 3

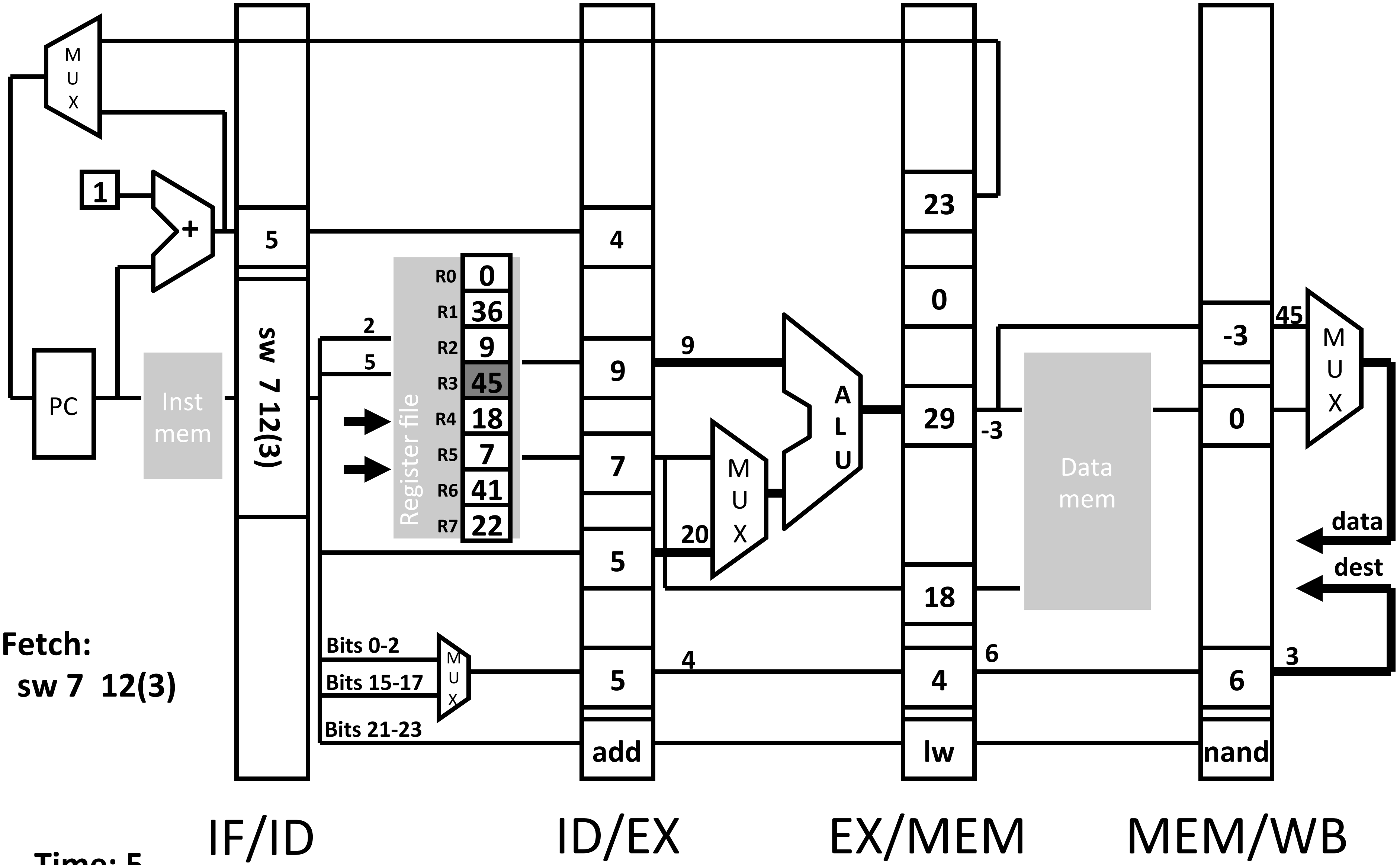
add 5 2 5 **lw 4 20(2)** **nand 6 4 5** **add 3 1 2**



Fetch:
add 5 2 5

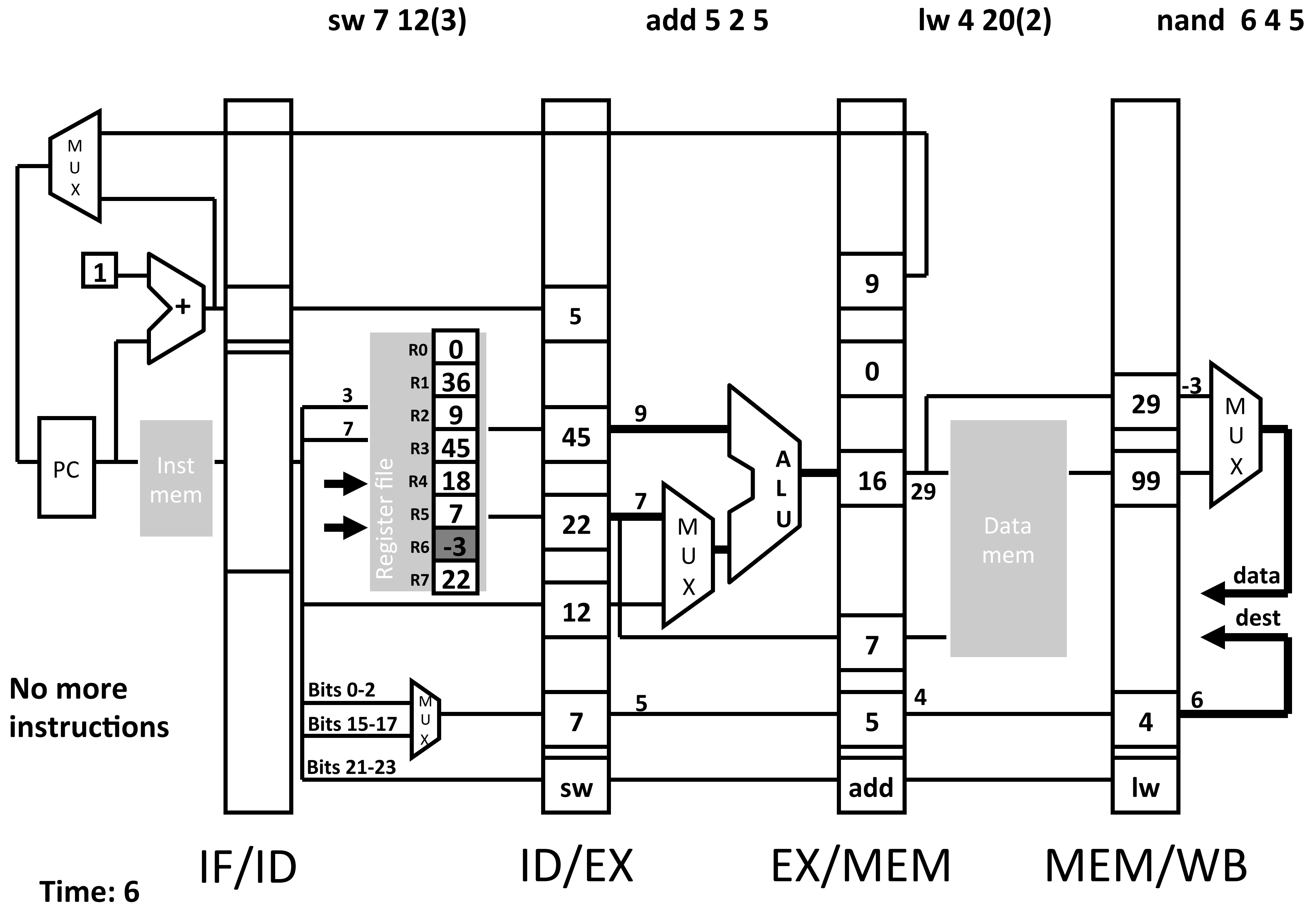
Time: 4

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

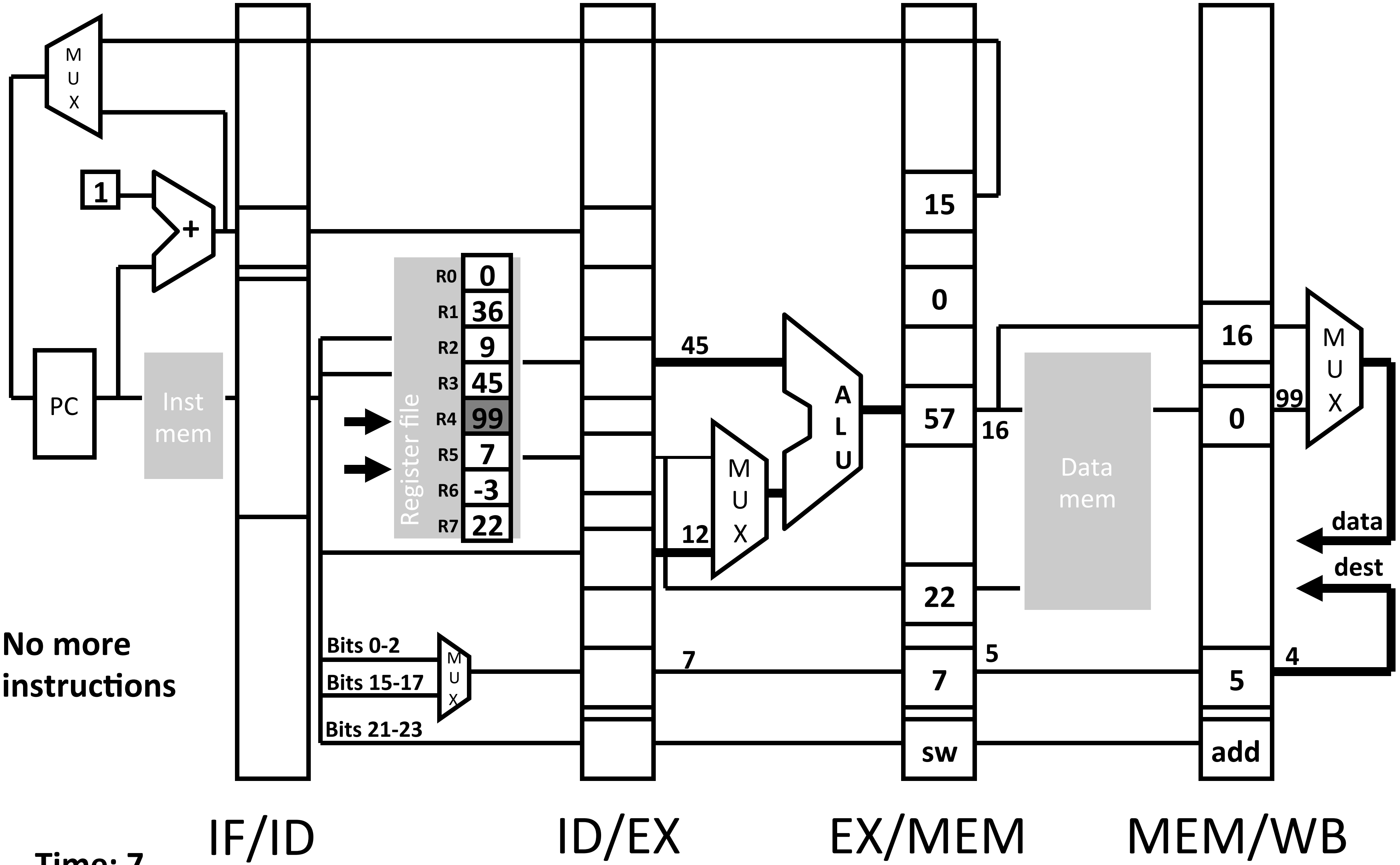


Fetch:
sw 7 12(3)

Time: 5


sw 7 12(3)
add 5 2 5
lw 4 20(2)
nand 6 4 5
M U X
1
+
PC
Inst mem
Register file
R0 0
R1 36
R2 9
R3 45
R4 18
R5 7
R6 -3
R7 22
3
7
5
45
22
12
7
sw
9
7
M U X
A L U
9
0
16
7
5
add
29
Data mem
4
29
99
4
lw
-3
M U X
data
dest
6
Bits 0-2
Bits 15-17
Bits 21-23
M U X
No more instructions
IF/ID
ID/EX
EX/MEM
MEM/WB
Time: 6

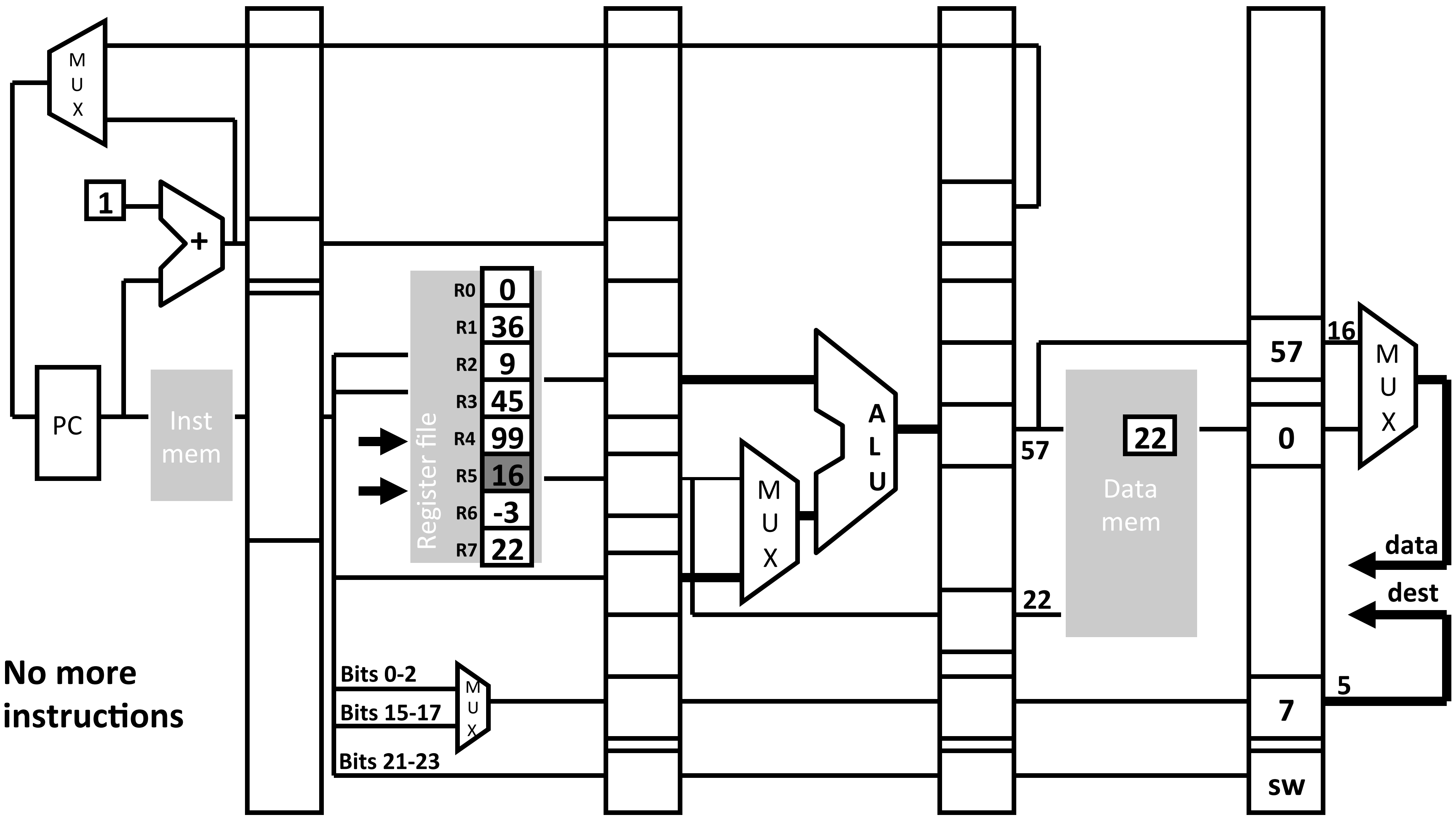
nop nop sw 7 12(3) add 5 2 5 lw 4 20(2)



No more instructions

Time: 7

nop nop nop sw 7 12(3) add 5 2 5

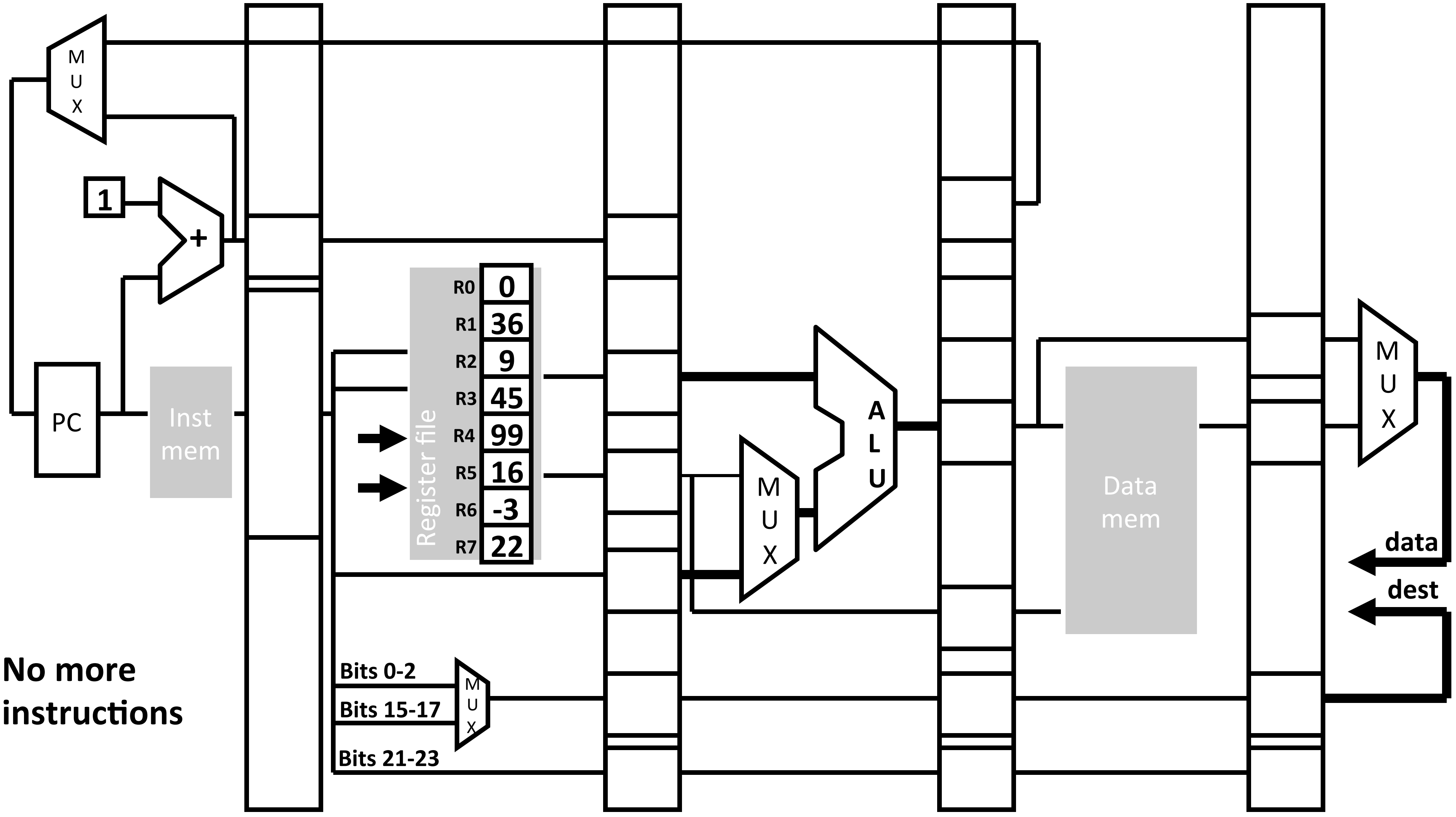


No more instructions

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

Time: 8

nop **nop** **nop** **nop** **sw 7 12(3)**



No more instructions

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

Time: 9