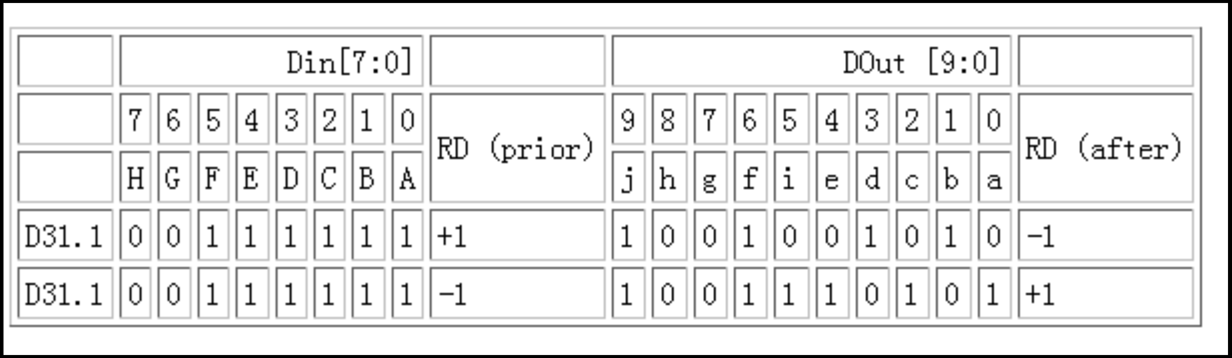
RISC Pipeline

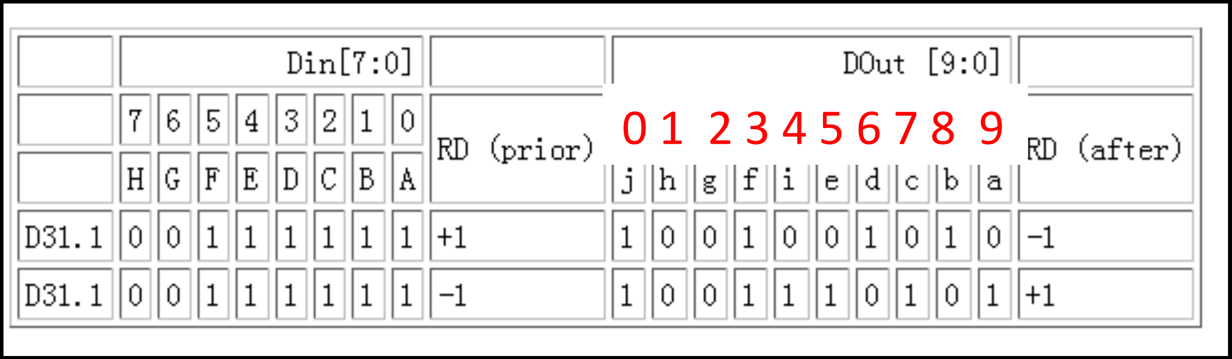
Han Wang
CS3410, Spring 2010
Computer Science
Cornell University

See: P&H Chapter 4.6

Homework 2



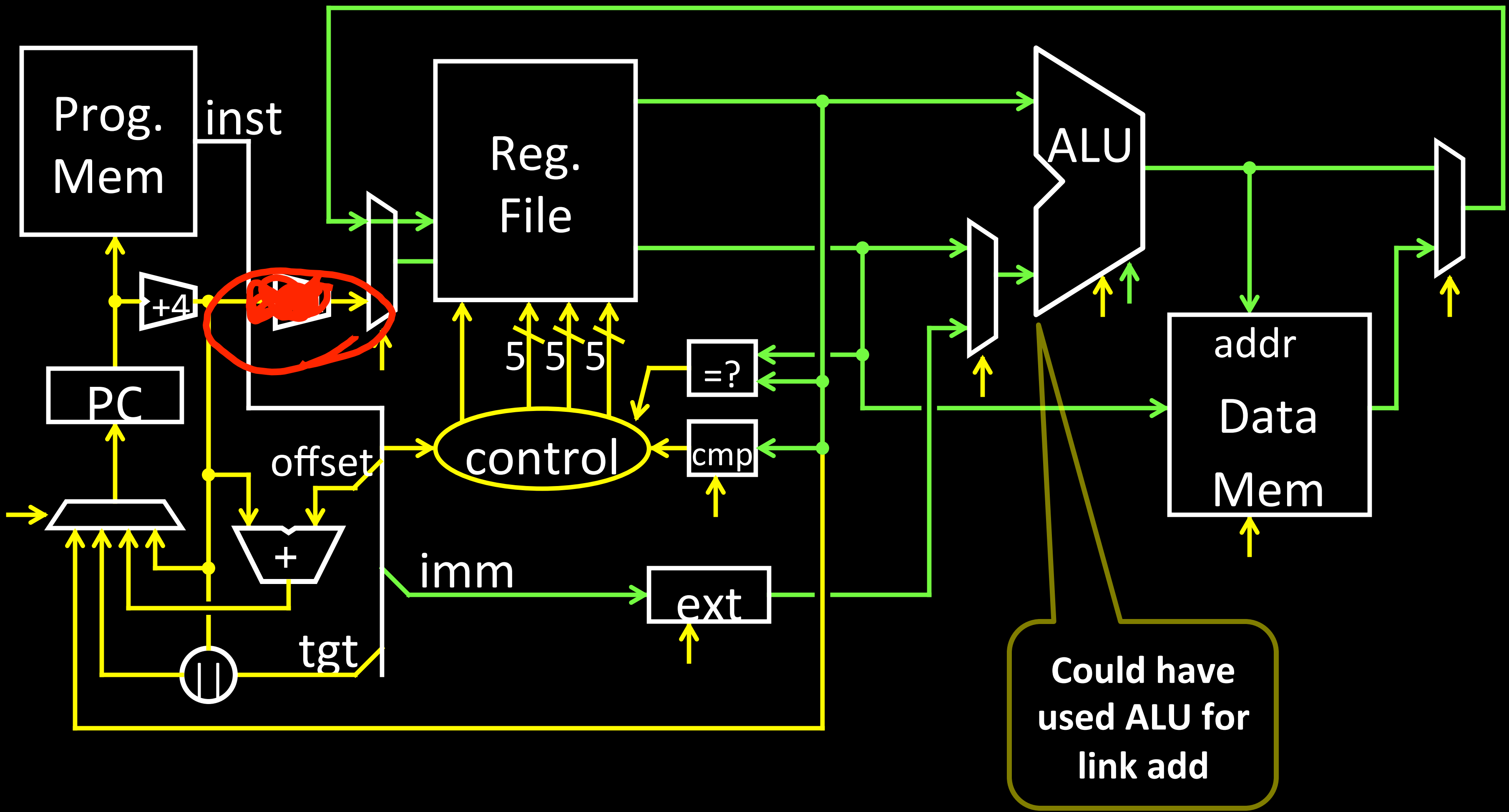
	Din[7:0]									DOut [9:0]										
	7	6	5	4	3	2	1	0	RD (prior)	9	8	7	6	5	4	3	2	1	0	RD (after)
	H	G	F	E	D	C	B	A		j	h	g	f	i	e	d	c	b	a	
D31.1	0	0	1	1	1	1	1	1	+1	1	0	0	1	0	0	1	0	1	0	-1
D31.1	0	0	1	1	1	1	1	1	-1	1	0	0	1	1	1	0	1	0	1	+1



	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

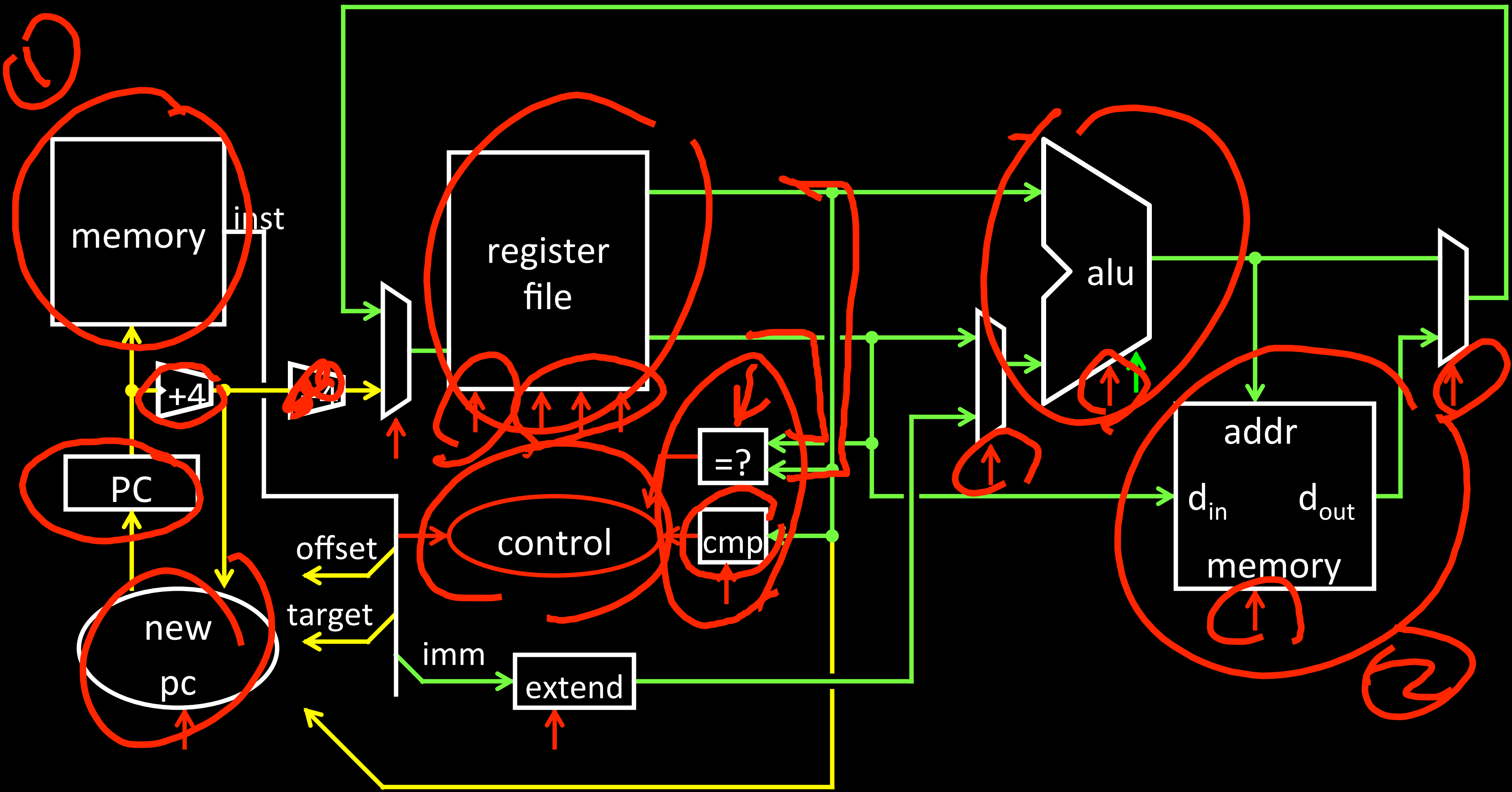
Announcements

- 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 = $(PC+4)_{31..28}$ || (target << 2)

Review: Single cycle processor



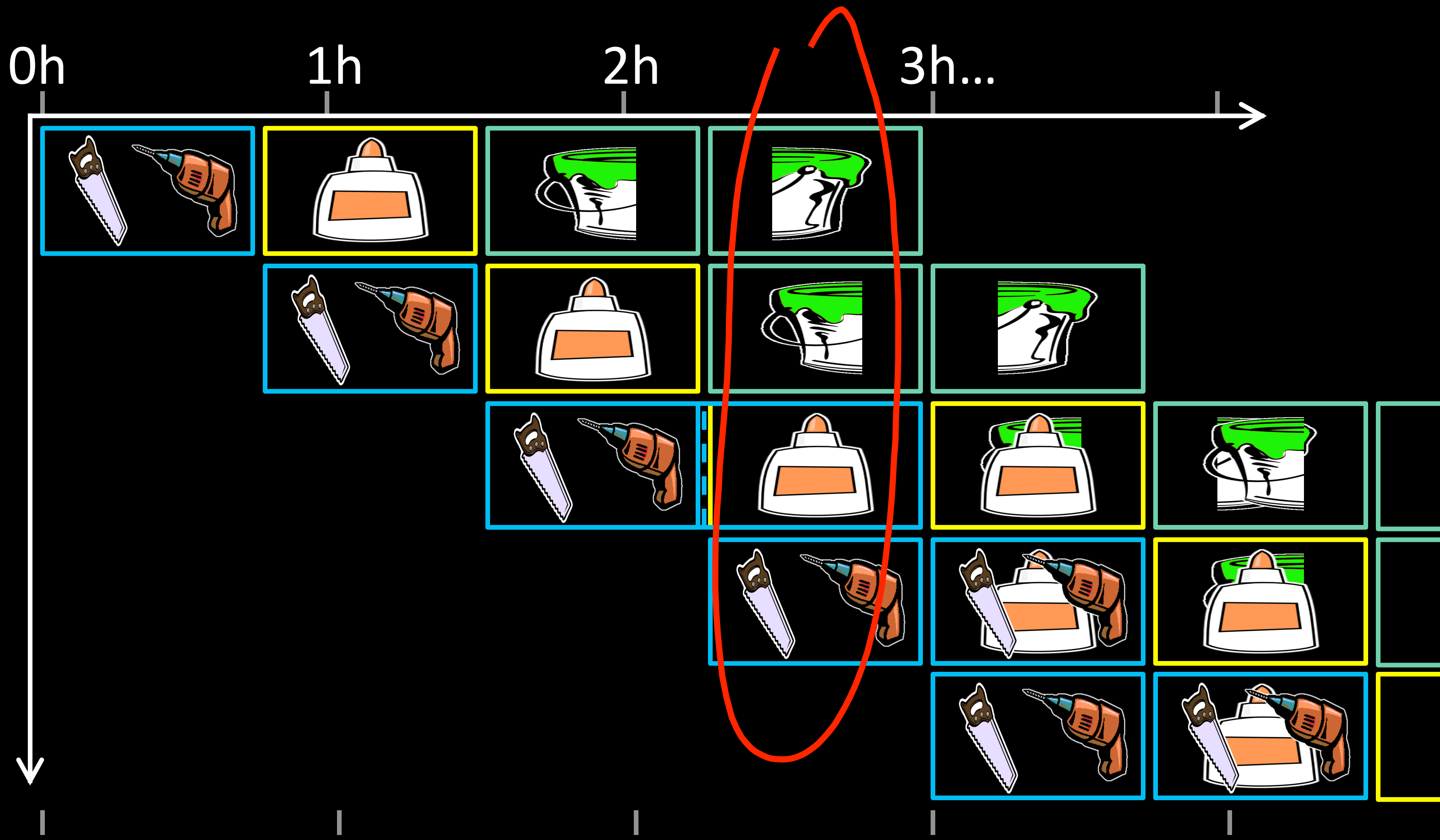
Single Cycle Processor

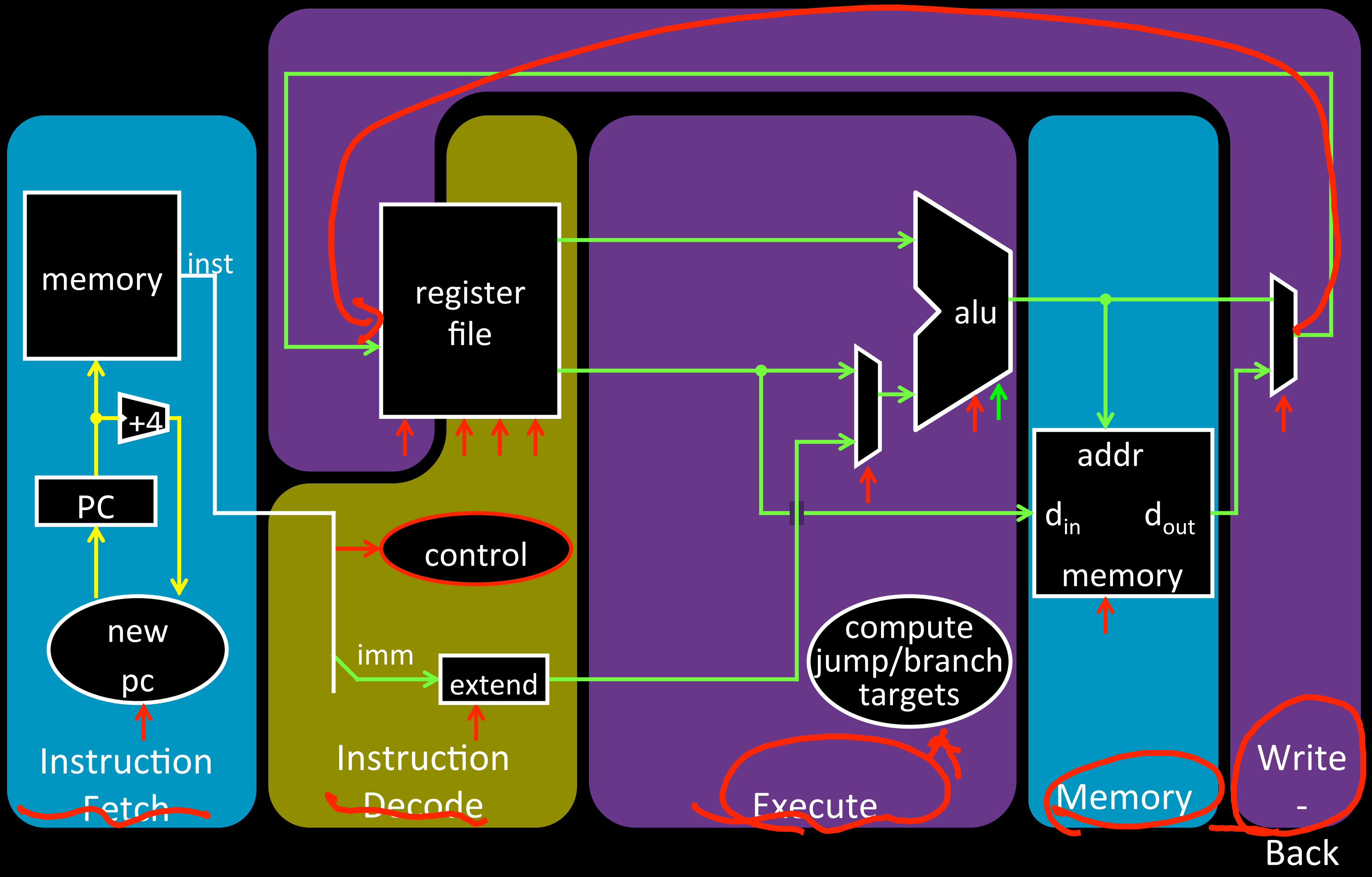
Advantages

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

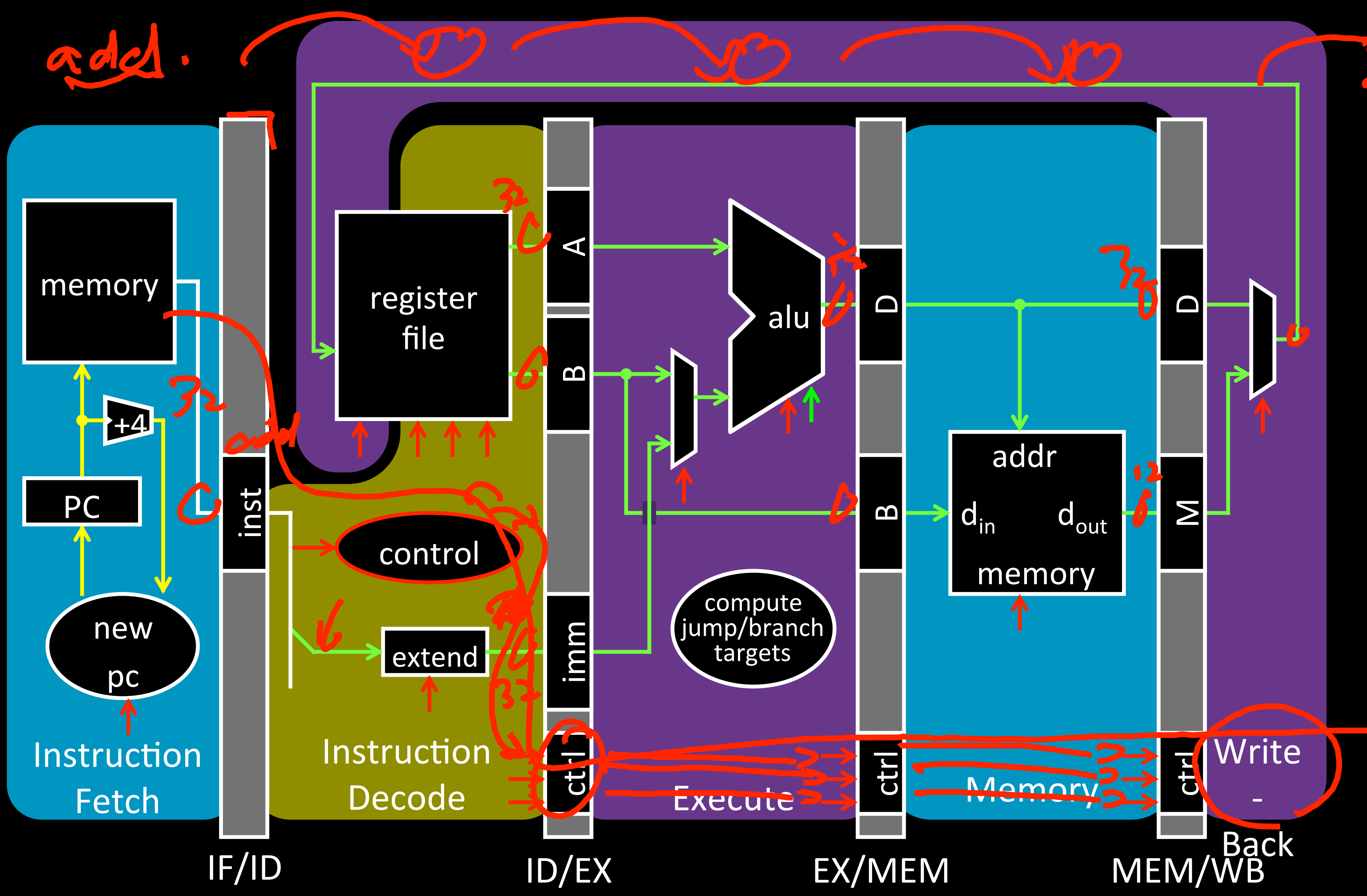
Five stage "RISC" load-store architecture

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

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
+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
32
32
32
32
32
32
add

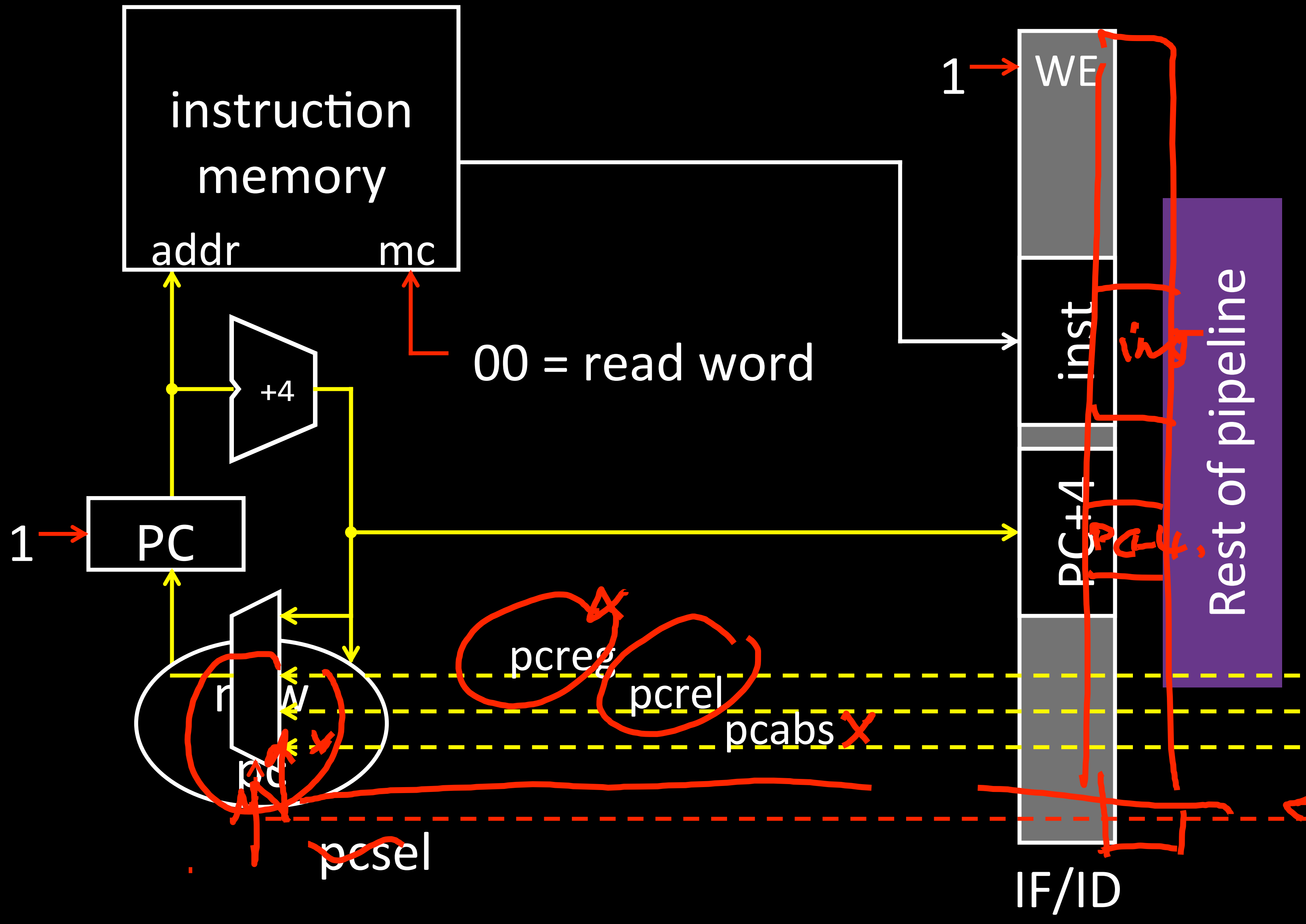
Stage 1: Instruction Fetch

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) $\leftarrow$
- PC+4 (for later computing branch targets) $\leftarrow$


instruction memory
addr
mc
00 = read word
+4
PC
1
1
WE
inst
PC+4
Rest of pipeline
pcreg
pcrel
pcabs
pcsel
IF/ID
inst
PC+4,

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

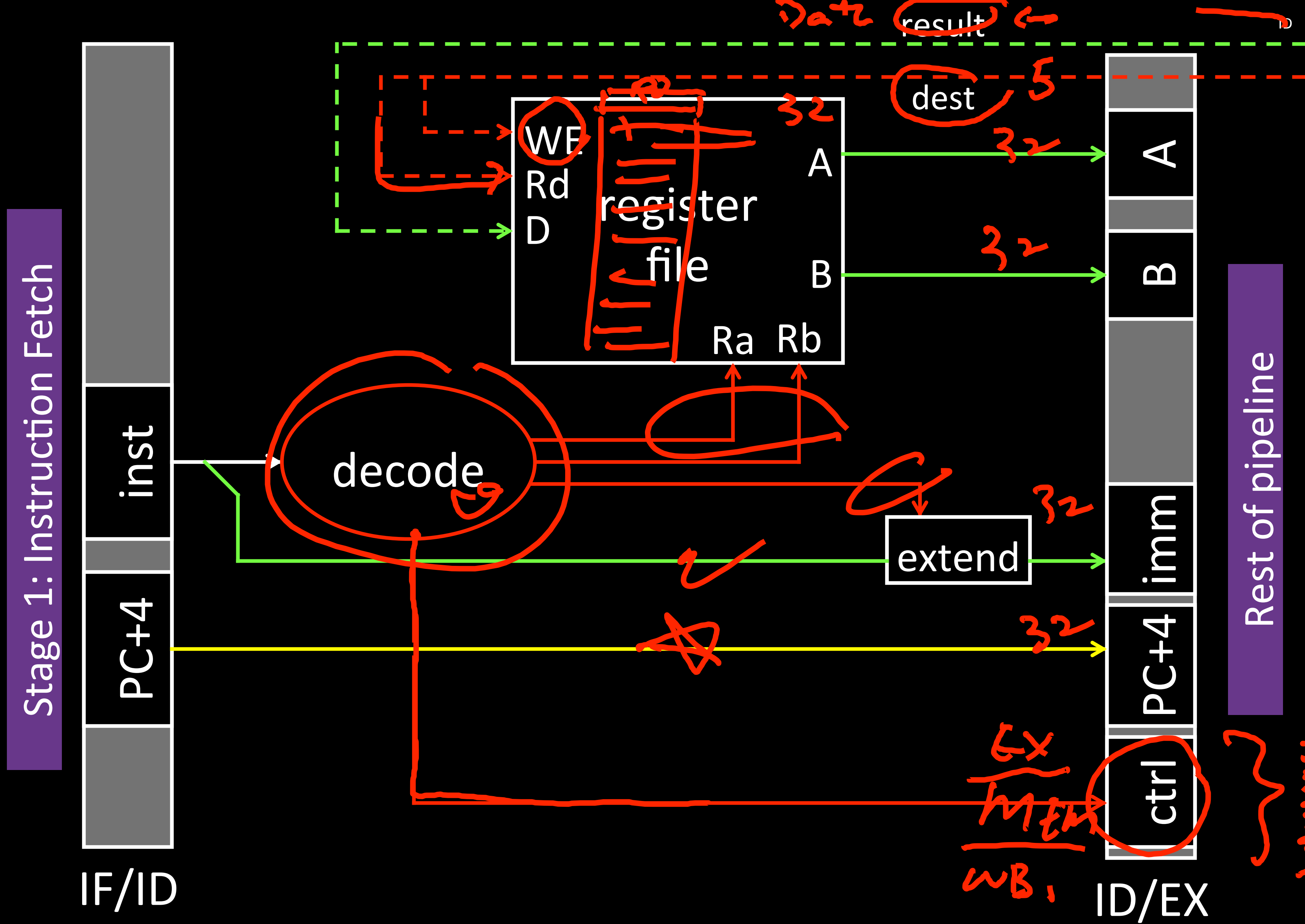
r2 100

addu R3, R2, R1

Ra Rb

PC+4

IF/ID



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

Stage 3: Execute

Jump. R1 , 0x 10000 100

On every cycle:

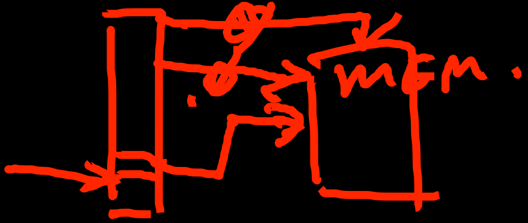
- Read ID/EX pipeline register to get values and control bits
- Perform ALU operation
- Compute targets (PC+4+offset, etc.) *in case* this is a branch
- Decide if jump/branch should be taken

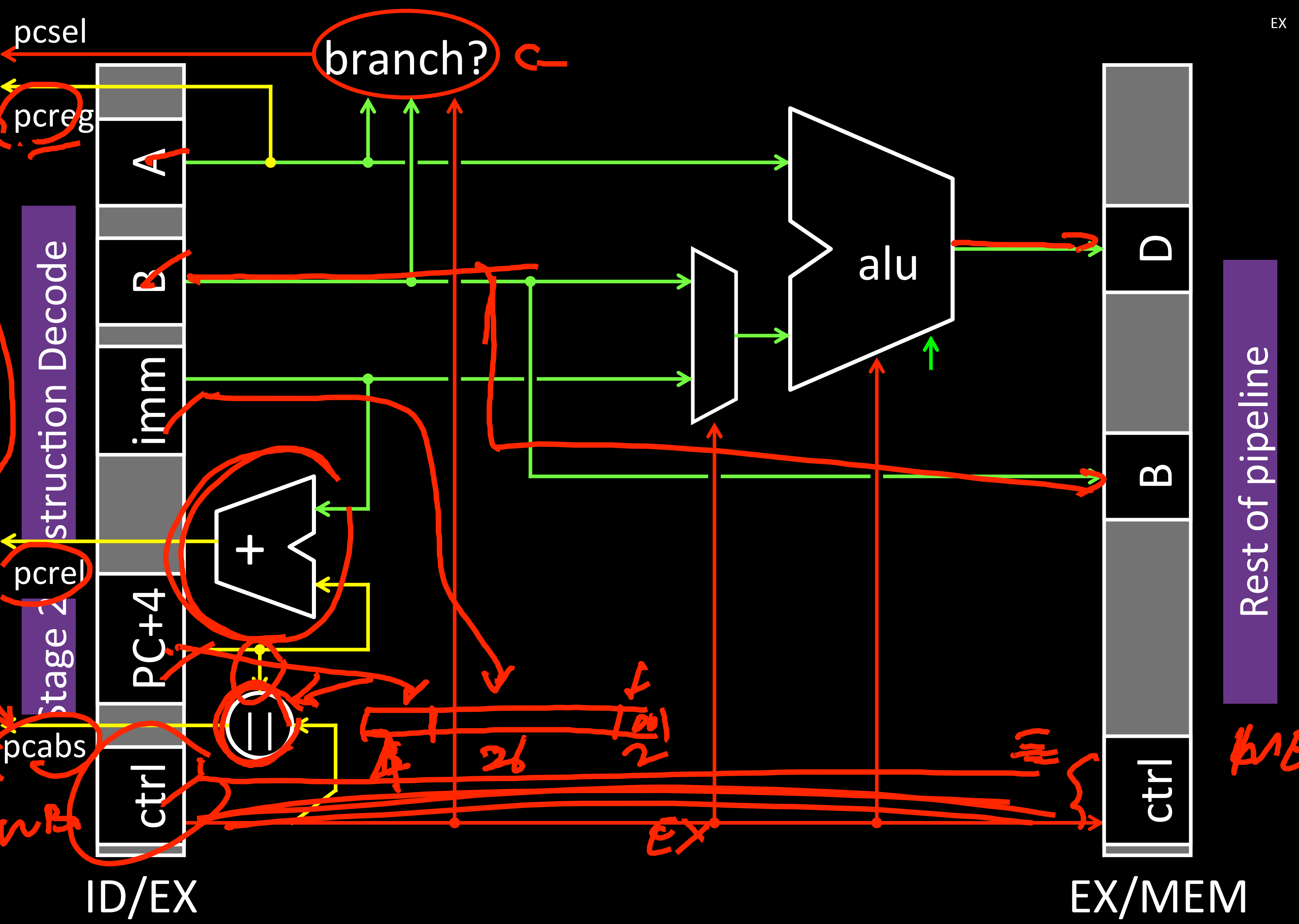
Branch Prediction:

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
alu
D
B
+
pcrel
PC+4
pcabs
ctrl
==
ctrl
Instruction Decode
Stage 2
Rest of pipeline
ID/EX
EX/MEM
26
WB
EX

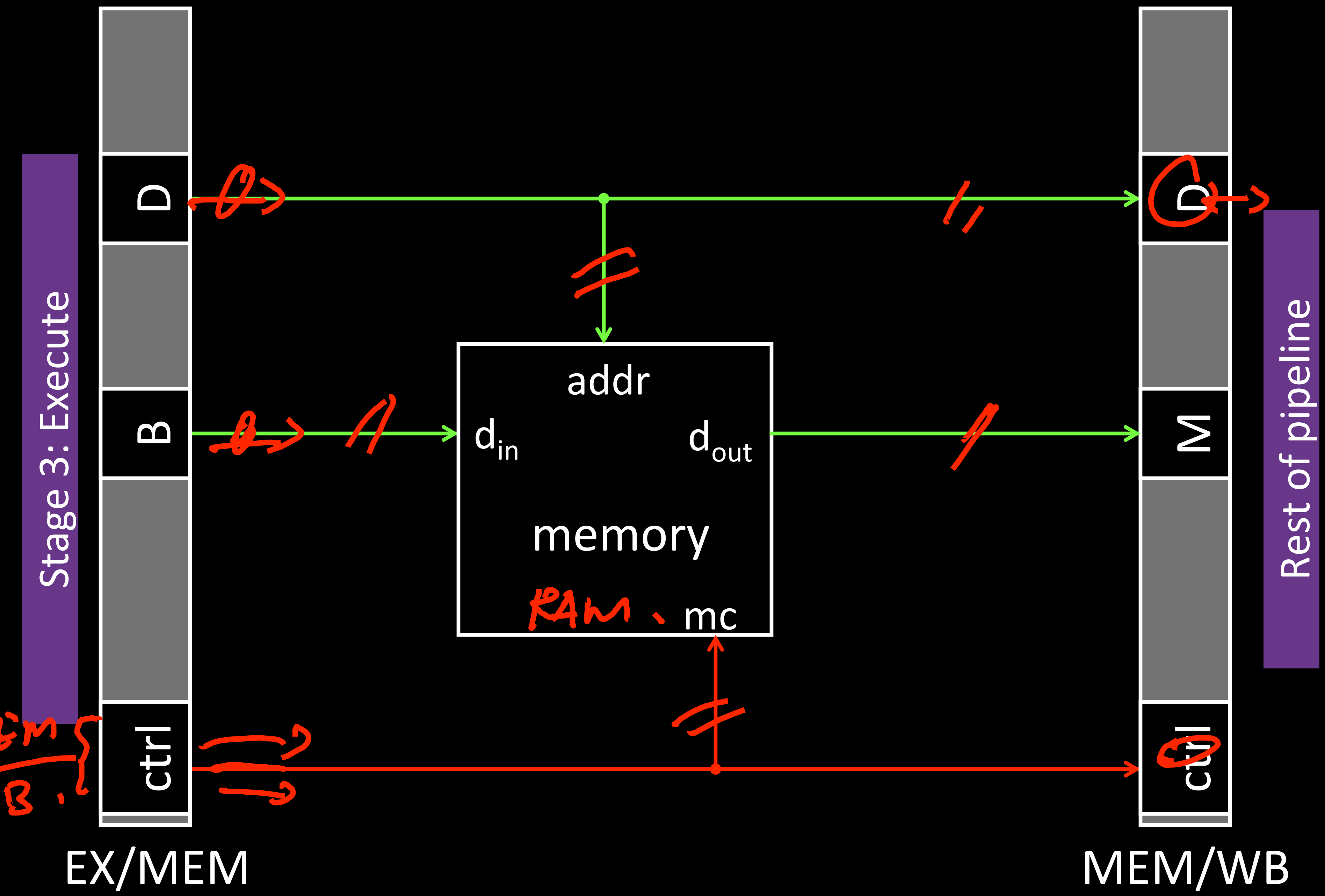
Stage 4: Memory

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)

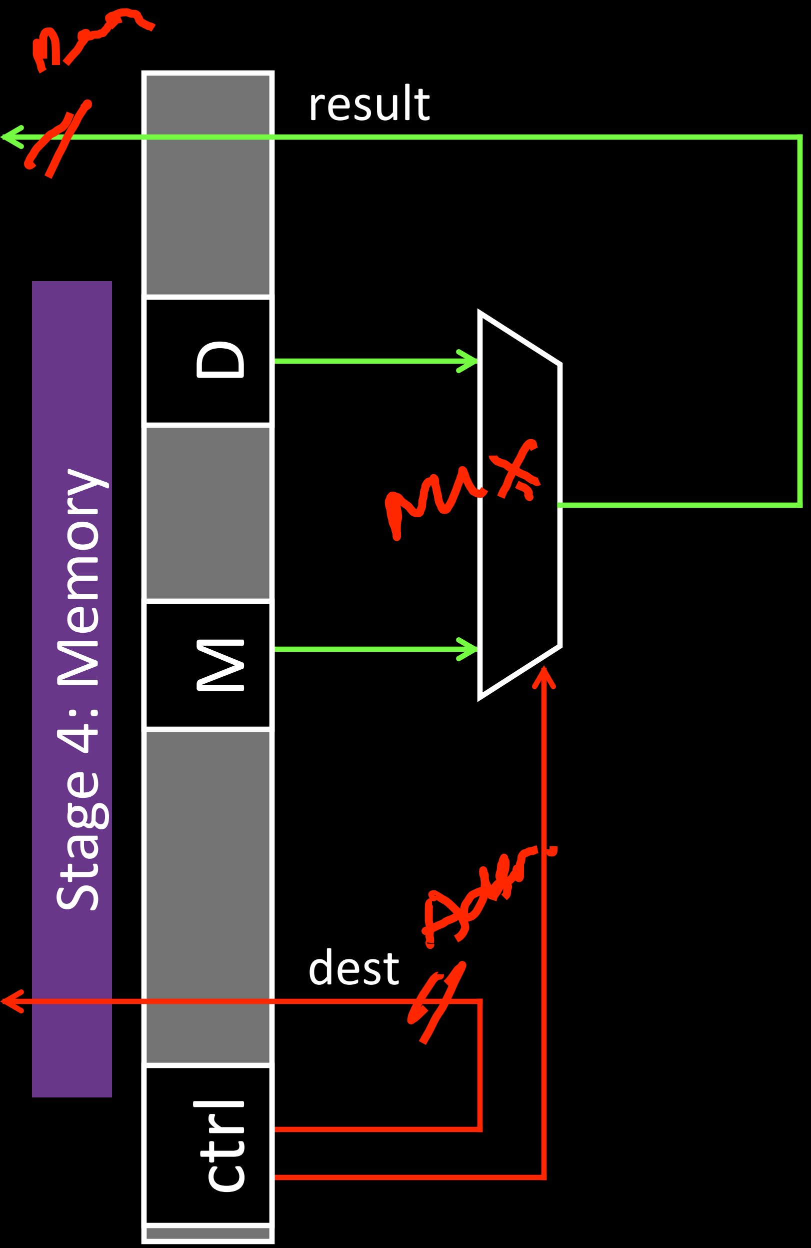
- Control information, Rd index, …
- Result of memory operation ←
- Pass result of ALU operation ←


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

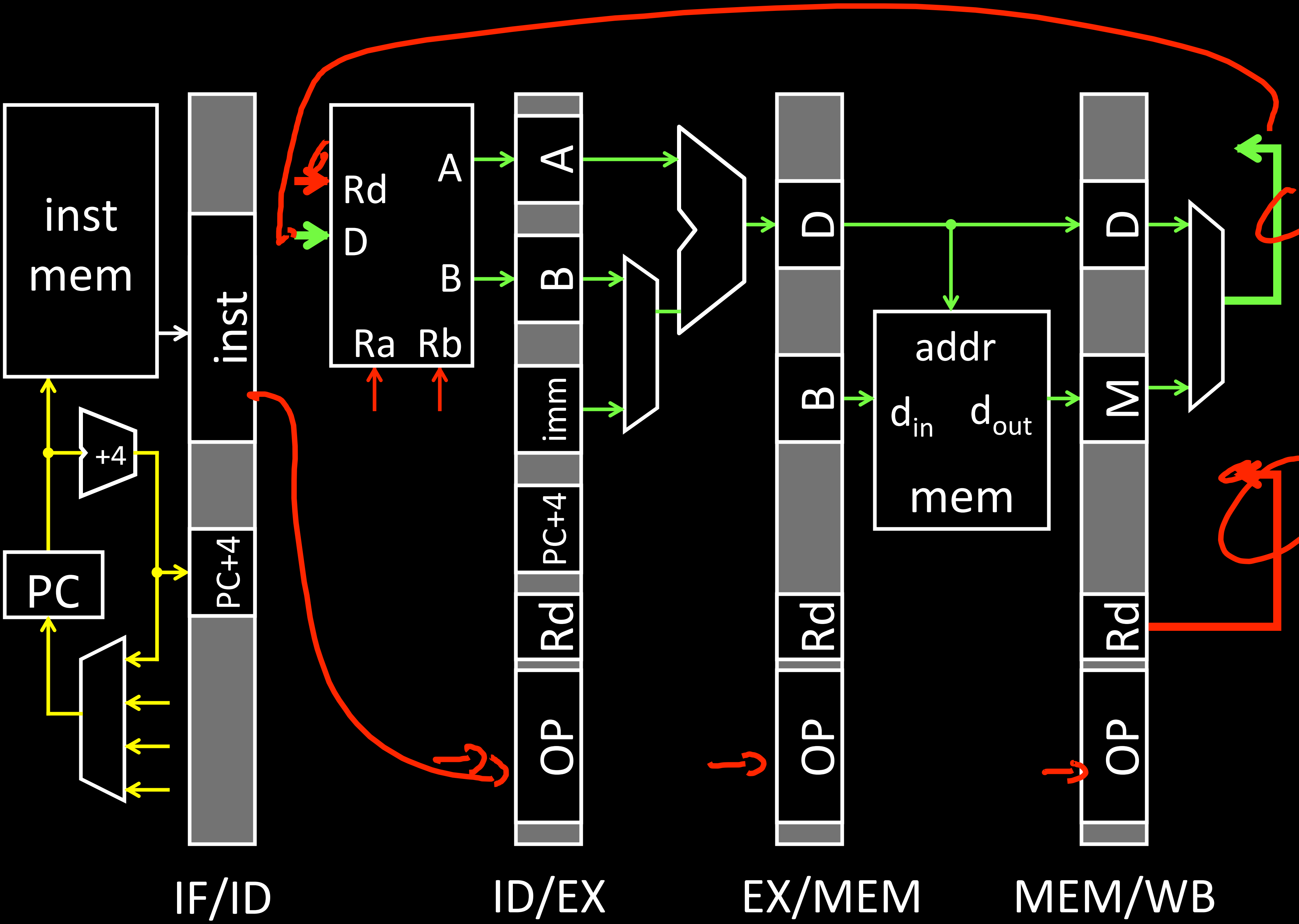
Stage 5: Write-back

On every cycle:

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



MEM/WB


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

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

r3 = r1 + r2 = 45

r6 = ~(r4 & r5) = 111

18 & 7

...0 010010

...00 00111

~~...0000~~

~~000~~

~~00010~~

~ 00010

...111101

= -3

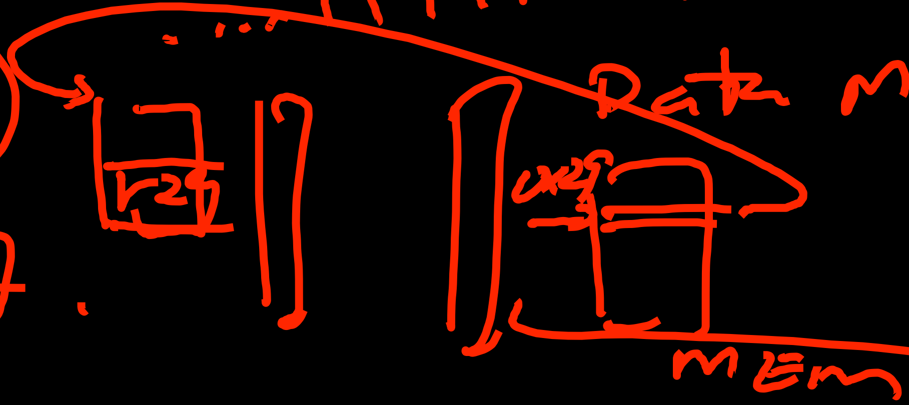
r4 = MEM[r2 + 20]

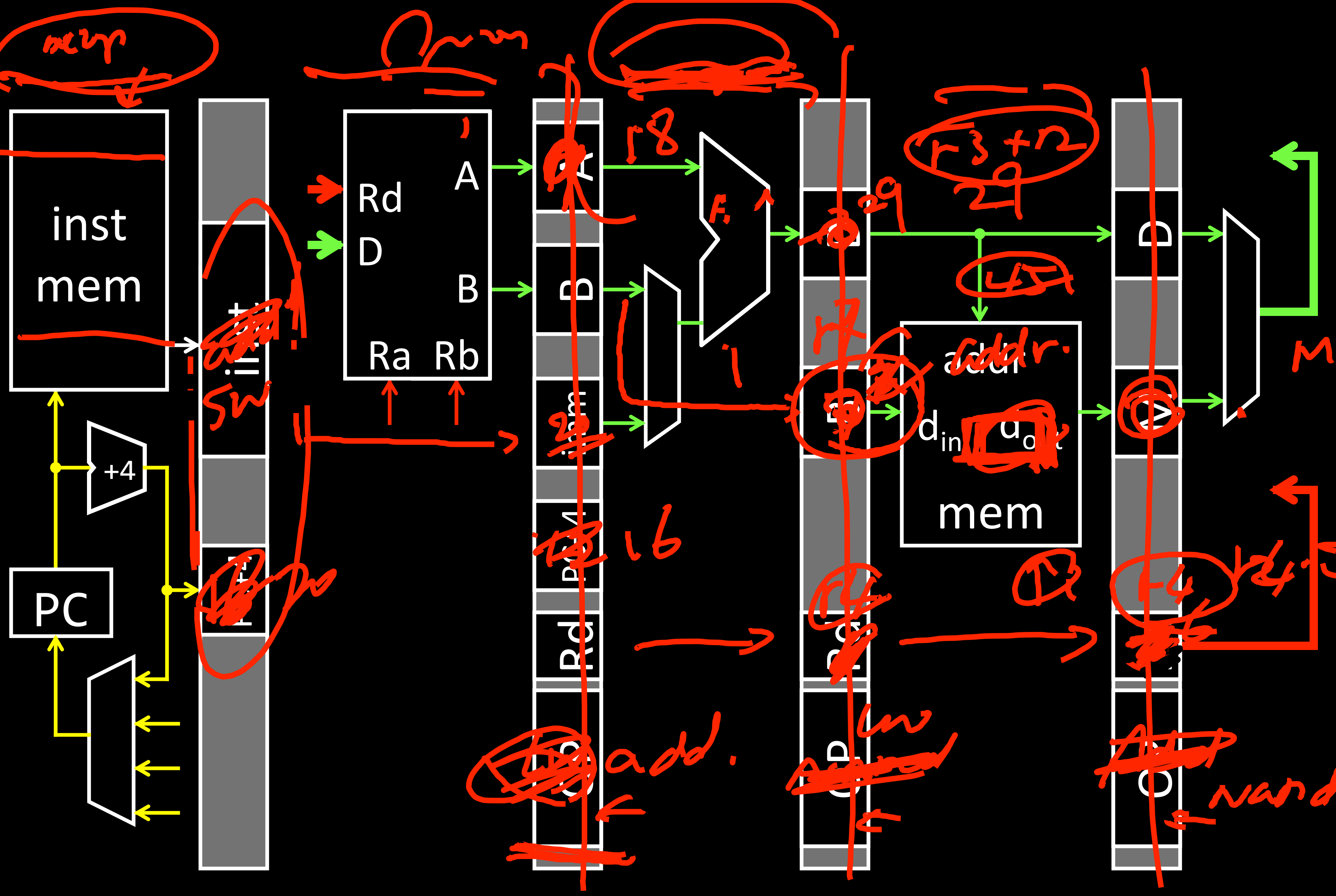
r5 = r2 + r5 = 16

9 + 7

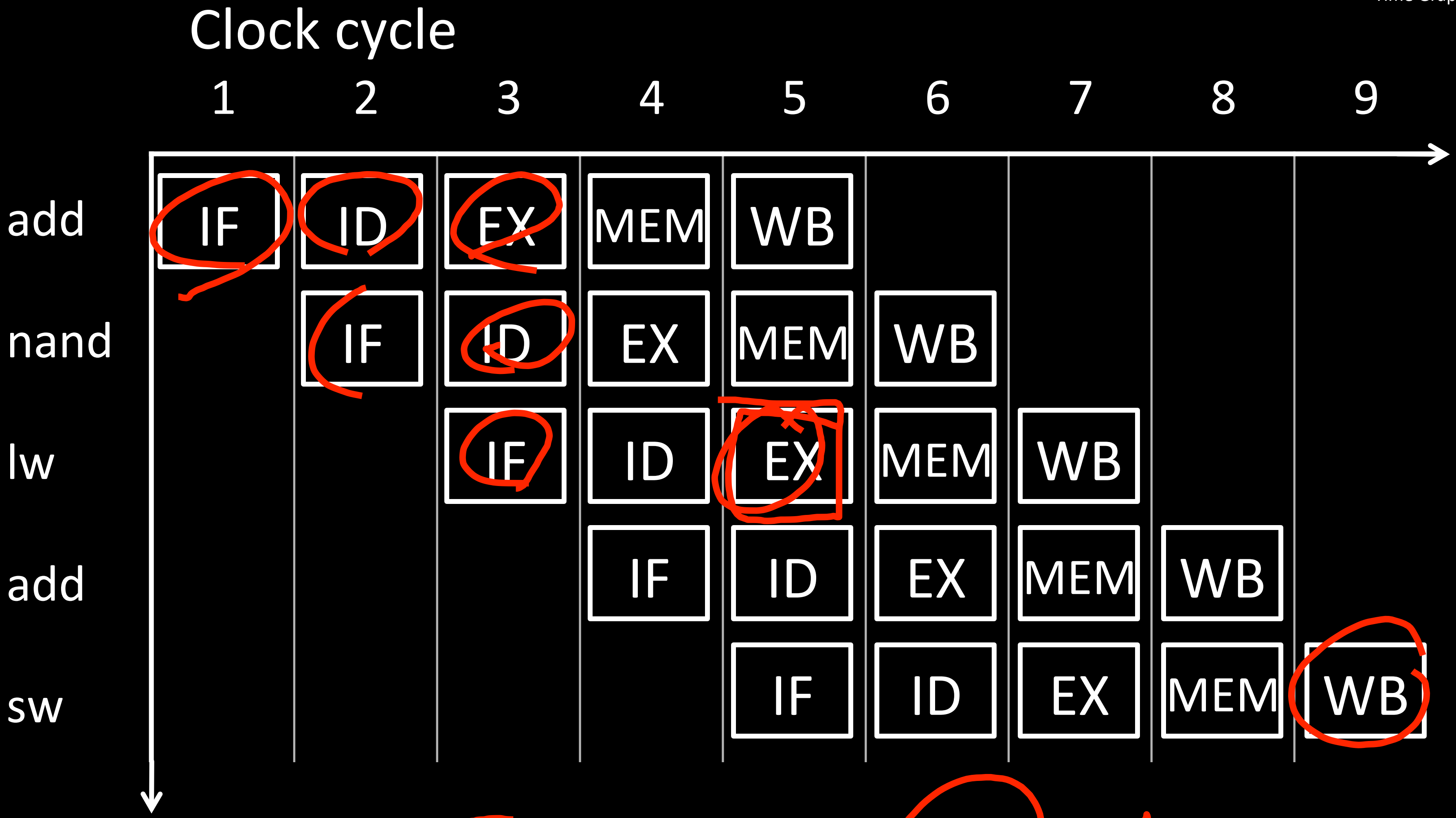
MEM[r3 + 12] = r7

77.





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



Latency: 5

Throughput: 1

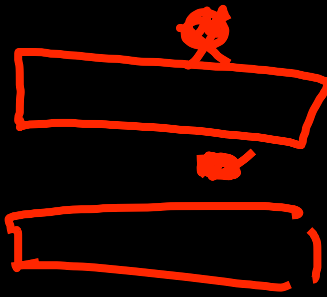
Concurrency: 5

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

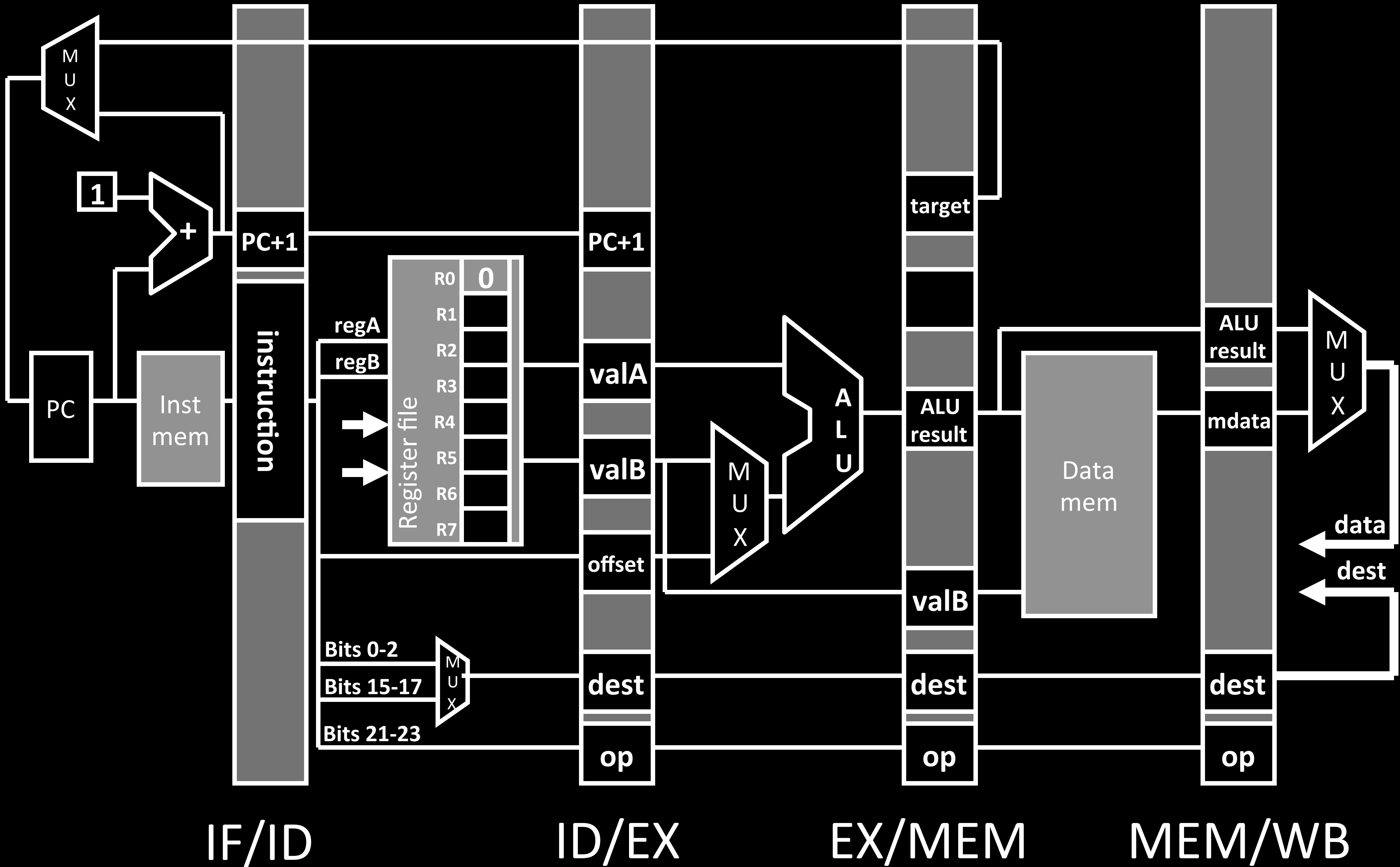
- Interface (ISA) vs. implementation (Pipeline)

The end

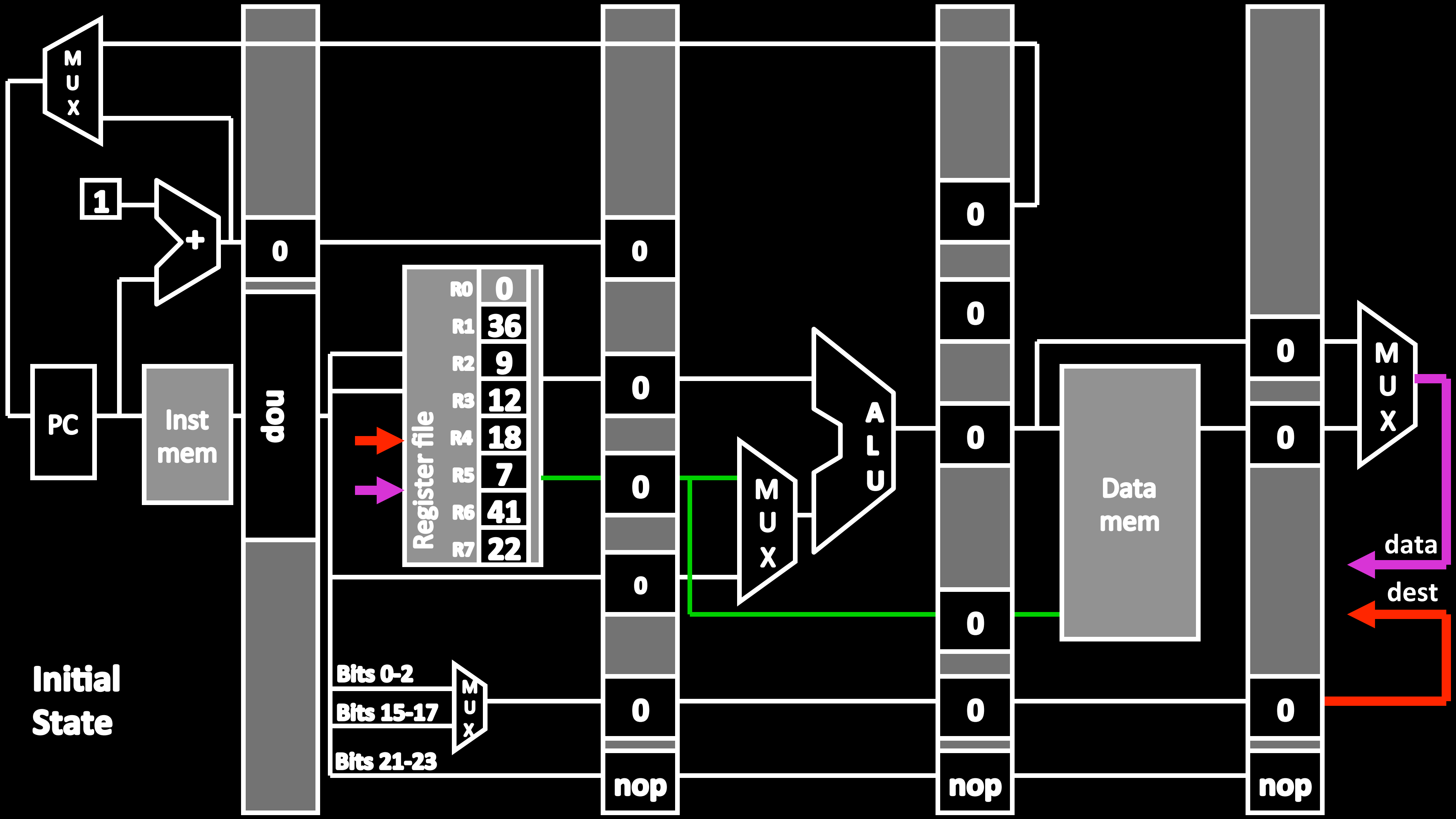
Assume eight-register machine

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
```



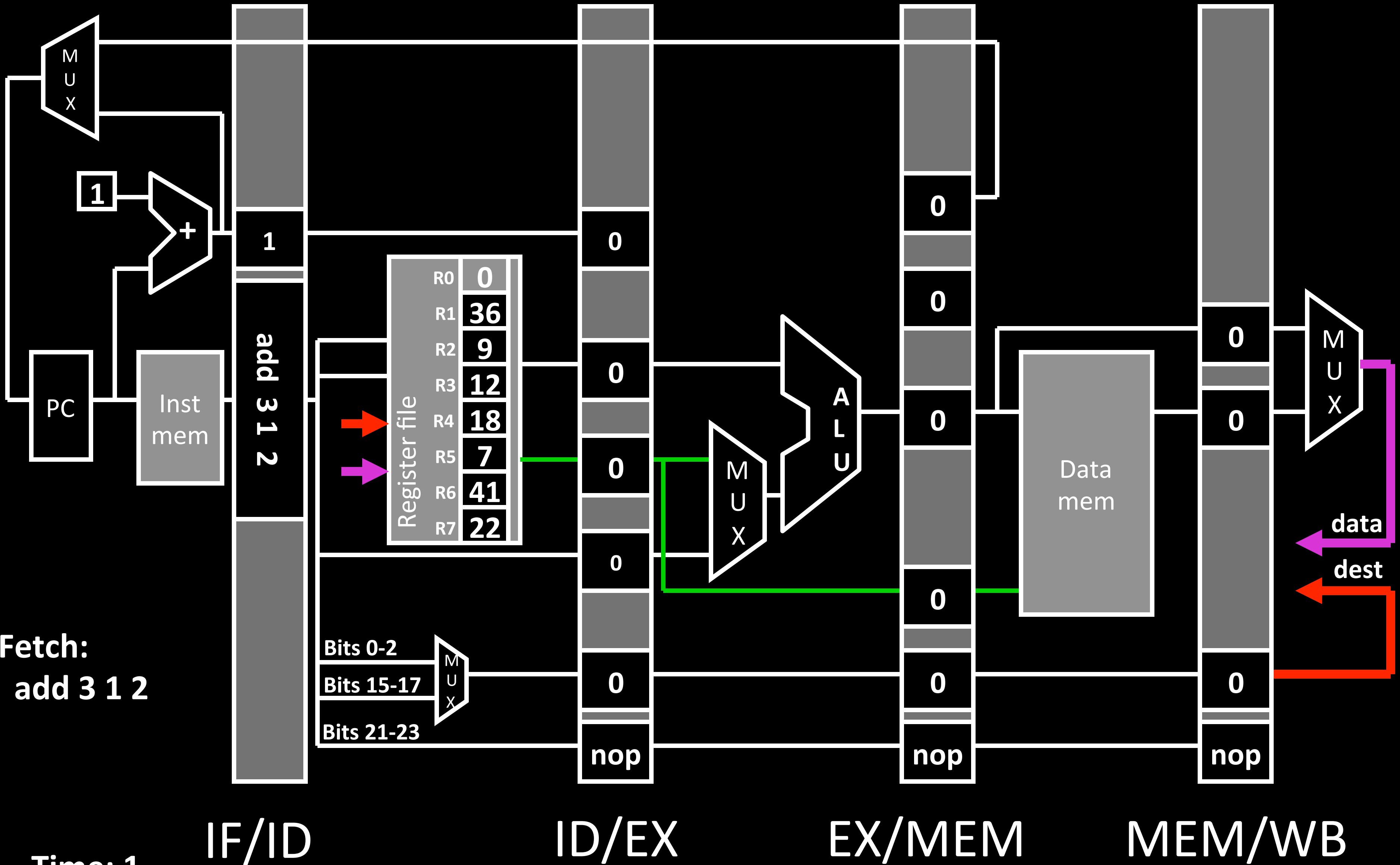
M U X
1
+
PC
Inst mem
PC+1
instruction
regA
regB
Register file
R0
0
R1
R2
R3
R4
R5
R6
R7
Bits 0-2
Bits 15-17
Bits 21-23
M U X
PC+1
valA
valB
offset
dest
op
M U X
A L U
target
ALU result
valB
dest
op
Data mem
ALU result
mdata
dest
op
M U X
data
dest
IF/ID
ID/EX
EX/MEM
MEM/WB



Initial State

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

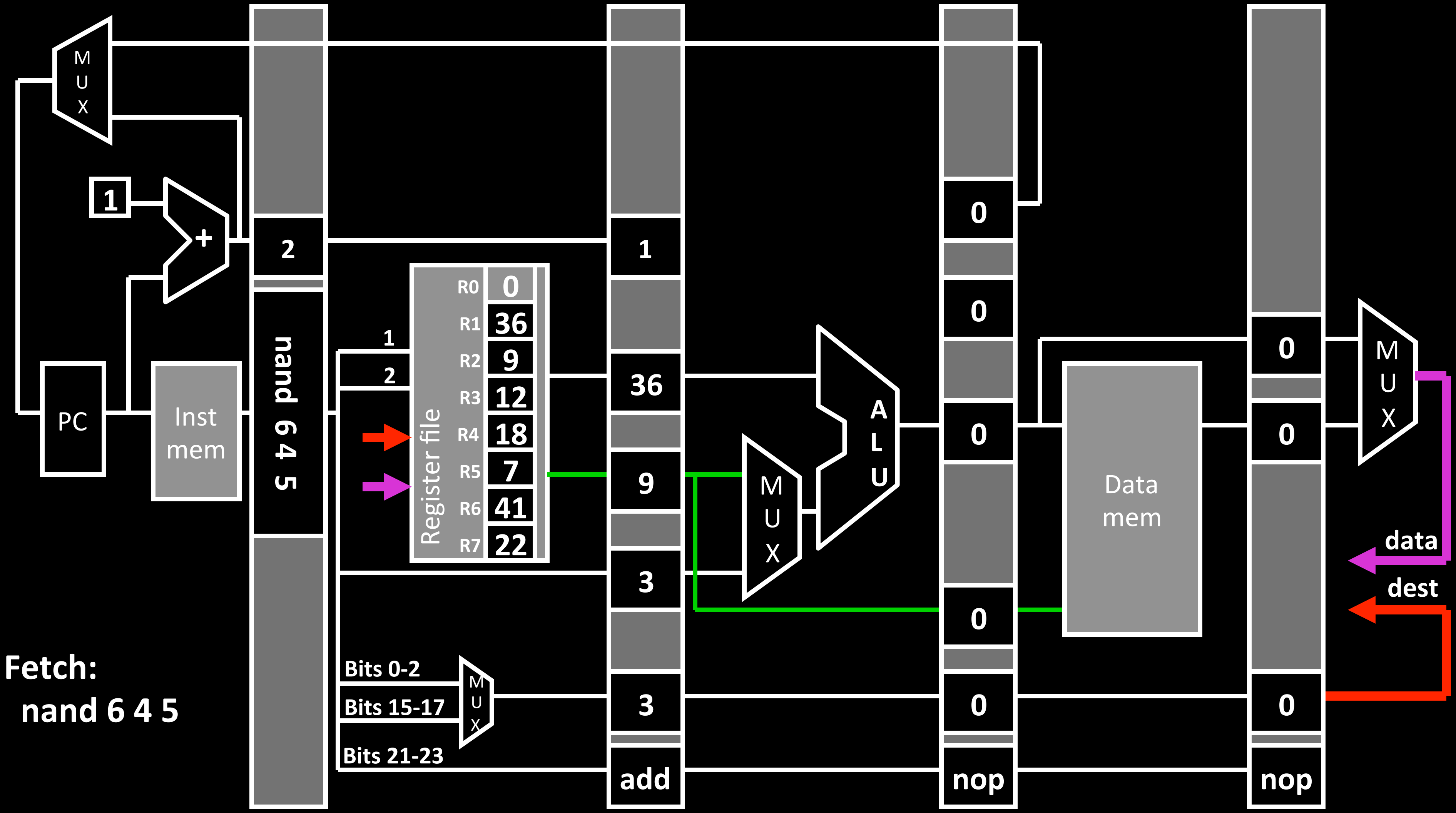
add 3 1 2



Fetch:
add 3 1 2

Time: 1

nand 6 4 5 add 3 1 2

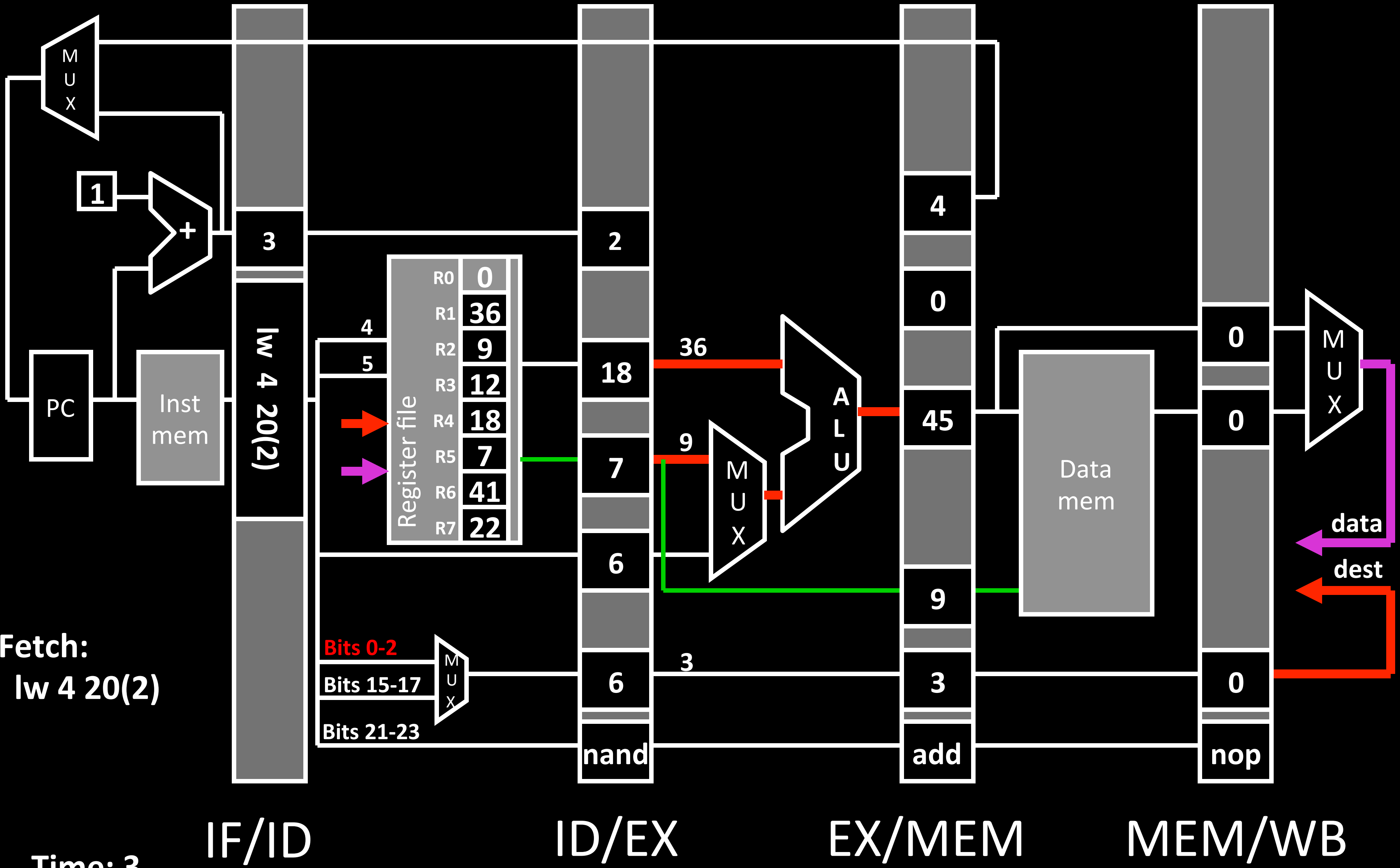


Fetch:
nand 6 4 5

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

Time: 2

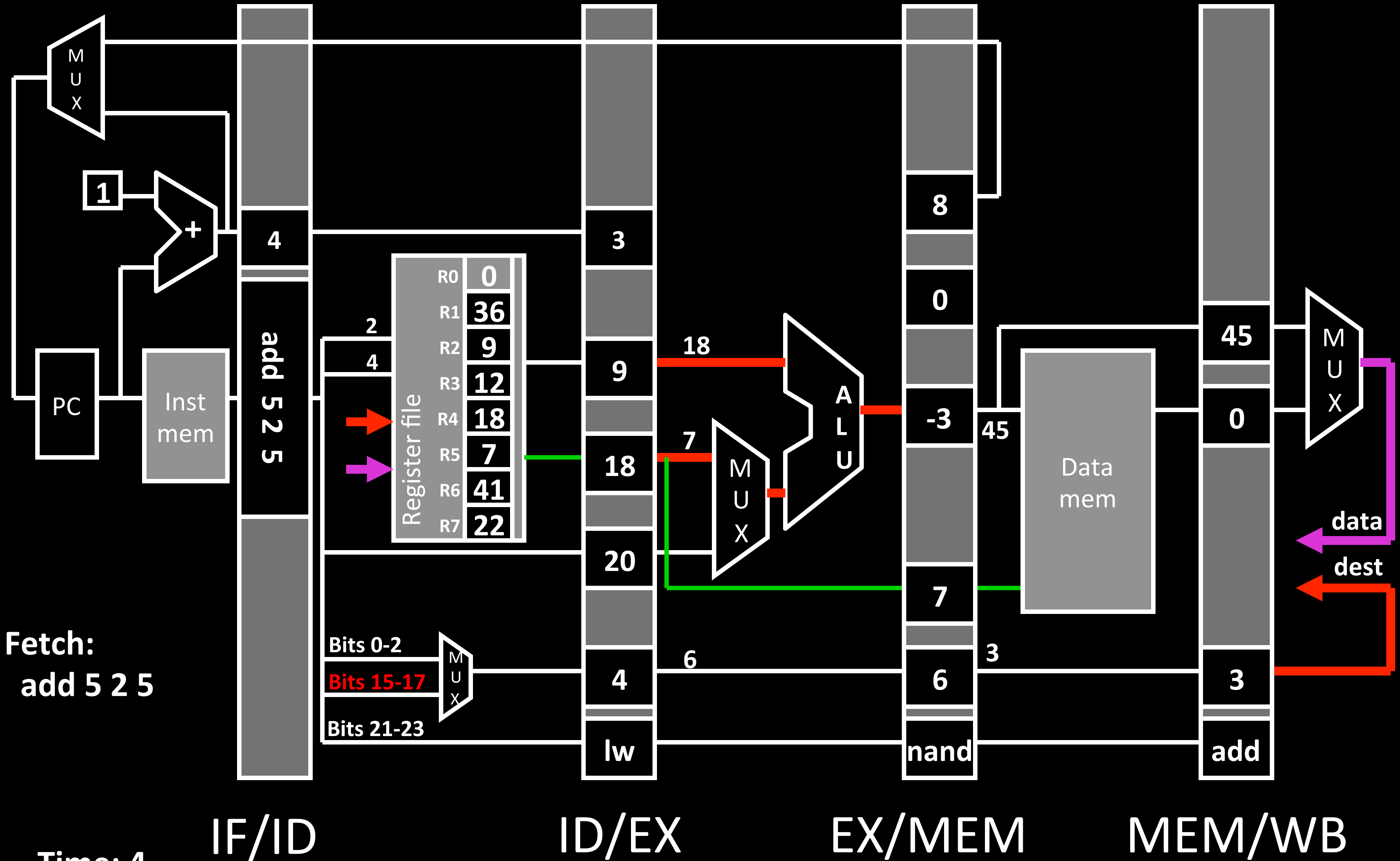
lw 4 20(2) nand 6 4 5 add 3 1 2



Fetch:
lw 4 20(2)

Time: 3

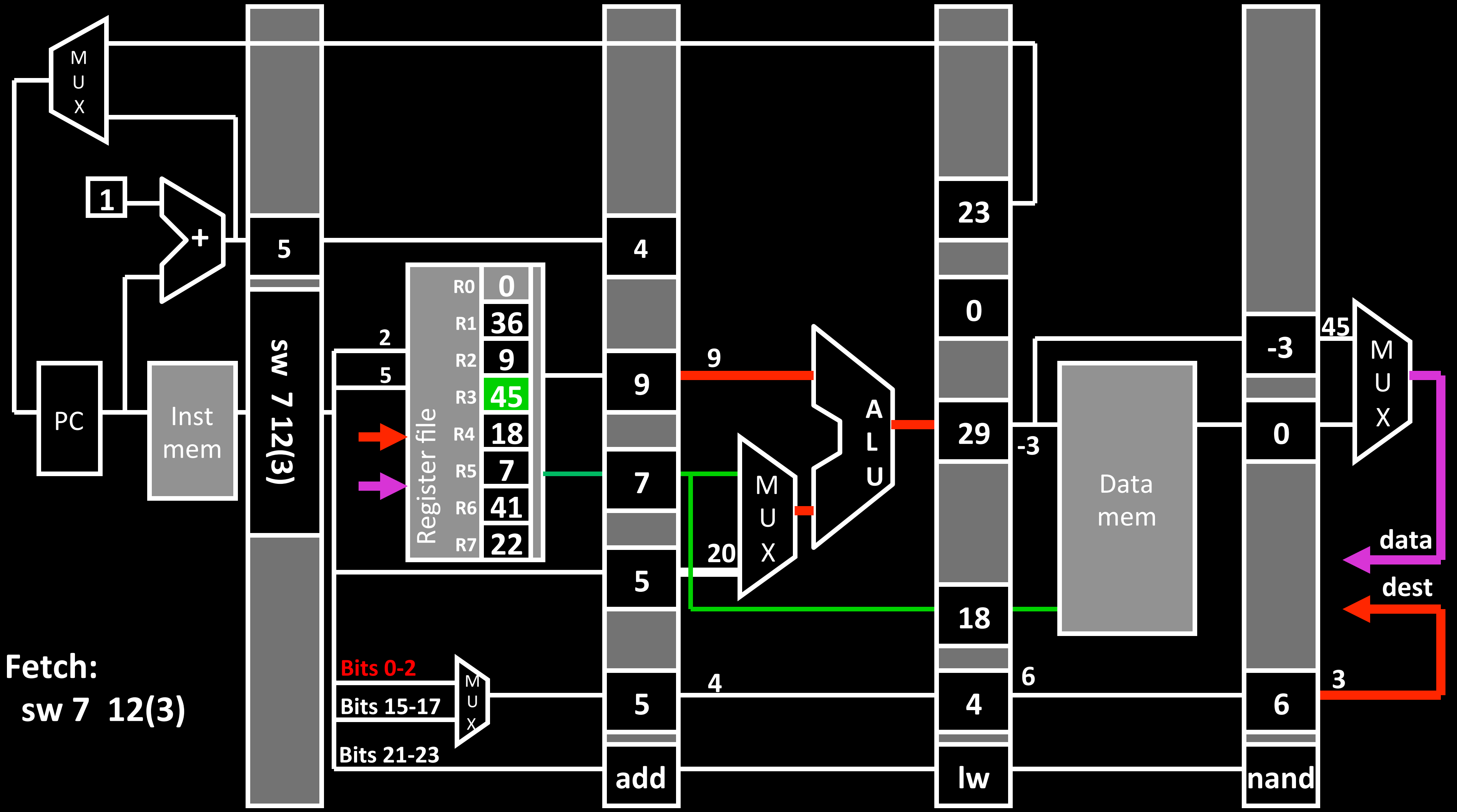
add 5 2 5 lw 4 20(2) nand 6 4 5 add 3 1 2



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

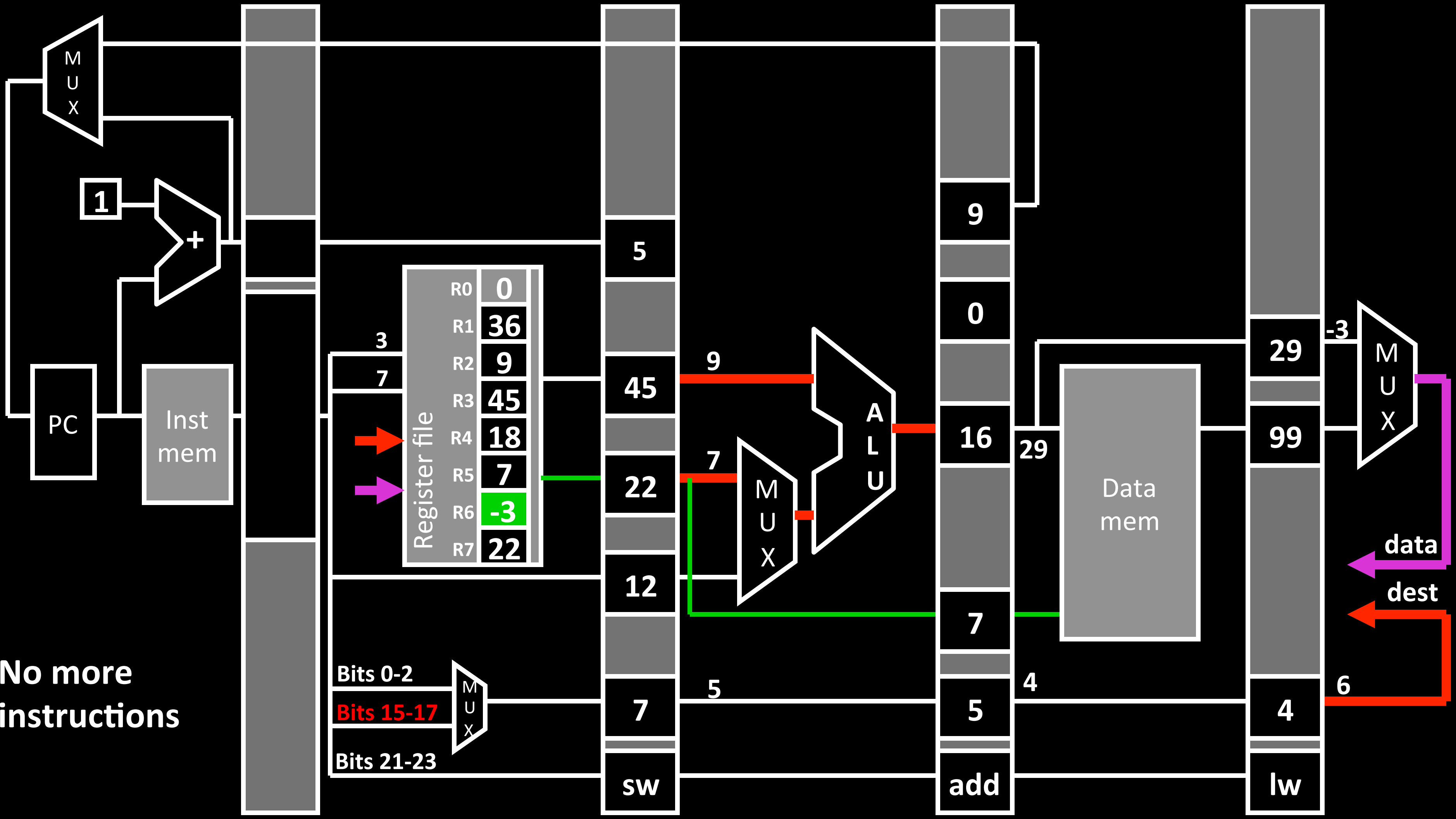


Fetch:
sw 7 12(3)

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

Time: 5

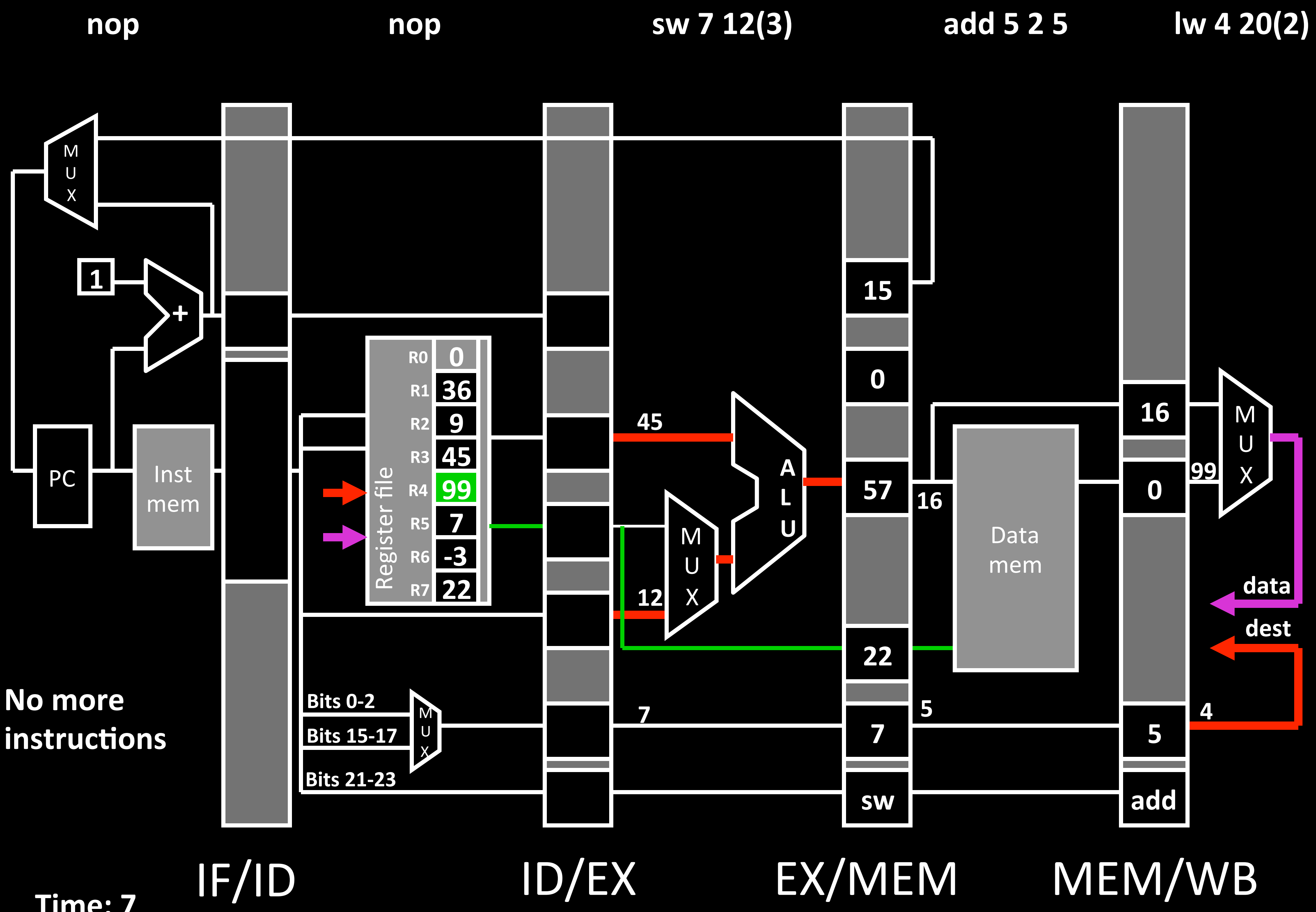
sw 7 12(3) add 5 2 5 lw 4 20(2) nand 6 4 5



No more instructions

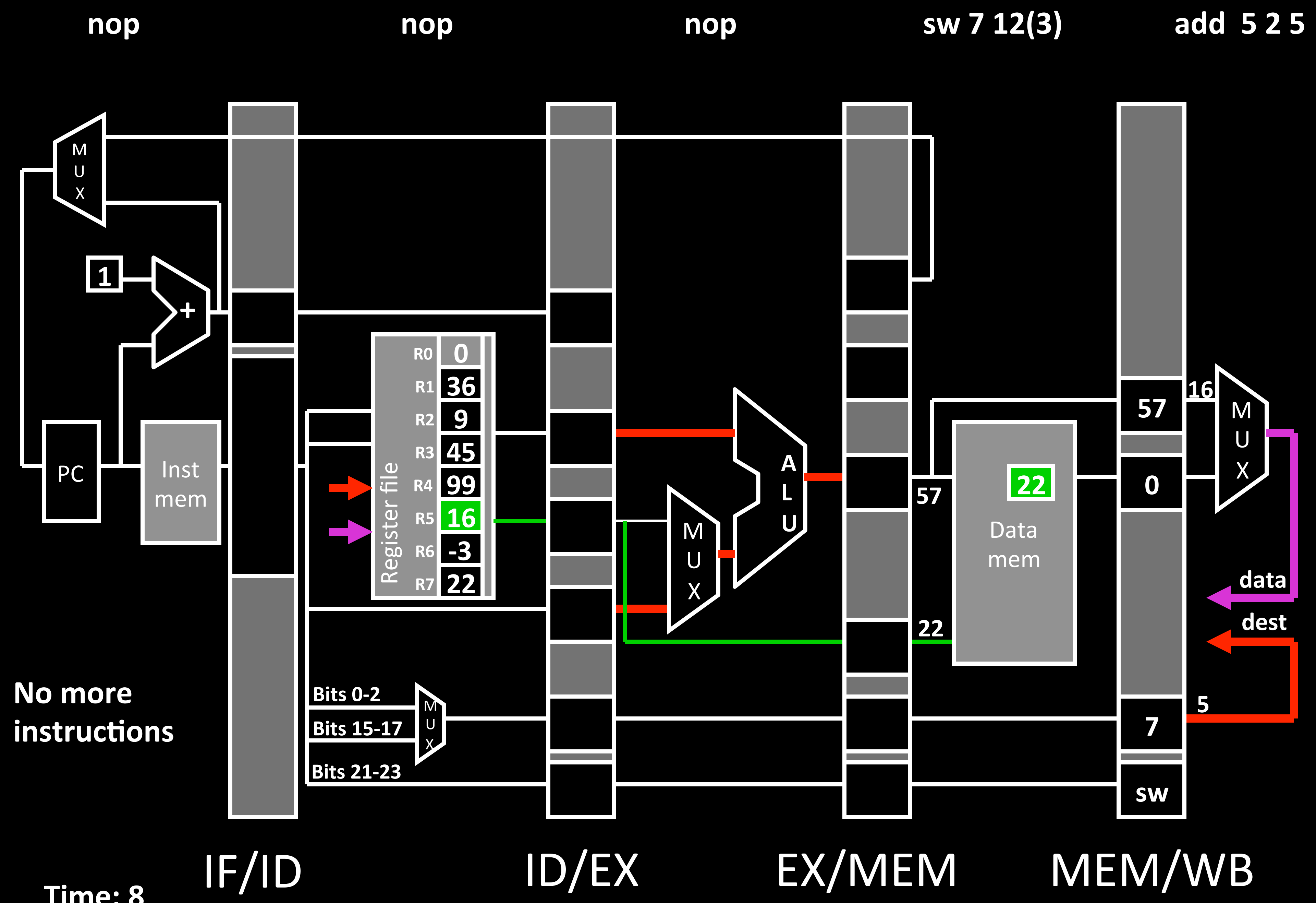
IF/ID ID/EX EX/MEM MEM/WB

Time: 6



No more instructions

Time: 7

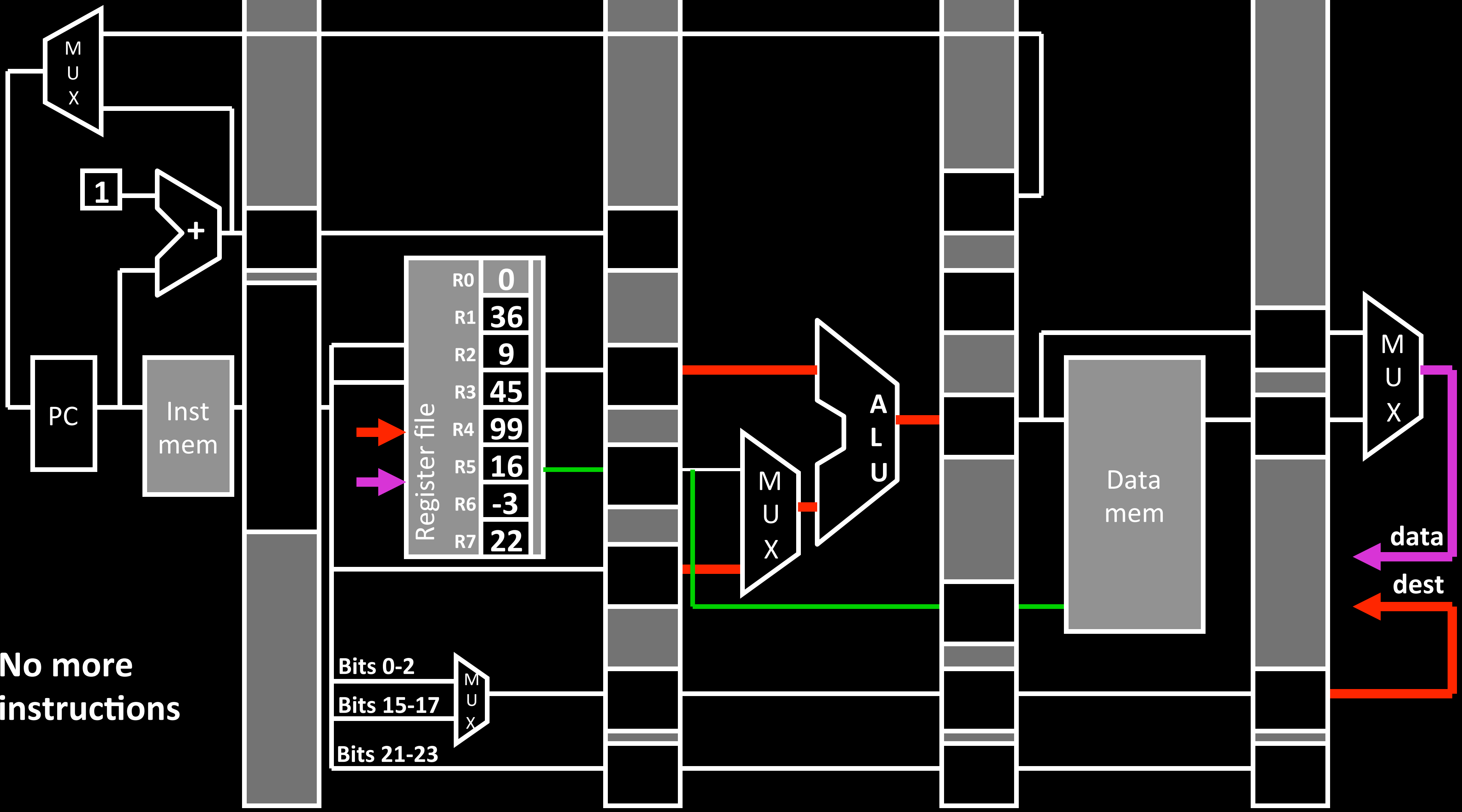


No more instructions

Time: 8

Slides thanks to Sally McKee

nop nop nop nop sw 7 12(3)



No more instructions

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

Time: 9