EECS 370 - Lecture 13

Pipelining and Data Hazards

Announcements

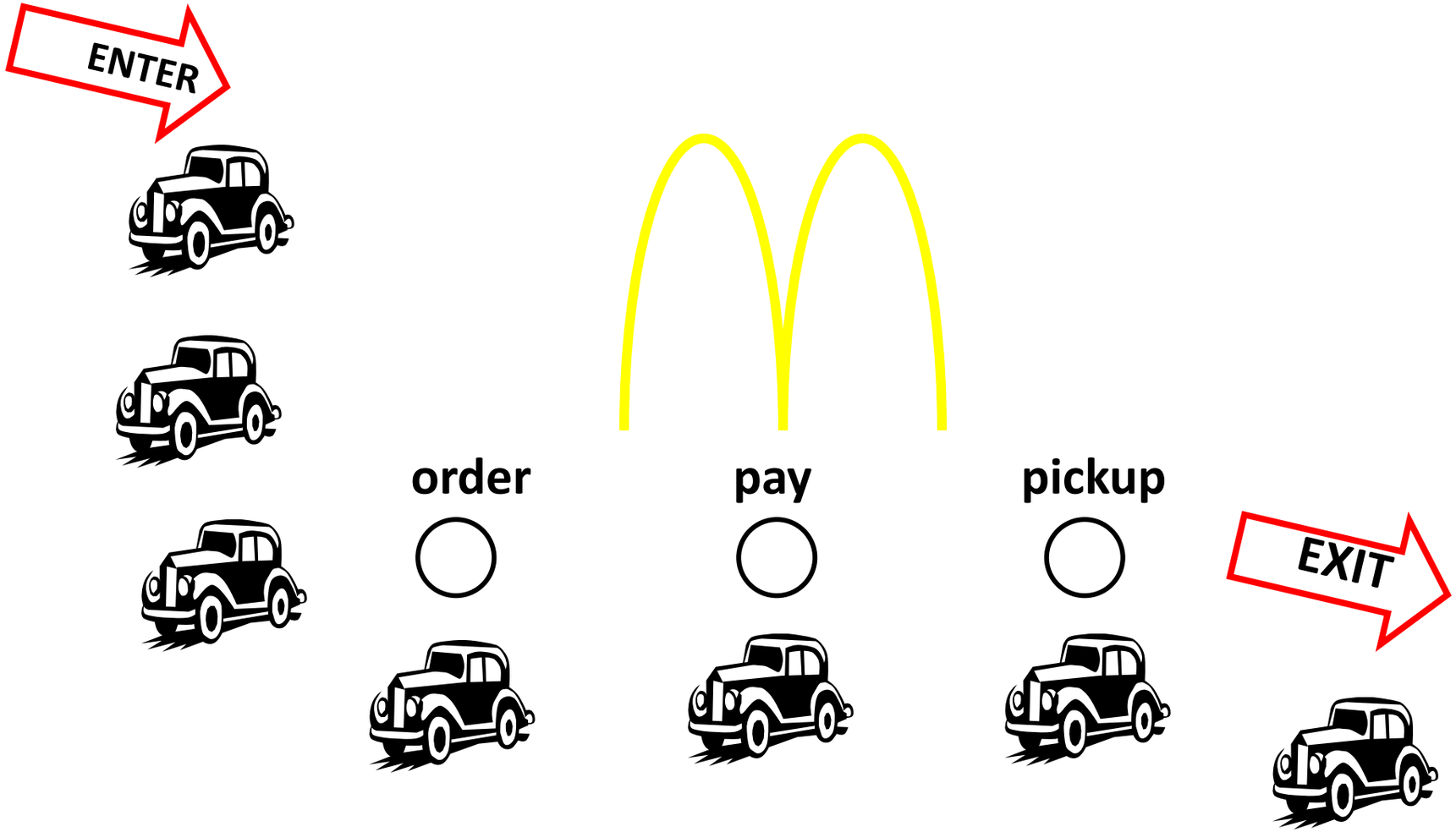
- **Project 2a due today at 11:55pm**

- **Homework 2 due Monday 2/23 @11:55pm**
  - Come to office hours with questions
  - Also there are special office hours in 1690 BBB on Monday from 6-9pm for last minute questions.

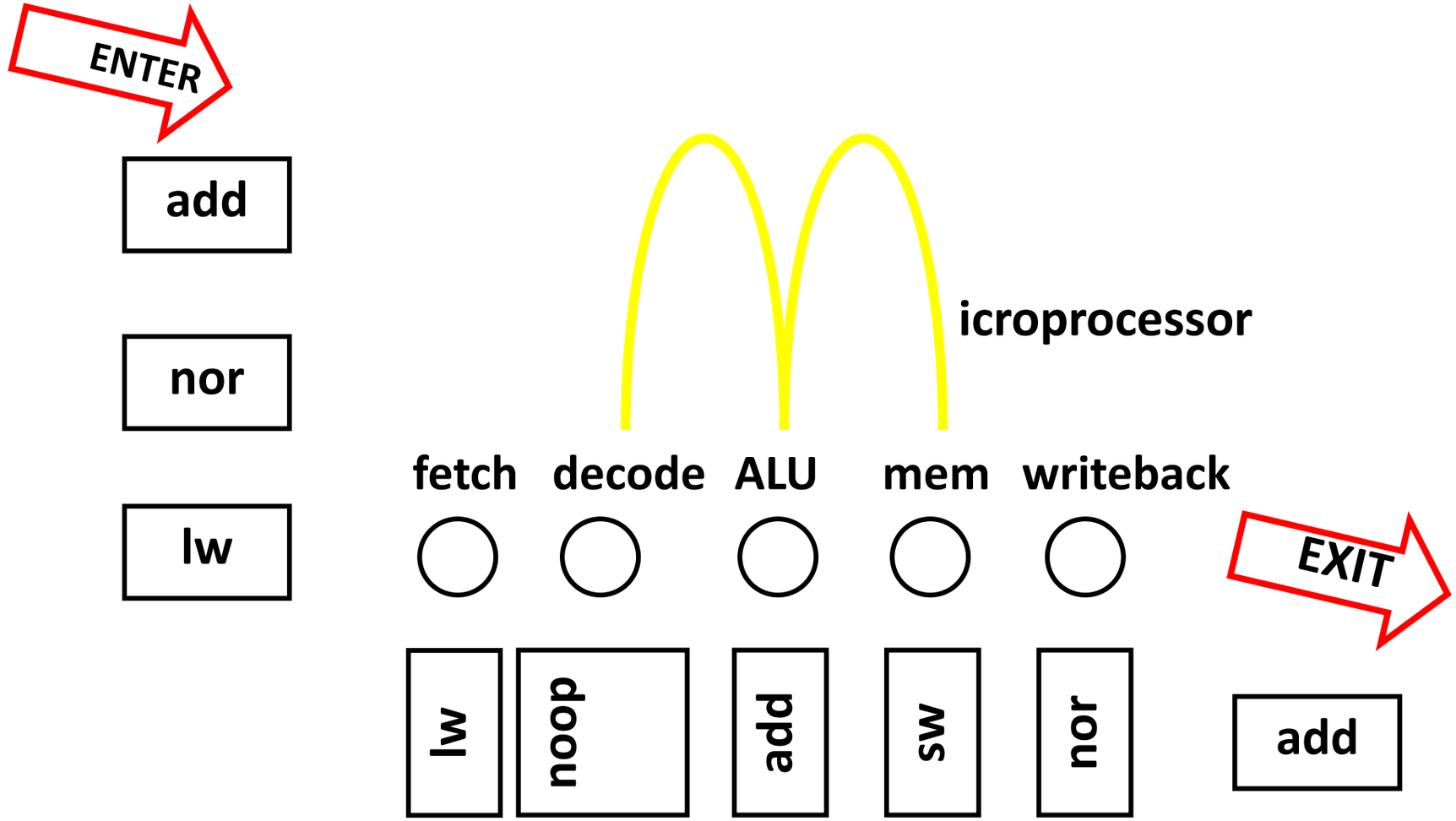
Pipelining

- Want to execute an instruction?
  - Build a processor (multi-cycle)
  - Find instructions
  - Line up instructions (1, 2, 3, …)
  - Overlap execution
    - Cycle #1: Fetch 1
    - Cycle #2: Decode 1 Fetch 2
    - Cycle #3: ALU 1 Decode 2 Fetch 3
    - . . . . . .
  - This is called pipelining instruction execution.
  - Used extensively for the first time on IBM 360 (1960s).
  - CPI approaches 1.

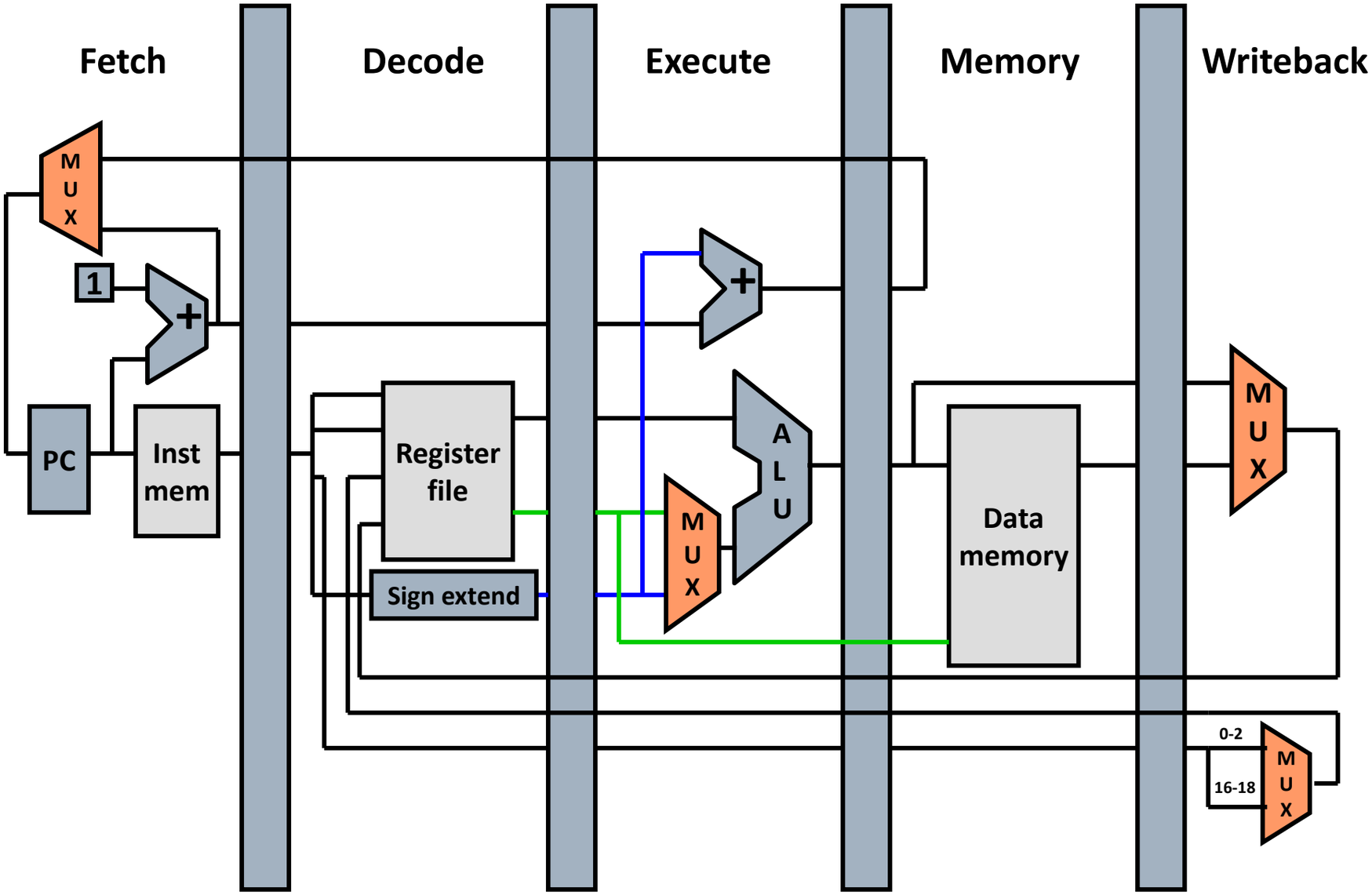
Pipelining



Pipelining


ENTER
add
nor
lw
icroprocessor
fetch
decode
ALU
mem
writeback
EXIT
lw
noop
add
sw
nor
add

Our new pipelined datapath

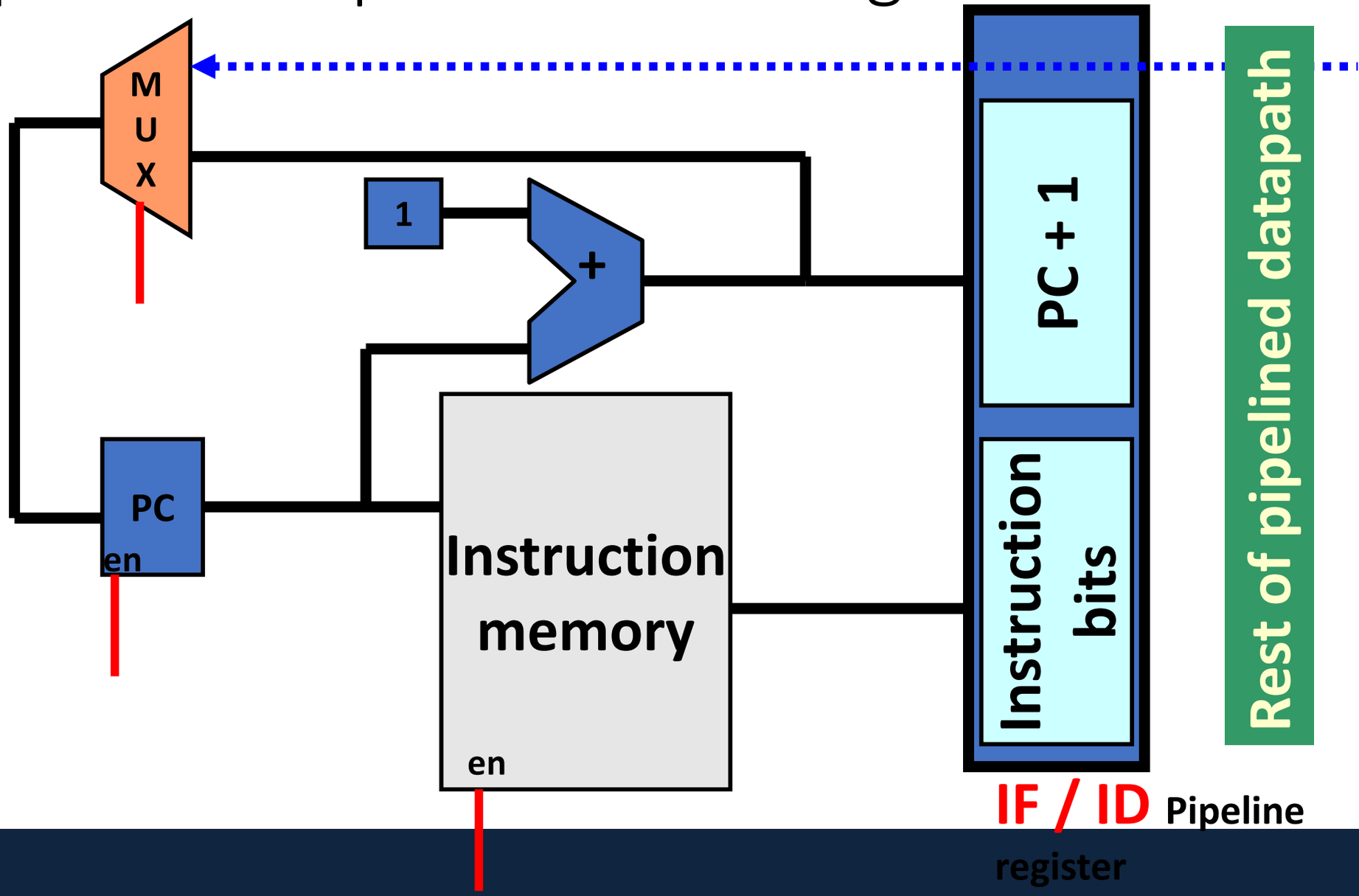


Stage 1: Fetch

- Design a datapath that can fetch an instruction from memory every cycle.
  - Use PC to index memory to read instruction
  - Increment the PC (assume no branches for now)

- Write everything needed to complete execution to the pipeline register (IF/ID)
  - The next stage will read this pipeline register

Pipeline datapath – Fetch stage

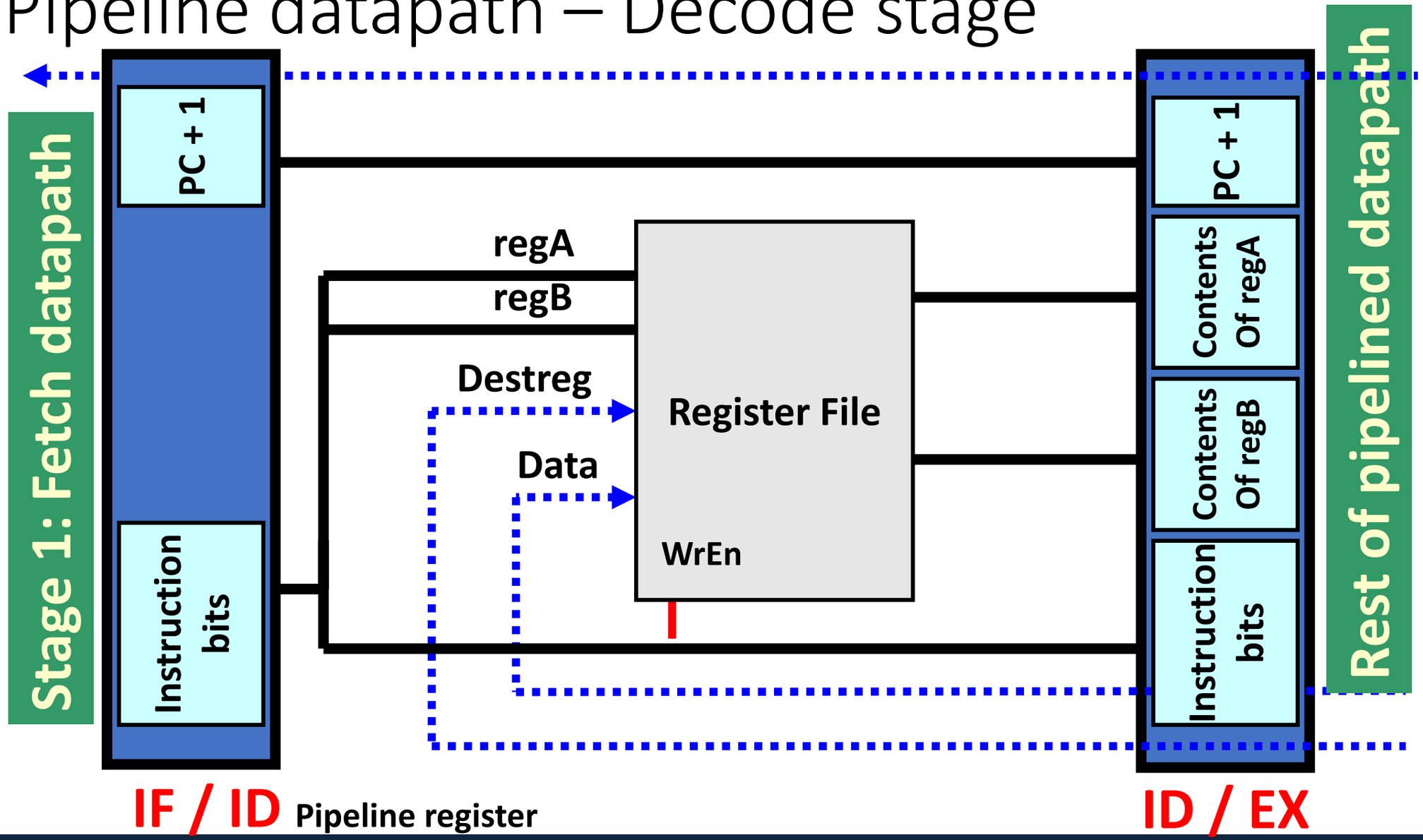


Stage 2: Decode

- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
  - Decode is easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.

- Write everything needed to complete execution to the pipeline register (ID/EX)
  - Pass on the offset field and both destination register specifiers (or simply pass on the whole instruction!).
  - Including PC+1 even though decode didn't use it.

Pipeline datapath – Decode stage

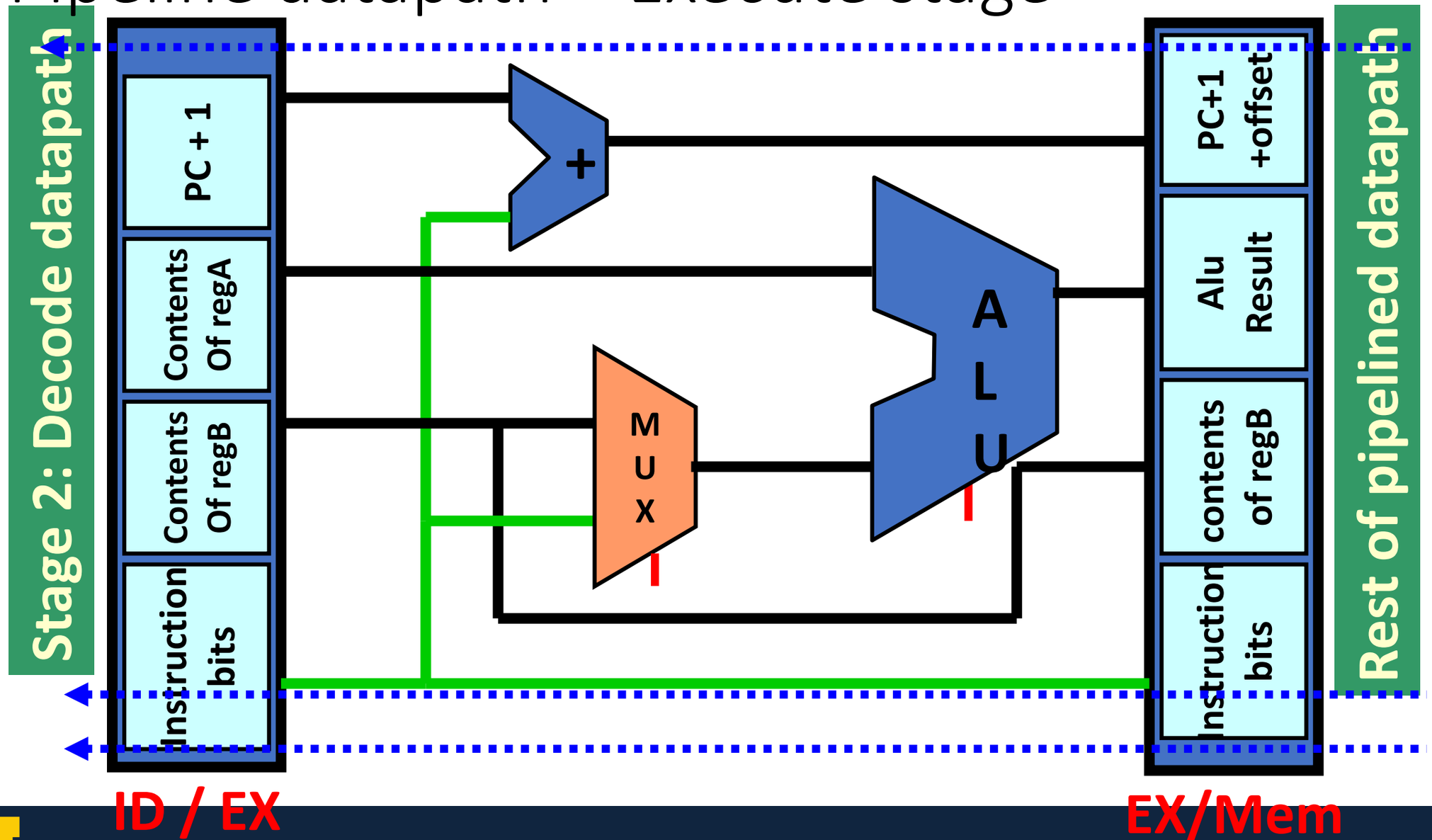


Stage 3: Execute

- Design a datapath that performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
  - The inputs are the contents of regA and either the contents of regB or the offset field on the instruction.
  - Also, calculate PC+1+offset in case this is a branch.

- Write everything needed to complete execution to the pipeline register (EX/Mem)
  - ALU result, contents of regB and PC+1+offset
  - Instruction bits for opcode and destReg specifiers
  - Result from comparison of regA and regB contents

Pipeline datapath – Execute stage

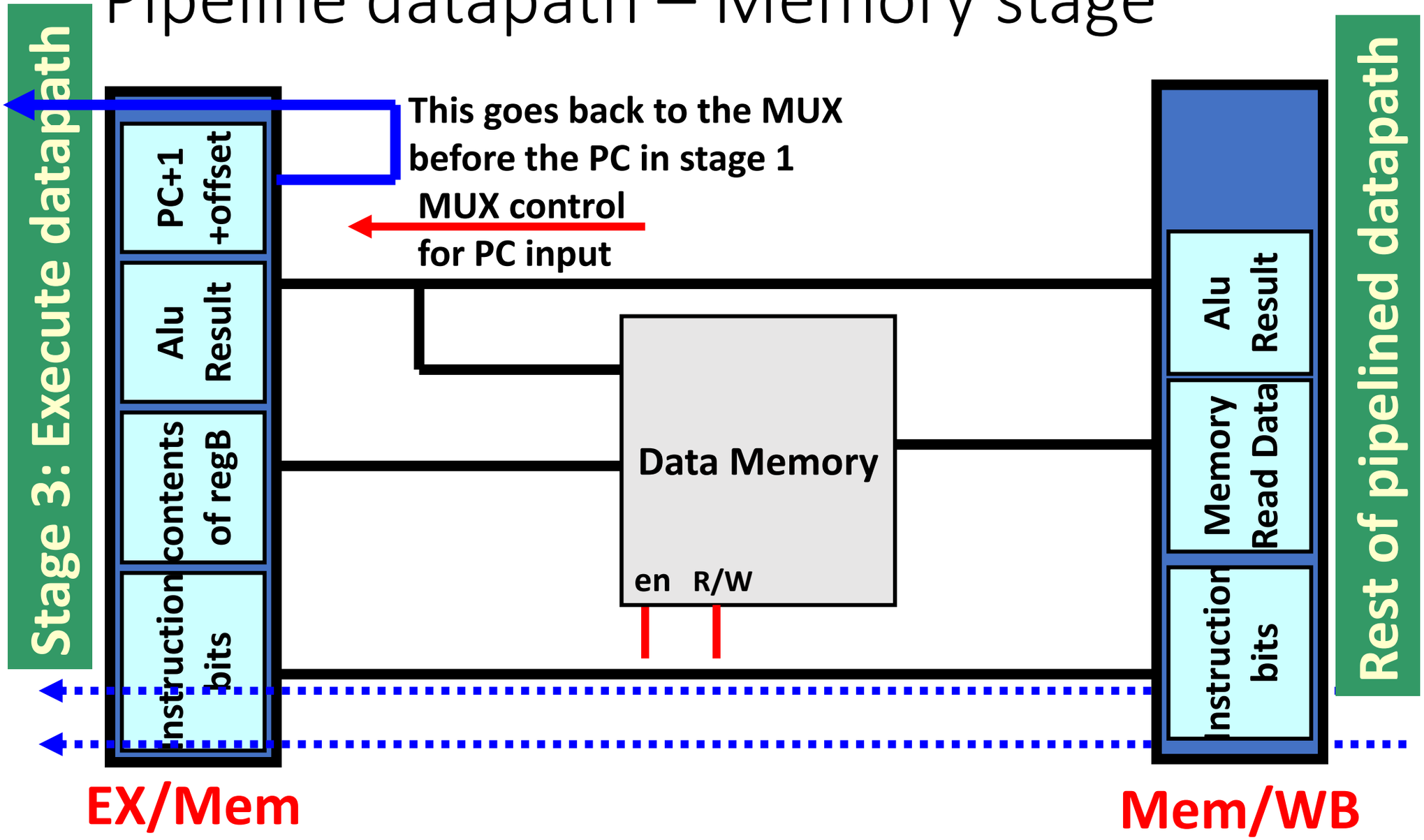


Stage 4: Memory Operation

- Design a datapath that performs the proper memory operation for the instruction specified and the values present in the EX/Mem pipeline register.
  - ALU result contains address for **ld** and **st** instructions.
  - Opcode bits control memory R/W and enable signals.

- Write everything needed to complete execution to the pipeline register (Mem/WB)
  - ALU result and MemData
  - Instruction bits for opcode and destReg specifiers

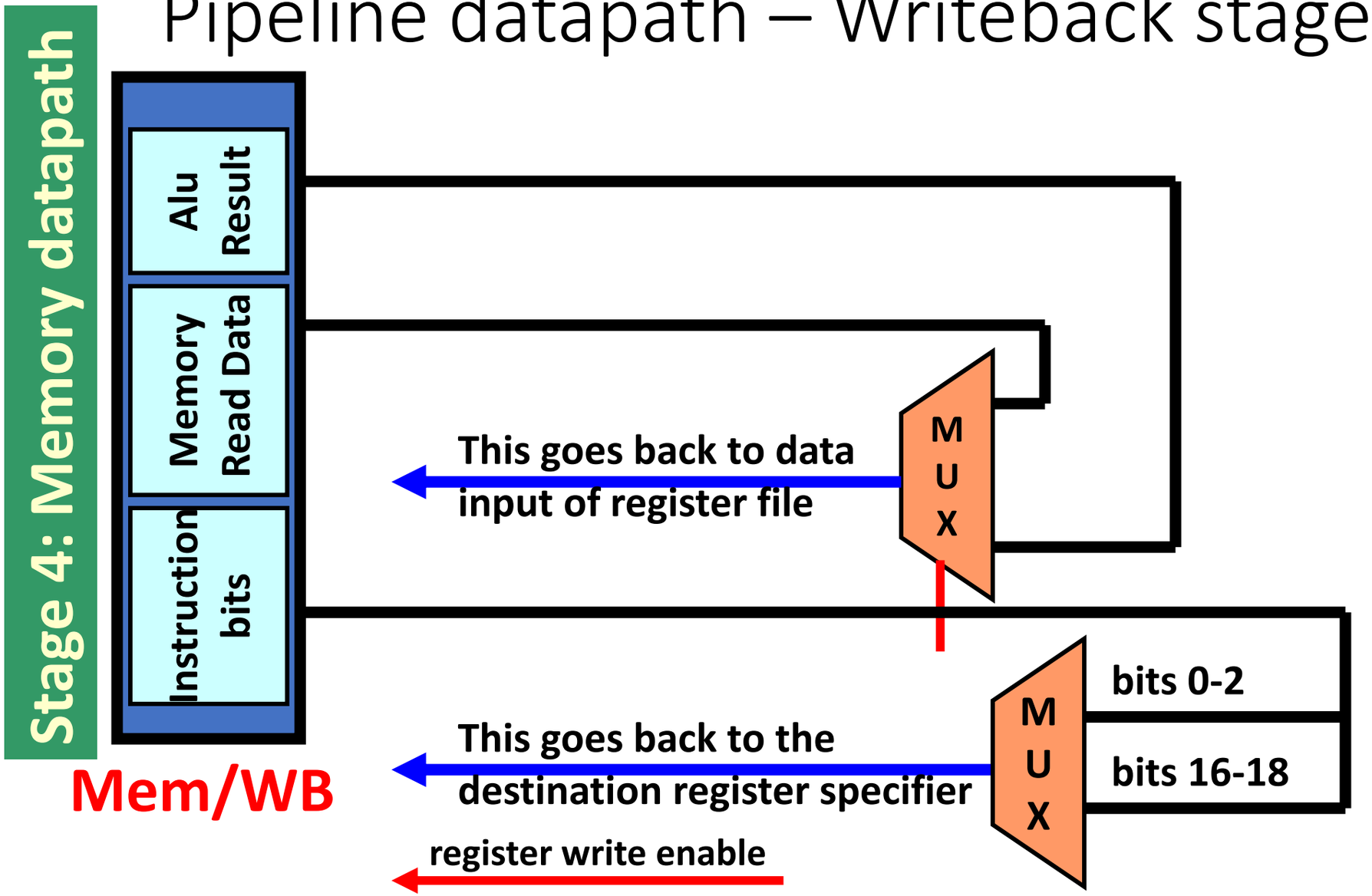
Pipeline datapath – Memory stage



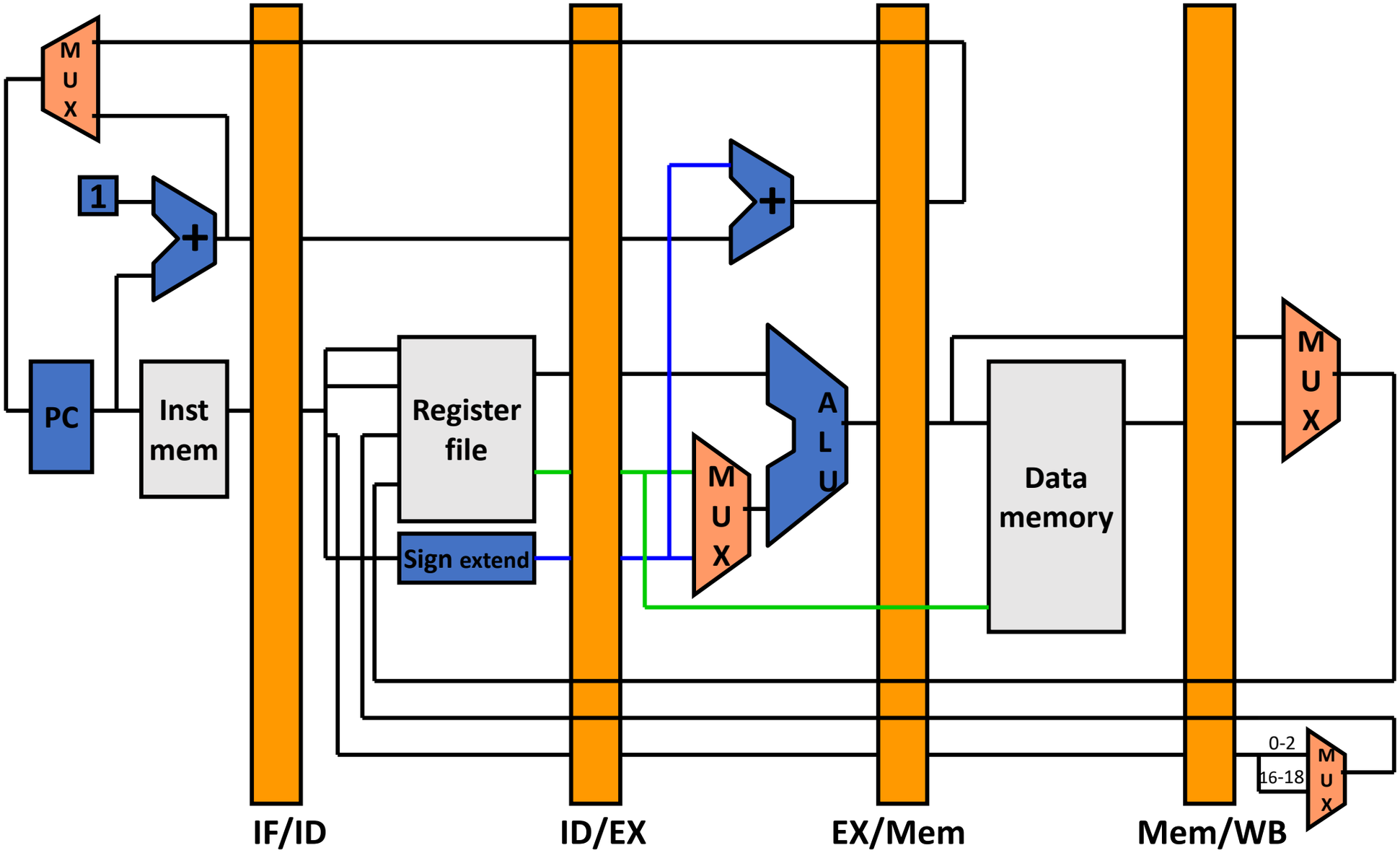
Stage 5: Write back

- Design a datapath that completes the execution of this instruction, writing to the register file if required.
  - Write MemData to destReg for ld instruction
  - Write ALU result to destReg for add or nor instructions.
  - Opcode bits also control register write enable signal.

Pipeline datapath – Writeback stage



Putting all together

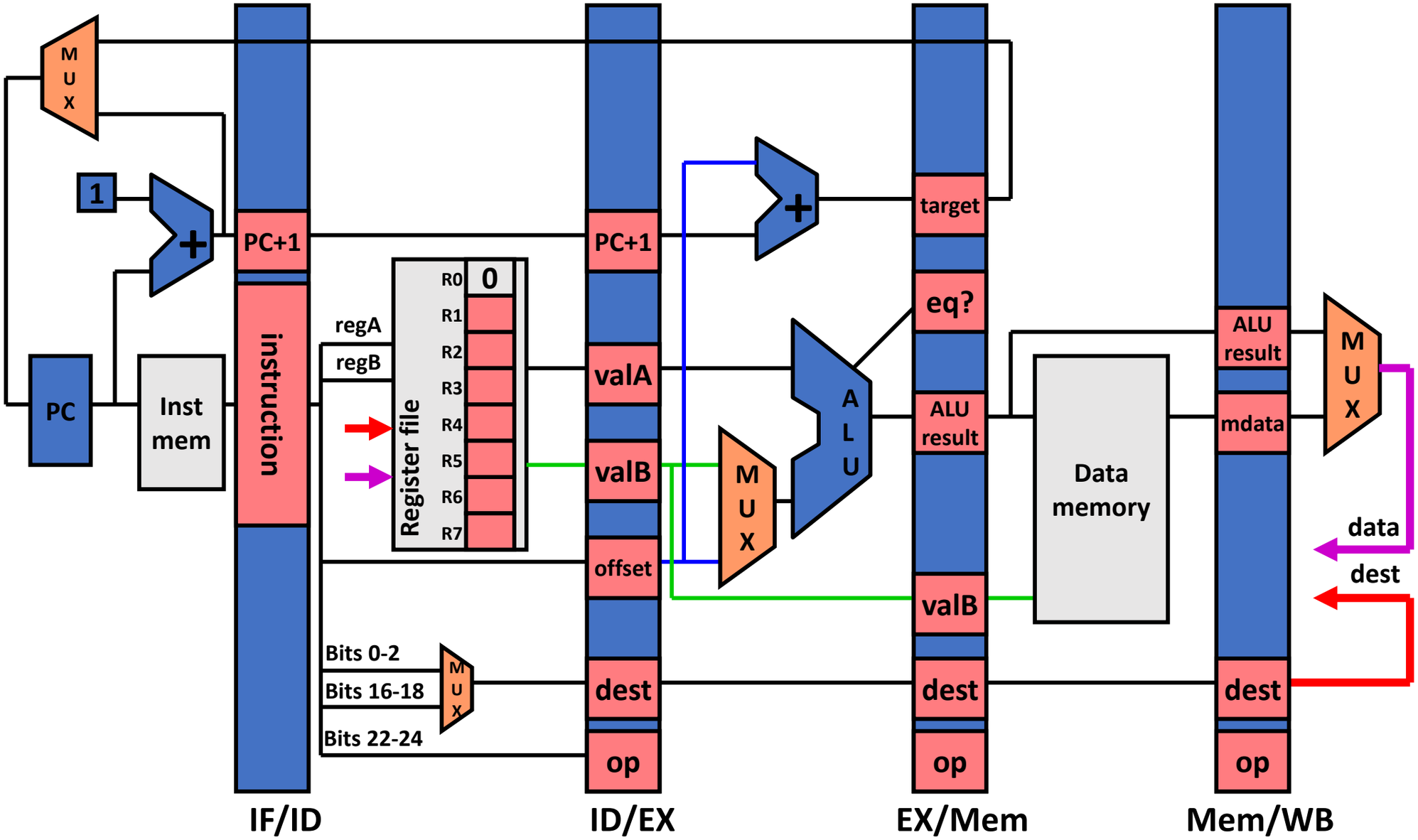


Sample Code (Simple)

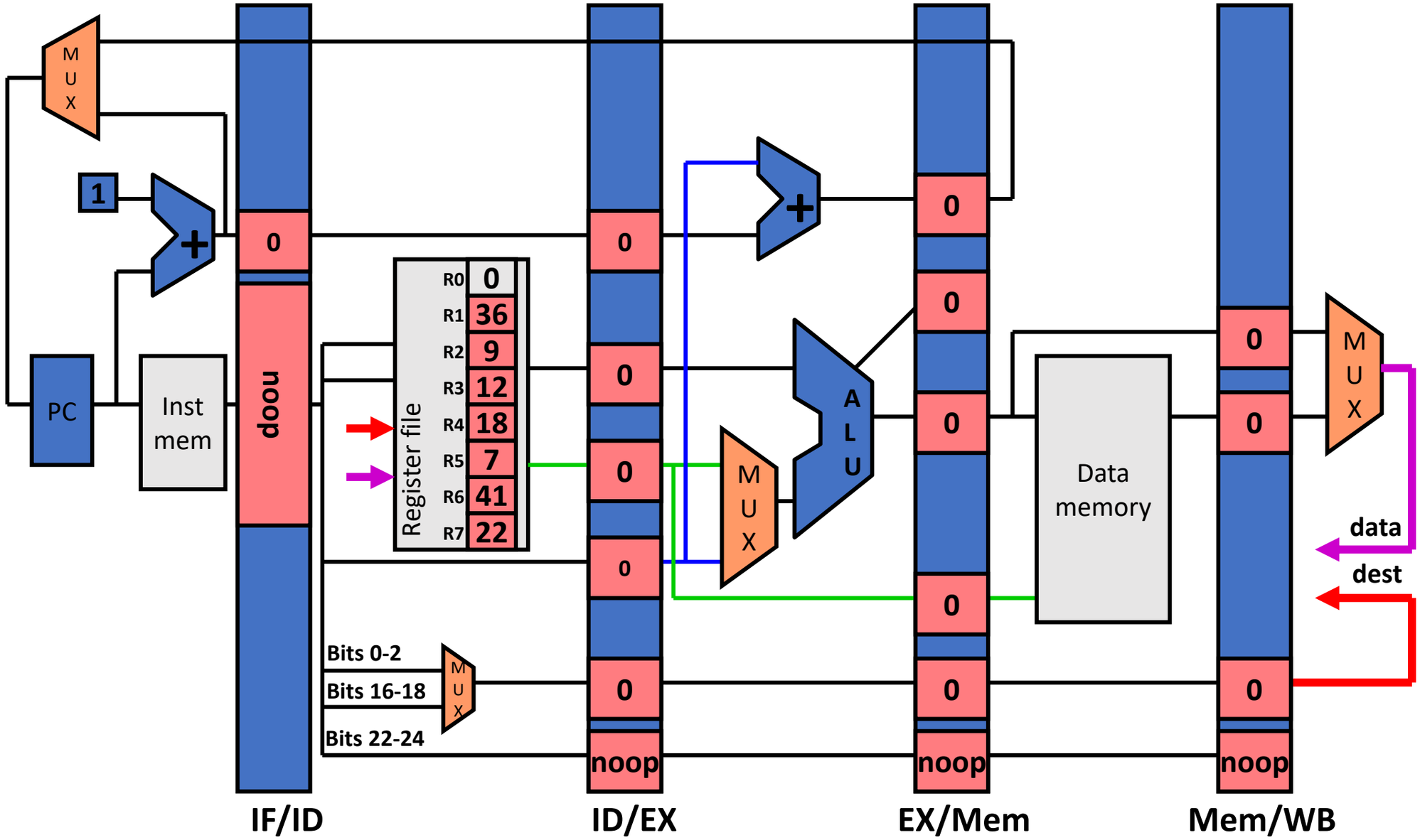
Let's run the following code on pipelined LC2K:

```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```

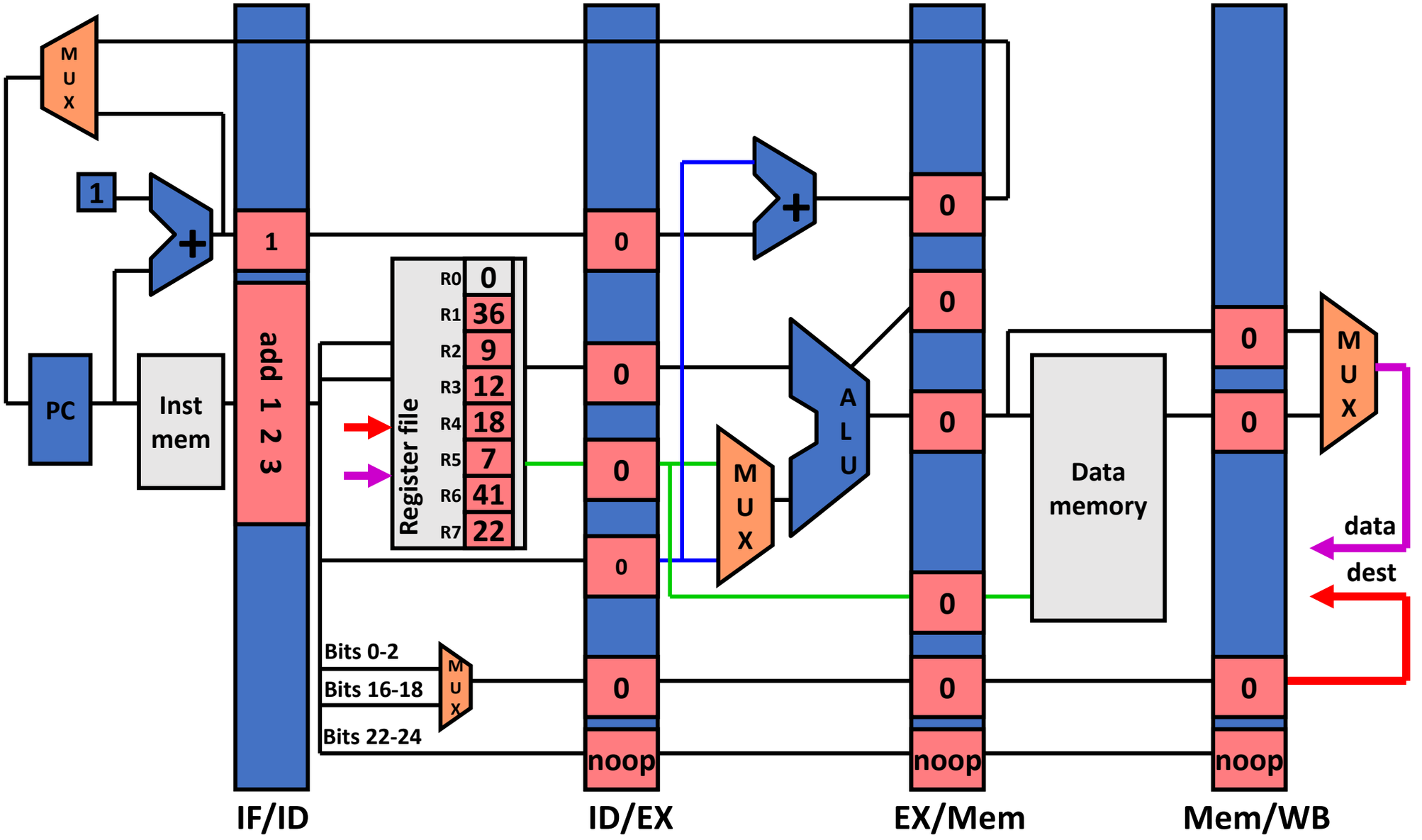
Pipeline datapath



Time 0 - Initial state

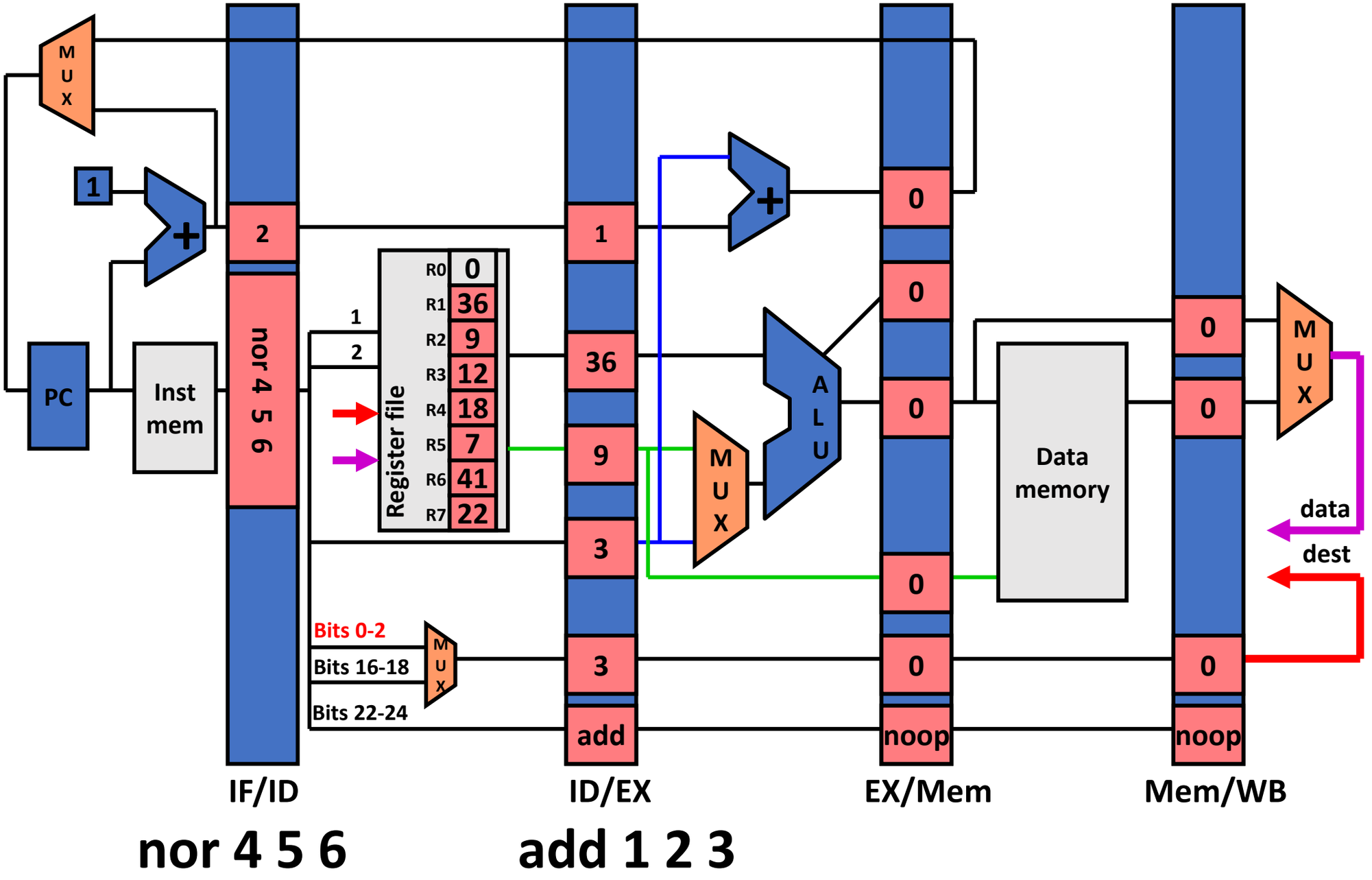


Time 1 - Fetch: add 1 2 3

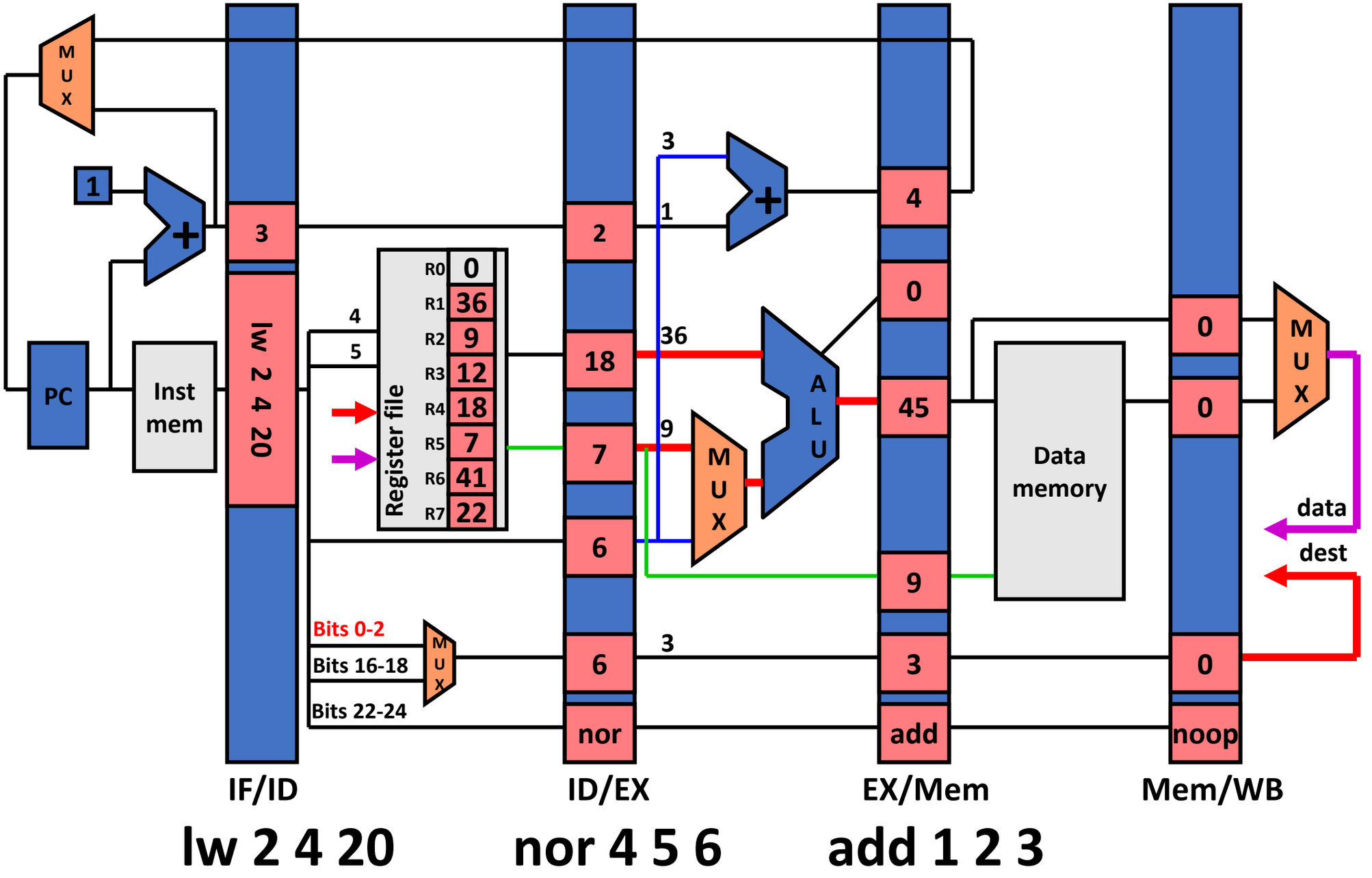


add 1 2 3

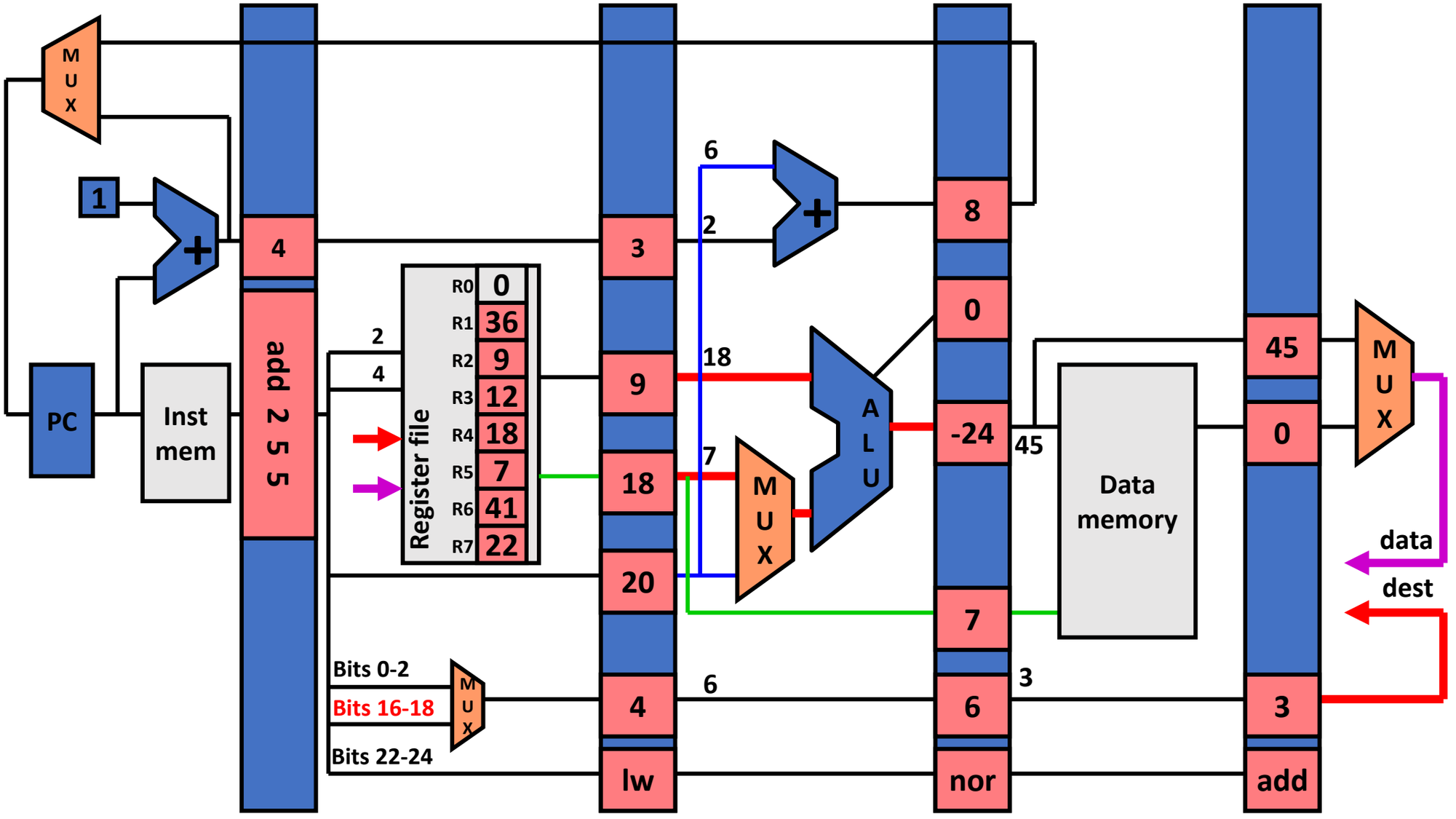
Time 2 - Fetch: nor 4 5 6



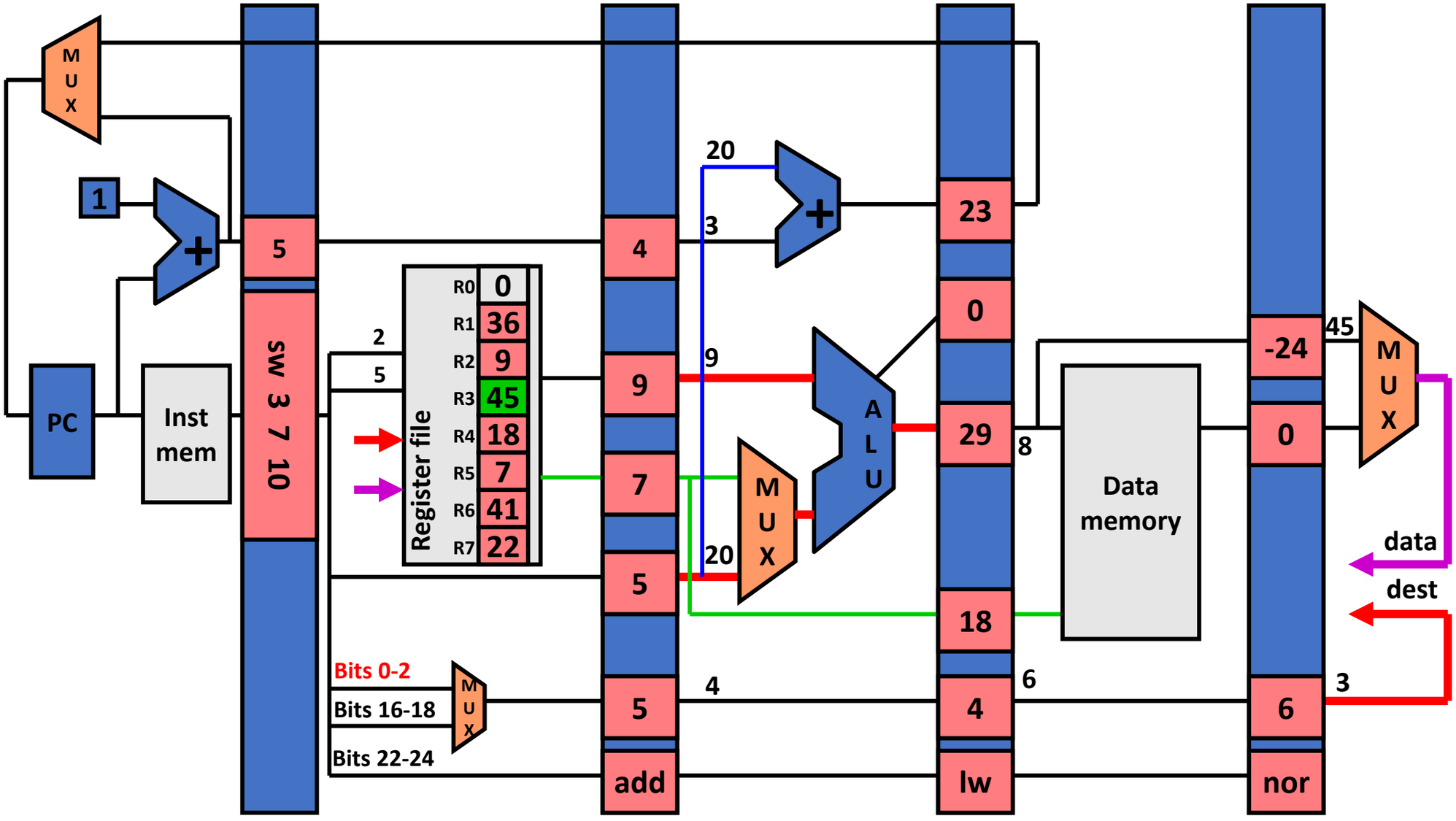
Time 3 - Fetch: lw 2 4 20



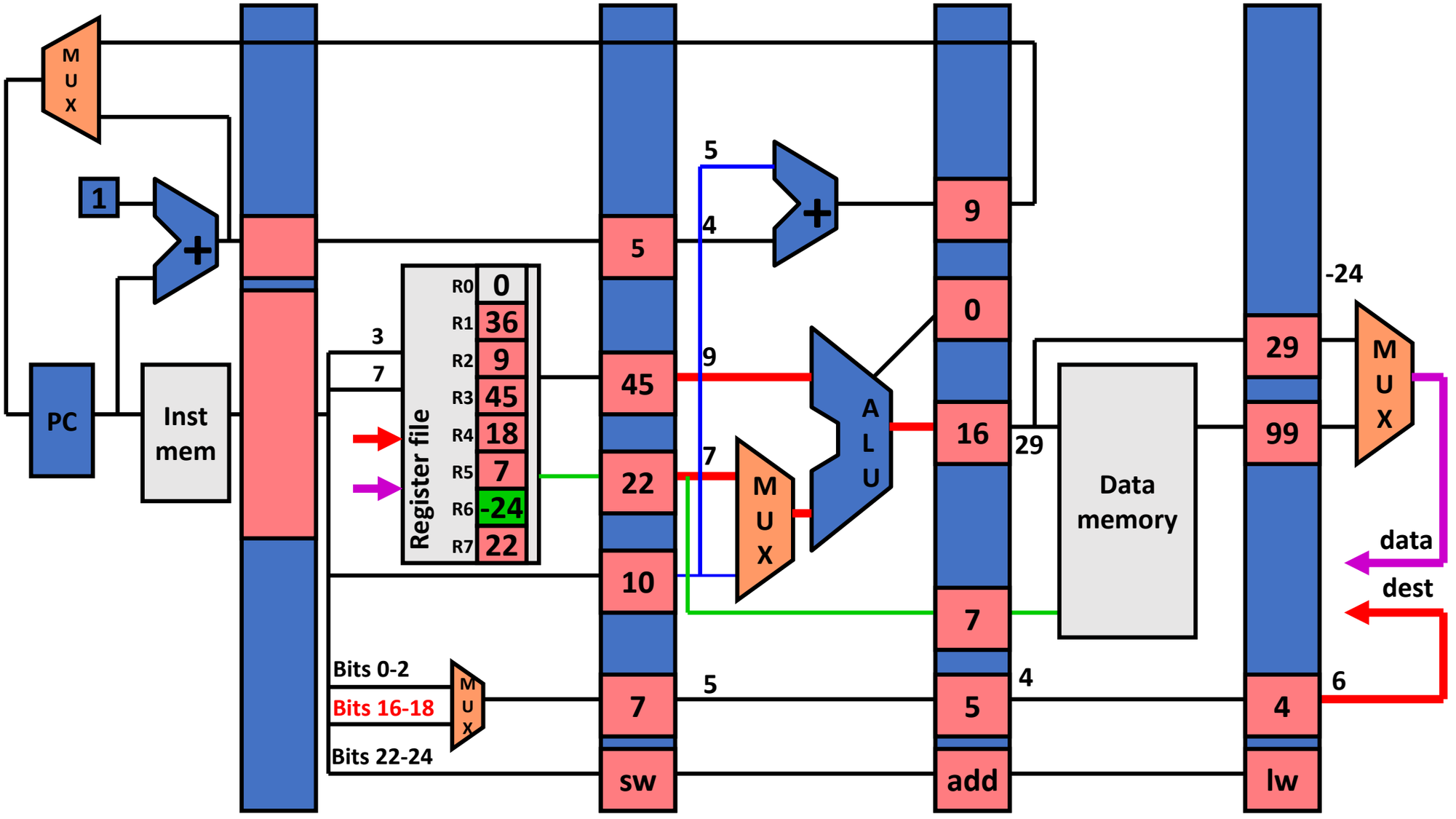
Time 4 - Fetch: add 2 5 5



Time 5 - Fetch: sw 3 7 10

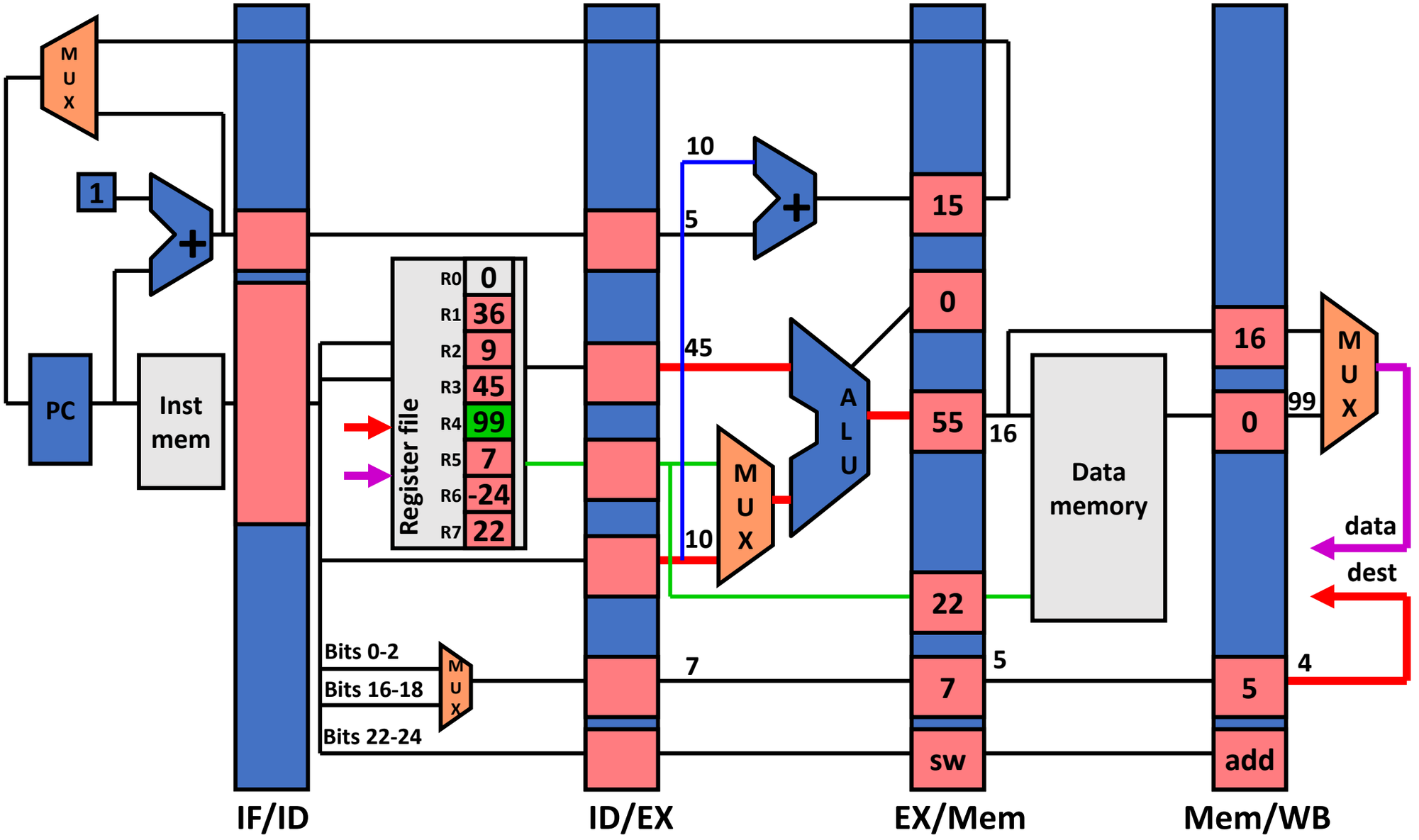


Time 6 – no more instructions



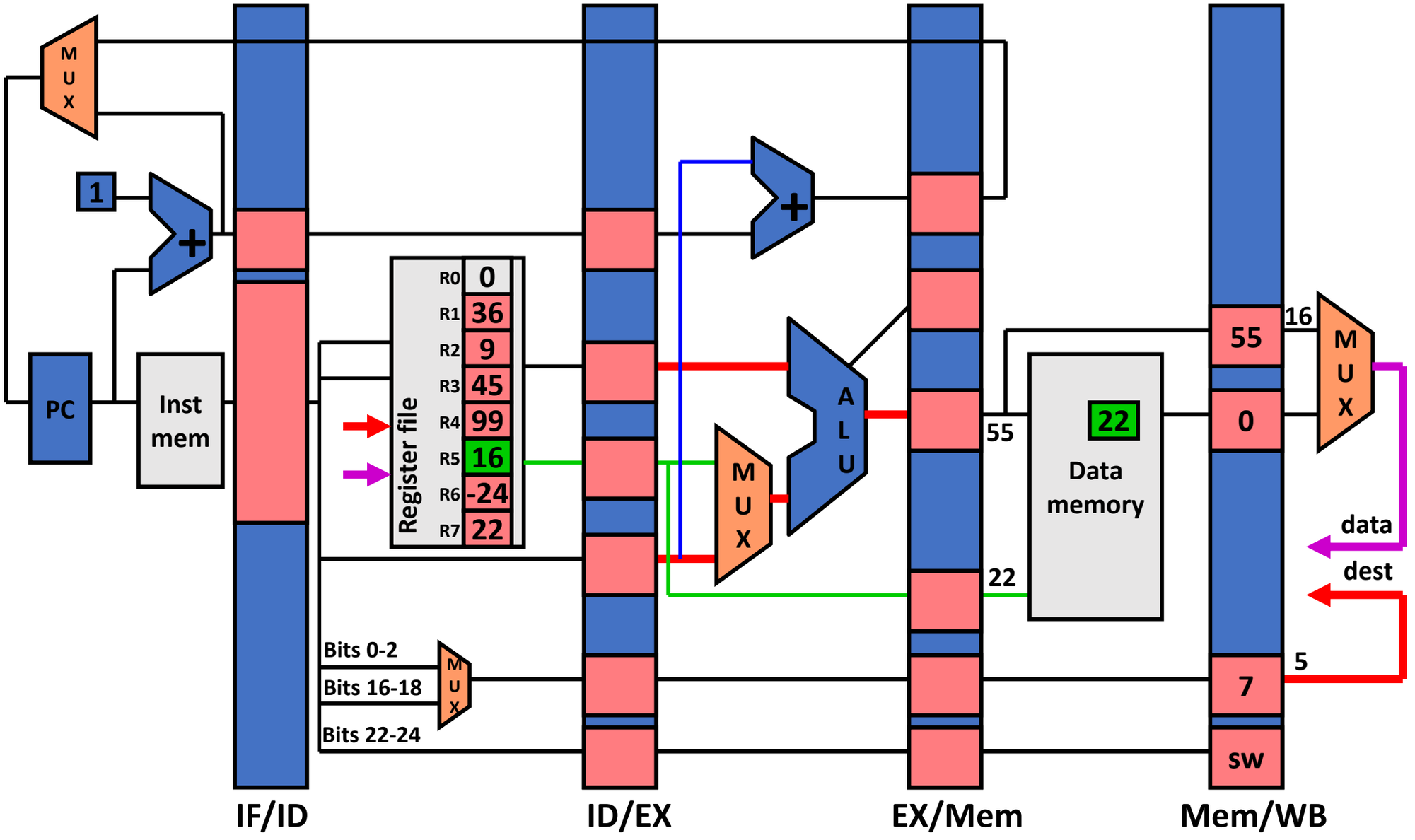
Time 7 – no more instructions

What all happens on the next clock cycle?



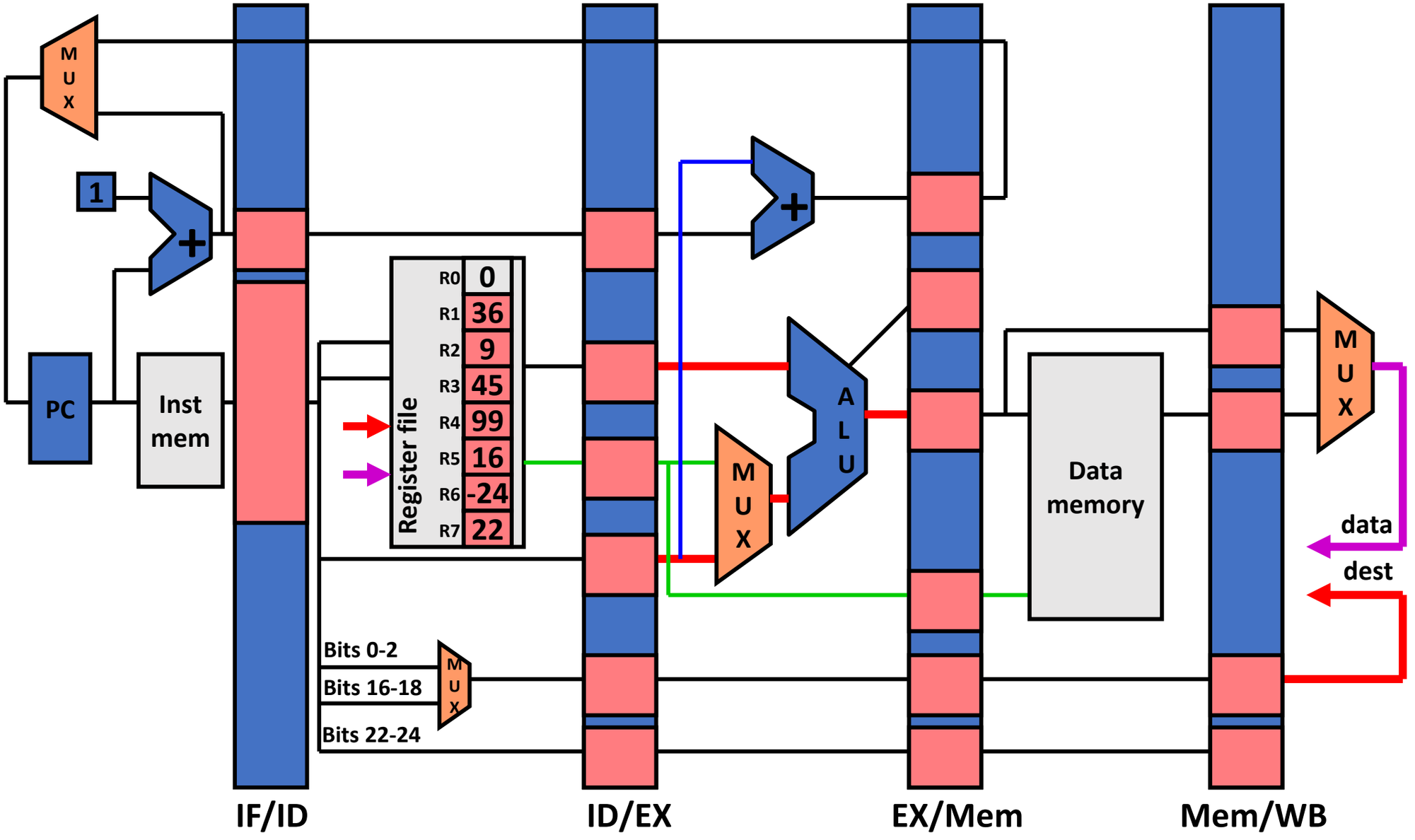
sw 3 7 10 **add 2 5 5** **lw**

Time 8 – no more instructions



sw 3 7 10 **add**

Time 9 – no more instructions



SW

Pipelining - What can go wrong?

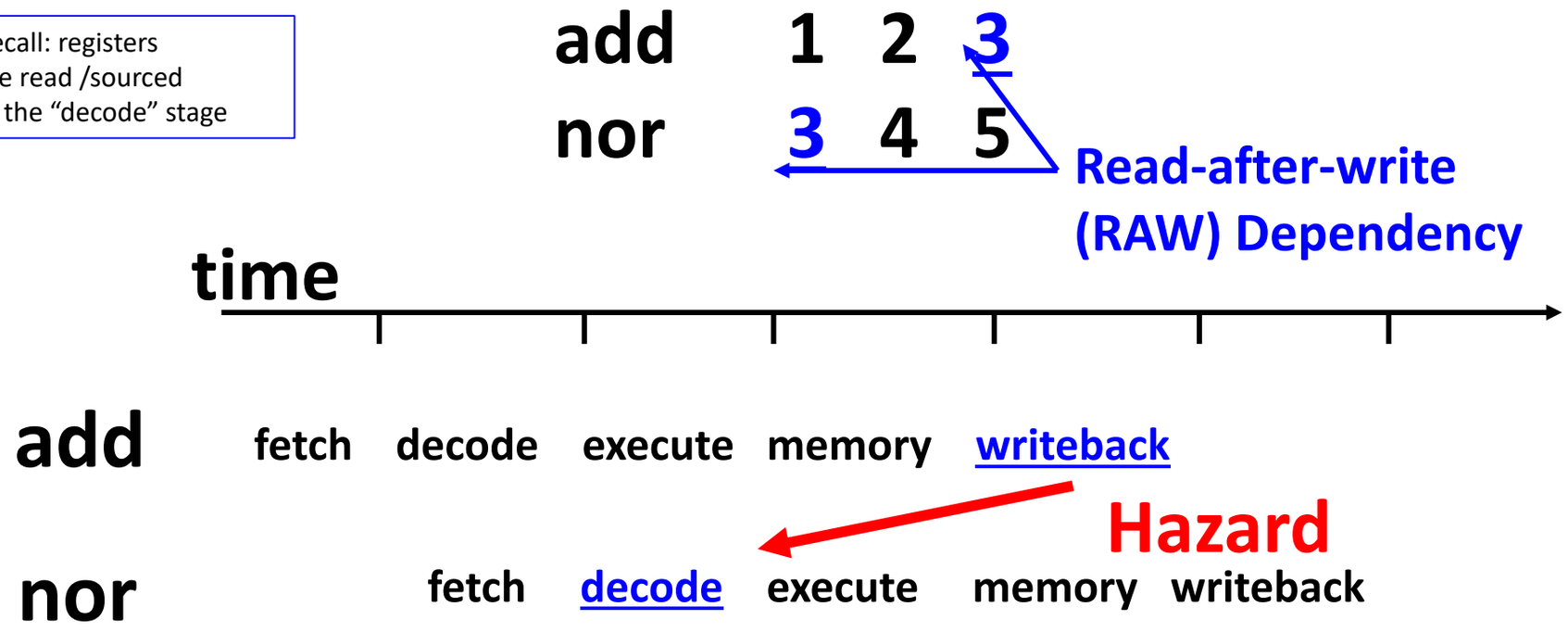
- Data hazards: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read an old / stale value before the correct value is written back.
- Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- Exceptions: Sometimes we need to pause execution, switch to another task (maybe the OS), and then resume execution… how to we make sure we resume at the right spot
- Now - Data hazards
  - What are they?
  - How do you detect them?
  - How do you deal with them?

Pipeline function for ADD

- Fetch: read instruction from memory
- Decode: **read source operands from reg**
- Execute: calculate sum
- Memory: pass results to next stage
- Writeback: **write sum into register file**

Data Hazards

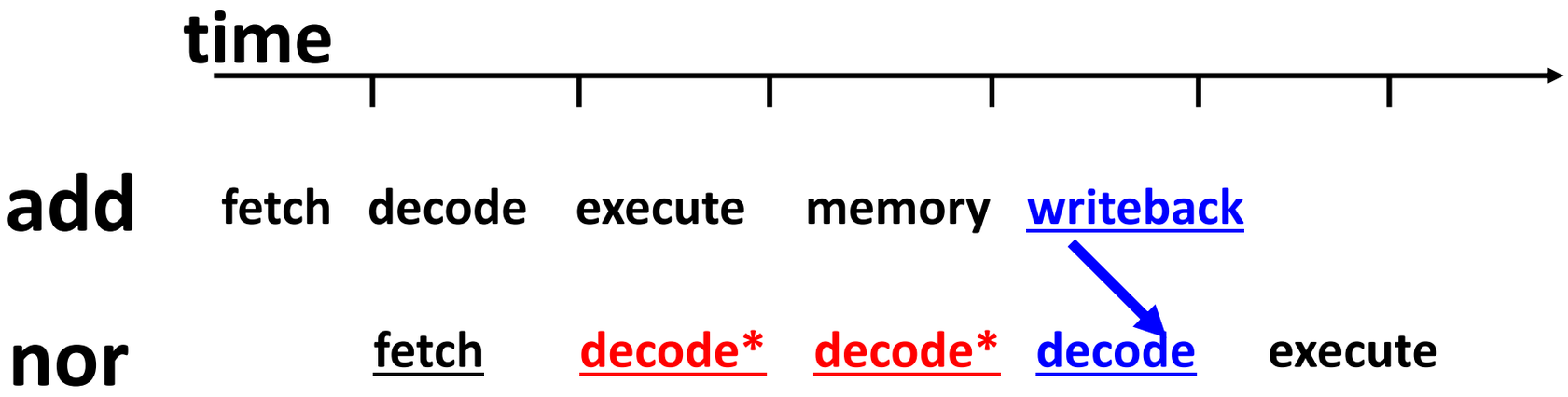
Recall: registers
are read /sourced
In the “decode” stage



If not careful, nor will read a stale value of register 3

Data Hazards

add 1 2 3
nor 3 4 5



Assume Register File gives the right value of register 3 when read/written during same cycle. This is consistent with most processors (ARM/x86), but not Project 3.

Definitions

- Data Dependency: *one instruction uses the result of a previous one*
  - Doesn't necessarily cause a problem
- Data Hazard: *one instruction has a data dependency that will cause a problem if we don't "deal with it"*

Class Problem 1

Which of these instructions has a data dependency on an earlier one? Which of those are data hazards in our 5-stage pipeline?

1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. lw 3 6 10
5. sw 6 2 12

What about here?

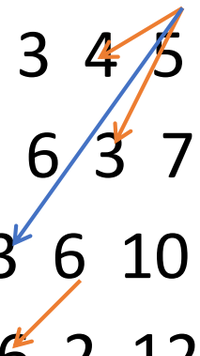
1. add 1 2 3
2. beq 3 4 1
3. add 3 5 6
4. add 3 6 7

Class Problem 1

Which read-after-write (RAW) dependences do you see?

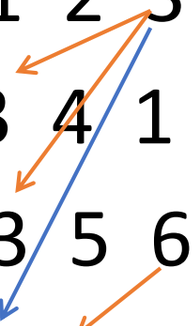
Which of those are data hazards?

1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. lw 3 6 10
5. sw 6 2 12



What about here?

1. add 1 2 3
2. beq 3 4 1
3. add 3 5 6
4. add 3 6 7



Three approaches to handling data hazards

- Avoid
  - Make sure there are no hazards in the code
- Detect and Stall
  - If hazards exist, stall the processor until they go away.
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)

Handling data hazards I: Avoid all hazards

- Assume the programmer (or the compiler) knows about the processor implementation.
  - Make sure no hazards exist.
    - Put noops between any dependent instructions.

```
add    1   2   3
noop
noop
nor    3   4   5
```

write register 3 in cycle 5

read register 3 in cycle 5

Problems with this solution

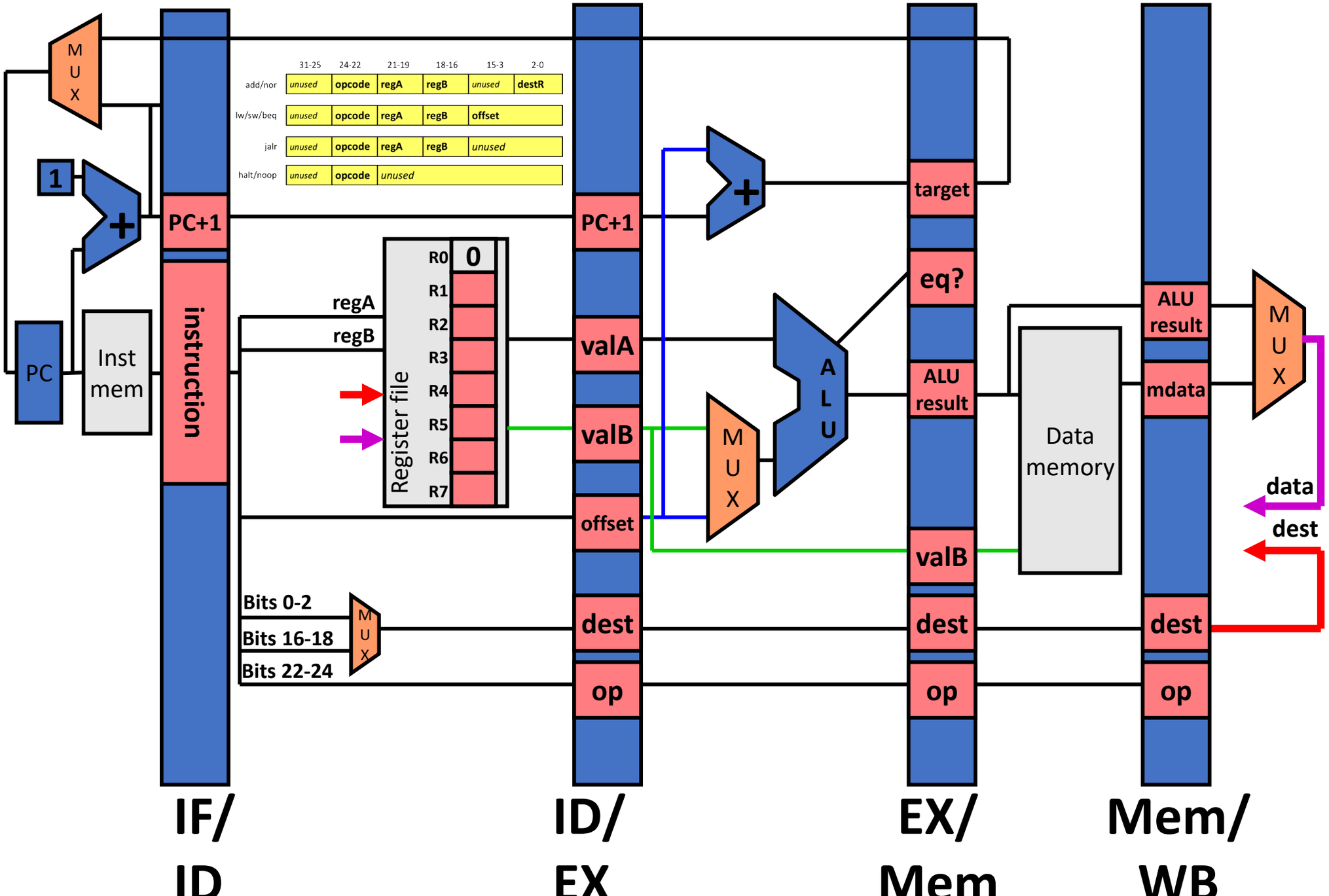
- Old programs (legacy code) may not run correctly on new implementations
  - Longer pipelines need more noops
- Programs get larger as noops are included
  - Especially a problem for machines that try to execute more than one instruction every cycle
  - Intel EPIC: Often 25% - 40% of instructions are noops
- Program execution is slower
  - CPI is 1, but some instructions are noops

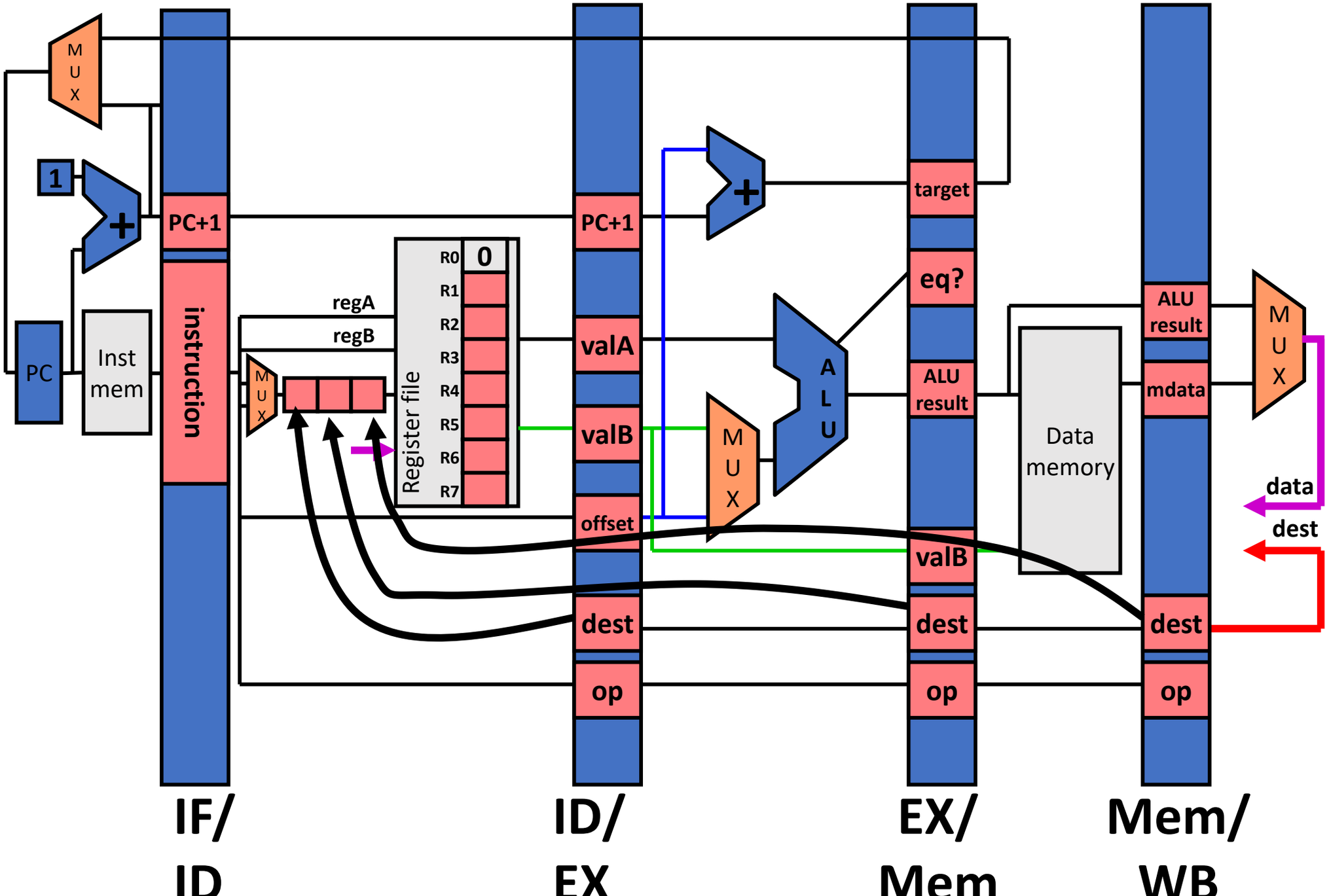
Handling data hazards II: Detect and stall until ready

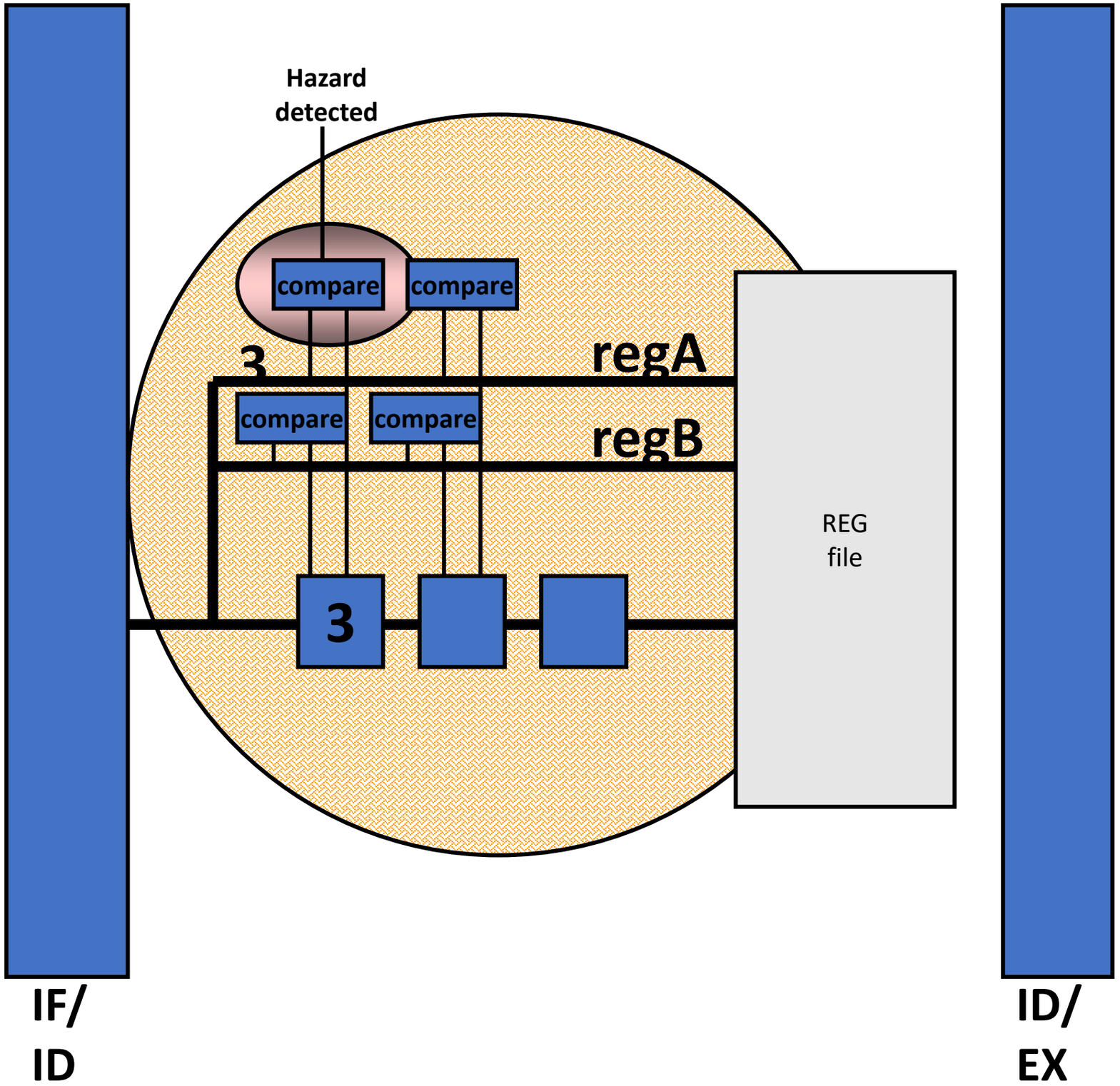
- Detect:
  - Compare regA with previous DestRegs
    - 3 bit operand fields
  - Compare regB with previous DestRegs
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute

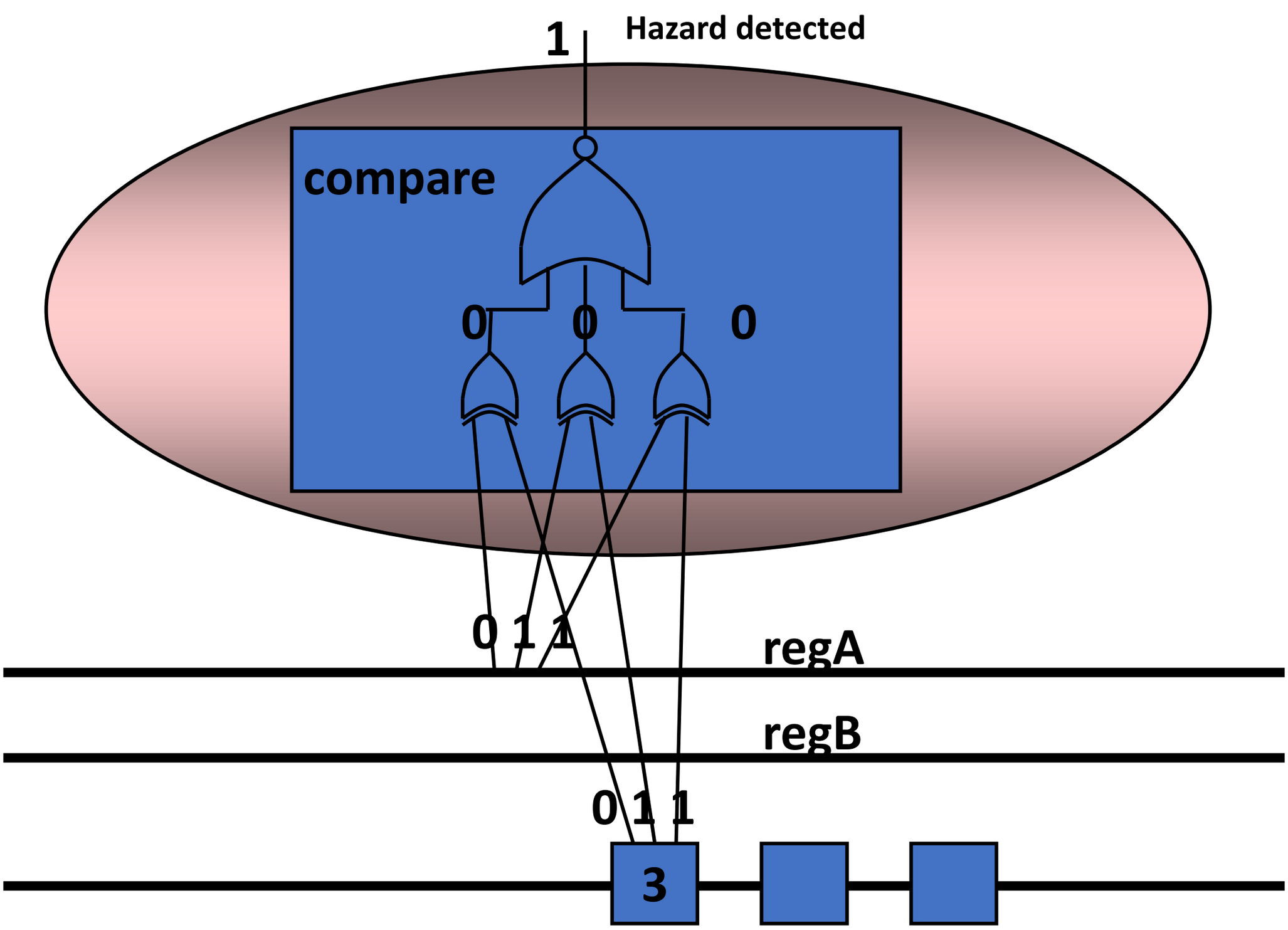
- How do we modify the pipeline to do this?

Our pipeline currently does not handle hazards—let's fix it




M
U
X
1
+
PC
Inst
mem
PC+1
instruction
regA
regB
M
U
X
Register file
R0
R1
R2
R3
R4
R5
R6
R7
0
PC+1
valA
valB
offset
dest
op
+
M
U
X
A
L
U
target
eq?
ALU
result
valB
dest
op
Data
memory
ALU
result
mdata
dest
op
M
U
X
data
dest
IF/
ID
ID/
EX
EX/
Mem
Mem/
WB


Hazard
detected
compare
compare
3
regA
compare
compare
regB
REG
file
3
IF/
ID
ID/
EX


1
Hazard detected
compare
0
0
0
0 1 1
regA
regB
0 1 1
3

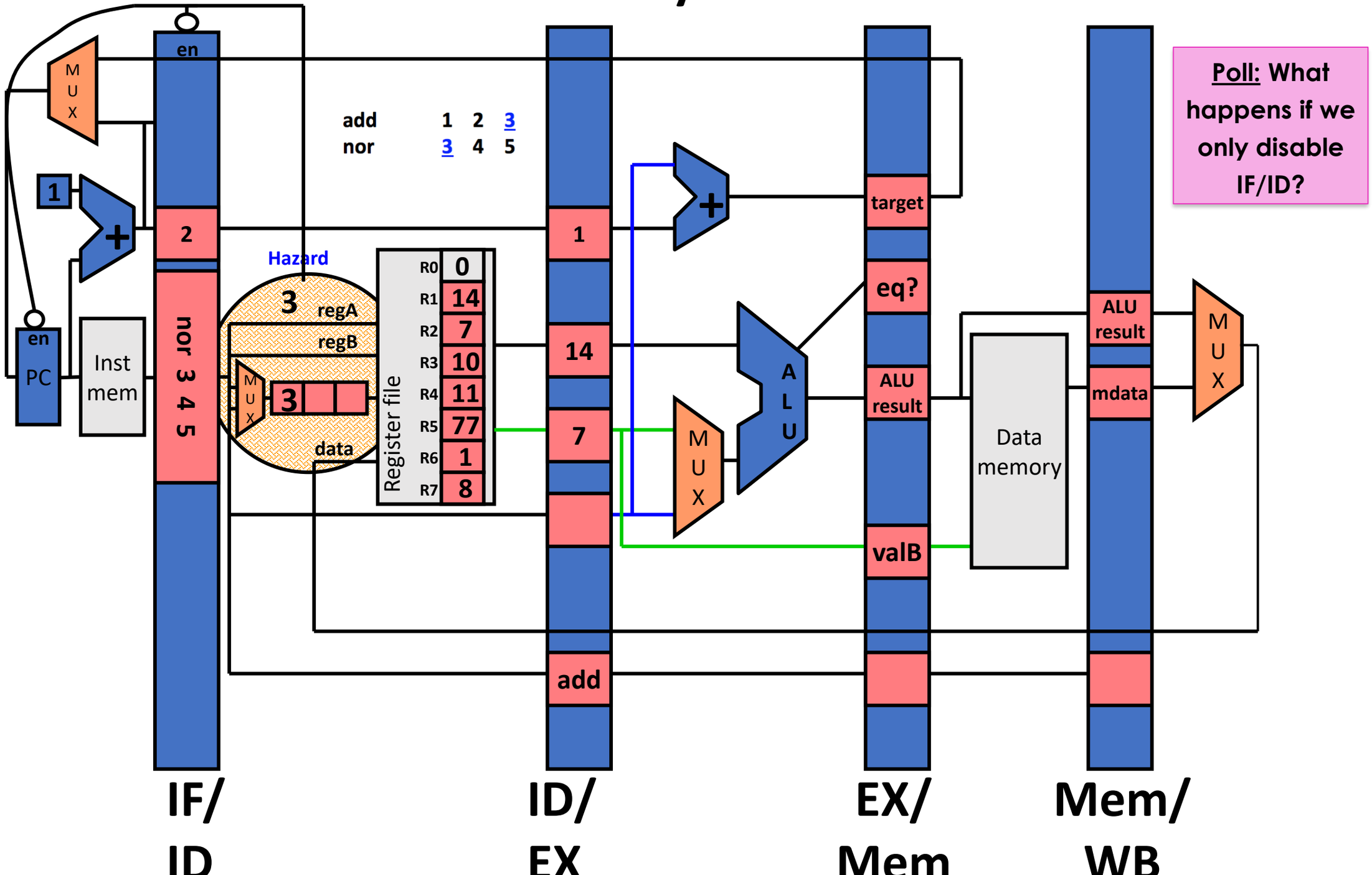
Example

- Let's run this program with a data hazard through our 5-stage pipeline
  add 1 2 3
  nor 3 4 5
- We will start at the beginning of cycle 3, where add is in the EX stage, and nor is in the ID stage, about to read a register value

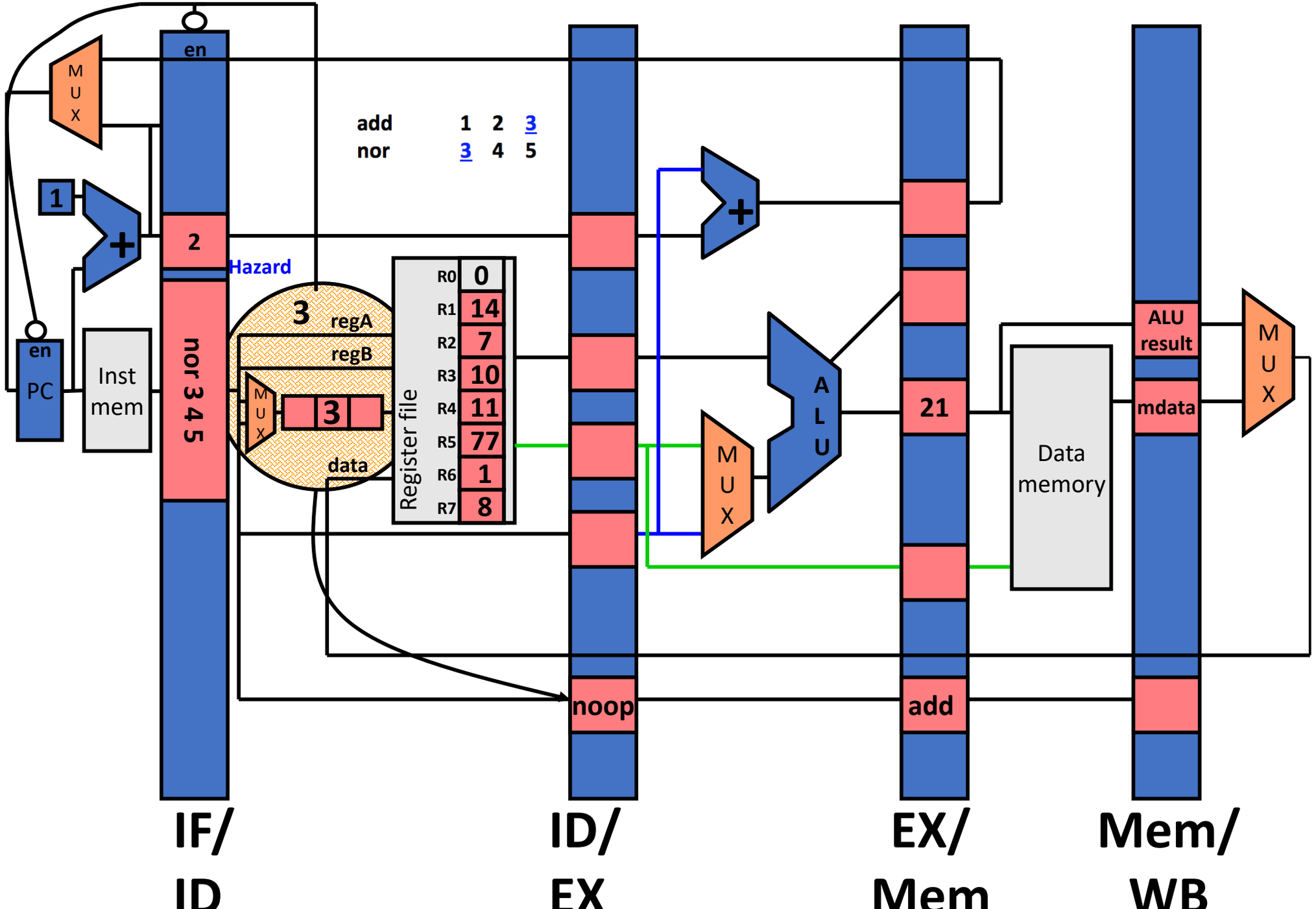
Time:	1	2	3
add 1 2 3	IF	ID	EX
nor 3 4 5		IF	ID

Hazard!

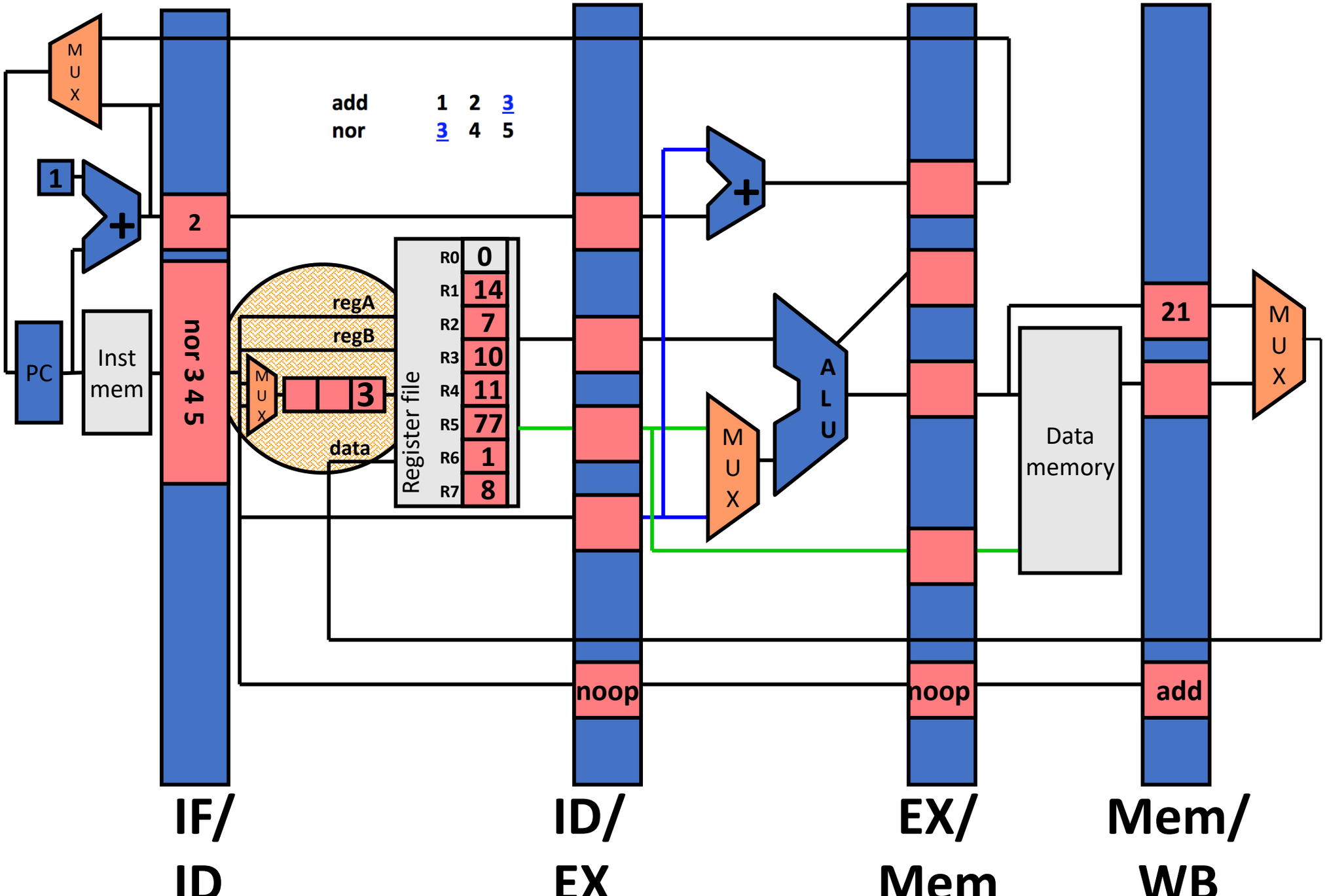
First half of cycle 3



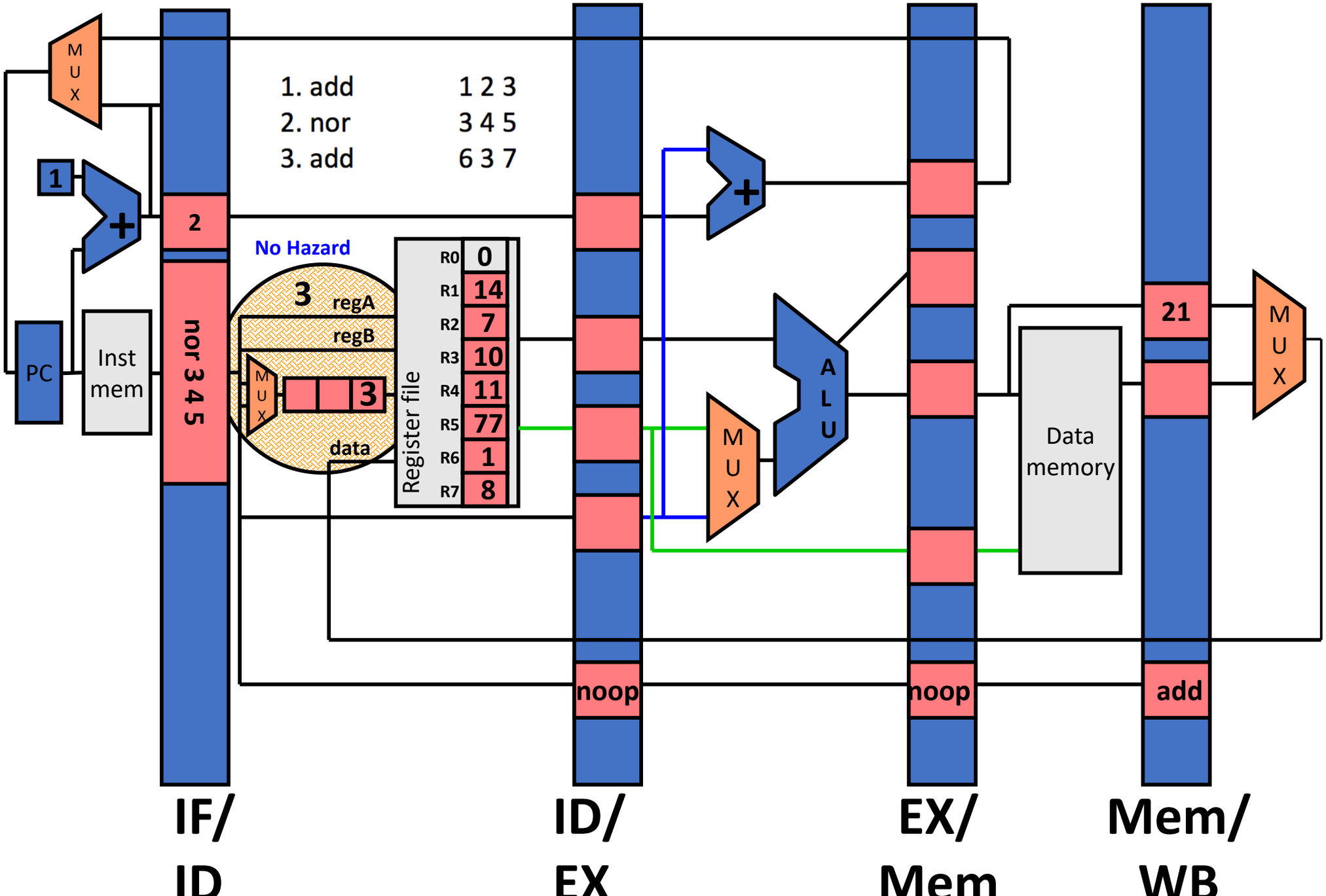
First half of cycle 4



End of cycle 4

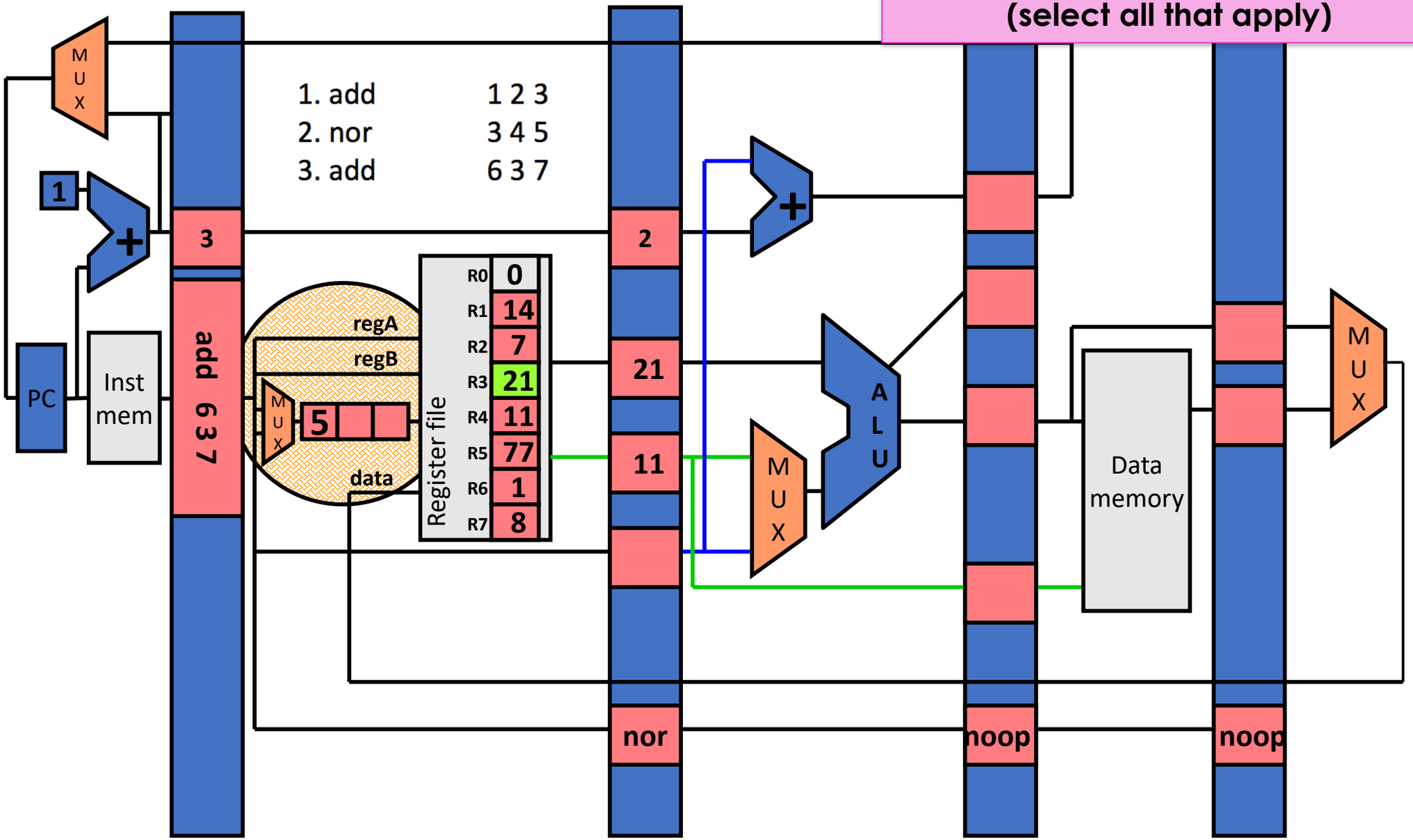


First half of cycle 5



End of cycle 5

How many cycles of delay can each hazard introduce? 3, 2, 1, 0 (select all that apply)



Time Graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF												
nor 3 4 5													

Time Graph

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID*	ID*	ID	EX	ME	WB					
add 6 3 7													
lw 3 6 10													
sw 6 2 12													

Solution

Which problems does "detect and stall" fix over "avoid hazards"? (select all)

1. Breaking backwards compatibility
2. Larger programs
3. Slower programs

Time:	1	2	3	4	5	6	7	8	9	10	11	12	13
add 1 2 3	IF	ID	EX	ME	WB								
nor 3 4 5		IF	ID*	ID*	ID	EX	ME	WB					
add 6 3 7					IF	ID	EX	ME	WB				
lw 3 6 10						IF	ID	EX	ME	WB			
sw 6 2 12							IF	ID*	ID*	ID	EX	ME	WB

Next time

- Resolving Hazards