13. Basic Processor Design – Pipelining With Data Hazards

EECS 370 – Introduction to Computer Organization – Winter 2020

EECS Department
University of Michigan in Ann Arbor, USA

What’s on the schedule?

- P2a due today!
  - If you didn’t get full points on P1a, you’ll need help. See Piazza post @1247
- P2l due Thursday 2/23
  - This is the harder one!
- HW3 due 2/20

EECS 370: Introduction to Computer Organization

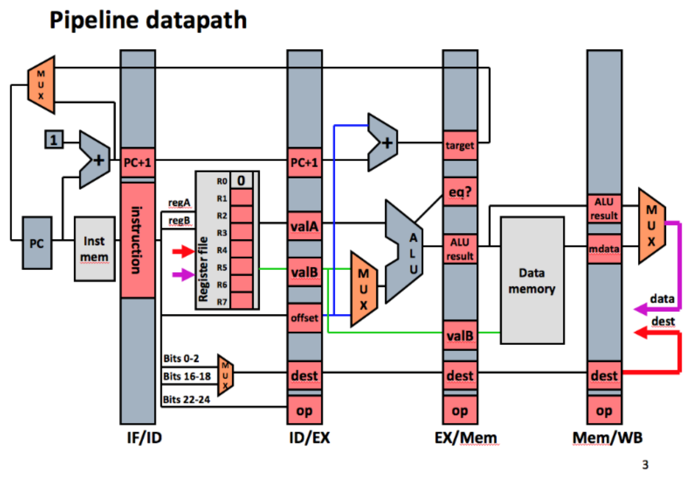
Time graphs (a.k.a. pipe trace)

Time:	1	2	3	4	5	6	7	8	9
add	fetch	decode	execute	memory	writeback				
nor		fetch	decode	execute	memory	writeback			
lw			fetch	decode	execute	memory	writeback		
add				fetch	decode	execute	memory	writeback	
sw					fetch	decode	execute	memory	writeback

A vertical slice reports the entire activity of the pipeline at time 5

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: How do you handle exceptions in a pipelined processor with 5 instructions in flight?
- Today - Data hazards
  - What are they?
  - How do you detect them?
  - How do you deal with them?

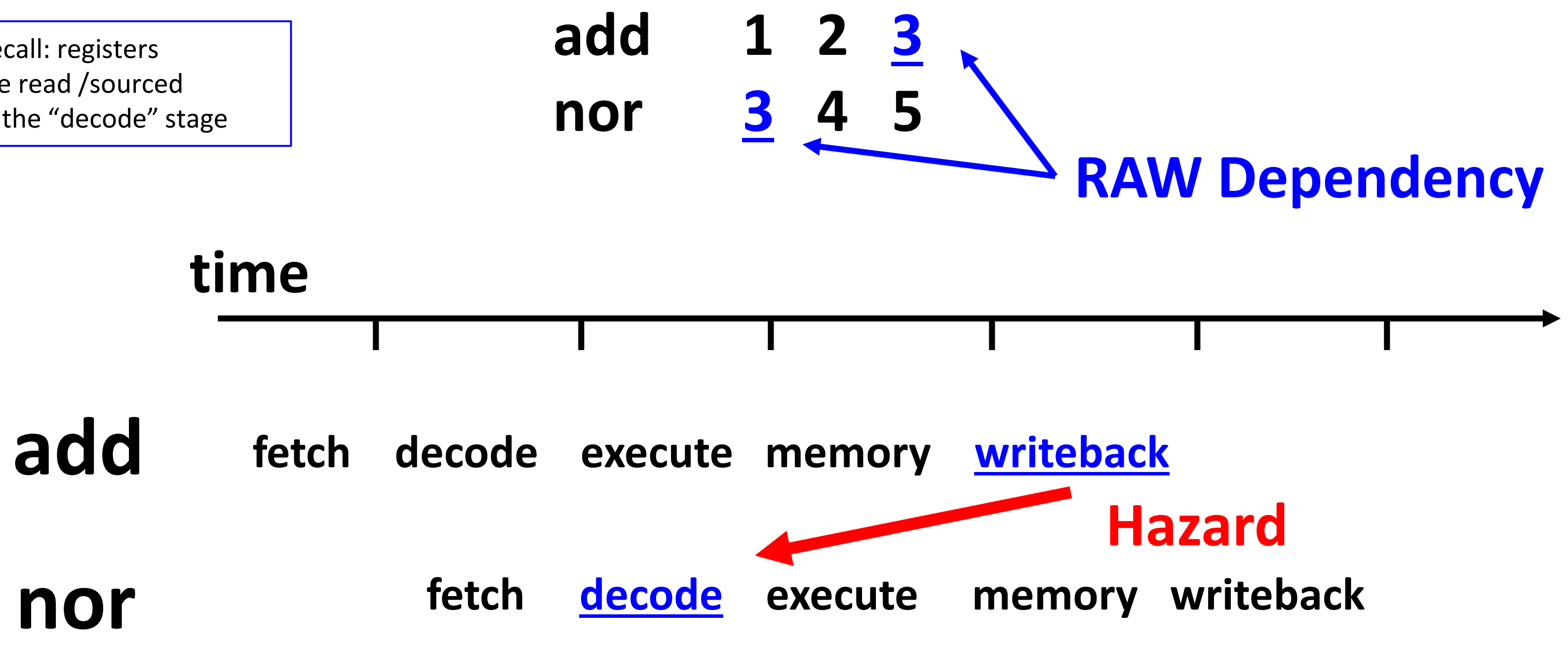


Pipeline function for ADD

- 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

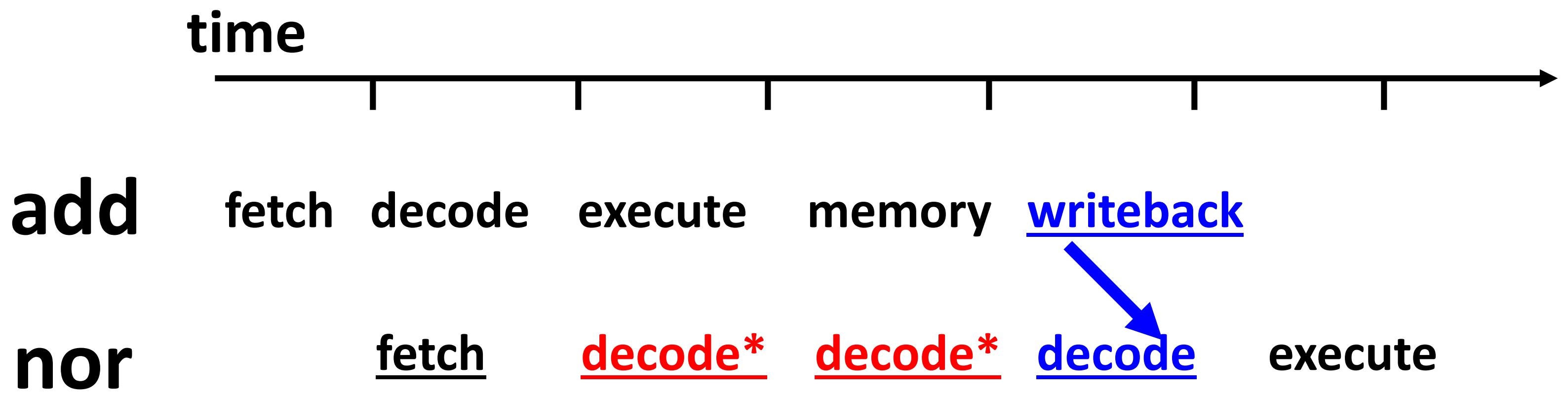


add 1 2 3
nor 3 4 5
RAW Dependency
time
add
fetch decode execute memory writeback
Hazard
nor
fetch decode execute memory writeback

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.

Class Problem

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

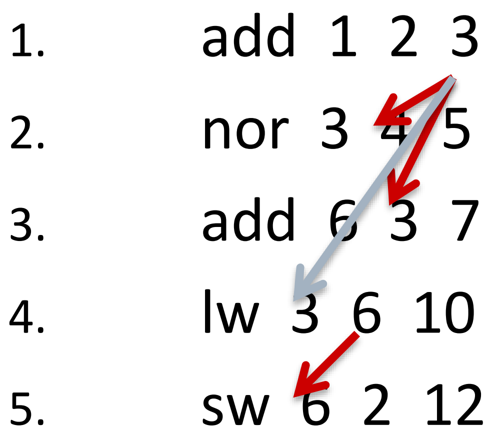
What about here?

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

Class Problem

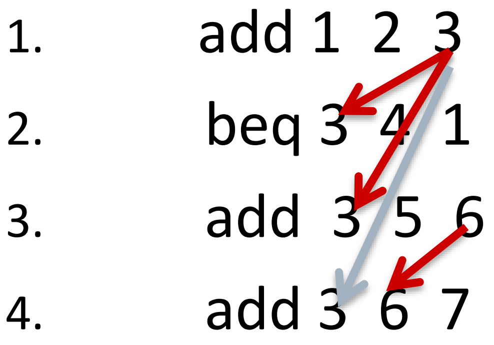
Which read-after-write (RAW) dependences do you see?

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   ←——   write register 3 in cycle 5
noop
noop
nor    3   4   5   ←——   read register 3 in cycle 5
```

Problems with this solution

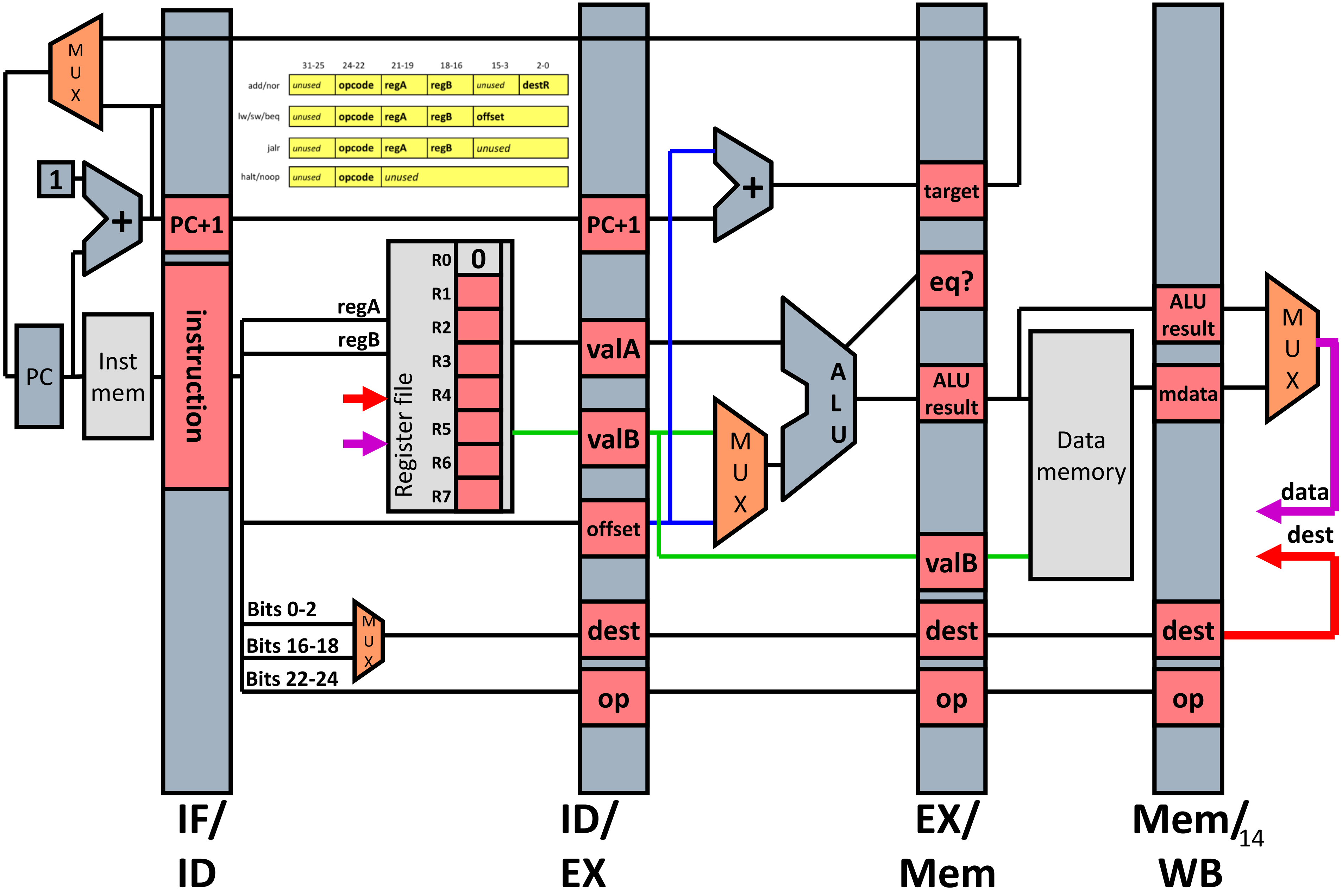
- Old programs (legacy code) may not run correctly on new implementations
  - Longer pipelines need more noops
- Programs get larger as noops are included
  - Especially a problem for machines that try to execute more than one instruction every cycle
  - Intel EPIC: Often 25% - 40% of instructions are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

Handling data hazards II: Detect and stall until ready

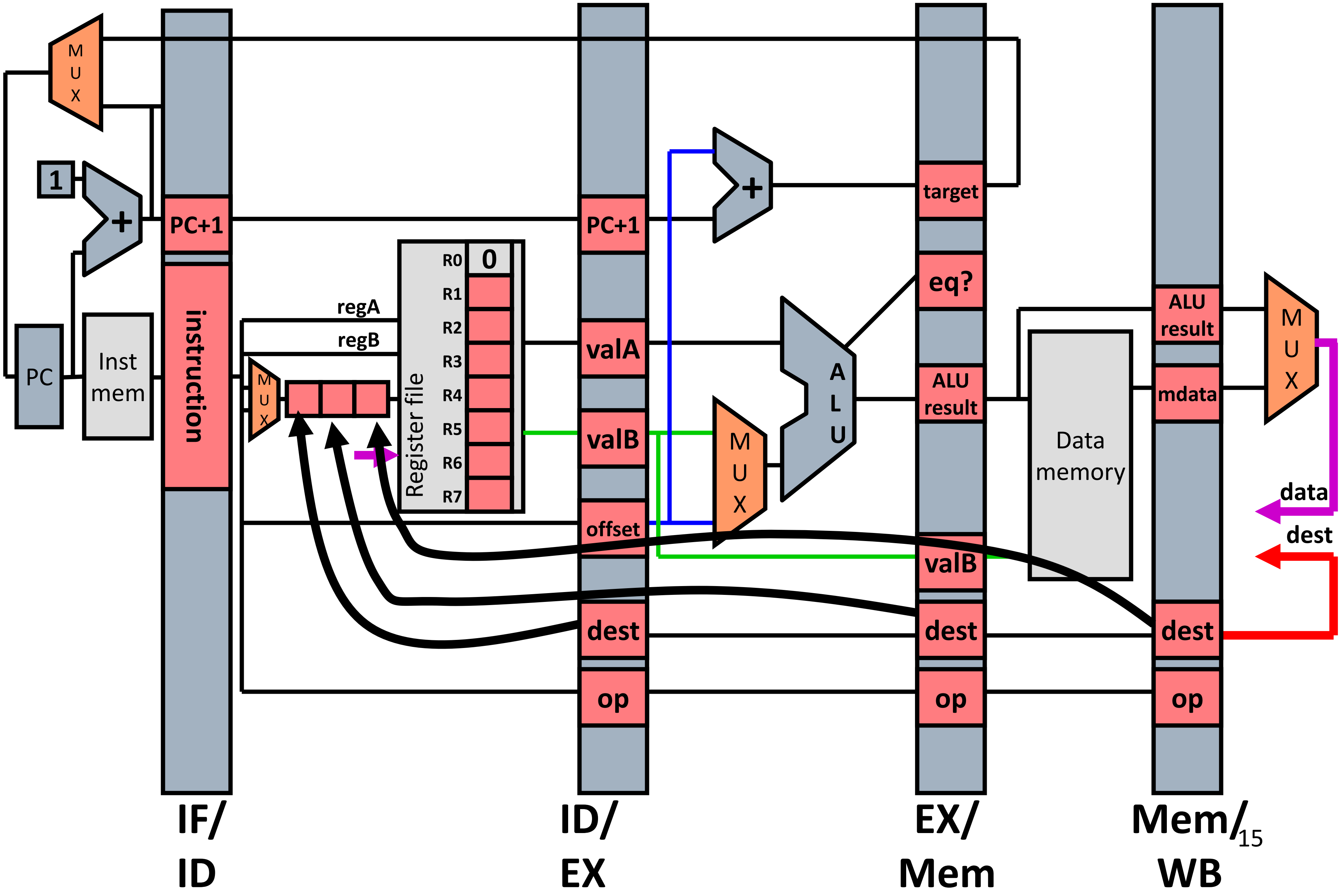
- Detect:
  - Compare regA with previous DestRegs
    - 3 bit operand fields
  - Compare regB with previous DestRegs
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute

- How do we modify the pipeline to do this?

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



	31-25	24-22	21-19	18-16	15-3	2-0
add/nor	*unused*	opcode	regA	regB	*unused*	destR
lw/sw/beq	*unused*	opcode	regA	regB	offset	
jalr	*unused*	opcode	regA	regB	*unused*	
halt/noop	*unused*	opcode	*unused*			



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

Example

- Let's run this program with a data hazard through our 5-stage pipeline

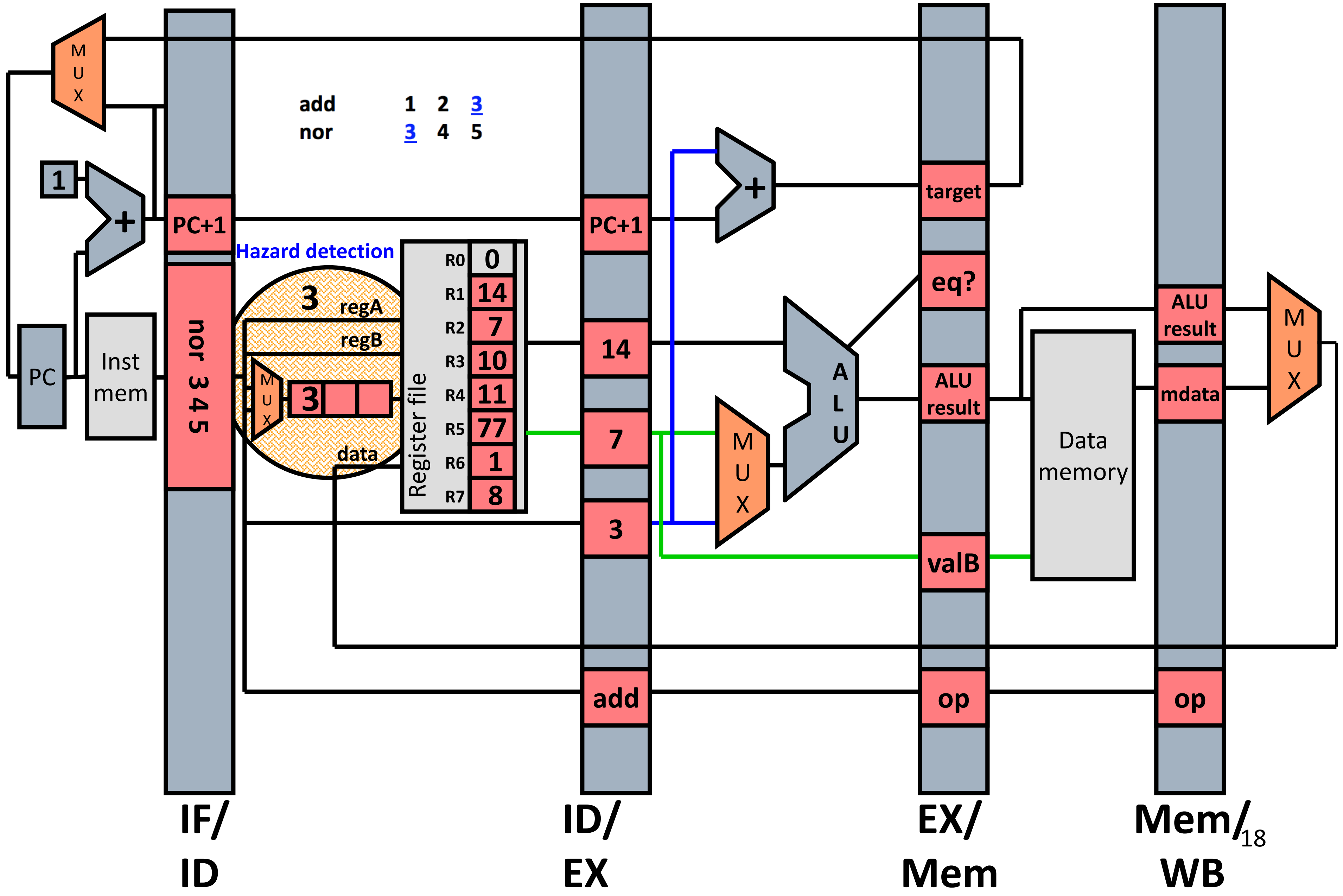
```
add    1  2  3
nor    3  4  5
```

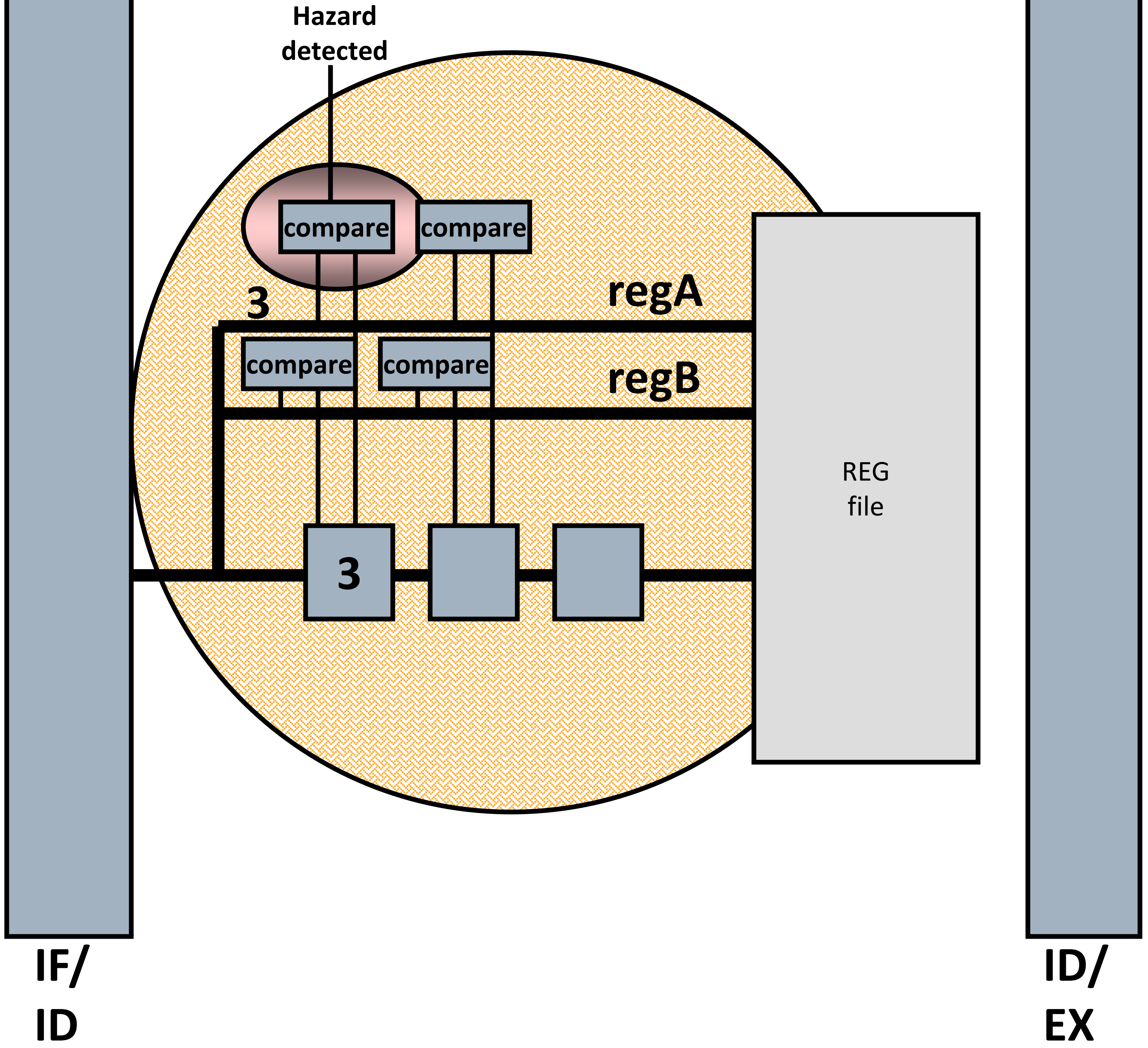
- We will start at the beginning of cycle 3, where add is in the EX stage, and nand is in the ID stage, about to read a register value

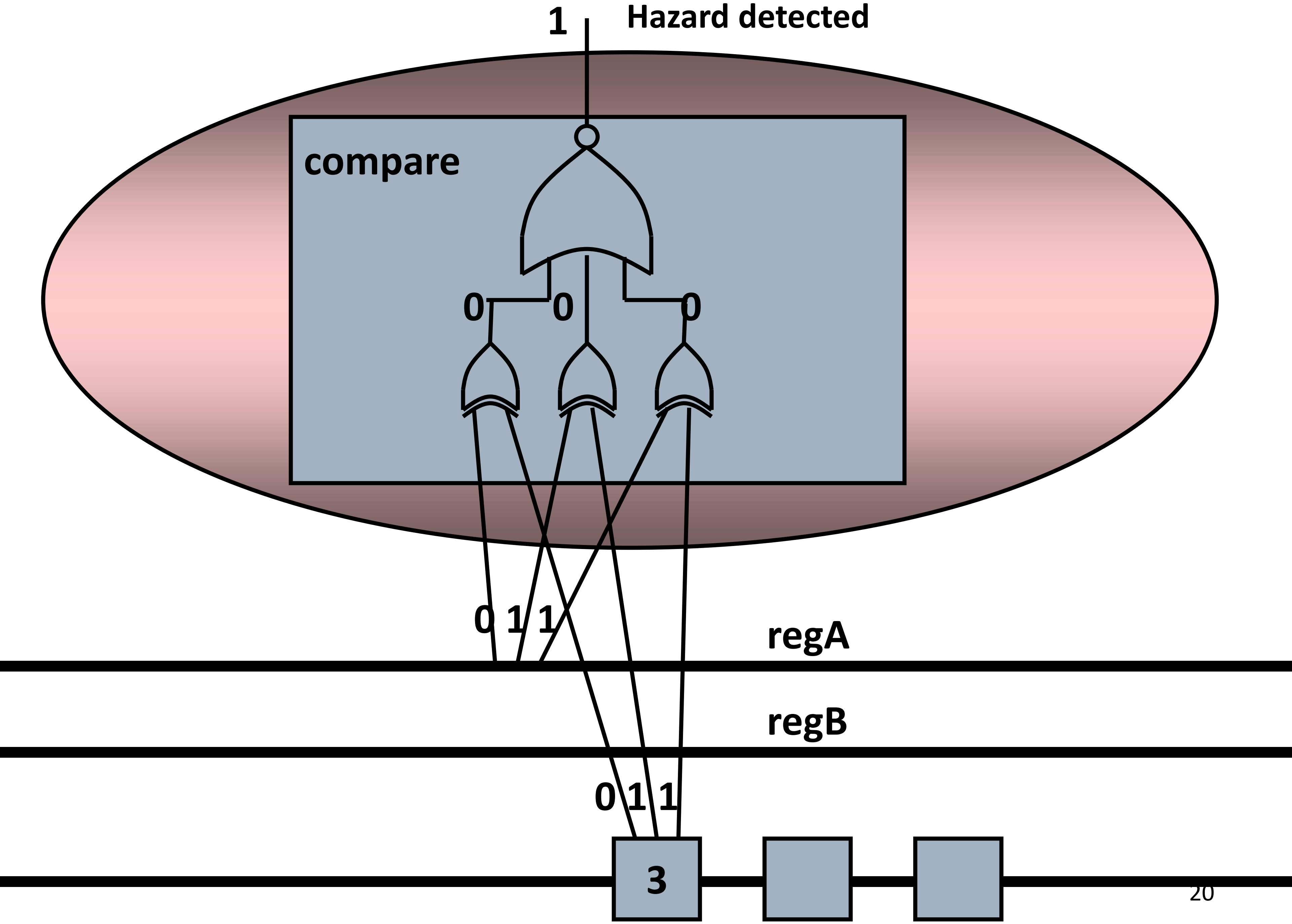
Time:	1	2	3
add 1 2 3	IF	ID	EX
nor 3 4 5		IF	ID

Hazard!

First half of cycle 3




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

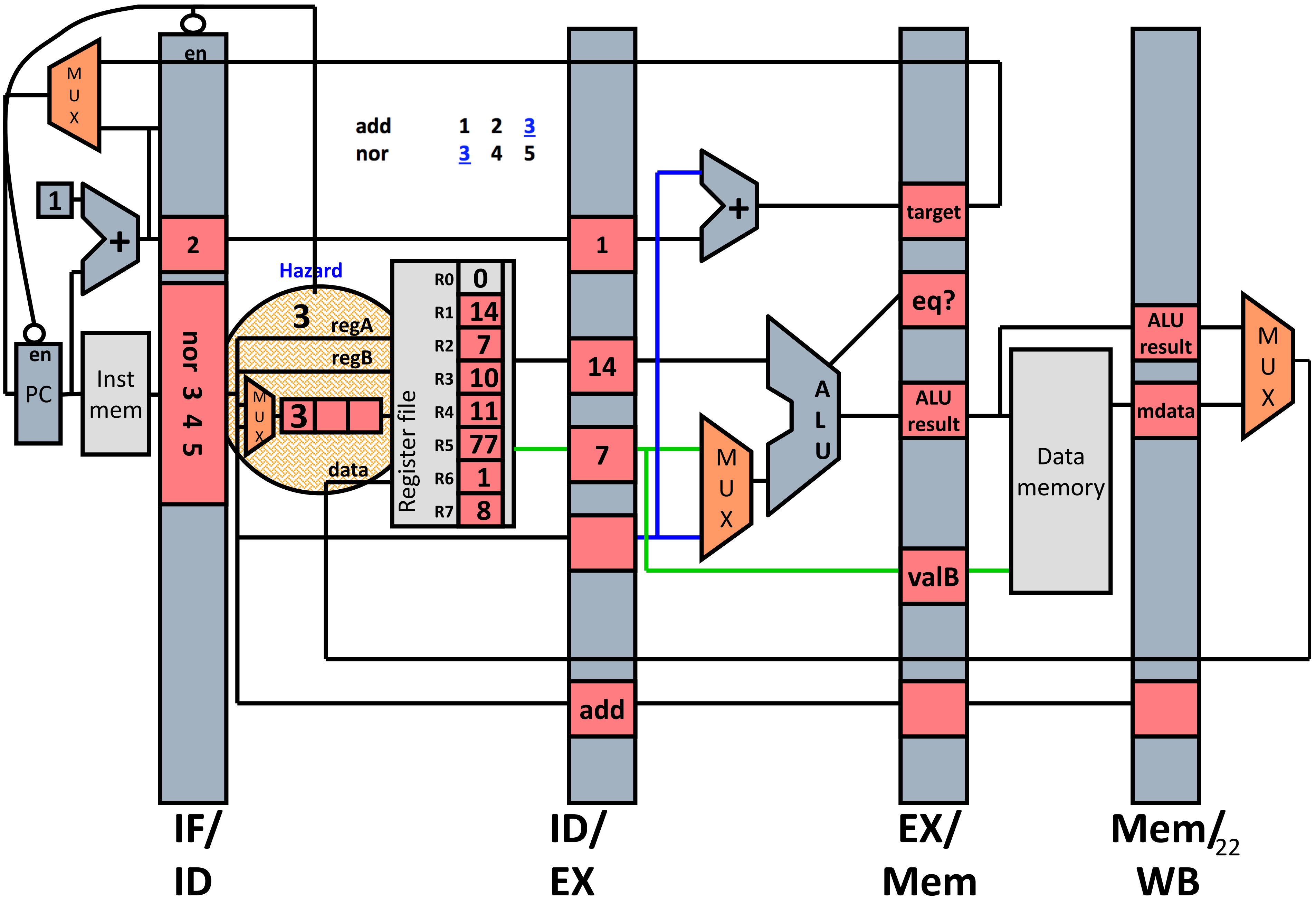
Handling data hazards II: Detect and stall until ready

- Detect:
  - Compare regA with previous DestReg
    - 3 bit operand fields
  - Compare regB with previous DestReg
    - 3 bit operand fields
- Stall:

  Keep current instructions in fetch and decode

  Pass a noop to execute

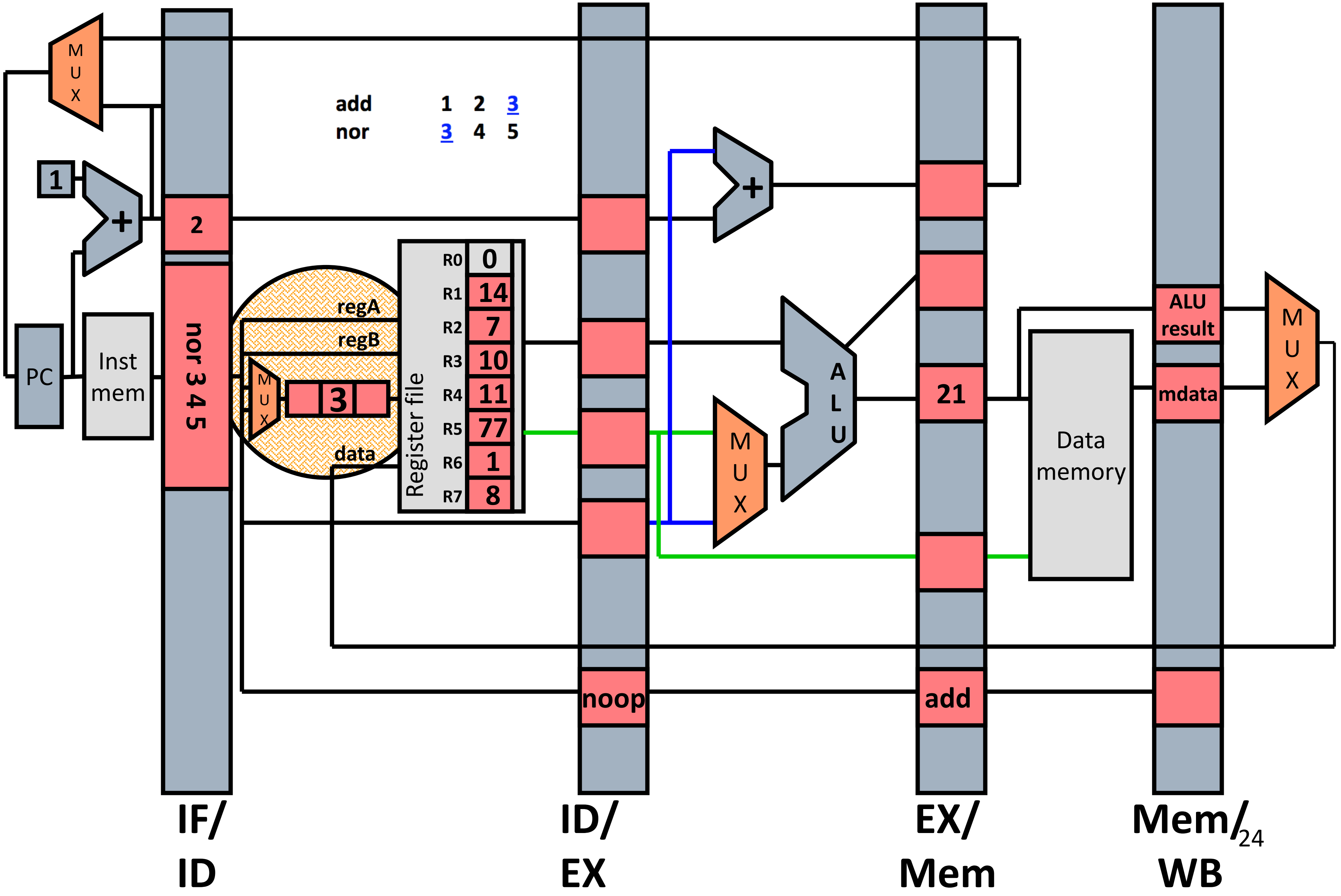
First half of cycle 3



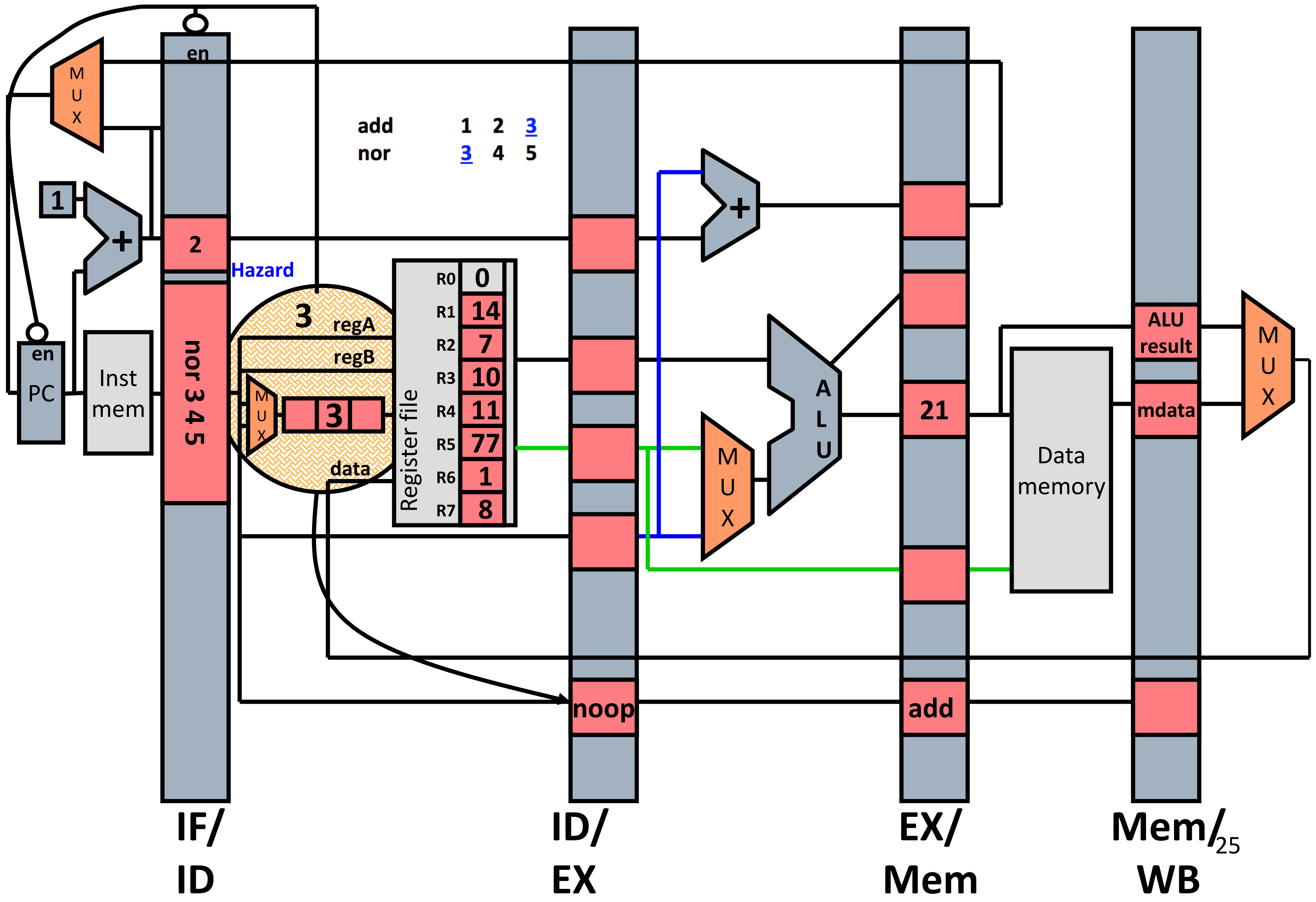
Handling data hazards II: Detect and stall until ready

- Detect:
  - Compare regA with previous DestReg
    - 3 bit operand fields
  - Compare regB with previous DestReg
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - **<u>Pass a noop to execute</u>**

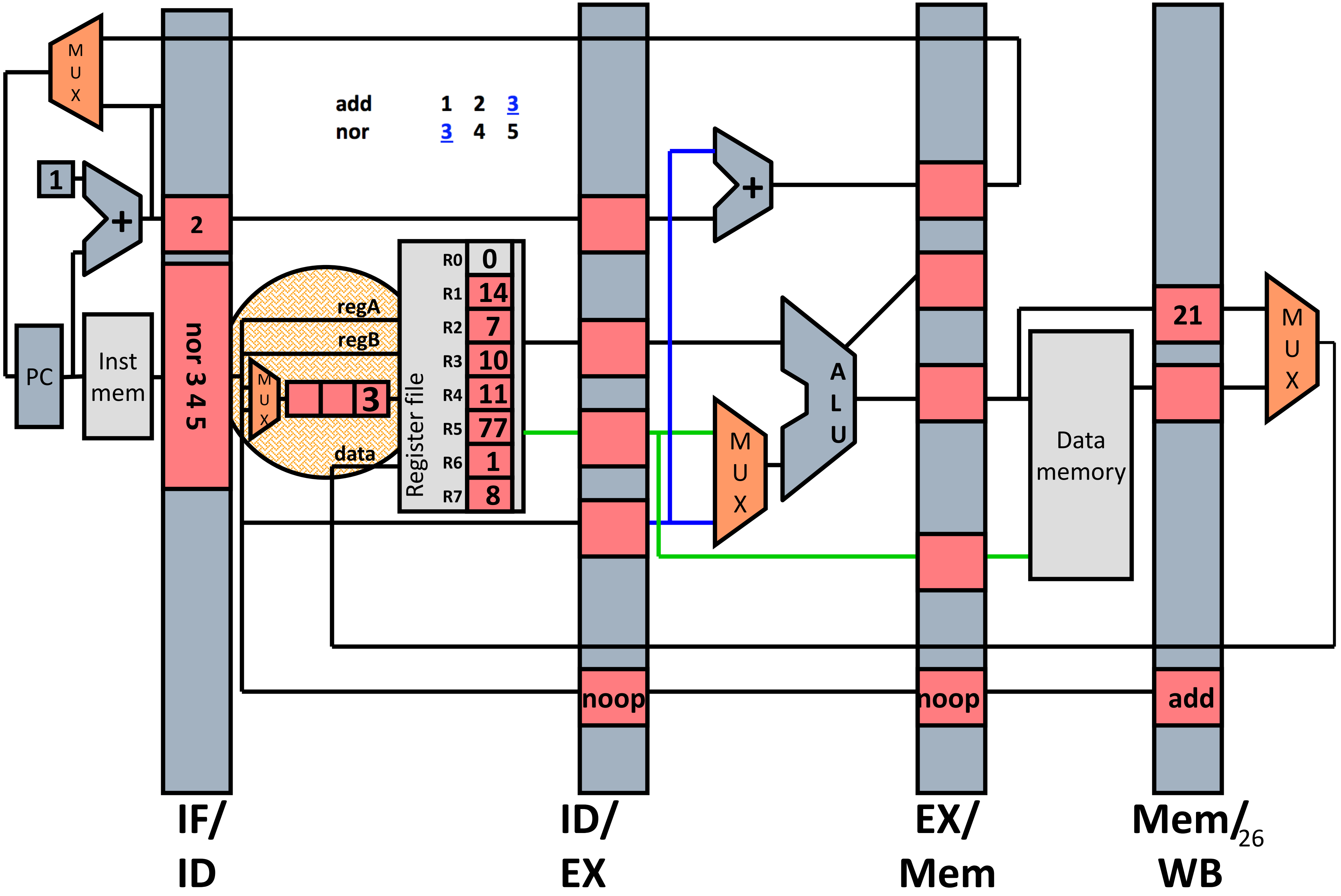
End of cycle 3



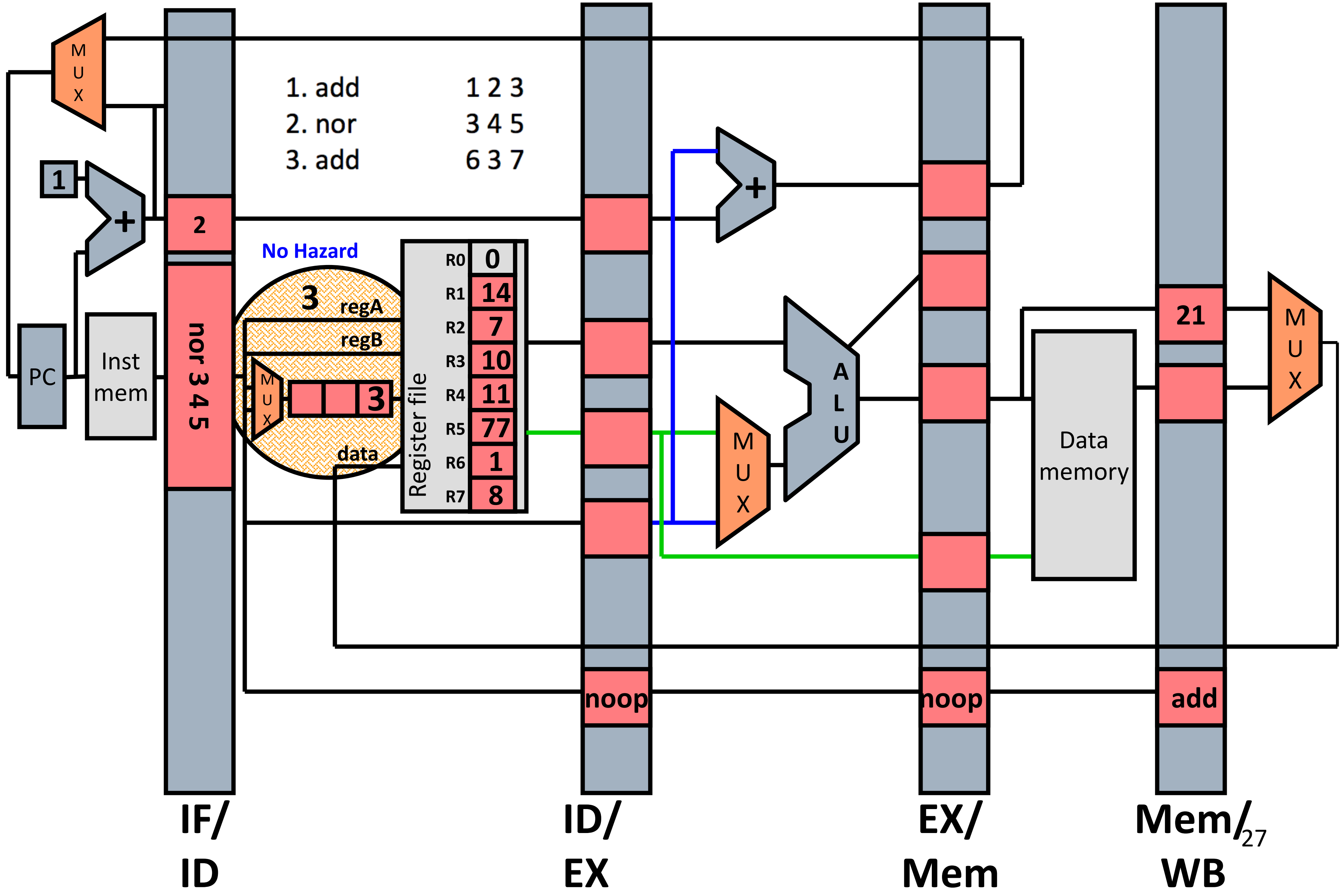
First half of cycle 4



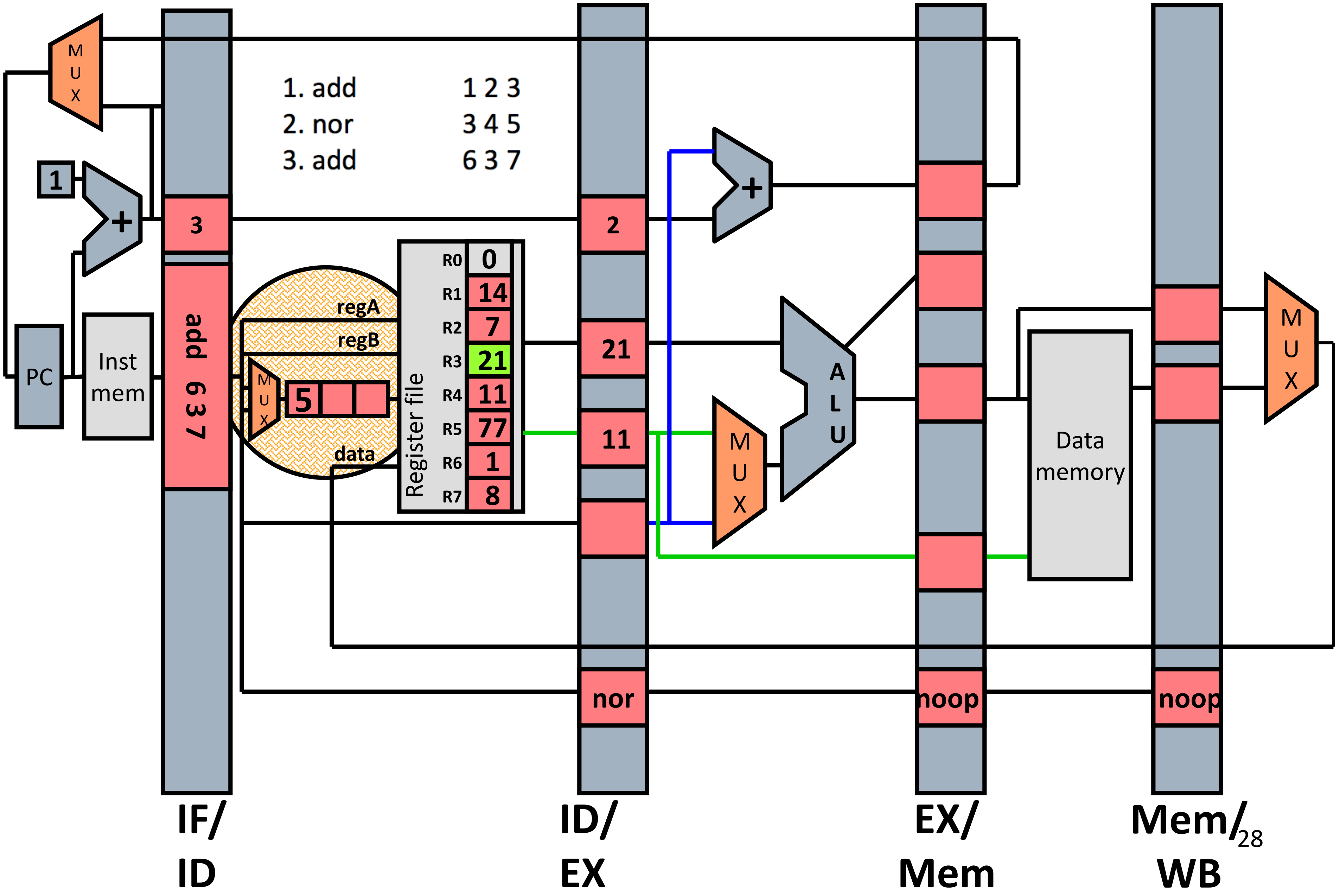
End of cycle 4



First half of cycle 5



End of cycle 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					

Exercise

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													

1. Identify the data hazards in this extended program
2. Complete the time graph

Solution

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

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													
sw 6 2 12													

Solution

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													

Solution

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

Problems with detect and stall

- CPI increases every time a hazard is detected!
- Is that necessary? Not always!
  - Re-route the result of the add to the nor
    - nor no longer needs to read R3 from reg file
    - It can get the data later (when it is ready)
    - This lets us complete the decode this cycle
      - But we need more control to remember that the data that we aren't getting from the reg file at this time will be found elsewhere in the pipeline at a later cycle.

Handling data hazards III: Detect and forward

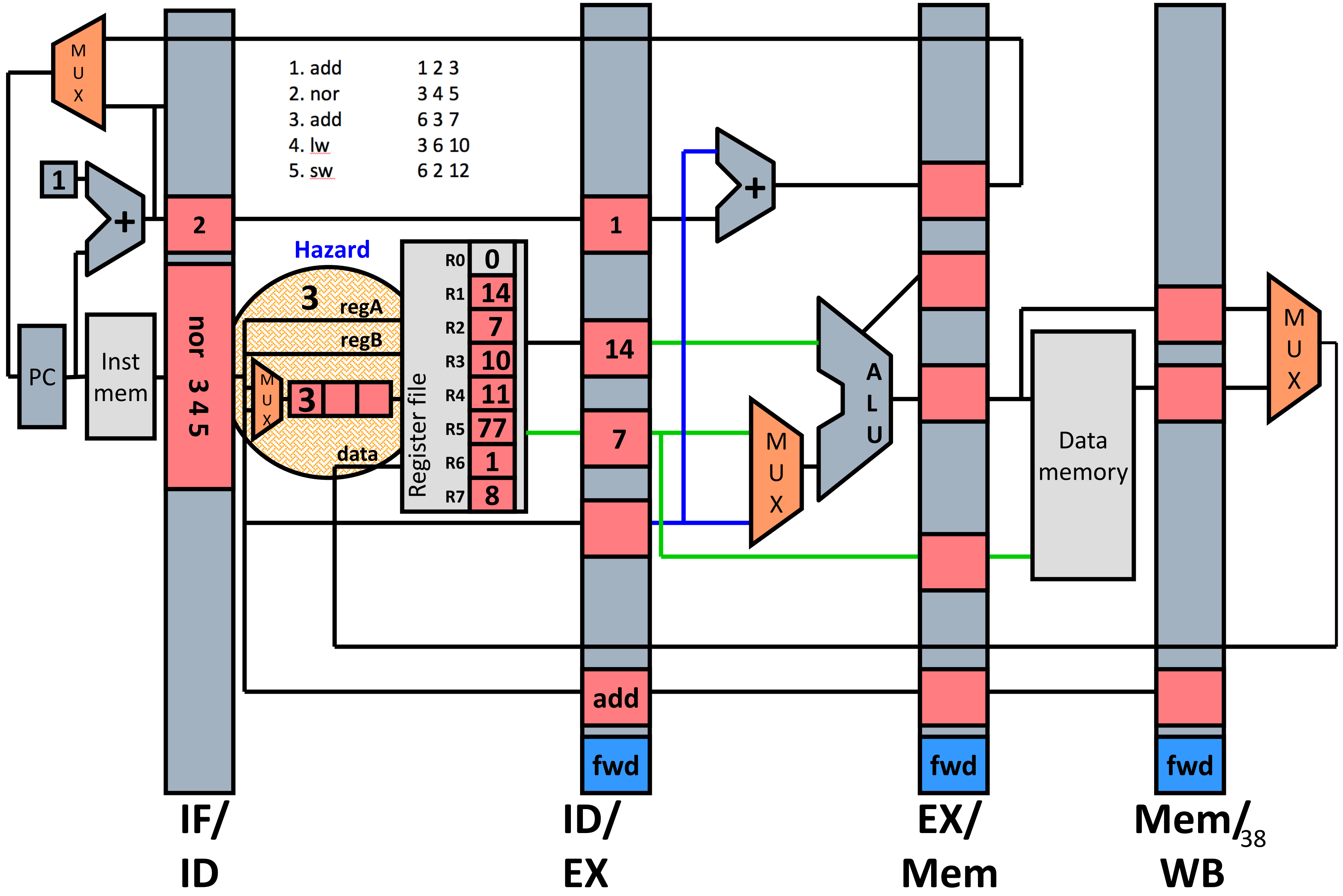
- Detect: same as detect and stall
  - Except that all 4 hazards have to be treated differently
    - i.e., you can’t logical-OR the 4 hazard signals
- Forward:
  - New bypass datapaths route computed data to where it is needed
  - New MUX and control to pick the right data
- Beware: Stalling may still be required even in the presence of forwarding

Forwarding example

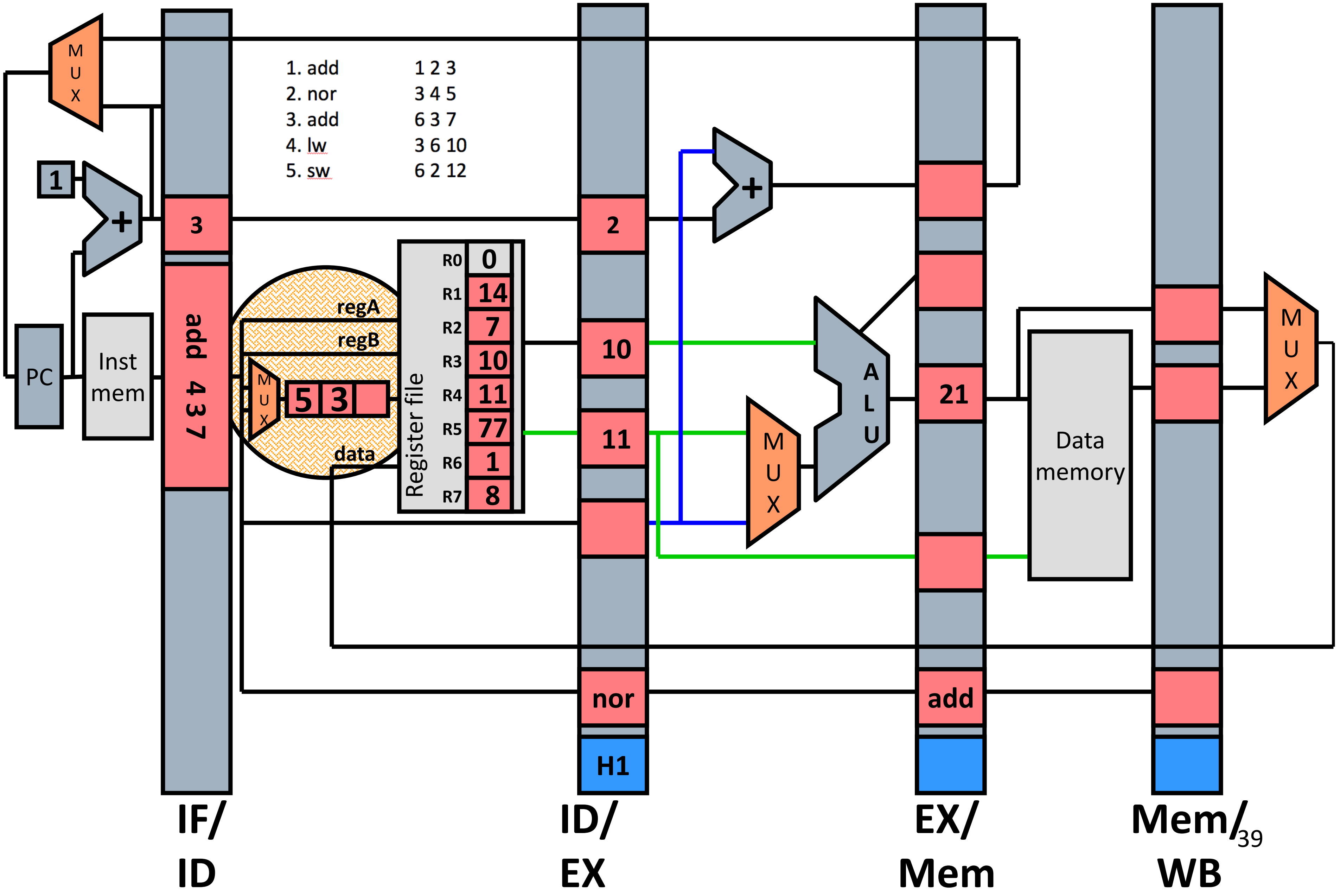
- We will use this program for the next example (same as last pipeline diagram example)

```
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
```

First half of cycle 3

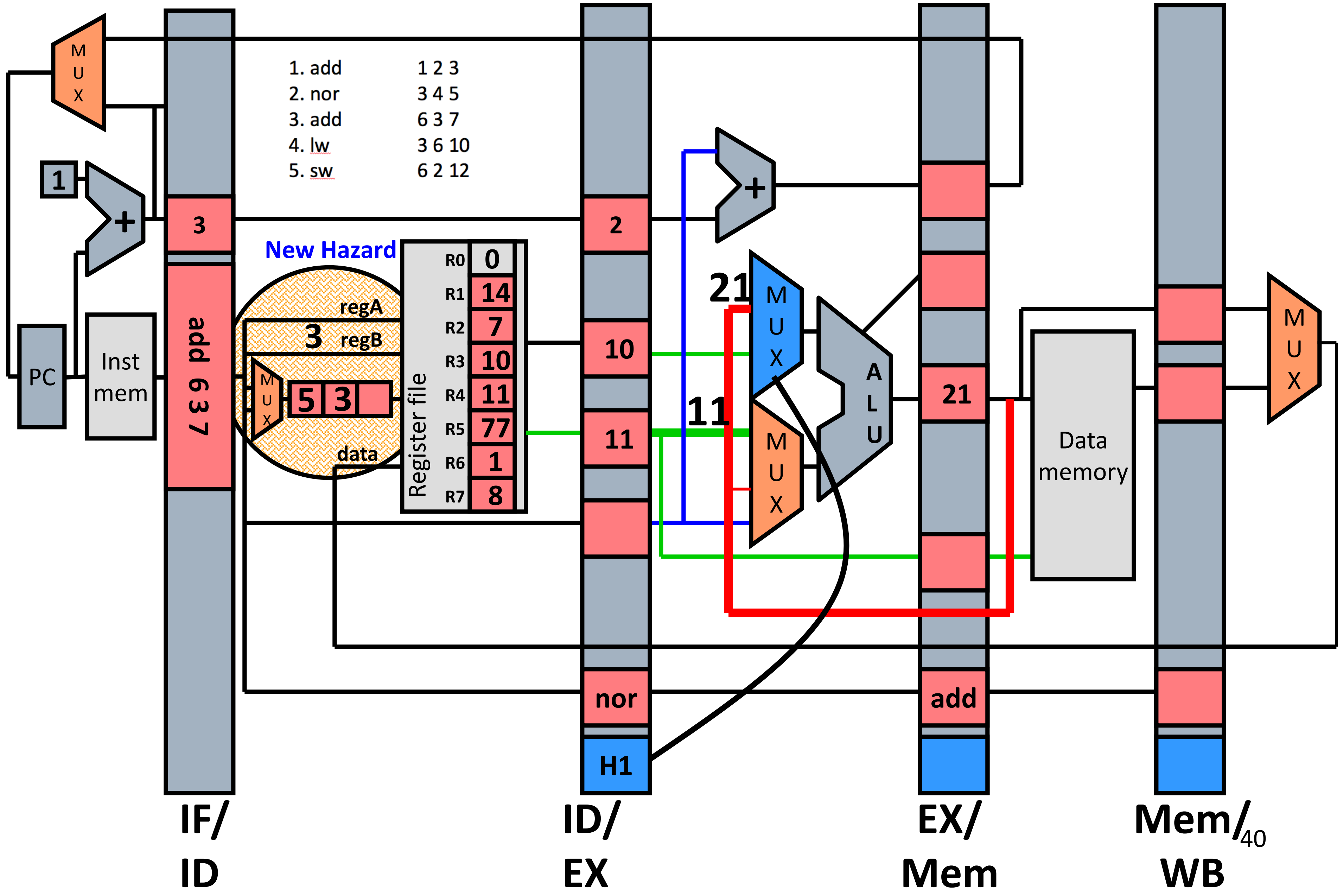


End of cycle 3

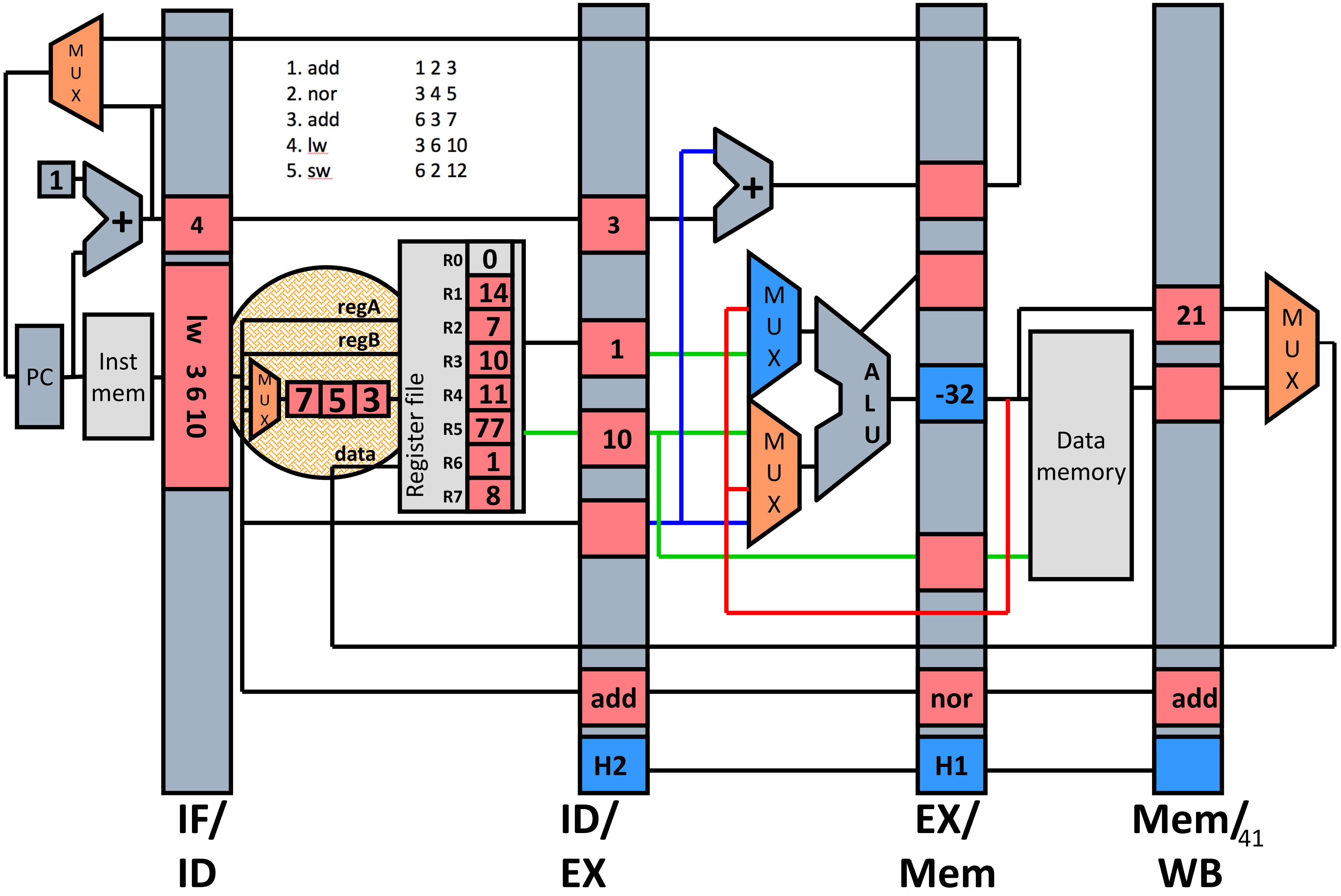


1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12

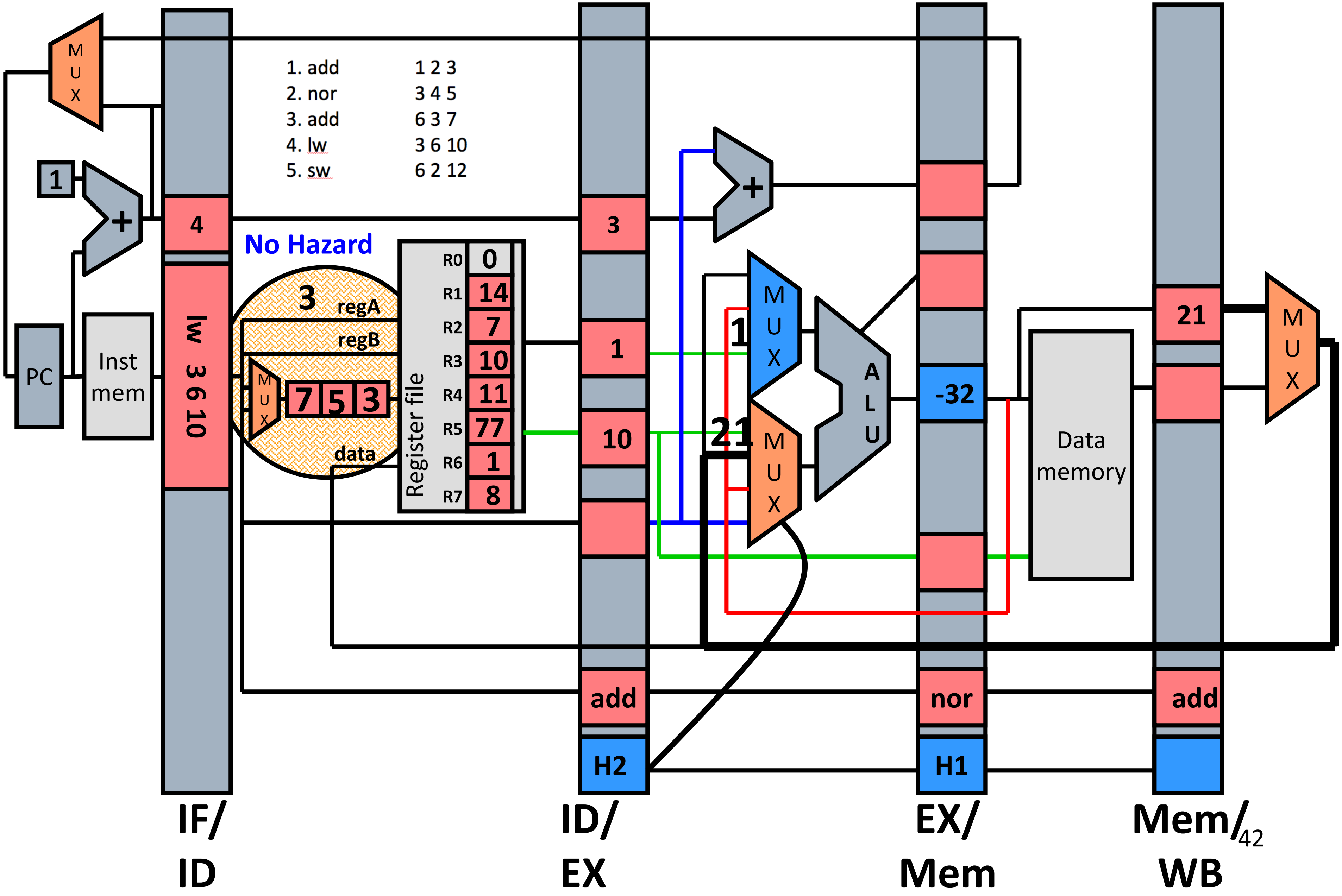
First half of cycle 4



End of cycle 4



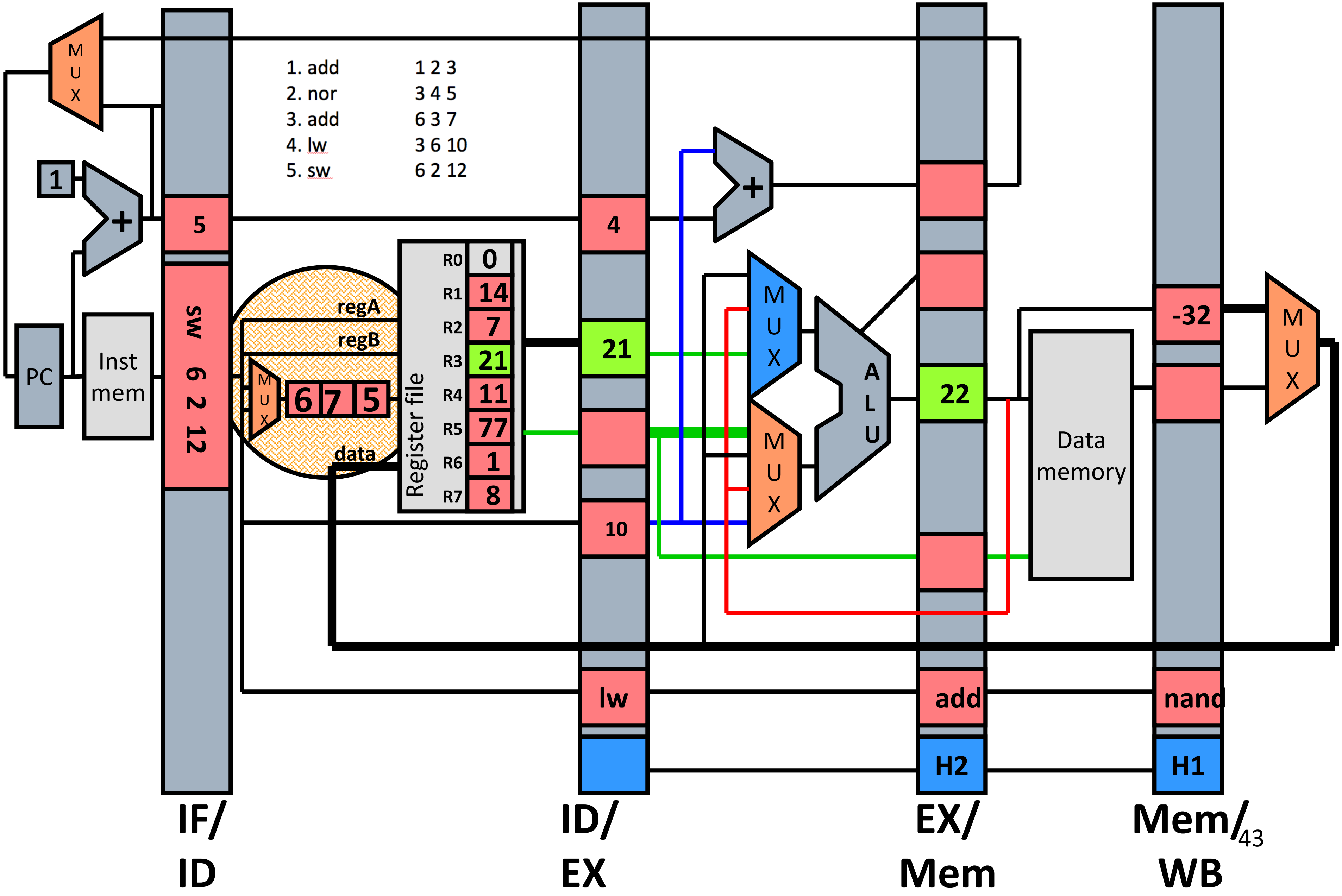
First half of cycle 5



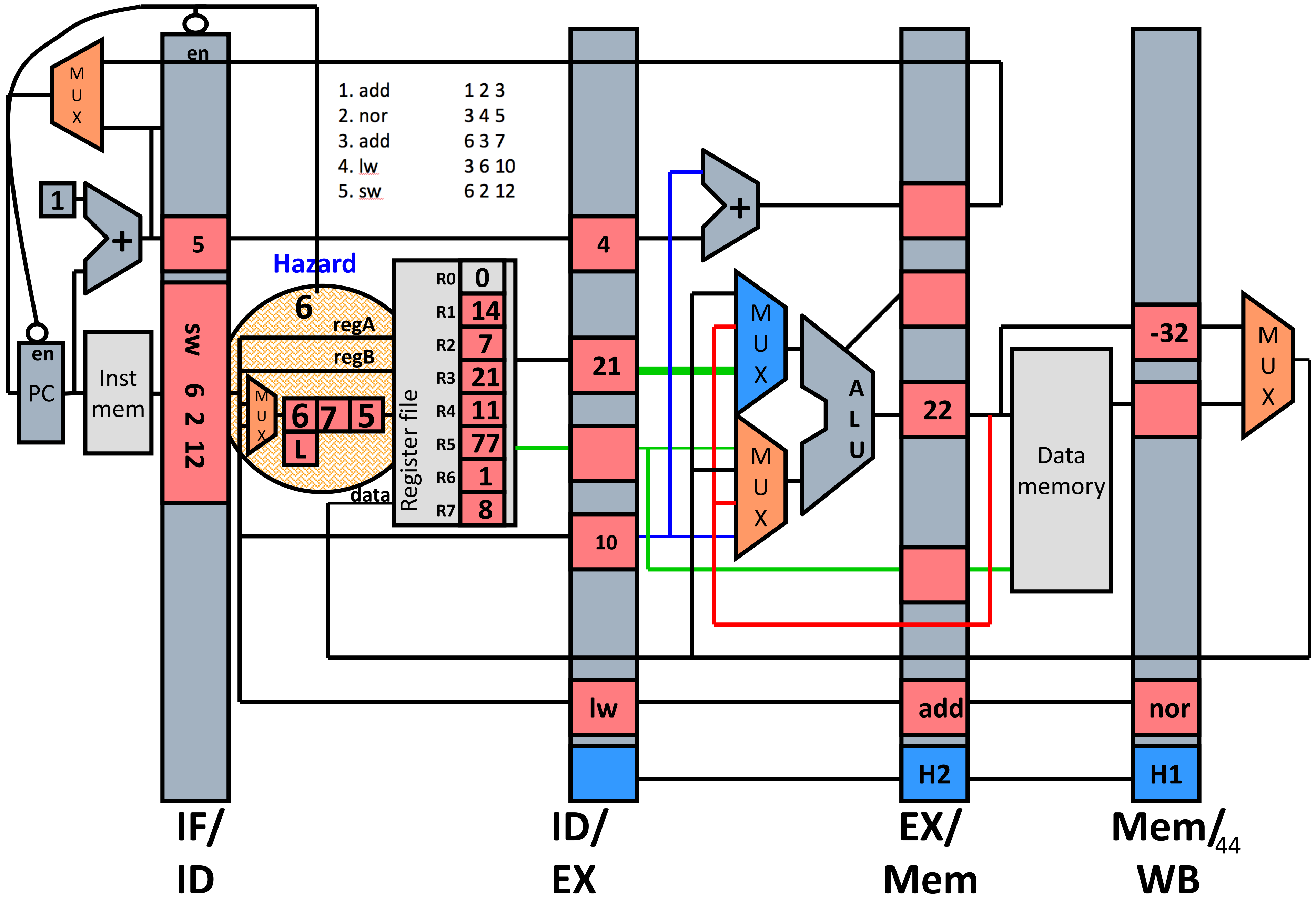
1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12

End of cycle 5

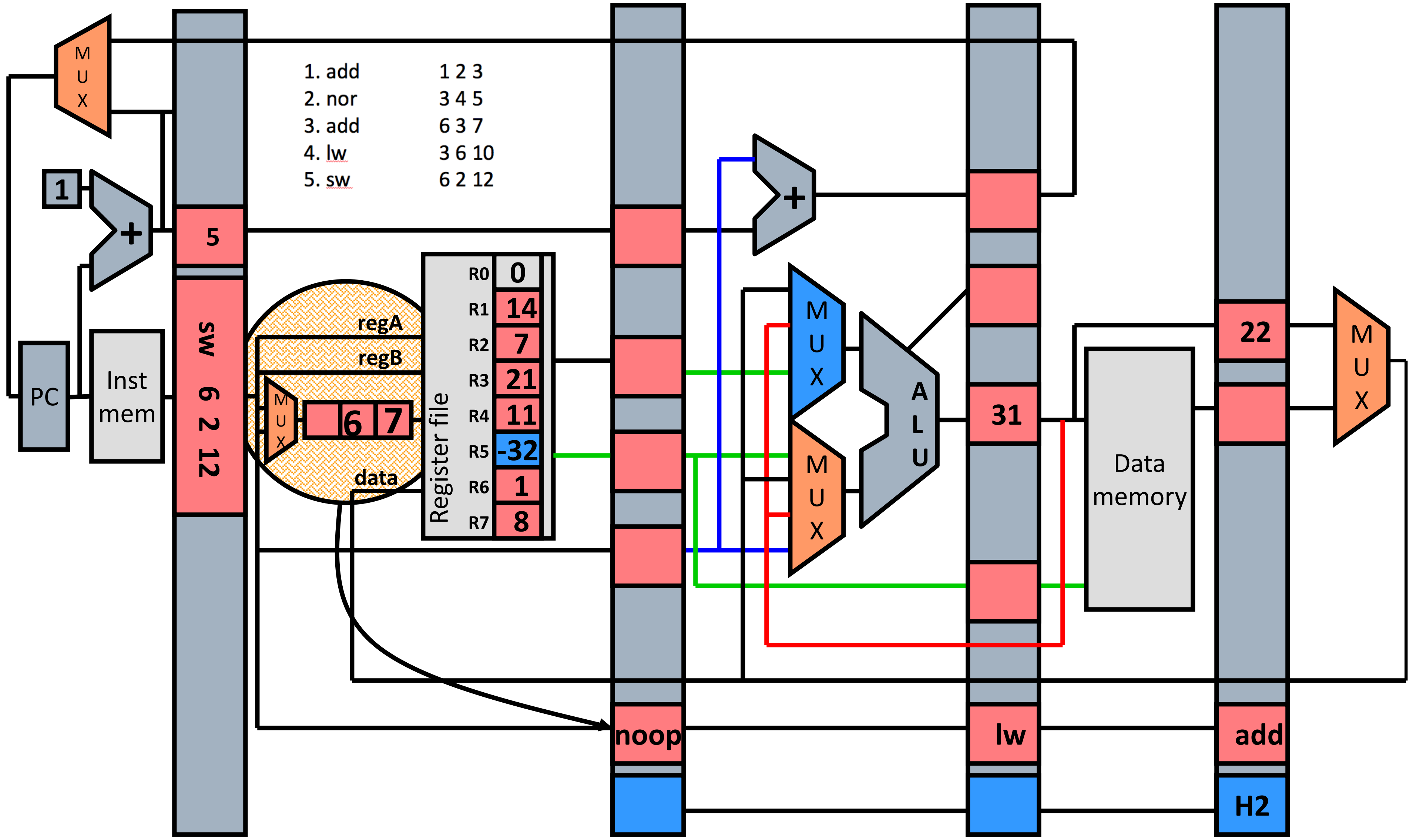
1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12



First half of cycle 6



End of cycle 6



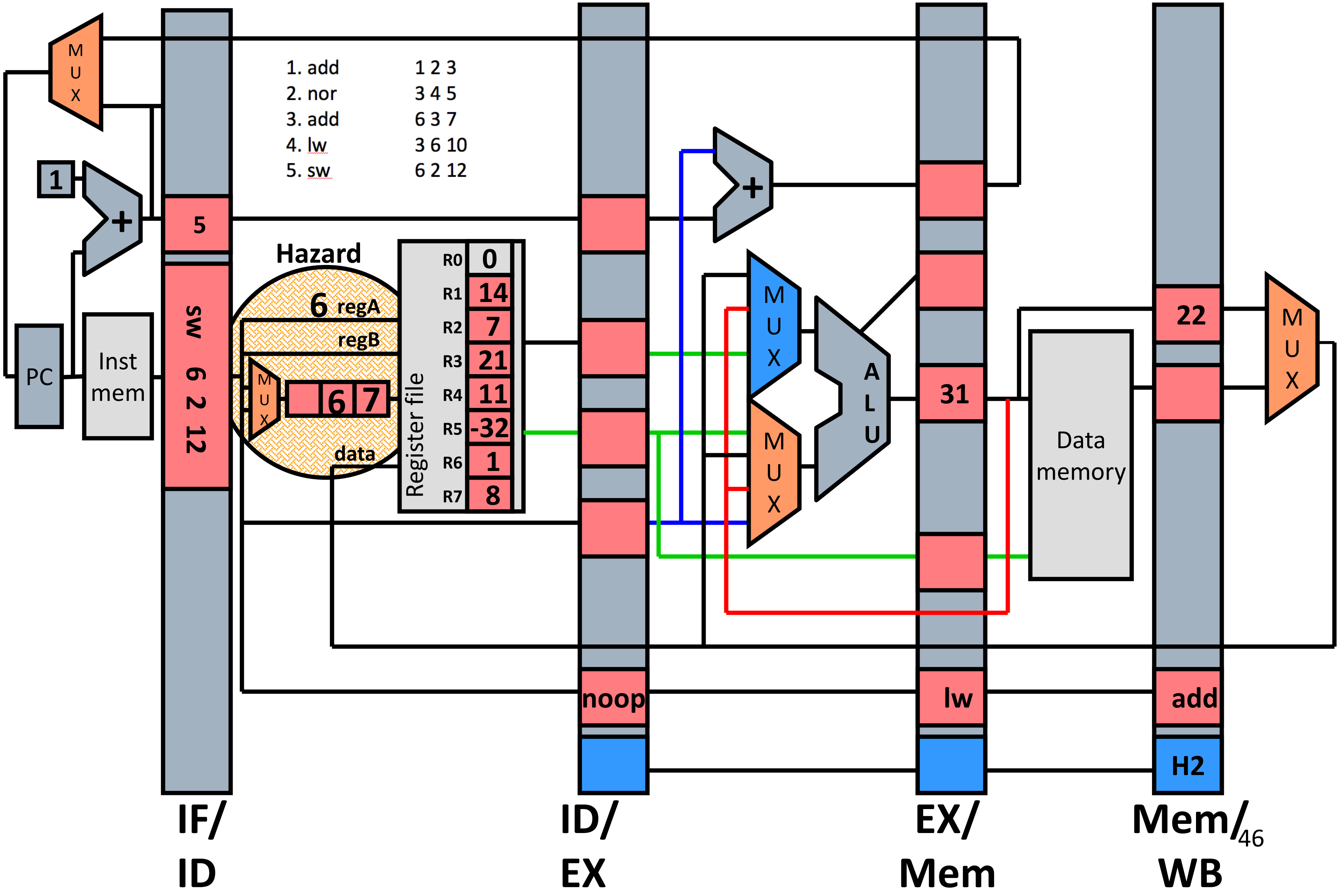
IF/ ID

ID/ EX

EX/ Mem

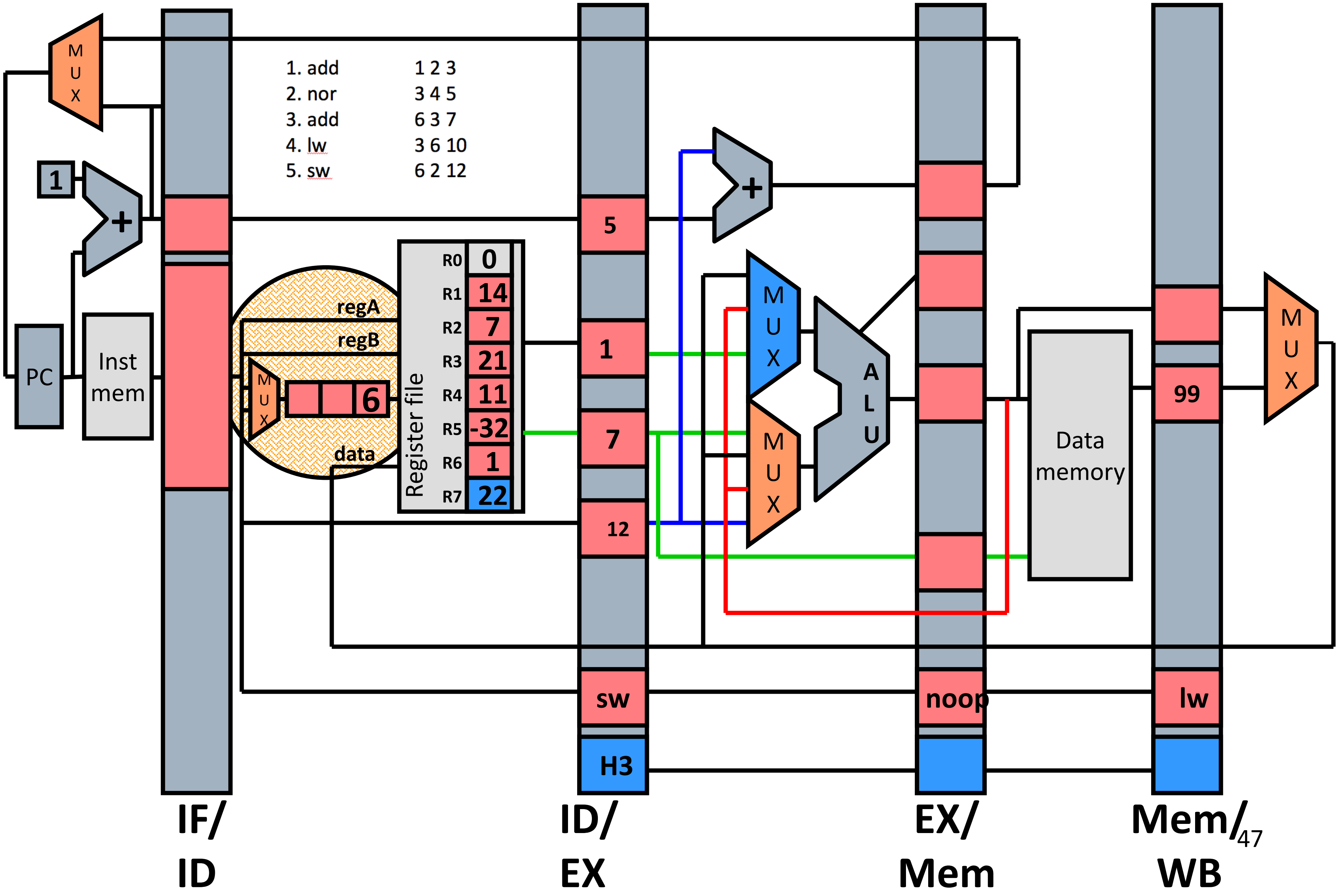
Mem/ WB

First half of cycle 7

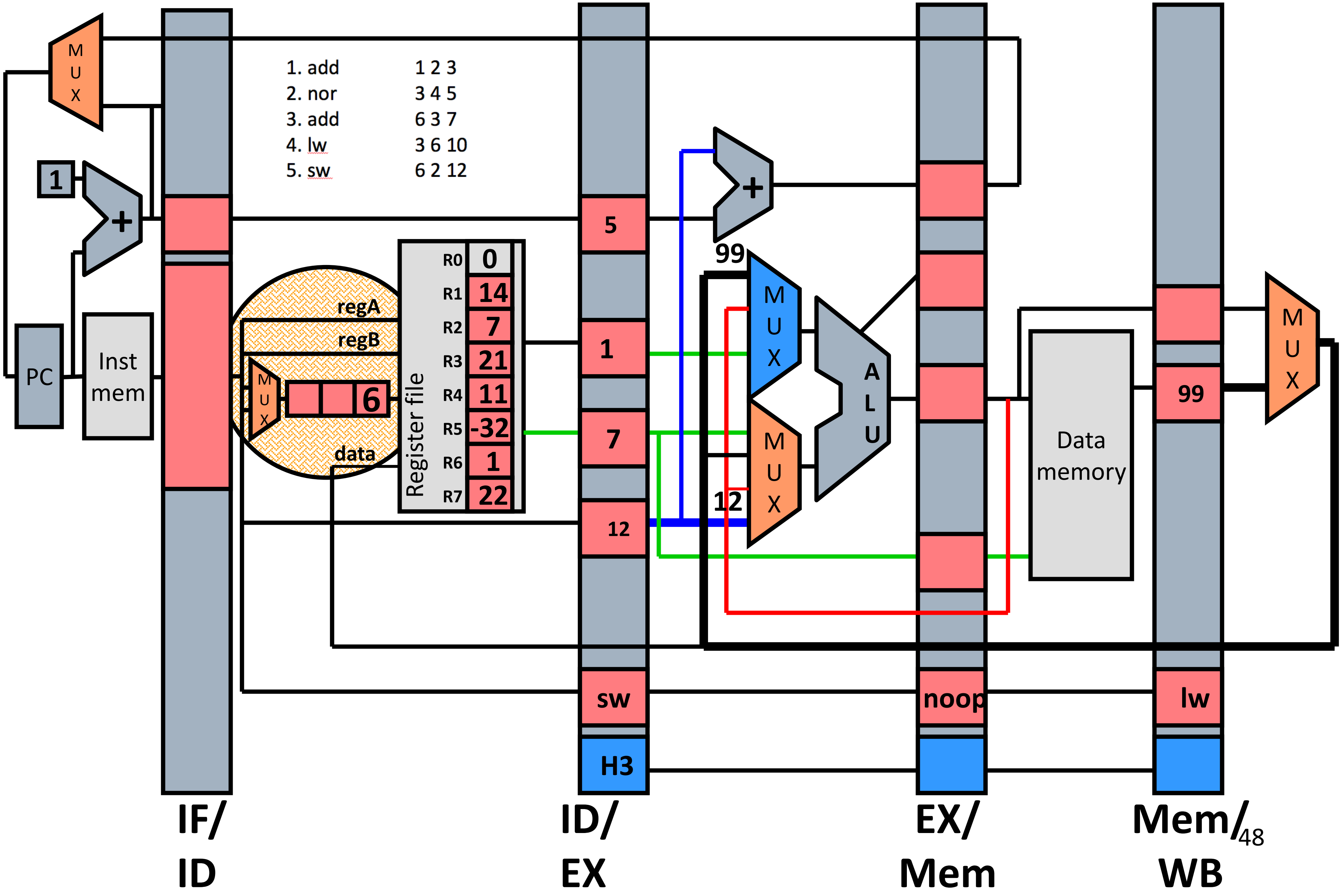


1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12

End of cycle 7

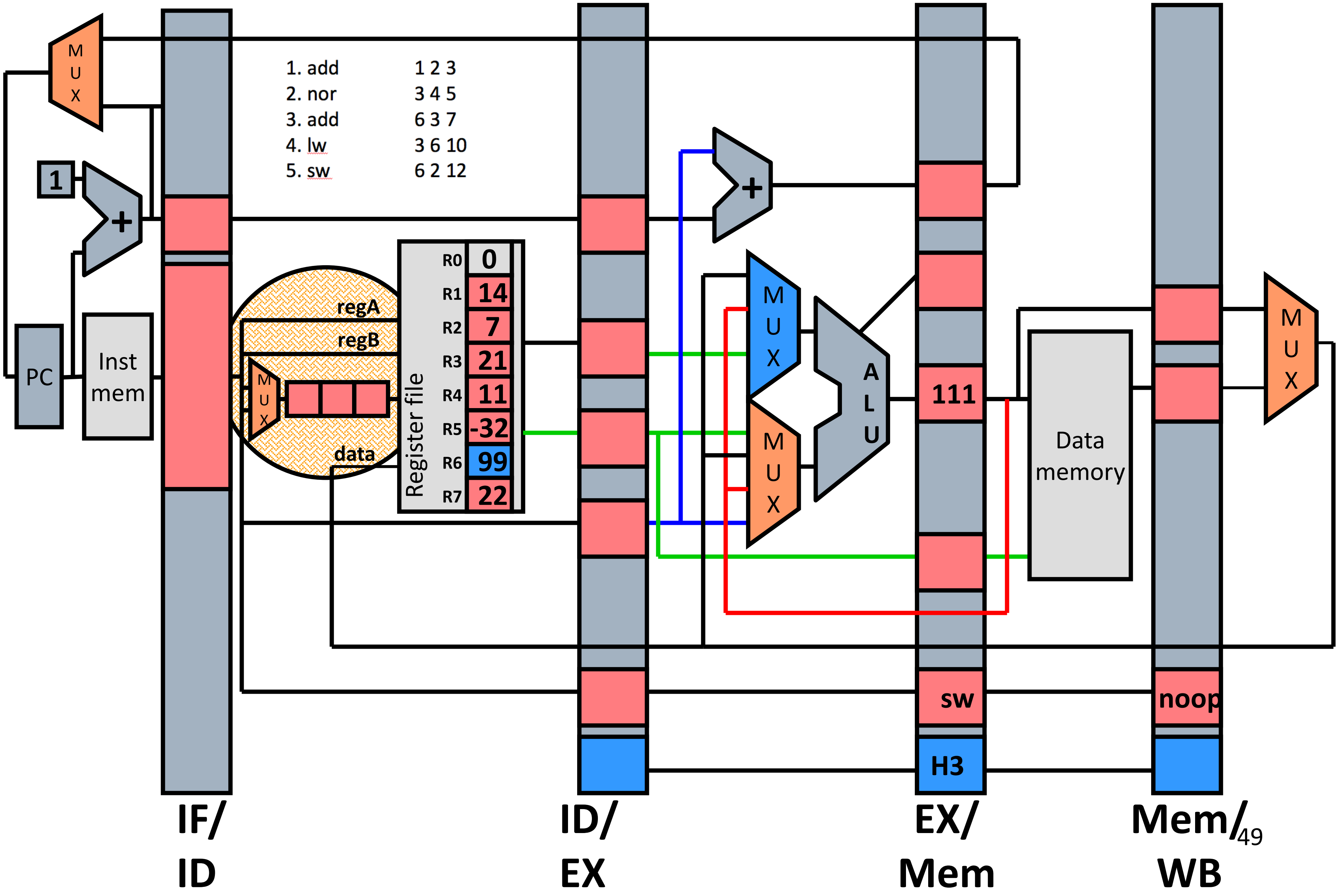


First half of cycle 8



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

End of cycle 8



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	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	EX	ME	WB			

Review: 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 old/stale values if is about to be written.
- Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- Exceptions: How do you handle exceptions in a pipelined processor with 5 instructions in flight?
- Next Time – Control Hazards:
  - What are they?
  - How do you detect them?
  - How do you deal with them?