EECS 370 - Lecture 14

Pipelining and Data Hazards II

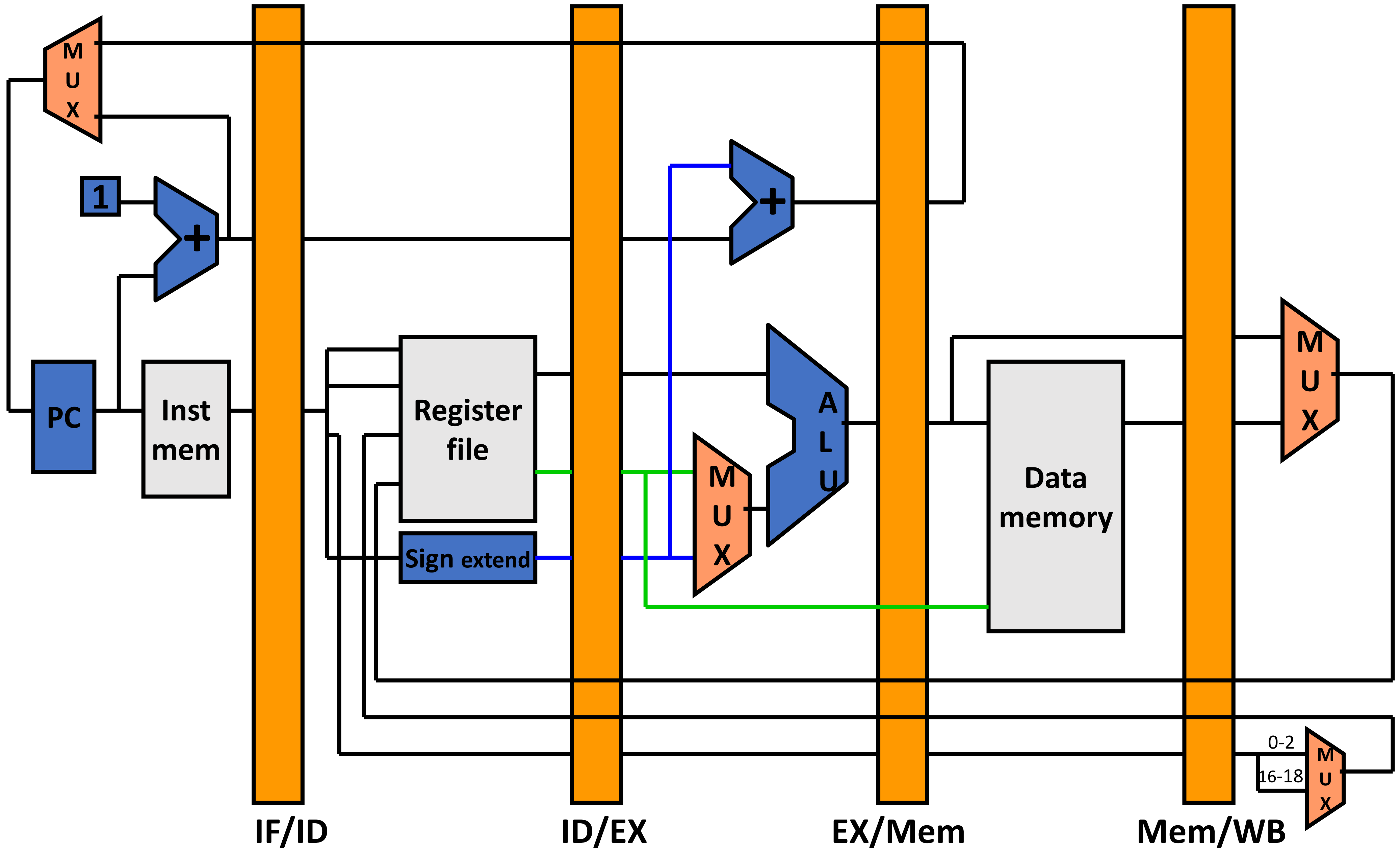
Announcements

- P2L due Thursday
- Midterm Thursday we come back
- Tuesday will be review session
- No discussion this Friday

Review: Pipelining

- Goal:
  - Achieve low clock period of multi-cycle processor…
  - … while maintaining low cycles-per-instruction (CPI) of single cycle processor (close to 1)
  - Achieve this by overlapping execution of multiple instructions simultaneously

Review: New Datapath



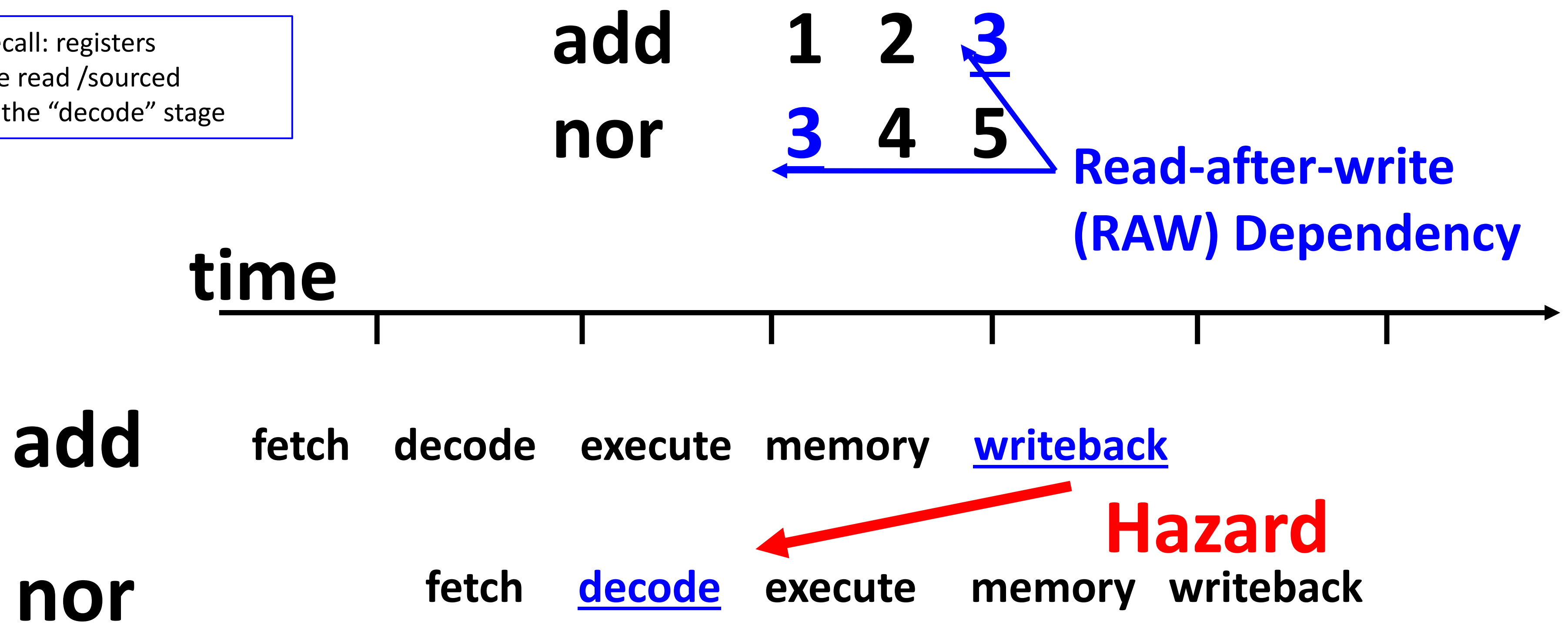
Review: Sample Code (Simple)

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

Data Hazards

Recall: registers
are read /sourced
In the "decode" stage



add 1 2 3
nor 3 4 5
Read-after-write
(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

Three approaches to handling data hazards

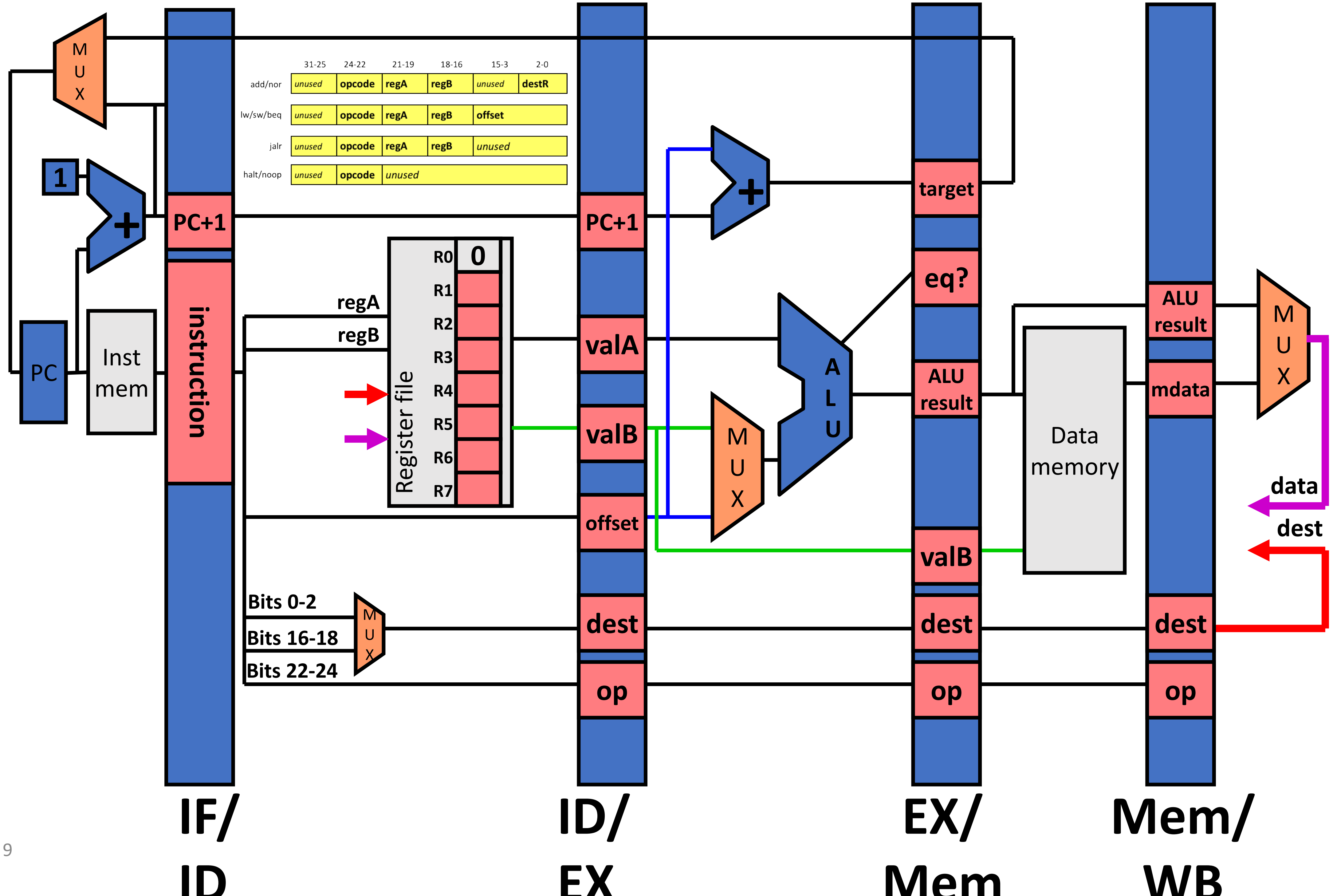
- Avoid
  - Make sure there are no hazards in the code
- Detect and Stall
  - If hazards exist, stall the processor until they go away.
- Detect and Forward
  - If hazards exist, fix up the pipeline to get the correct value (if possible)

Handling data hazards II: Detect and stall until ready

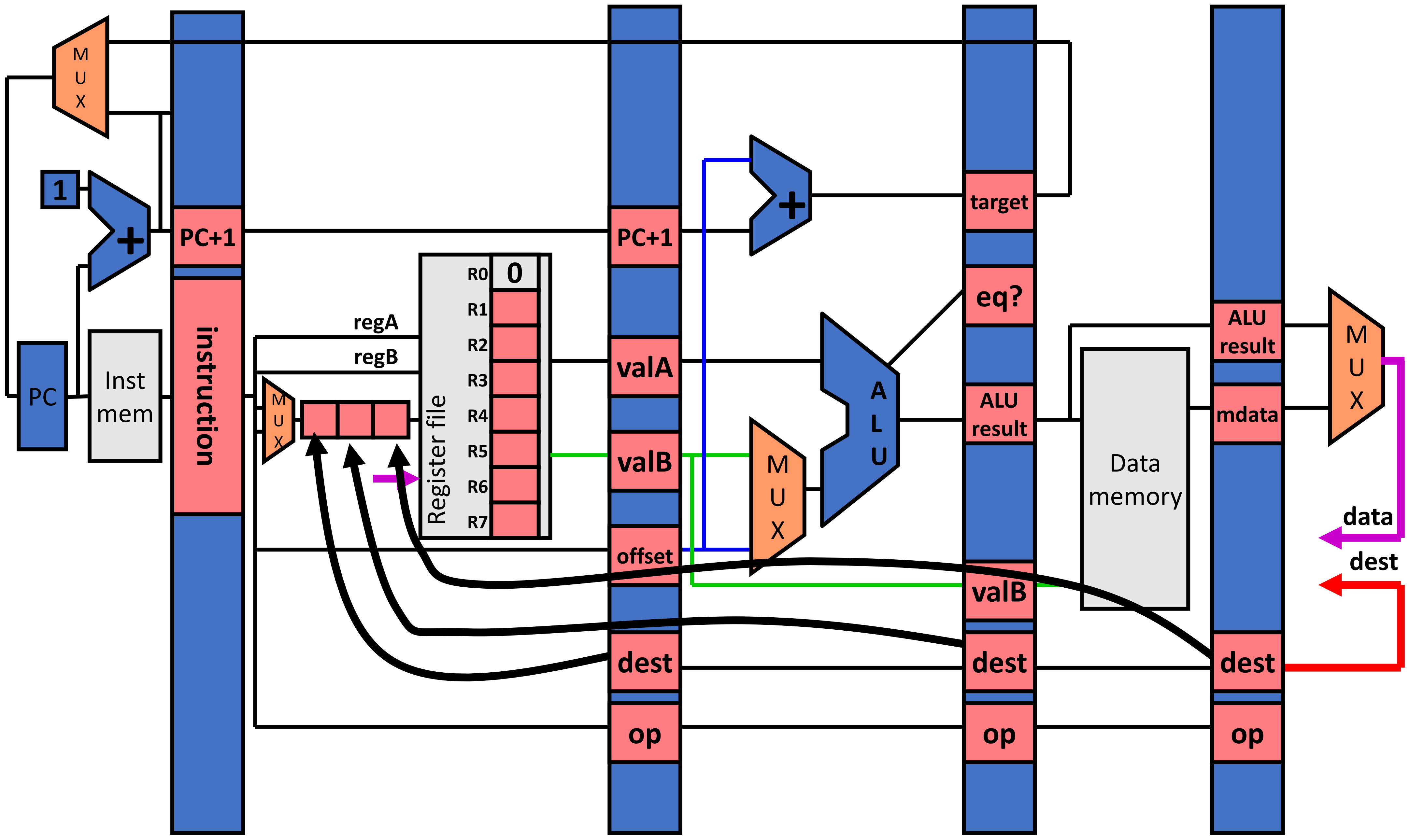
- Detect:
  - Compare regA with previous DestRegs
    - 3 bit operand fields
  - Compare regB with previous DestRegs
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute

- How do we modify the pipeline to do this?

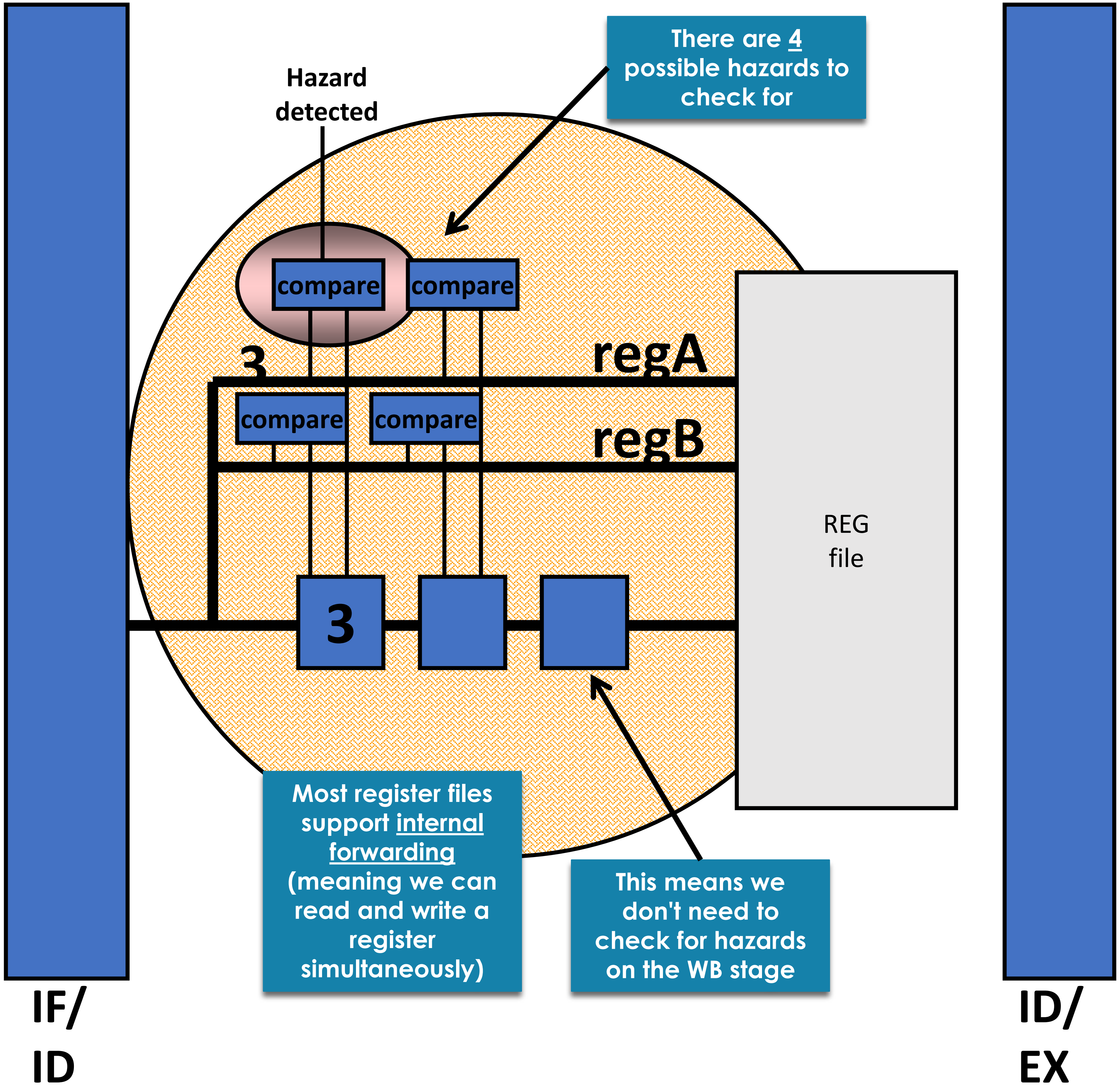
Our pipeline currently does not handle hazards—let's fix it



	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*			



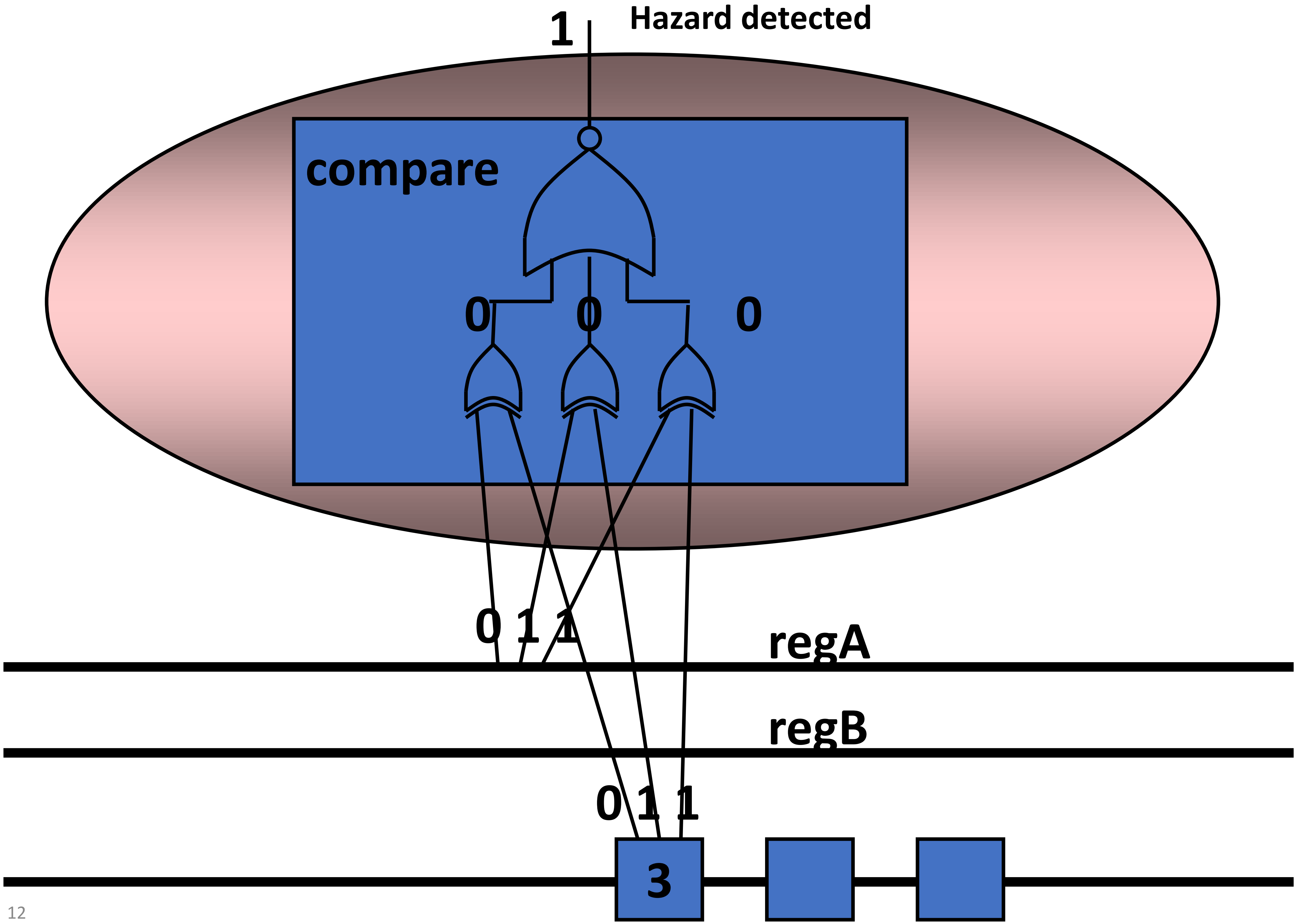
IF/ ID ID/ EX EX/ Mem Mem/ WB



There are 4 possible hazards to check for

Most register files support internal forwarding (meaning we can read and write a register simultaneously)

This means we don't need to check for hazards on the WB stage



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

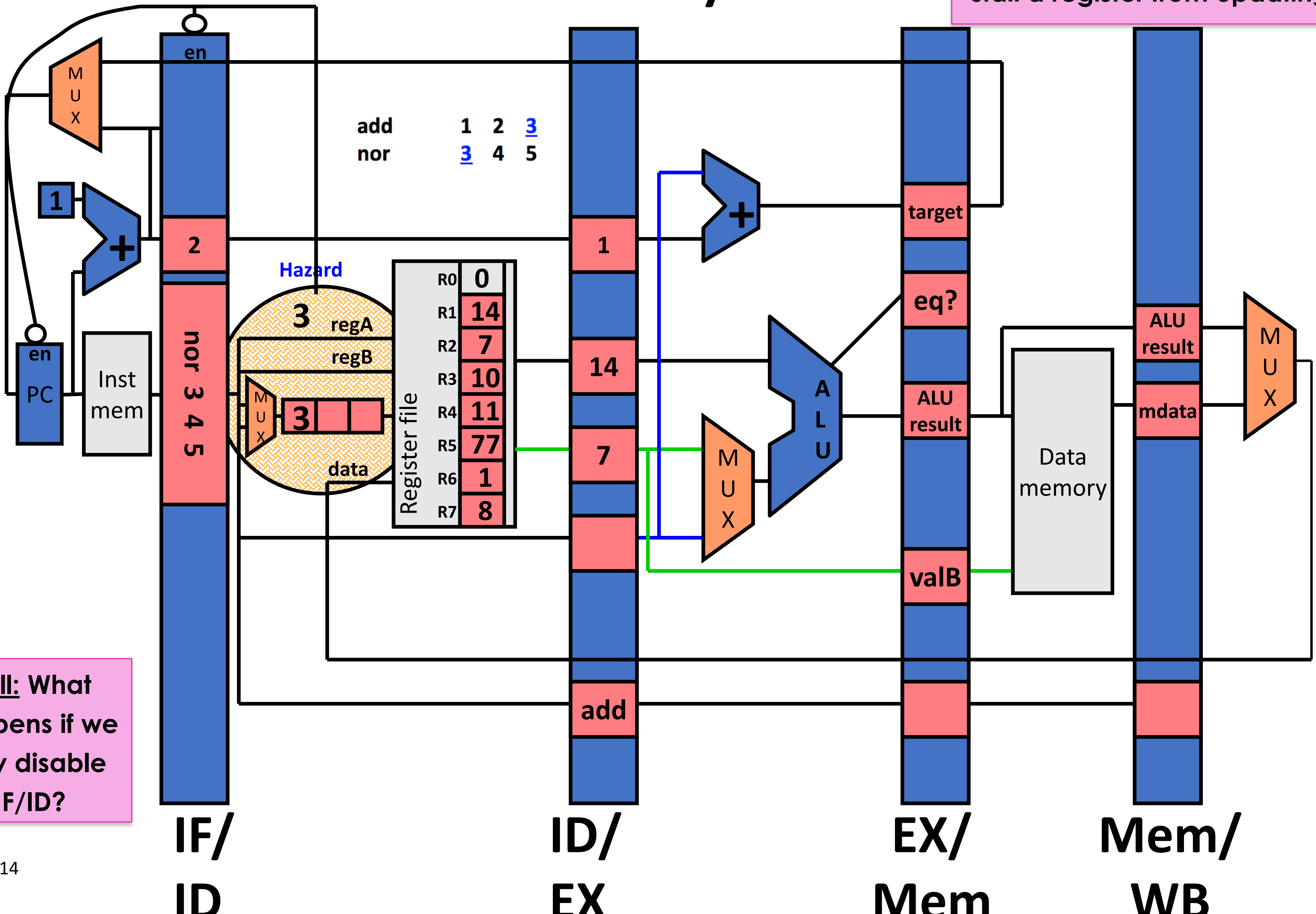
- We will start at the beginning of cycle 3, where add is in the EX stage, and nor is in the ID stage, about to read a register value

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

Hazard!

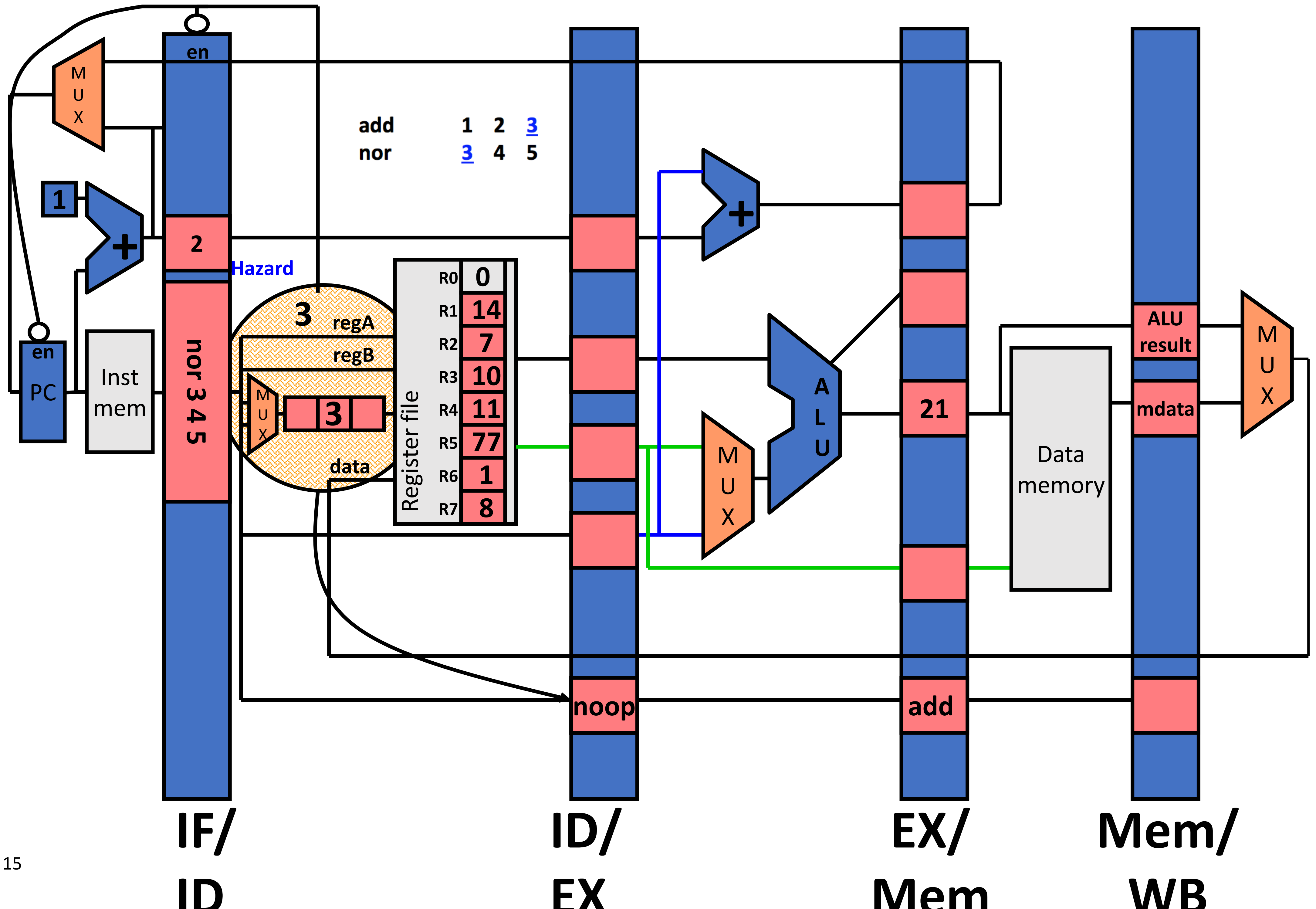
First half of cycle 3

Poll: What's the easiest way to stall a register from updating?

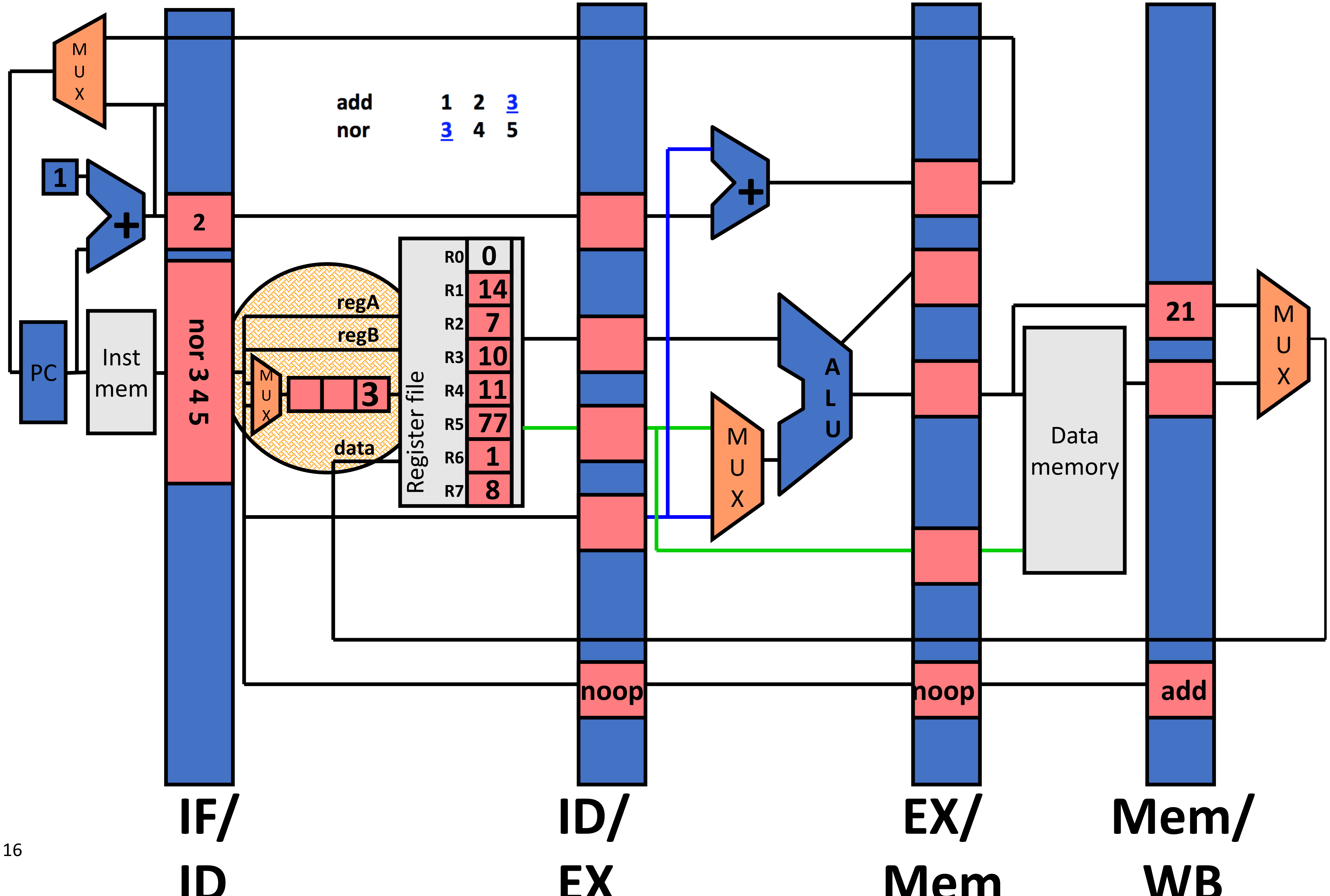


Poll: What happens if we only disable IF/ID?

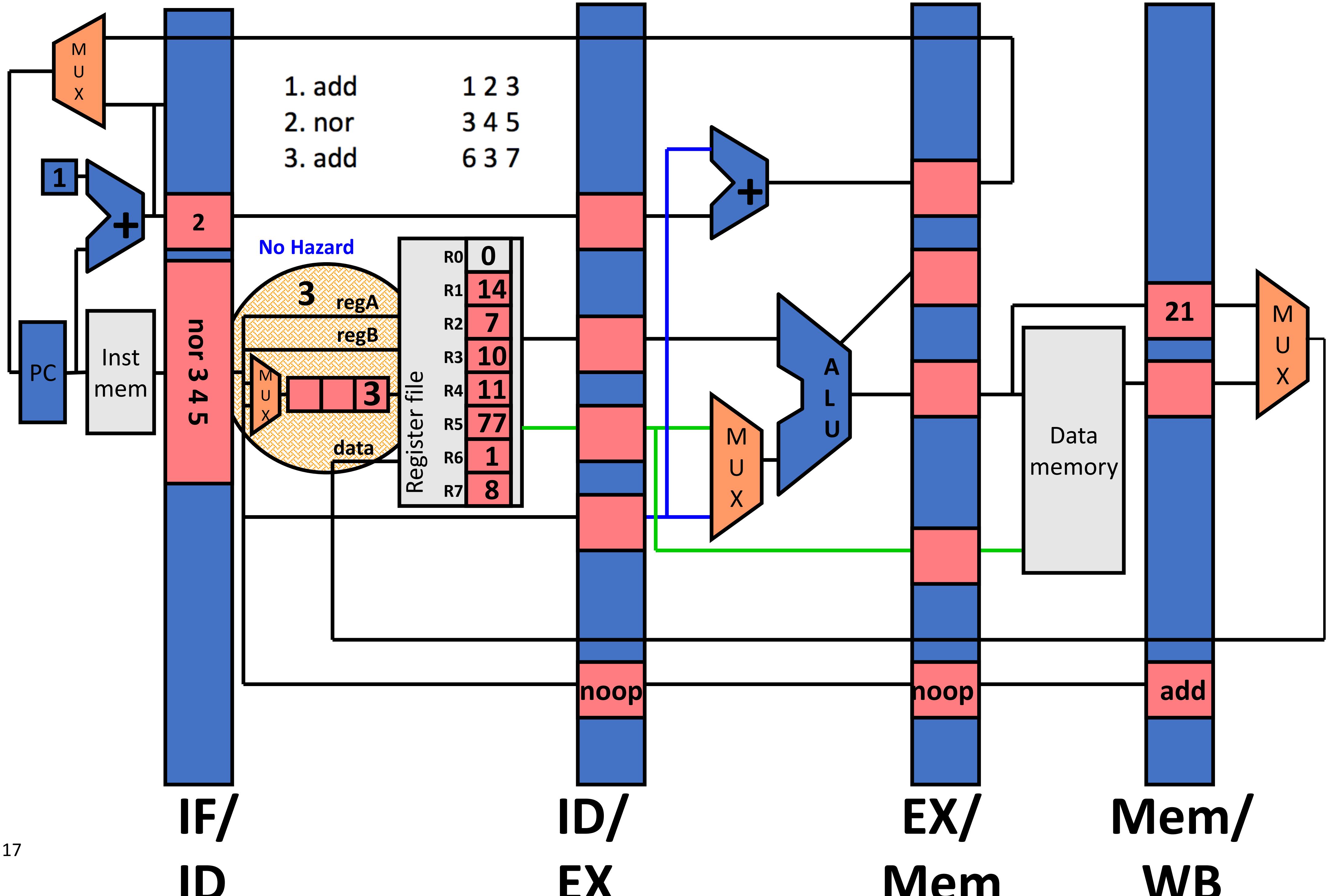
First half of cycle 4



End of cycle 4

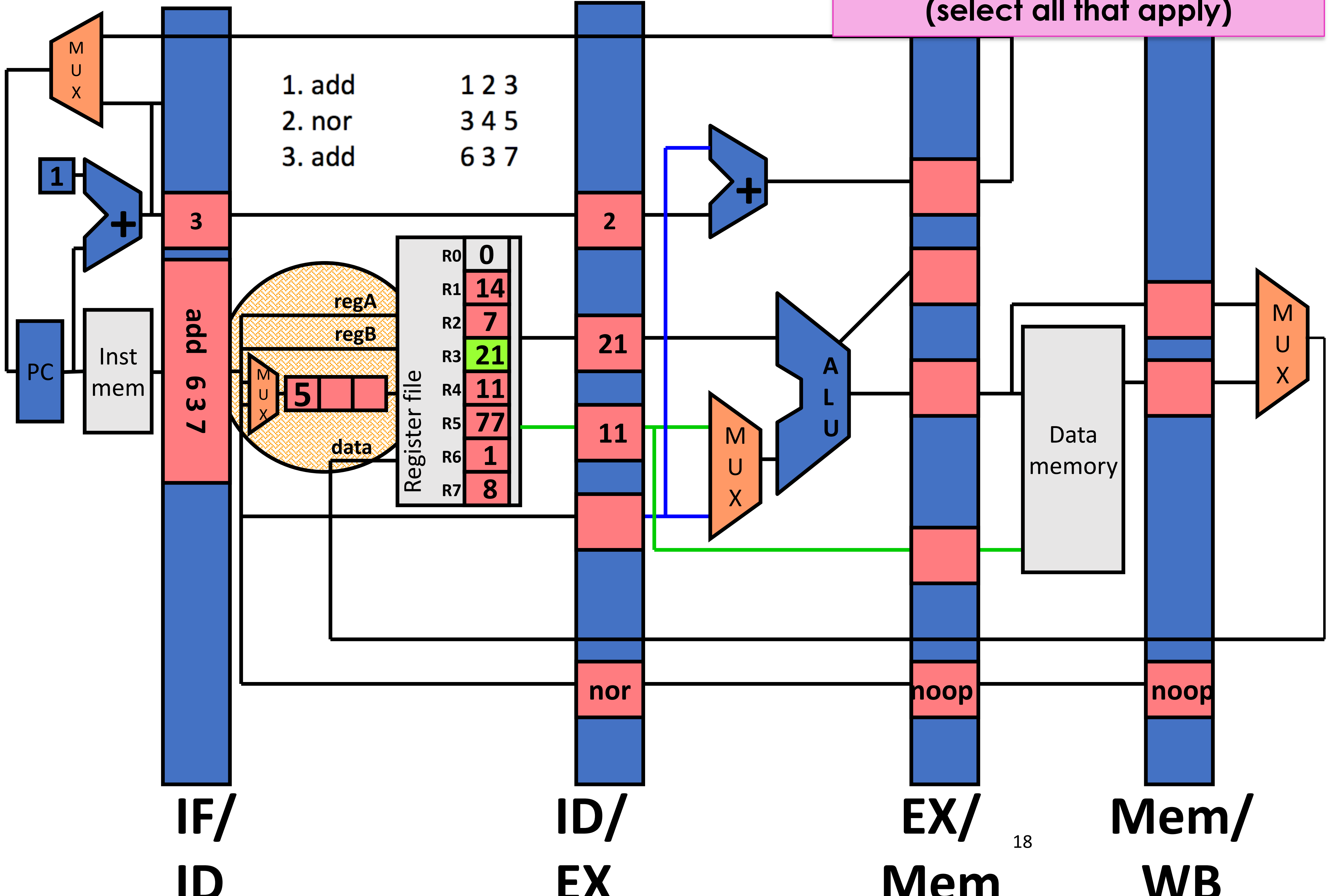


First half of cycle 5



End of cycle 5

Poll: How many cycles of delay can each hazard introduce? (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

Poll: **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

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 logic

Handling data hazards III: Detect and forward

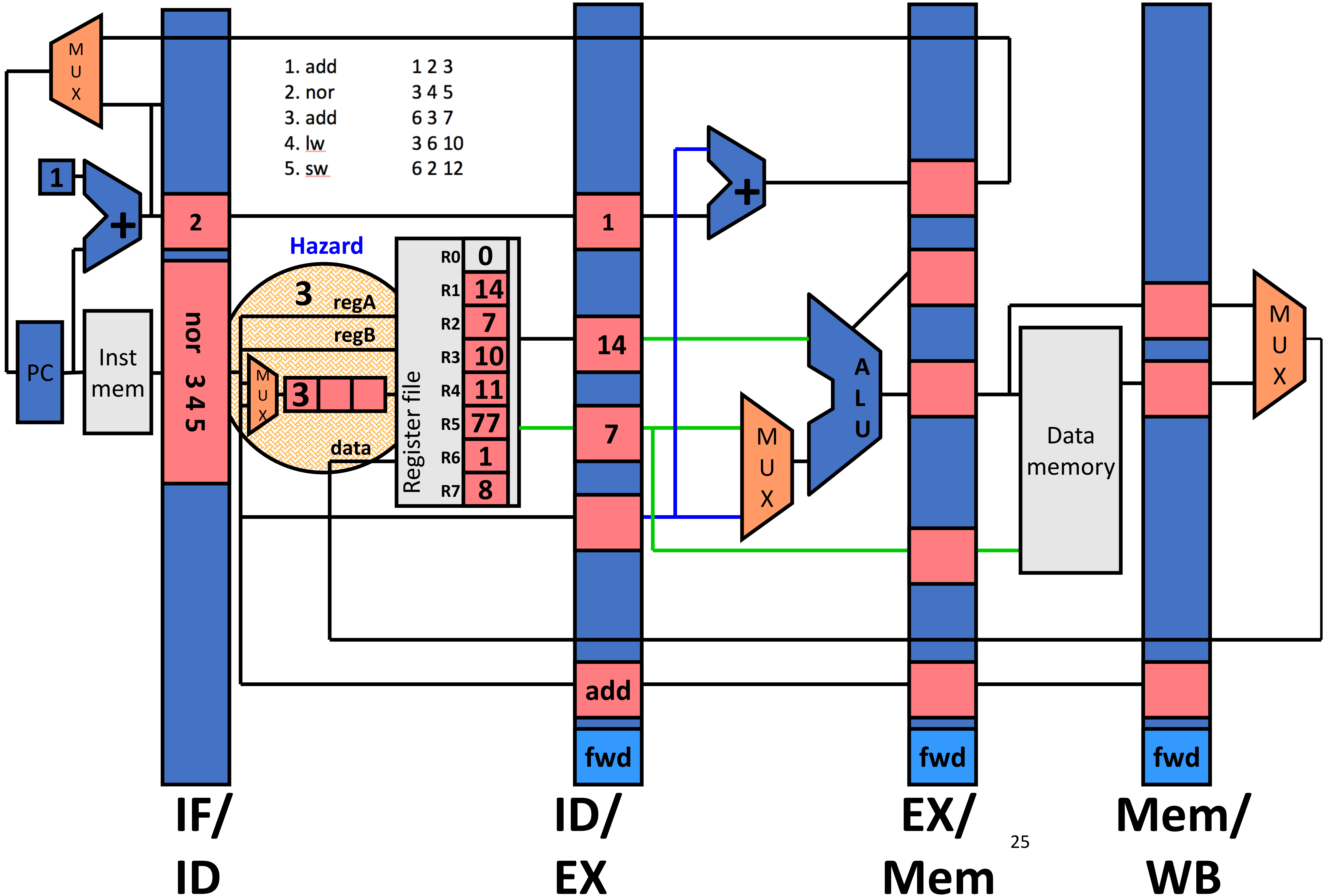
- Detect: same as detect and stall
  - Except that all 4 hazards have to be treated differently
    - i.e., you can't logical-OR the 4 hazard signals
- Forward:
  - New bypass datapaths route computed data to where it is needed
  - New MUX and control to pick the right data
- Beware: Stalling may still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

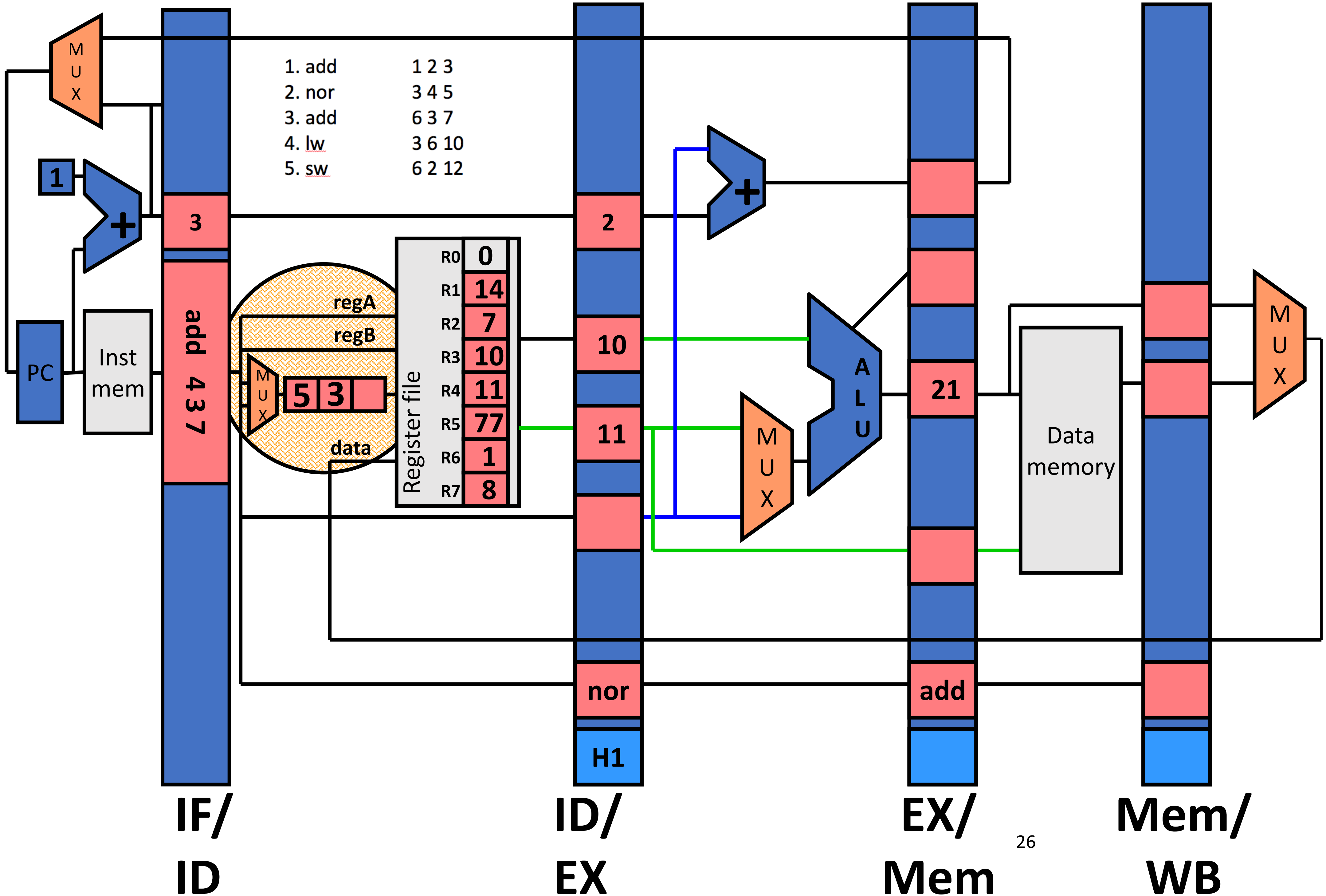
```
1. add    1 2 3
2. nor    3 4 5
3. add    6 3 7
4. lw     3 6 10
5. sw     6 2 12
```

First half of cycle 3



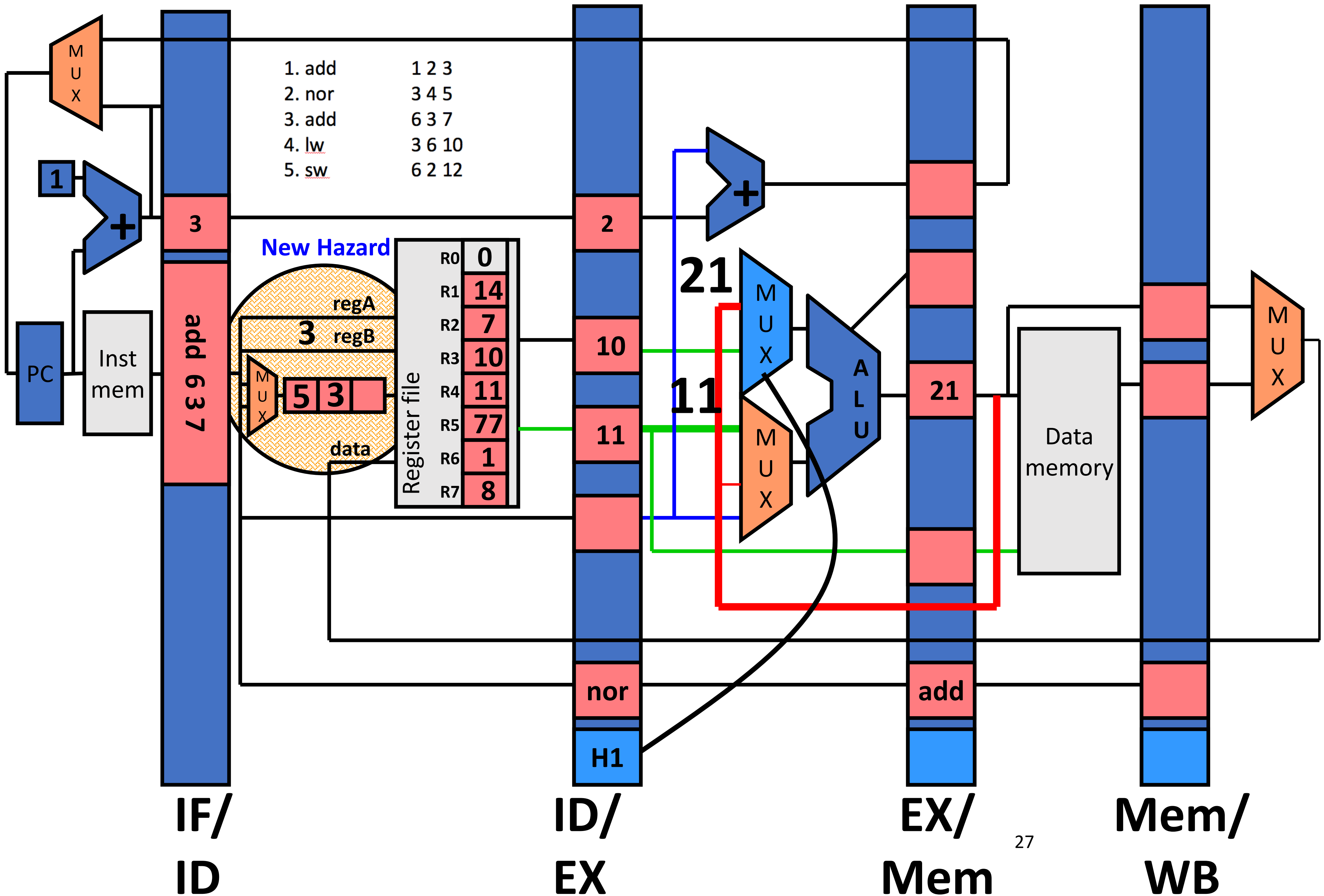
End of cycle 3

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

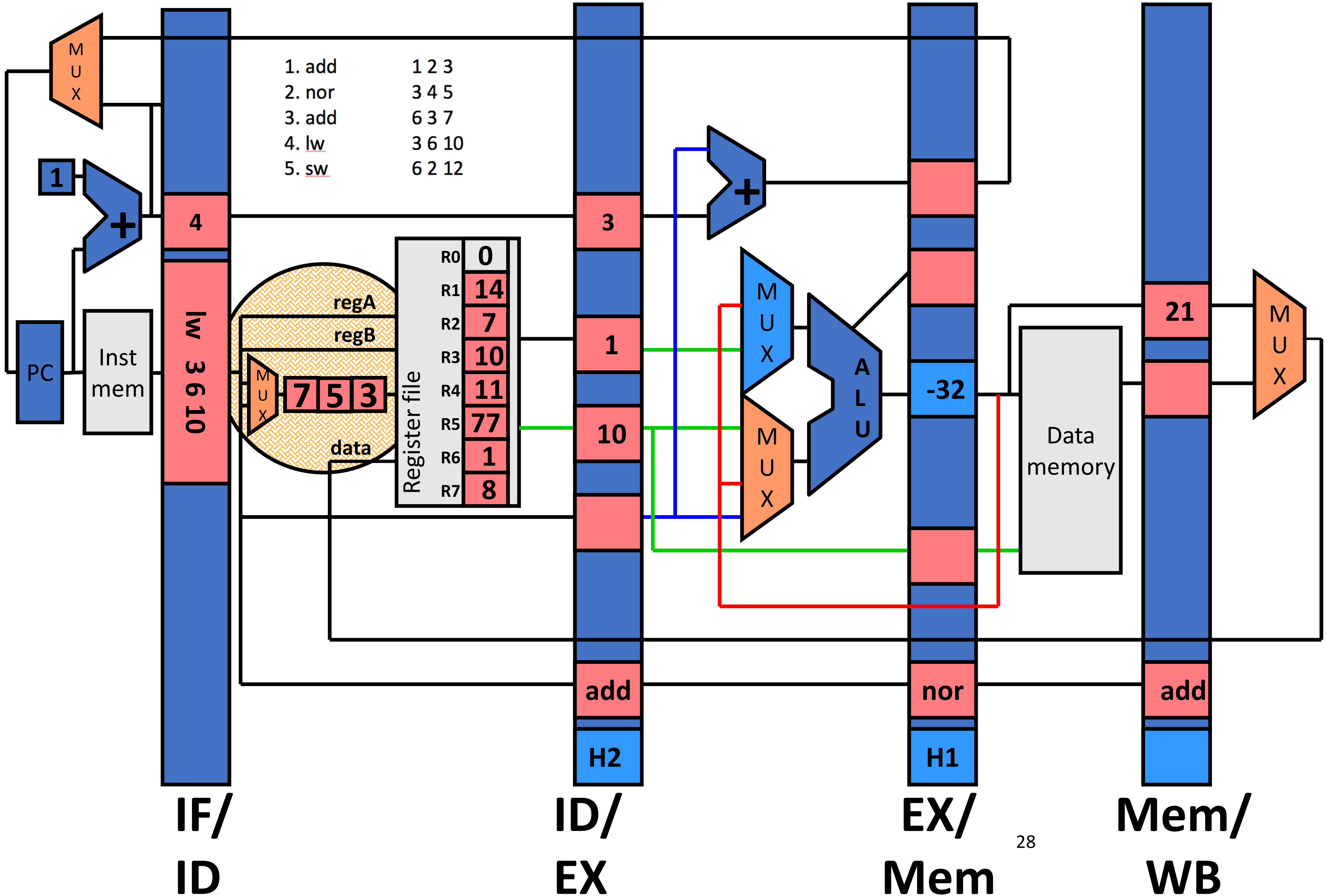


First half of cycle 4

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

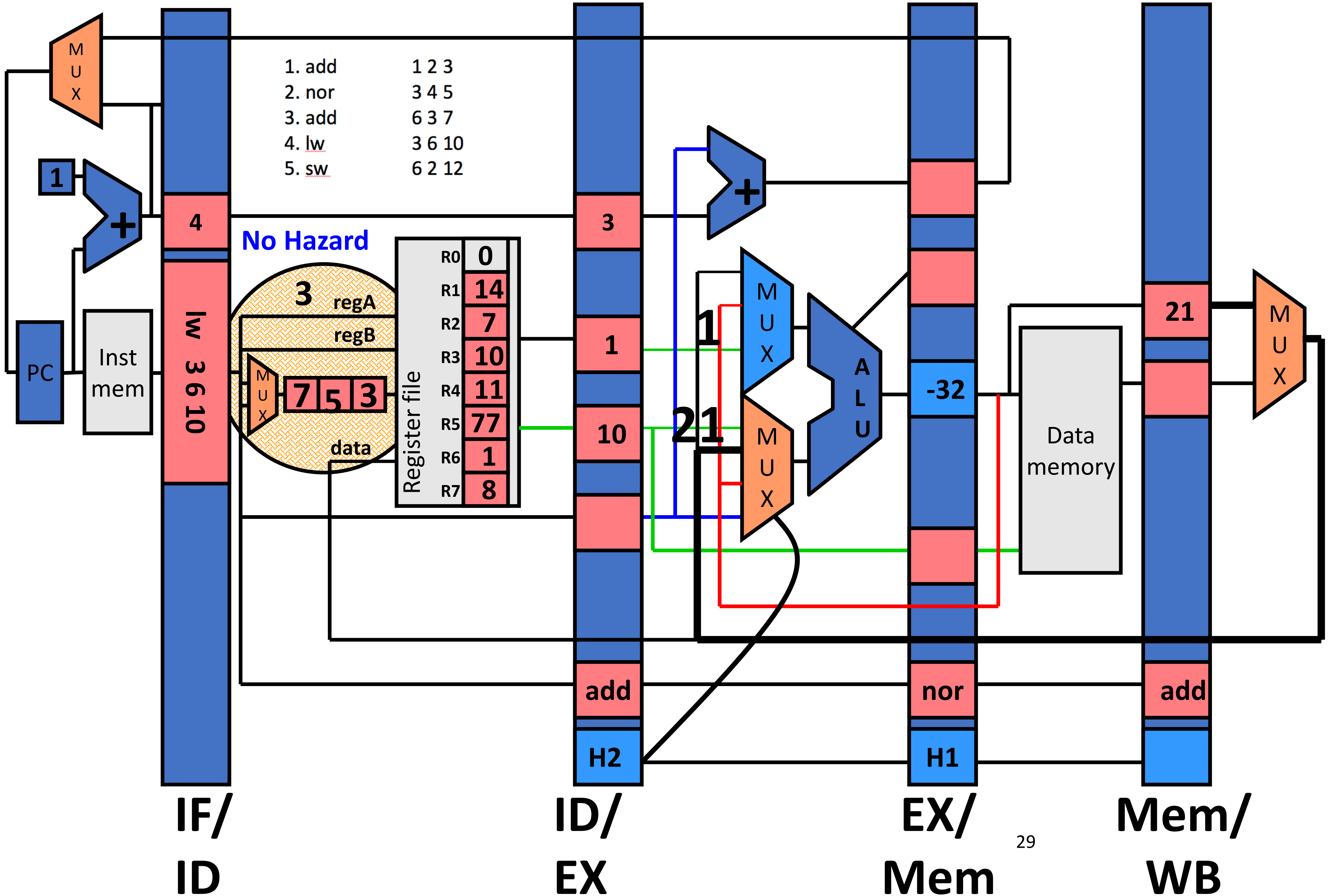


End of cycle 4

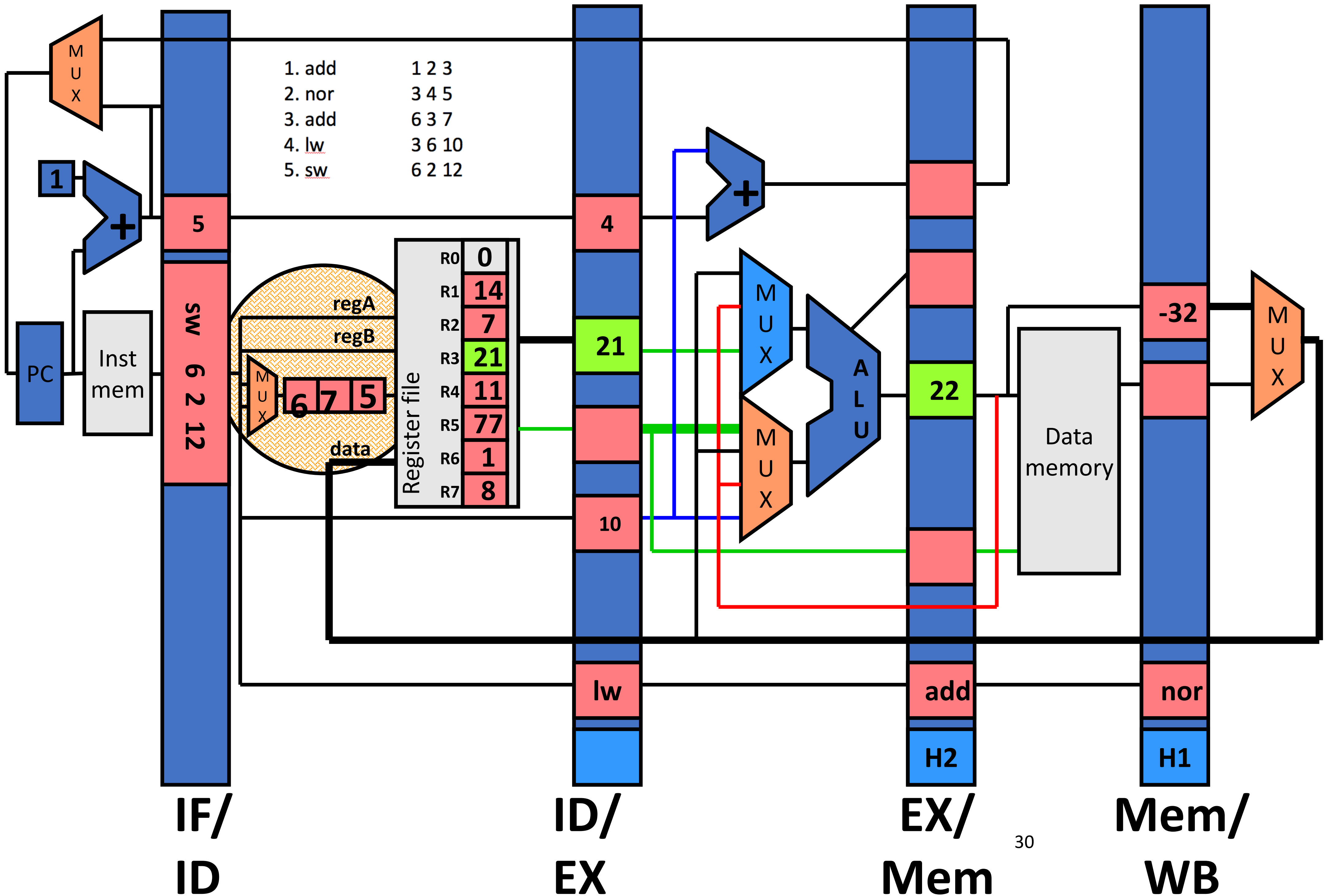


```
1. add     1 2 3
2. nor     3 4 5
3. add     6 3 7
4. lw      3 6 10
5. sw      6 2 12
```

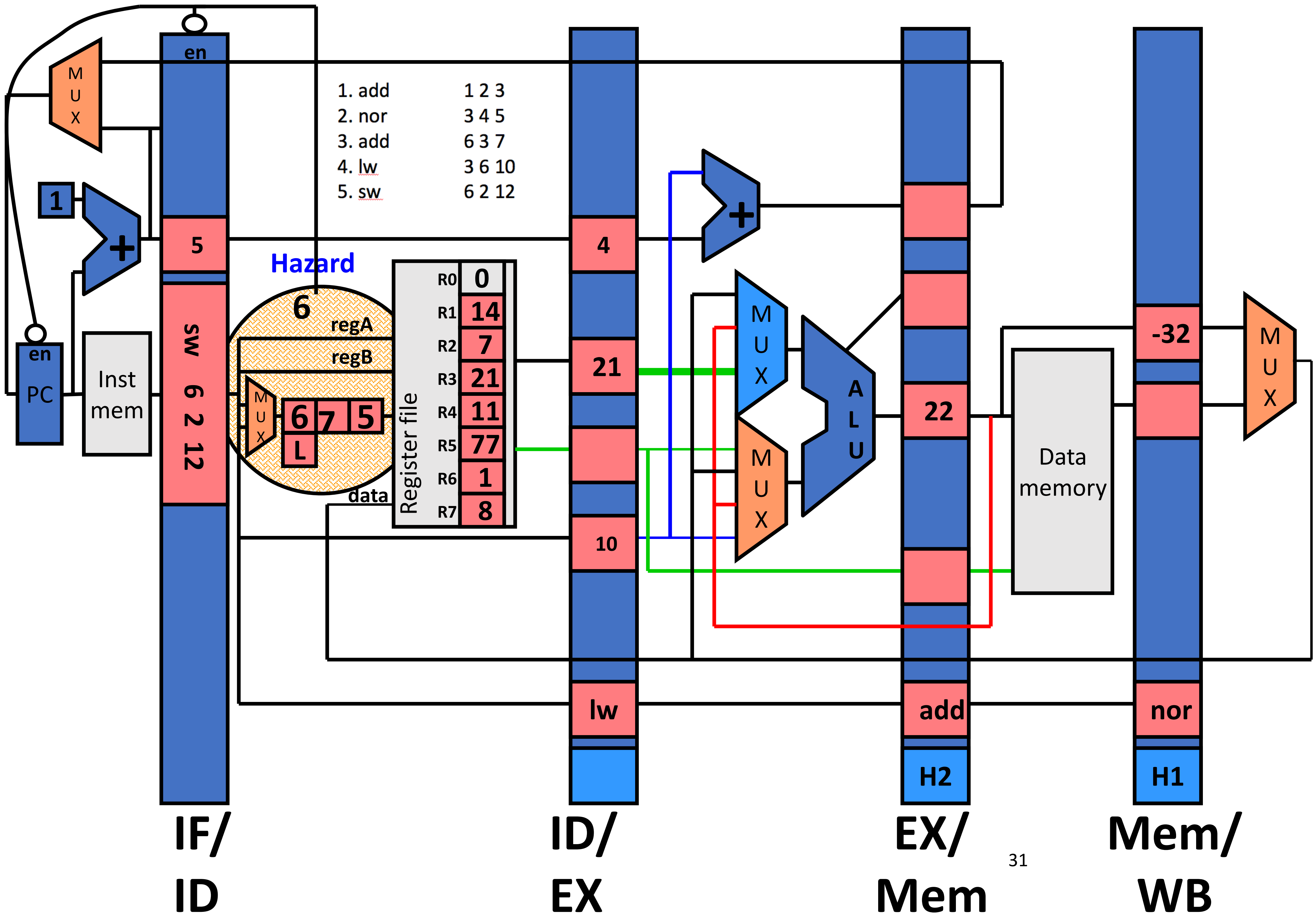
First half of cycle 5



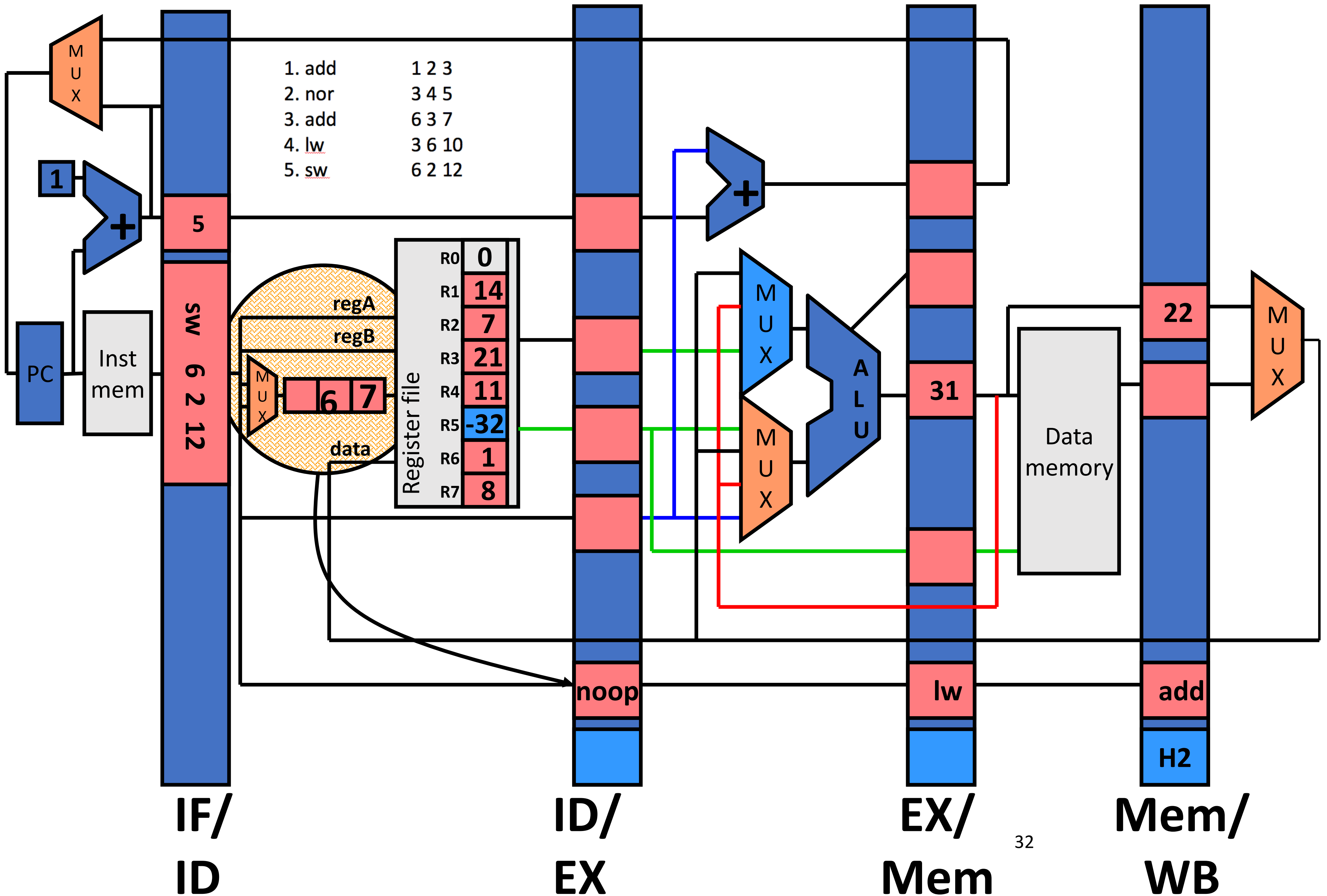
End of cycle 5



First half of cycle 6

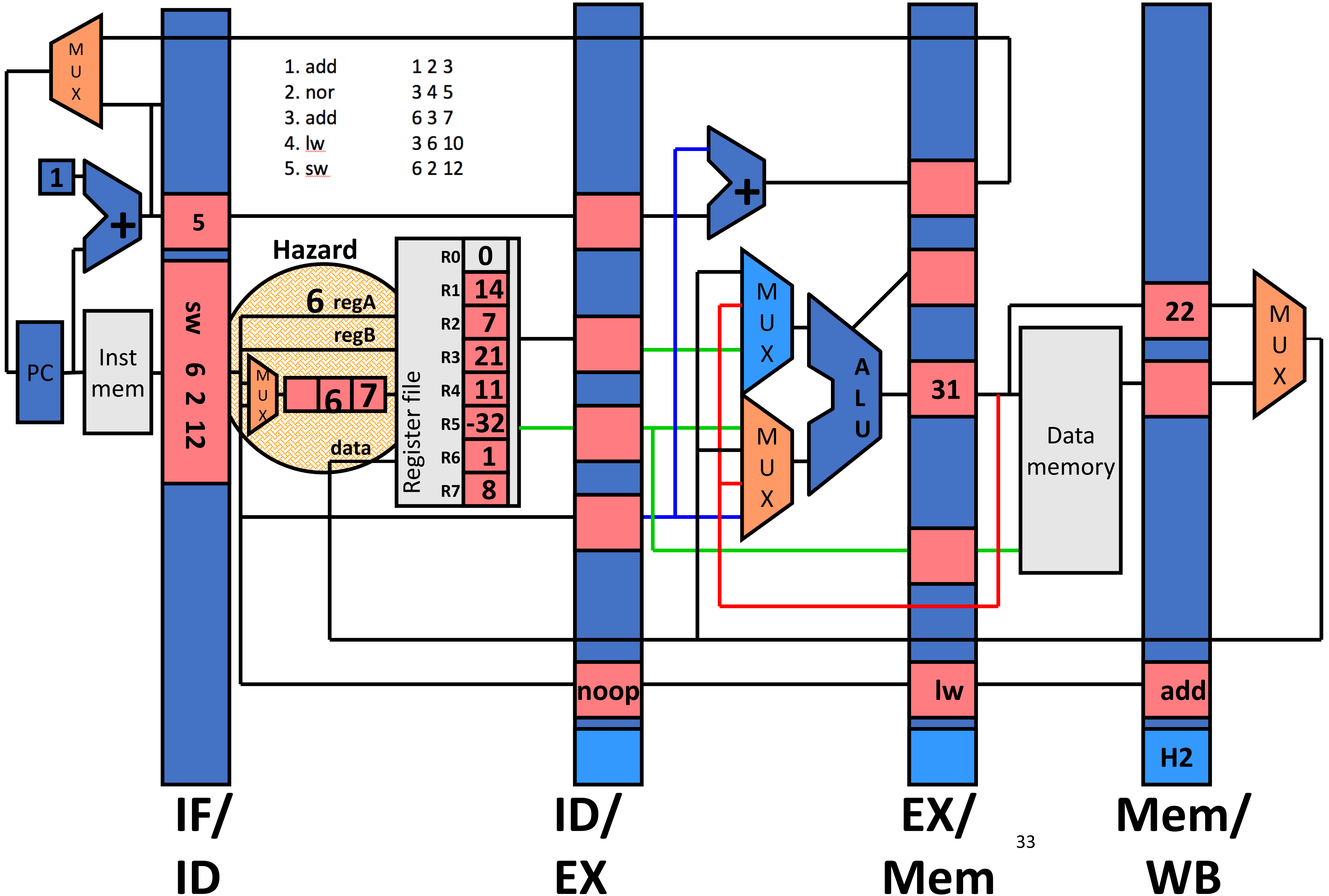


End of cycle 6

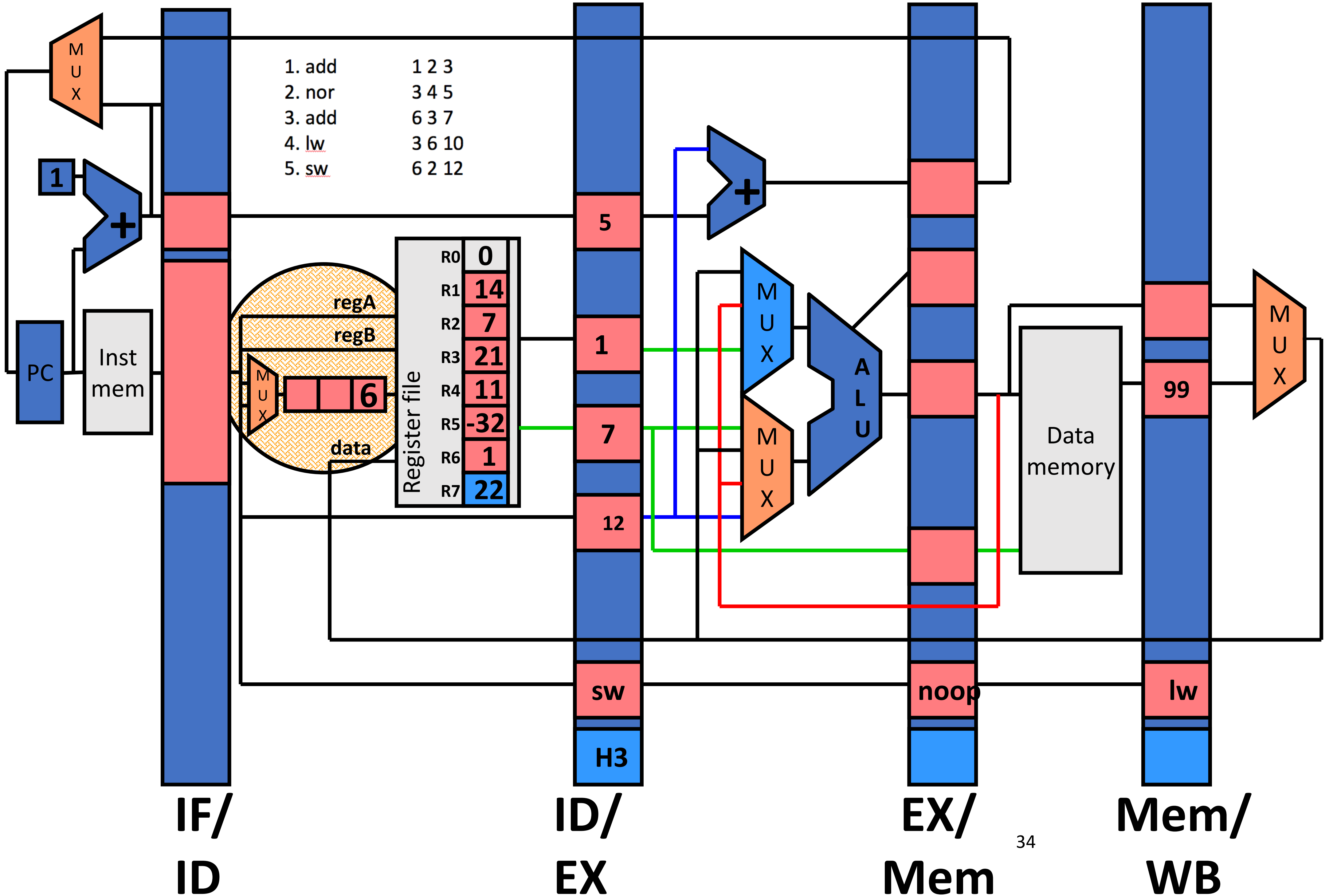


1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12

First half of cycle 7



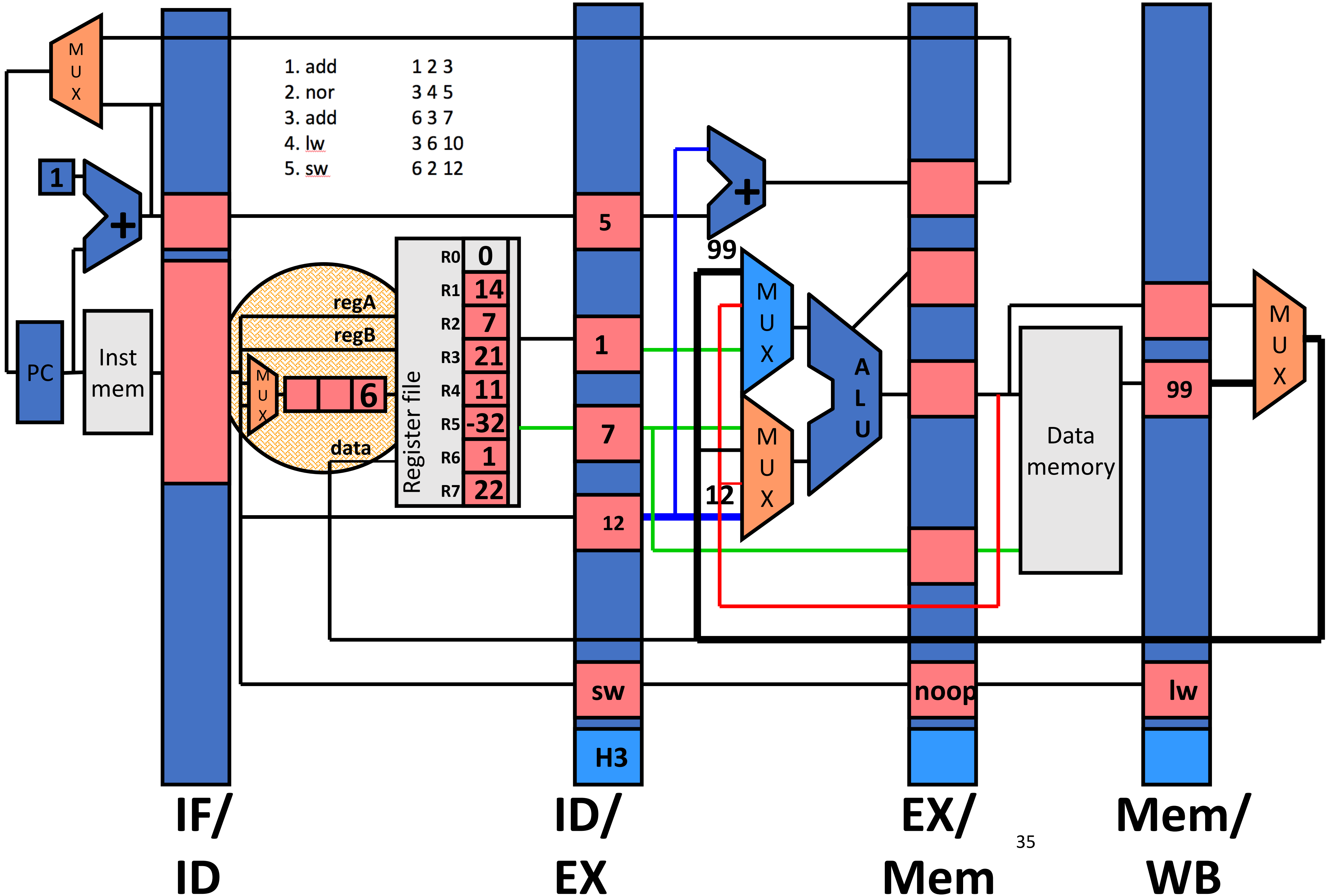
End of cycle 7



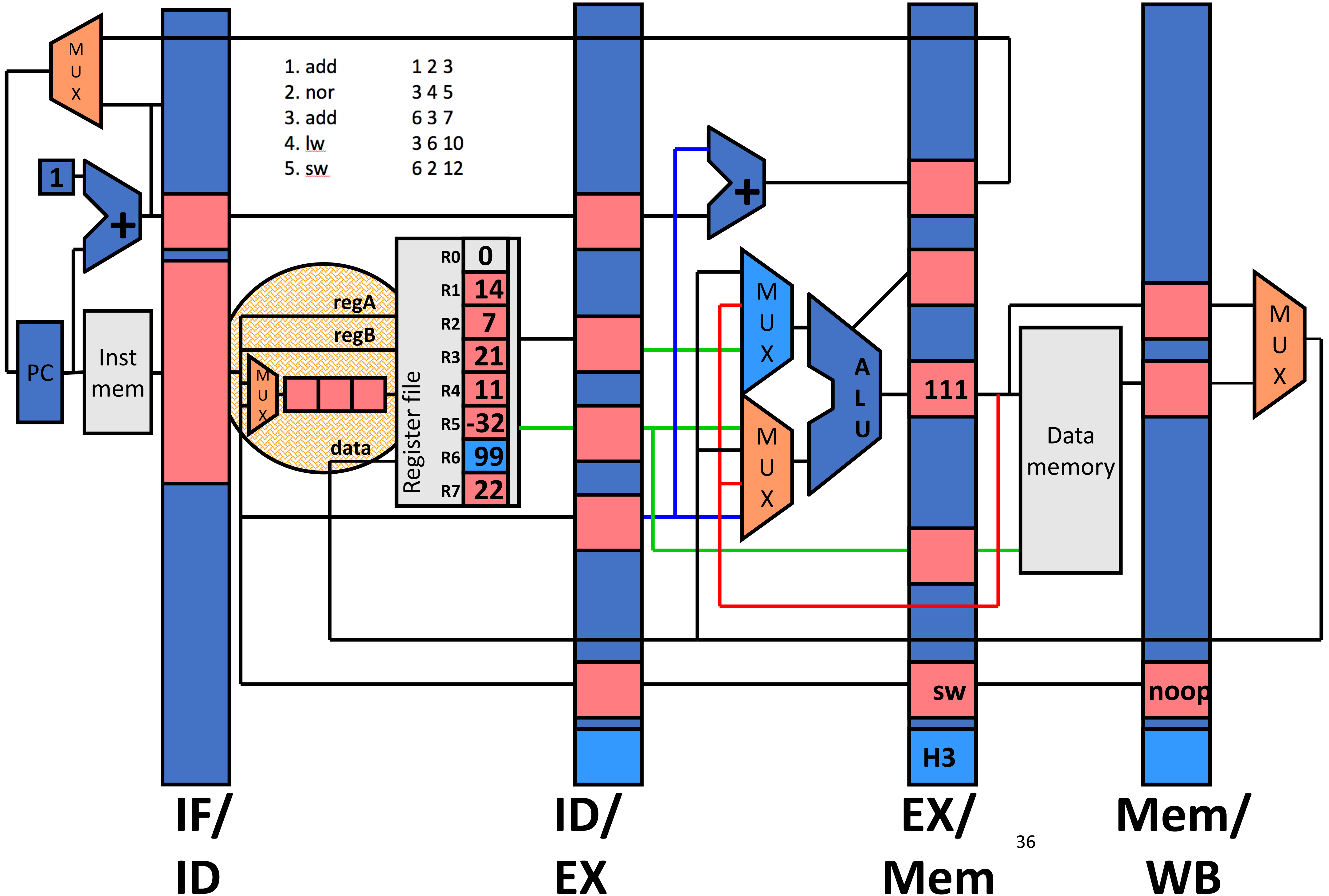
1. add	1 2 3
2. nor	3 4 5
3. add	6 3 7
4. lw	3 6 10
5. sw	6 2 12

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								
add 6 3 7			IF	ID	EX								
lw 3 6 10				IF	ID								
sw 6 2 12					IF								

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			

Next time

- Control hazards
  - How do branches work with pipelining?
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah

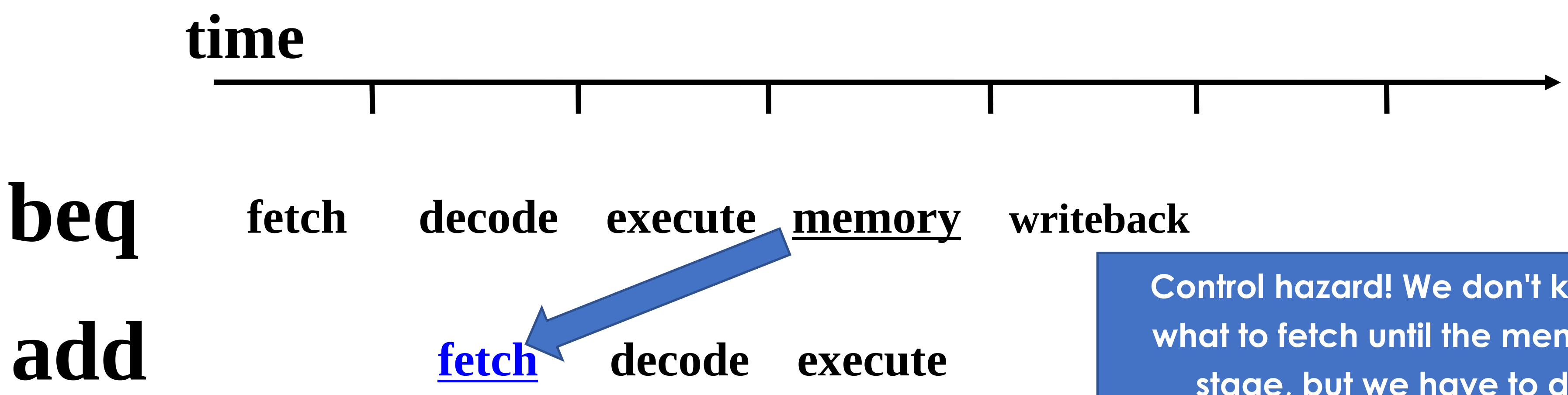
Extra Slides

Other issues

- What other instruction(s) have we been ignoring so far??
- Branches!! (Let's not worry about jumps yet)
- Sequence for BEQ:
    - Fetch: read instruction from memory
    - Decode: read source operands from registers
    - Execute: calculate target address and test for equality
    - Memory: **Send target to PC** if test is equal
    - Writeback: nothing
    - **Branch Outcomes**
        - Not Taken
            - PC = PC + 1
        - Taken
            - PC = Branch Target Address

Control Hazards

beq 1 1 10
add 3 4 5



Control hazard! We don't know what to fetch until the memory stage, but we have to do SOMETHING now!

Approaches to handling control hazards

- 3 strategies – similar to handling data hazards

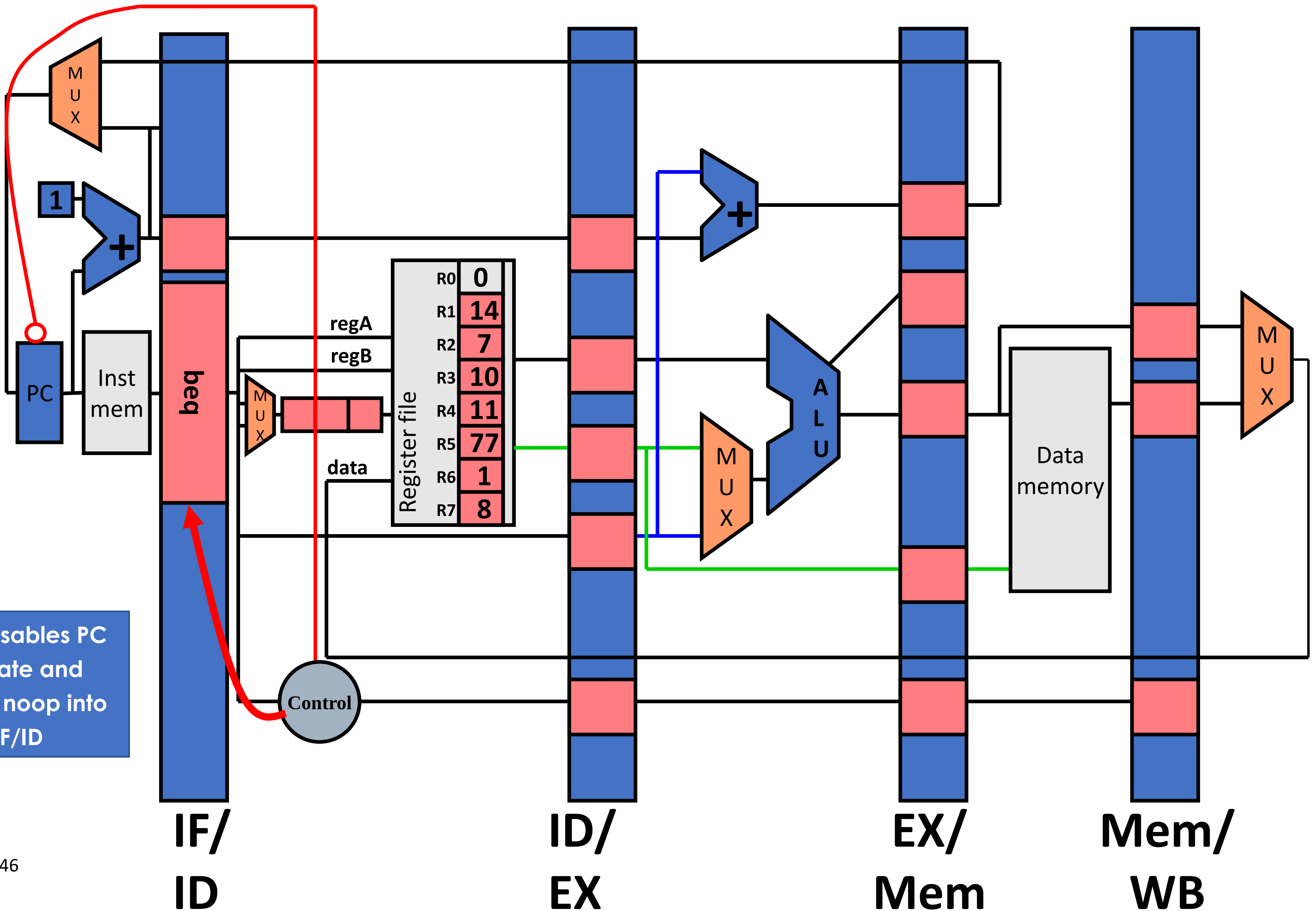
1. Avoid
   - Make sure there are no hazards in code
2. Detect and stall
   - Delay fetch until branch resolved
3. Speculate and squash-if-wrong
   - Guess outcome of branch
   - Fetch instructions assuming we're right
   - Stop them if they shouldn't have been executed

Avoiding Control Hazards

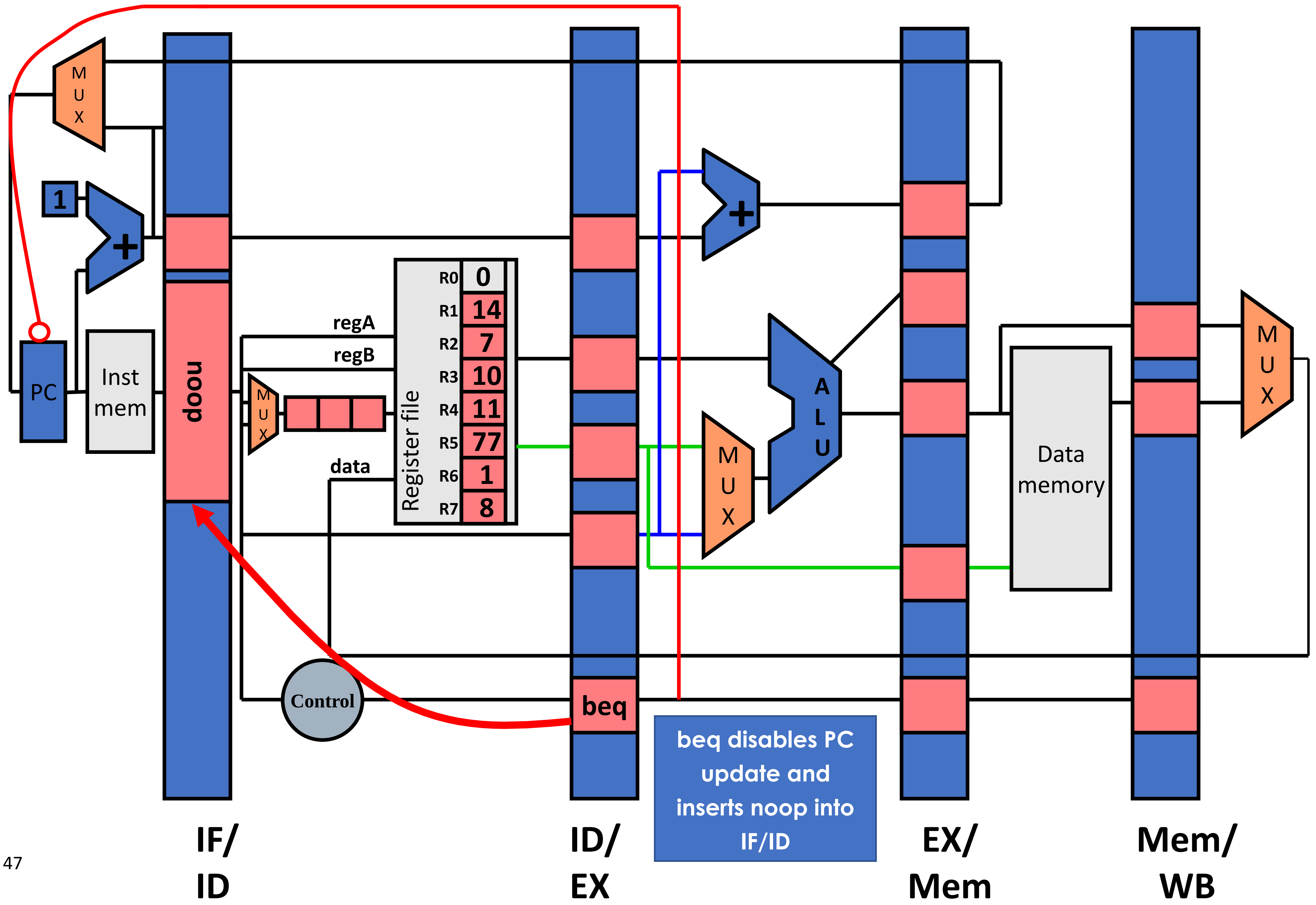
- Don't have branch instructions!
  - Possible, but not practical
  - ARM offers **predicated** instructions (instructions that throw away result if some condition is not met)
    - Allows replacement of if/else conditions
    - Hard to use for everything
    - Not covered more in this class

Detect and Stall

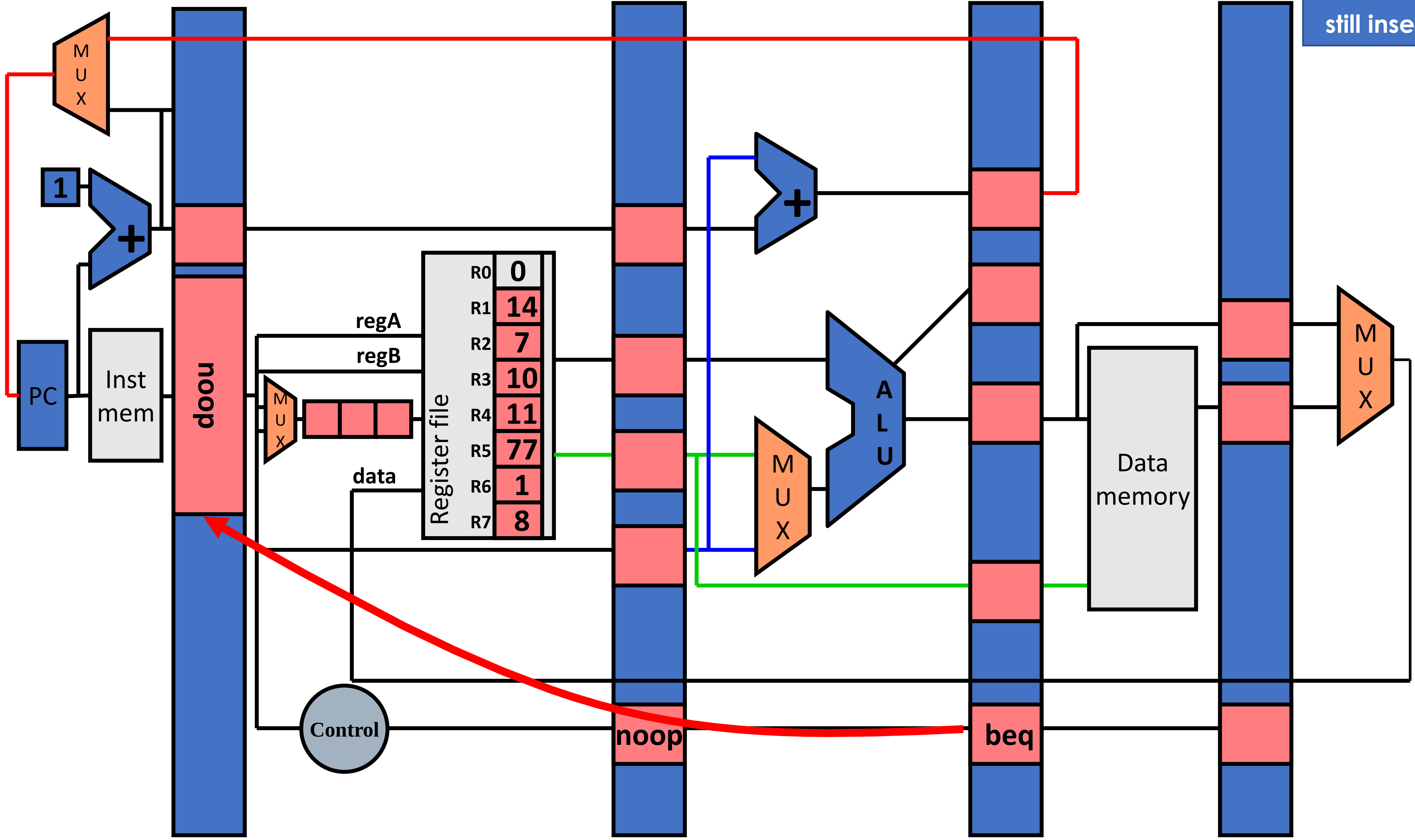
- Detection
  - Wait until decode
  - Check if opcode == beq or jalr
- Stall
  - Keep current instruction in fetch
  - Insert noops
  - Pass noop to decode stage, not execute!



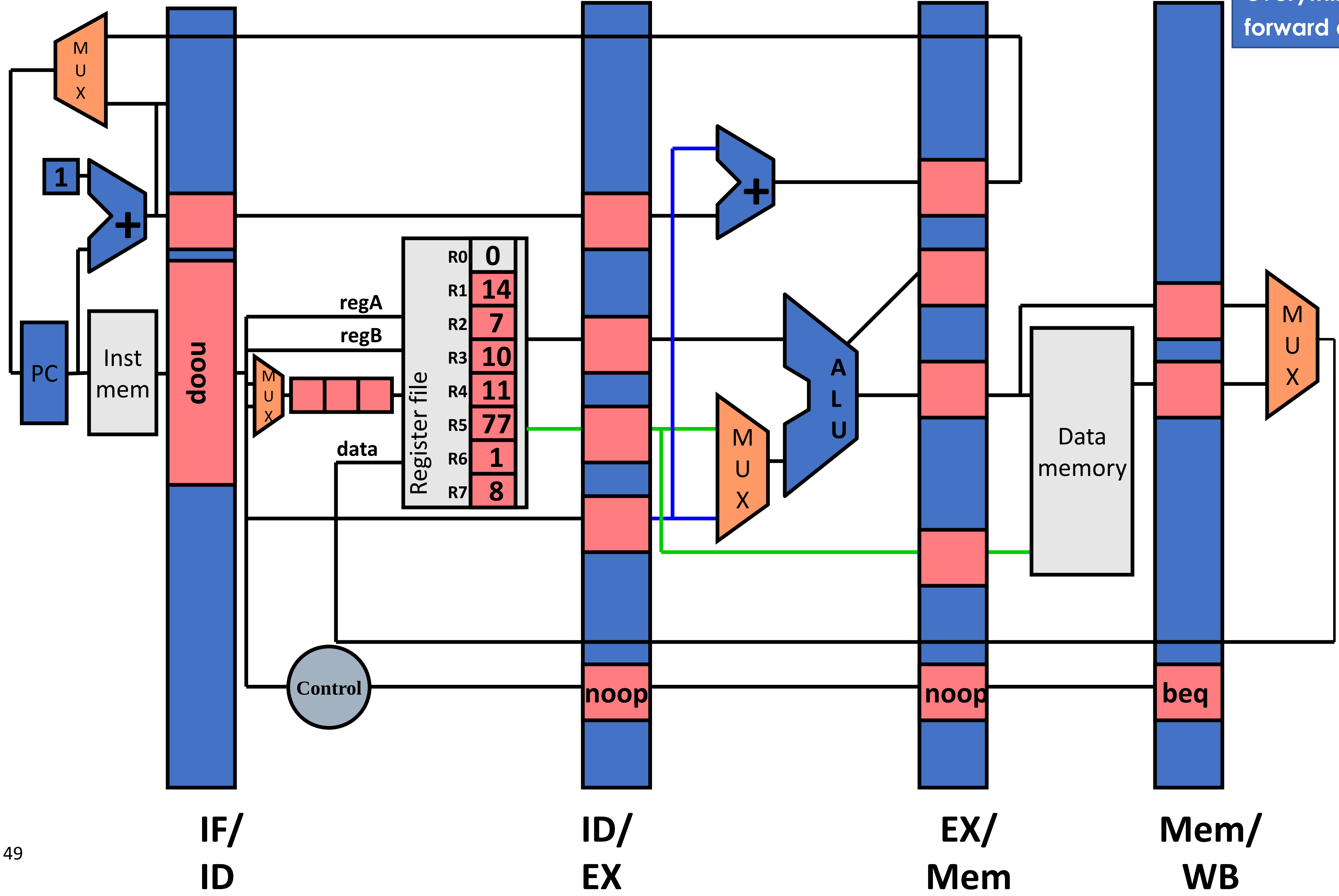
beq disables PC update and inserts noop into IF/ID

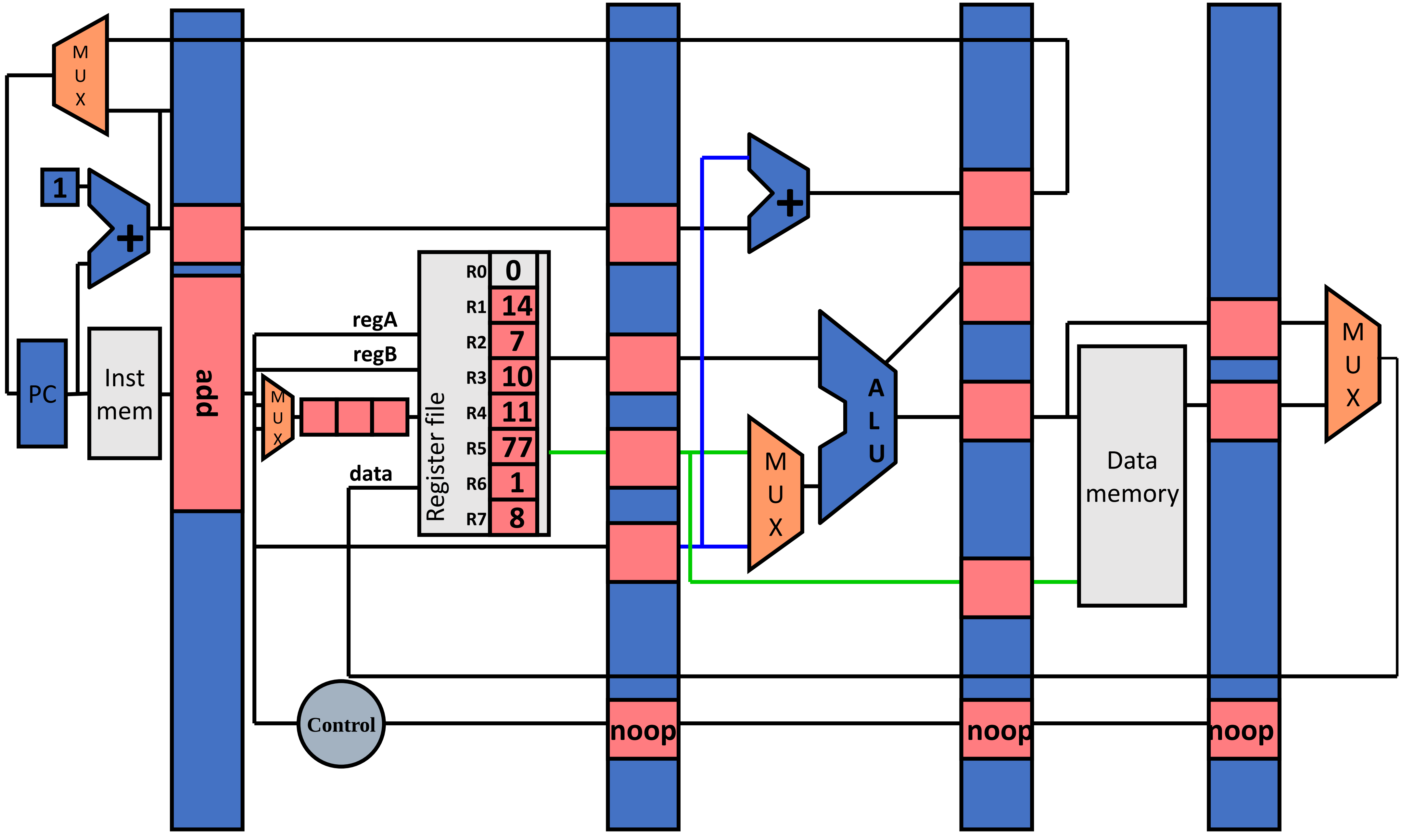


beq disables PC update and inserts noop into IF/ID beq DOESN'T disable PC and sends target, but still inserts NOOP



Target PC is now sent to memory, everything moves forward as normal




MUX
1
+
PC
Inst mem
add
MUX
regA
regB
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
Control
+
MUX
ALU
Data memory
MUX
noop
noop
noop
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB