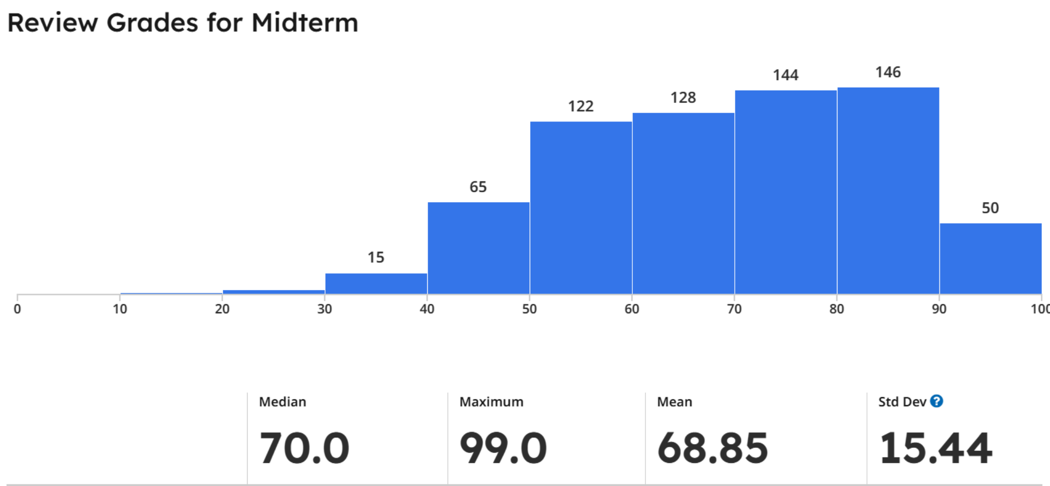
EECS 370 - Lecture 14

Pipelining and Data Hazards II

Announcements

- P2
  - P2L+P2R due Thursday
- Lab 8
  - **Pre-lab quiz due Thursday at 11:55 pm**
- Midterm scores out
  - Regrade requests by end of day Friday
  - Will announce any grade threshold changes after



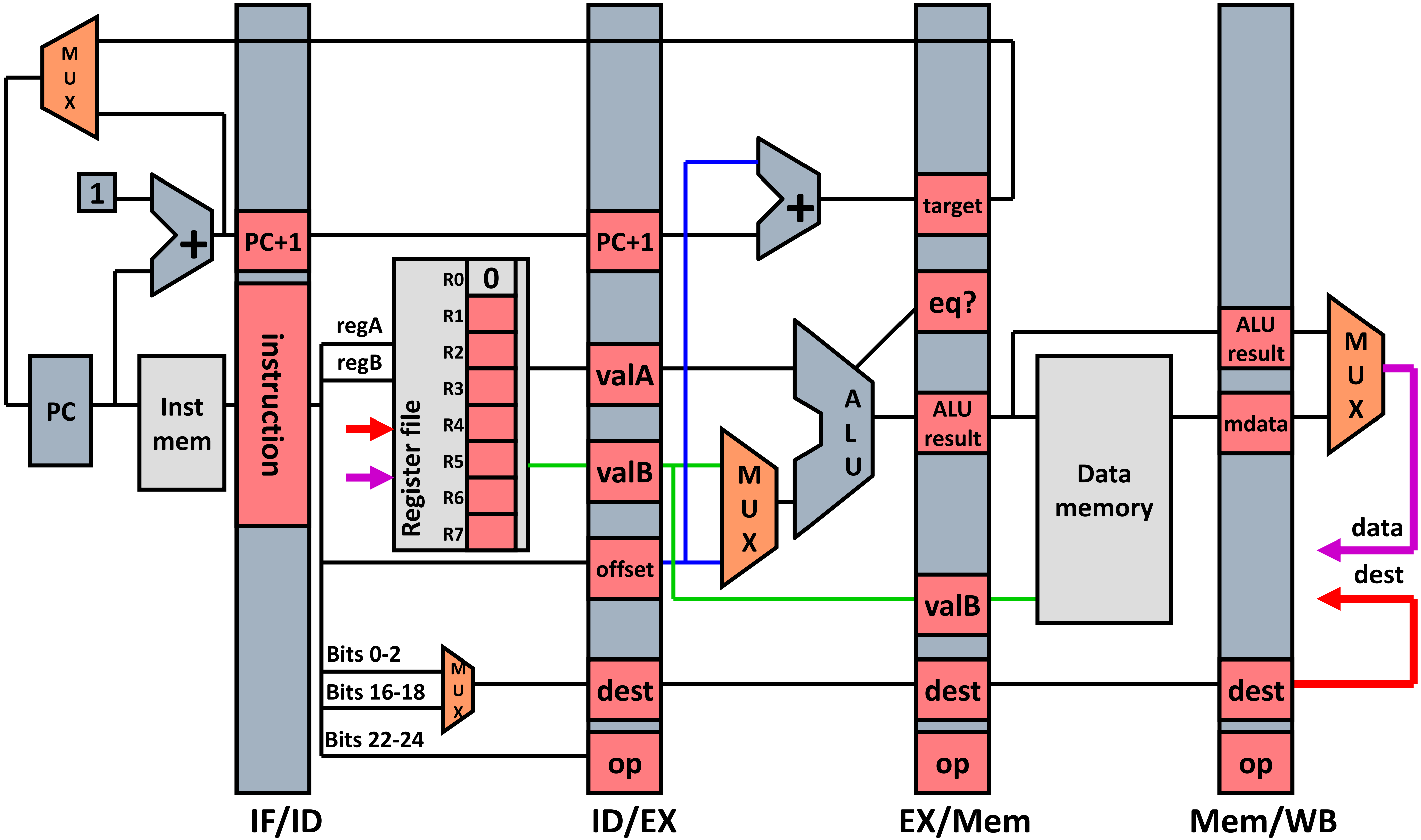
My OH

- My office hours today will be in 2901 BBB
- I’m holding extra OH this Wednesday @ 4pm over Zoom

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

Our new pipelined datapath



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	
nor		fetch	decode	execute	memory	writeback

Hazard

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

<u>Poll:</u> 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    ← write register 3 in cycle 5
noop
noop             ← read register 3 in cycle 5
nor   3  4  5
```

Problems with this solution

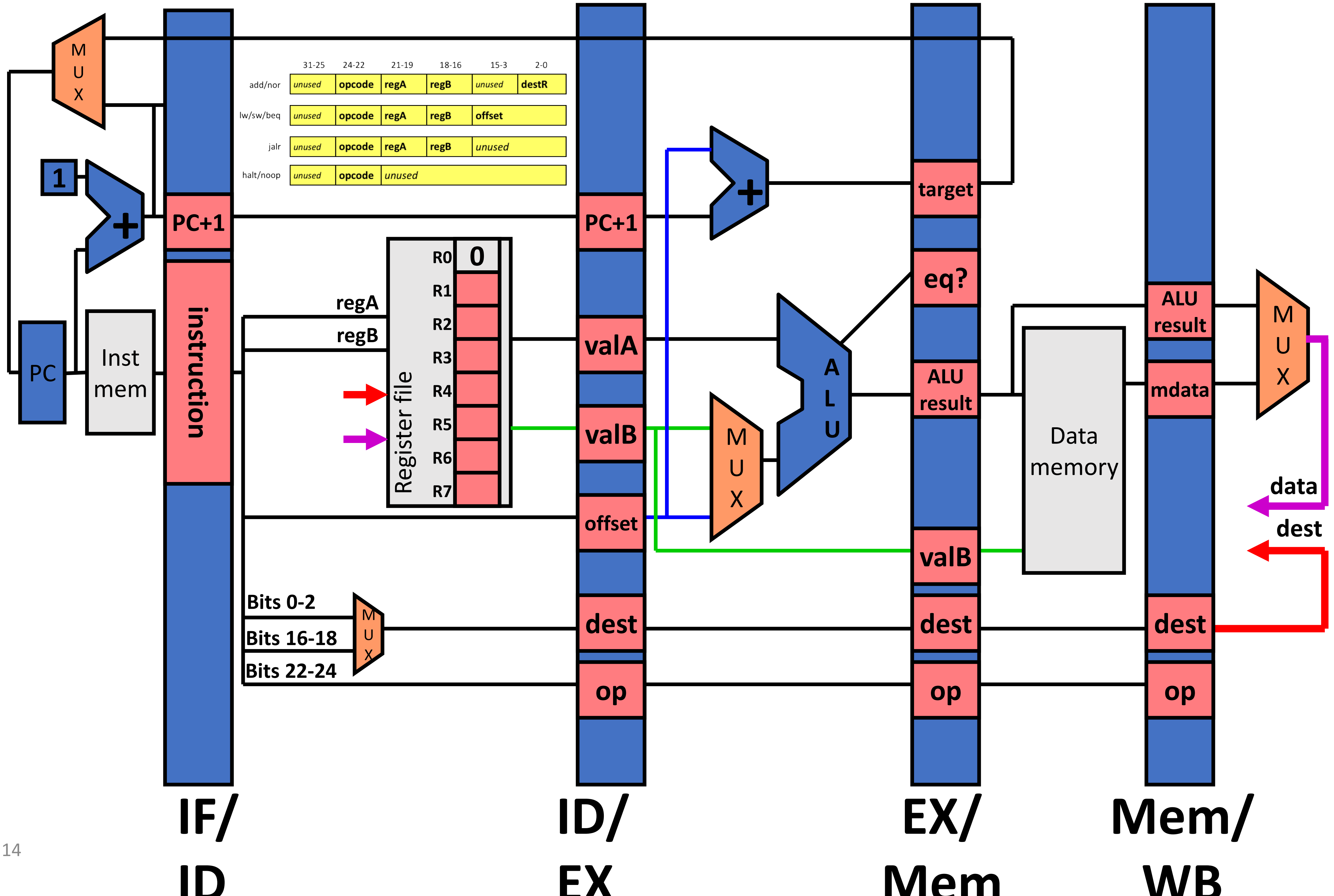
- Old programs (legacy code) may not run correctly on new implementations
  - Longer pipelines need more noops
- Programs get larger as noops are included
  - Especially a problem for machines that try to execute more than one instruction every cycle
  - Intel EPIC: Often 25% - 40% of instructions are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

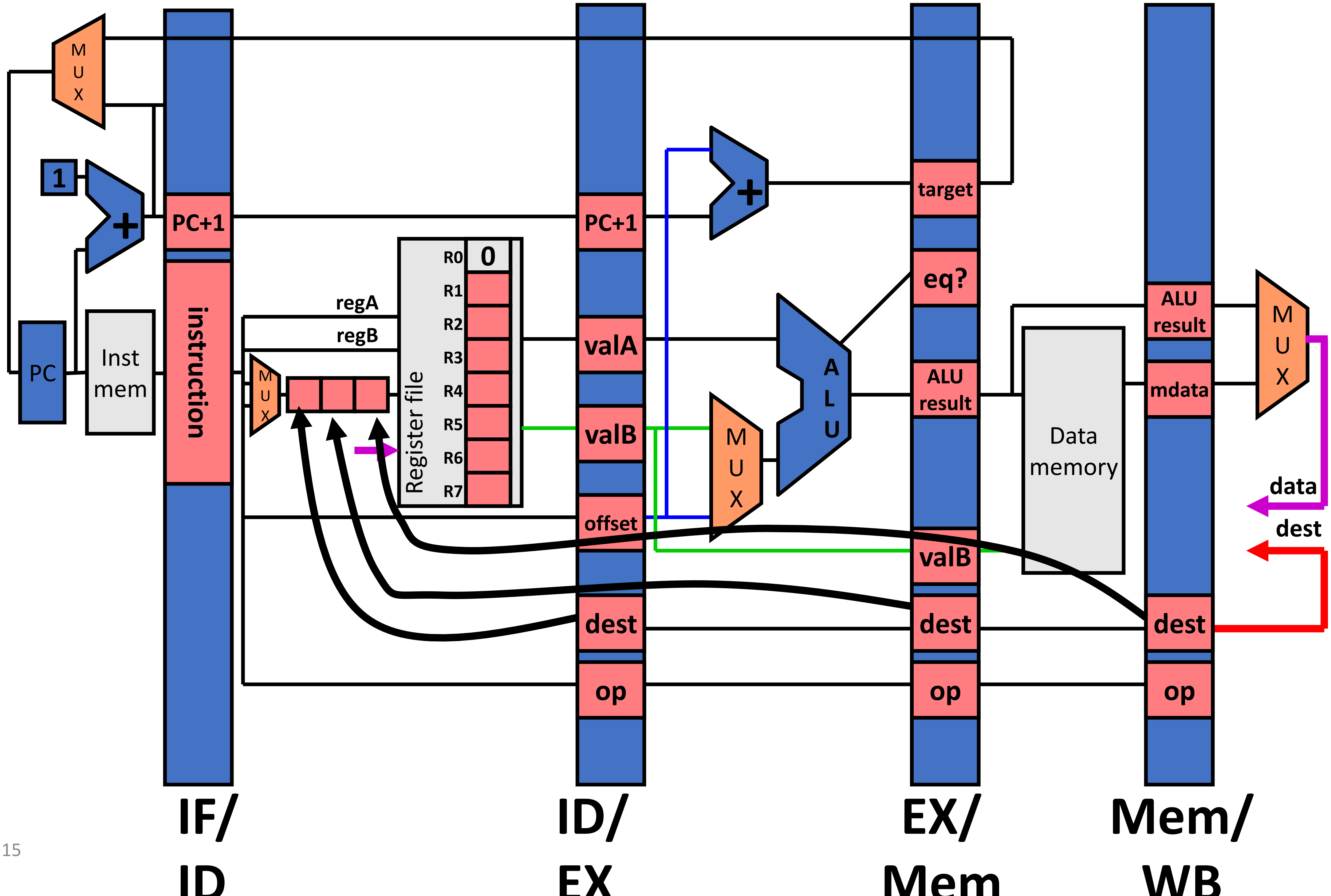
Handling data hazards II: Detect and stall until ready

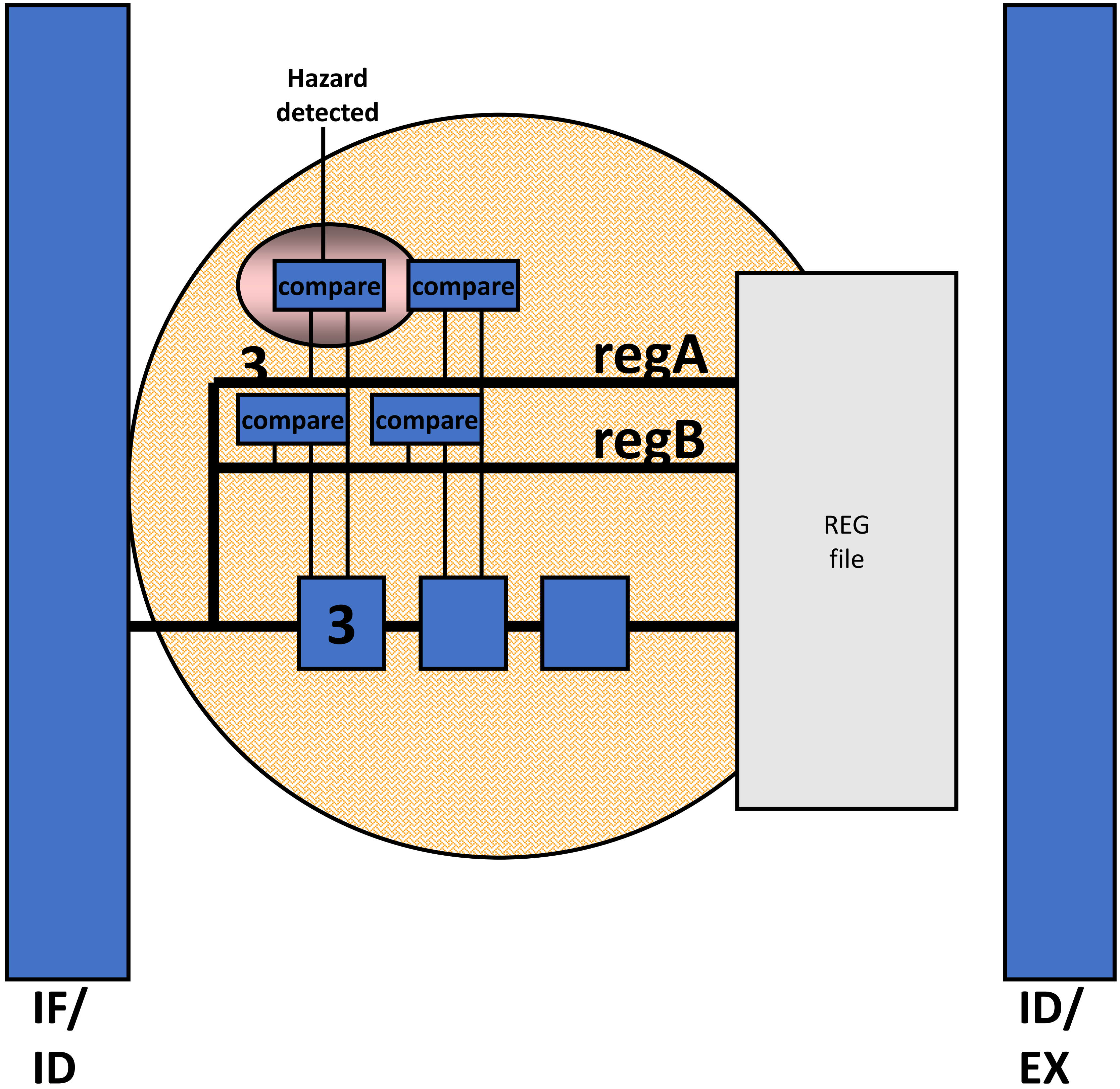
- Detect:
  - Compare regA with previous DestRegs
    - 3 bit operand fields
  - Compare regB with previous DestRegs
    - 3 bit operand fields
- Stall:
  - Keep current instructions in fetch and decode
  - Pass a noop to execute

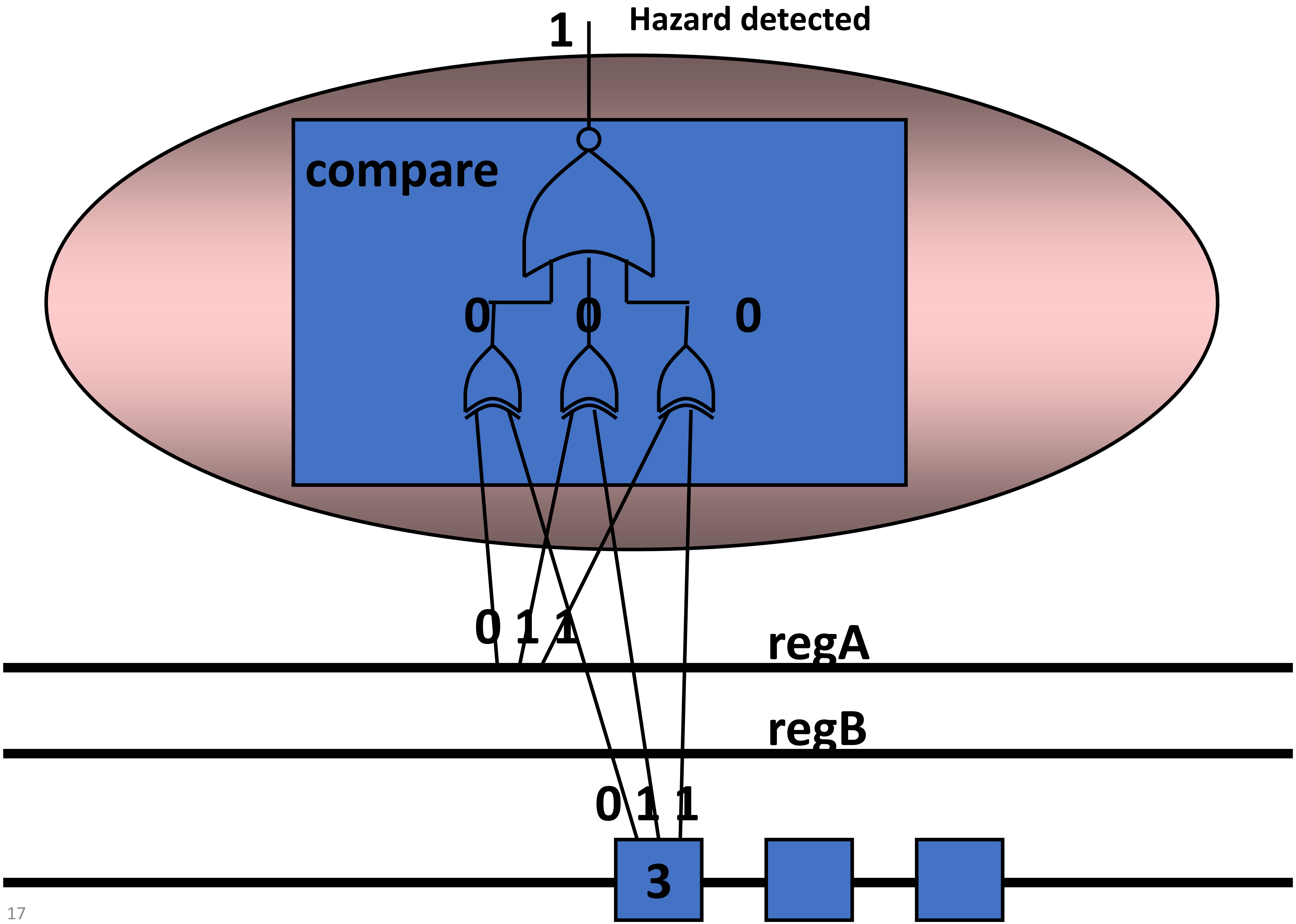
- How do we modify the pipeline to do this?

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




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

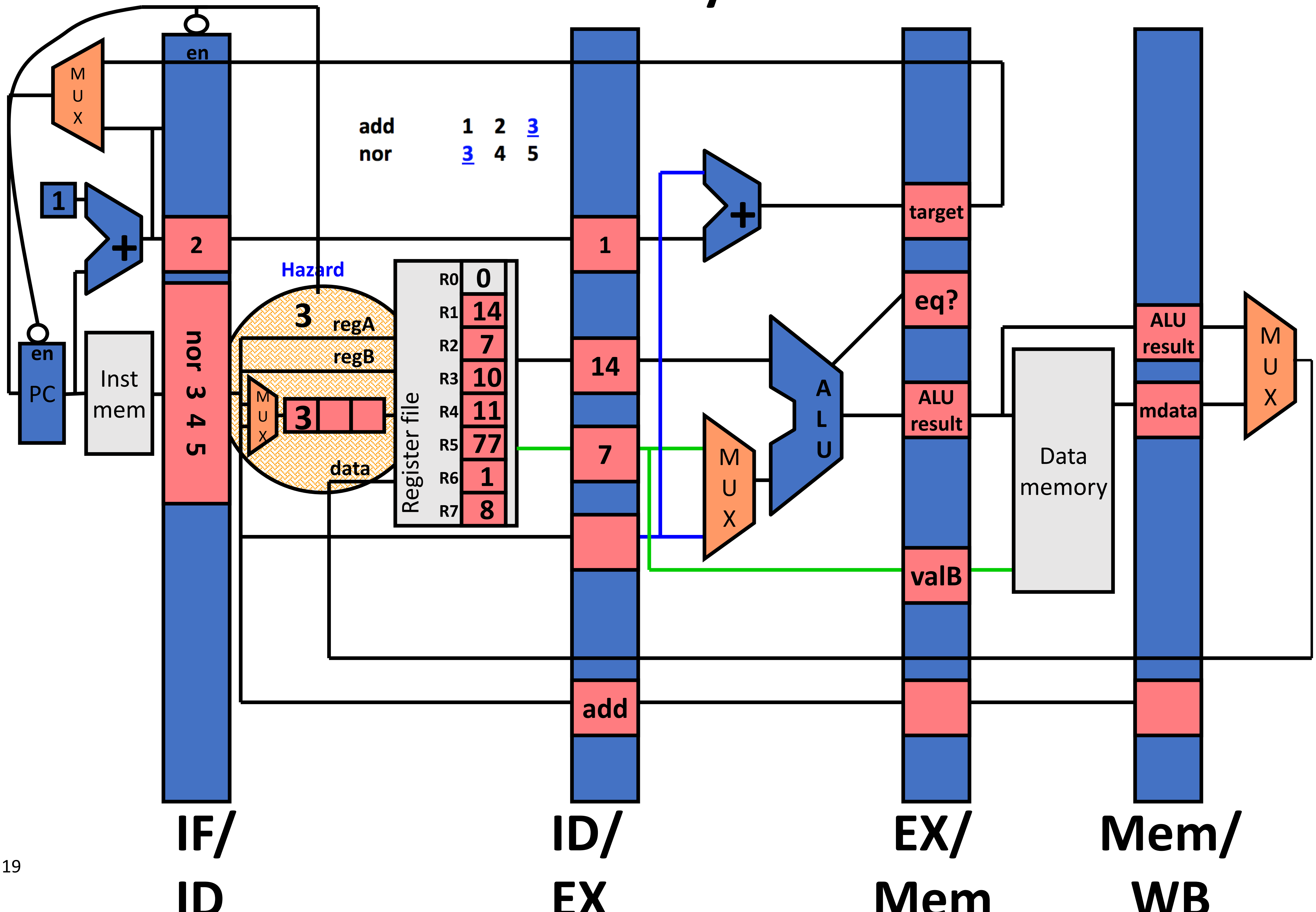
Example

- Let's run this program with a data hazard through our 5-stage pipeline
  add 1 2 <u>3</u>
  nor <u>3</u> 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

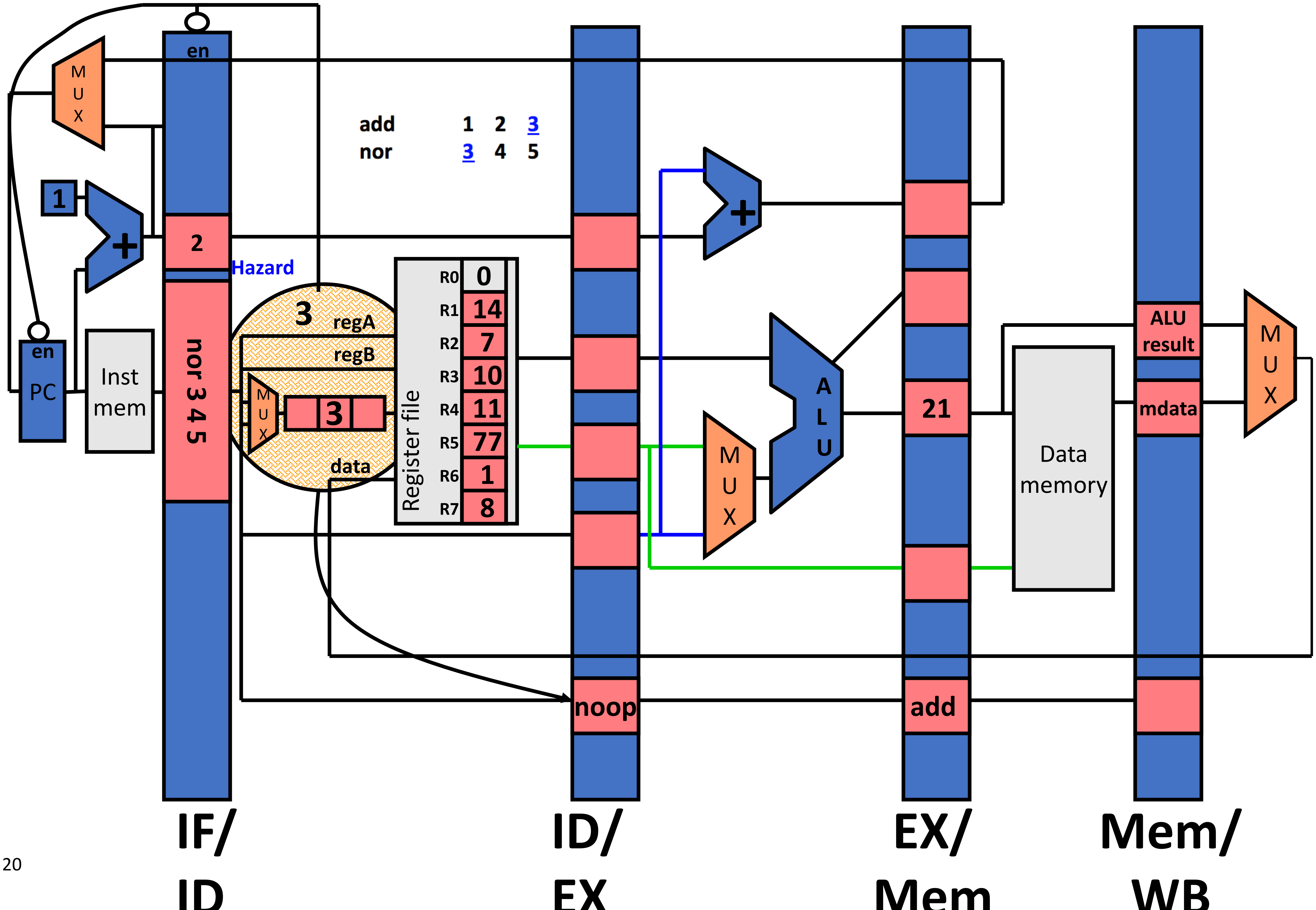
Time:	1	2	3
add 1 2 3	IF	ID	EX
nor 3 4 5		IF	ID

Hazard!

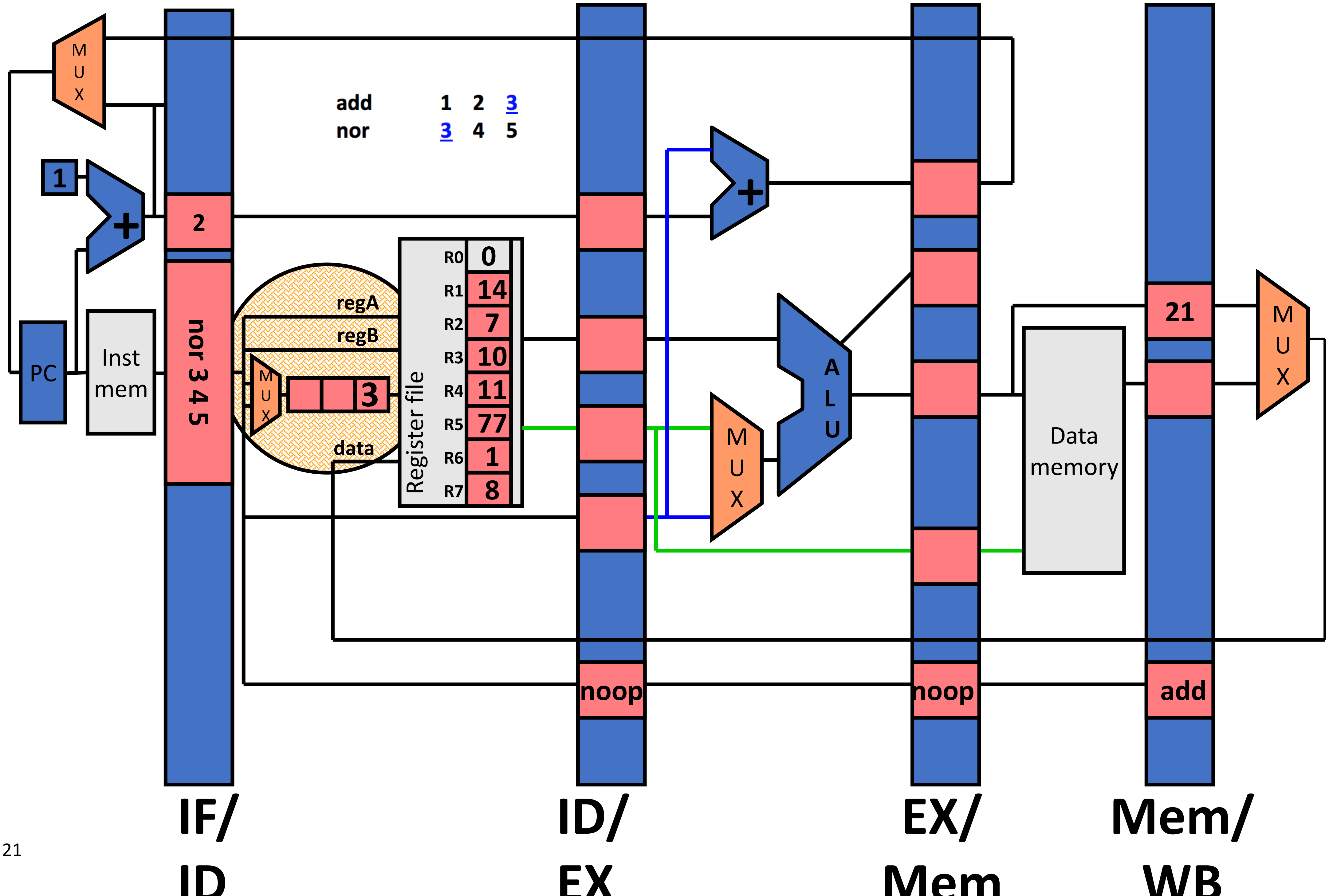
First half of cycle 3



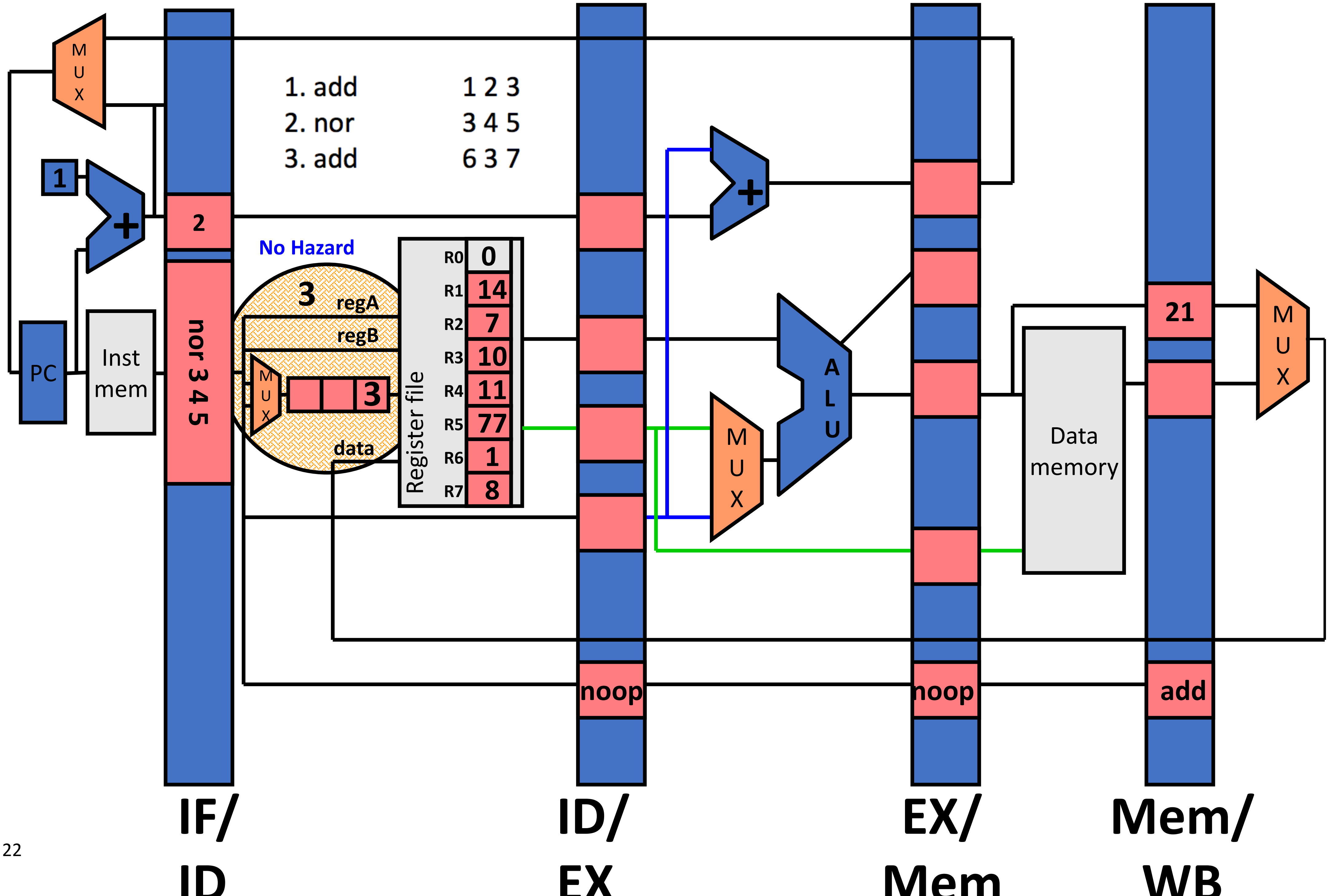
First half of cycle 4



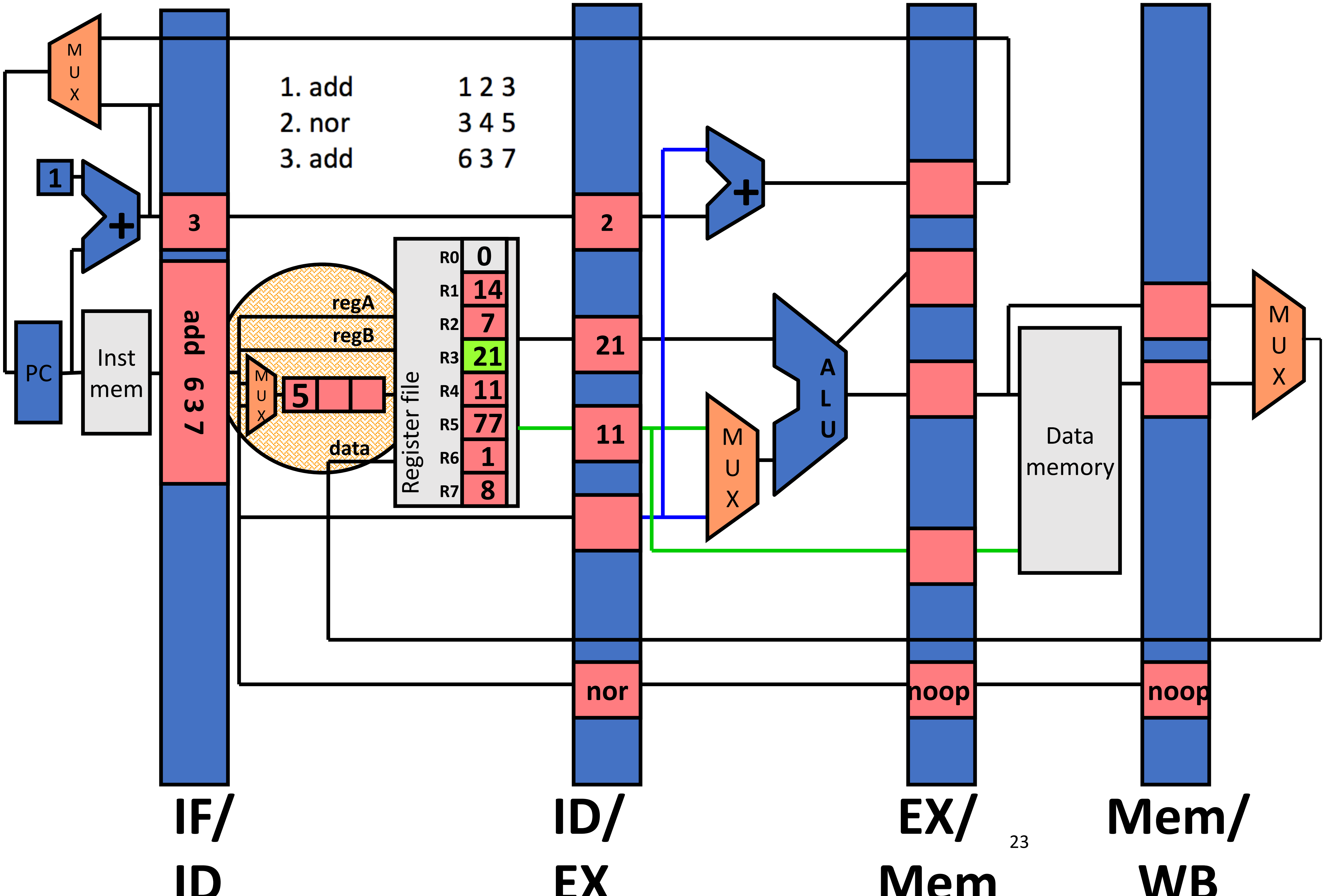
End of cycle 4



First half of cycle 5



End of cycle 5



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

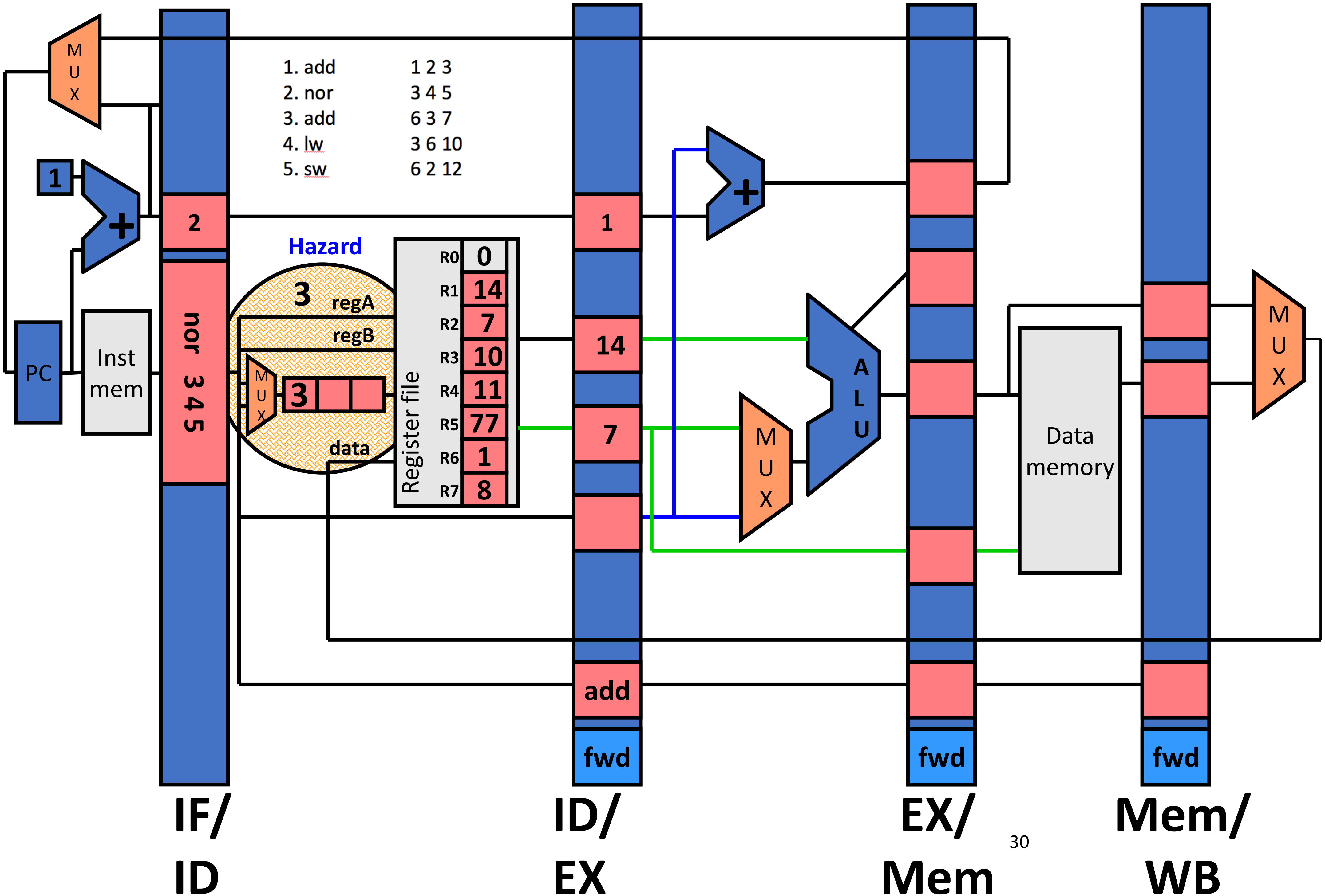
- Detect: same as detect and stall
  - Except that all 4 hazards have to be treated differently
    - i.e., you can’t logical-OR the 4 hazard signals
- Forward:
  - New bypass datapaths route computed data to where it is needed
  - New MUX and control to pick the right data
- Beware: Stalling may still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

Forwarding example

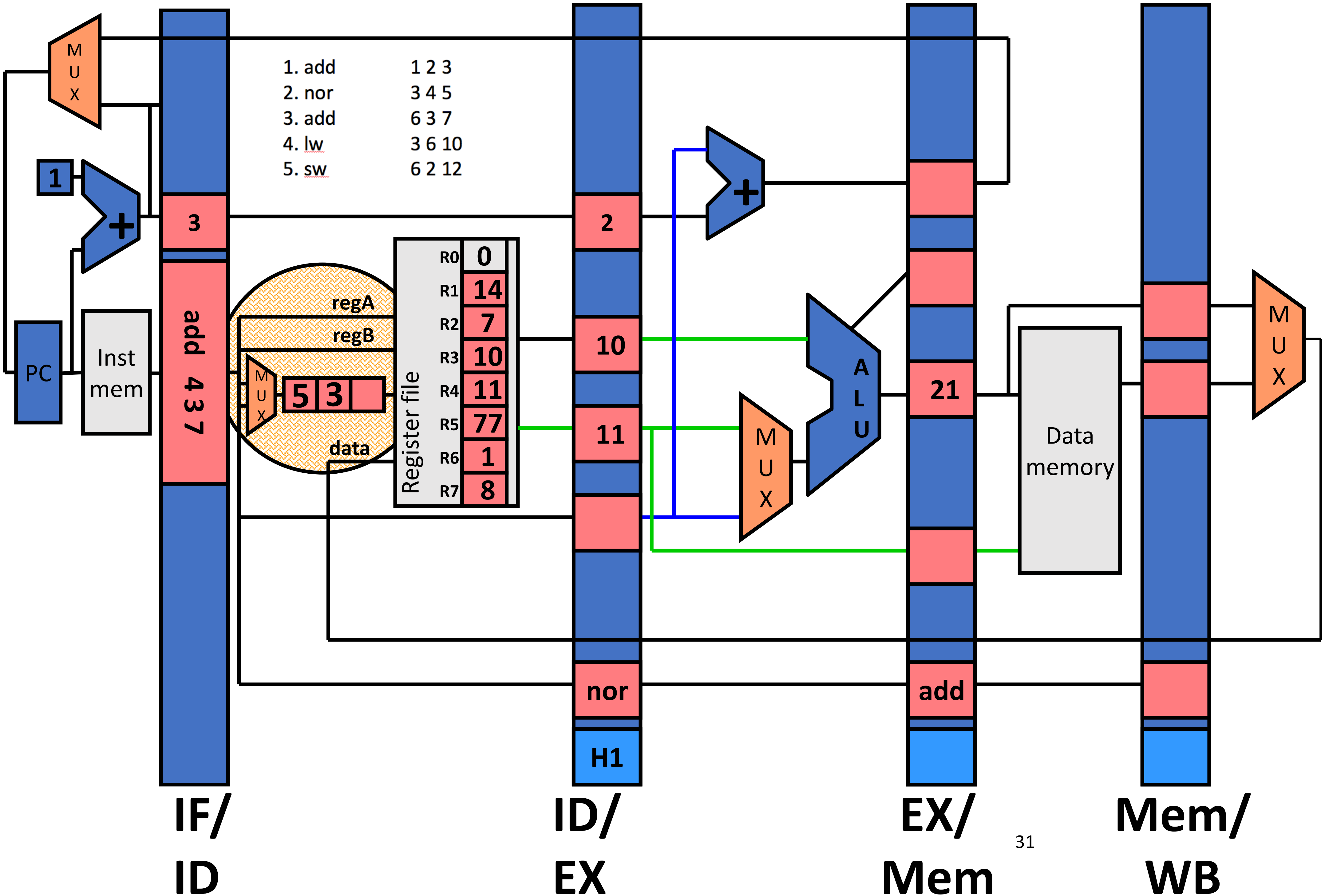
- We will use this program for the next example (same as last pipeline diagram example)

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

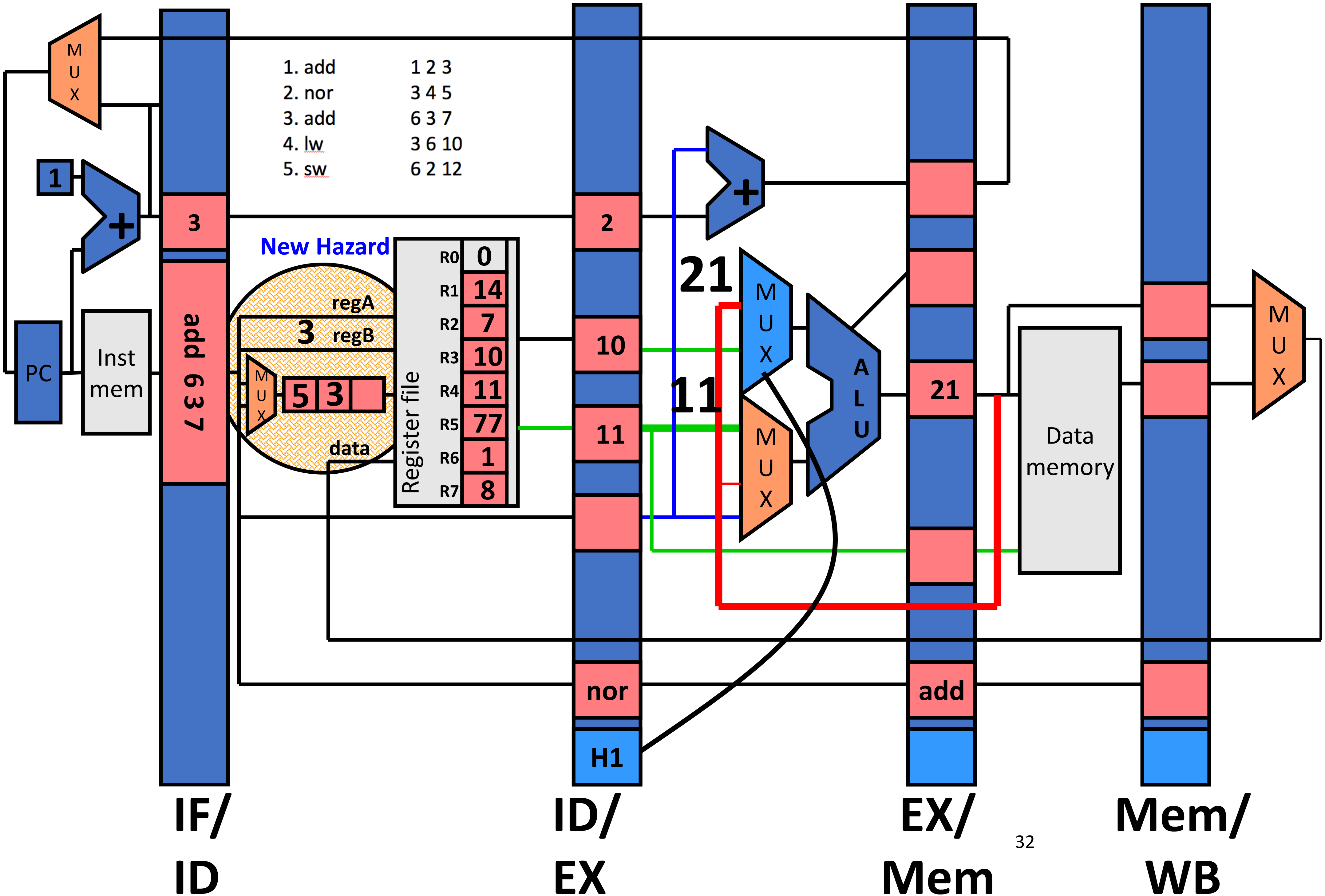
First half of cycle 3



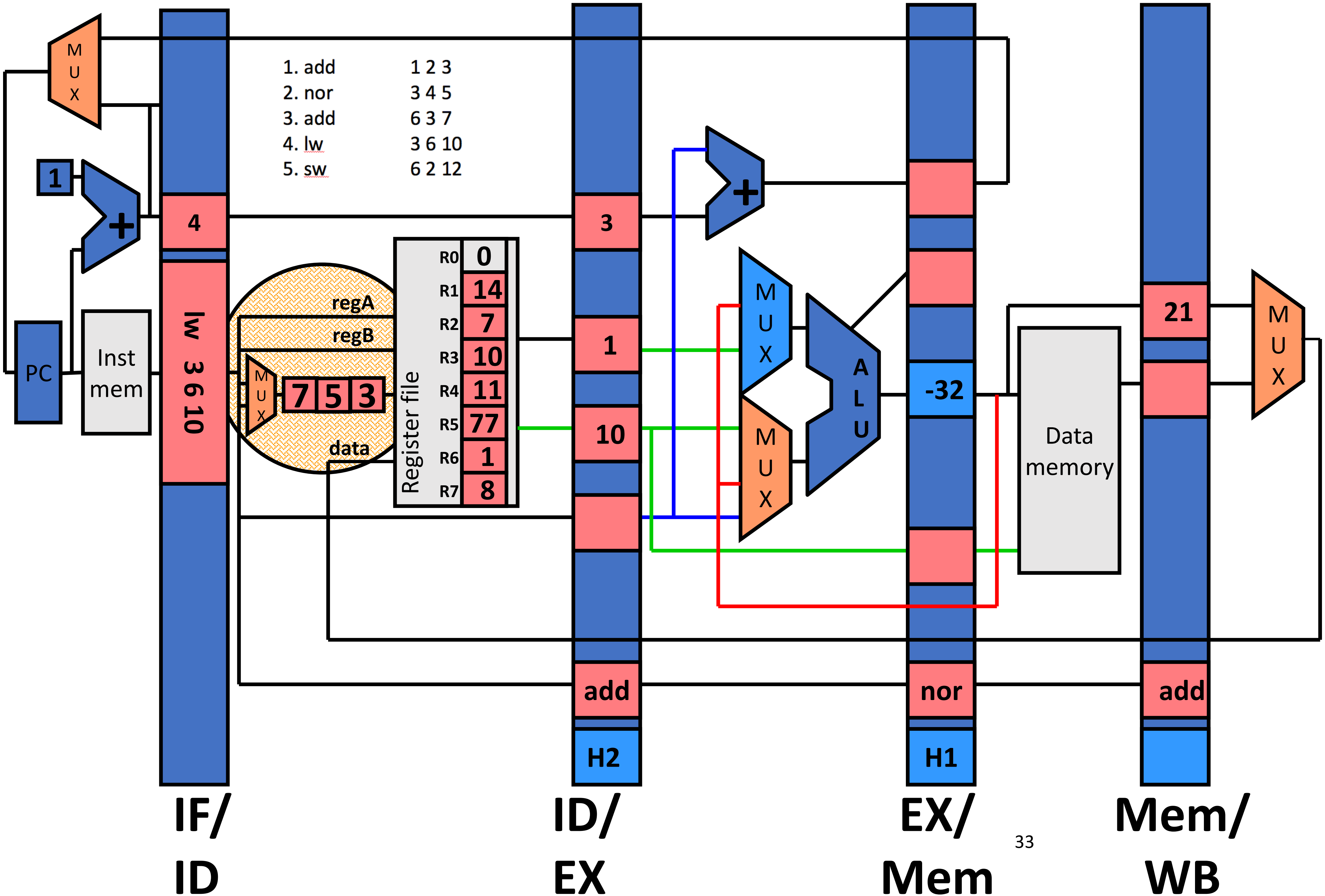
End of cycle 3



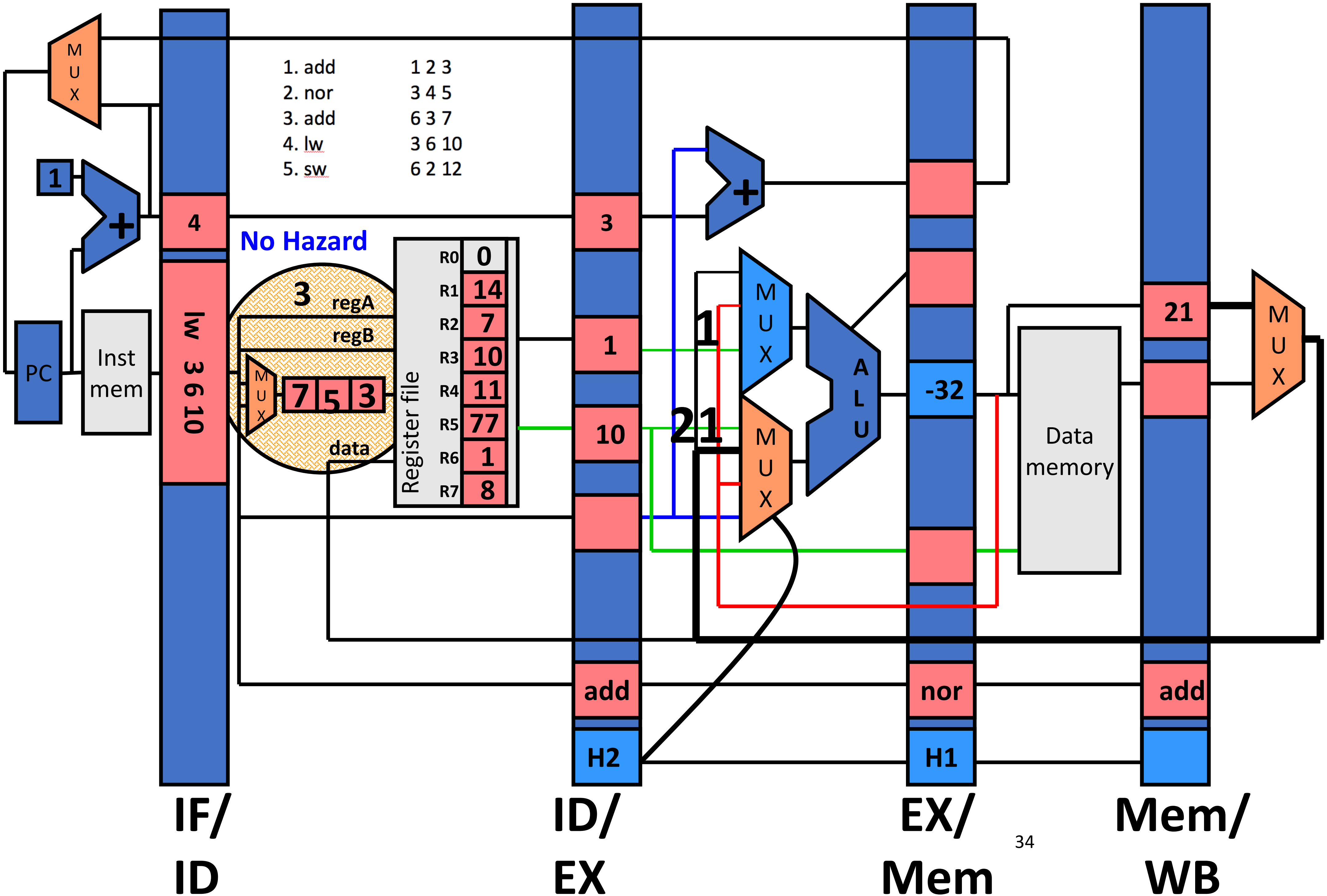
First half of cycle 4



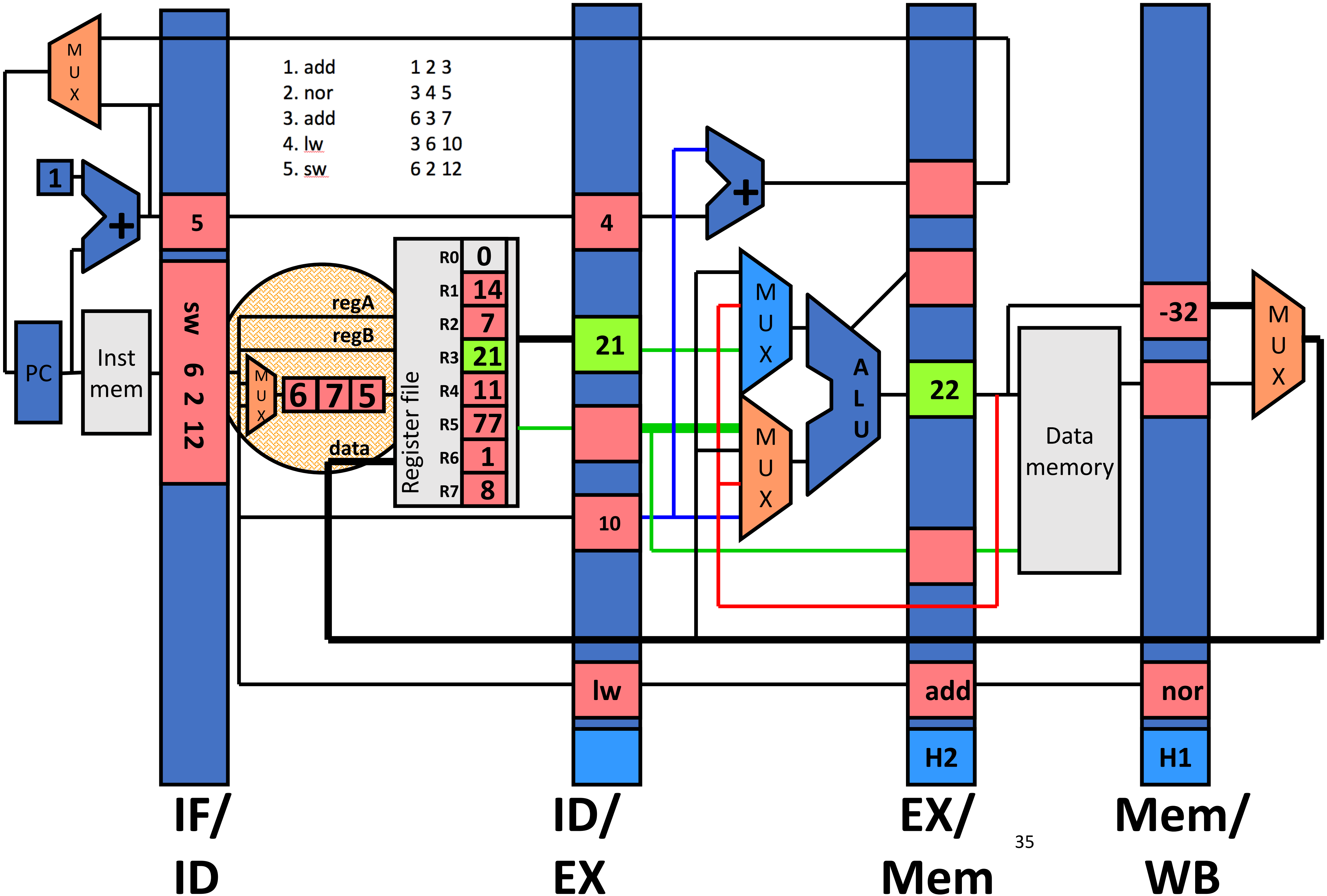
End of cycle 4



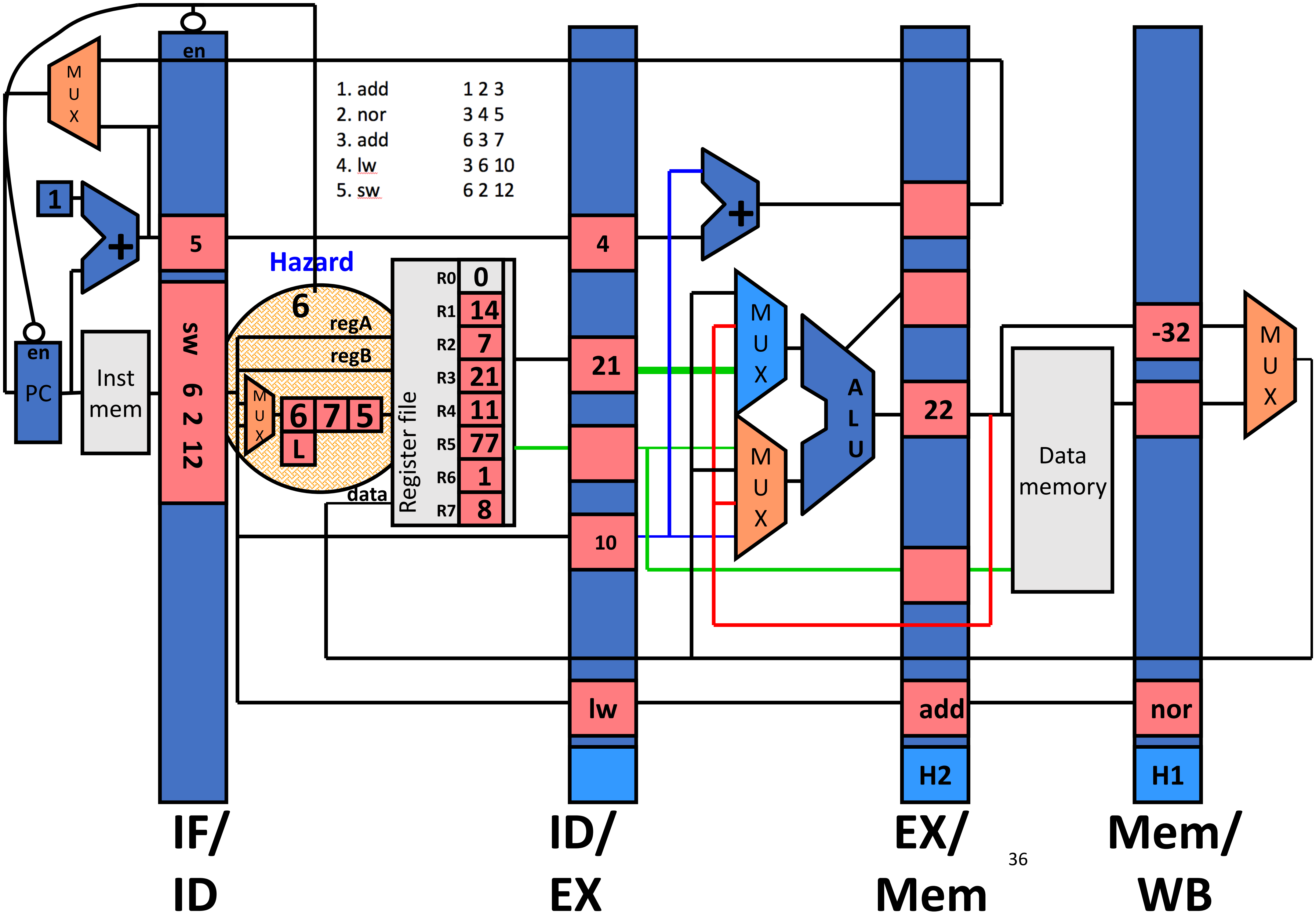
First half of cycle 5



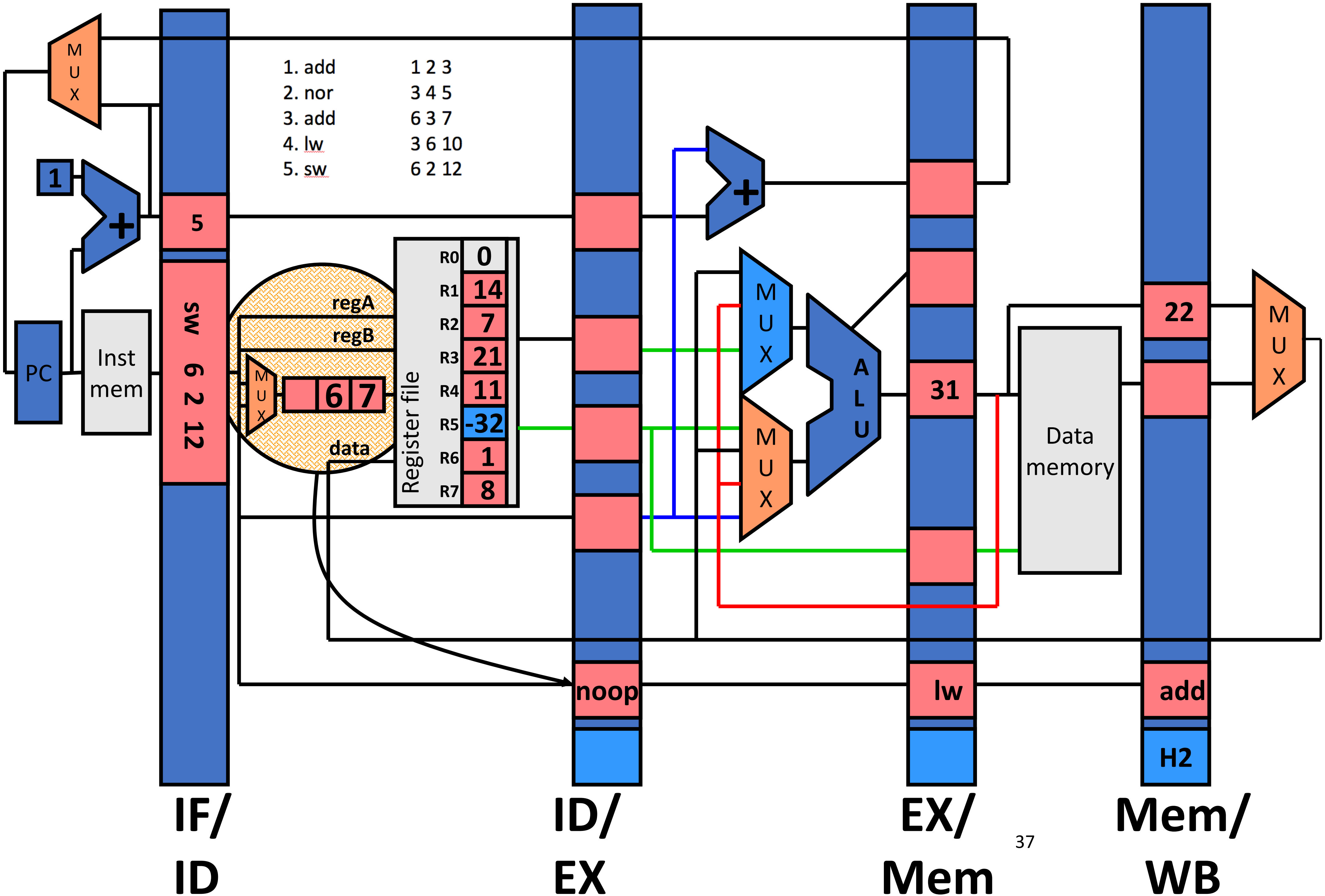
End of cycle 5



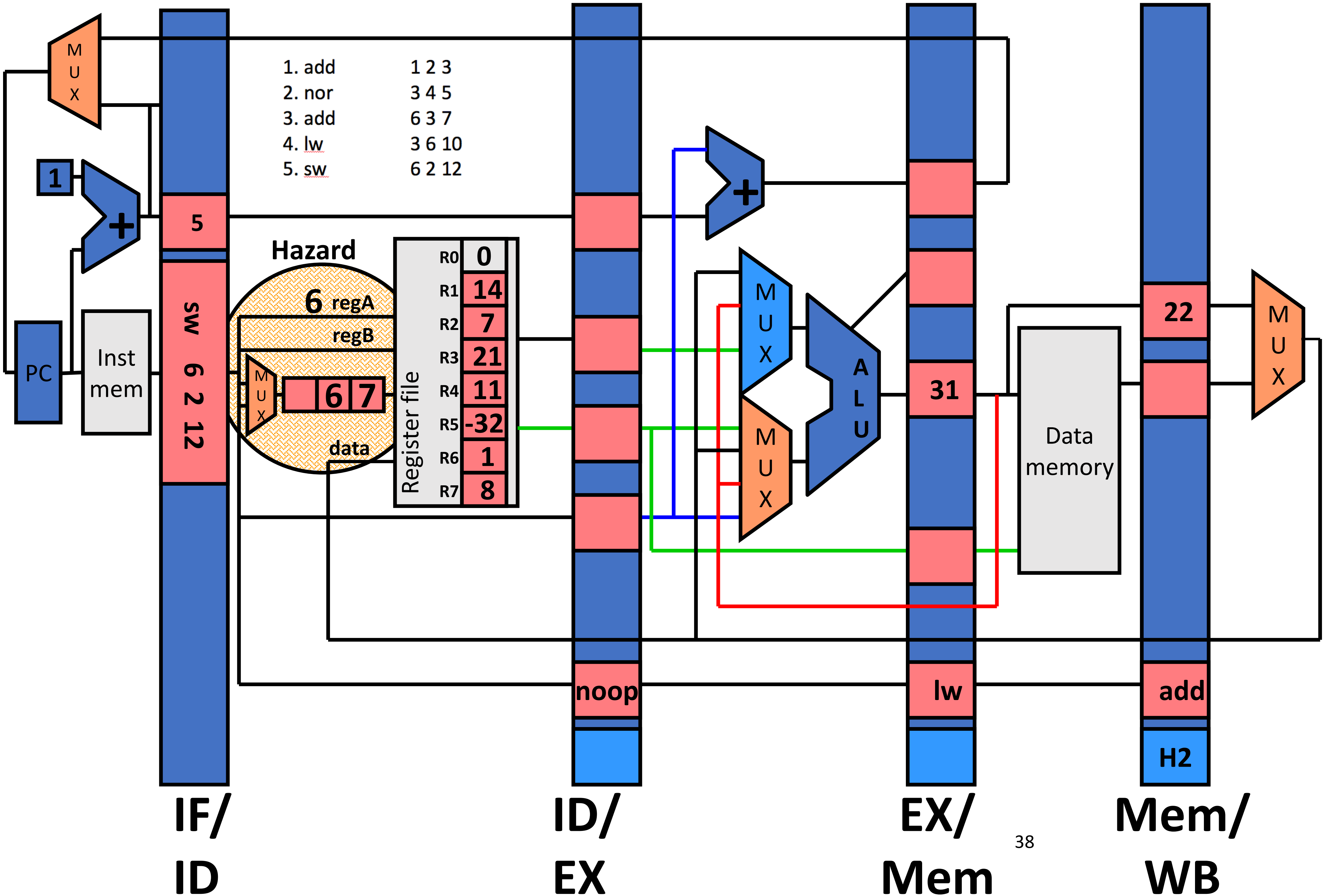
First half of cycle 6



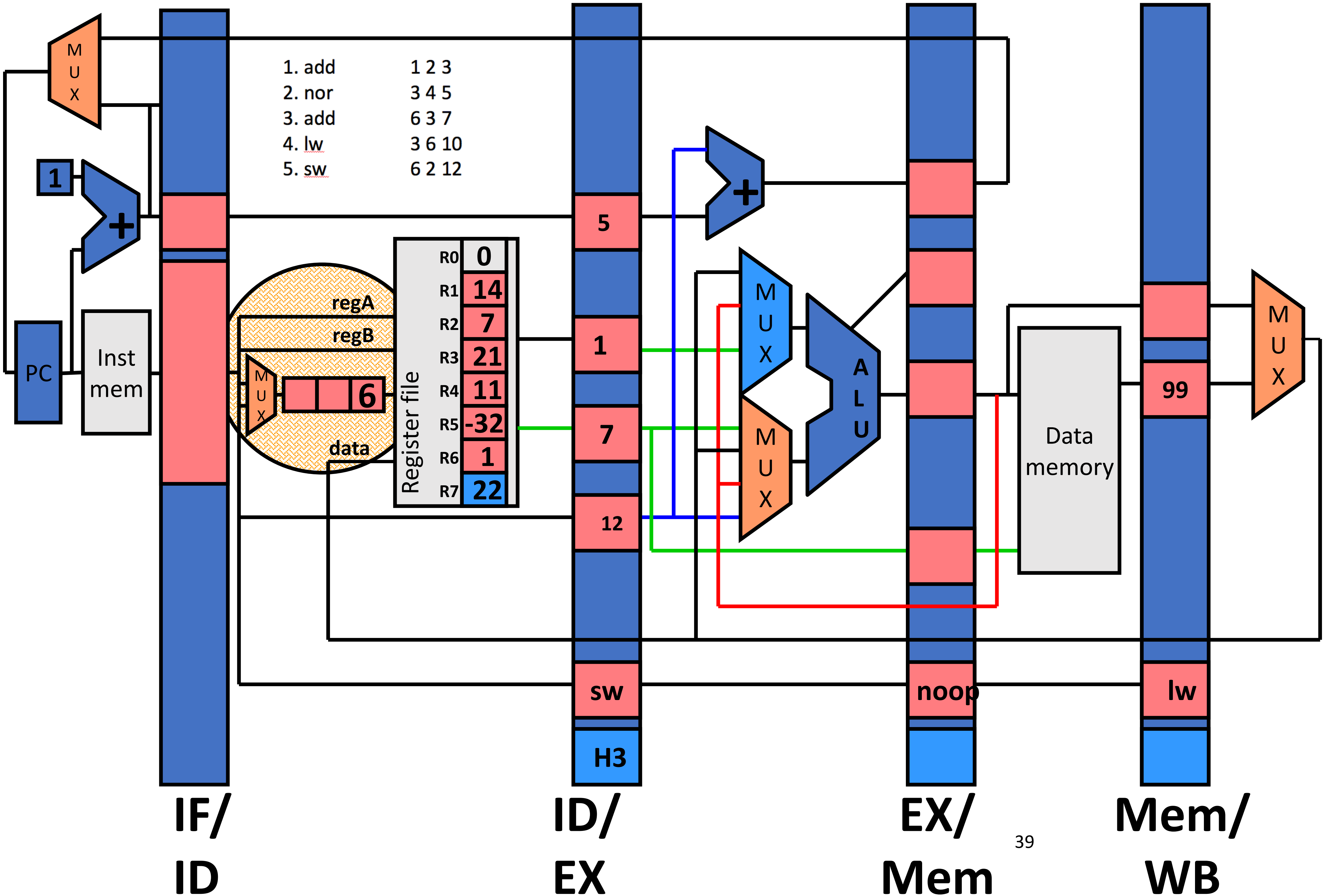
End of cycle 6



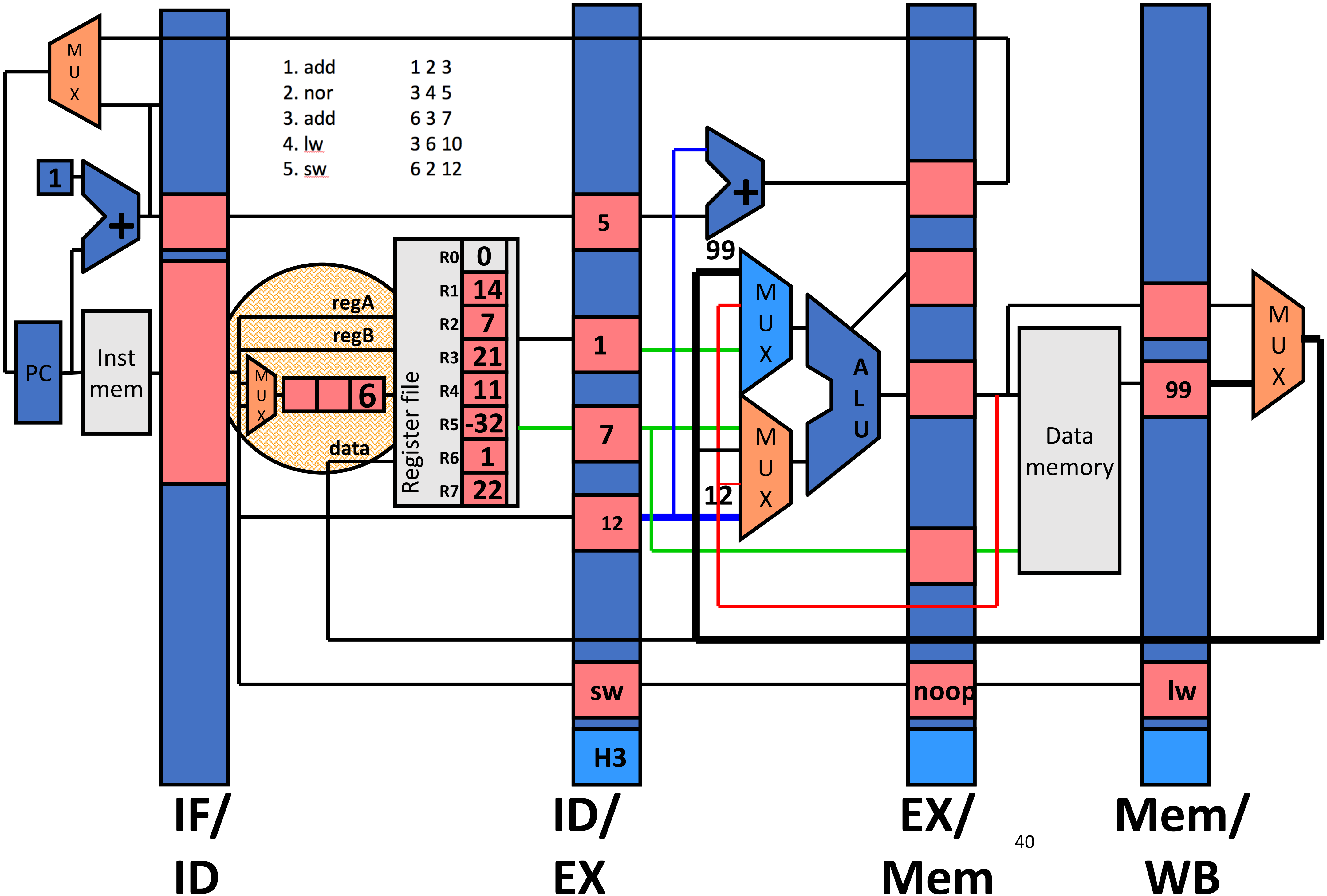
First half of cycle 7



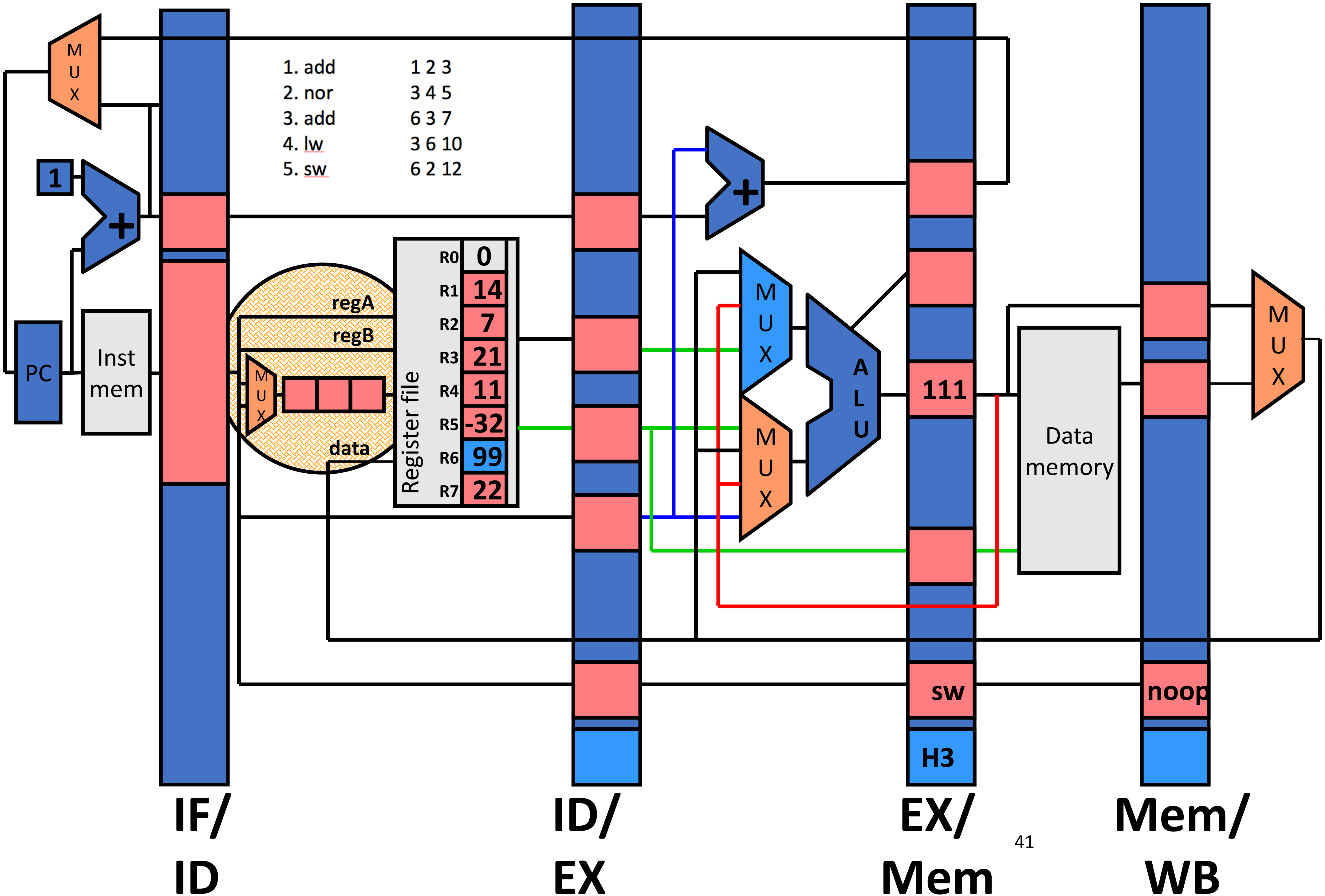
End of cycle 7



First half of cycle 8



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			

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

time

beq	**fetch**	**decode**	**execute**	**memory**	**writeback**
add		**fetch**	**decode**	**execute**	

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

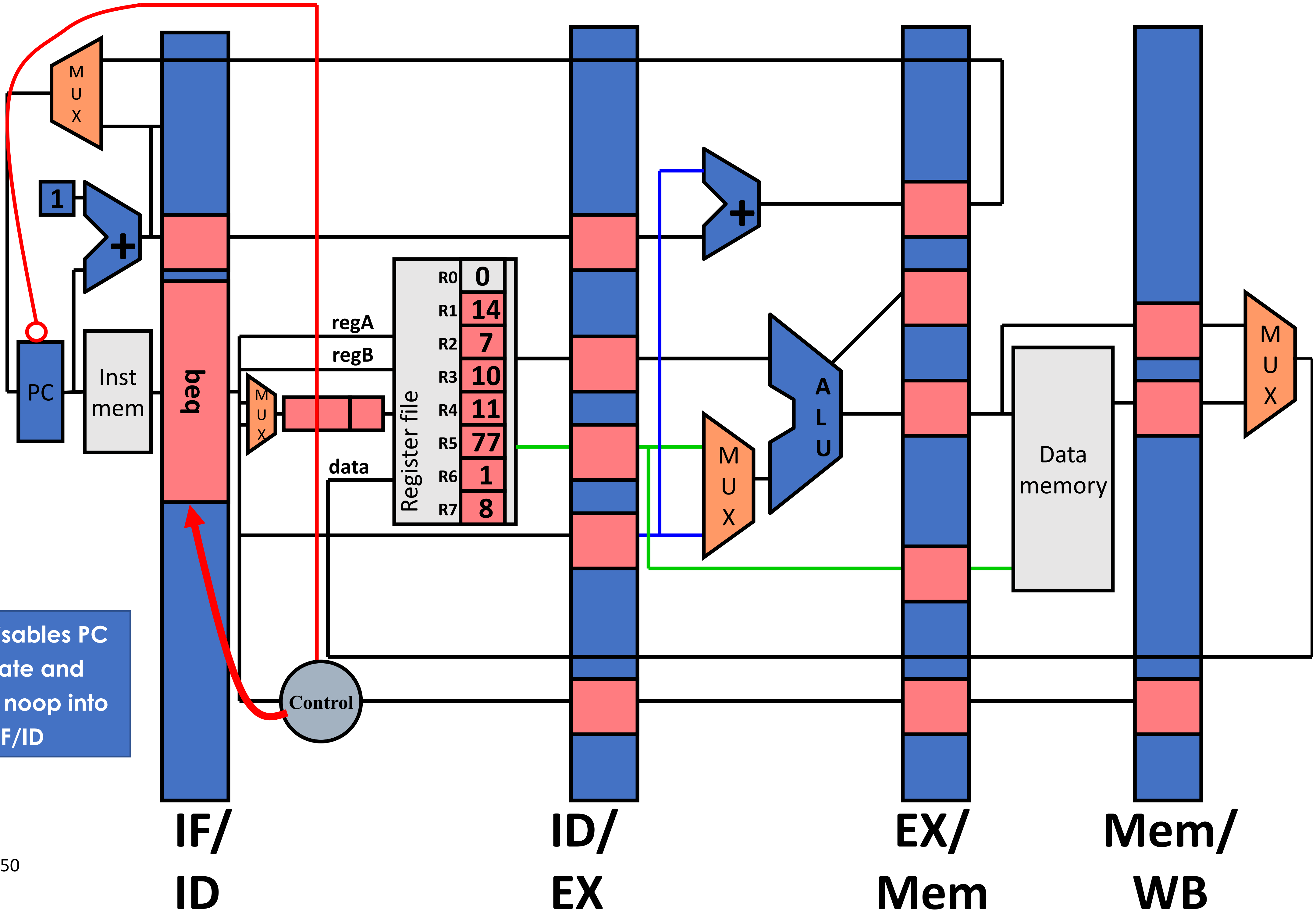
1. Avoid
    - Make sure there are no hazards in code
2. Detect and stall
    - Delay fetch until branch resolved
3. Speculate and squash-if-wrong
    - Guess outcome of branch
    - Fetch instructions assuming we're right
    - Stop them if they shouldn't have been executed

Avoiding Control Hazards

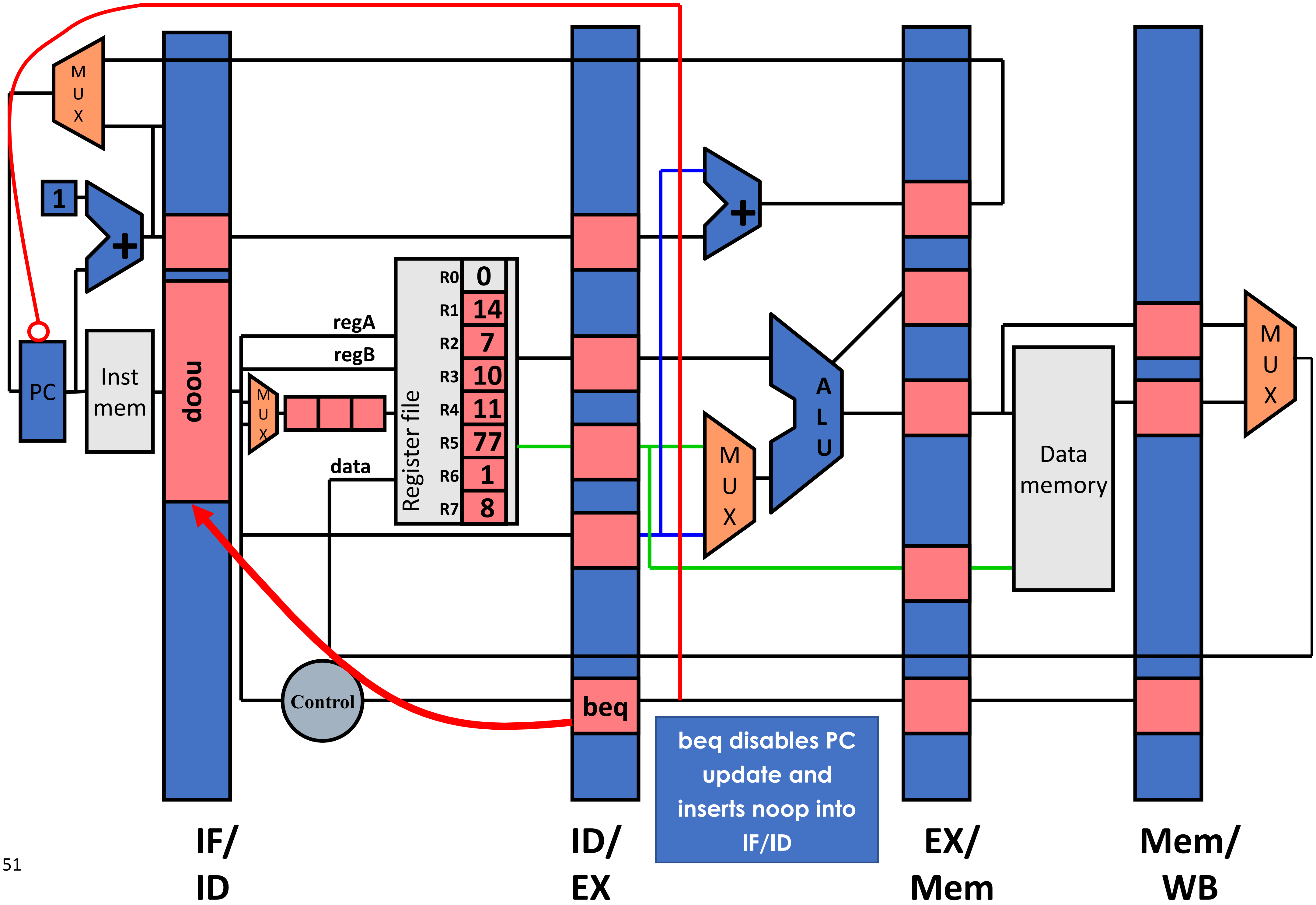
- Don't have branch instructions!
  - Possible, but not practical
  - ARM offers **predicated** instructions (instructions that throw away result if some condition is not met)
    - Allows replacement of if/else conditions
    - Hard to use for everything
    - Not covered more in this class

Detect and Stall

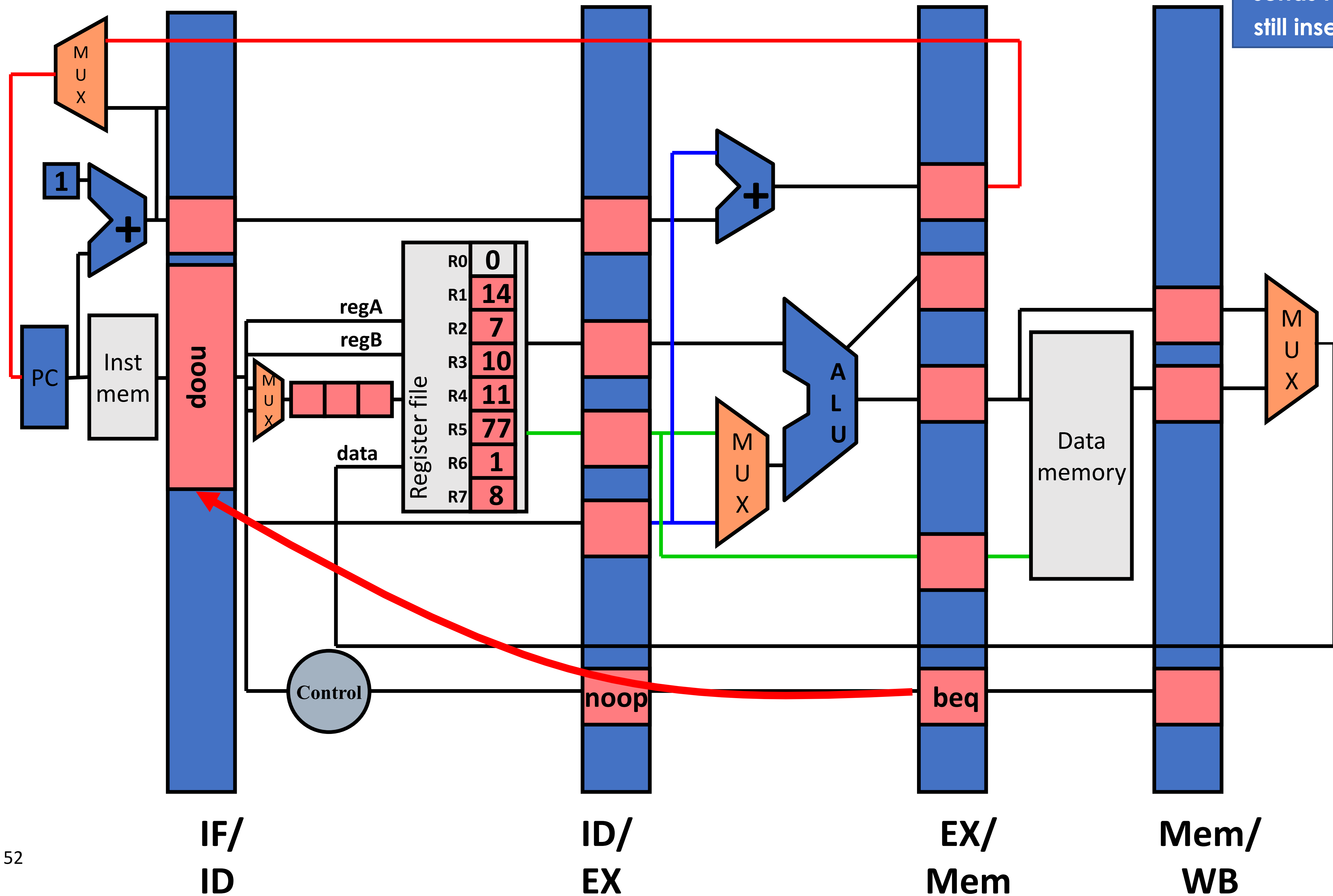
- Detection
  - Wait until decode
  - Check if opcode == beq or jalr
- Stall
  - Keep current instruction in fetch
  - Insert noops
  - Pass noop to decode stage, not execute!



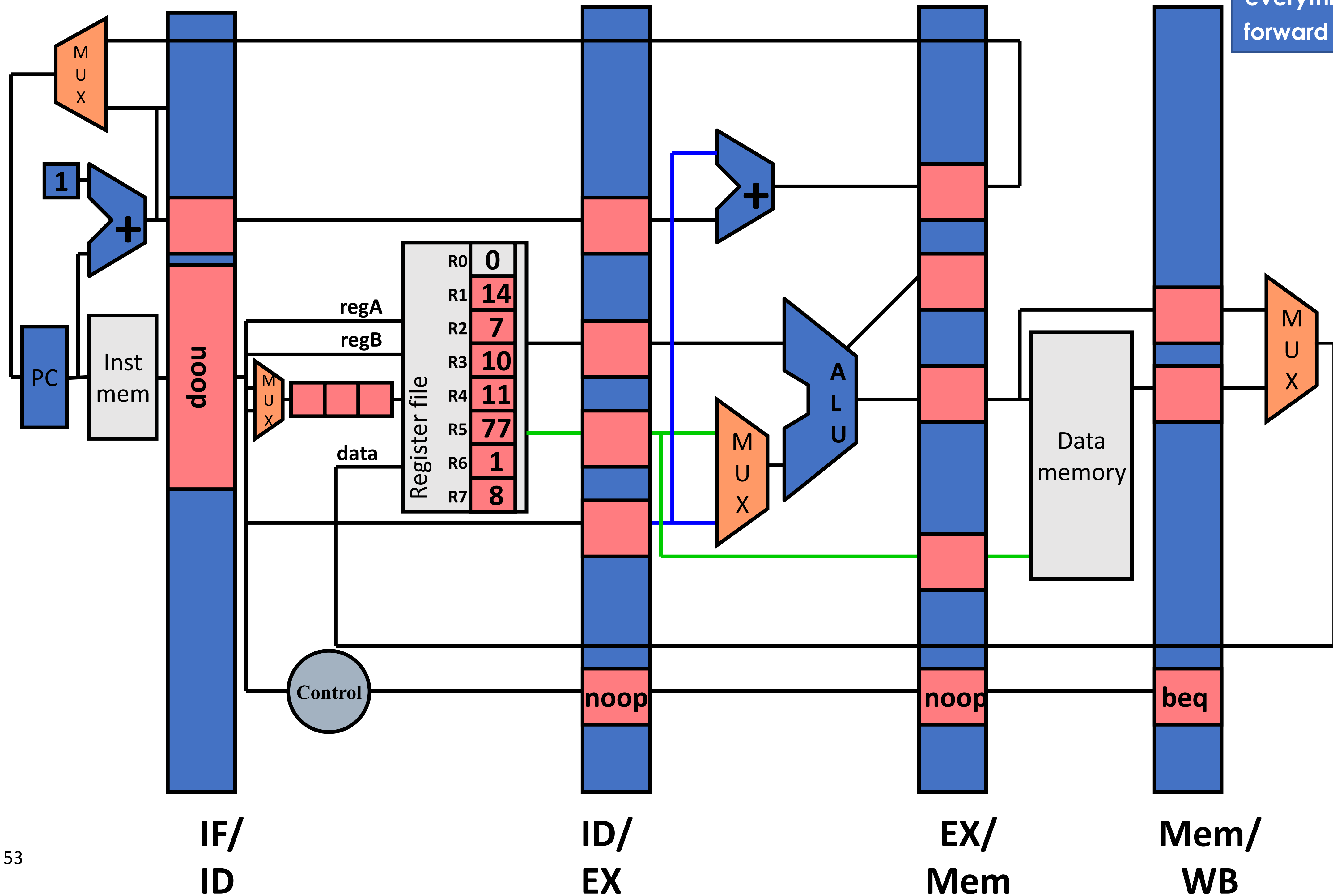
beq disables PC update and inserts noop into IF/ID

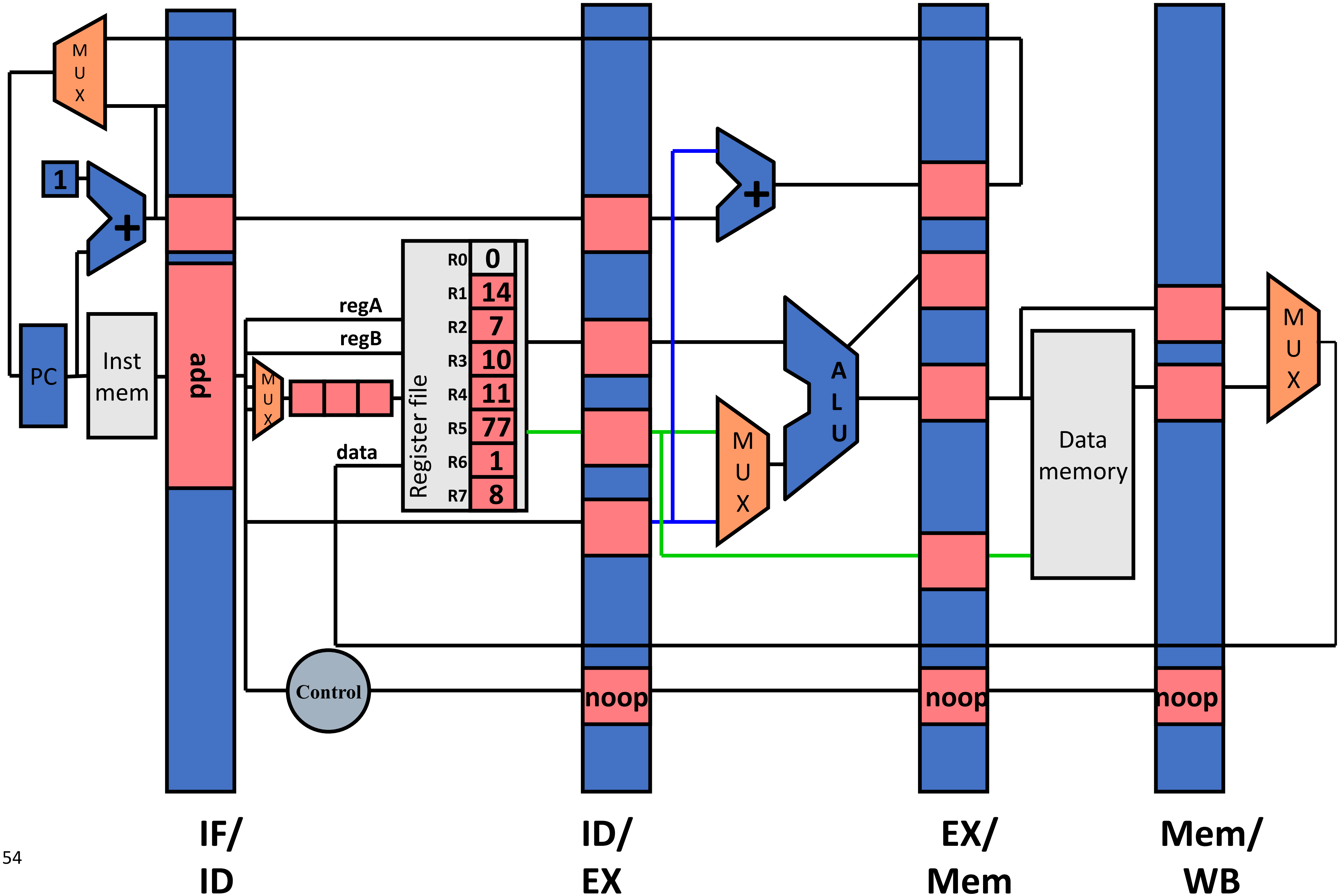

MUX
1
+
PC
Inst mem
noop
regA
regB
data
MUX
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
Control
beq
+
MUX
ALU
Data memory
MUX
beq disables PC update and inserts noop into IF/ID
IF/ID
ID/EX
EX/Mem
Mem/WB

beq DOESN'T disable PC and sends target, but still inserts NOOP



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




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

Control Hazards

beq 1 1 10
add 3 4 5

time

beq	**fetch**	**decode**	**execute**	**<u>memory</u>**	**writeback**
add		**fetch**	**<u>fetch</u>**	**<u>fetch</u>**	**fetch**
					or
Target:					**fetch**

Problems with Detect and Stall

- CPI increases every time a branch is detected!
- Is that necessary? Not always!
  - Branch not always taken
  - Let's assume it is NOT taken…
    - In this case, we can ignore the beq (treat it like a noop)
    - Keep fetching PC + 1
  - What if we're wrong?
  - OK, as long as we do not COMPLETE any instruction we mistakenly execute
  - I.e. DON'T write values to register file or memory

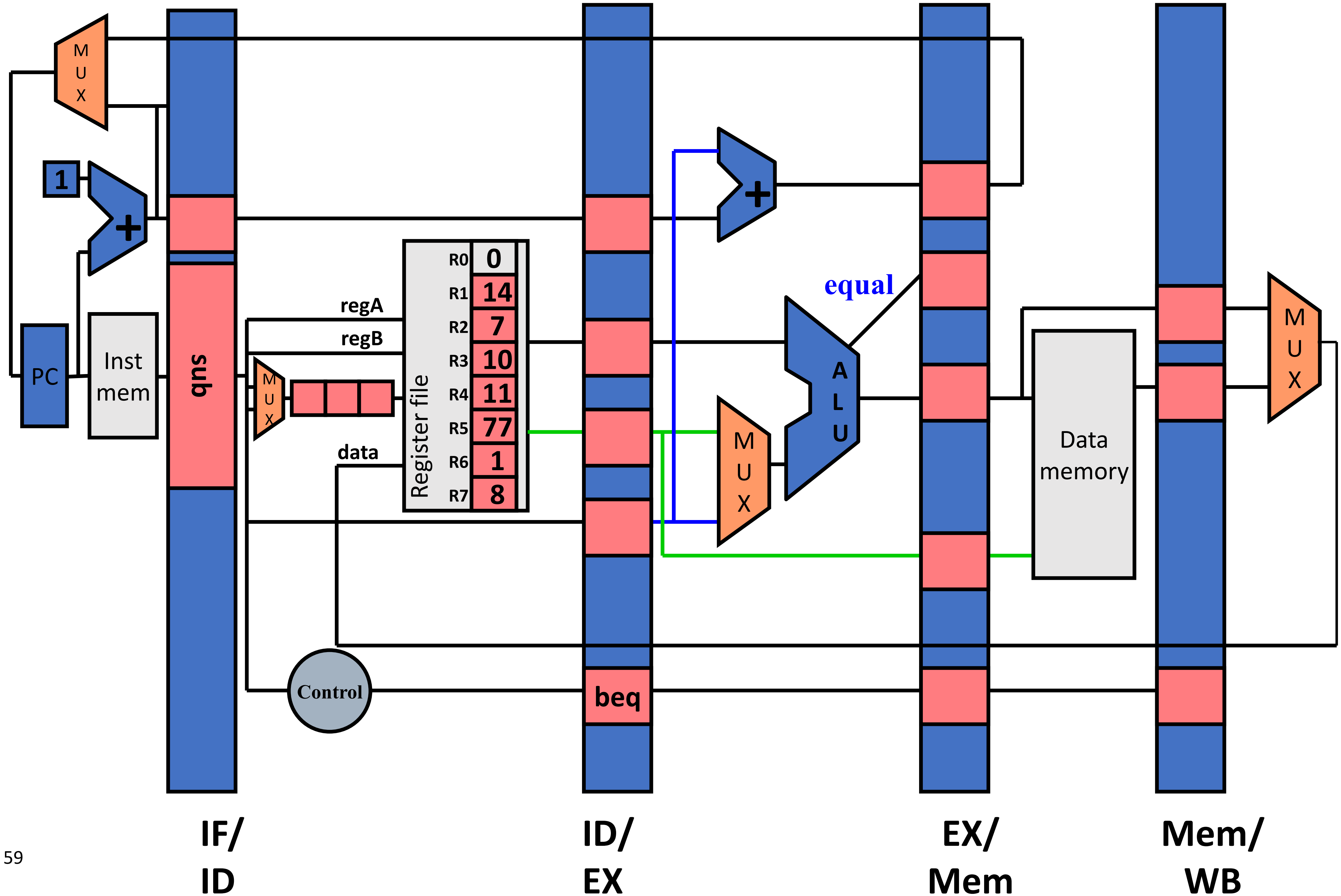
Agenda

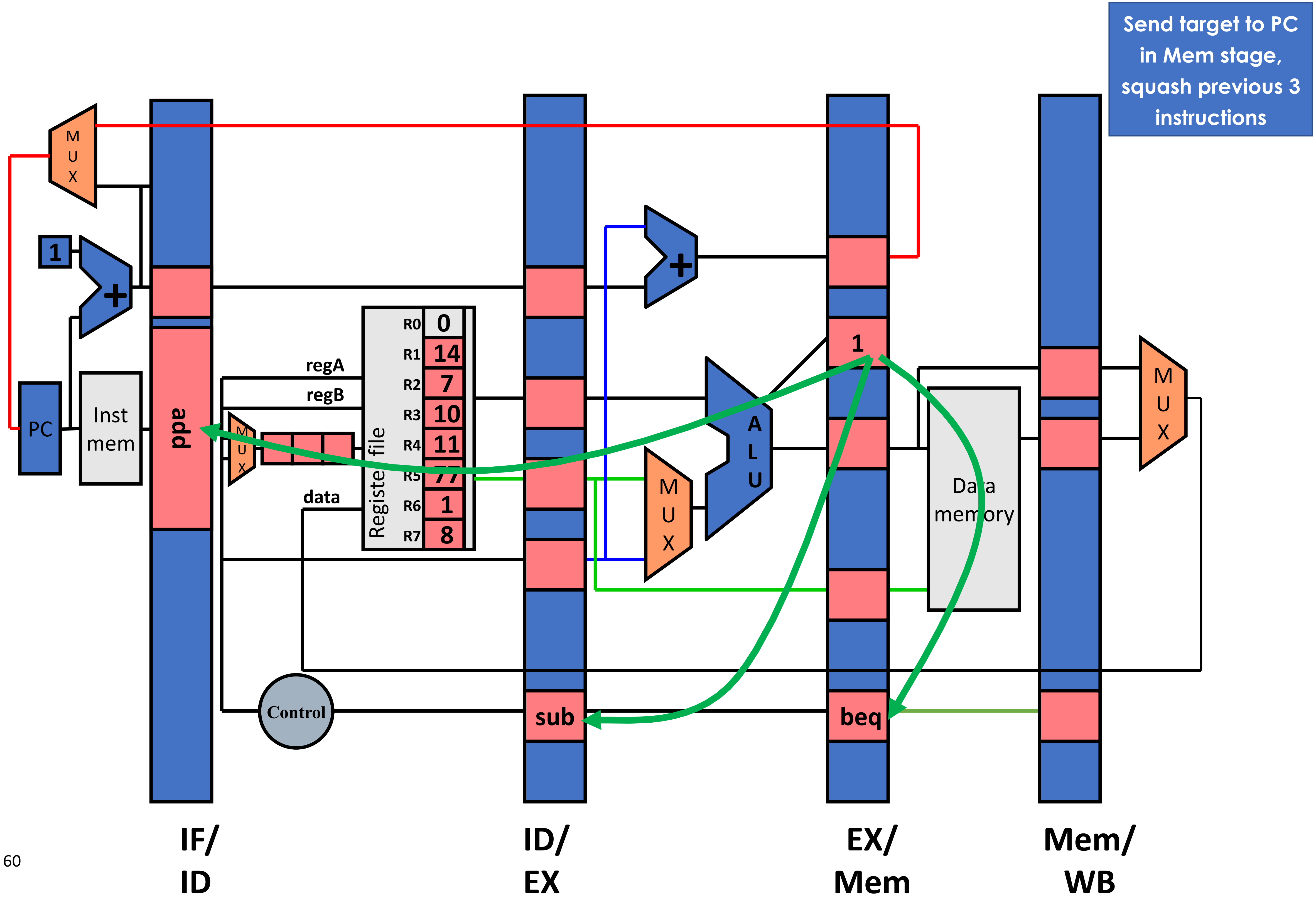
- Control Hazards and Basic Approaches
- Detect-and-Stall
- **Speculate-and-Squash**
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

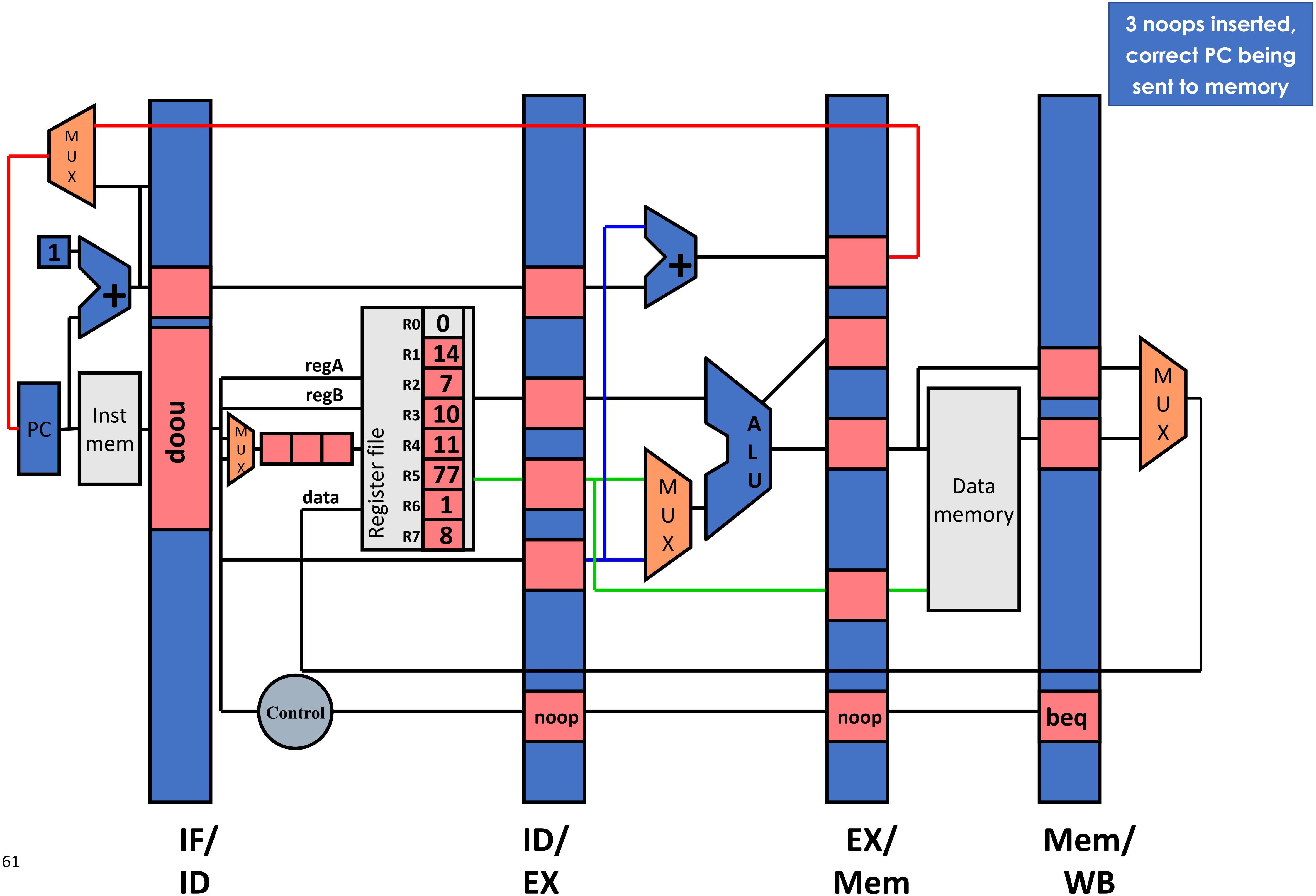
Speculate and Squash

- Speculate: assume not equal
  - Keep fetching from PC+1 until we know that the branch is really taken
- Squash: stop bad instructions if taken
  - Send a noop to Decode, Execute, and Memory
  - Sent target address to PC

Resolve branch in execute stage




Send target to PC in Mem stage, squash previous 3 instructions
MUX
1
+
PC
Inst mem
add
regA
regB
data
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
Control
sub
+
MUX
ALU
1
beq
Data memory
MUX
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB


3 noops inserted, correct PC being sent to memory
M U X
1
+
PC
Inst mem
noop
regA
regB
data
M U X
Register file
R0 0
R1 14
R2 7
R3 10
R4 11
R5 77
R6 1
R7 8
Control
+
M U X
A L U
noop
noop
Data memory
beq
M U X
IF/ ID
ID/ EX
EX/ Mem
Mem/ WB

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- **Exceptions**
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

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 the wrong value 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?

Exceptions

- Exception: when something unexpected happens during program execution.
  - Example: divide by zero.
  - The situation is more complex than the hardware can handle
    - So the hardware branches to a function, an “exception handler” which is code to try to deal with the problem.
- The exact way to set up such an exception handler will vary by ISA.
  - With C on x86 you would use <signal.h> functions to handle the “SIGFPE” signal.
  - There is a pretty good Hackaday article on this if you want to learn more.

Exceptions and Pipelining

- The hardware branches to the “exception handler”
  - This means that any instruction which can “throw” an exception *could* be a branch.
  - Throwing an exception should be rare (“exceptional”)

- So we would treat it much like a branch we predicted as “not taken”
  - Squash instructions behind it and then branch
  - It will introduce stalls, but since it should be rare, we don’t worry about it.
    - “Make the common case fast”.

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- **Practice Performance Problems**
  - **Problem 1**
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

Classic performance problem

- Program with following instruction breakdown:

lw	10%
sw	15%
beq	25%
R-type	50%

- Speculate "always not-taken" and squash. 80% of branches not-taken
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time per instruction if clock frequency is 100MHz?

Classic performance problem

- Program with following instruction breakdown:

lw	10%
sw	15%
beq	25%
R-type	50%

- Speculate “always not-taken” and squash. 80% of branches not-taken
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time per instruction if clock frequency is 100MHz?

CPI = 1 + 0.10 (loads) * 0.20 (load use stall)*1

+ 0.25 (branch) * 0.20 (miss rate)*3

CPI = 1 + 0.02 + 0.15 = 1.17

Time = 1.17 * 10ns =11.7ns per instruction

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - **Problem 2**
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

Classic performance problem (cont.)

- Assume branches are resolved at Execute?
  - What is the CPI?
  - What happens to cycle time?

Classic performance problem (cont.)

- Assume branches are resolved at Execute?
  - What is the CPI?
  - What happens to cycle time?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*1

+ 0.25 (branch) * 0.20 (miss rate)*2

CPI = 1 + 0.02 + 0.1 = 1.12

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - **Problem 3**
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

Performance with deeper pipelines

- Assume the setup of the previous problem.
- What if we have a 10 stage pipeline?
  - Instructions are fetched at stage 1.
  - Register file is read at stage 3.
  - Execution begins at stage 5.
  - Branches are resolved at stage 7.
  - Memory access is complete in stage 9.
- What's the CPI of the program?
- If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

Performance with deeper pipelines

- Assume the setup of the previous problem.
- What if we have a 10 stage pipeline?
  - Instructions are fetched at stage 1.
  - Register file is read at stage 3.
  - Execution begins at stage 5.
  - Branches are resolved at stage 7.
  - Memory access is complete in stage 9.
- What's the CPI of the program?
- If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

  CPI = 1 + 0.10 (loads) *0.20 (load use stall)*4 + 0.25 (branch) * 0.20 (N stalls)*6

  CPI = 1 + 0.08 + 0.30 = 1.38

  Time = 1.38 * 5ns = 6.9 ns per instruction