EECS 470

Further review: Pipeline Hazards and More

Lecture 2 – Winter 2024

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin.

Announcements

- HW1 due 1/18 @10pm (2 days)
  - Use office hours, this isn't trivial.
  - Some review, some stuff to learn on your own.
- Programming assignment 1 due 1/23 (7 days)
  - Hand-in electronically by 10pm
- Should be reading
  - C.1-C.3 (review)
  - 3.1, 3.4-3.5 (new material)
- Get on 470's Piazza site (link on website)

Today

- Touch on performance
- Cover a bit on ISAs
- Pickup where we left off on pipelining

Performance - Key Points

Amdahl's law

$S_{overall} = 1 / ((1-f) + f/S)$

Iron law

$$\frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}$$

Averaging Techniques

Arithmetic Time	Harmonic Rates
$\frac{1}{n}\sum_{i=1}^{n} Time_i$	$\frac{n}{\sum_{i=1}^{n} \frac{1}{Rate_i}}$

Speedup

- While speedup is generally is used to explain the impact of parallel computation, we can also use it to discuss any performance improvement.
  - Keep in mind that if execution time stays the same, speedup is 1.
    - Speedup= $\frac{T_{old}}{T_{new}}$
  - A speedup of 2.0 means that it takes half as long to do something.
  - So 0.5 “speedup” actually means it takes twice as long to do something.
    - Be careful when reading papers, folks sometimes use it incorrectly
    - Sometimes they use %

Instruction Set Architecture

Instruction Set Architecture

"Instruction set architecture (ISA) is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"

IBM introducing 360 in 1964

- IBM 360 is a family of binary-compatible machines with distinct microarchitectures and technologies, ranging from Model 30 *(8-bit datapath, up to 64KB memory)* to Model 70 *(64-bit datapath, 512KB memory)* and later Model 360/91 *(the Tomasulo).*

- IBM 360 replaced 4 concurrent, but incompatible lines of IBM architectures developed over the previous 10 years

ISA: A contract between HW and SW

- ISA (instruction set architecture)
  - A well-defined hardware/software interface

  - The **"contract"** between software and hardware
    - **Functional definition** of operations, modes, and storage locations supported by hardware
    - **Precise description** of how to invoke, and access them

  - No guarantees regarding
    - How operations are implemented
    - Which operations are fast and which are slow and when
    - Which operations take more power and which take less

Components of an ISA

- Programmer-visible states
  - Program counter, general purpose registers, memory, control registers
- Programmer-visible behaviors (state transitions)
  - What to do, when to do it

Example "register-transfer-level" description of an instruction

```
if imem[pc]=="add rd, rs, rt"
then
        pc ⇐ pc+1
        gpr[rd]=gpr[rs]+grp[rt]
```

- A binary encoding

ISAs last 25+ years (because of SW cost)…

…be careful what goes in

RISC vs CISC

- Recall "Iron" law:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- CISC (Complex Instruction Set Computing)
  - Improve "instructions/program" with "complex" instructions
  - Easy for assembly-level programmers, good code density
- RISC (Reduced Instruction Set Computing)
  - Improve "cycles/instruction" with many single-cycle instructions
  - Increases "instruction/program", but hopefully not as much
    - Help from smart compiler
  - Perhaps improve clock cycle time (seconds/cycle)
    - via aggressive implementation allowed by simpler instructions

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Implementability**
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4,…

Typical Instructions (Opcodes)

Type	Example Instruction
Arithmetic and logical	and, add
Data transfer	move, load
Control	branch, jump, call, return
System	trap, rett
Floating point	add, mul, div, sqrt
Decimal	addd, convert
String	move, compare

What operations are necessary? *{sub, ld & st, conditional br.}*

What is the minimum complete ISA for a von Neuman machine?

Too little or too simple → not expressive enough

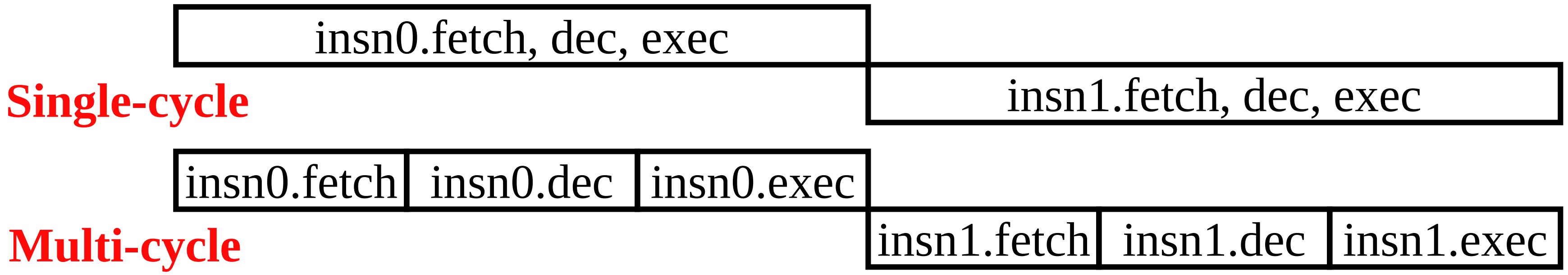
- difficult to program (by hand)
- programs tend to be bigger

Too much or too complex → most of it won’t be used

- too much “baggage” for implementation.
- difficult choices during compiler optimization

Basic Pipelining

Before there was pipelining…



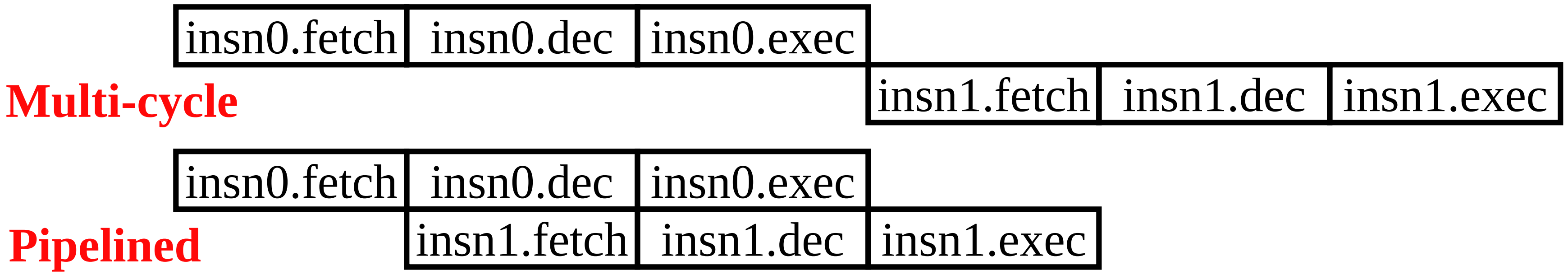
Basic **datapath**: fetch, decode, execute

- **Single-cycle control**: hardwired
  - + Low CPI (1)
  - – Long clock period (to accommodate slowest instruction)
- **Multi-cycle control**: micro-programmed
  - + Short clock period
  - – High CPI

Can we have both low CPI and short clock period?

- Not if datapath executes only one instruction at a time
- No good way to make a single instruction go faster

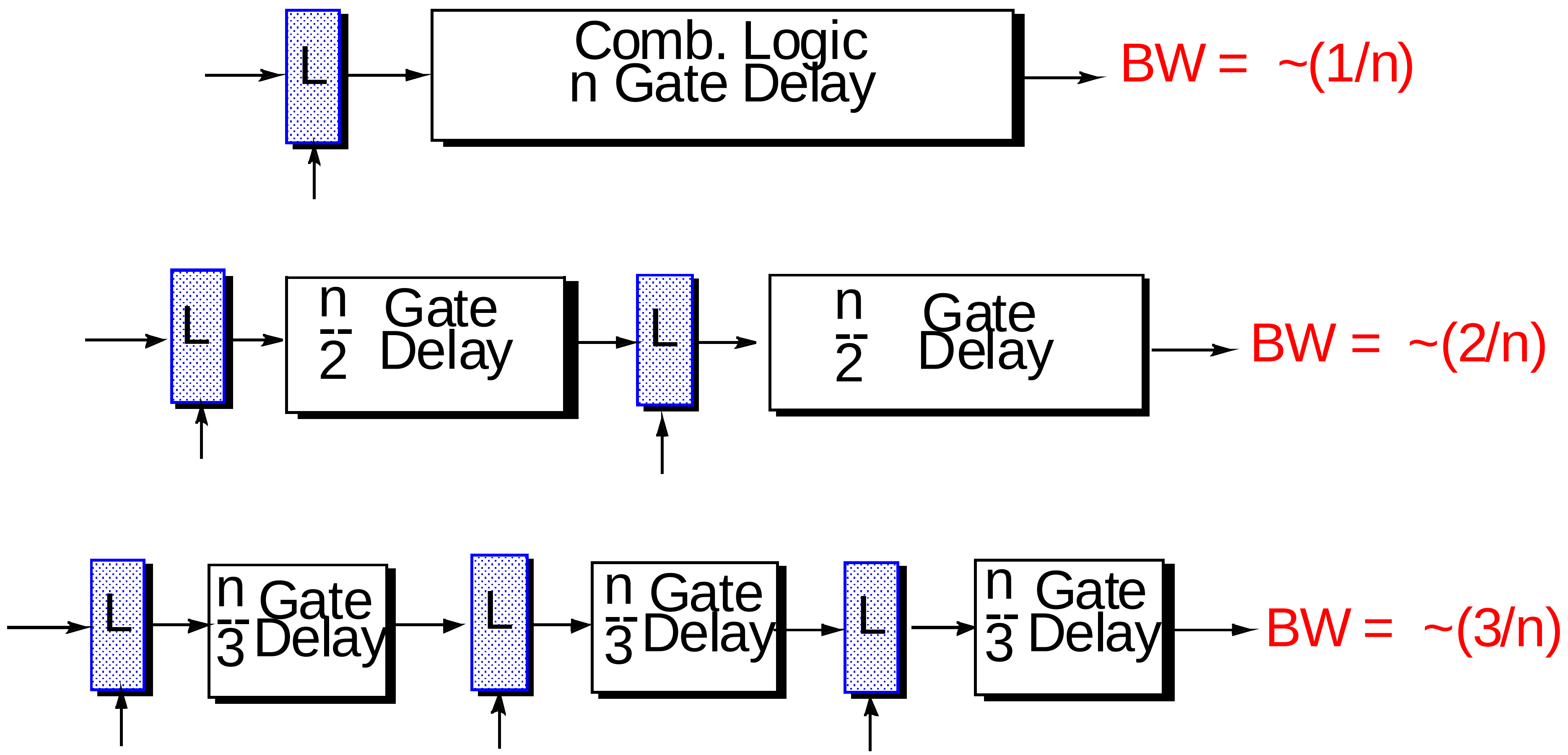
Pipelining



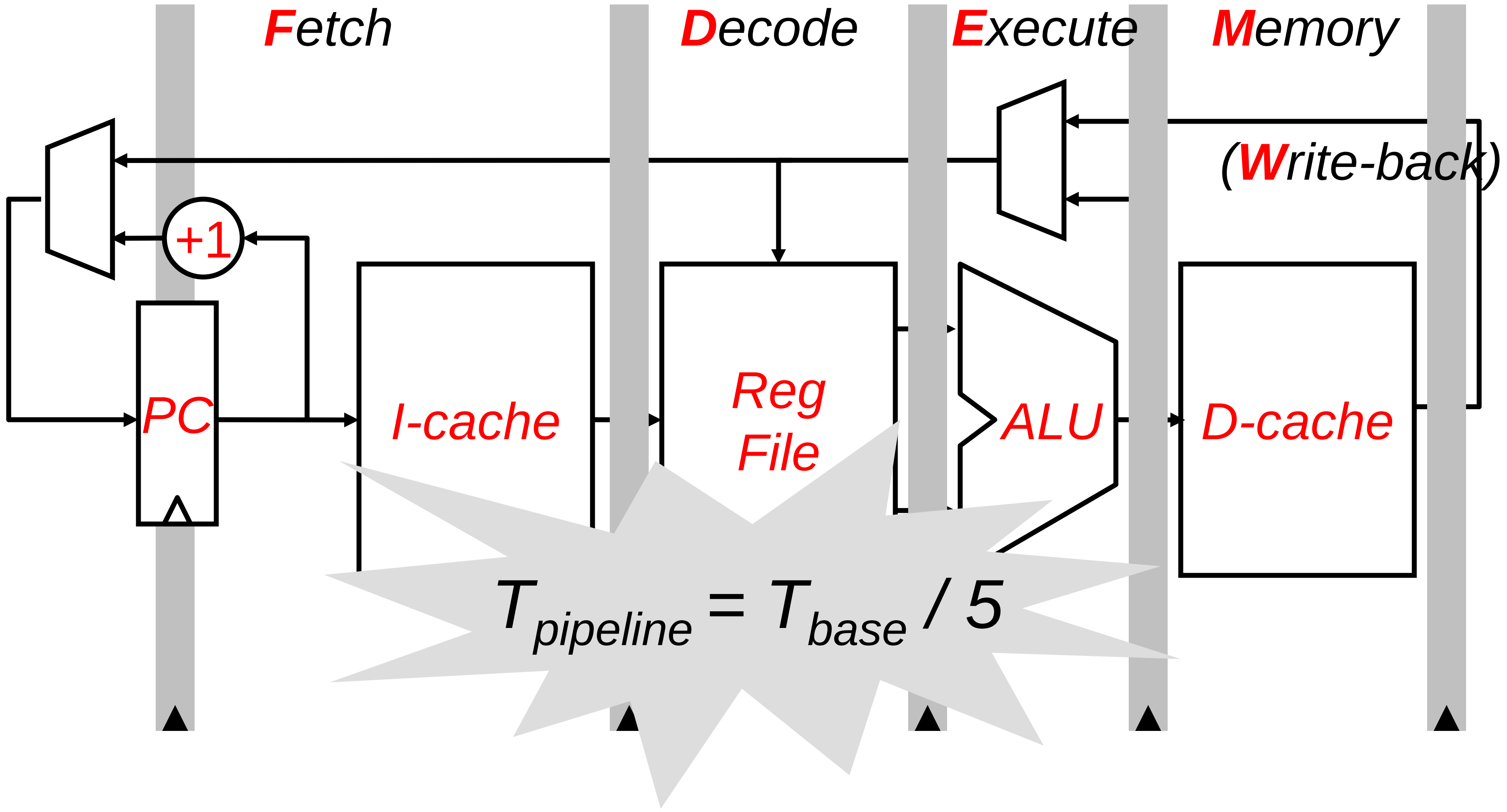
- Important performance technique
  - **Improves throughput at the expense of latency**
    - *Why does latency go up?*
- Begin with multi-cycle design
  - When instruction advances from stage 1 to 2…

    … allow next instruction to enter stage 1
  - Each instruction still passes through all stages
  - **But instructions enter and leave at a much faster rate**
- Automotive assembly line analogy

Pipeline Illustrated:



370 Processor Pipeline Review

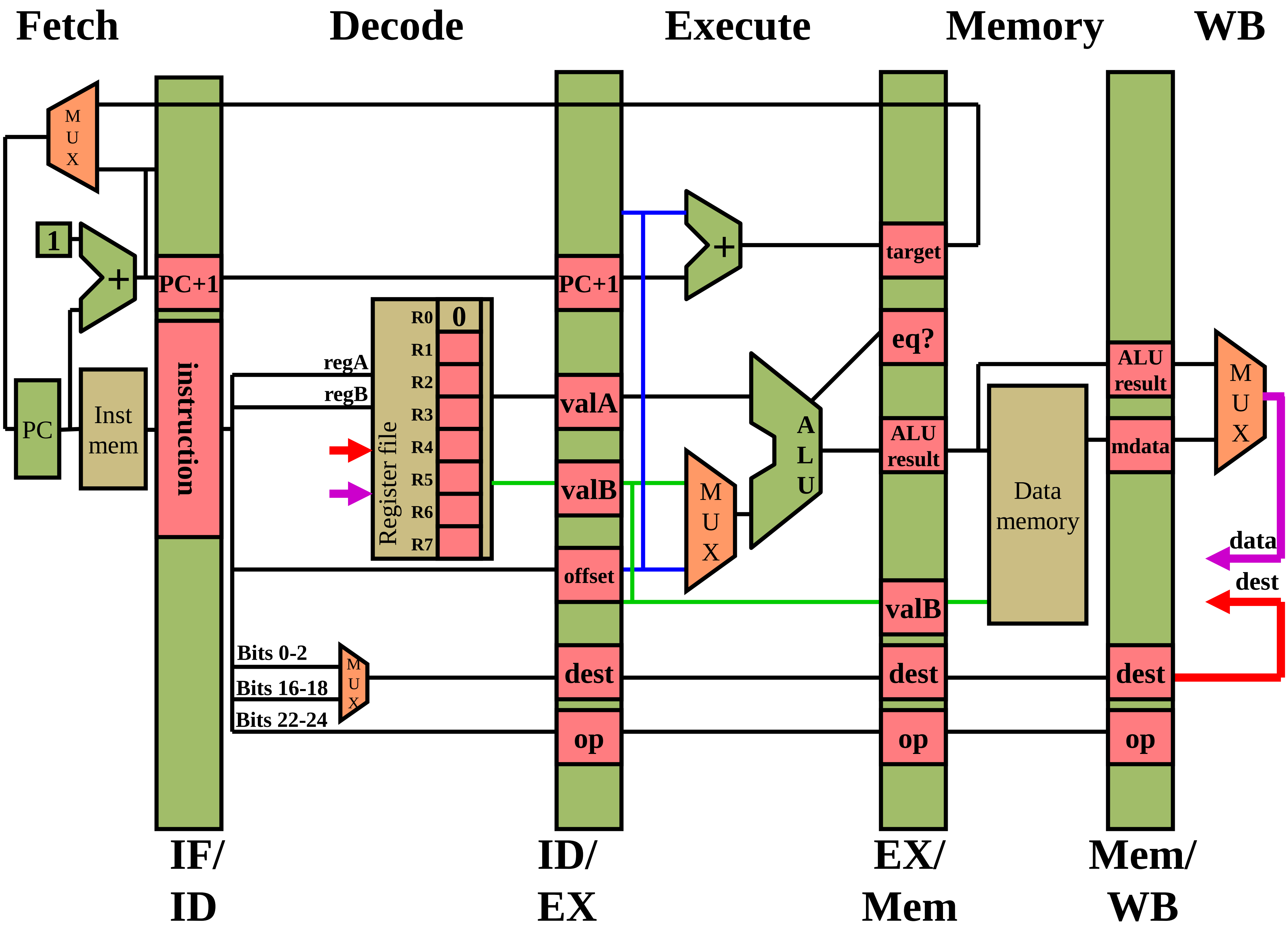


$T_{pipeline} = T_{base} / 5$

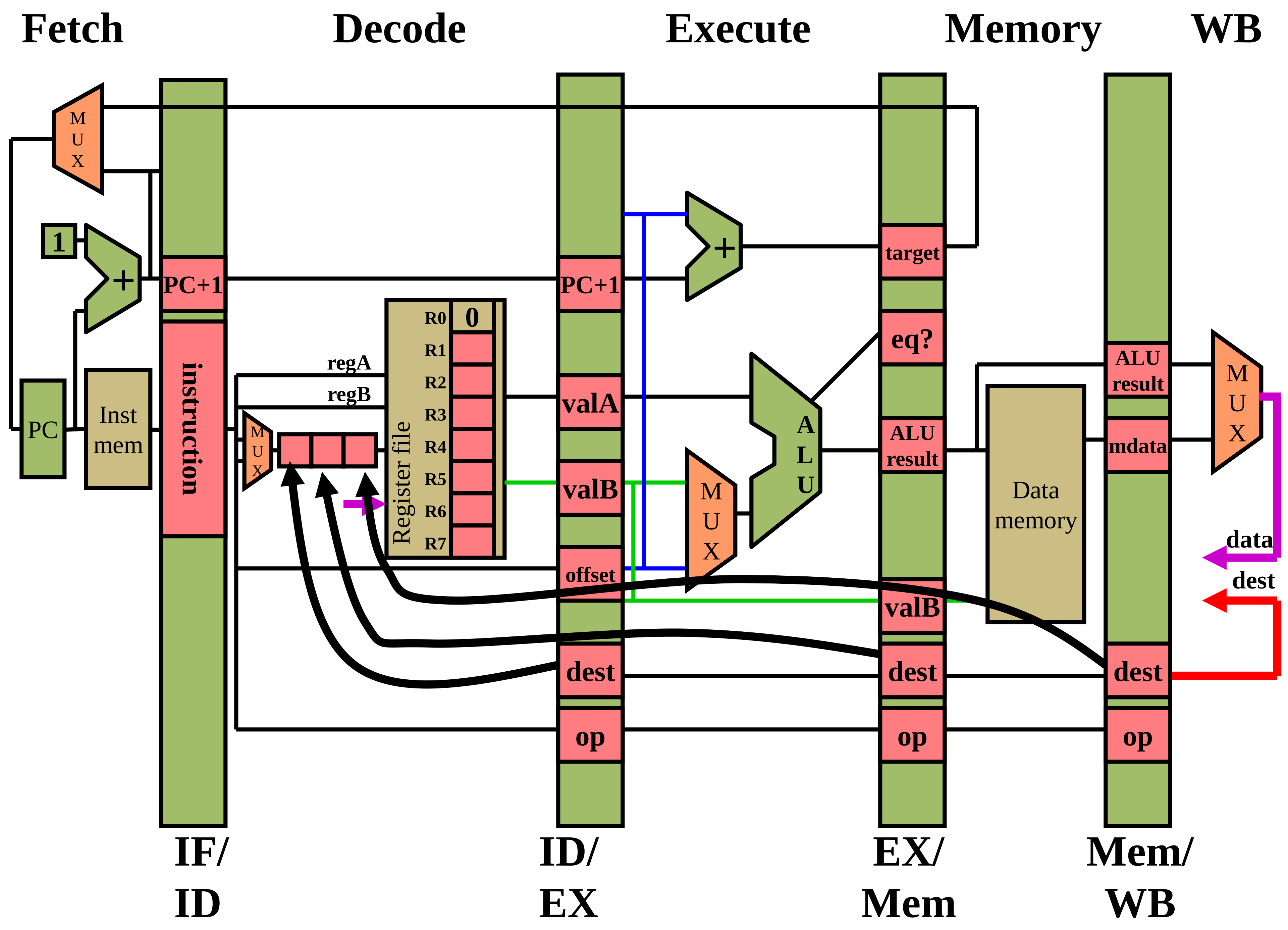
Basic Pipelining

- Data hazards
  - What are they?
  - How do you detect them?
  - How do you deal with them?
- Micro-architectural changes
  - Pipeline depth
  - Pipeline width
- Forwarding ISA (minor point)
- Control hazards (time allowing)

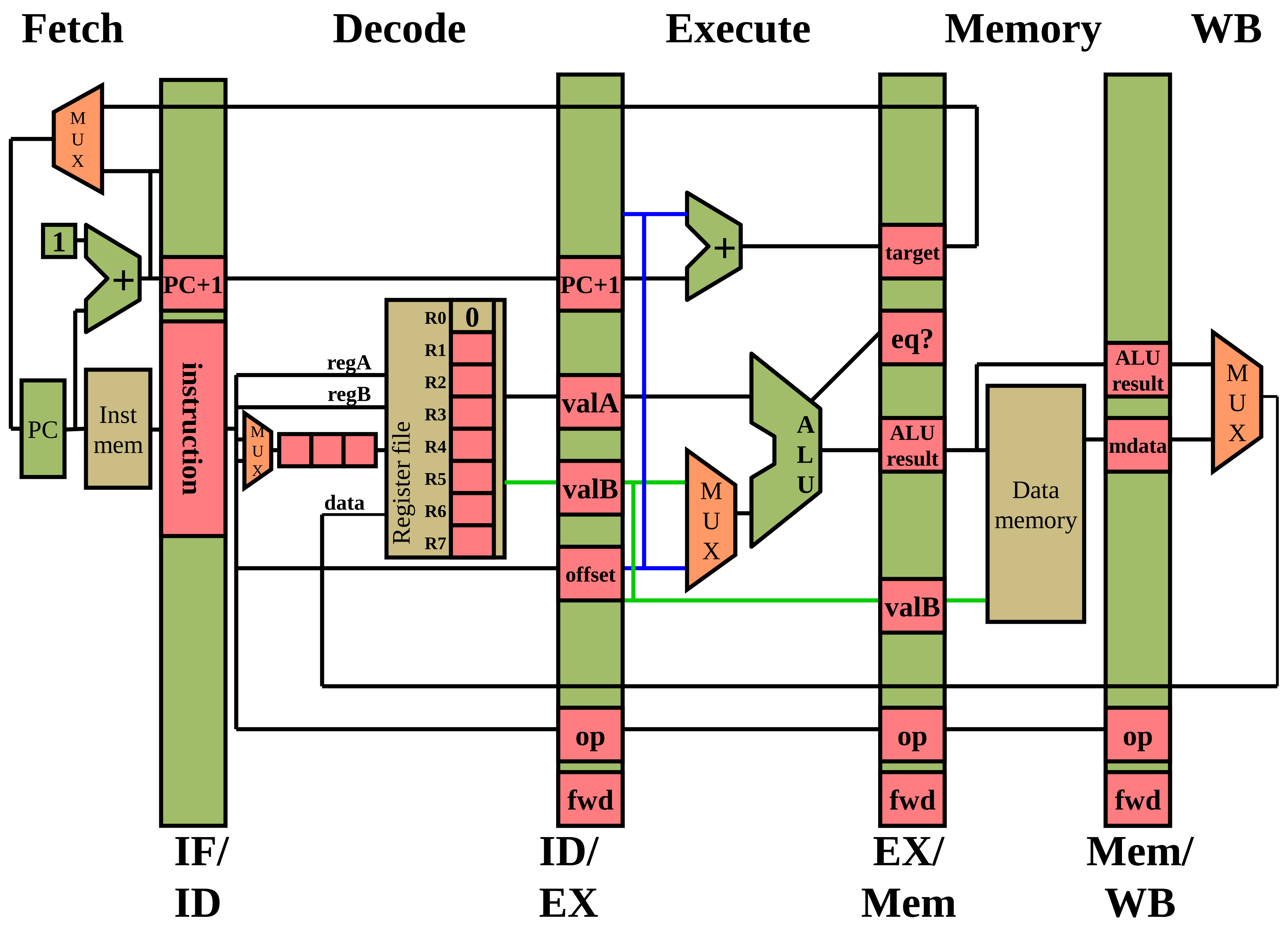
Basic Pipelining



Basic Pipelining



Basic Pipelining

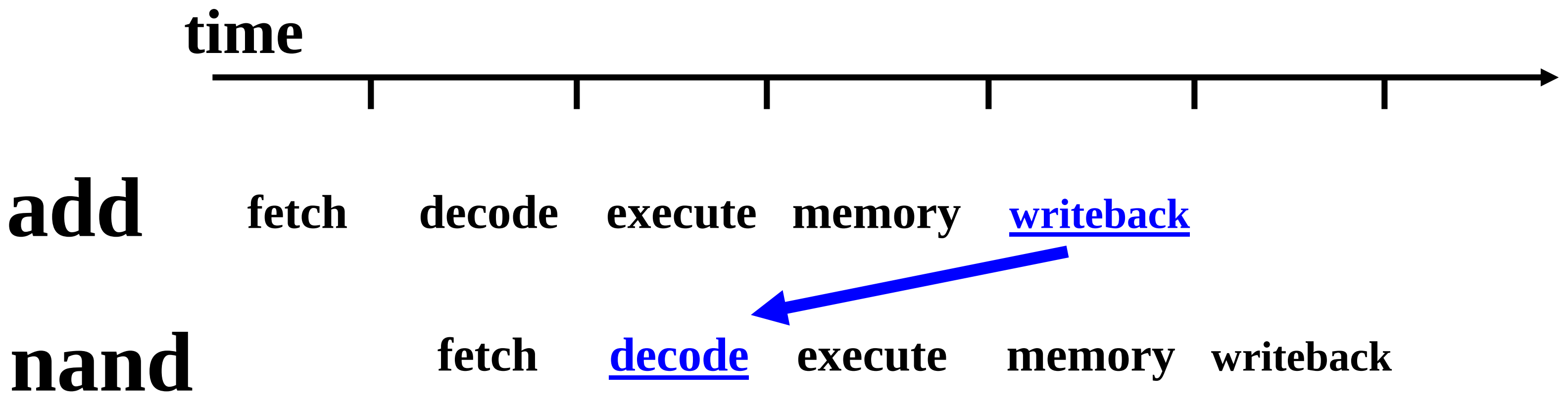


Pipeline function for ADD

- 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

add	1	2	3
nand	3	4	5



If not careful, you will read the wrong value of R3

Three approaches to handling data hazards

- Avoidance
  - 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: 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 R3 in cycle 5**
noop
noop
nand 3 4 5 ⟵ **read R3 in cycle 6**

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 one, but some I’s are noops

Handling data hazards: detect and stall

- Detection:
  - 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

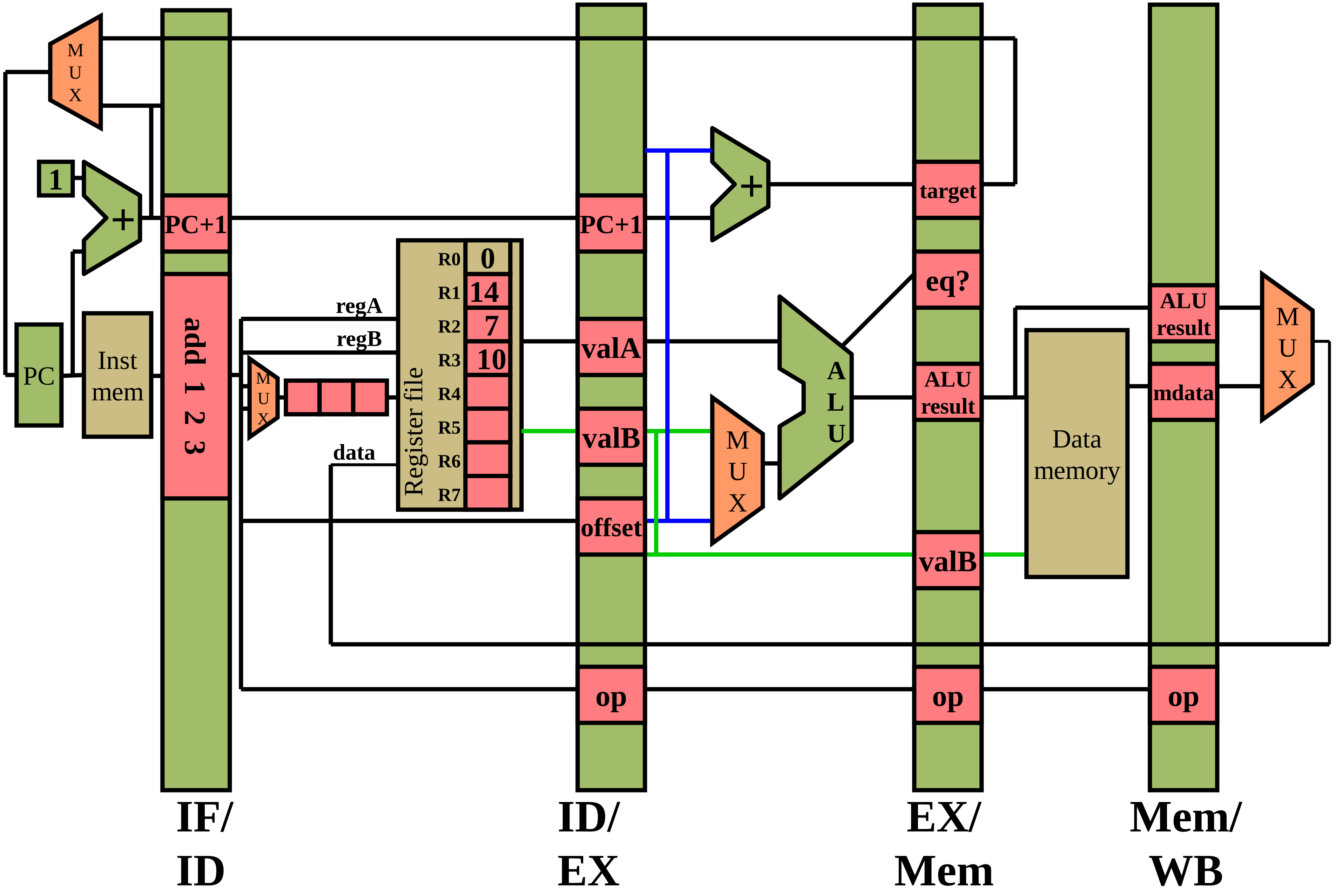
Pipelining & Data Hazards

Avoidance

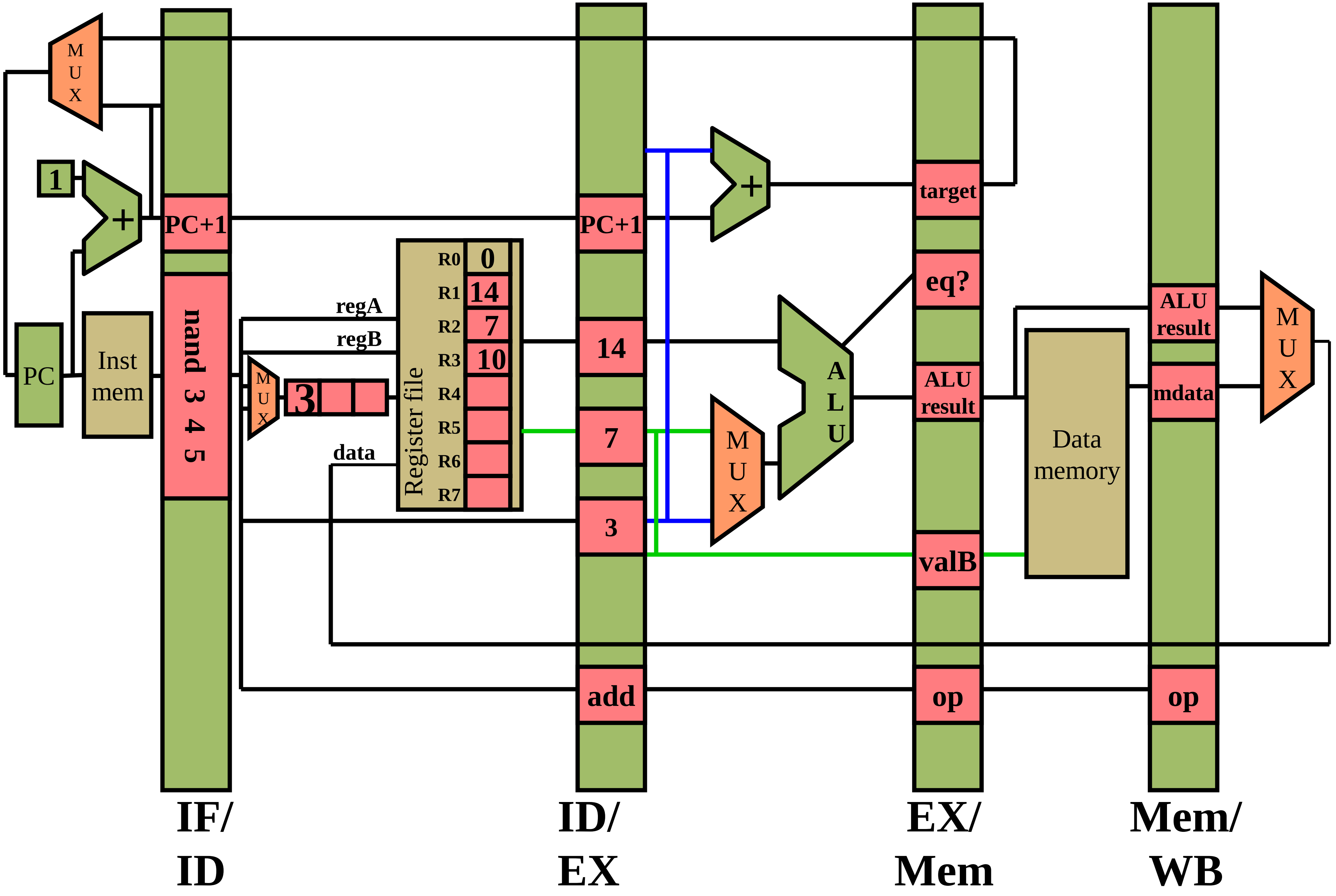
Detect and Stall

Detect and Forward

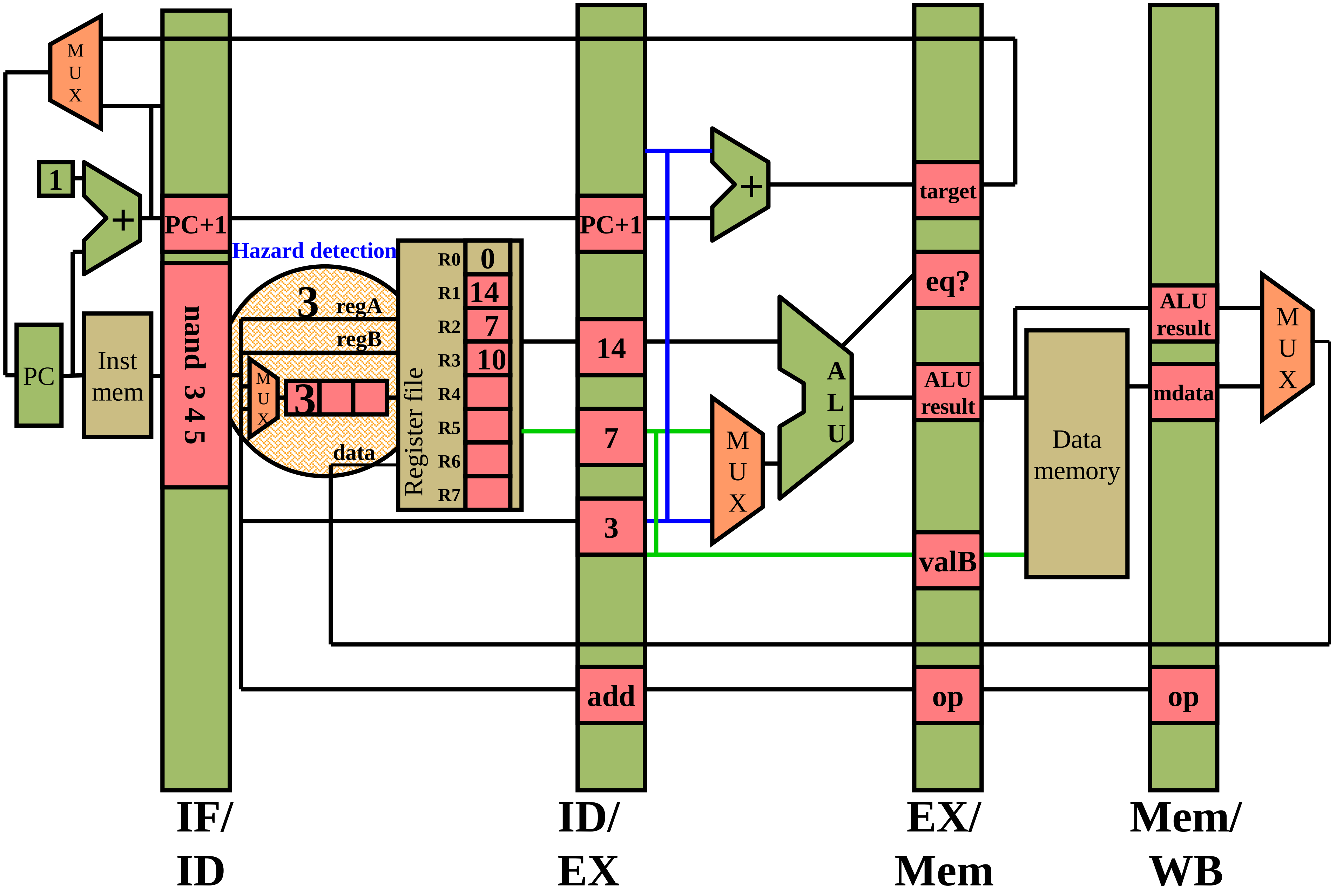
End of Cycle 1



End of Cycle 2



First half of cycle 3

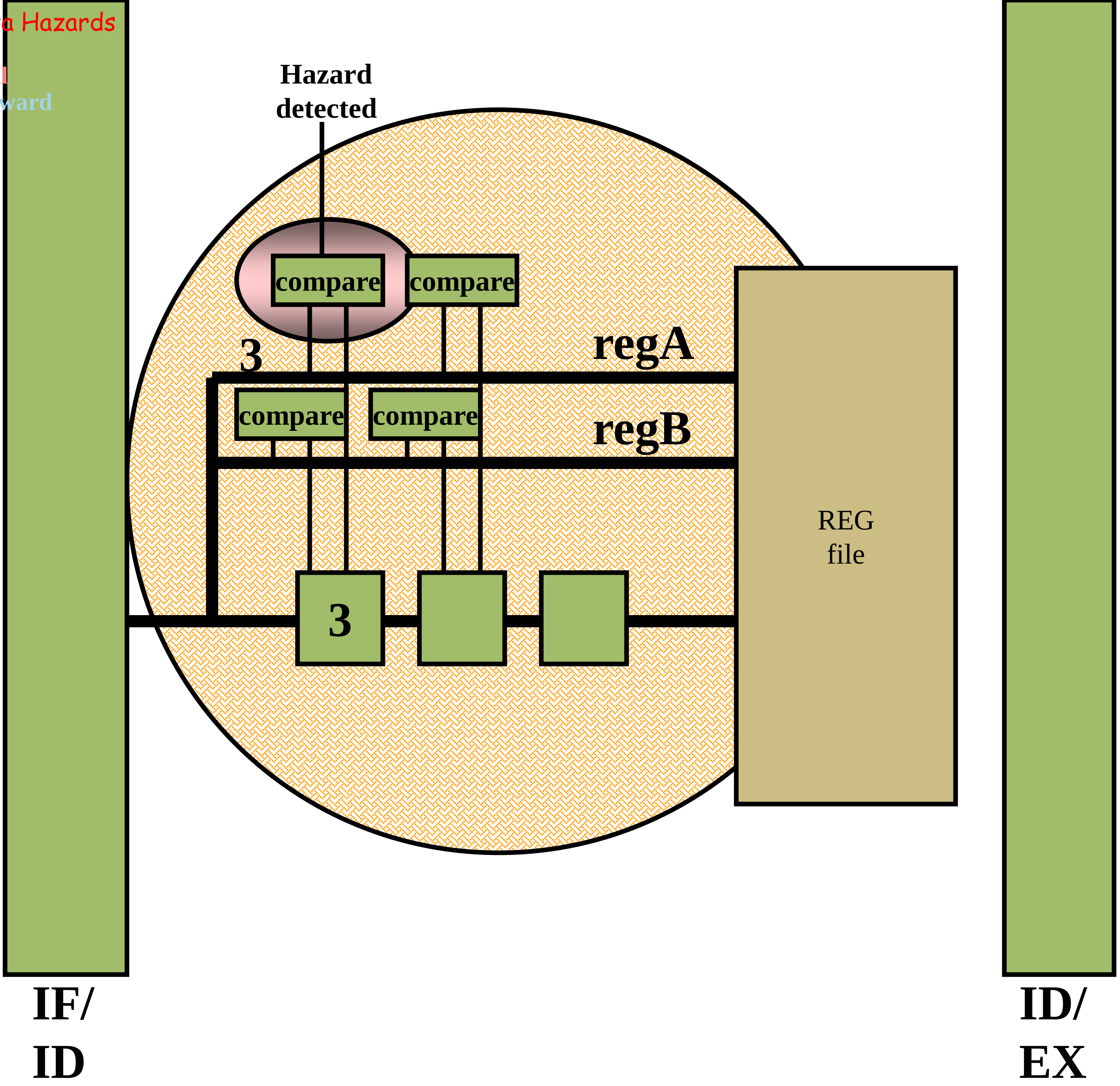


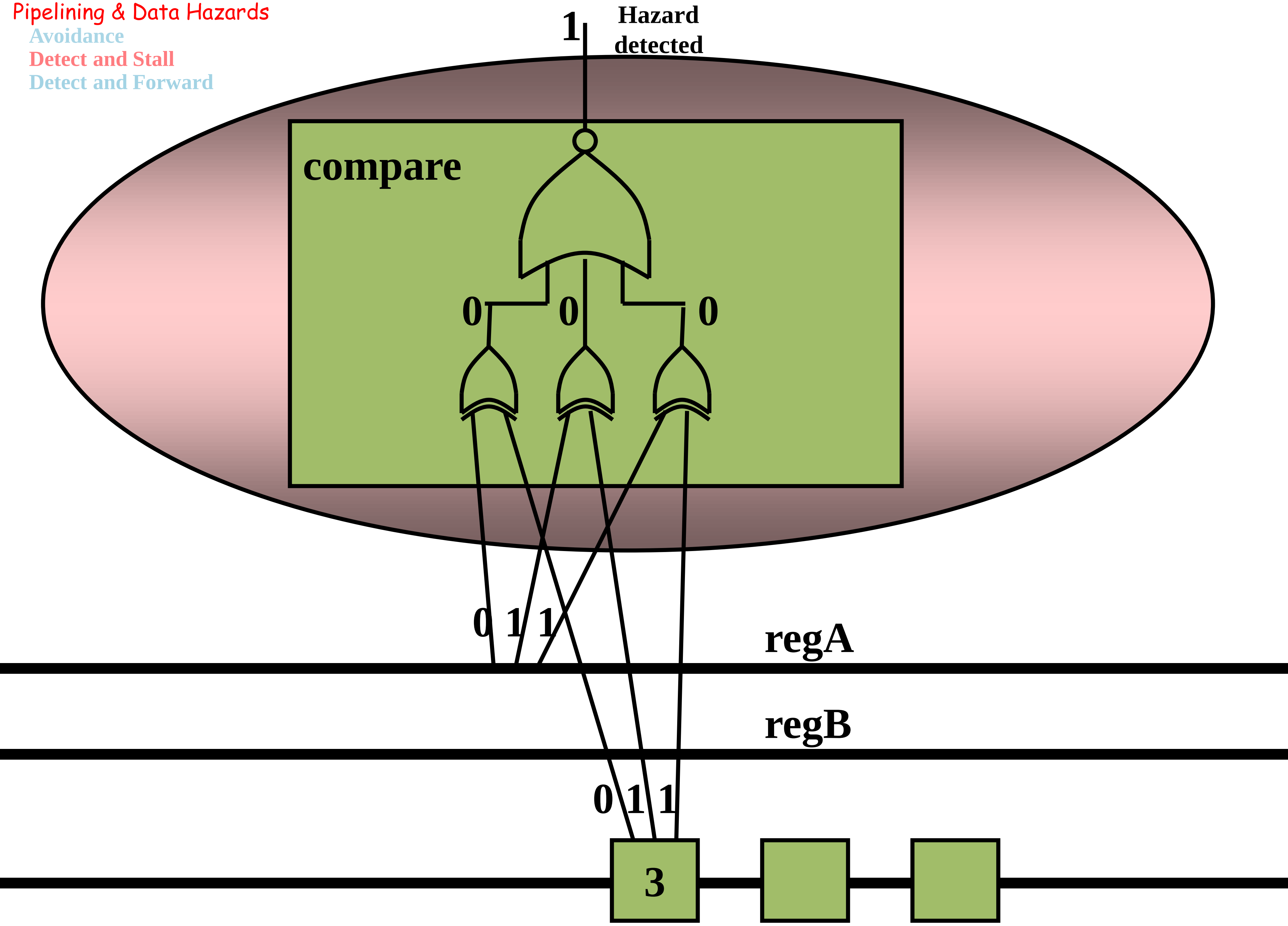
Pipelining & Data Hazards

Avoidance

Detect and Stall

Detect and Forward




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

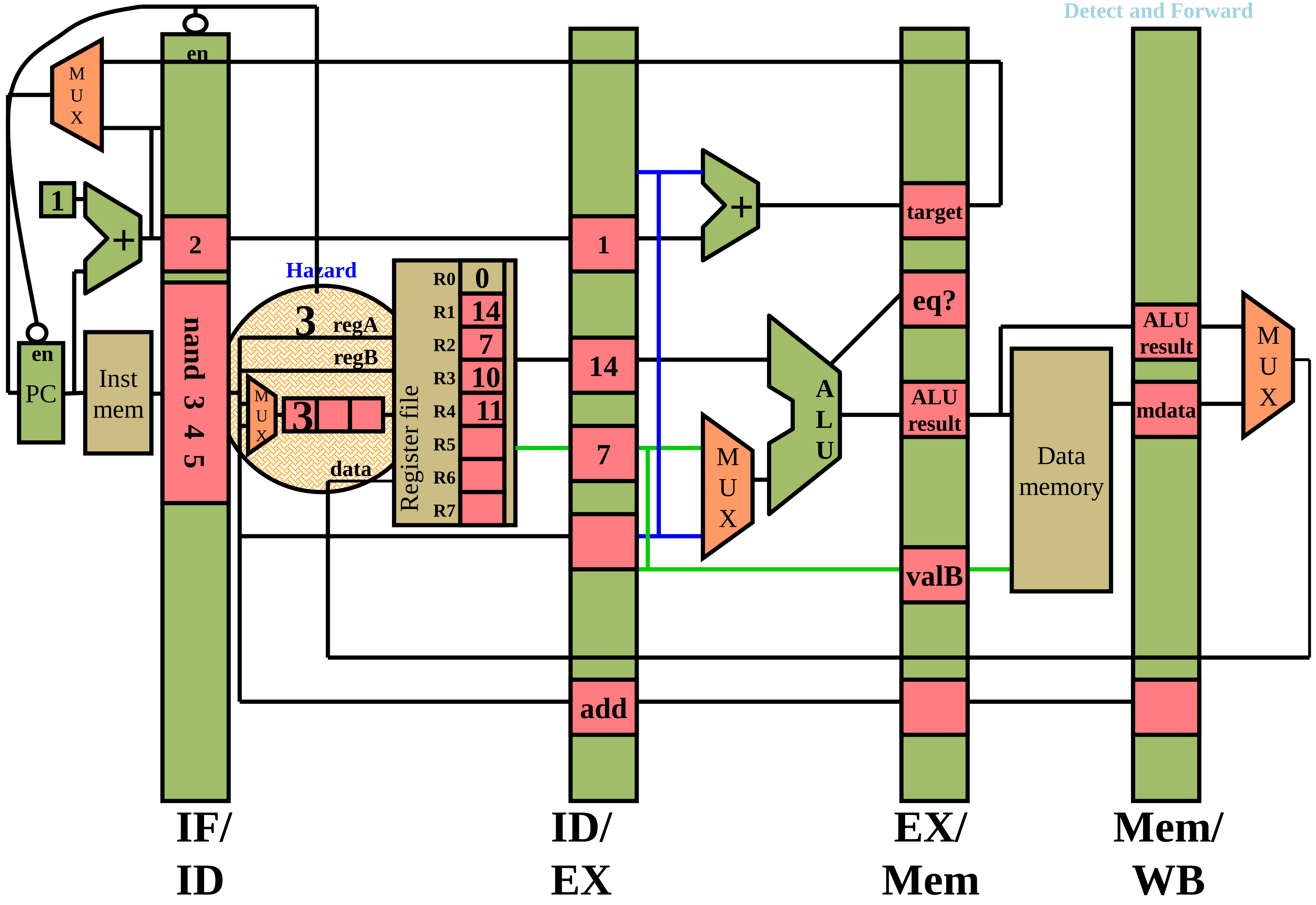
Handling data hazards: detect and stall the pipeline until ready

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

  <u>**Keep current instructions in fetch and decode**</u>

  Pass a noop to execute

First half of cycle 3

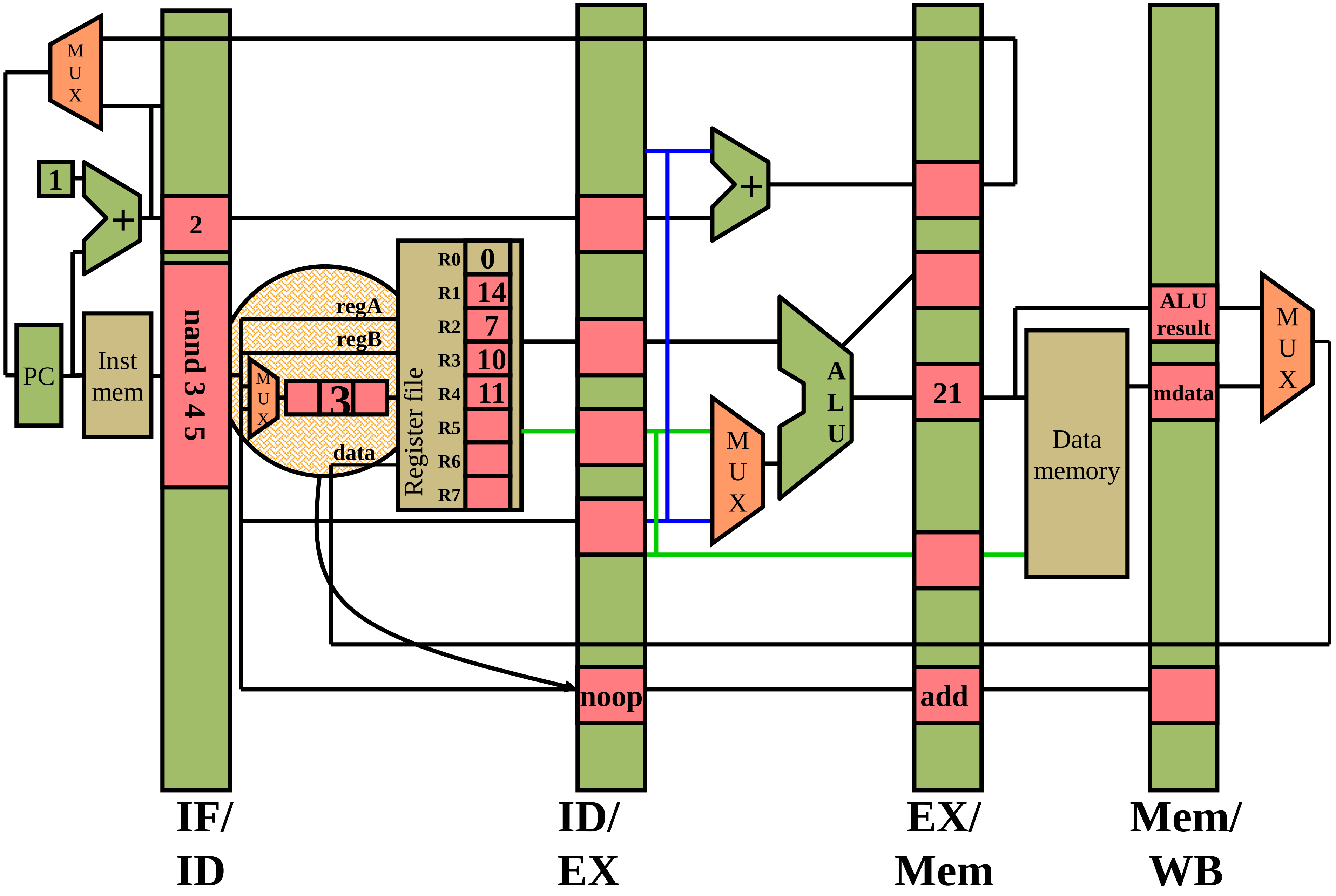


Handling data hazards: detect and stall the pipeline until ready

- Detection:
  - 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**

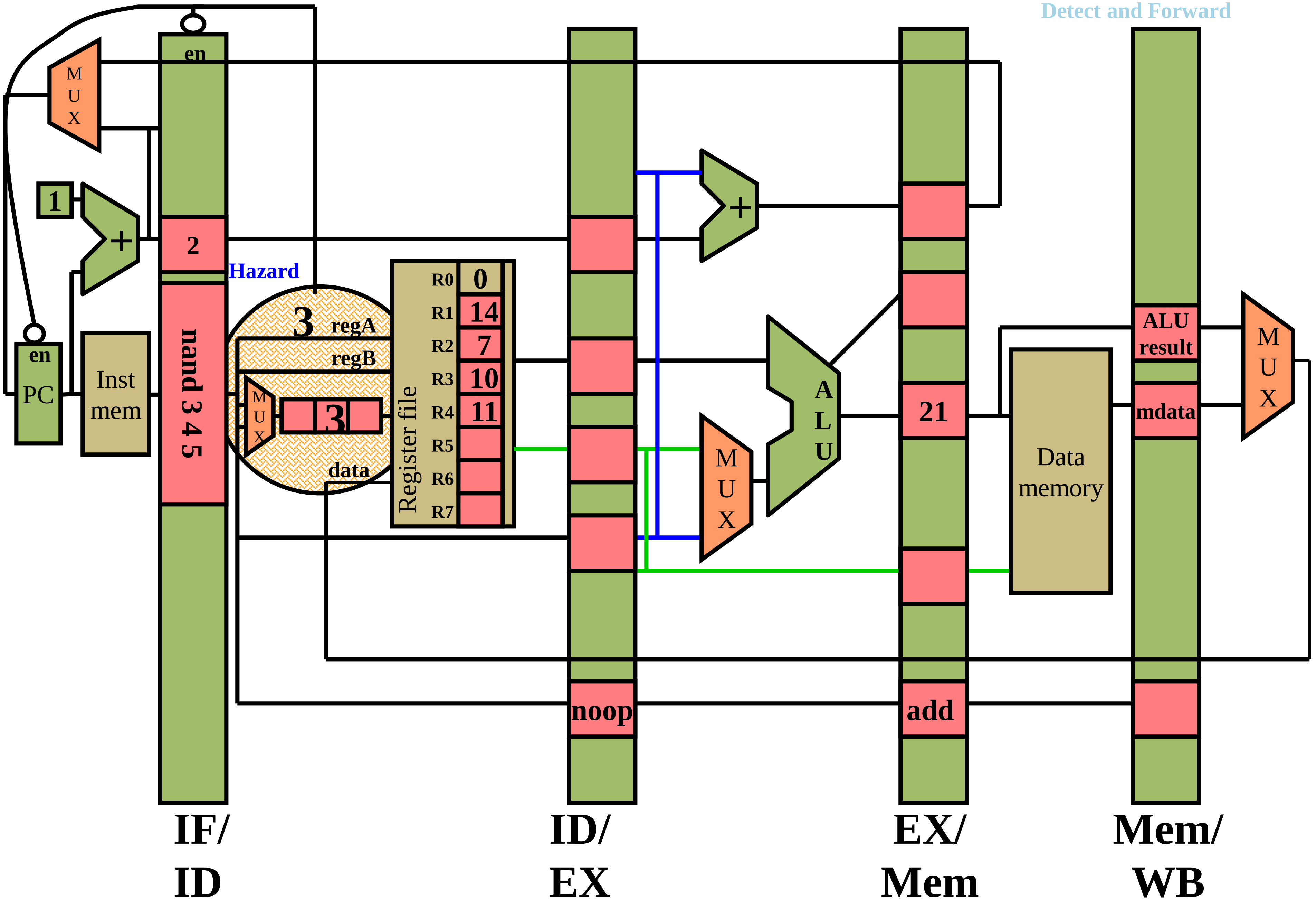
Pipelining & Data Hazards
Avoidance
Detect and Stall
Detect and Forward

End of cycle 3

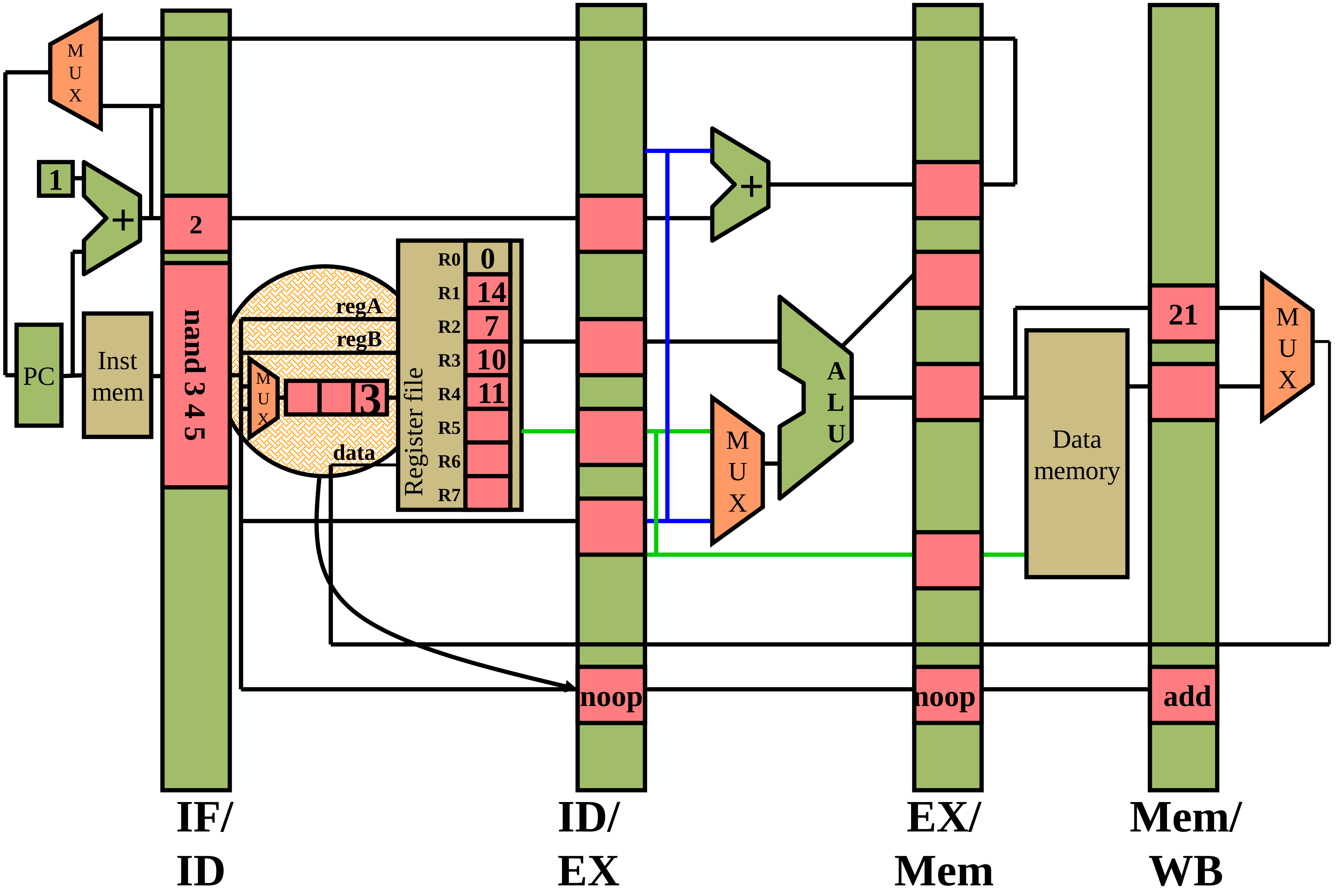


Pipelining & Data Hazards
Avoidance
Detect and Stall
Detect and Forward

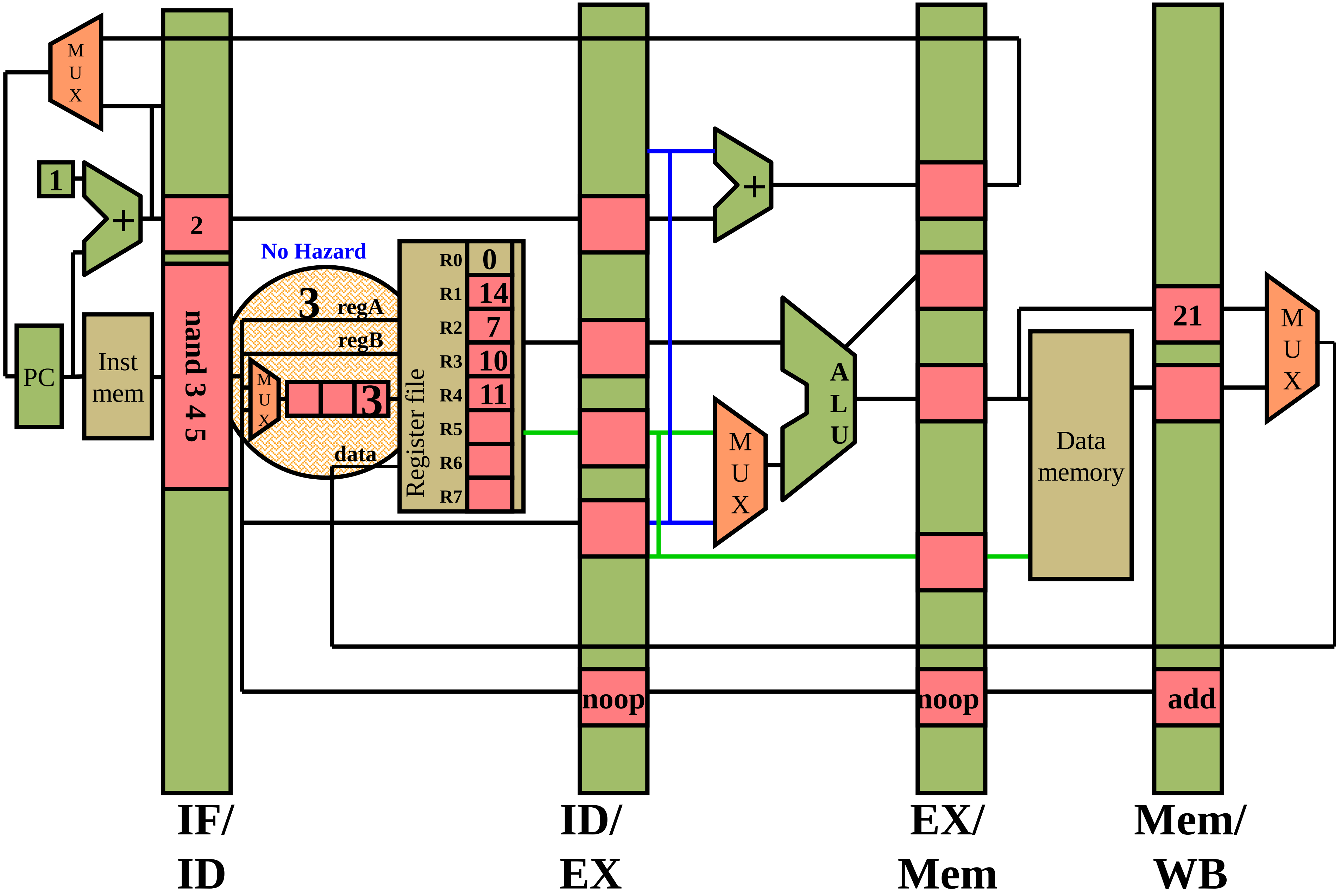
First half of cycle 4



End of cycle 4



First half of cycle 5



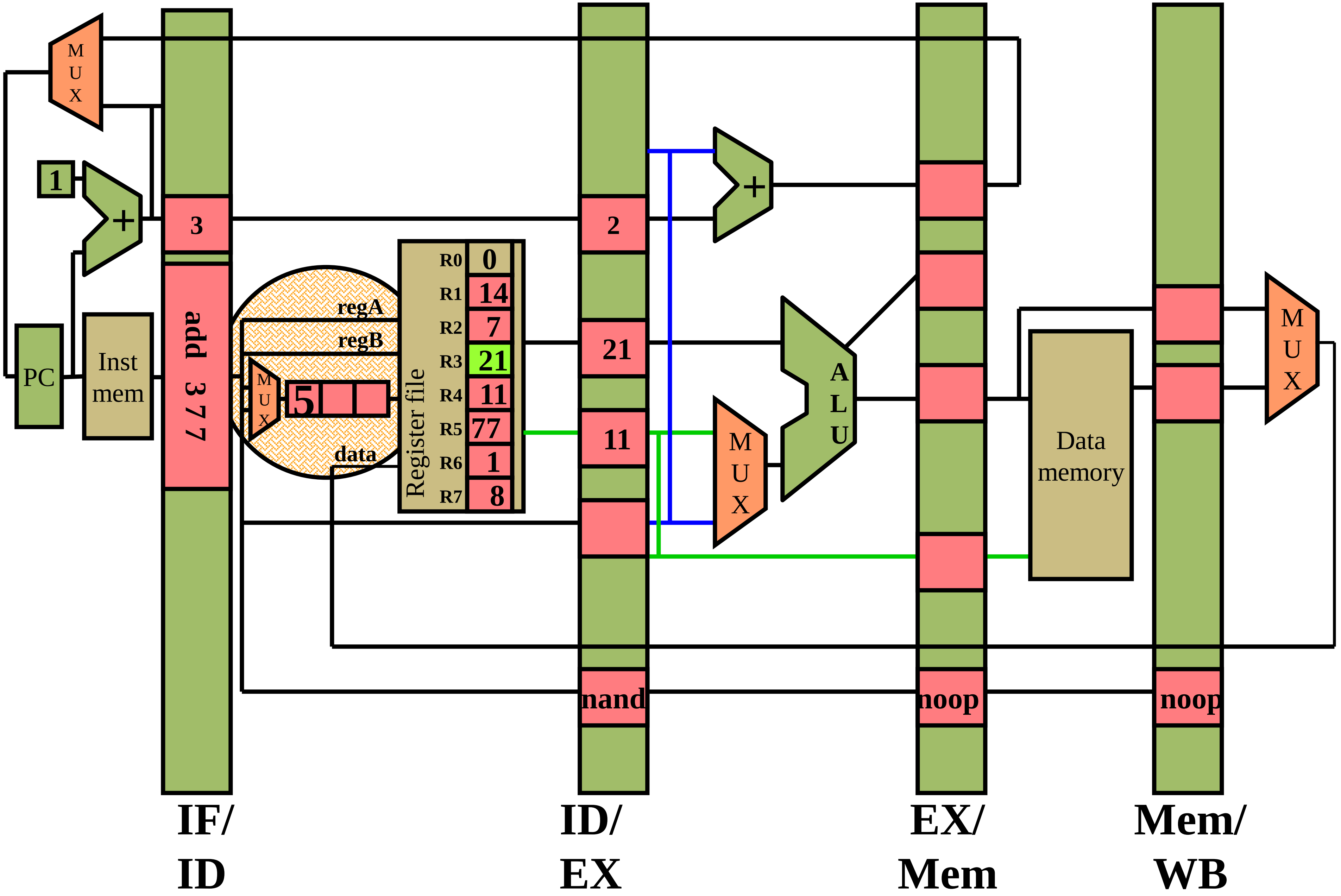
Pipelining & Data Hazards

Avoidance

Detect and Stall

Detect and Forward

End of cycle 5



No more hazard: stalling

add	1	2	3
nand	3	4	5

time

add	fetch	decode	execute	memory	writeback	
nand		fetch	decode	decode	decode	execute
			hazard	hazard		

We are careful to get the right value of R3

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 nand
    - nand 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.

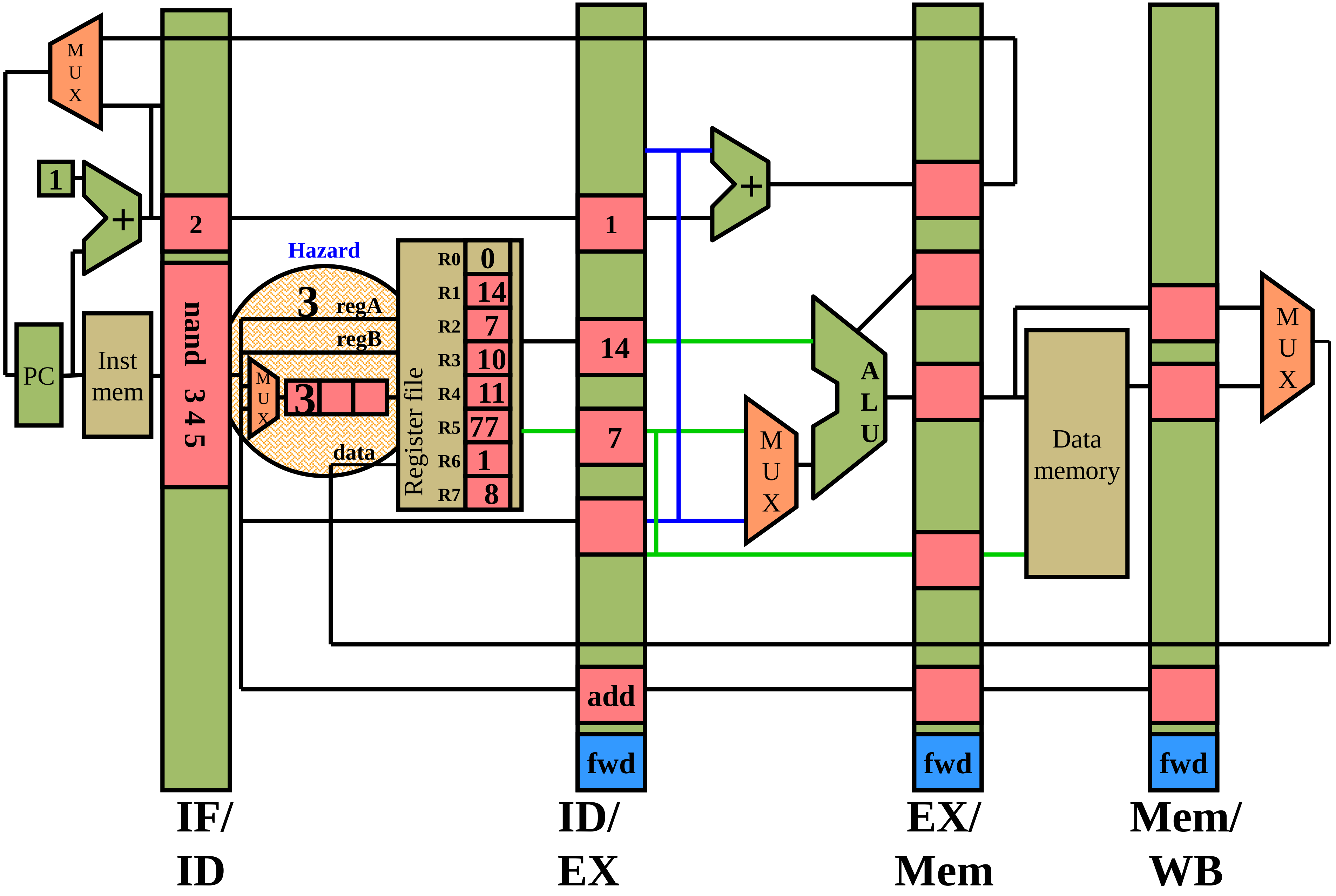
Handling data hazards: detect and forward

- Detection: same as detect and stall
  - Except that all 4 hazards are treated differently
    - i.e., you can’t logical-OR the 4 hazard signals
- Forward:
  - New datapaths to route computed data to where it is needed
  - New Mux and control to pick the right data

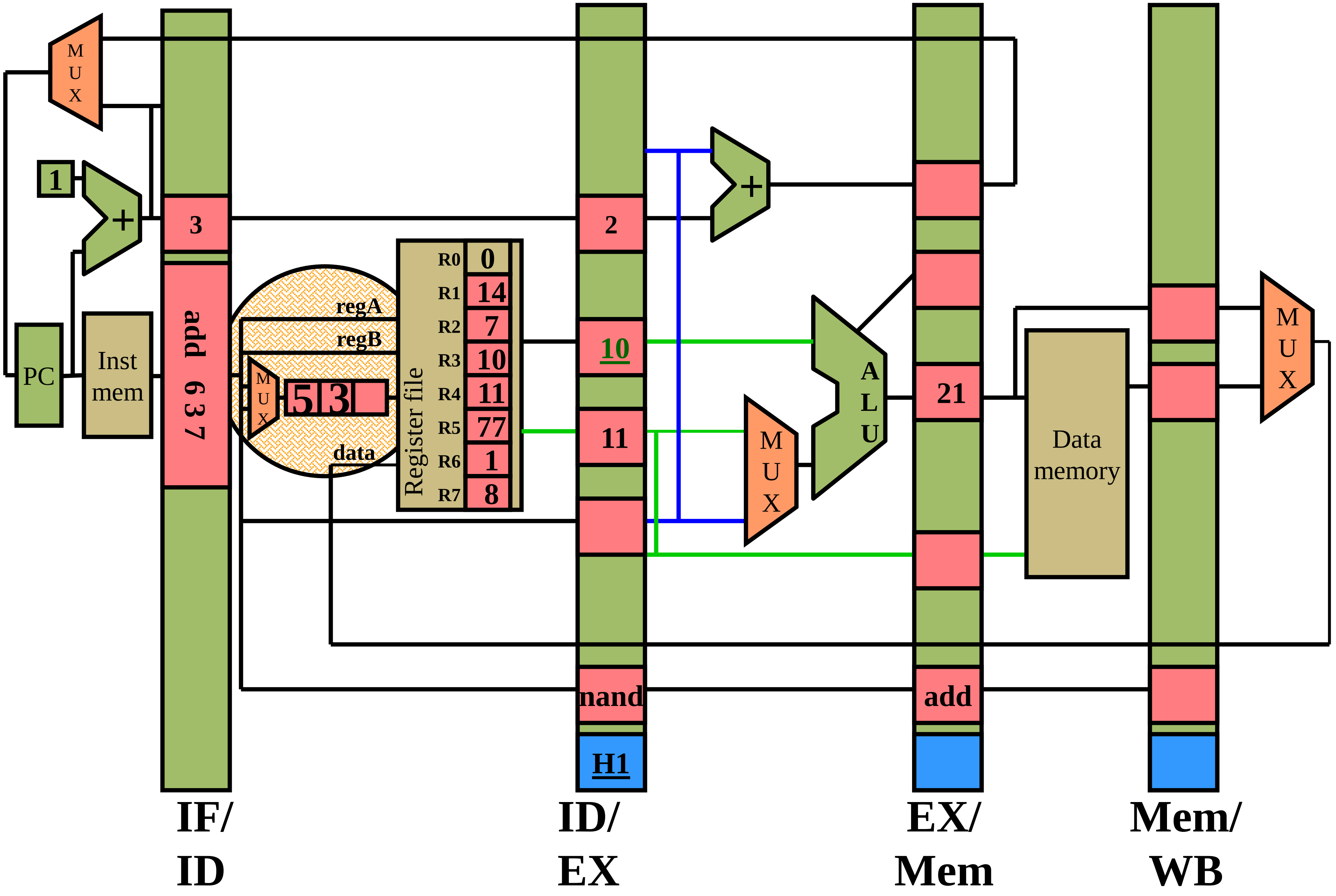
Detect and Forward Example

```
add 1 2 3         // r3 = r1 + r2

nand 3 4 5        // r5 = r3 NAND r4

add 6 3 7         // r7 = r3 + r6

lw 3 6 10         // r6 = MEM[r3+10]

sw 6 2 12         // MEM[r6+12]=r2
```

First half of cycle 3



End of cycle 3



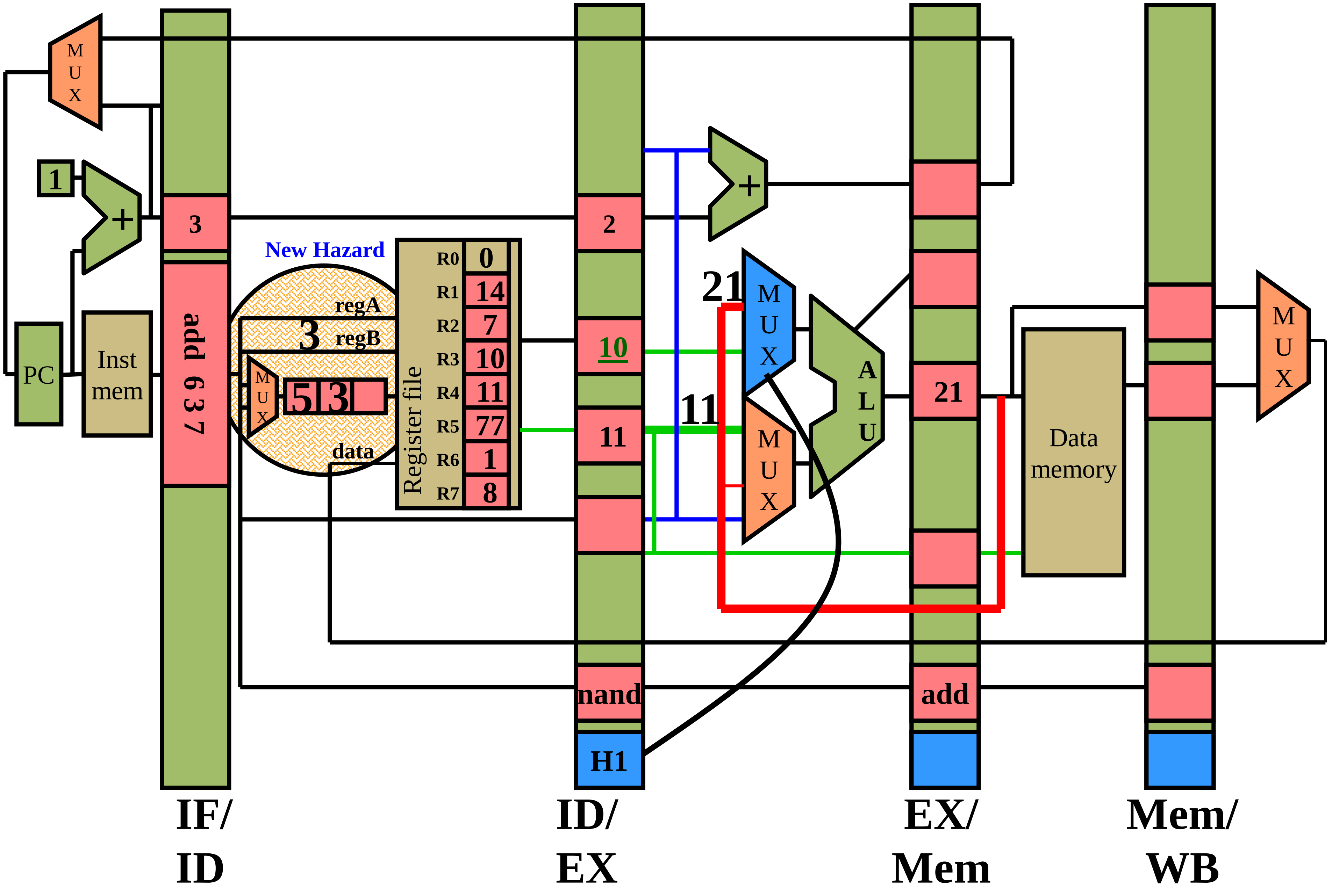
Pipelining & Data Hazards

Avoidance

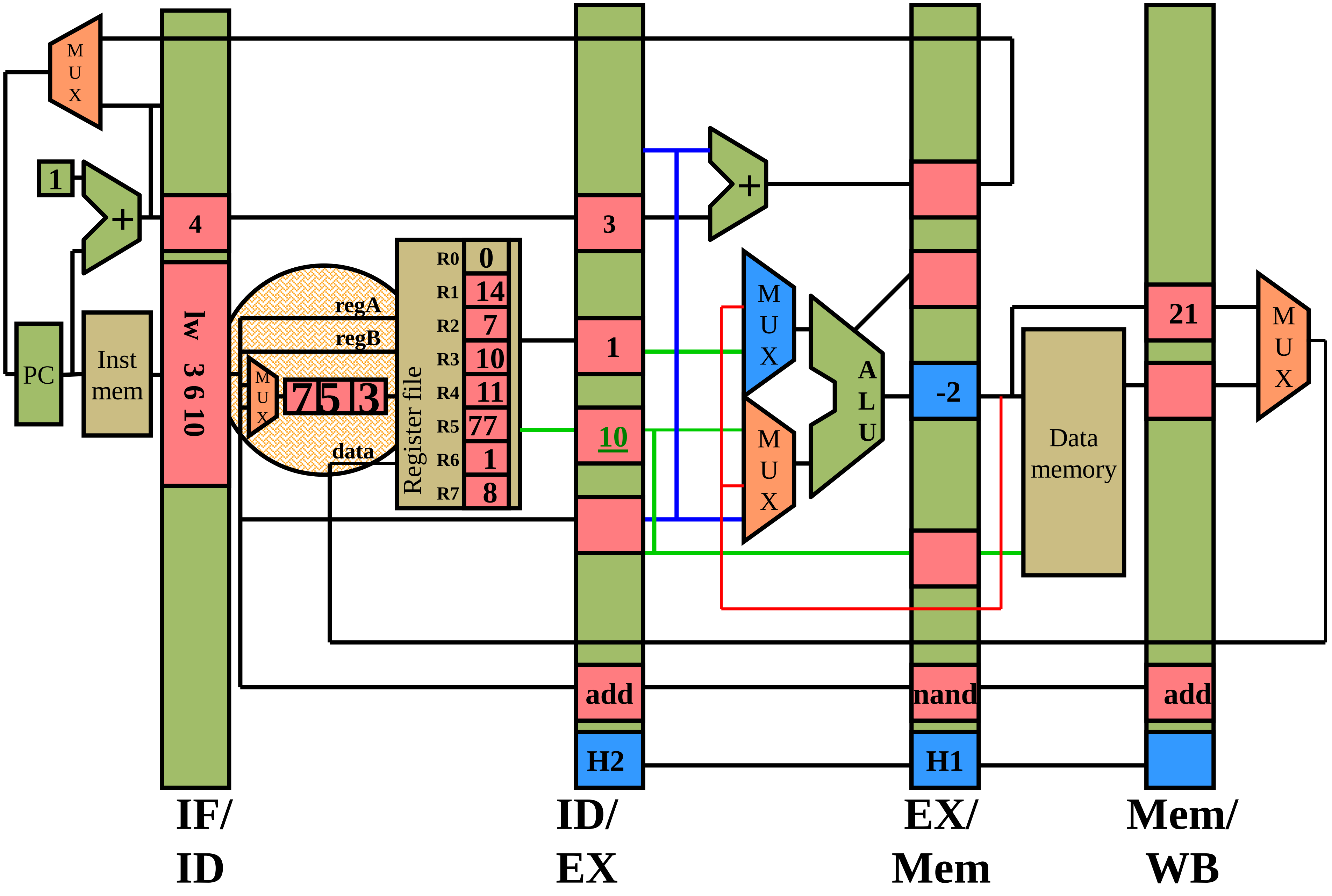
Detect and Stall

Detect and Forward

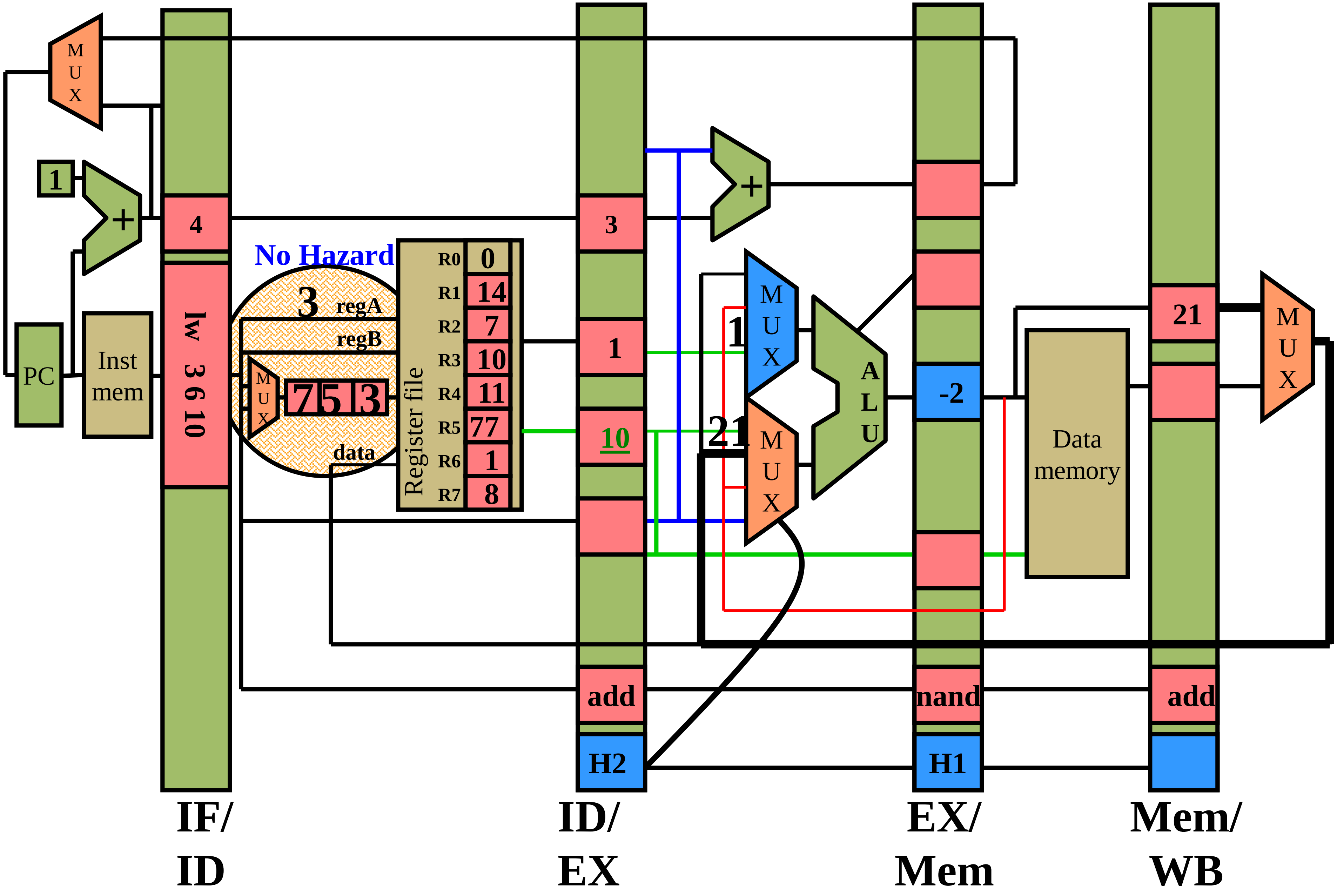
First half of cycle 4



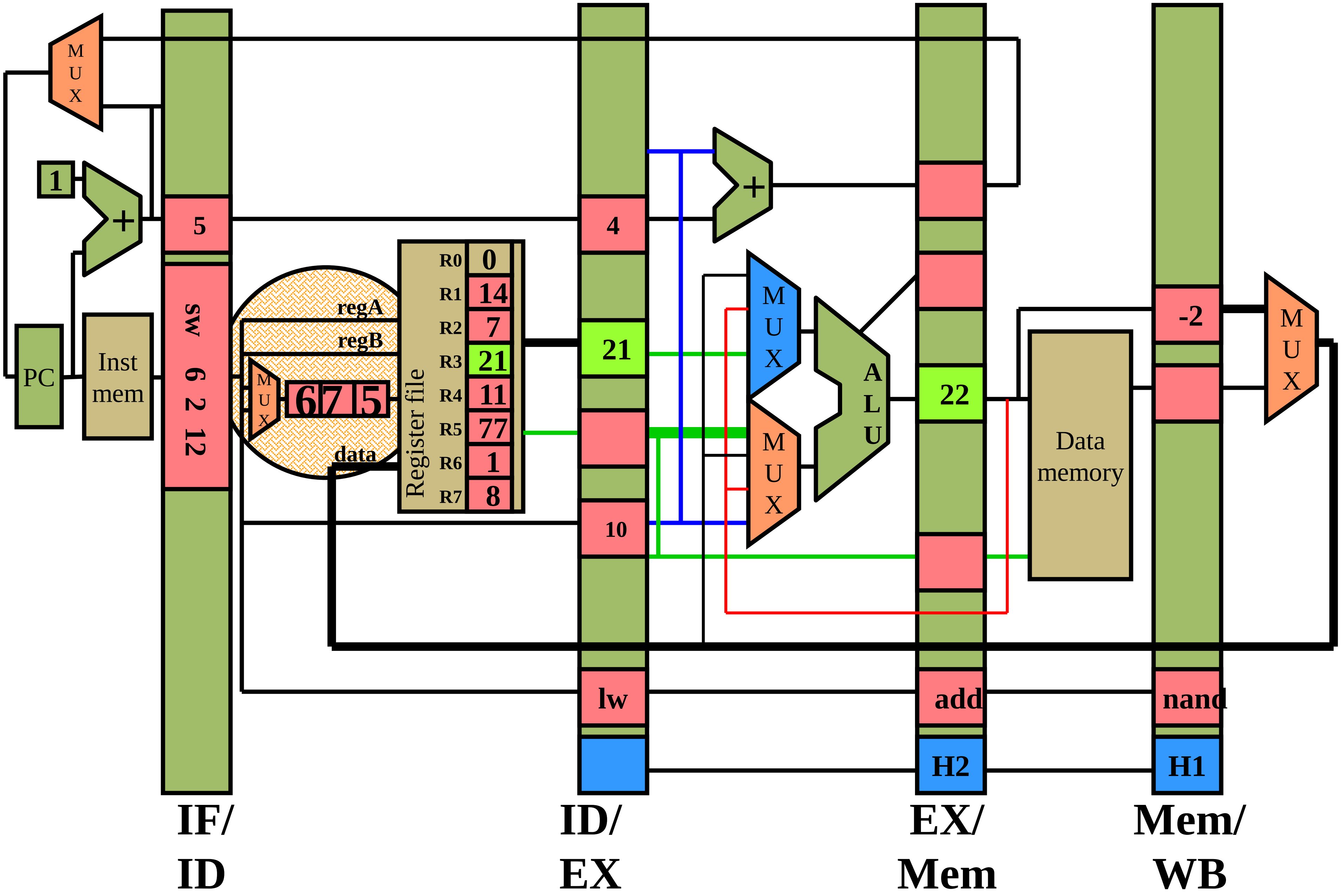
End of cycle 4



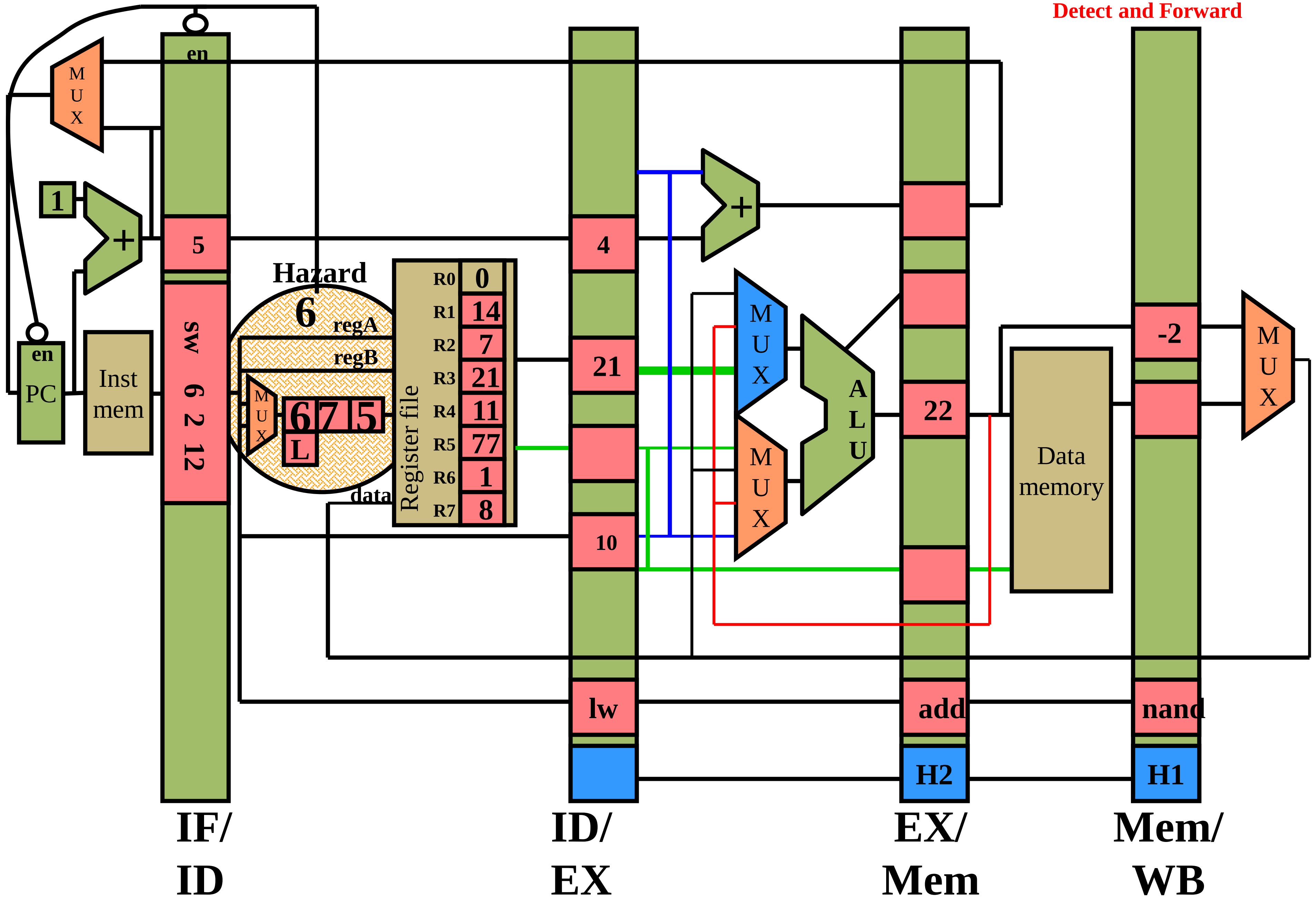
First half of cycle 5



End of cycle 5



First half of cycle 6



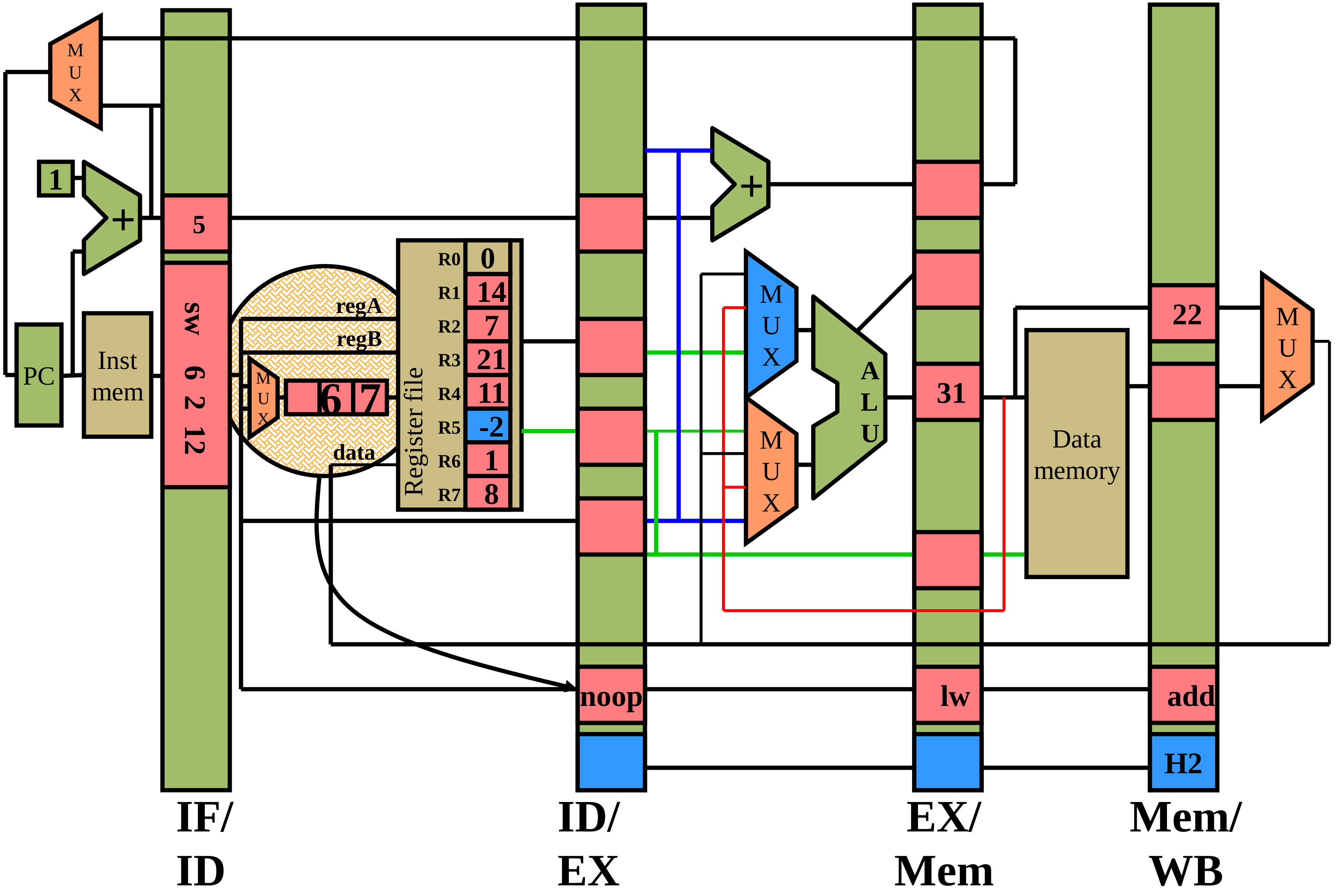
Pipelining & Data Hazards

Avoidance

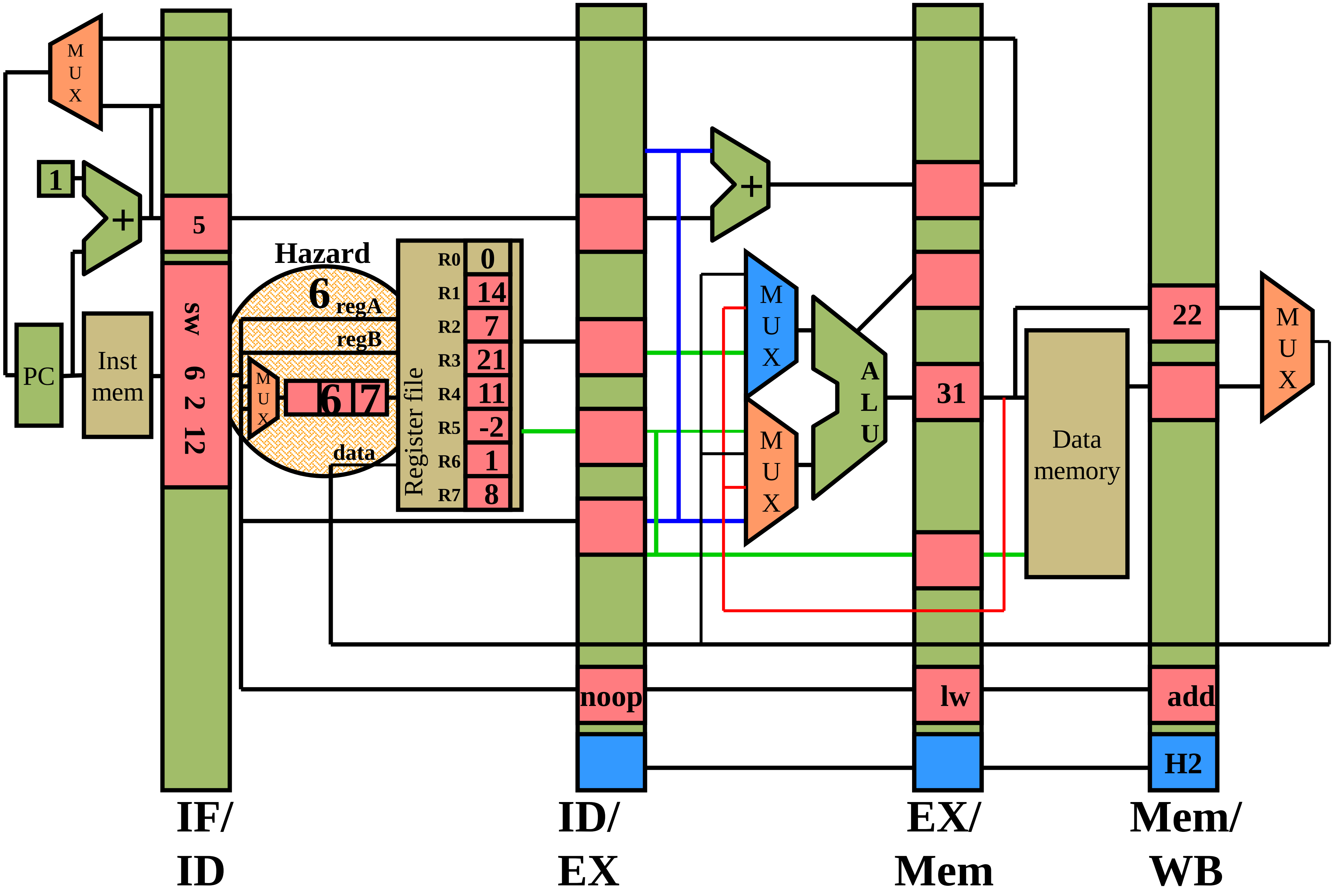
Detect and Stall

Detect and Forward

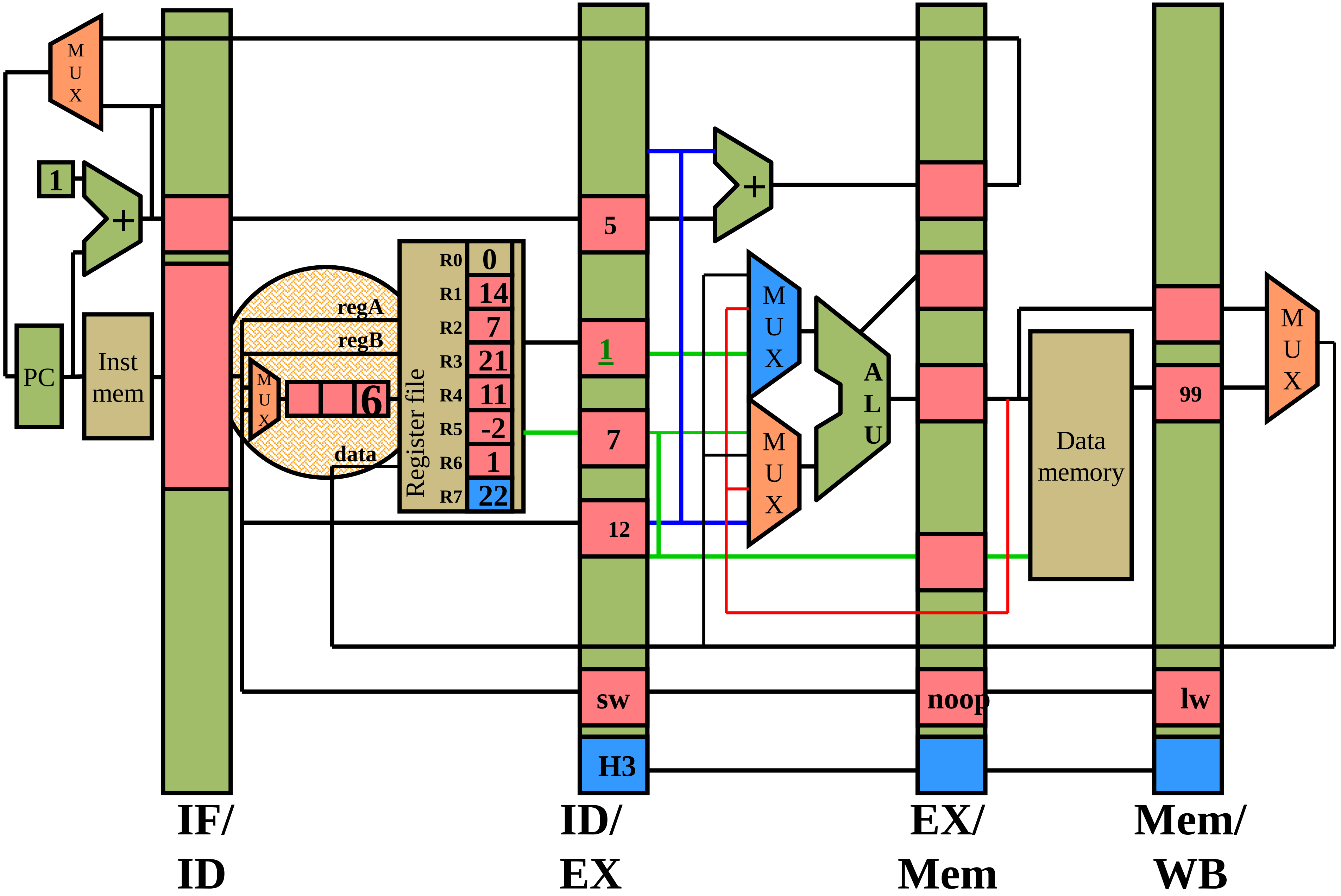
End of cycle 6



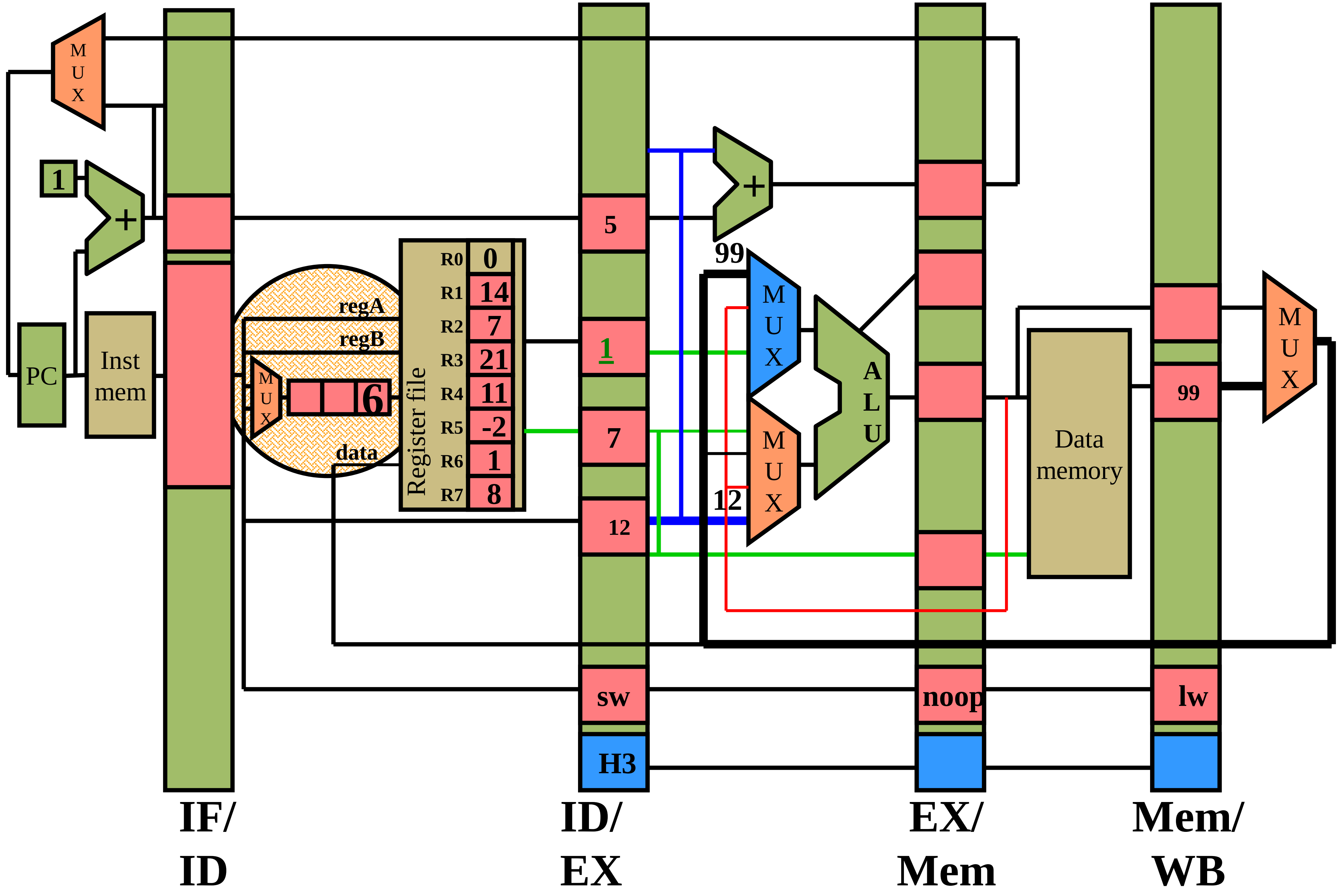
First half of cycle 7



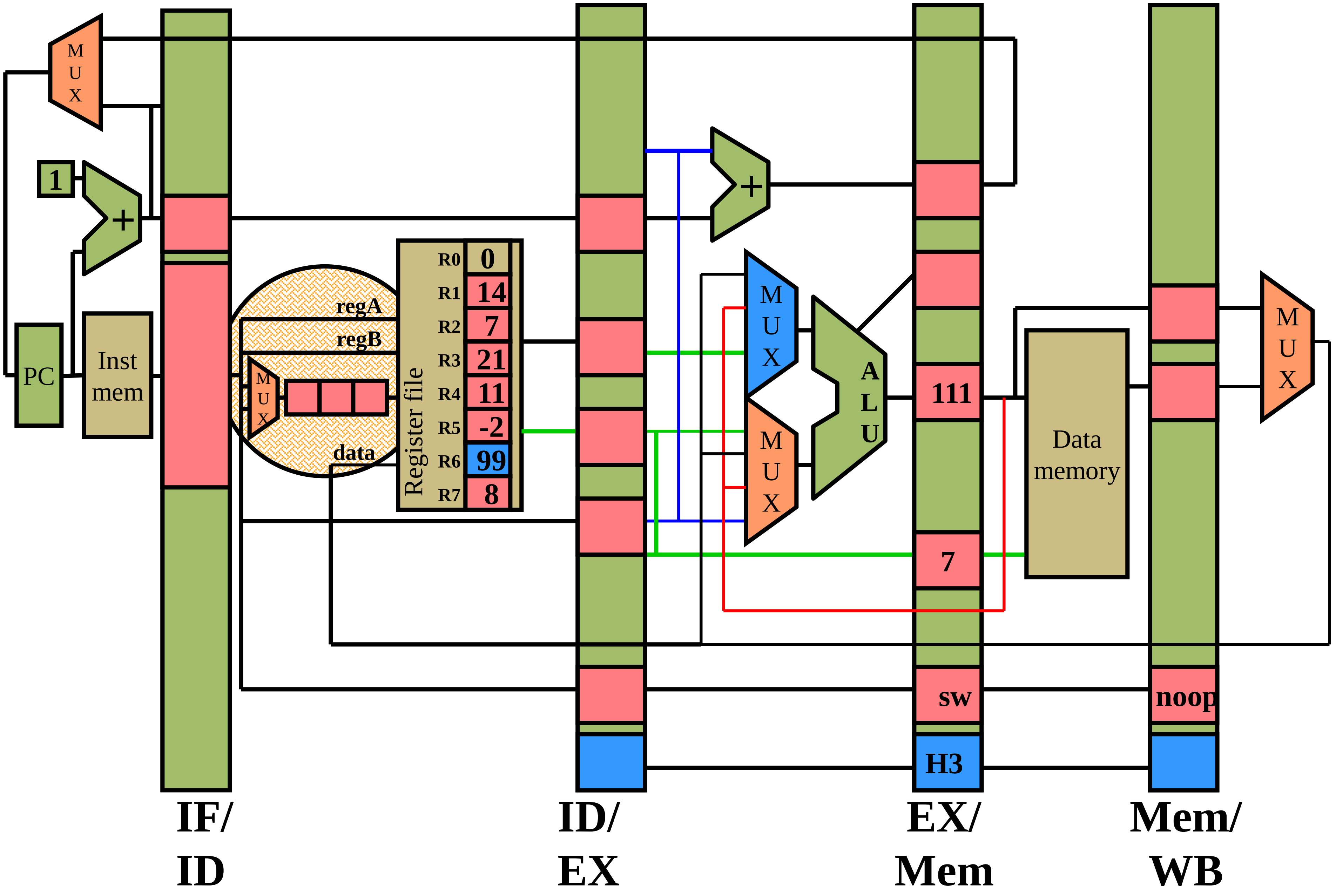
End of cycle 7



First half of cycle 8



End of cycle 8



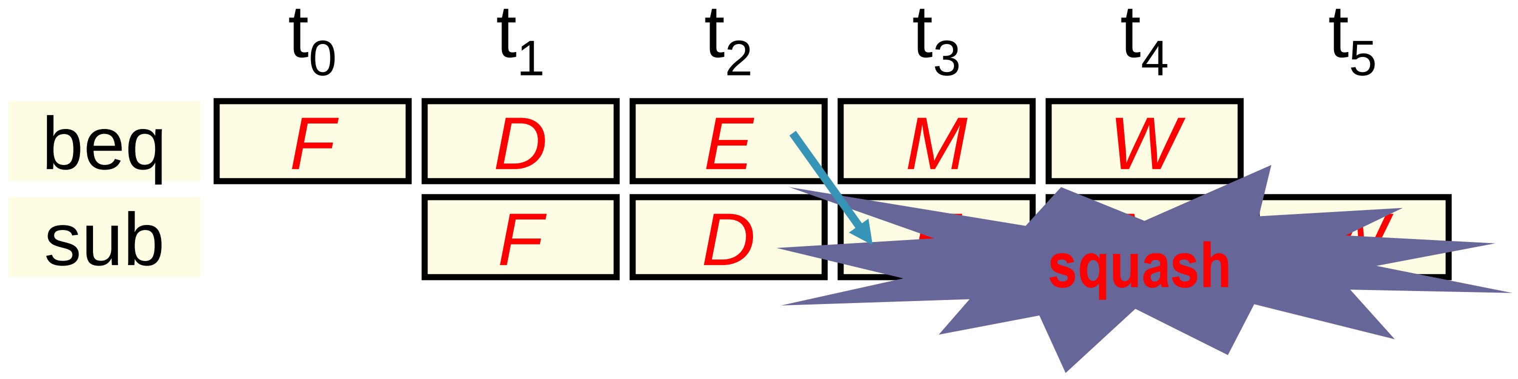
Pipeline function for BEQ

- Fetch: read instruction from memory
- Decode: read source operands from reg
- Execute: calculate target address and test for equality
- Memory: **Send target to PC** if test is equal
- Writeback: Nothing left to do

Control Hazards

beq 1 1 10

sub 3 4 5



Handling Control Hazards

Avoidance (static)
- No branches?
- Convert branches to predication
  - Control dependence becomes data dependence

Detect and Stall (dynamic)
- Stop fetch until branch resolves

Speculate and squash (dynamic)
- Keep going past branch, throw away instructions if wrong

Avoidance Via Predication

```
if (a == b) {
    x++;
    y = n / d;
}
```

```
sub     t1 ← a, b
jnz     t1, PC+2
add     x ← x, #1
div     y ← n, d
```

```
sub          t1 ← a, b
add(!t1)     x ← x, #1
div(!t1)     y ← n, d
```

```
sub          t1 ← a, b
add          t2 ← x, #1
div          t3 ← n, d
cmov(!t1)    x ← t2
cmov(!t1)    y ← t3
```

Handling Control Hazards: Detect & Stall

Detection

- In decode, check if opcode is branch or jump

Stall

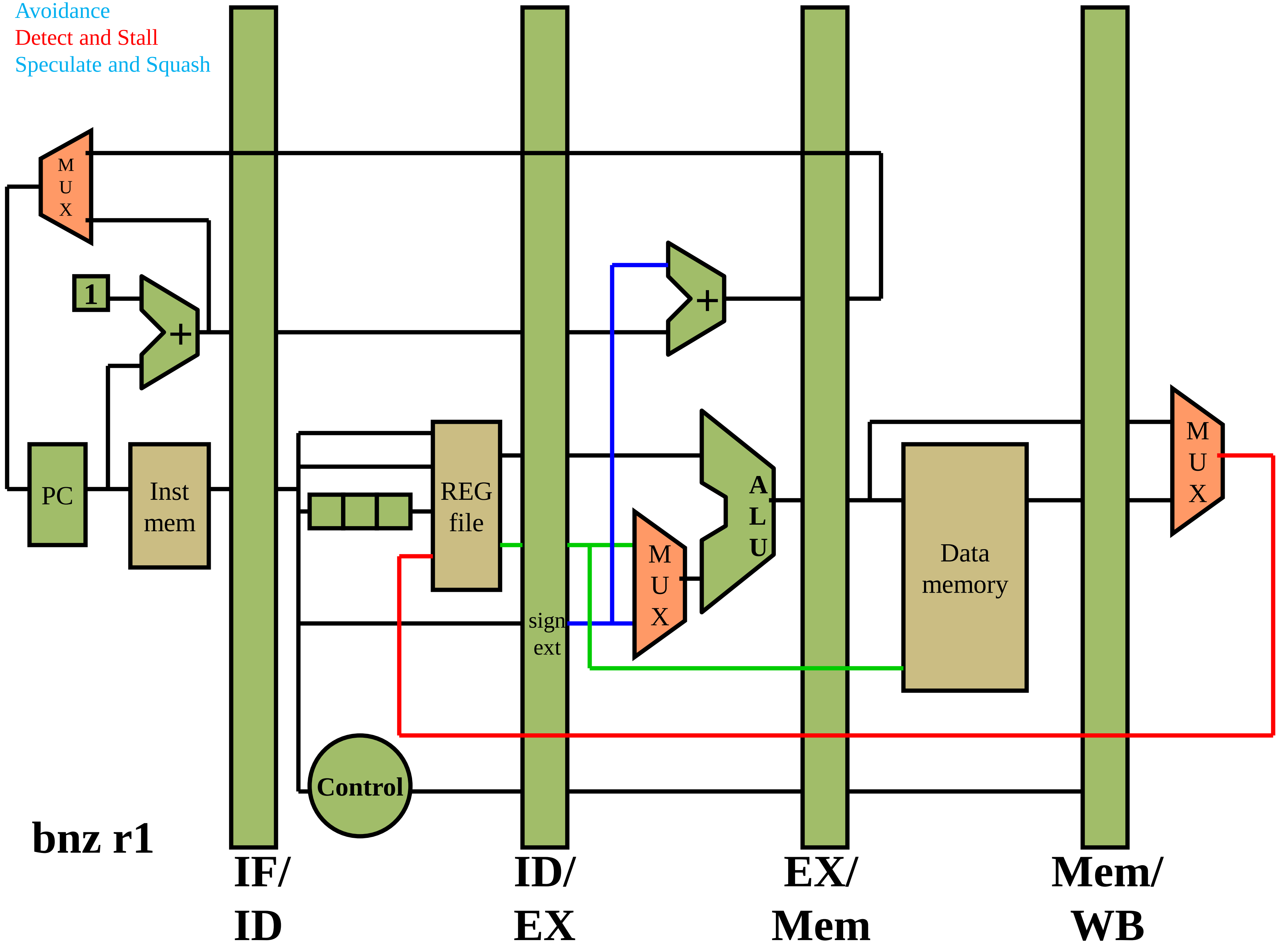
- Hold next instruction in Fetch
- Pass noop to Decode

Pipelining & Control Hazards

Avoidance

Detect and Stall

Speculate and Squash



Control Hazards

beq 1 1 10
sub 3 **4 5**

time

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

Problems with Detect & Stall

CPI increases on every branch

Are these stalls necessary? Not always!

- Branch is only taken half the time
- Assume branch is NOT taken
  - Keep fetching, treat branch as noop
  - If wrong, make sure bad instructions don't complete

Handling Control Hazards: Speculate & Squash

Speculate "Not-Taken"

- Assume branch is not taken

Squash

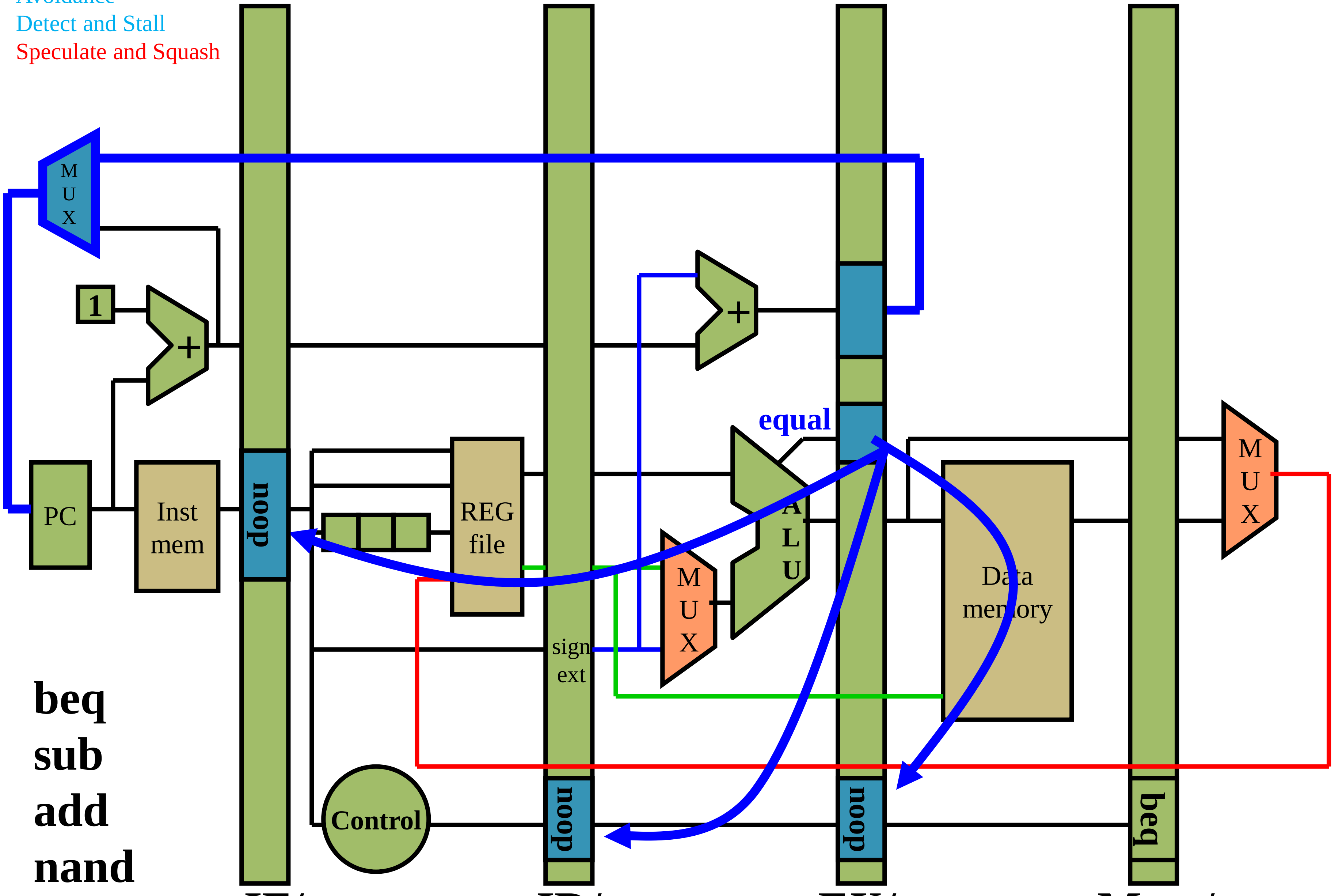
- Overwrite opcodes in Fetch, Decode, Execute with noop
- Pass target to Fetch

Pipelining & Control Hazards

Avoidance

Detect and Stall

Speculate and Squash



beq
sub
add
nand

IF/ ID

ID/ EX

EX/ Mem

Mem/ WB

Problems with Speculate & Squash

Always assumes branch is not taken

Can we do better? Yes.

- Predict branch direction and target!
- Why possible? Program behavior repeats.

More on branch prediction to come...

Branch Delay Slot (MIPS, SPARC)

	t_0	t_1	t_2	t_3	t_4	t_5	
branch:	F	D	E	M	W		
next:		F	Squash				
target:			F	D	E	M	W

- Instruction in delay slot executes even on taken branch

branch:	F	D	E	M	W		
delay:		F	D	E	M	W	
target:			F	D	E	M	W

```
i:      beq 1, 2, tgt
j:      add 3, 4, 5
```

What can we put here?

Improving pipeline performance

- Add more stages
- Widen pipeline

Adding pipeline stages

- Pipeline frontend
  - Fetch, Decode
- Pipeline middle
  - Execute
- Pipeline backend
  - Memory, Writeback

Adding stages to fetch, decode

- Delays hazard detection
- No change in forwarding paths
- No performance penalty with respect to data hazards

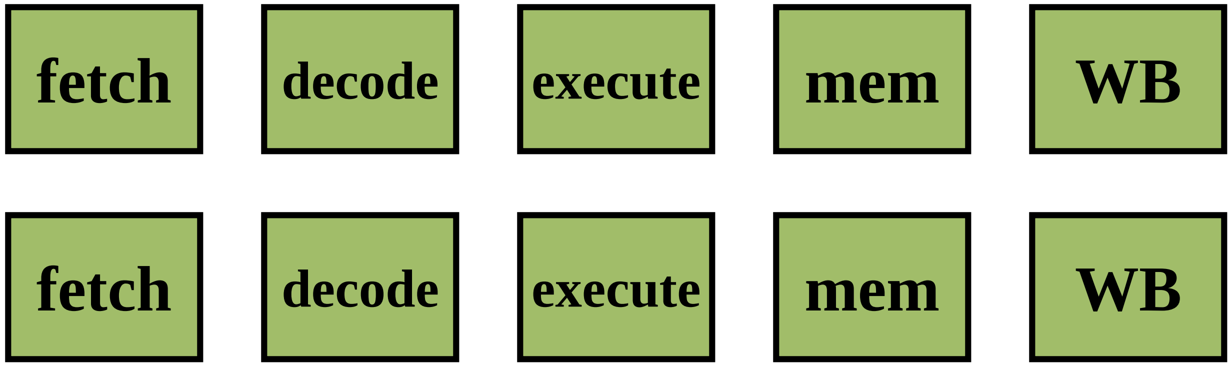
Adding stages to execute

- Check for structural hazards
  - ALU not pipelined
  - Multiple ALU ops completing at same time
- Data hazards may cause delays
  - If multicycle op hasn't computed data before the dependent instruction is ready to execute
- Performance penalty for each stall

Adding stages to memory, writeback

- Instructions ready to execute may need to wait longer for multi-cycle memory stage
- Adds more pipeline registers
  - Thus more source registers to forward
    - More complex hazard detection
    - Wider muxes
    - More control bits to manage muxes

Wider pipelines



More complex hazard detection

- **2X pipeline registers to forward from**
- **2X more instructions to check**
- **2X more destinations (muxes)**
- **Need to worry about dependent instructions in the same stage**

Making forwarding explicit

- add r1 $\leftarrow$ r2, EX/Mem ALU result
  - Include direct mux controls into the ISA
  - Hazard detection is now a compiler task
  - New micro-architecture leads to new ISA
    - Is this why this approach always seems to fail? (e.g., simple VLIW, Motorola 88k)
  - Can reduce some resources
    - Eliminates complex conflict checkers