Pipelining

Anne Bracy

CS 3410

Computer Science

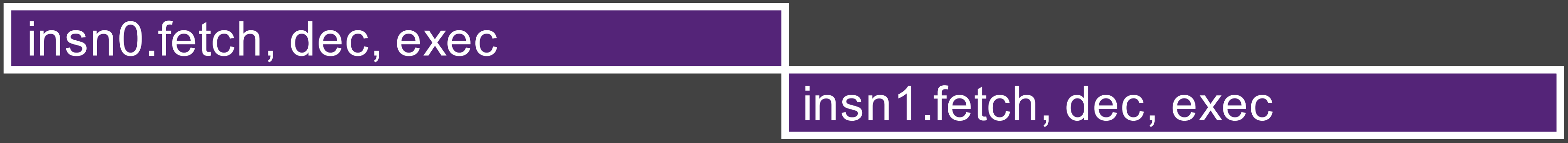
Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.

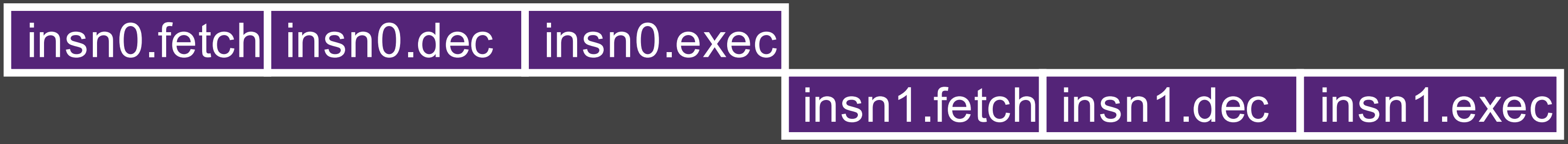
See P&H Chapter: 4.5-4.8

Single Cycle → Multi-Cycle → Pipelining

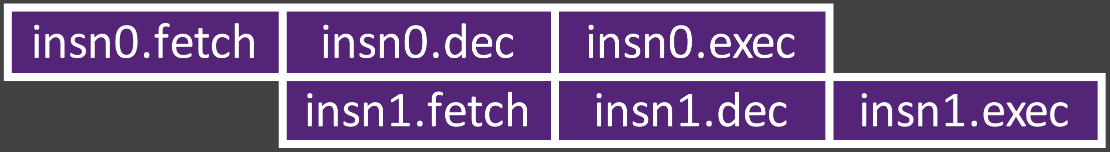
Single-cycle



Multi-cycle



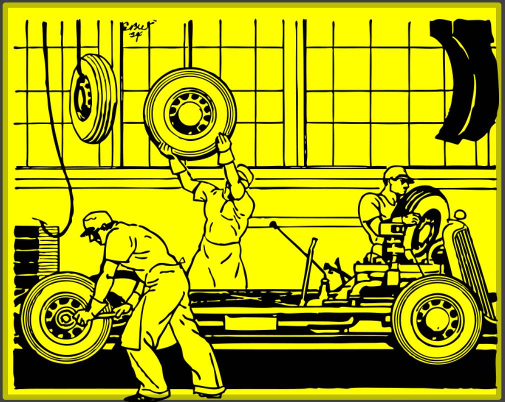
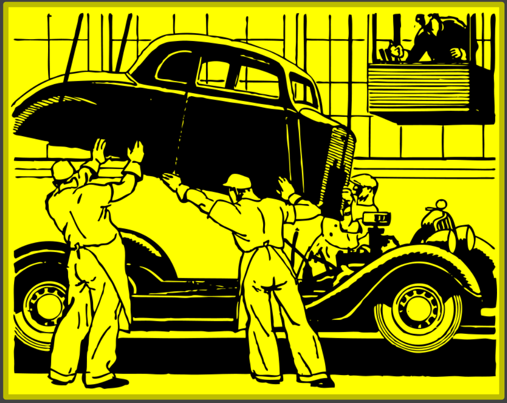
Pipelined



Agenda

5-stage Pipeline

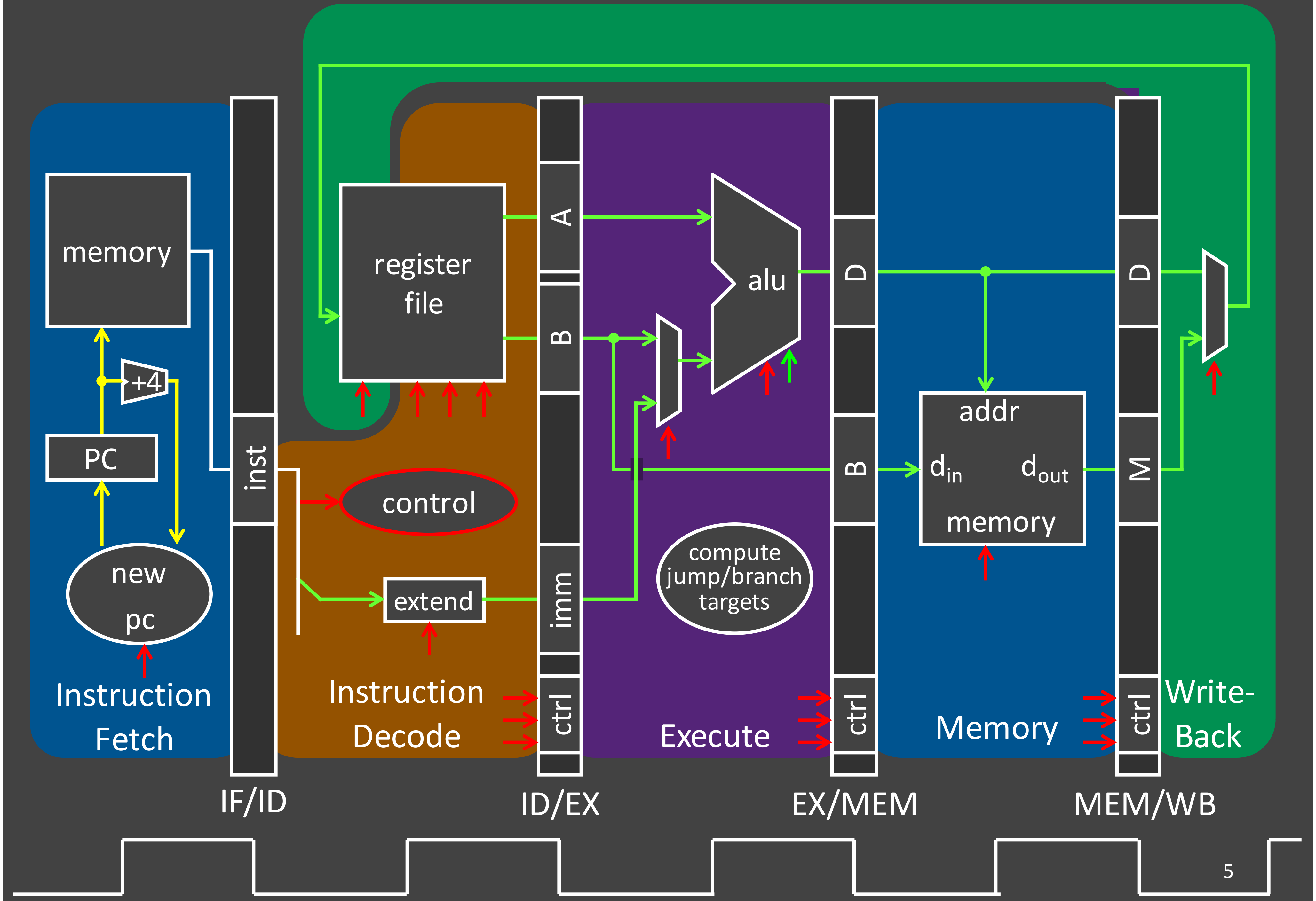
- Implementation
- Working Example

Hazards

- Structural
- Data Hazards
- Control Hazards

Pipelined Processor



Time Graphs

Cycle	1	2	3	4	5	6	7	8	9
add	IF	ID	EX	MEM	WB				
nand		IF	ID	EX	MEM	WB			
lw			IF	ID	EX	MEM	WB		
add				IF	ID	EX	MEM	WB	
sw					IF	ID	EX	MEM	WB

↓

Latency: 5 cycles

Throughput: 1 insn/cycle

CPI = 1

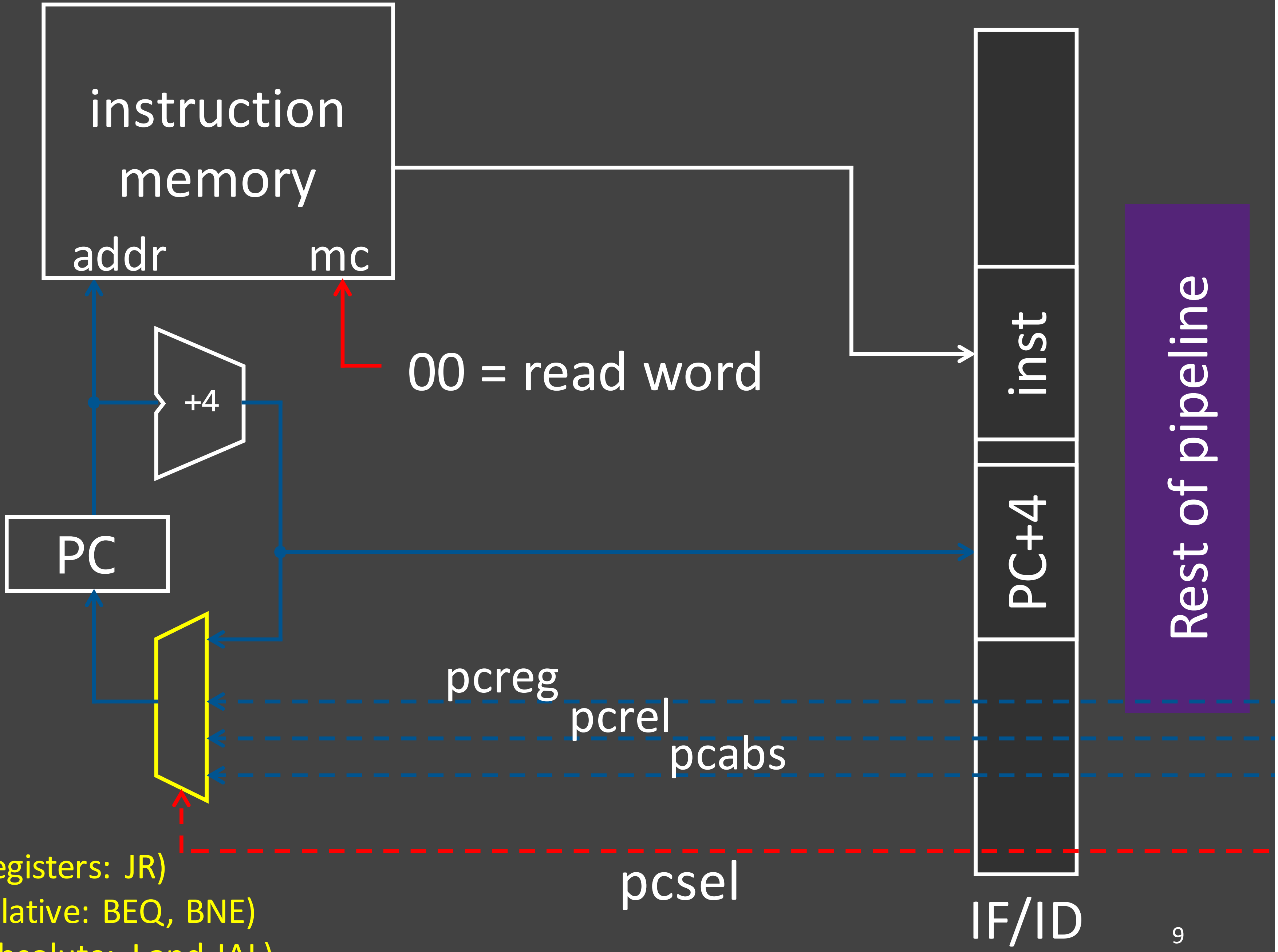
Principles of Pipelined Implementation

- Break datapath into multiple cycles (here 5)
    - Parallel execution increases throughput
    - Balanced pipeline very important
        - Slowest stage determines clock rate
        - Imbalance kills performance
- Add pipeline registers (flip-flops) for isolation
    - Each stage begins by reading values *from* latch
    - Each stage ends by writing values *to* latch
- Resolve hazards

Pipeline Stages

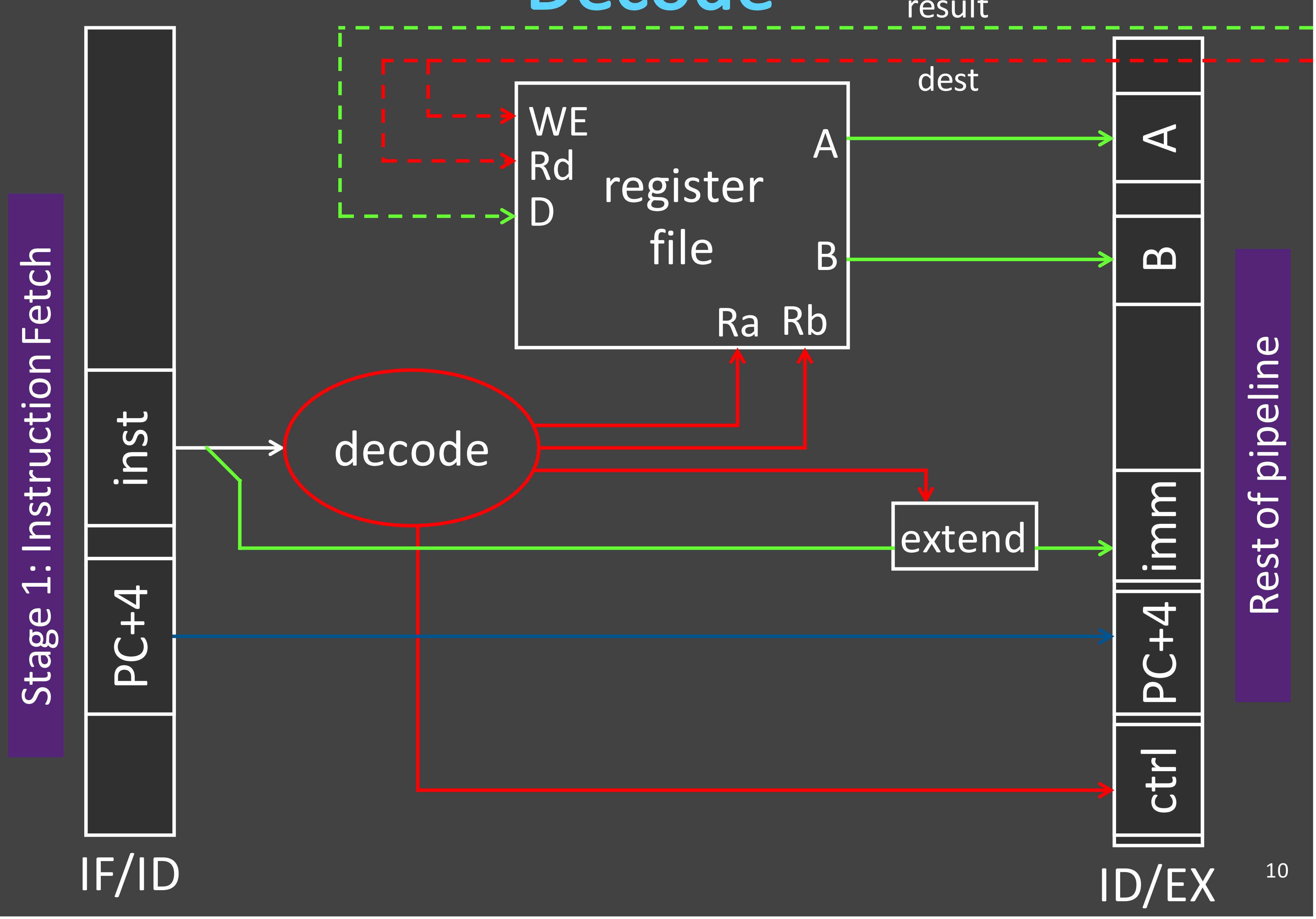
Stage	Perform Functionality	Latch values of interest
Fetch	Use PC to index Program Memory, increment PC	Instruction bits (to be decoded) PC + 4 (to compute branch targets)
Decode	Decode instruction, generate control signals, read register file	Control information, Rd index, immediates, offsets, register values (Ra, Rb), PC+4 (to compute branch targets)
Execute	Perform ALU operation Compute targets (PC+4+offset, etc.) in case this is a branch, decide if branch taken	Control information, Rd index, *etc.* Result of ALU operation, value in case this is a store instruction
Memory	Perform load/store if needed, address is ALU result	Control information, Rd index, *etc.* Result of load, pass result from execute
Writeback	Select value, write to register file	

Instruction Fetch

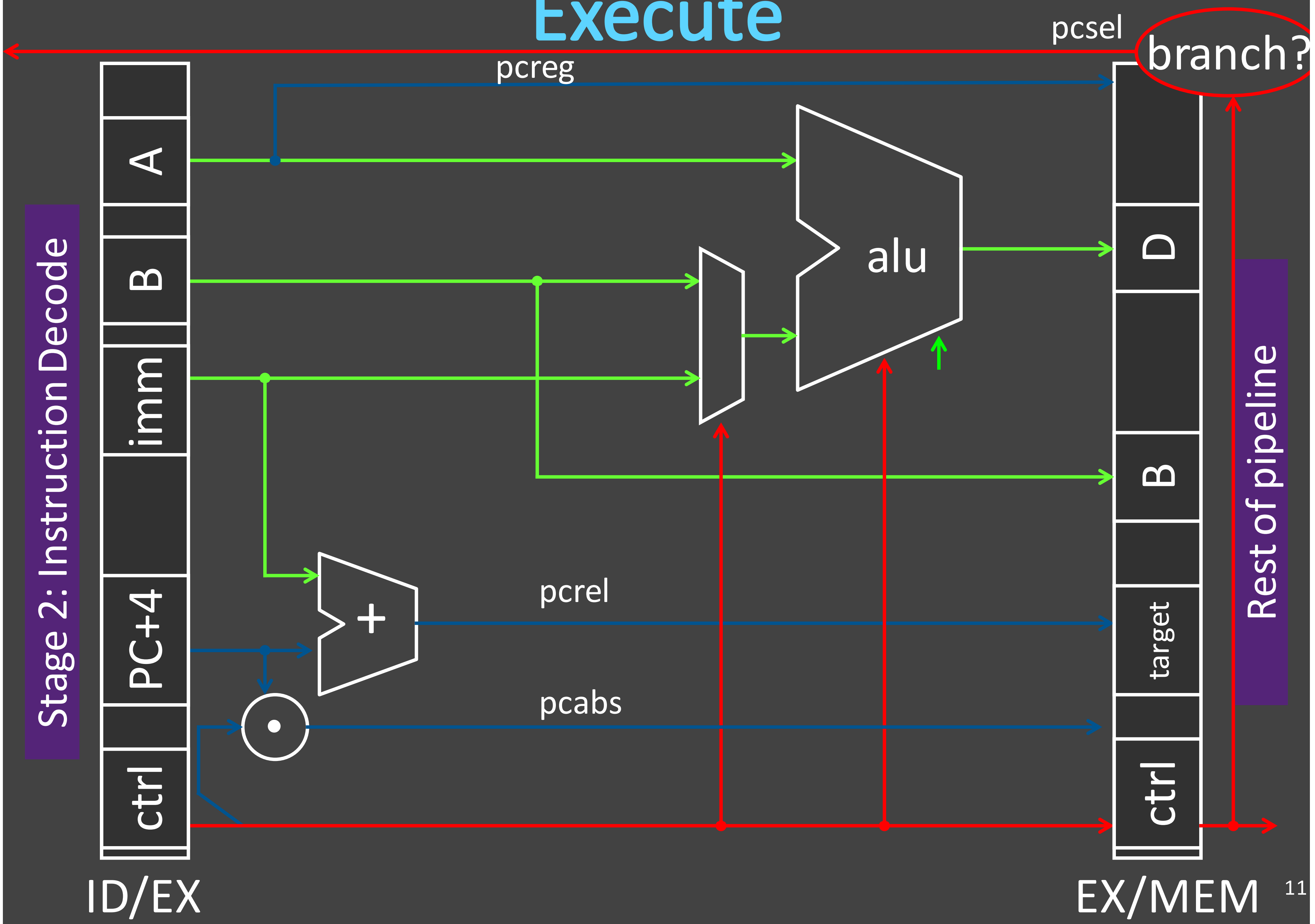


- PC+4
- pcreg (PC registers: JR)
- pcrel (PC-relative: BEQ, BNE)
- pcabs (PC absolute: J and JAL)

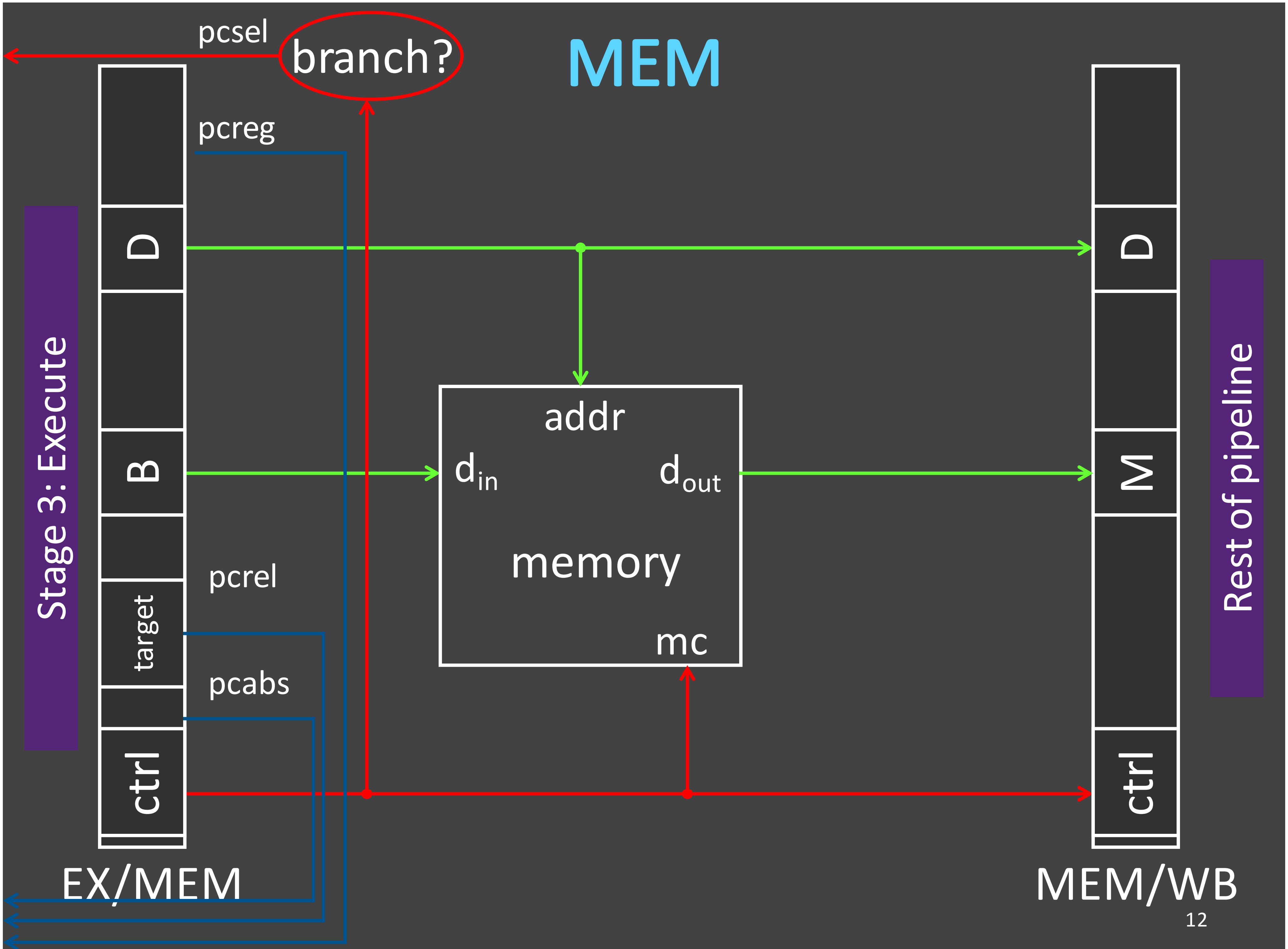
Decode


Stage 1: Instruction Fetch
inst
PC+4
IF/ID
result
dest
WE
Rd
D
register file
A
B
Ra
Rb
decode
extend
A
B
imm
PC+4
ctrl
ID/EX
Rest of pipeline

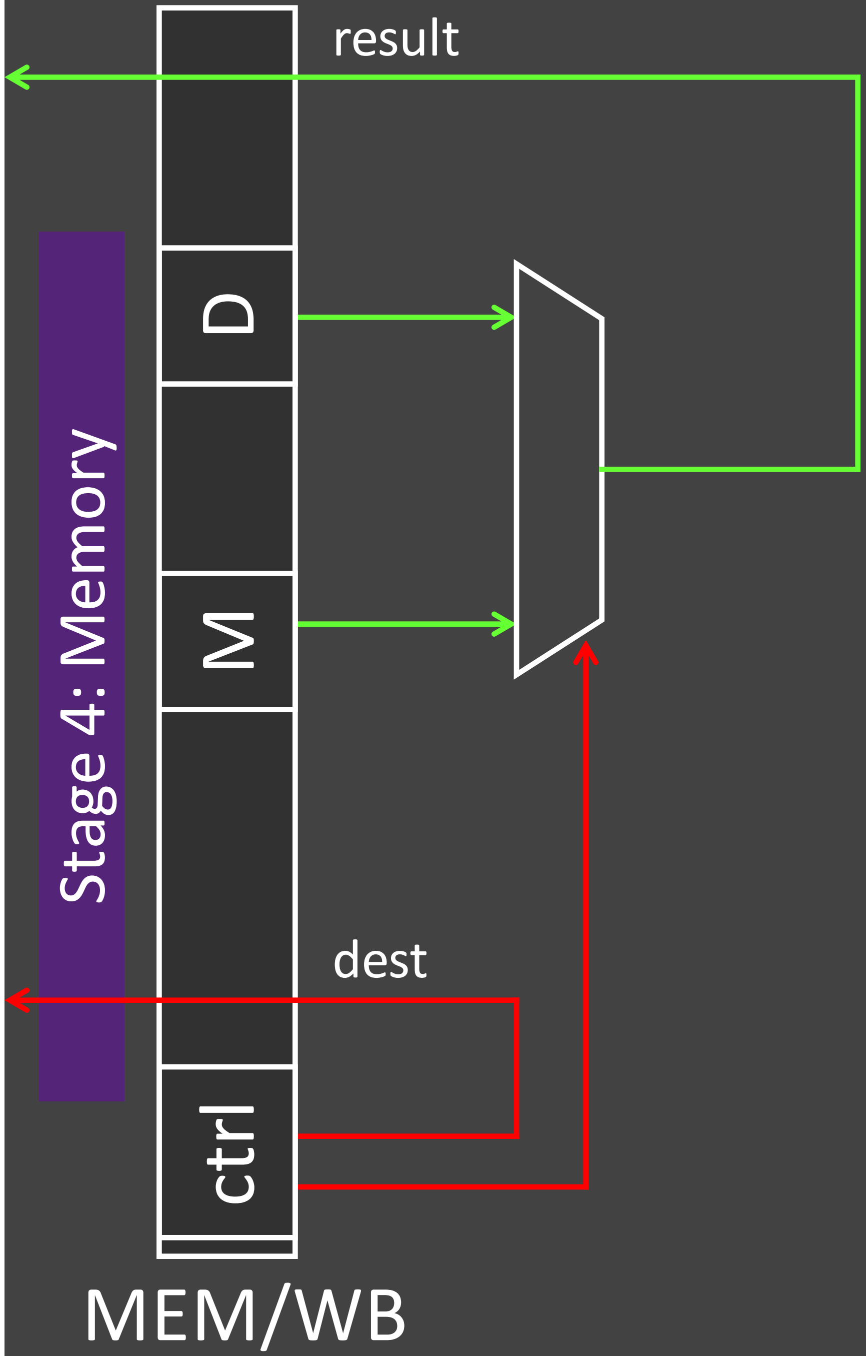
Execute


pcsel
branch?
pcreg
A
B
imm
PC+4
ctrl
alu
+
pcrel
pcabs
D
B
target
ctrl
Stage 2: Instruction Decode
Rest of pipeline
ID/EX
EX/MEM

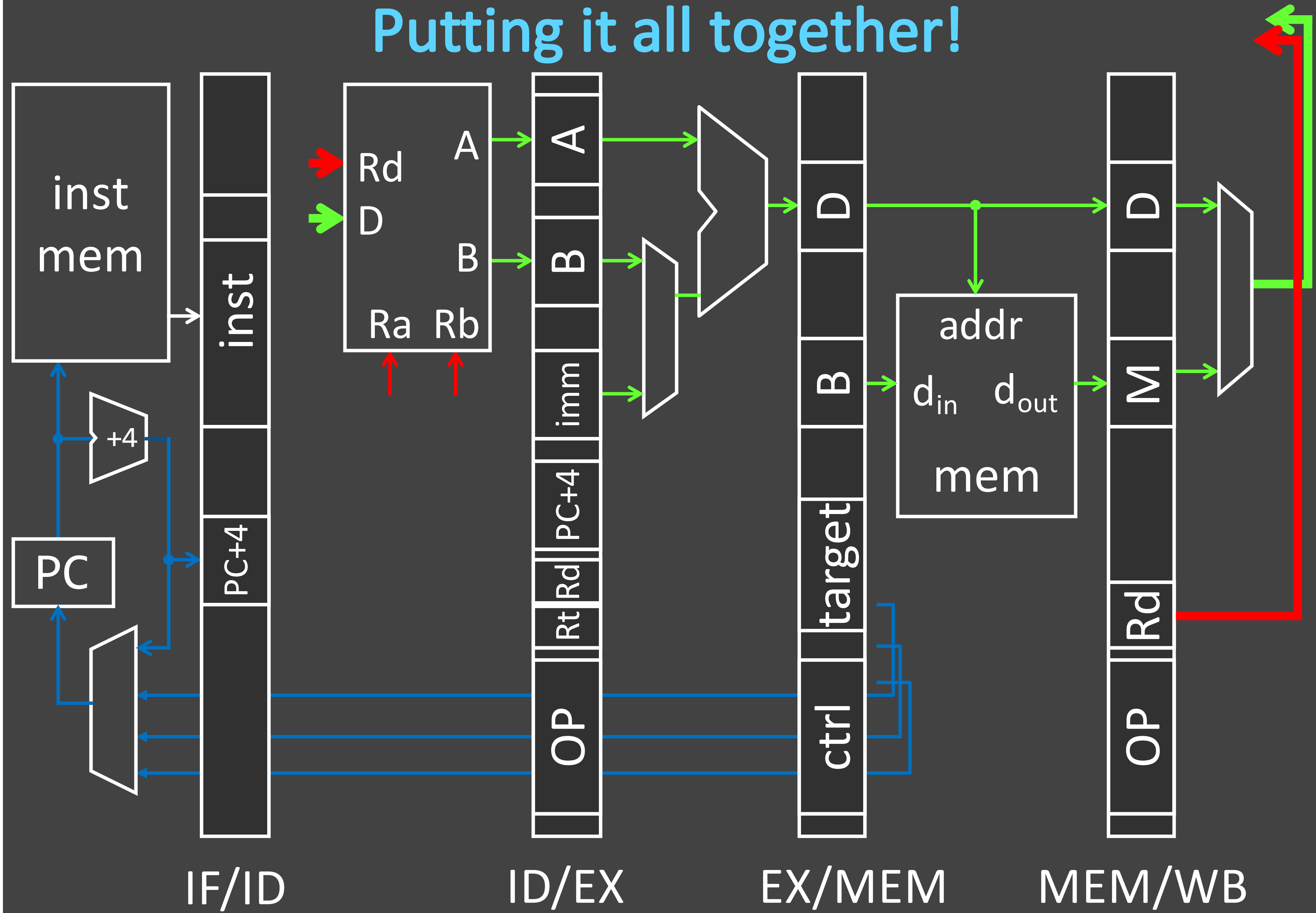
MEM


pcsel
branch?
pcreg
Stage 3: Execute
D
B
target
ctrl
pcrel
pcabs
EX/MEM
addr
d_{in}
d_{out}
memory
mc
D
M
ctrl
Rest of pipeline
MEM/WB

WB



Putting it all together!



MIPS designed for pipelining

- Instructions same length
    - 32 bits, easy to fetch and then decode

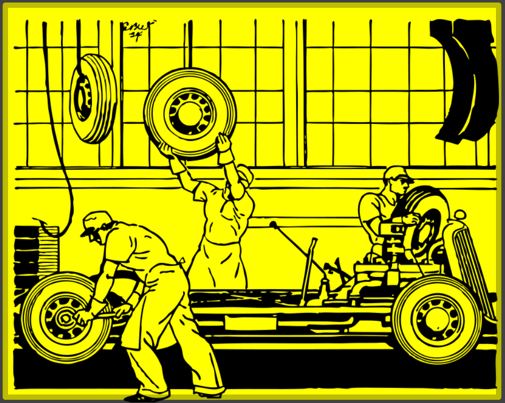
- 3 types of instruction formats
    - Easy to route bits between stages
    - Can read a register source before even knowing what the instruction is
- Memory access through lw and sw only
    - Access memory after ALU

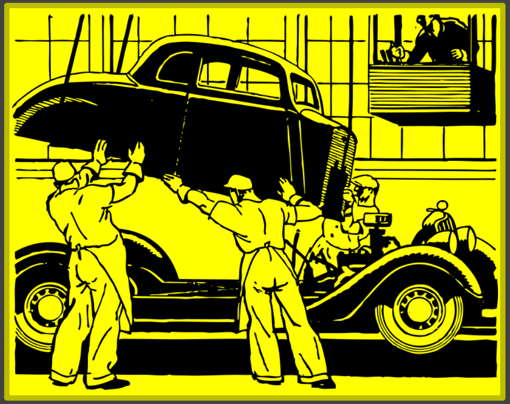
iClicker Question

Agenda

5-stage Pipeline

- Implementation
- Working Example





Hazards

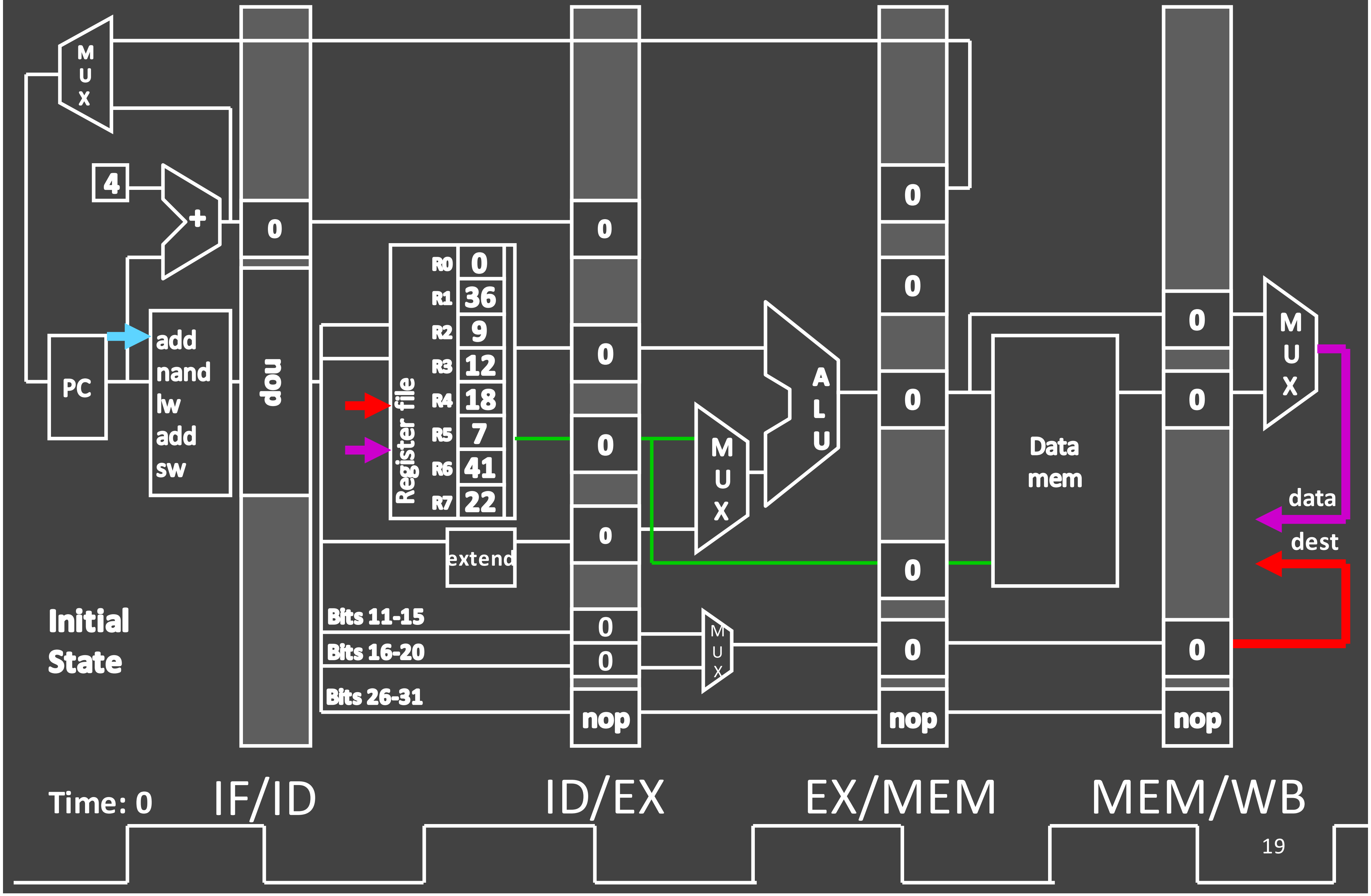
- Structural
- Data Hazards
- Control Hazards

Example: : Sample Code (Simple)

```
add     r3 ← r1, r2
nand    r6 ← r4, r5
lw      r4 ← 20(r2)
add     r5 ← r2, r5
sw      r7 → 12(r3)
```

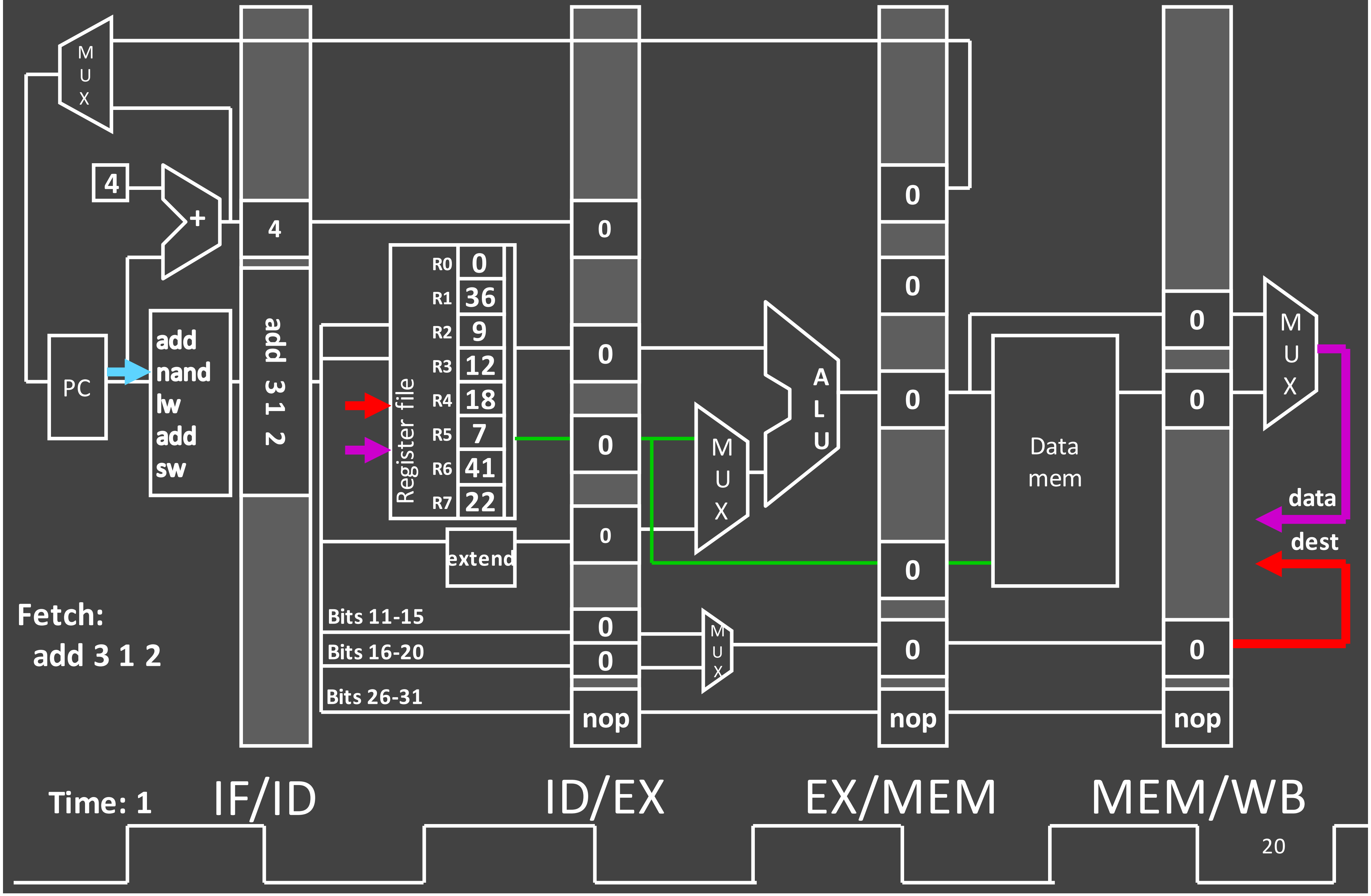
Assume 8-register machine

Example: Start State @ Cycle 0



Cycle 1: Fetch add

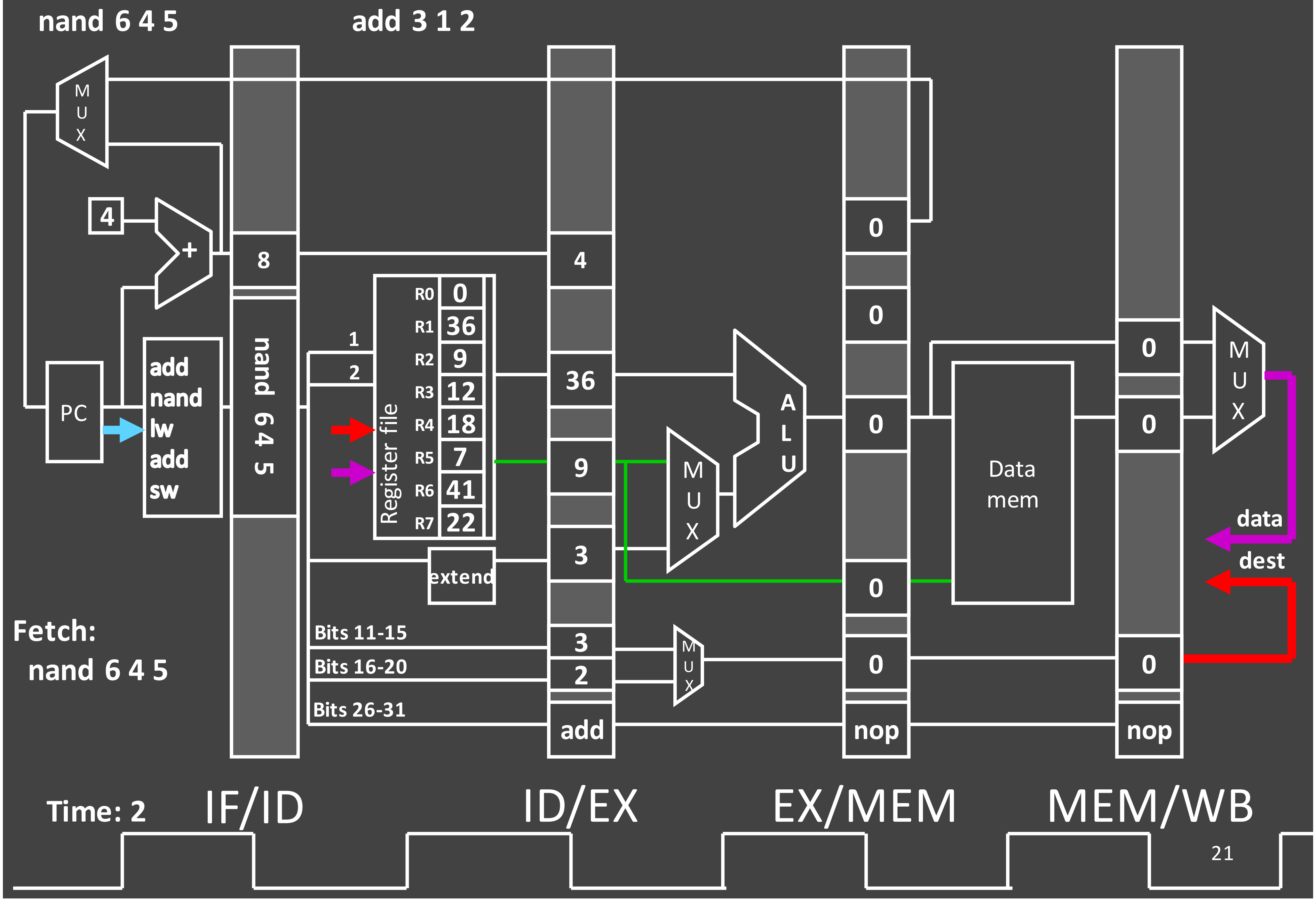
add 3 1 2



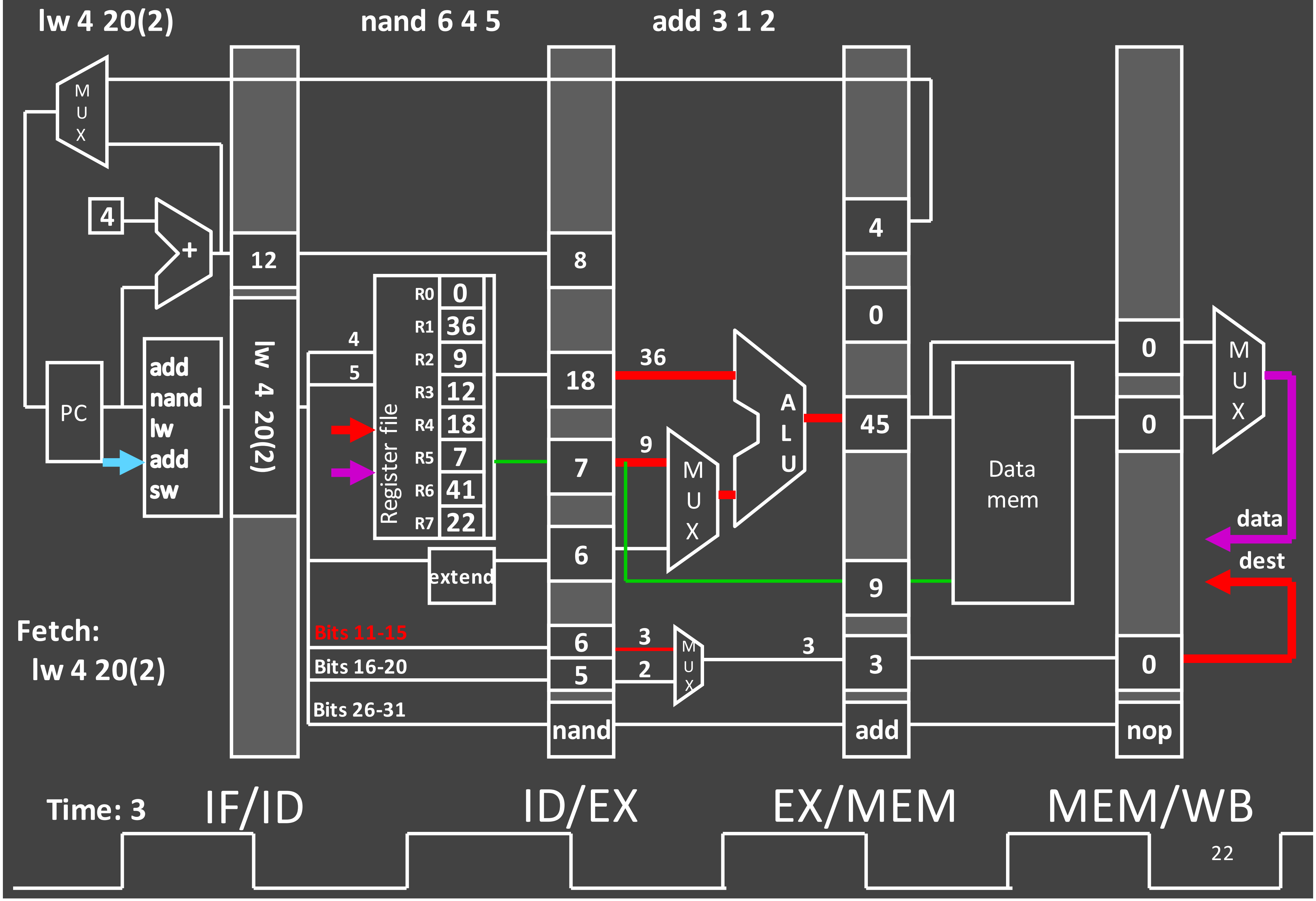
Fetch:
add 3 1 2

Time: 1

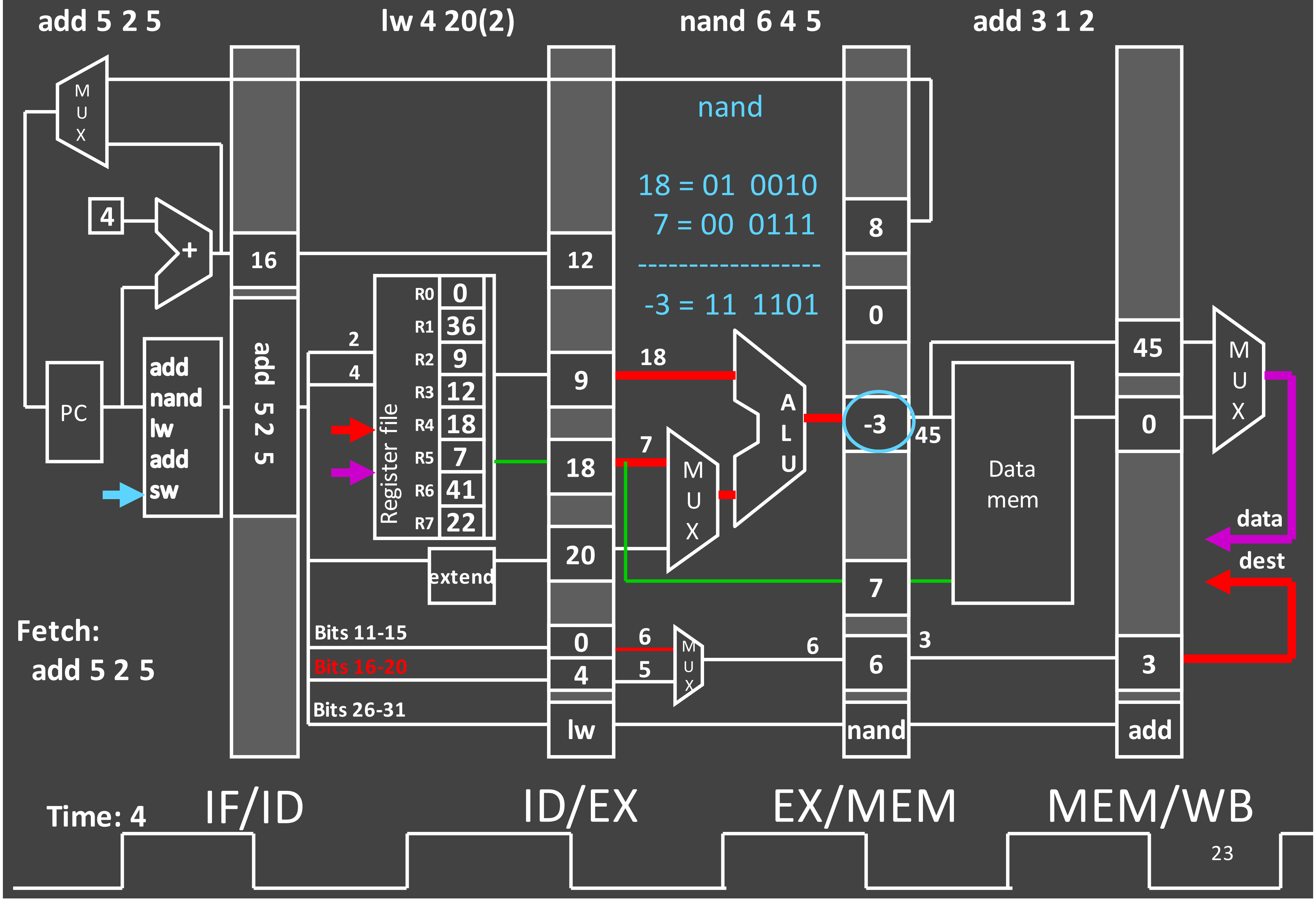
Cycle 2: Fetch nand, Decode add



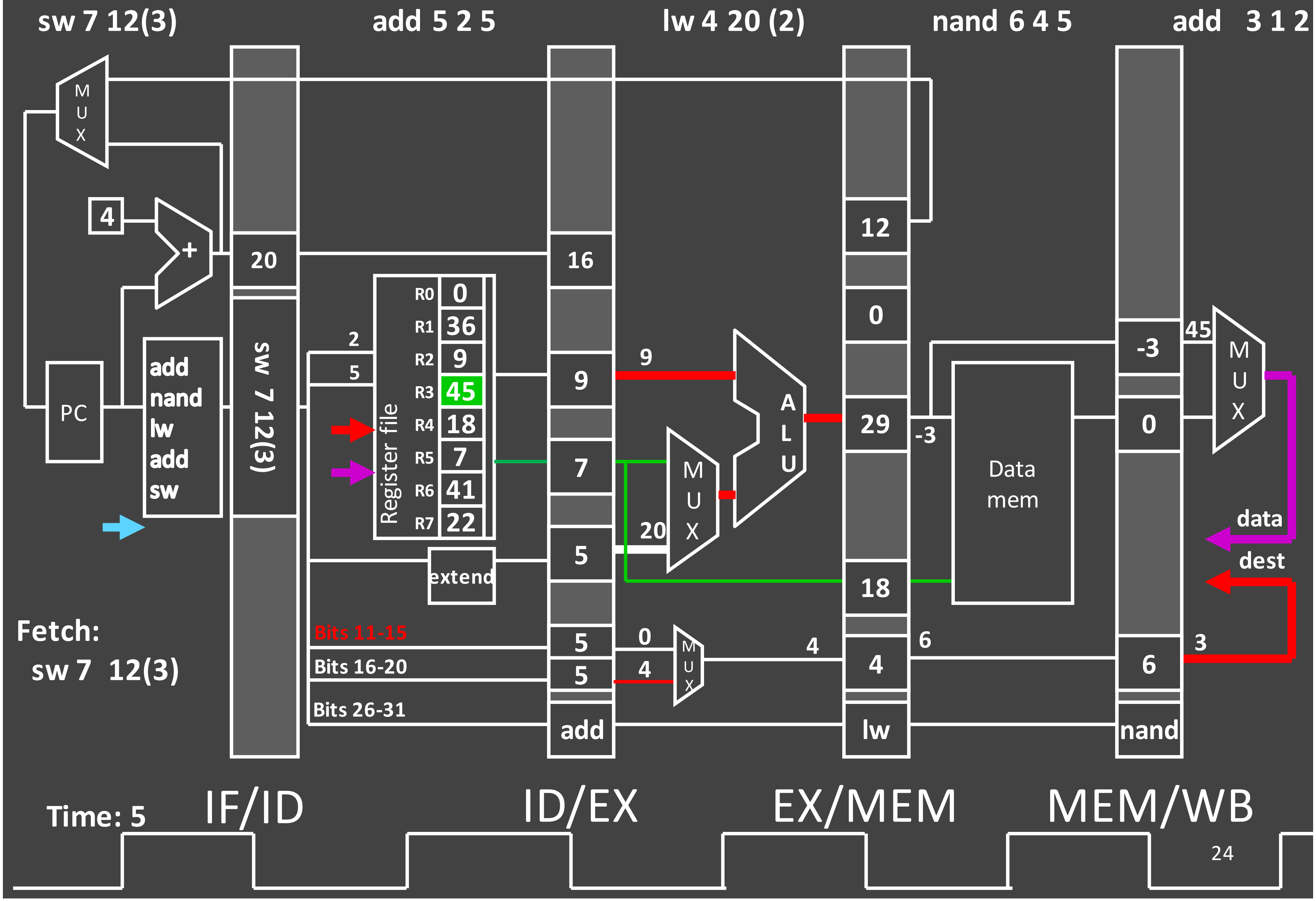
Cycle 3: Fetch lw, Decode nand, …



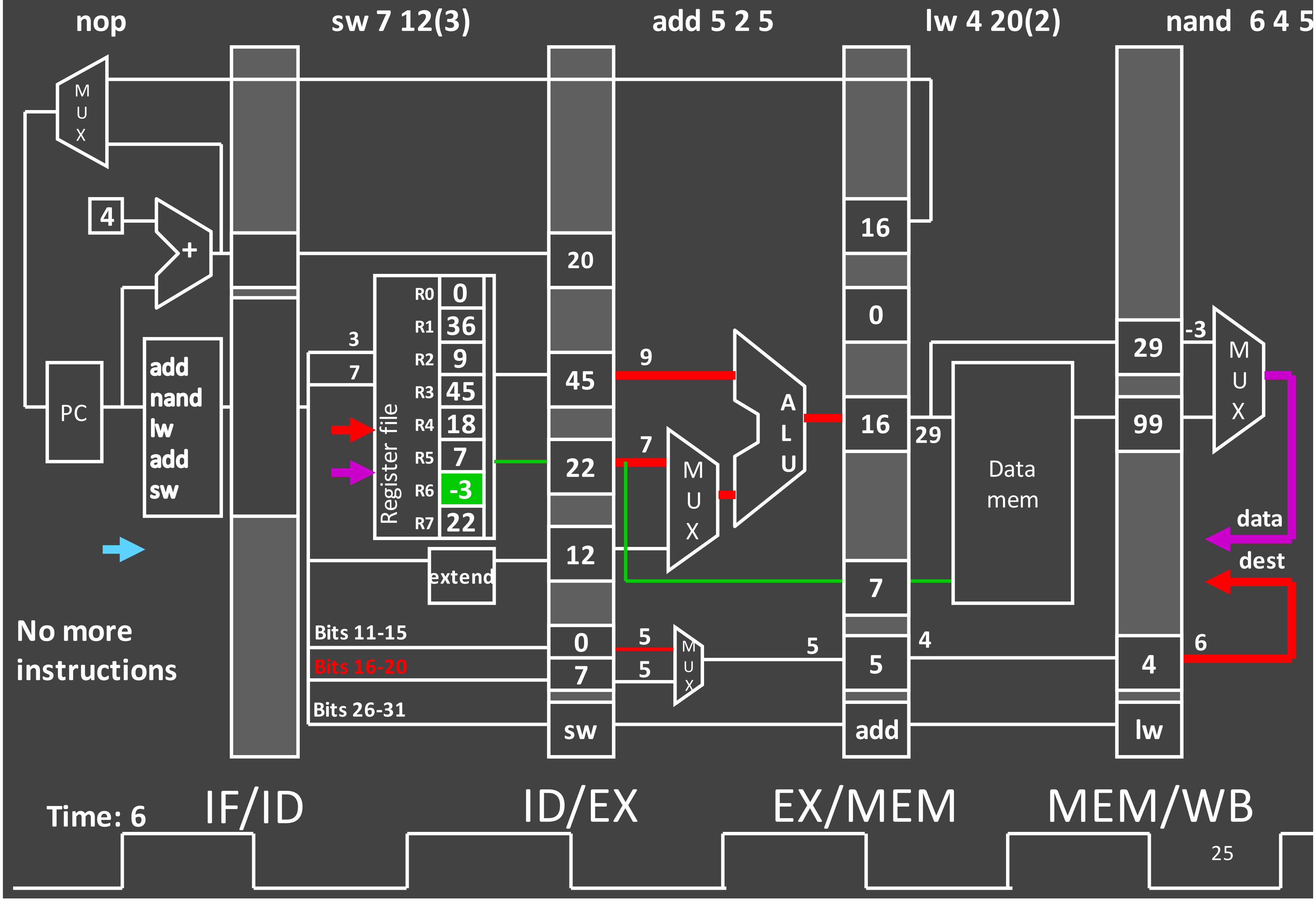
Cycle 4: Fetch add, Decode lw, …



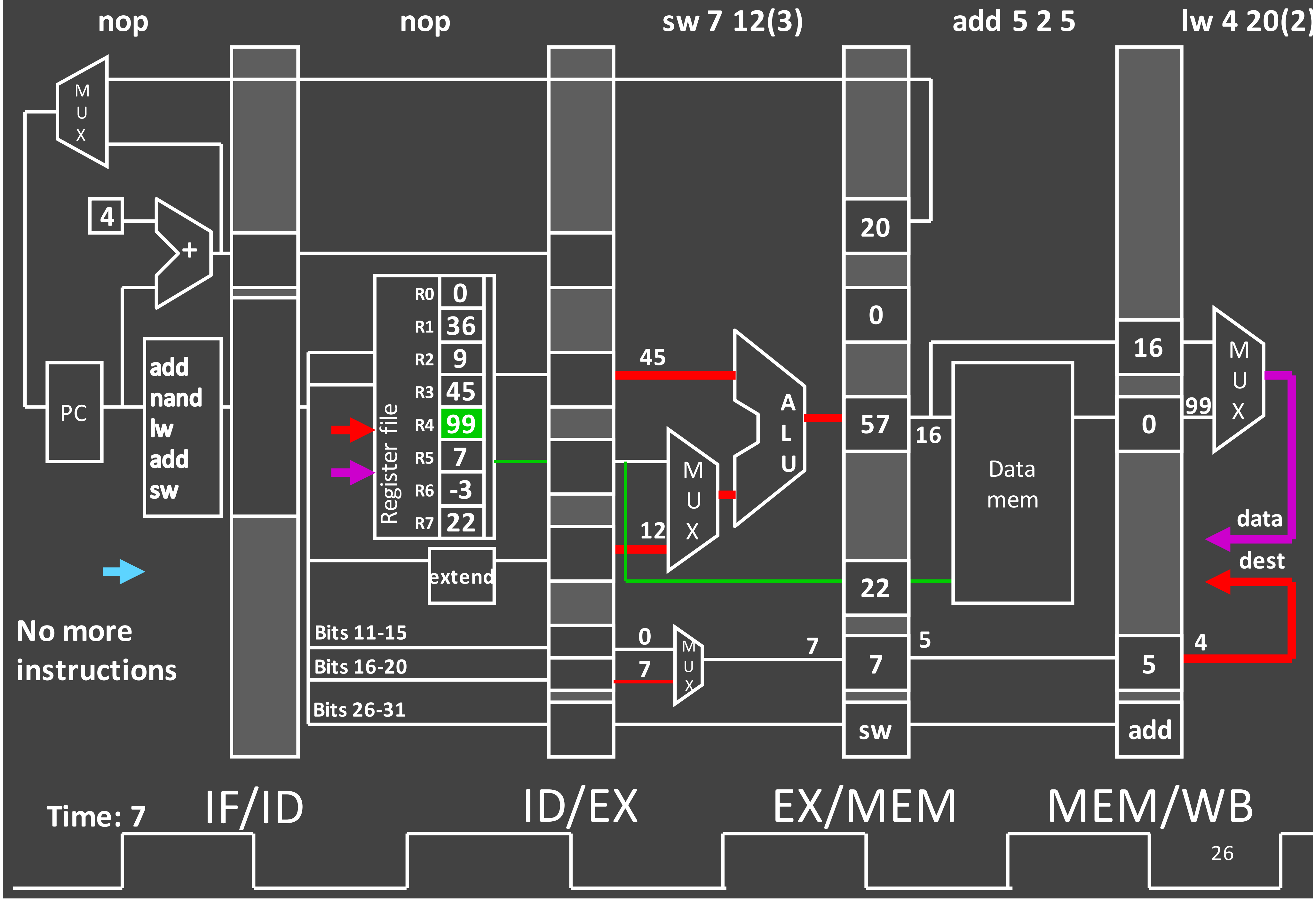
Cycle 5: Fetch sw, Decode add, ...



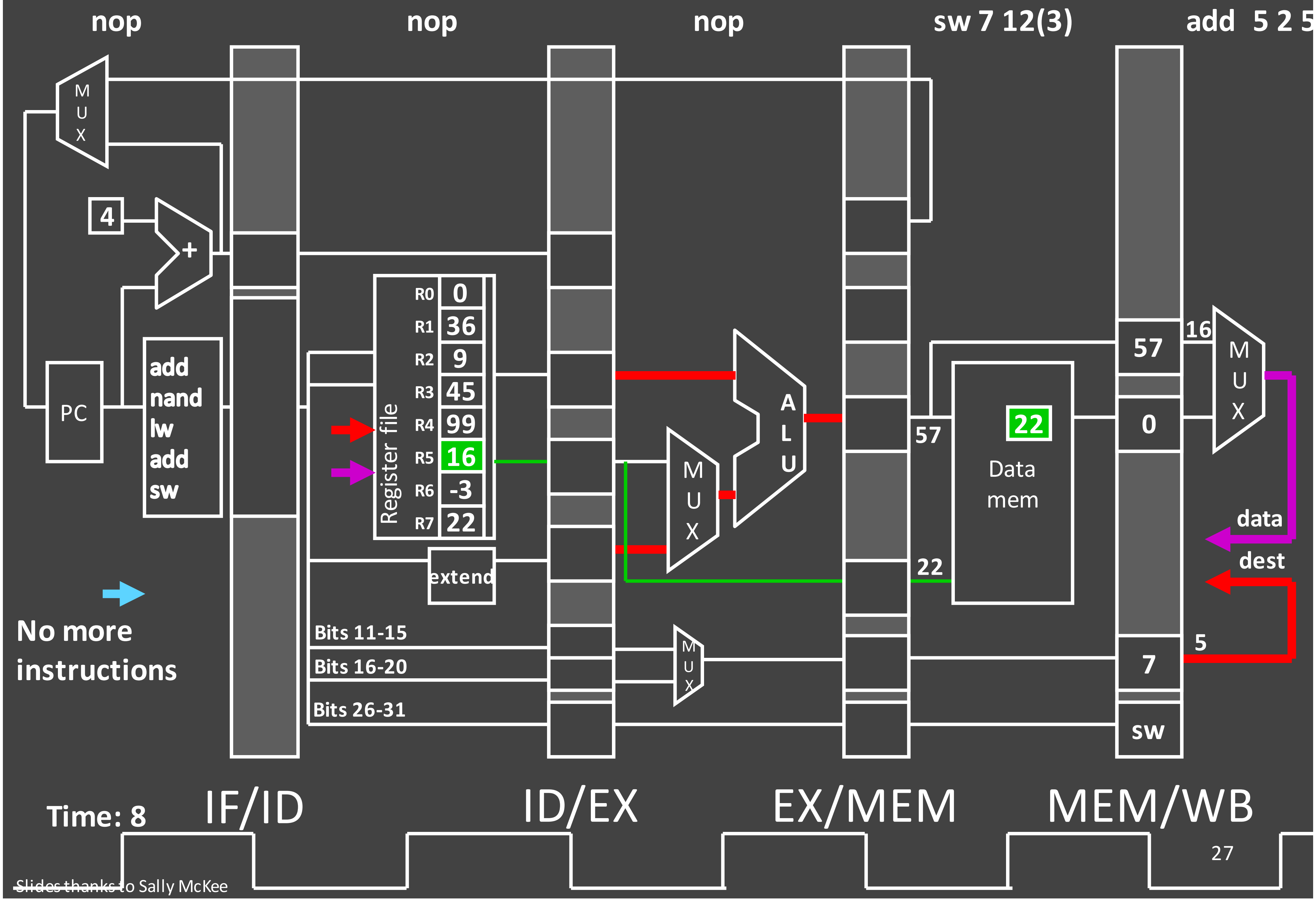
Cycle 6: Decode sw, ...



Cycle 7: Execute sw, ...



Cycle 7: Memory sw, ...

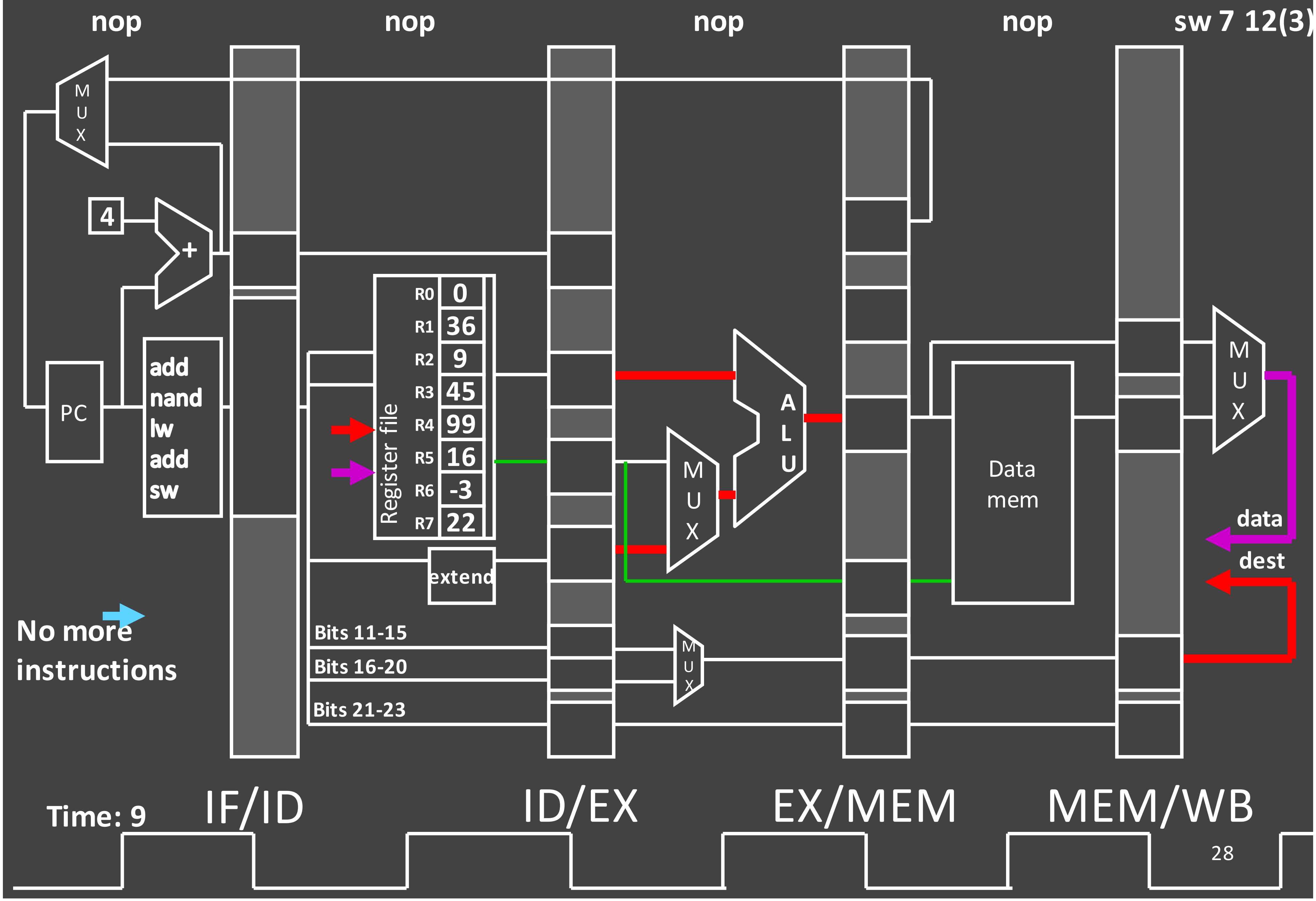


No more instructions

Time: 8

Slides thanks to Sally McKee

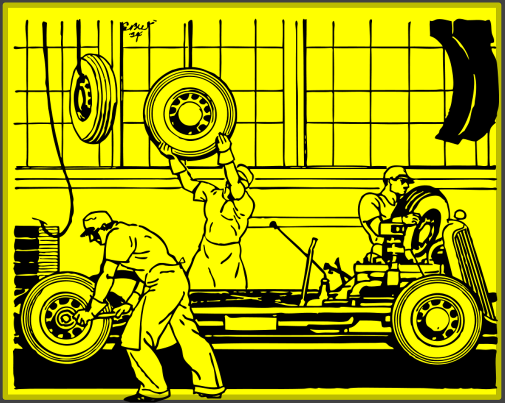
Cycle 7: Writeback sw, ...

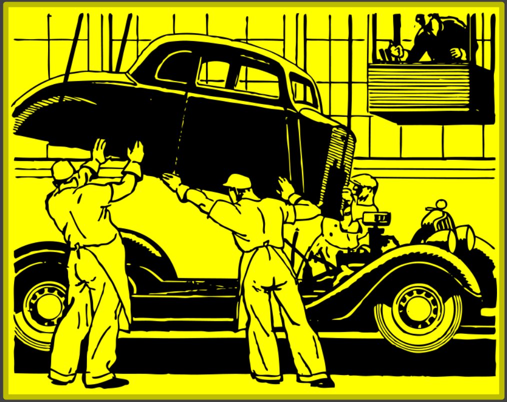


Agenda

5-stage Pipeline

- Implementation
- Working Example





Hazards

- Structural
- Data Hazards
- Control Hazards

Hazards

Correctness problems associated w/processor design

1. Structural hazards

   Same resource needed for different purposes at the same time (Possible: ALU, Register File, Memory)

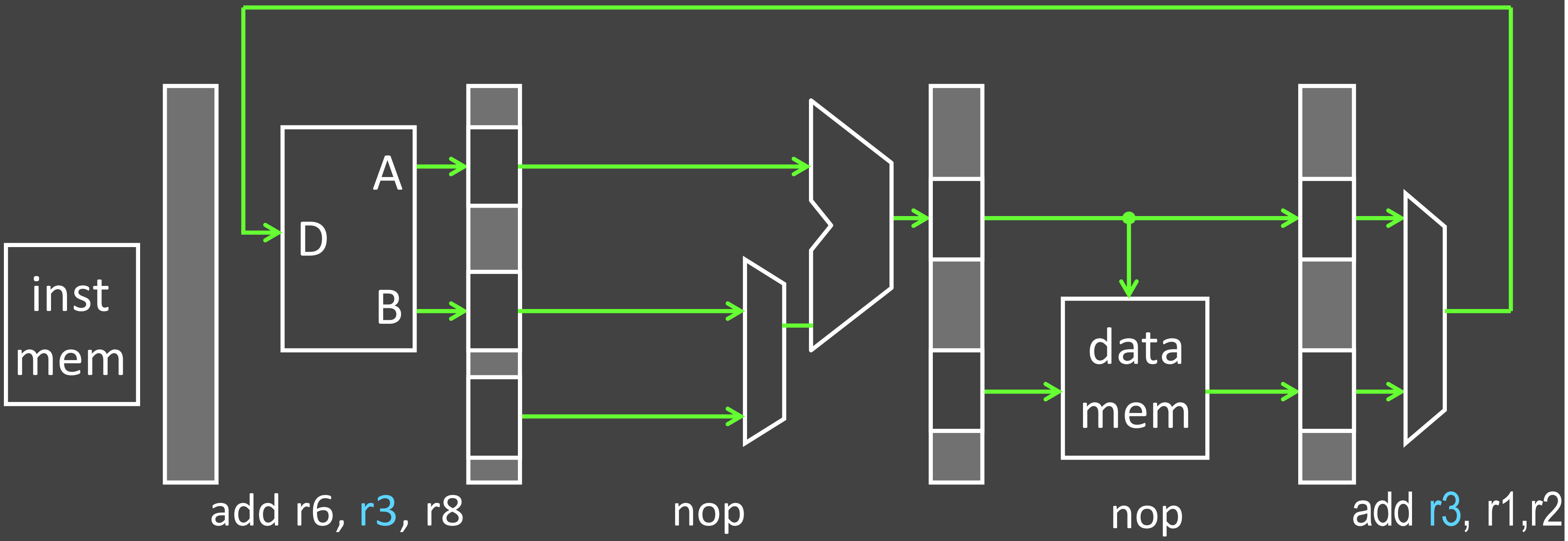
2. Data hazards

   Instruction output needed before it’s available

3. Control hazards

   Next instruction PC unknown at time of Fetch

Resolving Register File Structural Hazard



add r3, r1, r2	IF	ID	Ex	M	W				
nop		IF	ID	Ex	M	W			
nop			IF	ID	Ex	M	W		
add r6, r3, r8				IF	ID	Ex	M	W	

Problem: Need to read a value that is currently being written

Solution: negate RF clock: write first half, read second half

Dependences and Hazards

Dependence: relationship between two insns

- **Data:** two insns use same storage location
- **Control:** 1 insn affects whether another executes at all
- *Not a bad thing*, programs would be boring otherwise
- Enforced by making older insn go before younger one
  - Happens naturally in single-/multi-cycle designs
  - But not in a pipeline

Hazard: dependence & possibility of wrong insn order

- Effects of wrong insn order cannot be externally visible
- *Hazards are a bad thing*: most solutions either complicate the hardware or reduce performance

Data Hazards

Data Hazards

- register file reads occur in stage 2 (ID)
- register file writes occur in stage 5 (WB)
- next instructions may read values about to be written

```
add r3, r1, r2
sub r5, r3, r4
```

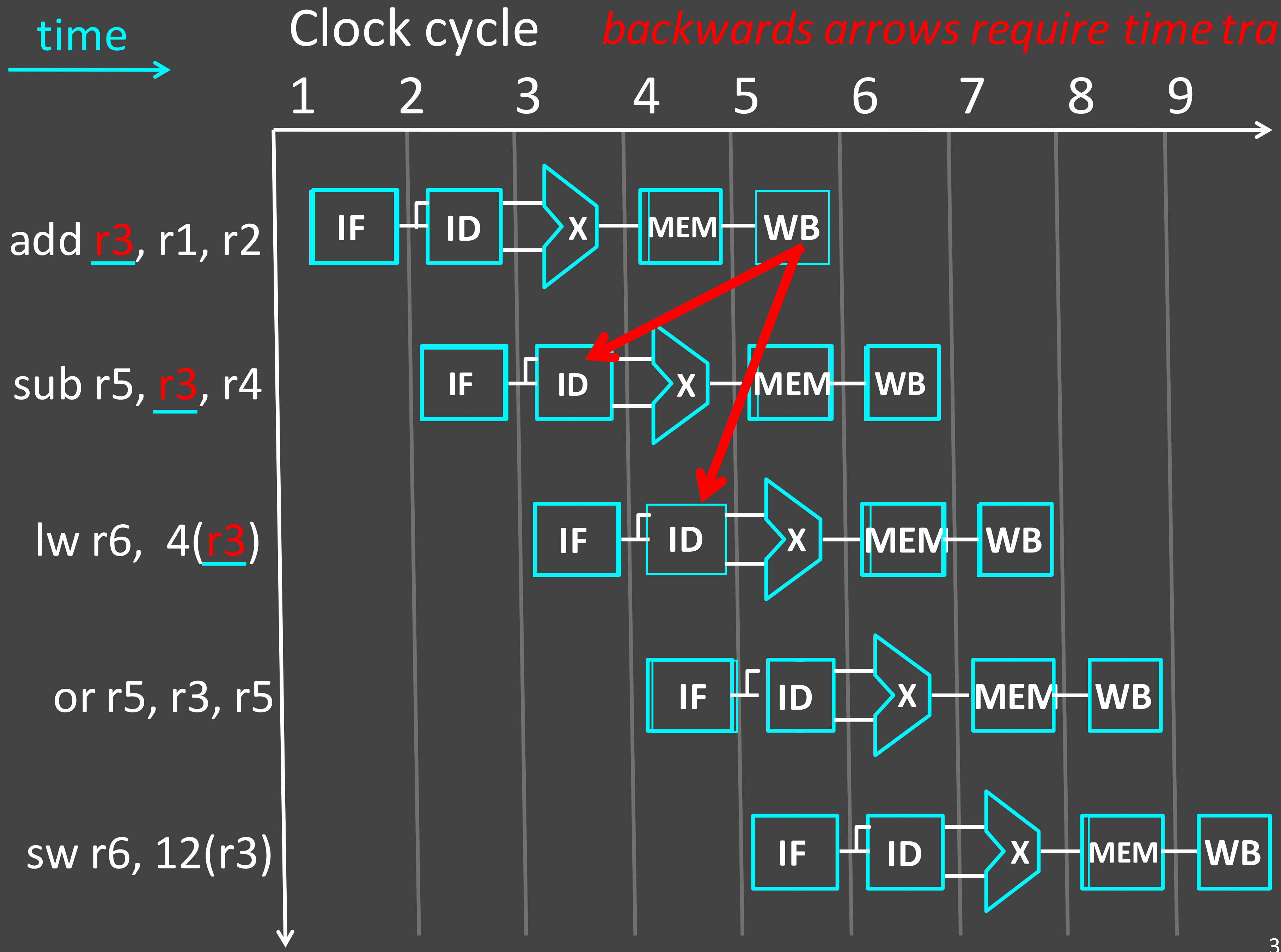
Is there a dependence?

Is there a hazard?

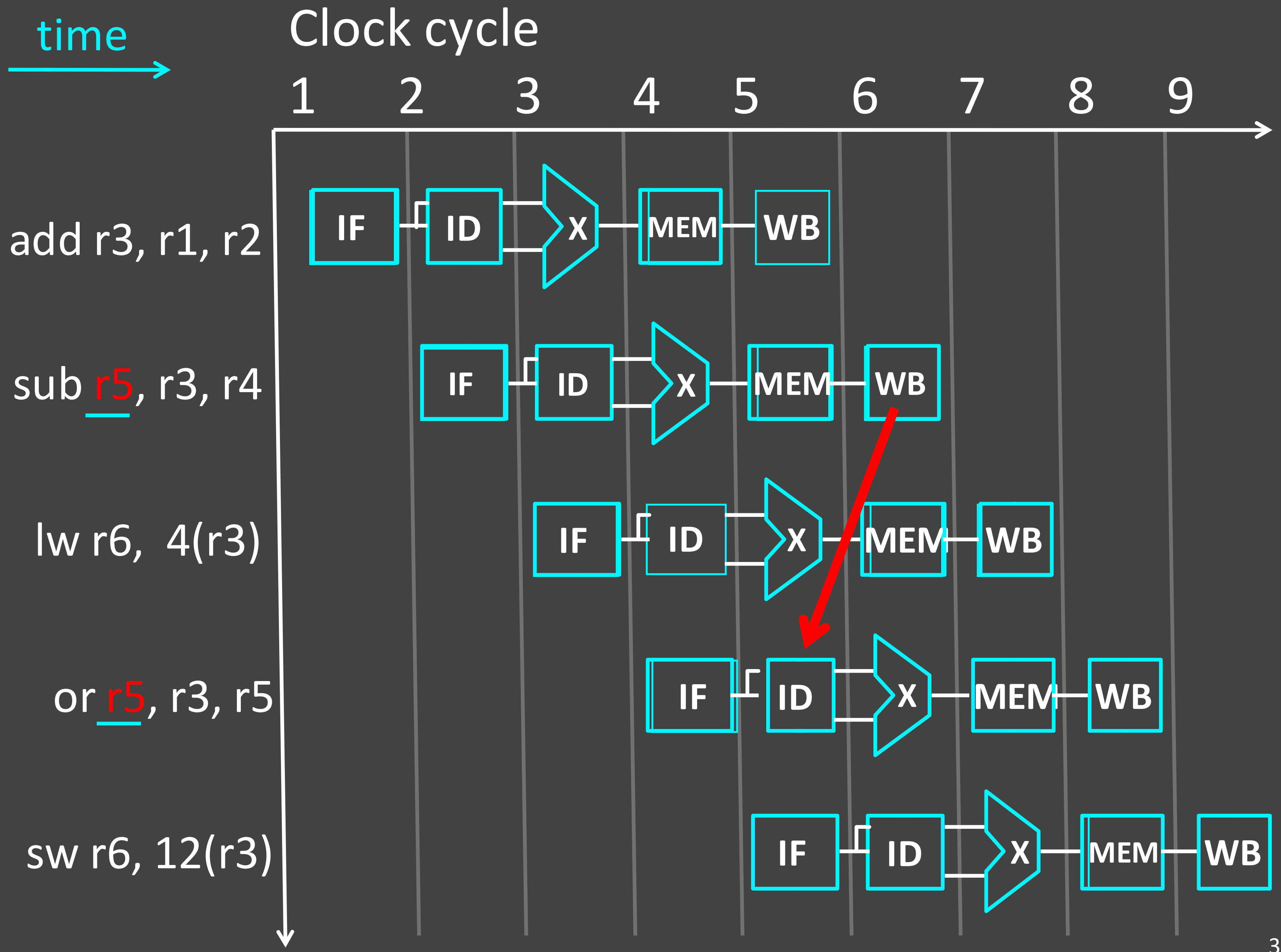
How do we detect this?

Visualizing Data Hazards (1)

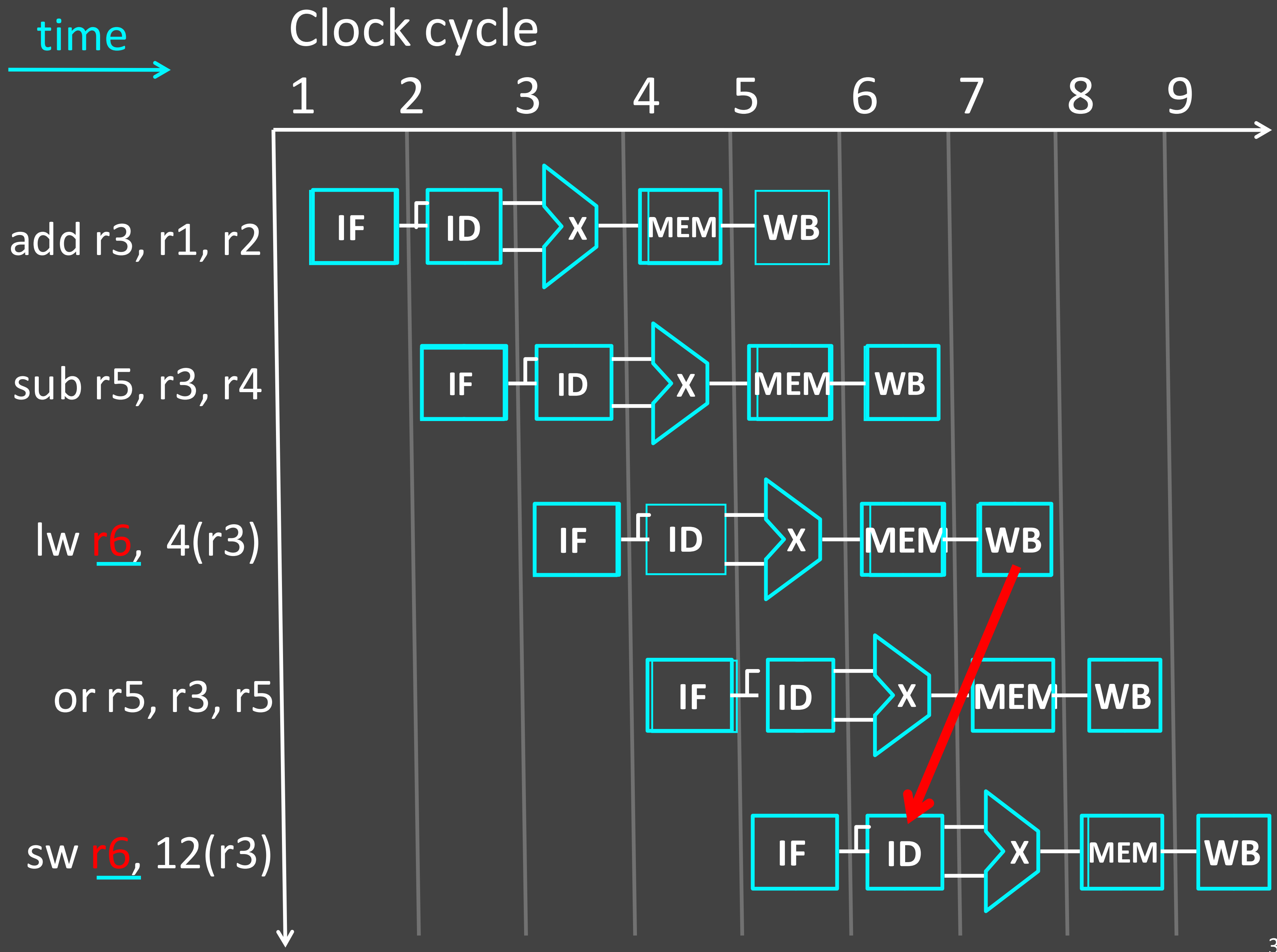
backwards arrows require time travel



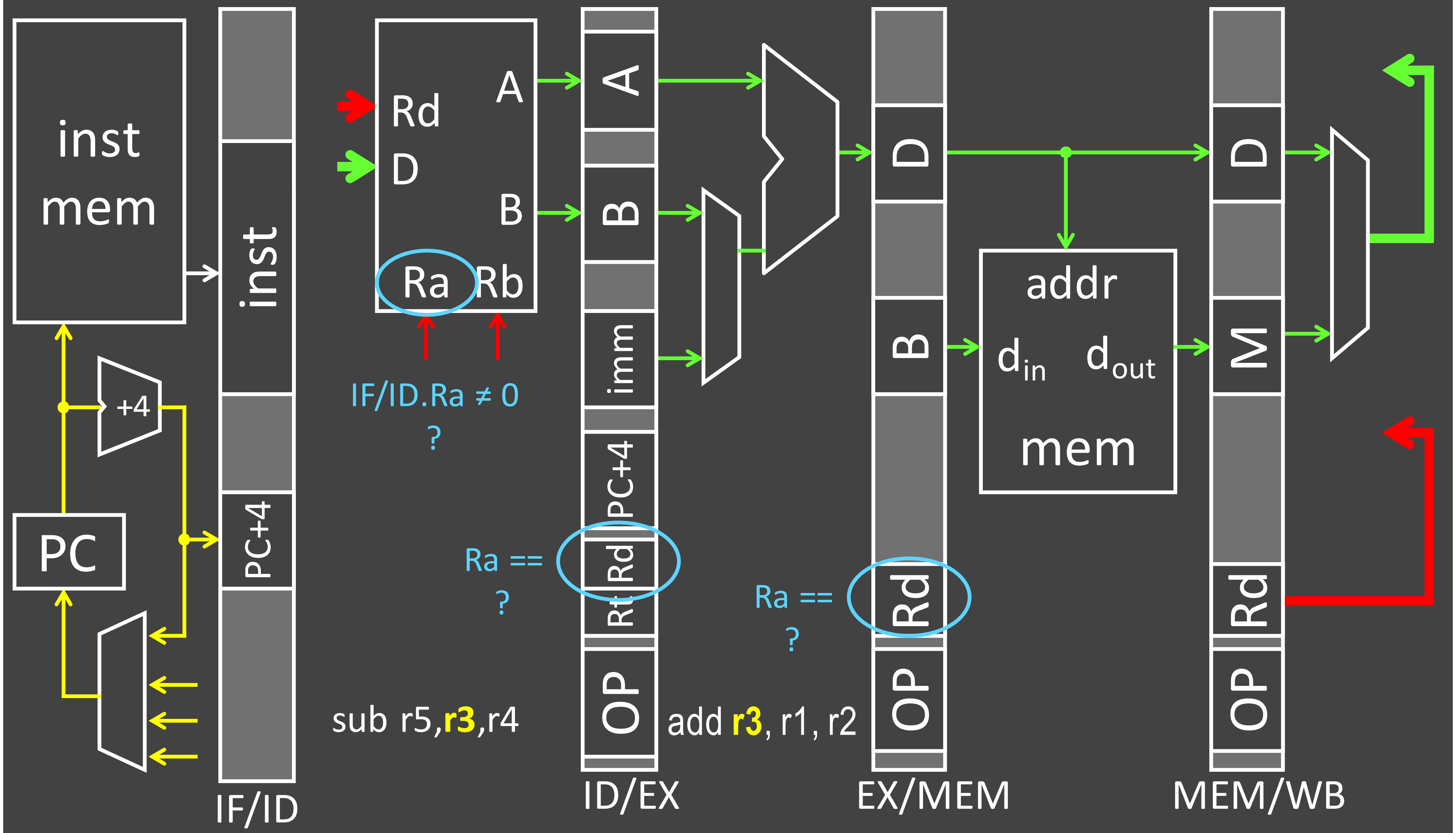
Visualizing Data Hazards (2)



Visualizing Data Hazards (3)



Detecting Data Hazards



Stall = (IF/ID.Ra != 0 && (IF/ID.Ra == ID/EX.Rd
|| IF/ID.Ra == EX/M.Rd))

repeat for Rb

Possible Responses to Data Hazards

1. Do Nothing
   - Change the ISA to match implementation
   - "Hey compiler: don't create code w/data hazards!"

     (*We can do better than this*)
2. Stall
   - Pause current and subsequent instructions till safe
3. Forward/bypass
   - Forward data value to where it is needed

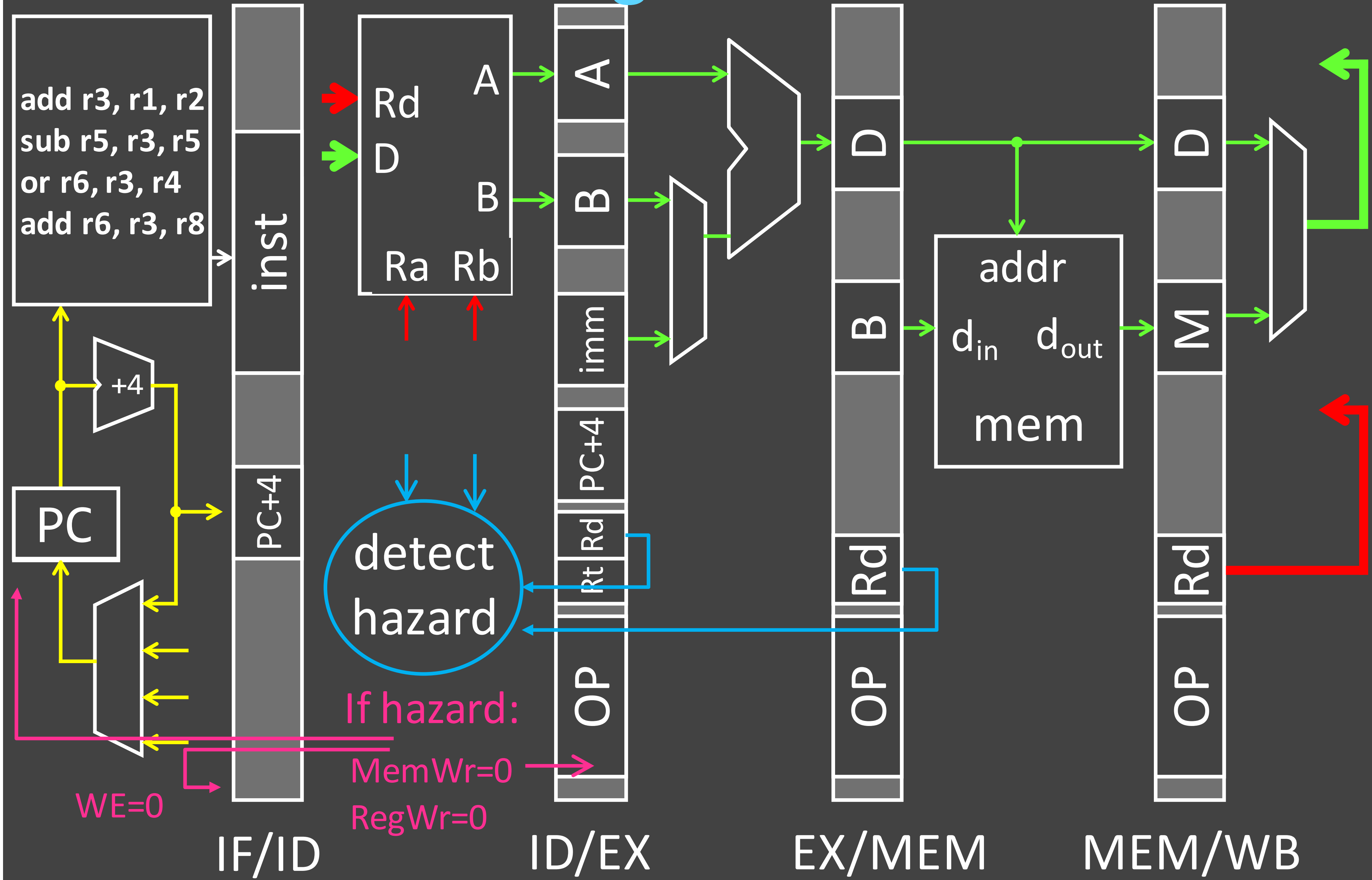
     (Only works if value actually exists already)

Stalling

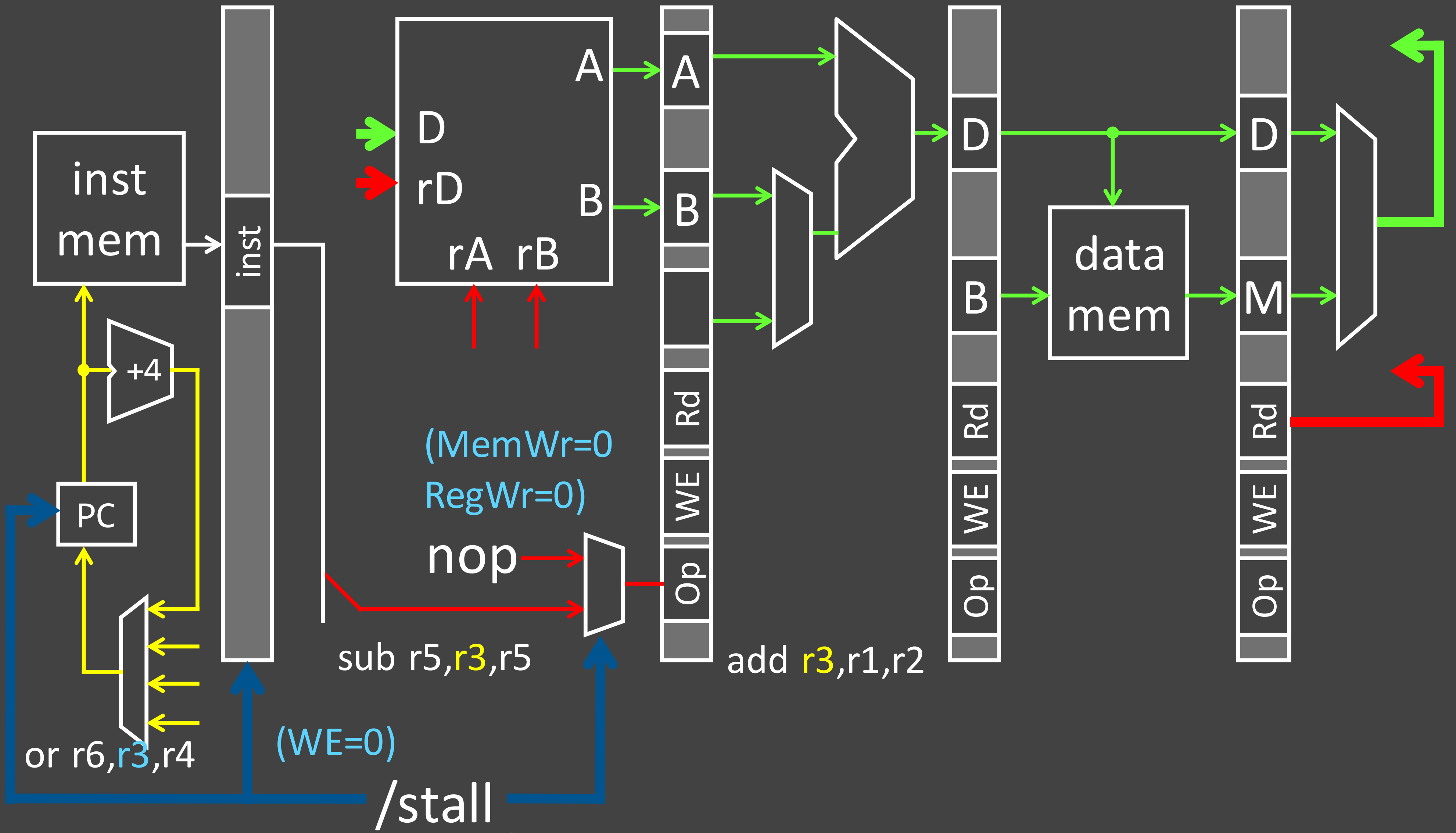
How to stall an instruction in ID stage

- prevent IF/ID pipeline register update
  - stalls the ID stage instruction
- convert ID stage insn into nop for later stages
  - innocuous “bubble” passes through pipeline
- prevent PC update
  - stalls the next (IF stage) instruction

Control Signals for a Stall



Detecting the Hazard

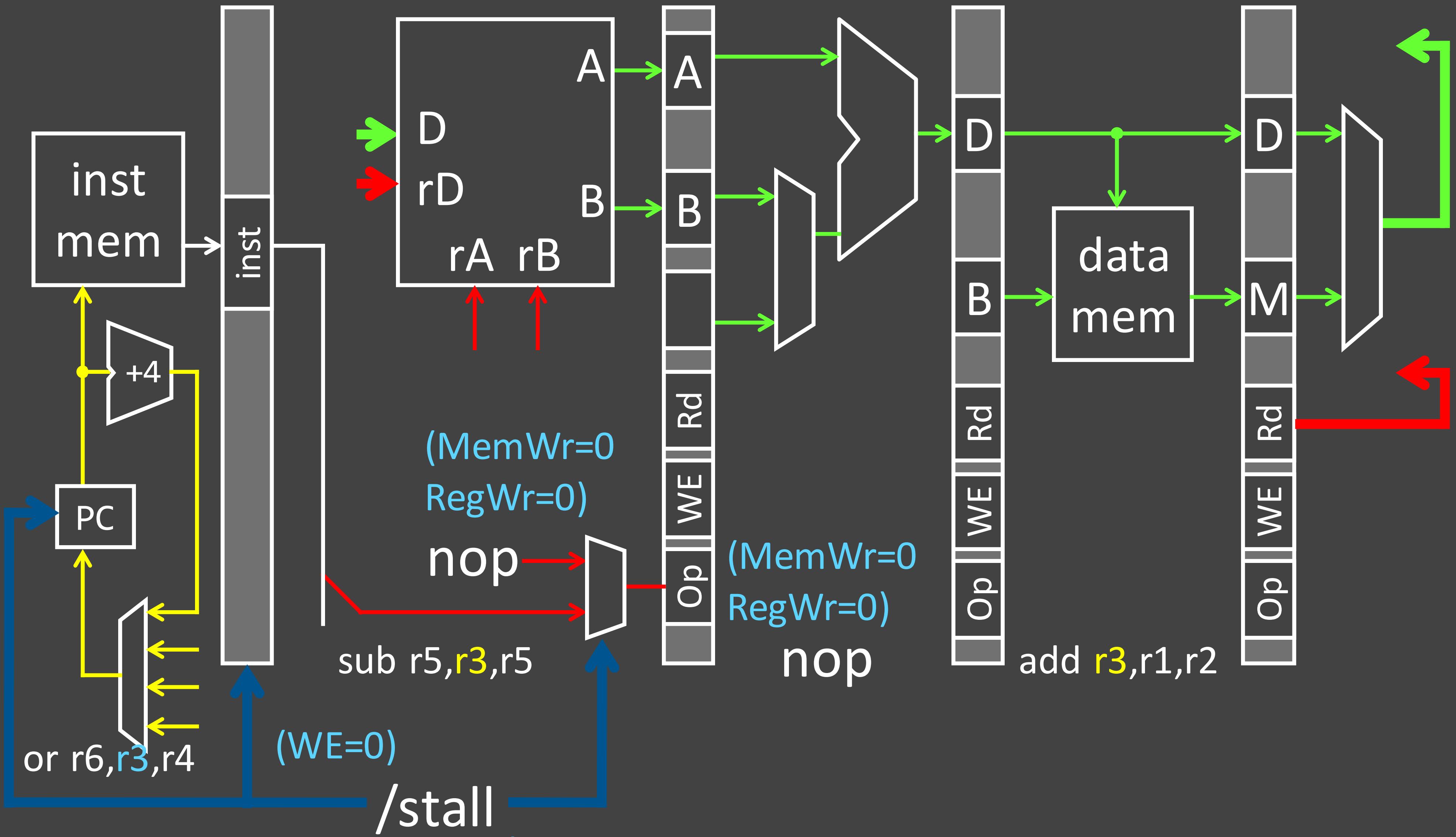


NOP = If(IF/ID.rA ≠ 0 &&

(IF/ID.rA==ID/Ex.Rd ← STALL CONDITION MET

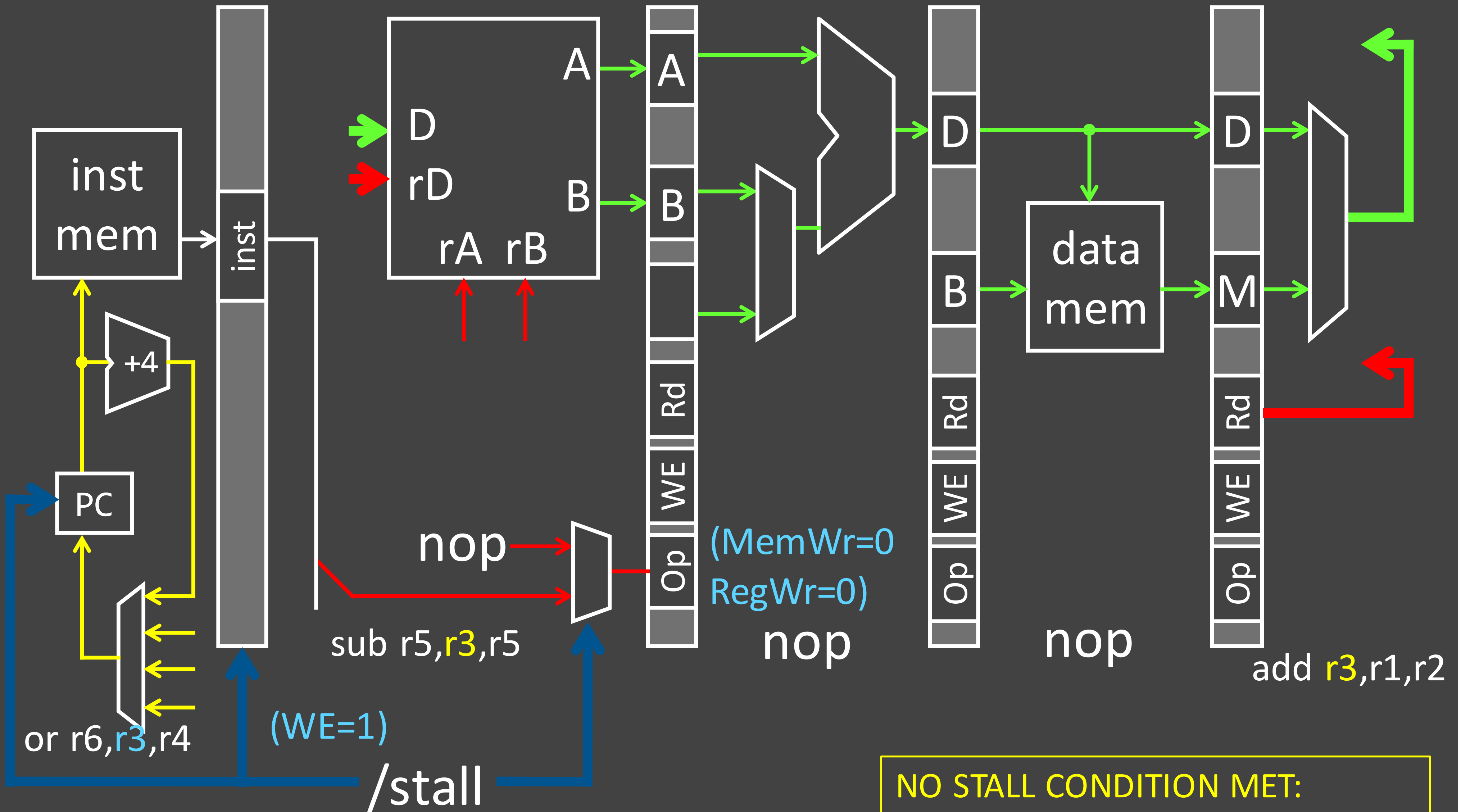
IF/ID.rA==Ex/M.Rd))

First Stall Cycle (nop in X)



NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd)) ← STALL CONDITION MET

Second Stall Cycle (nop in X, MEM)



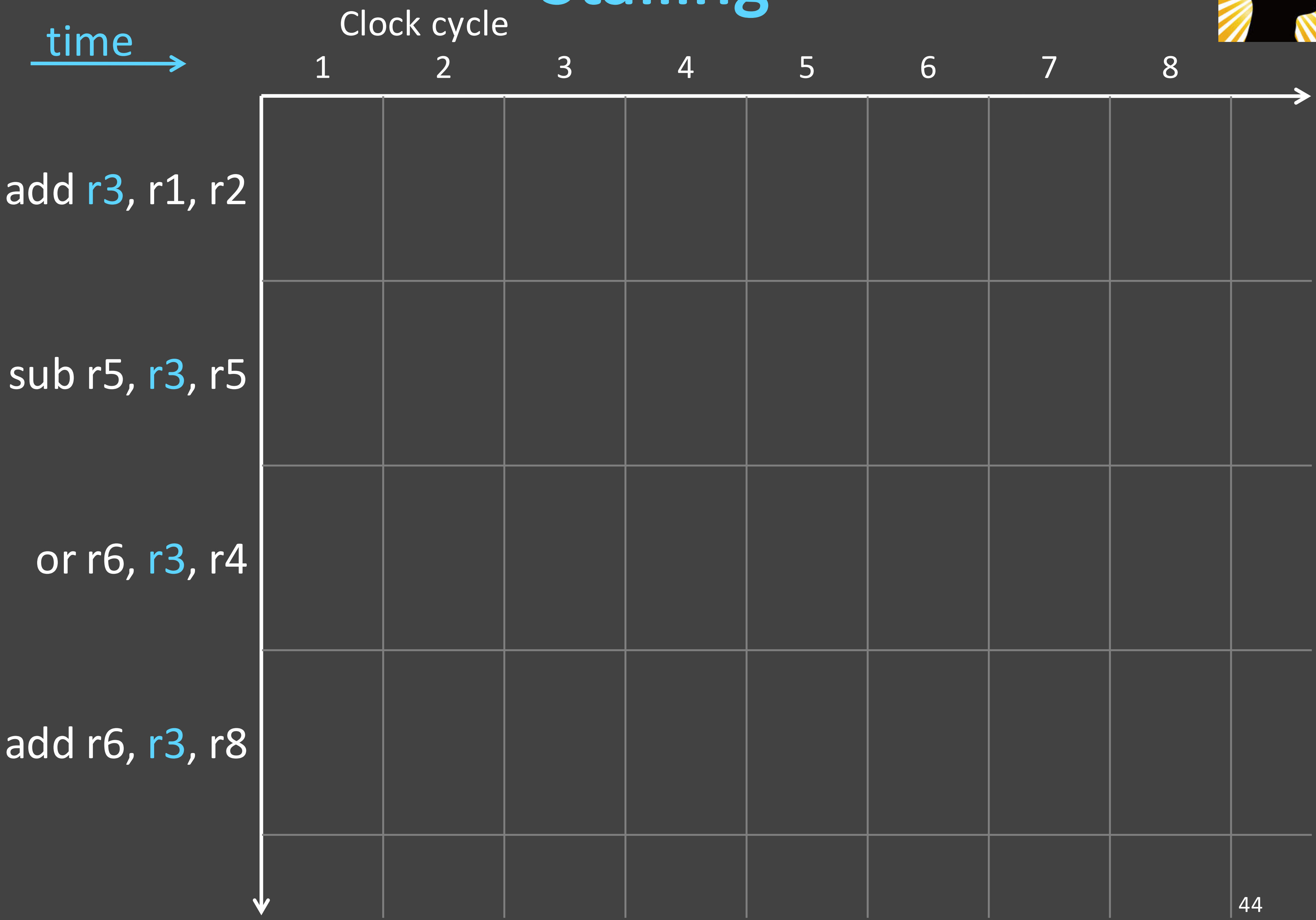
NOP = If(IF/ID.rA ≠ 0 &&

(IF/ID.rA==ID/Ex.Rd

IF/ID.rA==Ex/M.Rd))

NO STALL CONDITION MET:
sub allowed to leave decode stage

Stalling


time
Clock cycle
1
2
3
4
5
6
7
8
add r3, r1, r2
sub r5, r3, r5
or r6, r3, r4
add r6, r3, r8

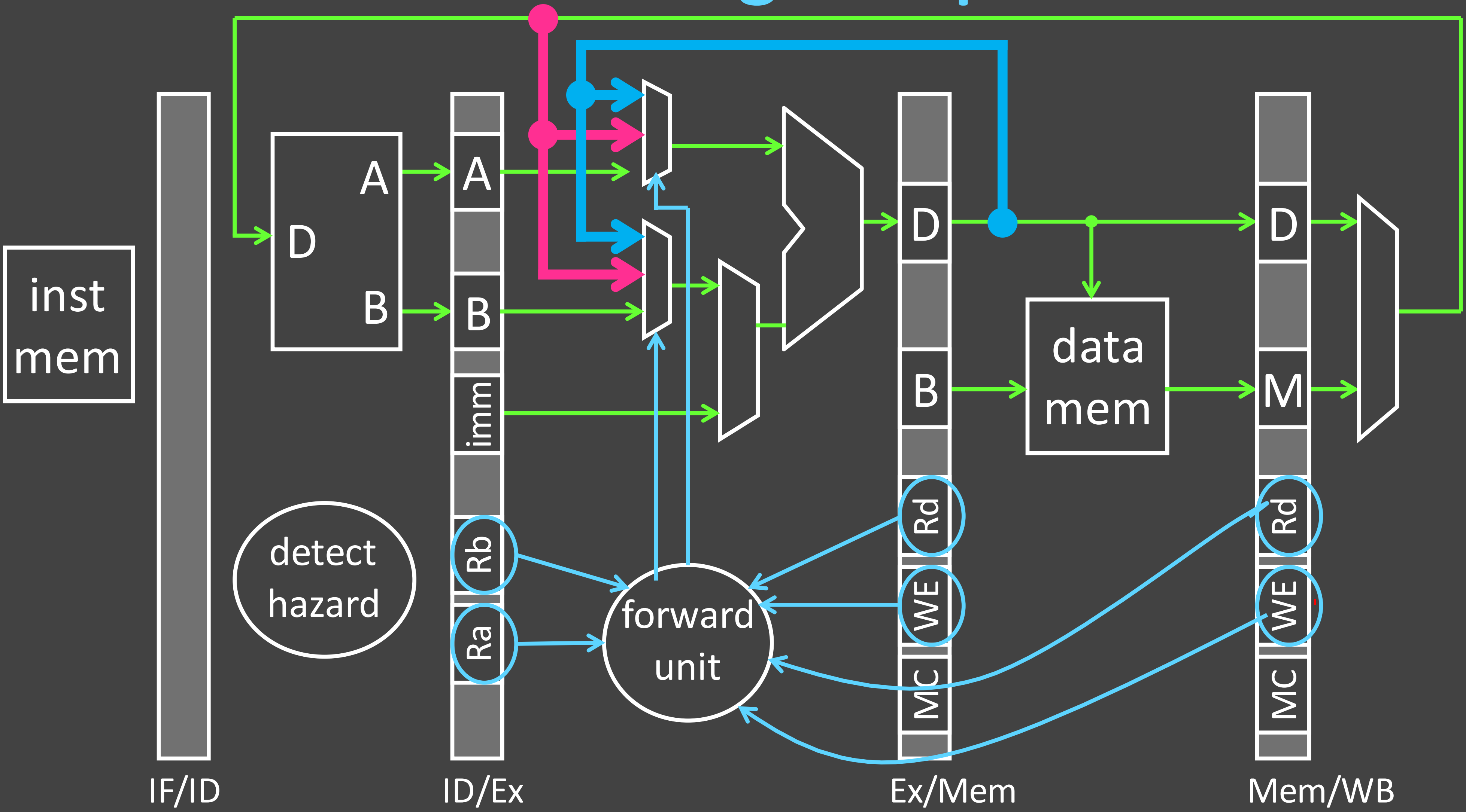
Possible Responses to Data Hazards

1. Do Nothing
   - Change the ISA to match implementation
   - “Compiler: don’t create code with data hazards!”

     (Nice try, we can do better than this)
2. Stall
   - Pause current and subsequent instructions till safe
3. Forward/bypass
   - Forward data value to where it is needed

     (Only works if value actually exists already)

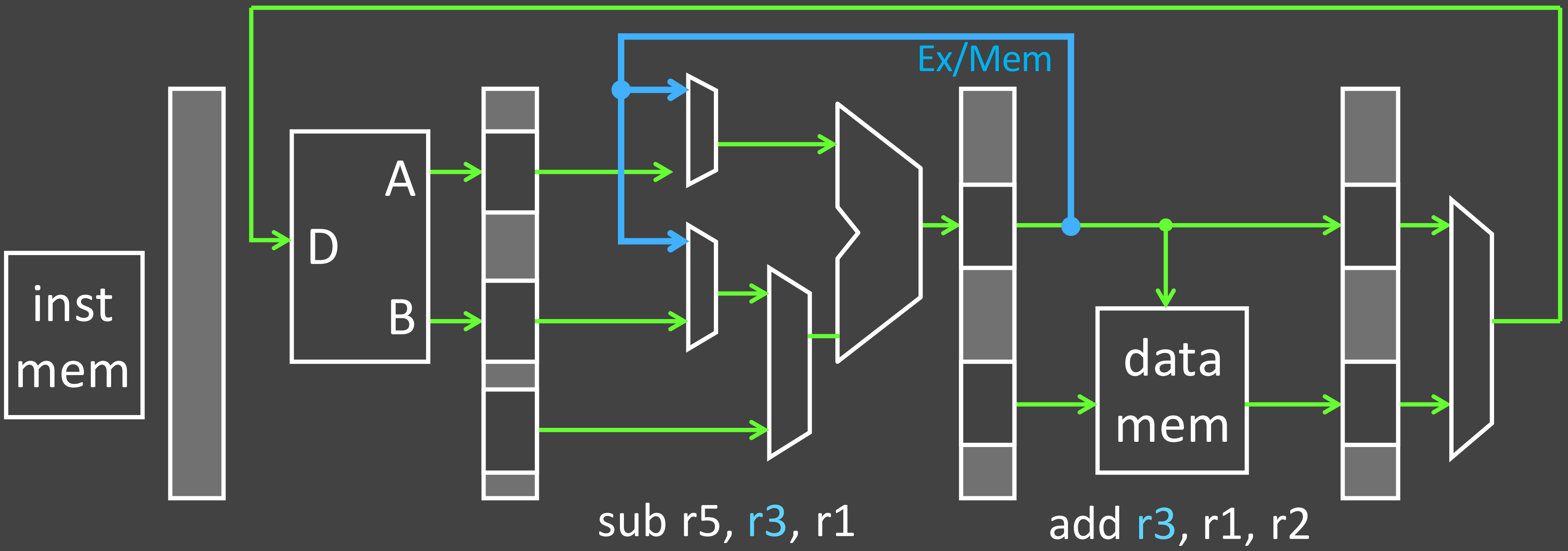
Forwarding Datapath

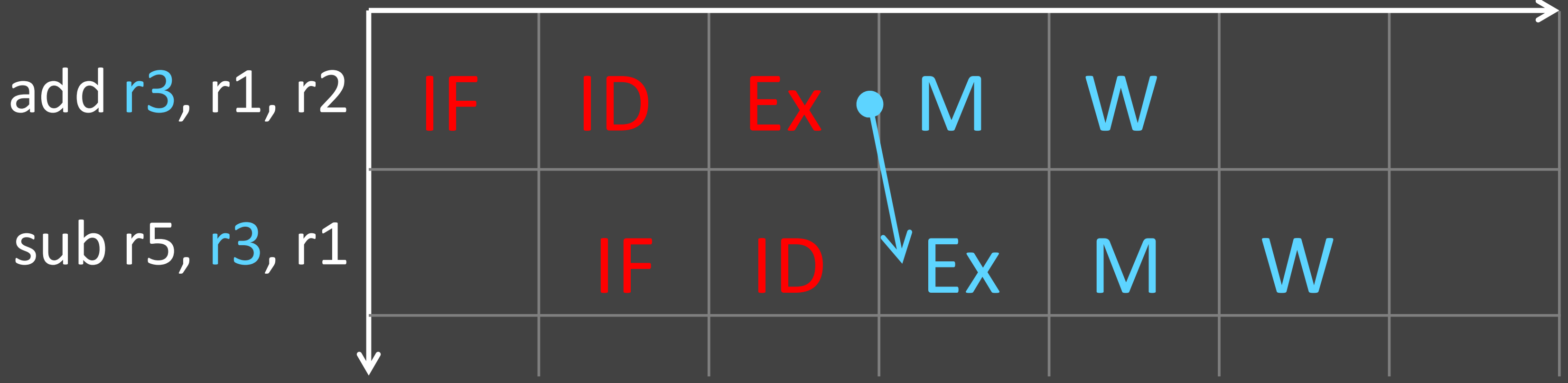


Two types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M→Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)

Forwarding Datapath 1: Ex/MEM → EX



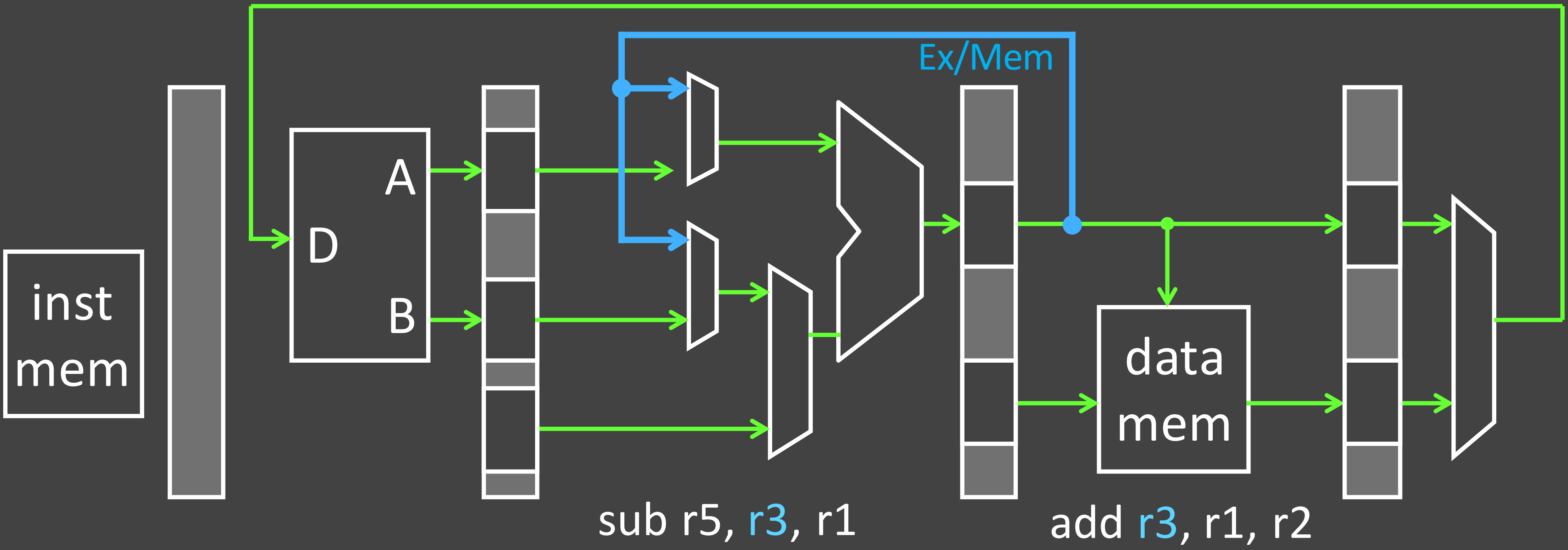


add r3, r1, r2	IF	ID	Ex	M	W		
sub r5, r3, r1		IF	ID	Ex	M	W	

Problem: EX needs ALU result that is in MEM stage

Solution: add a bypass from EX/MEM.D to start of EX

Forwarding Datapath 1: Ex/MEM → EX



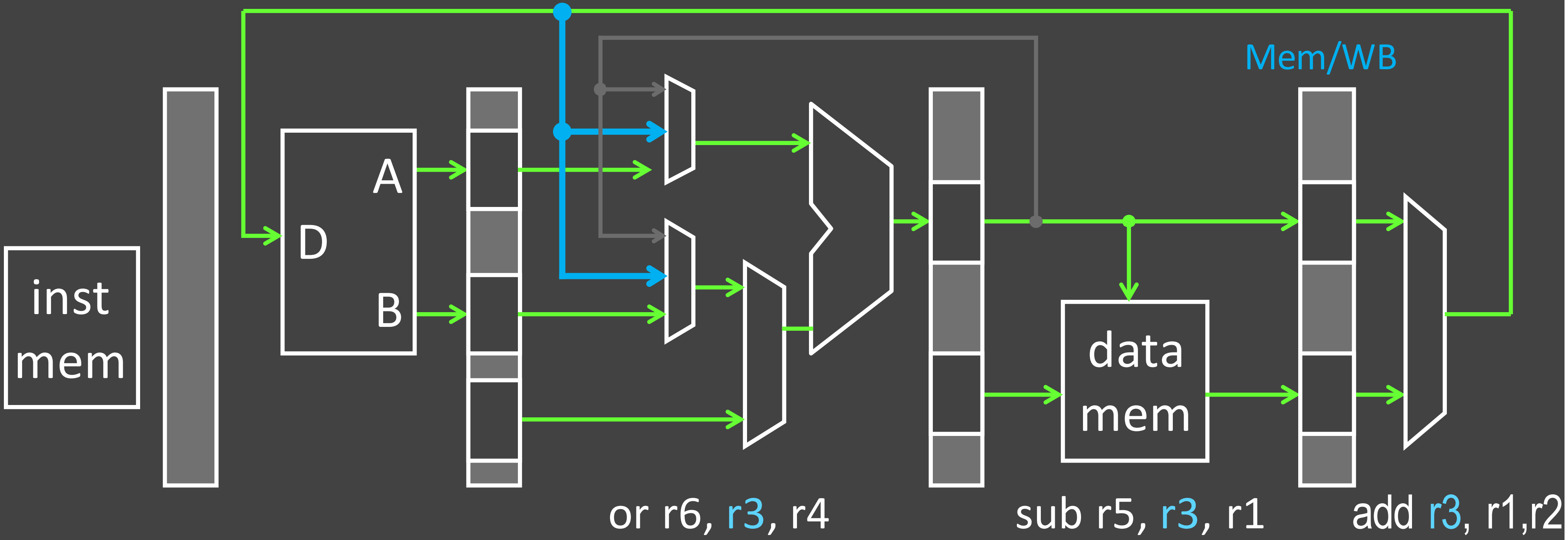
Detection Logic in Ex Stage:

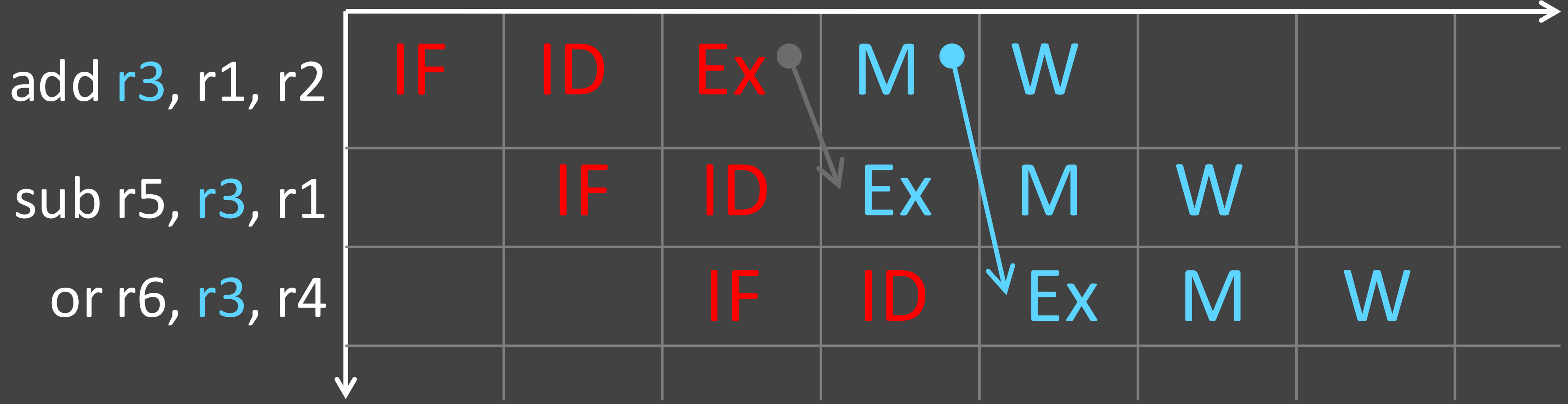
forward = (Ex/M.WE && EX/M.Rd != 0 &&

ID/Ex.Ra == Ex/M.Rd)

|| (same for Rb)

Forwarding Datapath 2: Mem/WB → EX



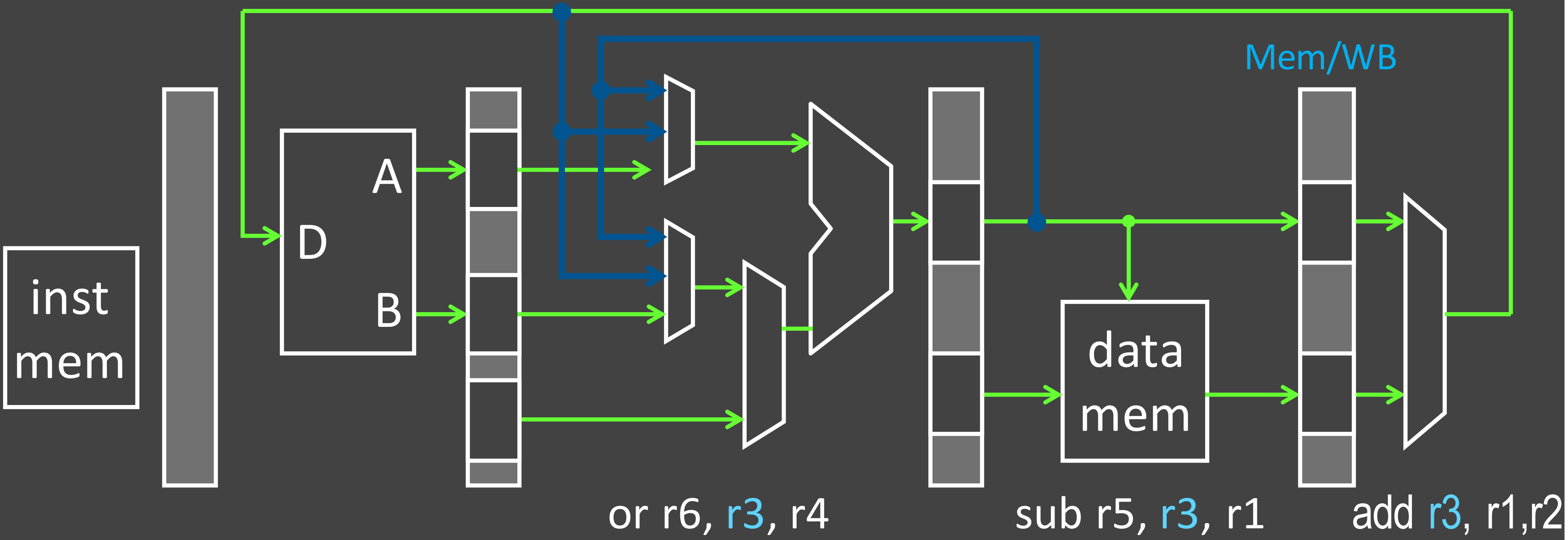


add r3, r1, r2	IF	ID	Ex	M	W			
sub r5, r3, r1		IF	ID	Ex	M	W		
or r6, r3, r4			IF	ID	Ex	M	W	

Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

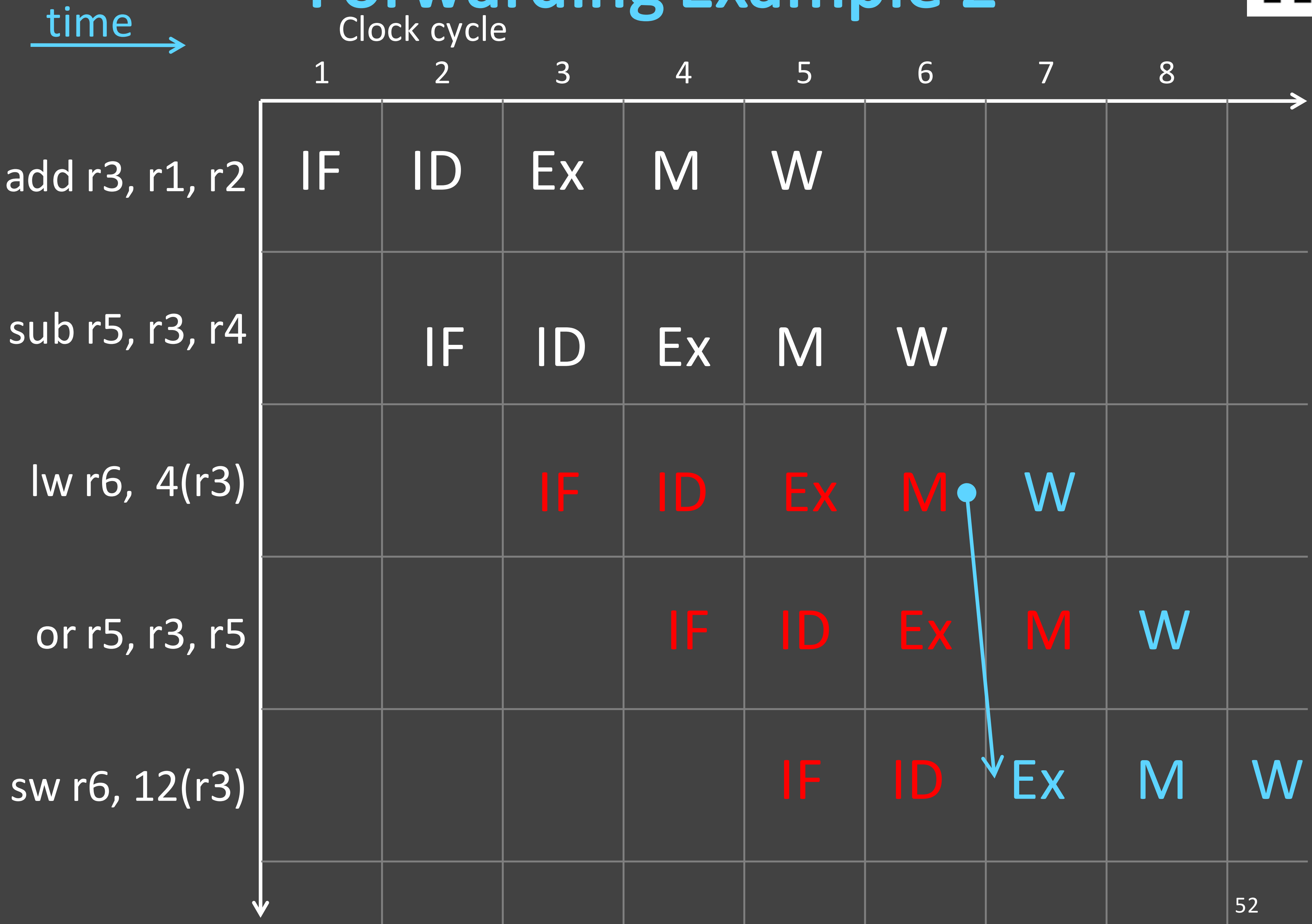
Forwarding Datapath 2: Mem/WB → EX



Detection Logic:

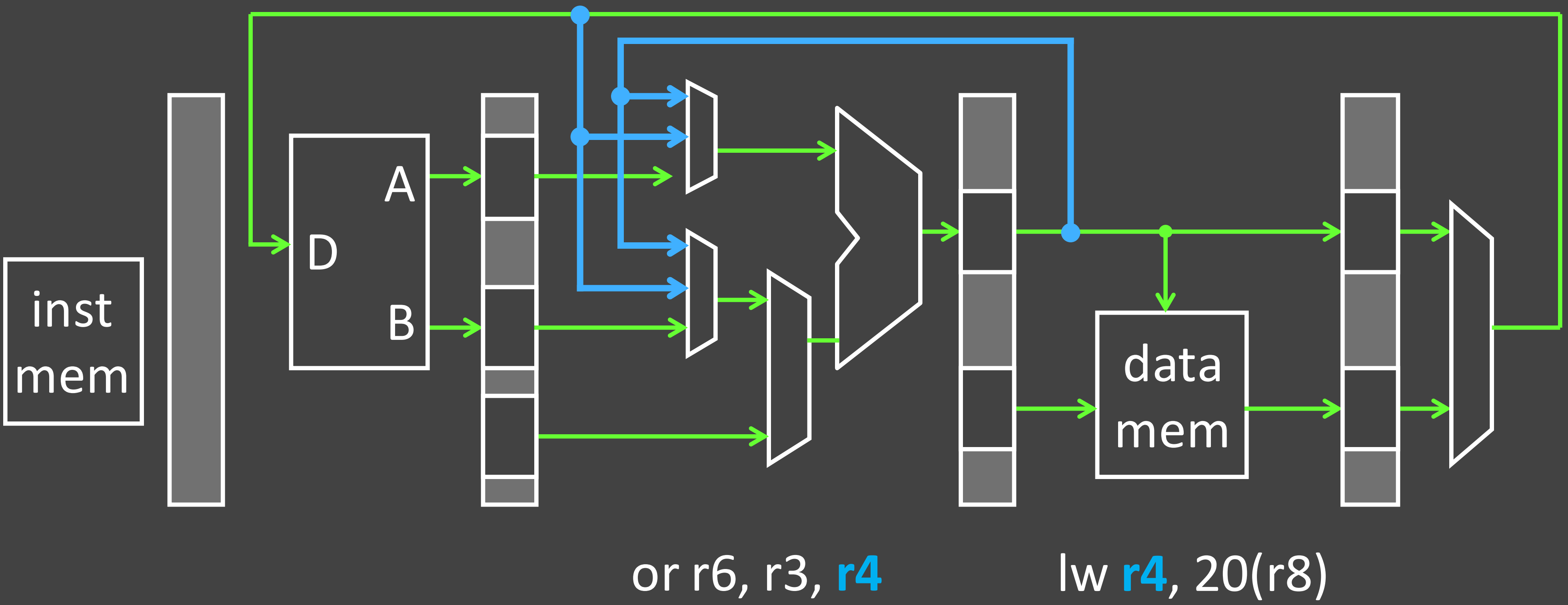
forward = (M/WB.WE && M/WB.Rd != 0 &&
ID/Ex.Ra == M/WB.Rd &&
not (ID/Ex.WE && Ex/M.Rd != 0 &&
ID/Ex.Ra == Ex/M.Rd)
|| (same for Rb)

Forwarding Example 2



	1	2	3	4	5	6	7	8	
add r3, r1, r2	IF	ID	Ex	M	W				
sub r5, r3, r4		IF	ID	Ex	M	W			
lw r6, 4(r3)			IF	ID	Ex	M	W		
or r5, r3, r5				IF	ID	Ex	M	W	
sw r6, 12(r3)					IF	ID	Ex	M	W

Load-Use Hazard Explained



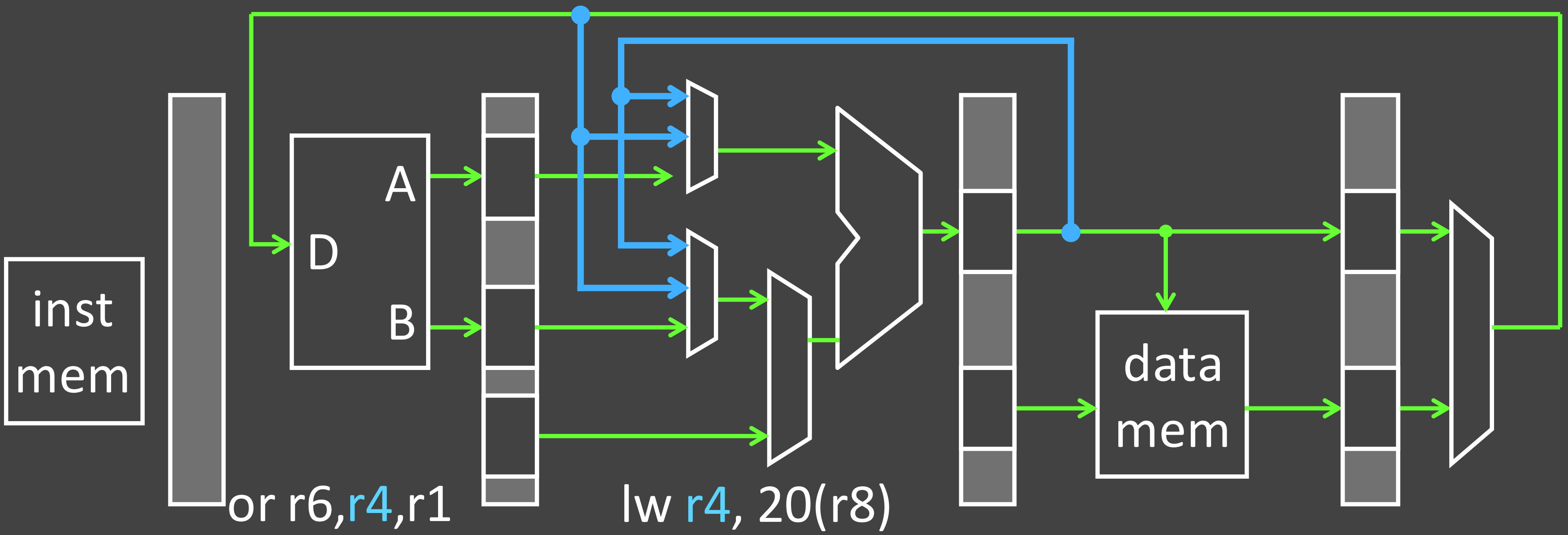
Data dependency after a load instruction:

- Value not available until *after* the M stage

→Next instruction cannot proceed if dependent

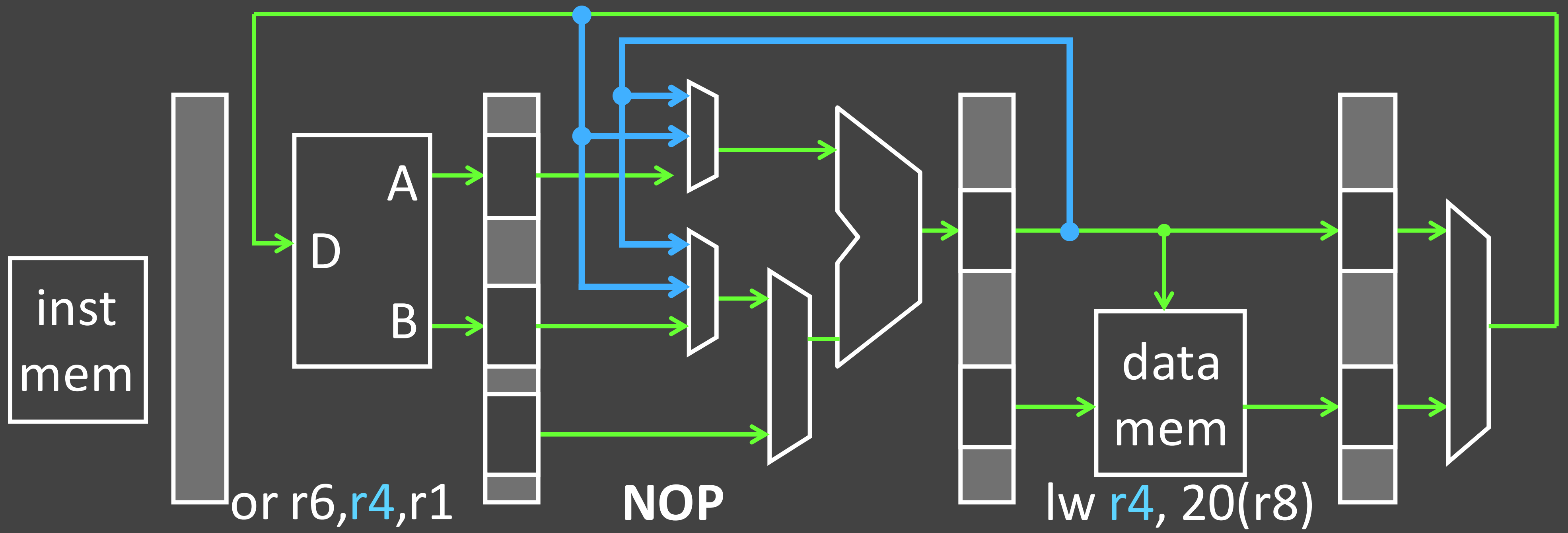
THE KILLER HAZARD

Load-Use Stall (1)



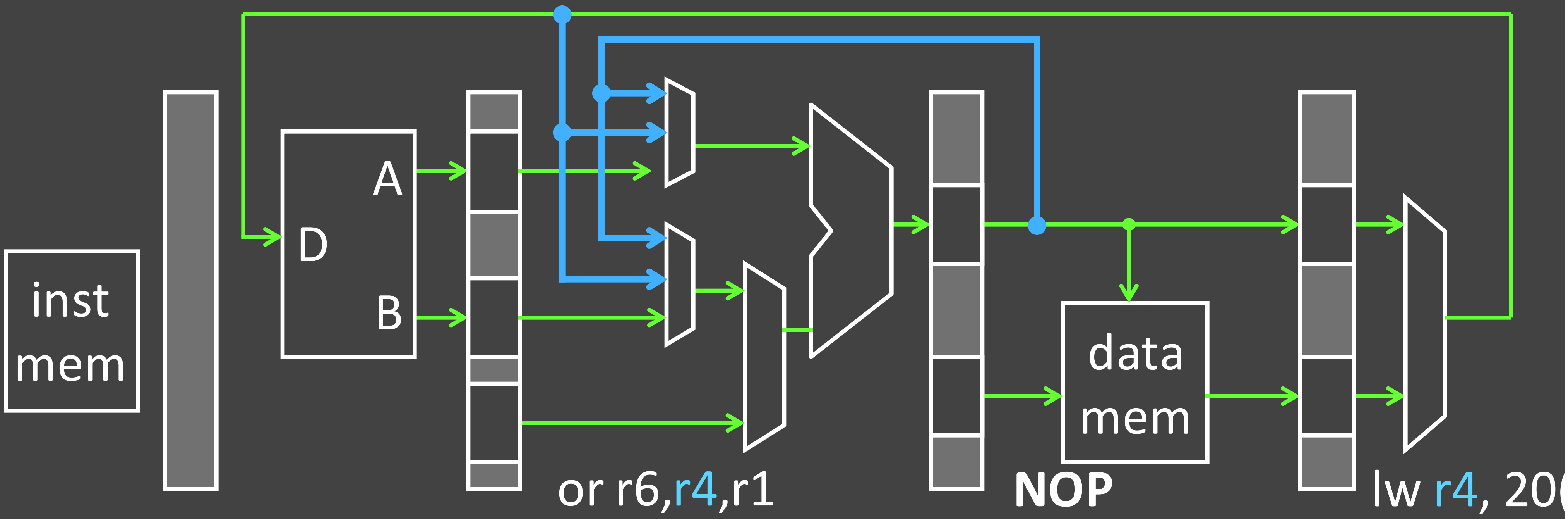
lw r4, 20(r8)	IF	ID	Ex						
or r6, r3, r4		IF	ID						

Load-Use Stall (2)



lw r4, 20(r8)	IF	ID	Ex	M	W		
or r6, r3, r4		IF	ID	Stall ID*	Ex	M	W

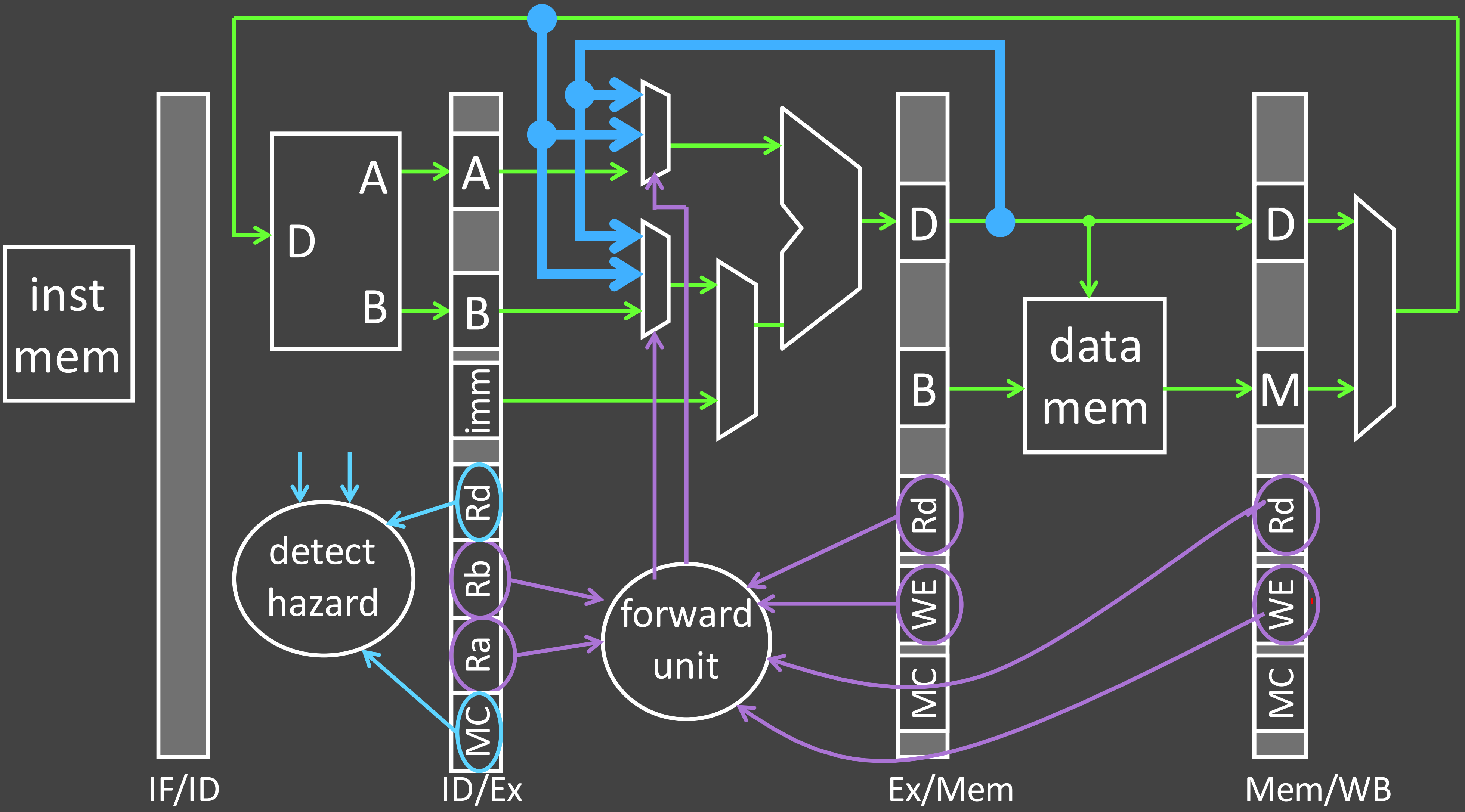
Load-Use Stall (3)



lw r4, 20(r8)	IF	ID	Ex	M	W			
or r6, r3, r4		IF	ID	ID*	Ex	M	W	

Stall

Load-Use Detection



Stall = If(ID/Ex.MemRead &&

IF/ID.Ra == ID/Ex.Rd

Resolving Load-Use Hazards

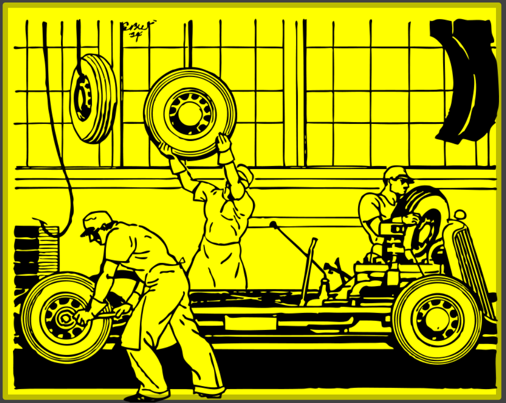
Two MIPS Solutions:

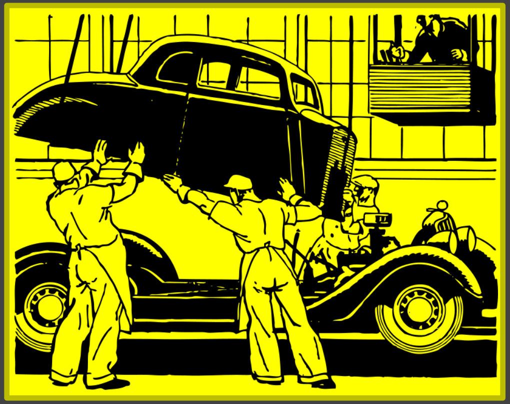
- MIPS 2000/3000: delay slot
  - ISA says results of loads are not available until one cycle later
  - Assembler inserts nop, or reorders to fill delay slot
- MIPS 4000 onwards: stall
  - But really, programmer/compiler reorders to avoid stalling in the load delay slot

Agenda

5-stage Pipeline

- Implementation
- Working Example





Hazards

- Structural
- Data Hazards
- Control Hazards

Control Hazards

Control Hazards

- instructions are fetched in stage 1 (IF)
- branch and jump decisions occur in stage 3 (EX)

→ next PC not known until ***2 cycles*** *after* branch/jump

```
0x10:       beq r1, r2, L
0x14:       add r3, r0, r3
0x18:       sub r5, r4, r6
0x1C: L:    or   r3, r2, r4
```

Branch ***not*** taken?

No Problem!

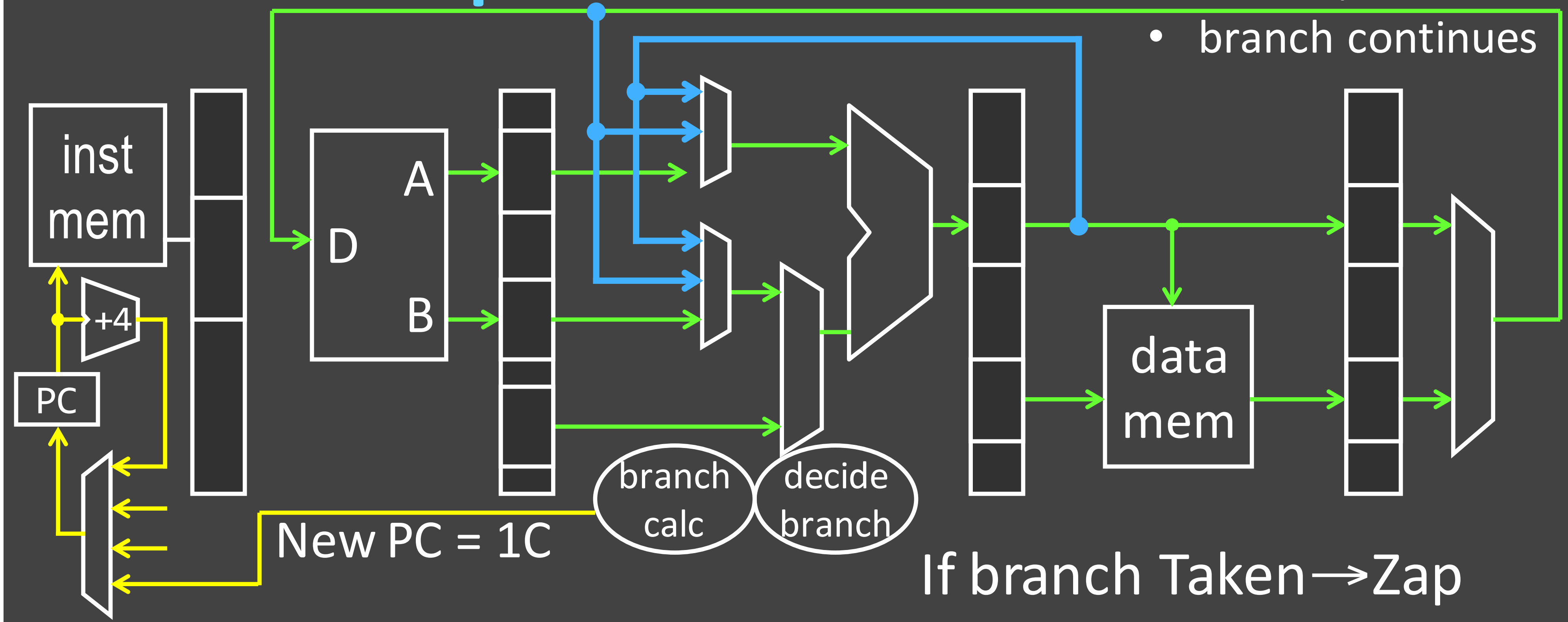
Branch taken?

Just fetched add, sub...

→ **Zap & Flush**

Zap & Flush

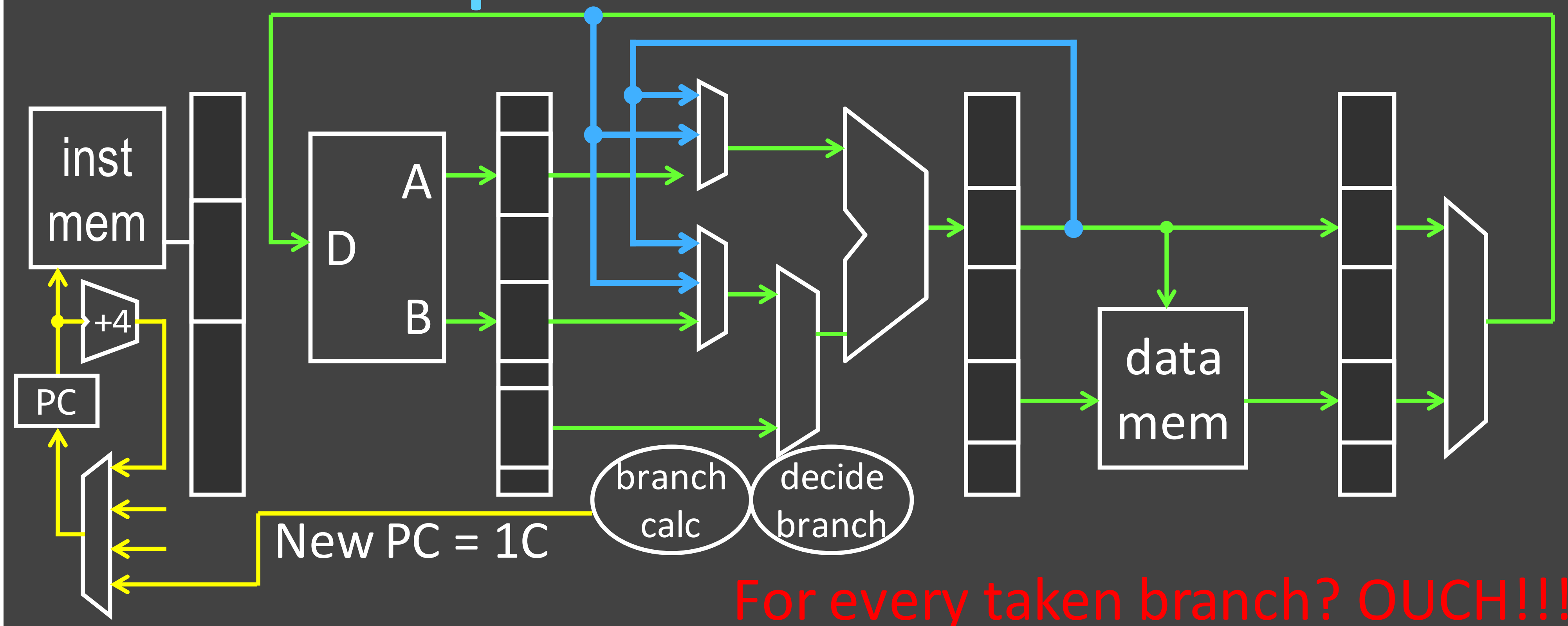
- prevent PC update
- clear IF/ID latch
- branch continues



If branch Taken→Zap

10: beq r1, r2, L	IF	ID	Ex	M	W				
14: add r3, r0, r3		IF	ID	NOP	NOP	NOP			
18: sub r5, r4, r6			IF	NOP	NOP	NOP	NOP		
1C: L: or r3, r2, r4				IF	ID	Ex	M	W	

Zap & Flush



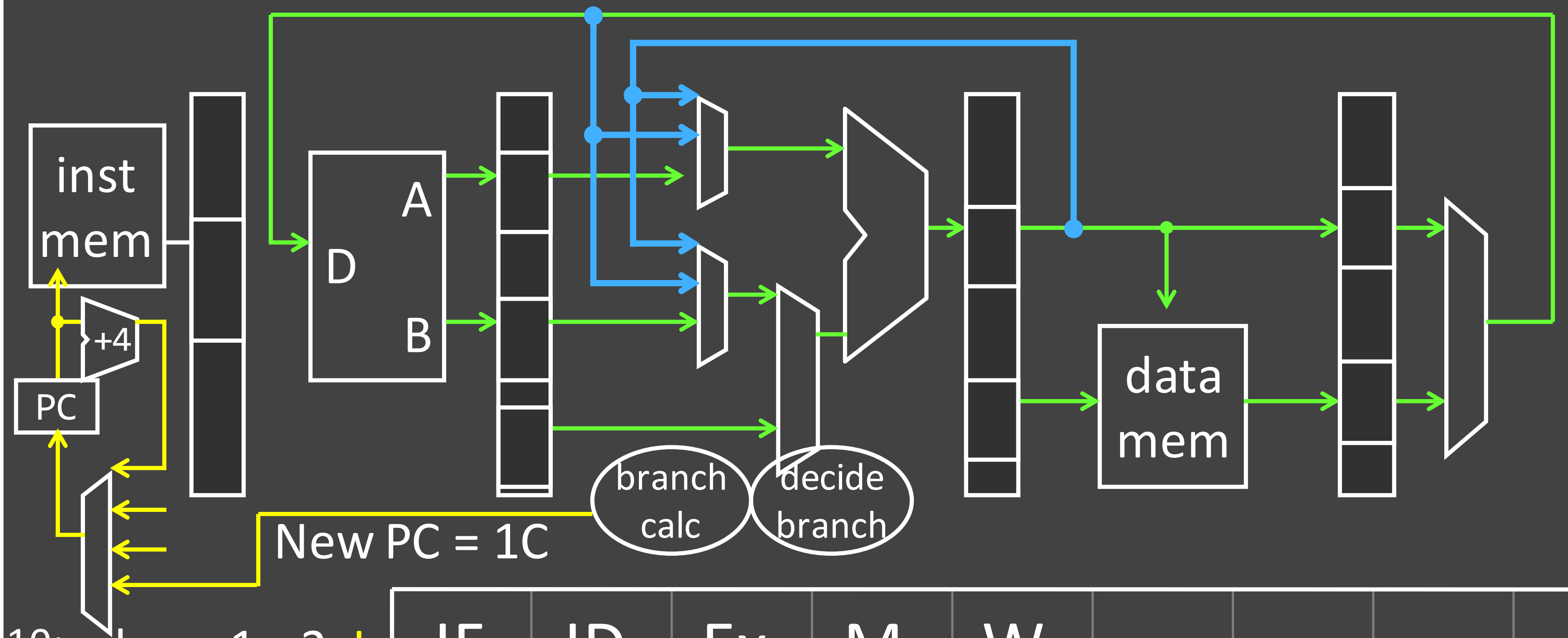
For every taken branch? OUCH!!!

10: beq r1, r2, L	IF	ID	Ex	M	W				
~~14: add r3, r0, r3~~		~~IF~~	~~ID~~	~~NOP~~	~~NOP~~	~~NOP~~			
~~18: sub r5, r4, r6~~			~~IF~~	~~NOP~~	~~NOP~~	~~NOP~~	~~NOP~~		
1C: L: or r3, r2, r4				IF	ID	Ex	M	W	

Reducing the cost of control hazard

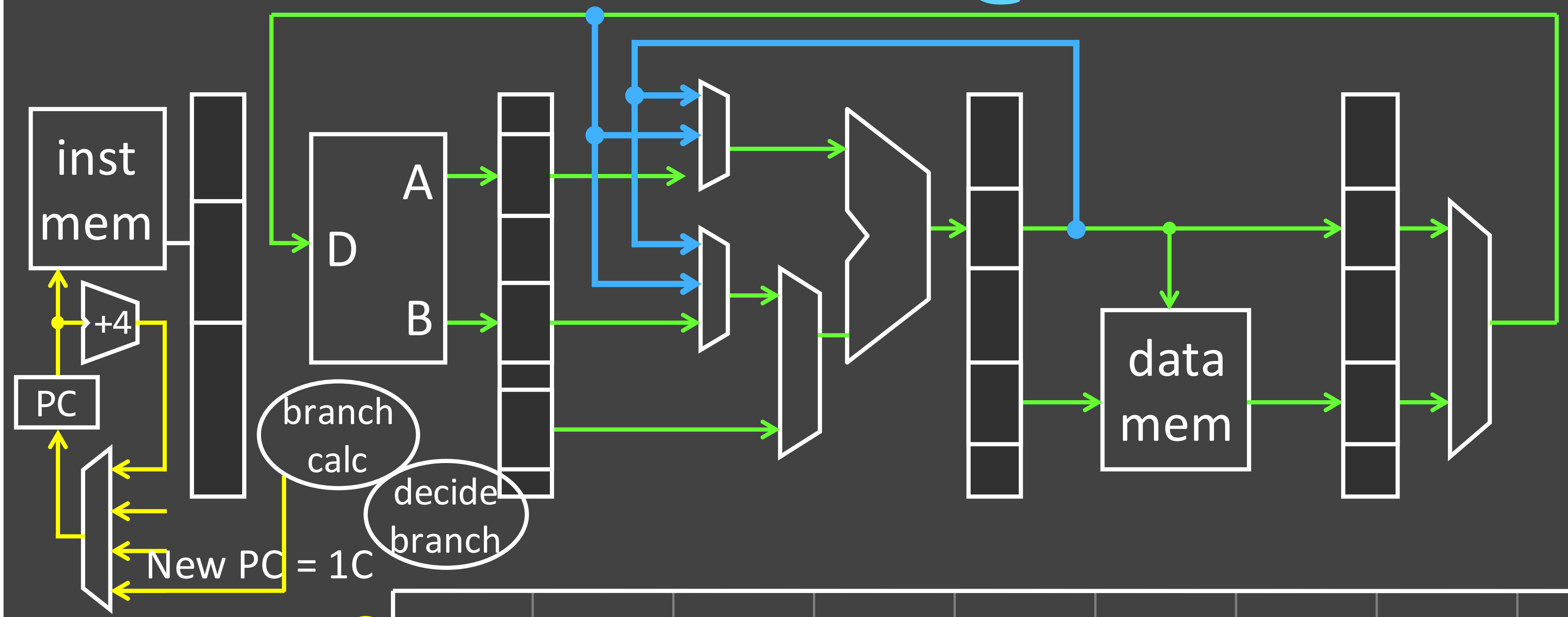
1. Delay Slot
    - You MUST do this
    - MIPS ISA: 1 insn after ctrl insn ***always*** executed
        - Whether branch taken or not
2. Resolve Branch at Decode
    - Some groups do this for Project 2, your choice
    - Move branch calc from EX to ID
    - Alternative: just zap 2nd instruction when branch taken
3. Branch Prediction
    - Not in 3410, but every processor worth *anything* does this

Delay Slot



10: beq r1, r2, L	IF	ID	Ex	M	W				
14: add r3, r0, r3		IF	ID	Ex	M	W	ALWAYS SAFE		
18: ~~sub r5, r4, r6~~			~~IF~~	~~NOP~~	~~NOP~~	~~NOP~~	~~NOP~~		
1C: L: or r3, r2, r4				IF	ID	Ex	M	W	

Resolve Branches @ Decode



10: beq r1, r2, L	IF	ID	Ex	M	W			
14: add r3, r0, r3		IF	ID	Ex	M	W	ALWAYS SAFE	
18: sub r5, r4, r6								
1C: L: or r3, r2, r4			IF	ID	Ex	M	W	

Branch Prediction

Most processor support **Speculative Execution**

- *Guess* direction of the branch
  - Allow instructions to move through pipeline
  - Zap them later if guess turns out to be wrong
- A *must* for long pipelines

Data Hazard Takeaways

Data hazards occur when a operand (register) depends on the result of a previous instruction that may not be computed yet. Pipelined processors need to detect data hazards.

Stalling, preventing a dependent instruction from advancing, is one way to resolve data hazards. Stalling introduces NOPs (“bubbles”) into a pipeline. Introduce NOPs by (1) preventing the PC from updating, (2) preventing writes to IF/ID registers from changing, and (3) preventing writes to memory and register file. Nops significantly decrease performance.

Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register). Better performance than stalling.

Control Hazard Takeaways

Control hazards occur because the PC following a control instruction is not known until control instruction is executed. If branch is taken → need to zap instructions. 1 cycle performance penalty.

Delay Slots can potentially increase performance due to control hazards. The instruction in the delay slot will ***always*** be executed. Requires software (compiler) to make use of delay slot. Put nop in delay slot if not able to put useful instruction in delay slot.

We can reduce cost of a control hazard by moving branch decision and calculation from Ex stage to ID stage. With a delay slot, this removes the need to flush instructions on taken branches.