# Pipelining

**Anne Bracy**

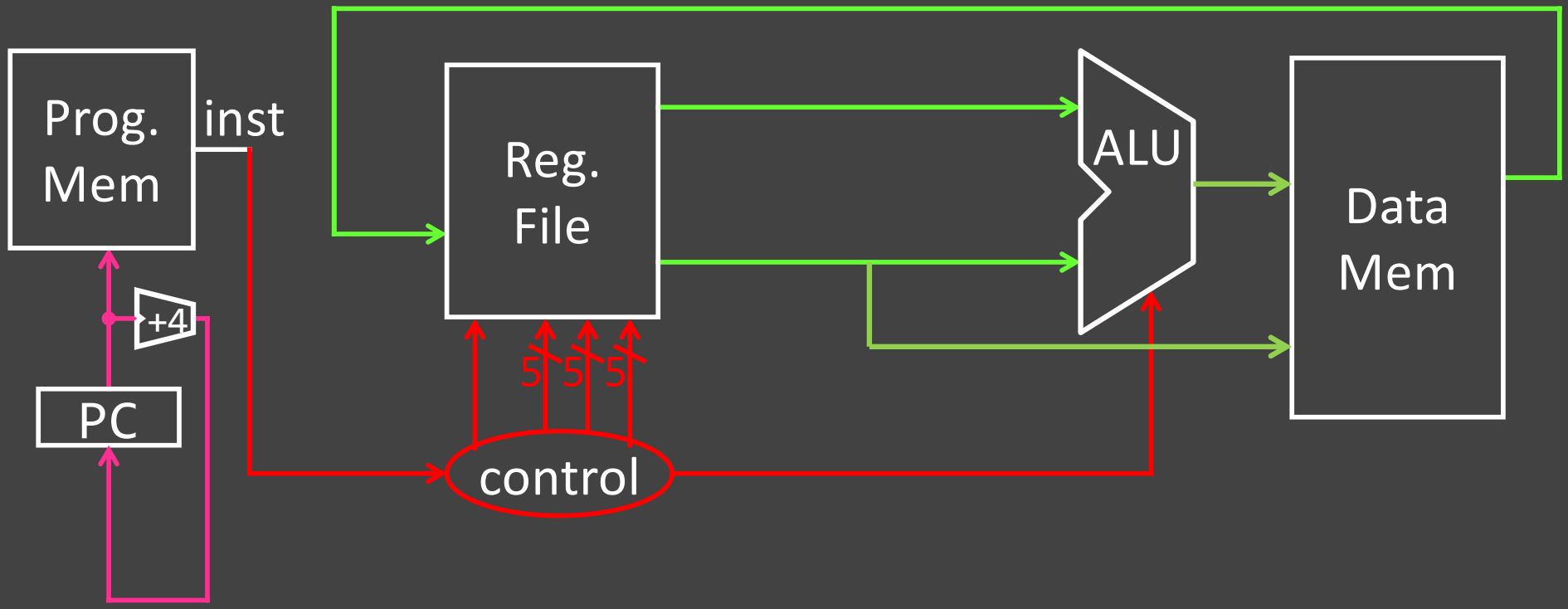
**CS 3410**

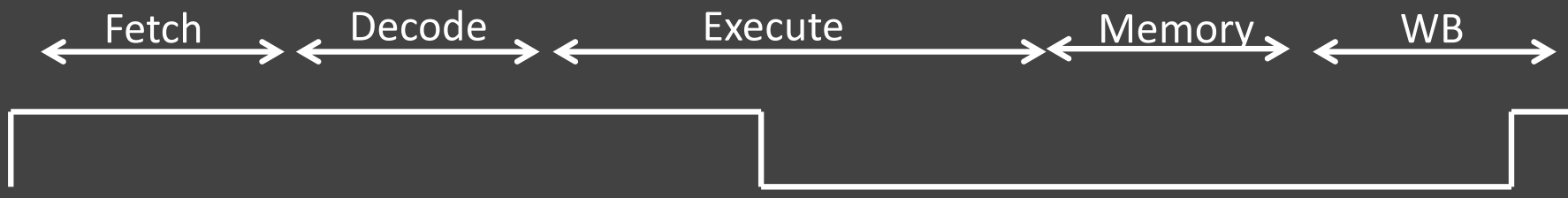
Computer Science

Cornell University

[K. Bala, A. Bracy, S. McKee, E. Sirer, H. Weatherspoon]

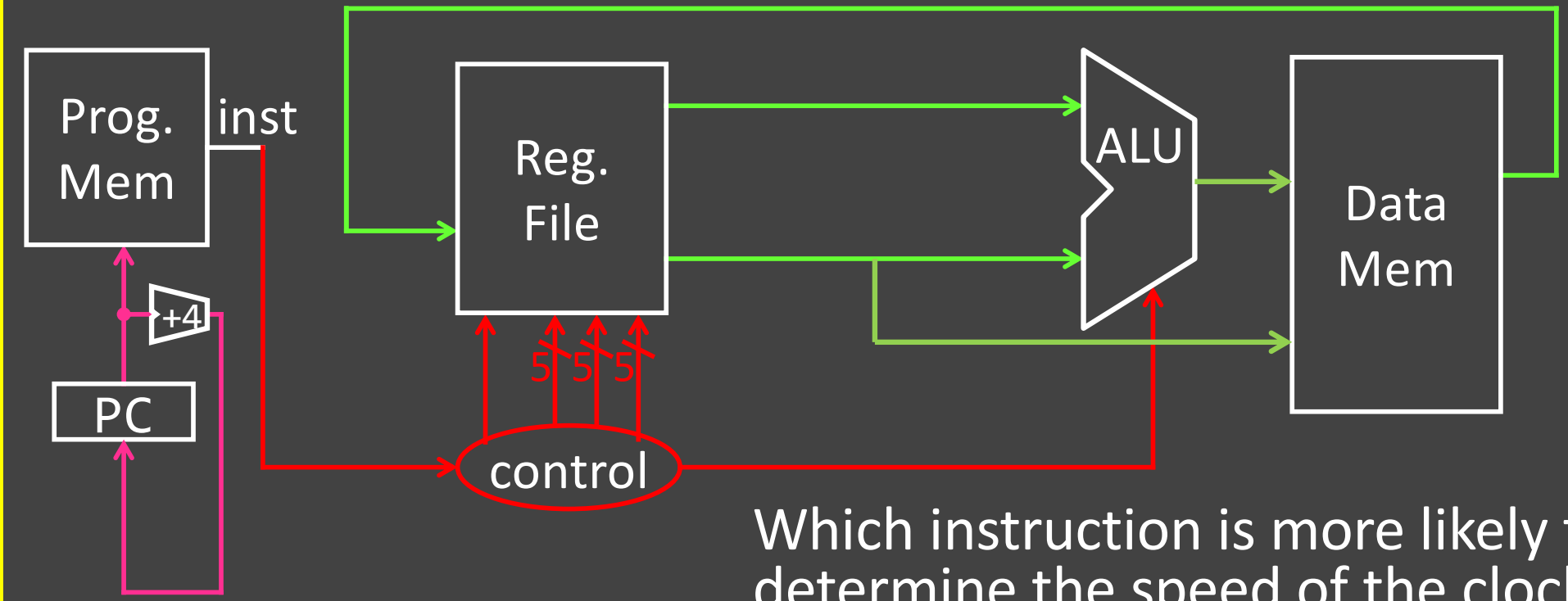
# Single-Cycle MIPS Datapath




A Single cycle processor – this diagram is not 100% spatial

# Clicker Question




Which instruction is more likely to determine the speed of the clock?

A. Jump Register
B. Add
C. Store
D. Load
E. Either Load or Store

# Five Stages of MIPS datapath

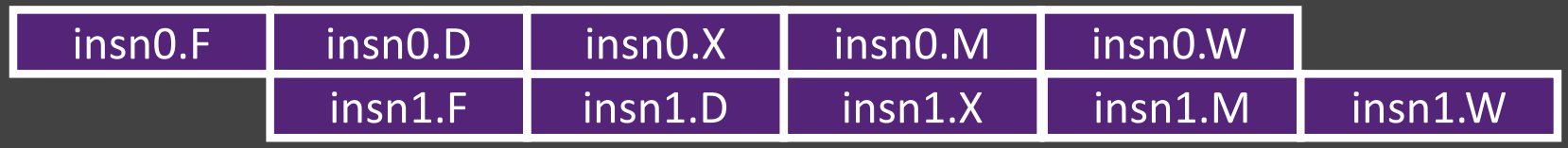
Basic CPU execution loop

1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback

# Single Cycle → Pipelining

**Single-cycle**

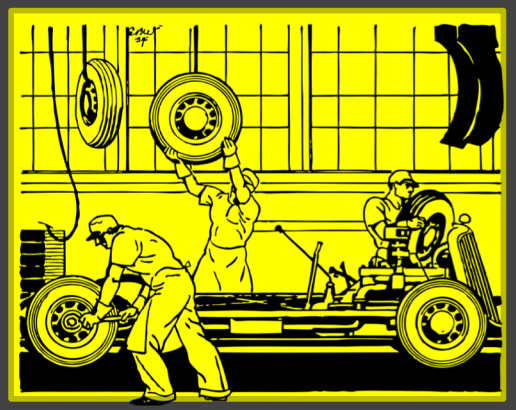

**Pipelined**

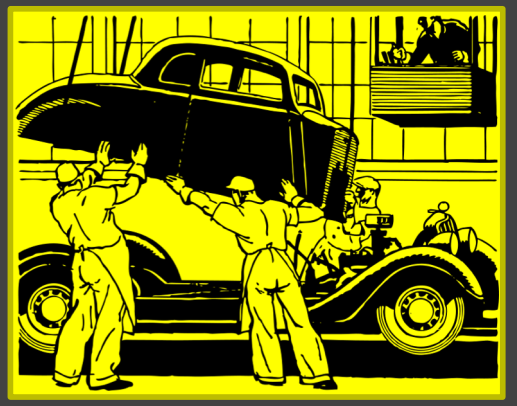


# Agenda

## 5-stage Pipeline

- Implementation
- Working Example

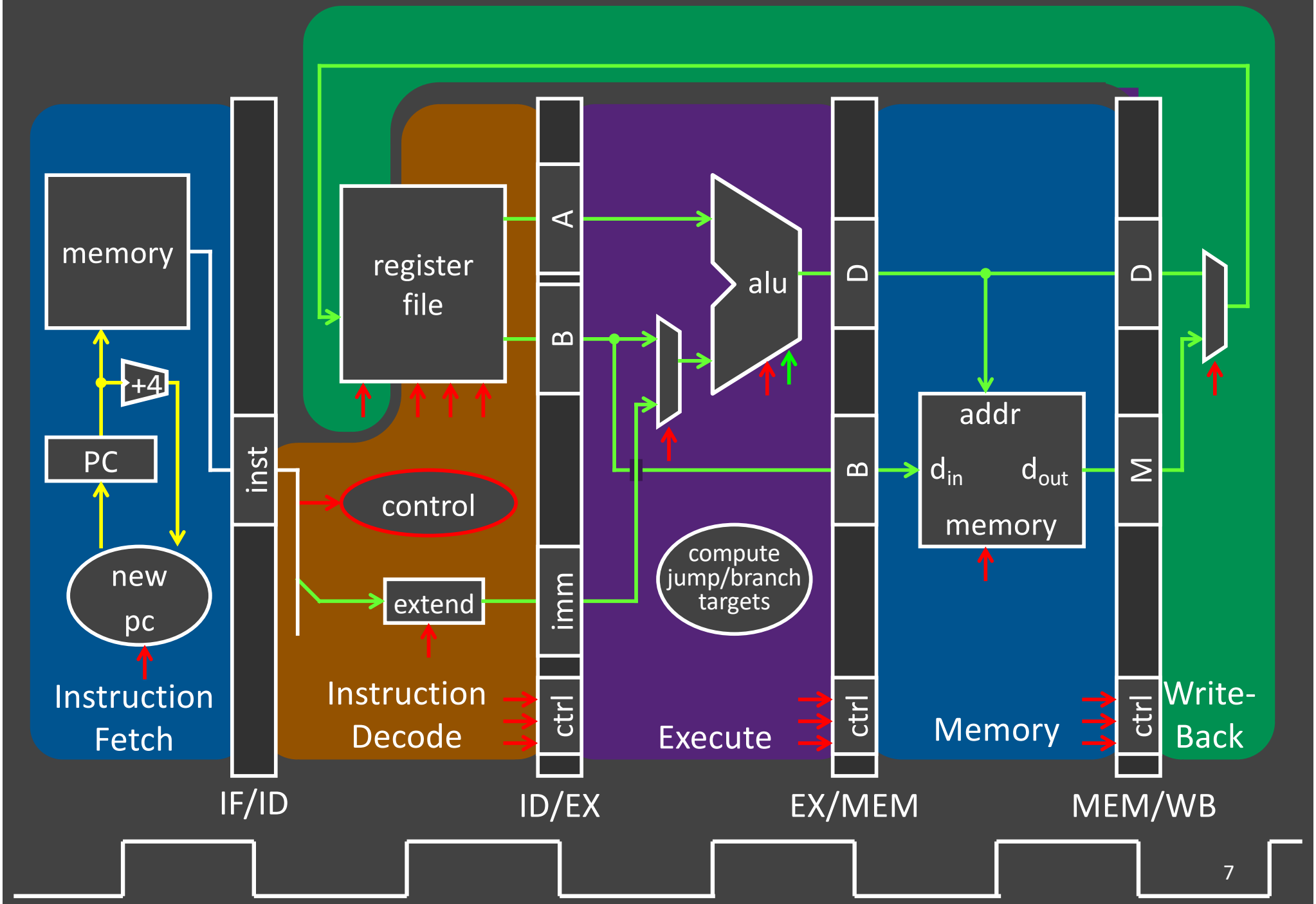




## Hazards

- Structural
- Data Hazards
- Control Hazards

# Pipelined Processor



# Time Graphs


Latency: 5 cycles

Throughput: 1 insn/cycle

CPI = 1

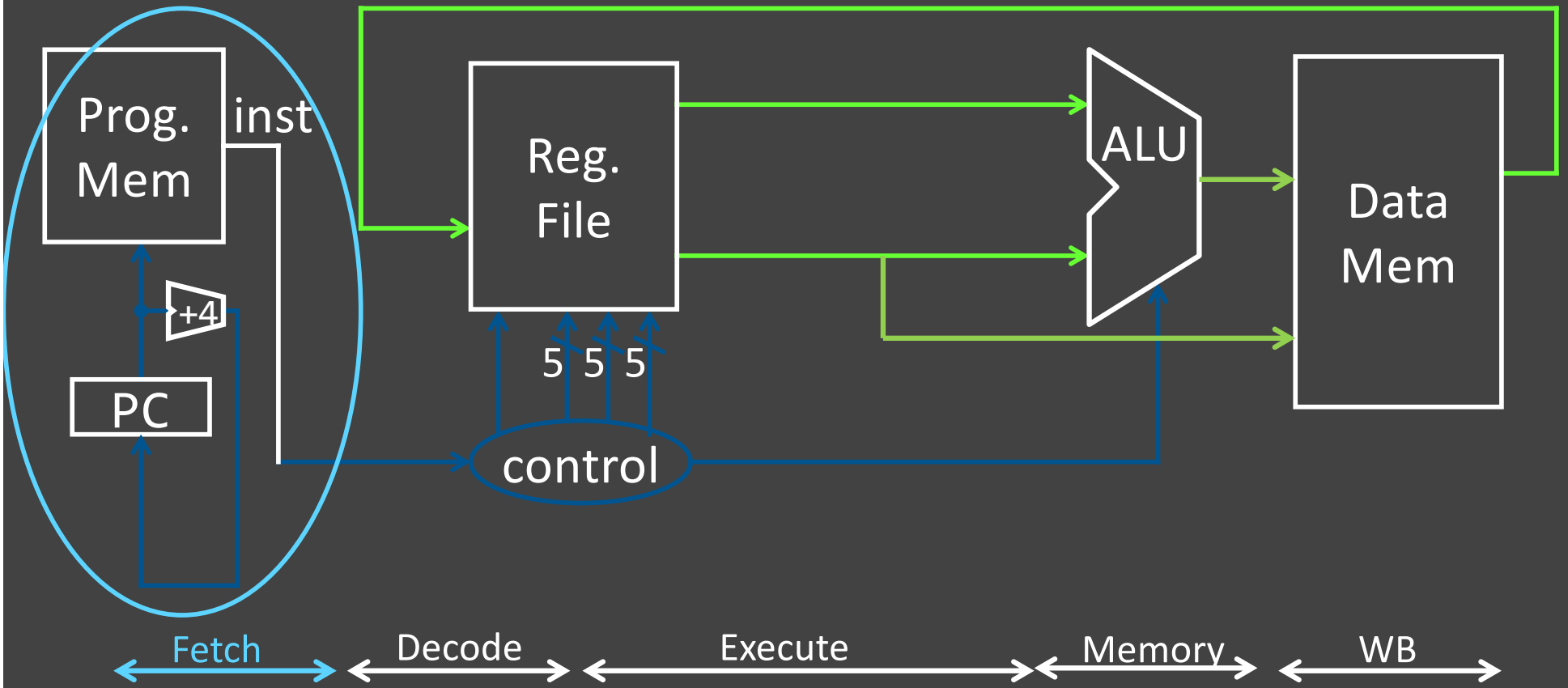
# Principles of Pipelined Implementation

- 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

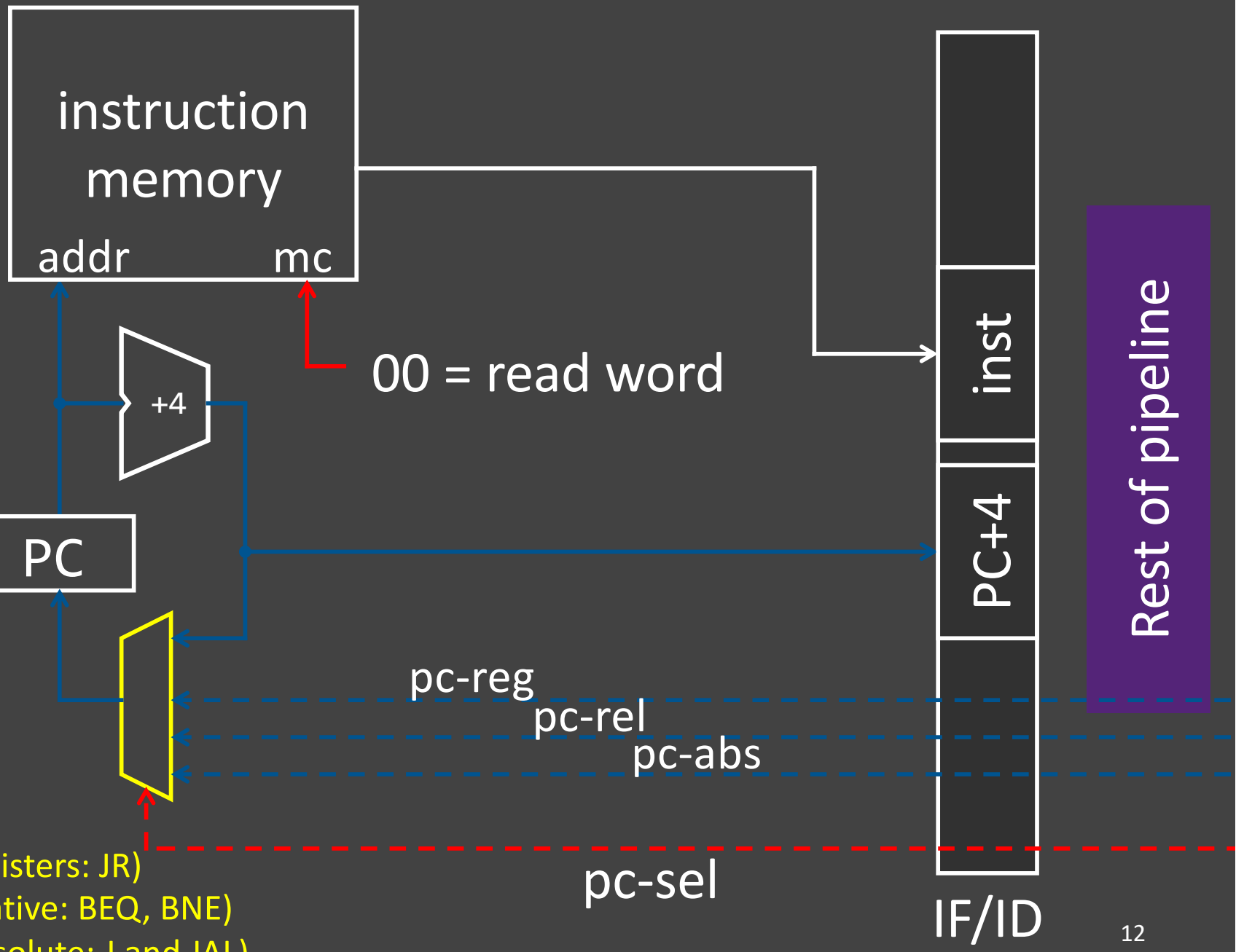
| Stage | Perform Functionality | Latch values of interest |
|---|---|---|
| **Fetch** | Use PC to index Program Memory, increment PC | Instruction bits (to be decoded)<br>PC + 4 (to compute branch targets) |
| **Decode** | Decode instruction, generate control signals, read register file | Control information,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

# Instruction Fetch (single-cycle)



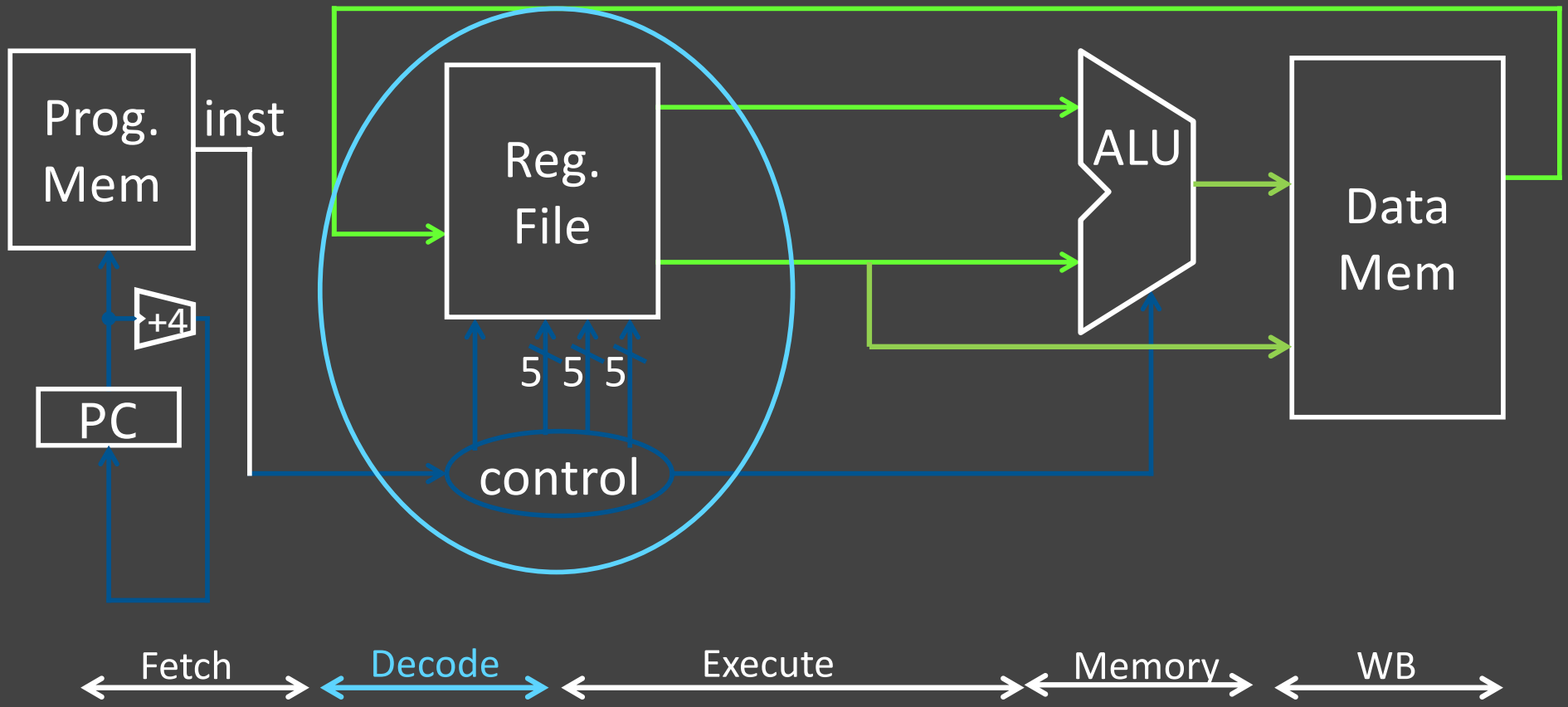

- Fetch 32-bit instruction from memory
- Increment PC = PC + 4

# Instruction Fetch (pipelined)



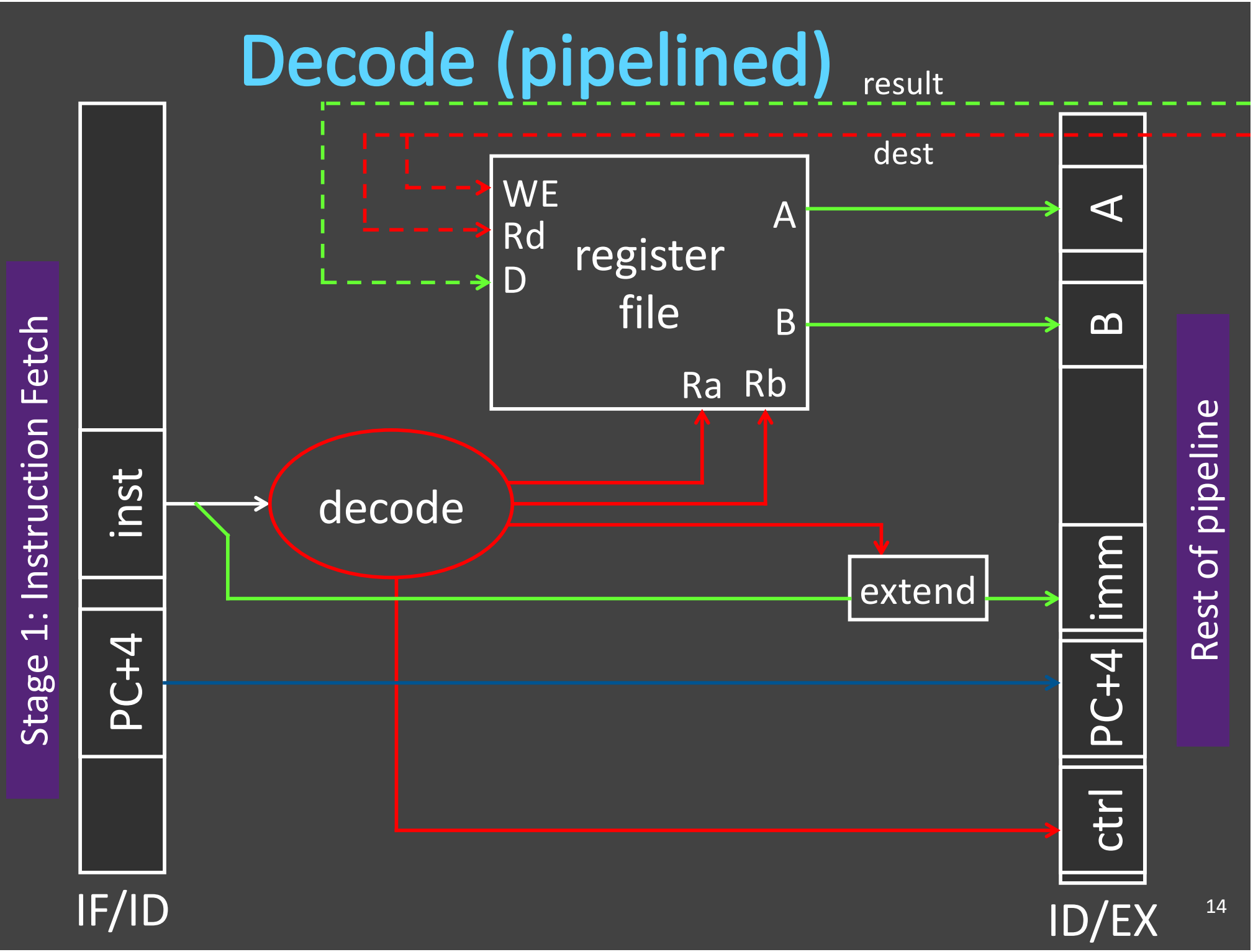

- PC+4
- pc-reg (PC registers: JR)
- pc-rel (PC-relative: BEQ, BNE)
- pc-abs (PC absolute: J and JAL)

# Instruction Decode (single-cycle)

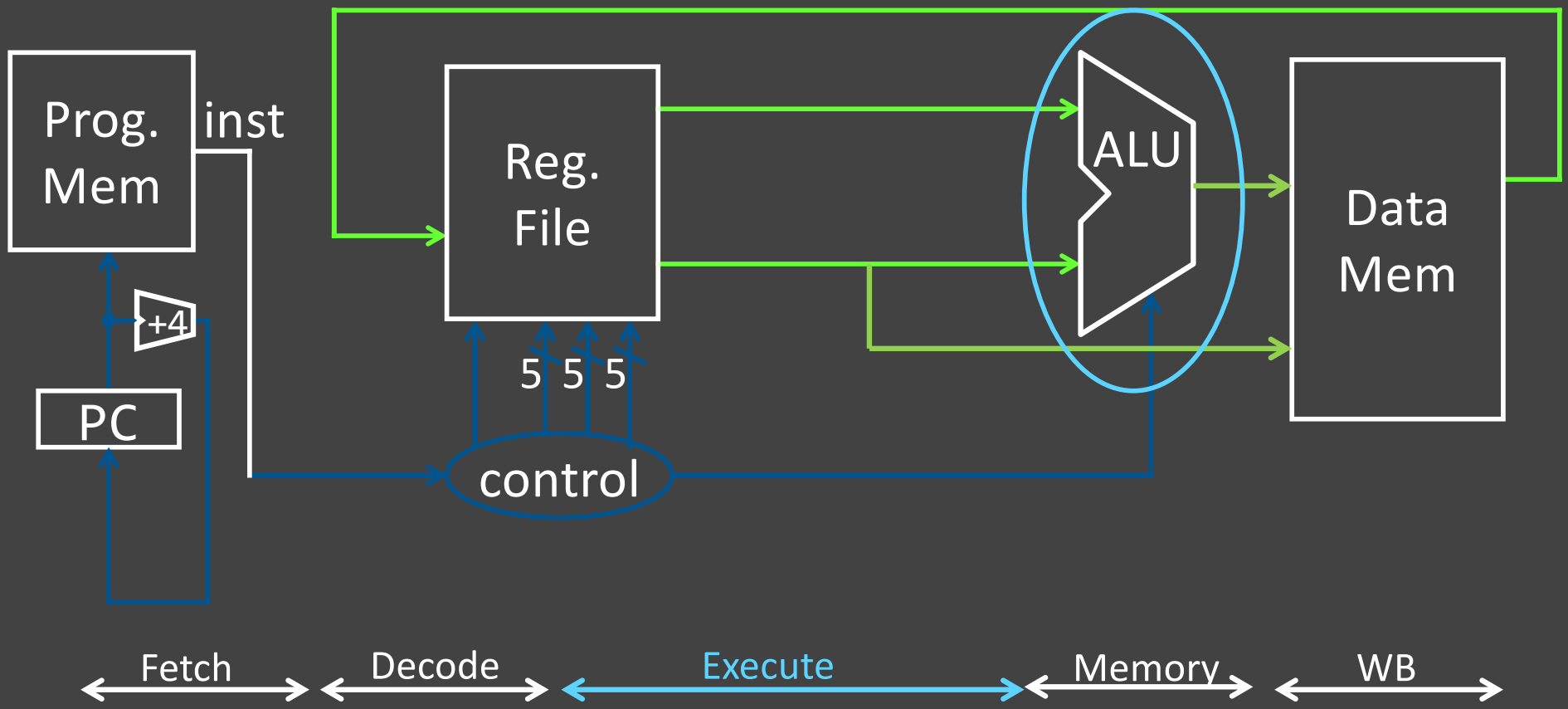



- Gather data from the instruction
- Read opcode; determine instruction type, field lengths
- Read in data from register file
  (0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

# Decode (pipelined)

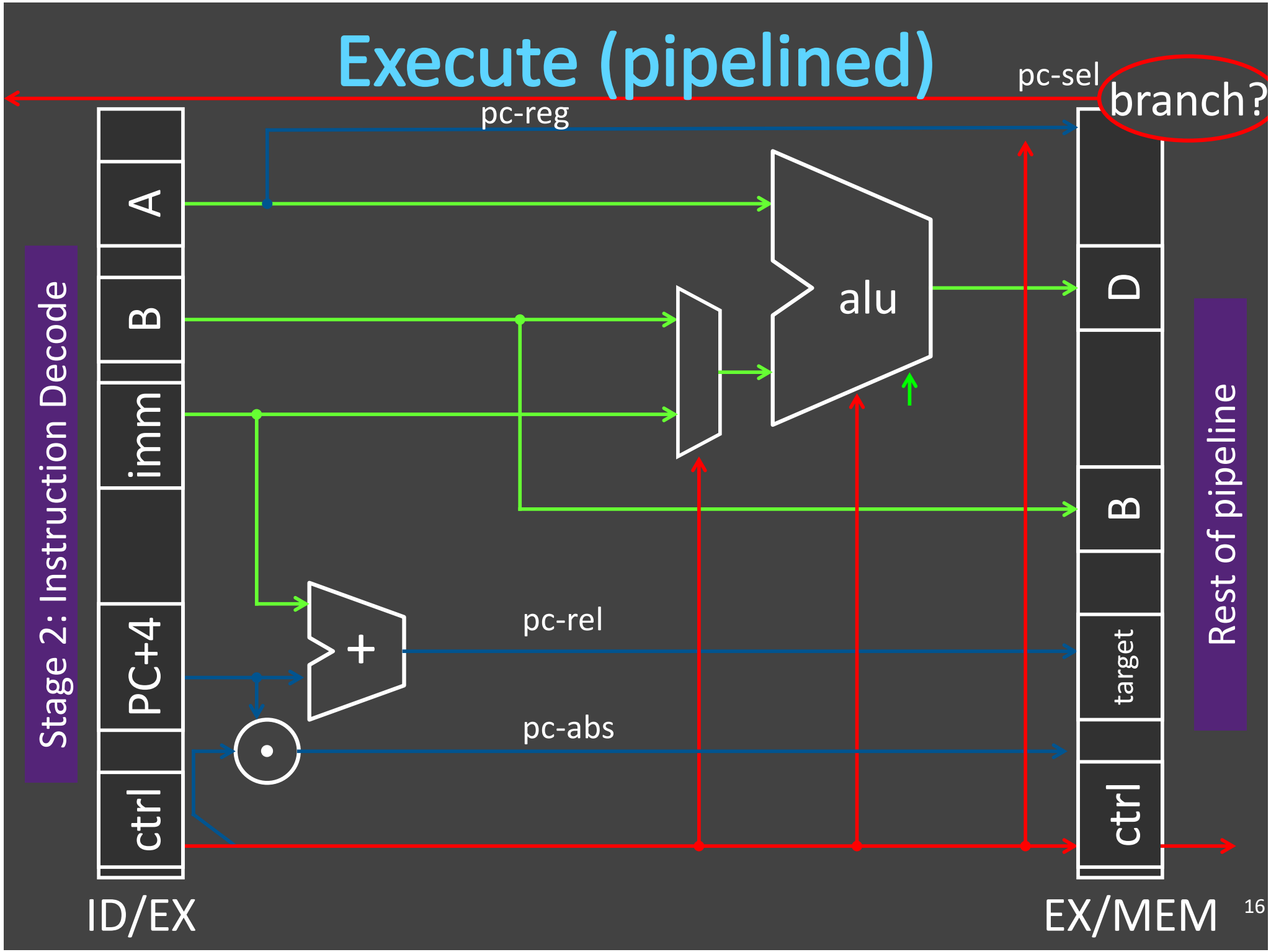


# Execution (single-cycle)

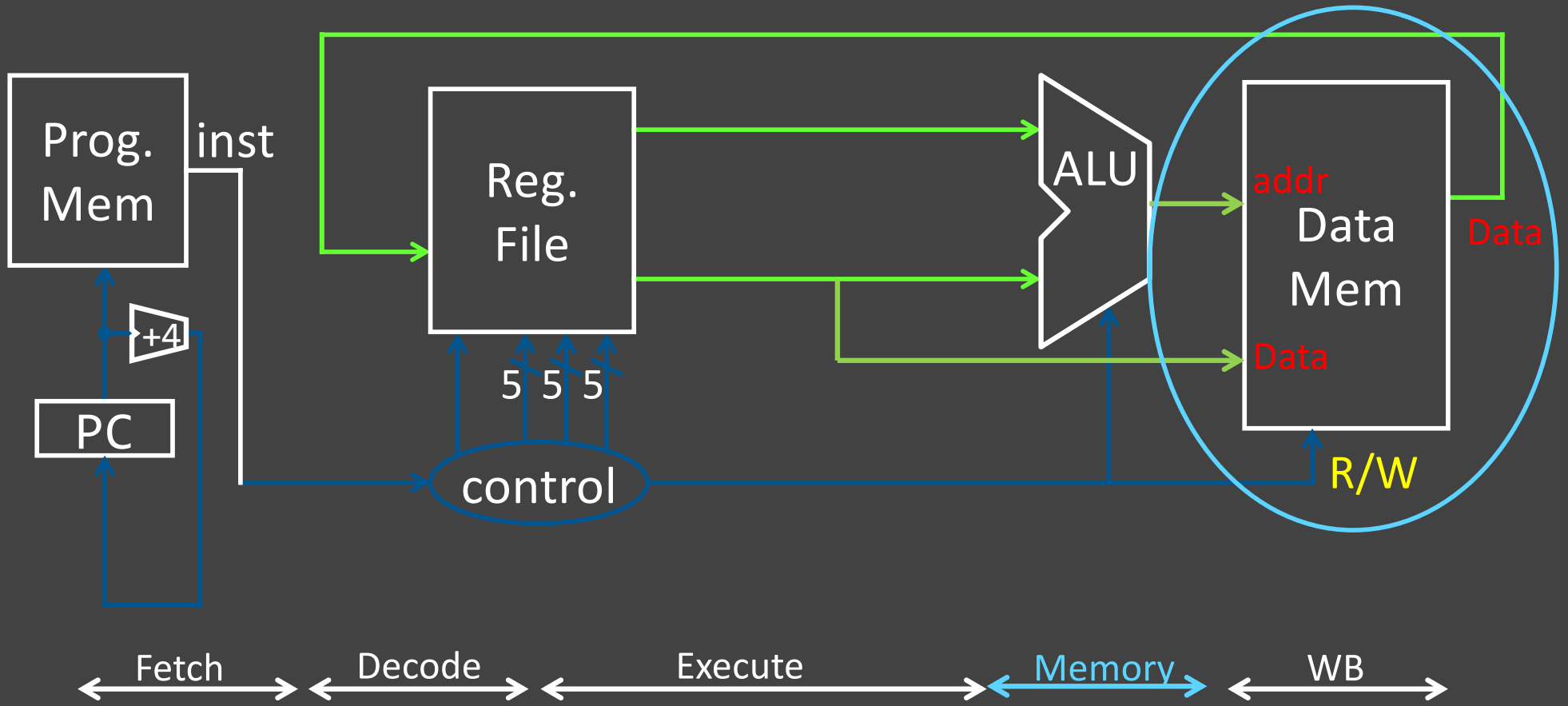



- Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)
- Load/Store? lw $t2, 32($t3) → Compute address

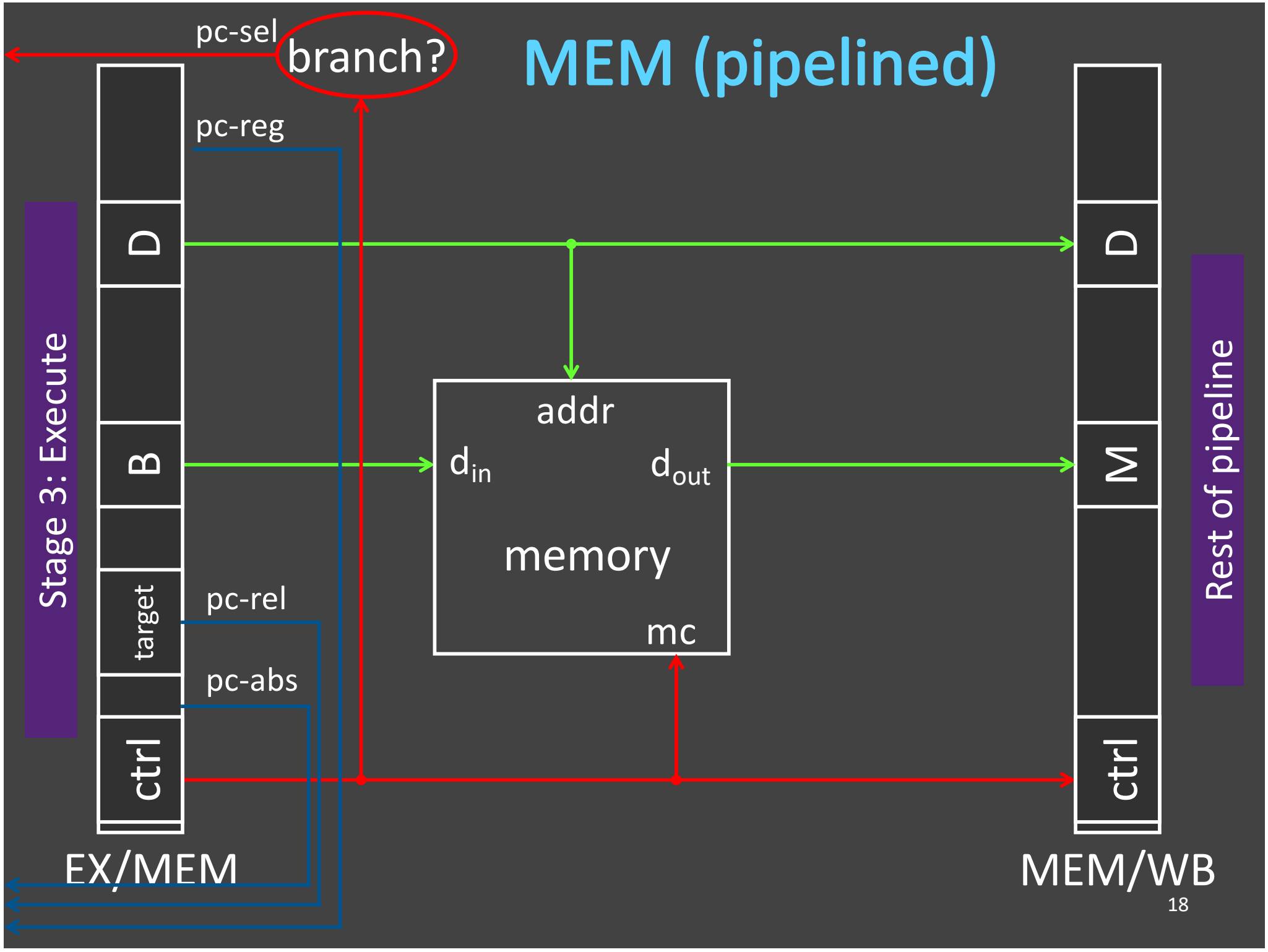
# Execute (pipelined)



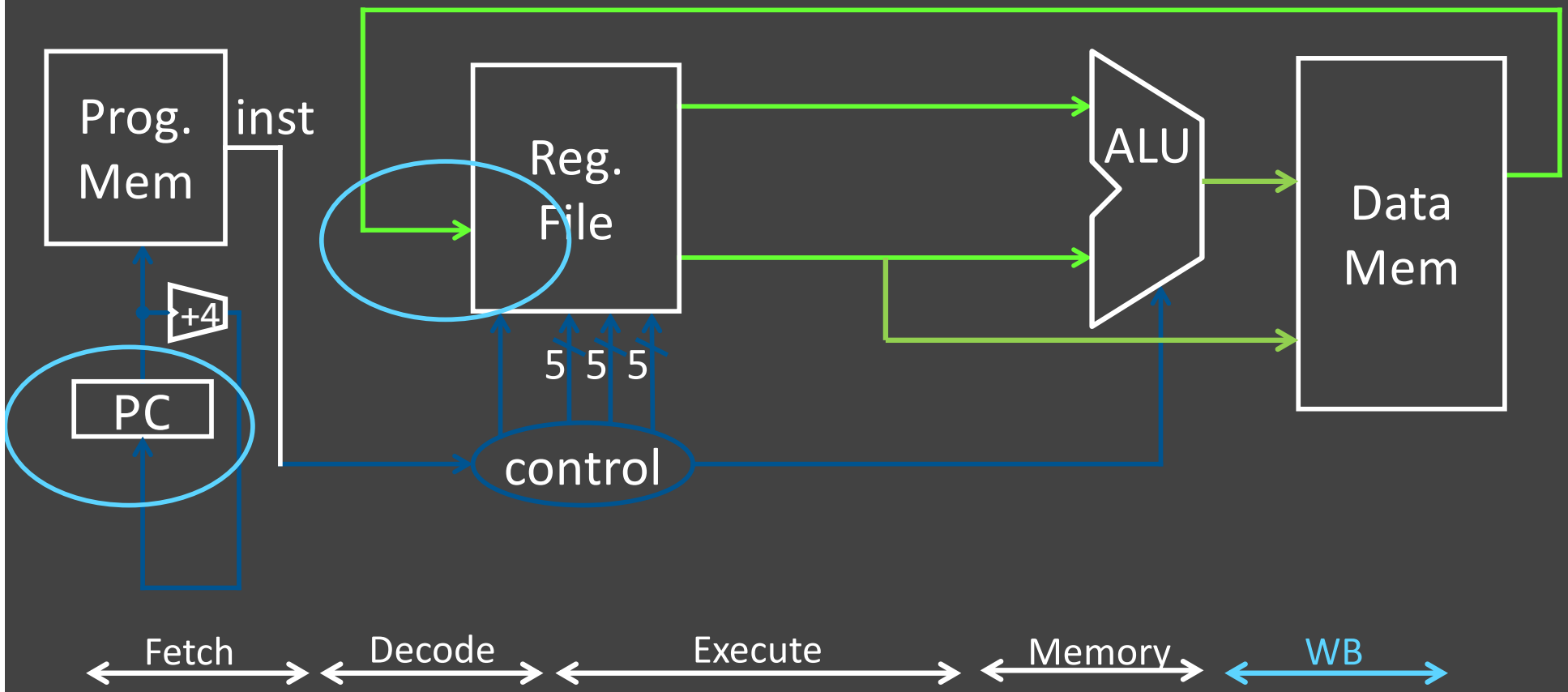
# Memory access (single-cycle)




- Used by load and store instructions only
- Other instructions will skip this stage

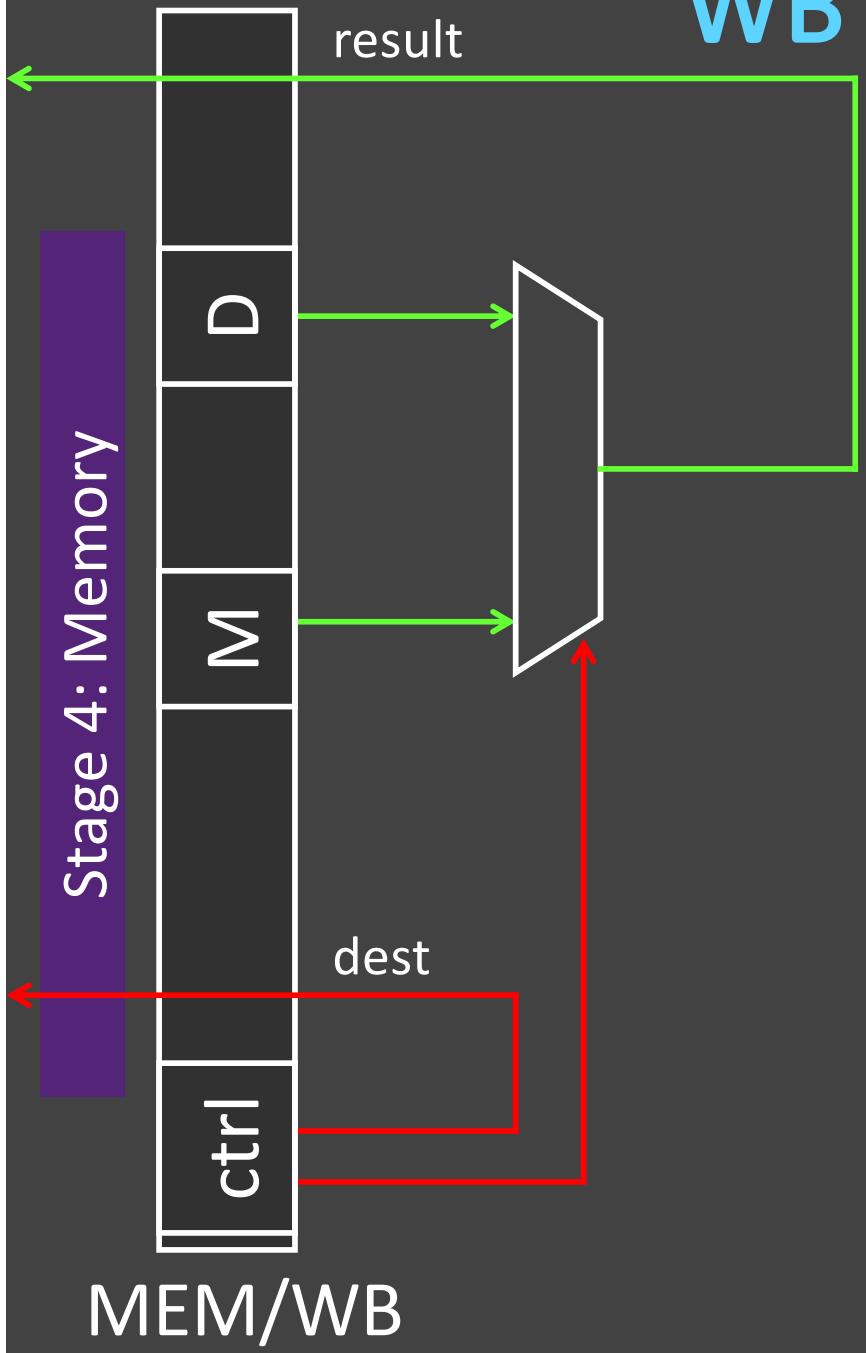

MEM (pipelined)
pc-sel
branch?
pc-reg
D
B
target
ctrl
pc-rel
pc-abs
addr
$d_{in}$
$d_{out}$
memory
mc
D
M
ctrl
Stage 3: Execute
Rest of pipeline
EX/MEM
MEM/WB

# Writeback (single-cycle)

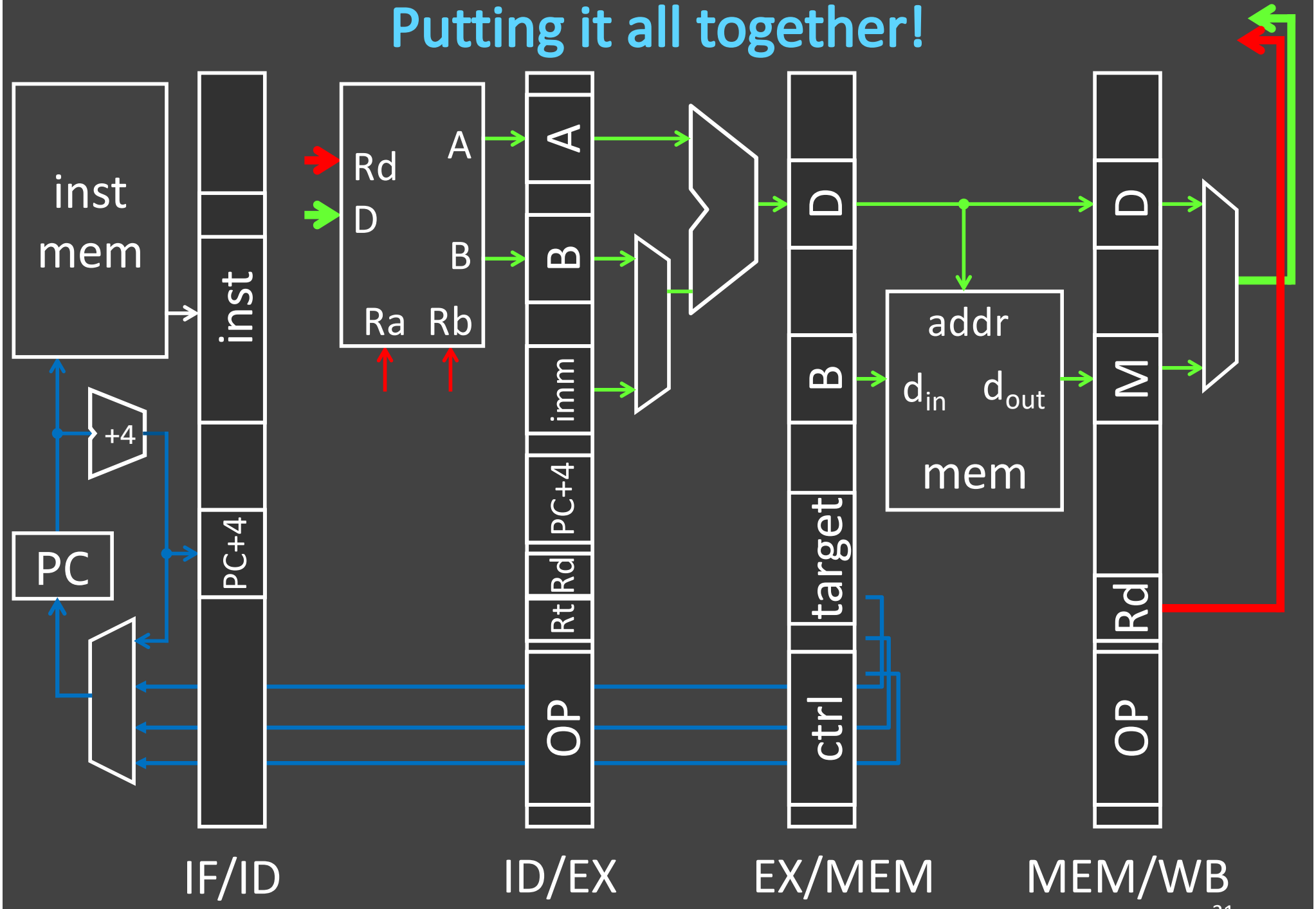



- Write to register file
  - For arithmetic ops, logic, shift, etc, load. What about stores?
- Update PC
  - For branches, jumps

# WB (pipelined)



# Putting it all together!



# iClicker Question

Consider a non-pipelined processor with clock period C (*e.g.*, 50 ns). If you divide the processor into N stages (*e.g.*, 5) , your new clock period will be:

A. C
B. N
C. less than C/N
D. C/N
E. greater than C/N

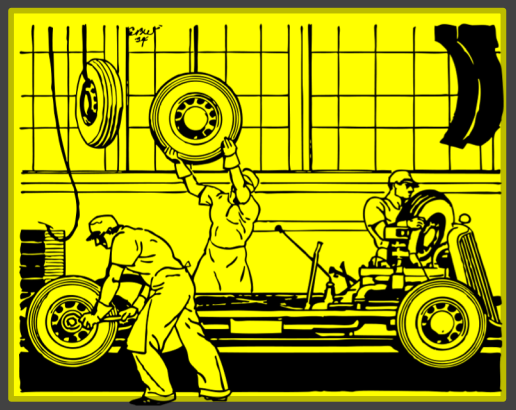
# MIPS is *designed* for pipelining

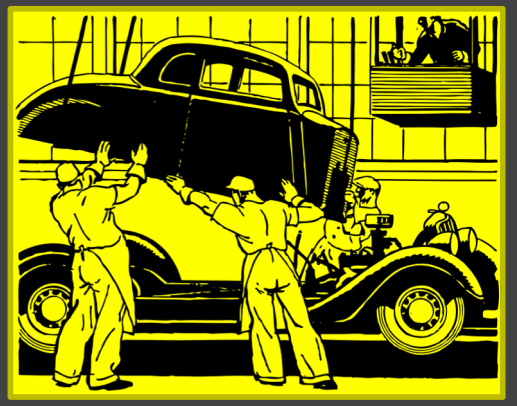
- Instructions same length
  - 32 bits, easy to fetch and then decode
- 3 types of instruction formats
  - Easy to route bits between stages
  - Can read a register source before even knowing what the instruction is
- Memory access through lw and sw only
  - Access memory after ALU

# Agenda

5-stage Pipeline

- Implementation
- Working Example





Hazards

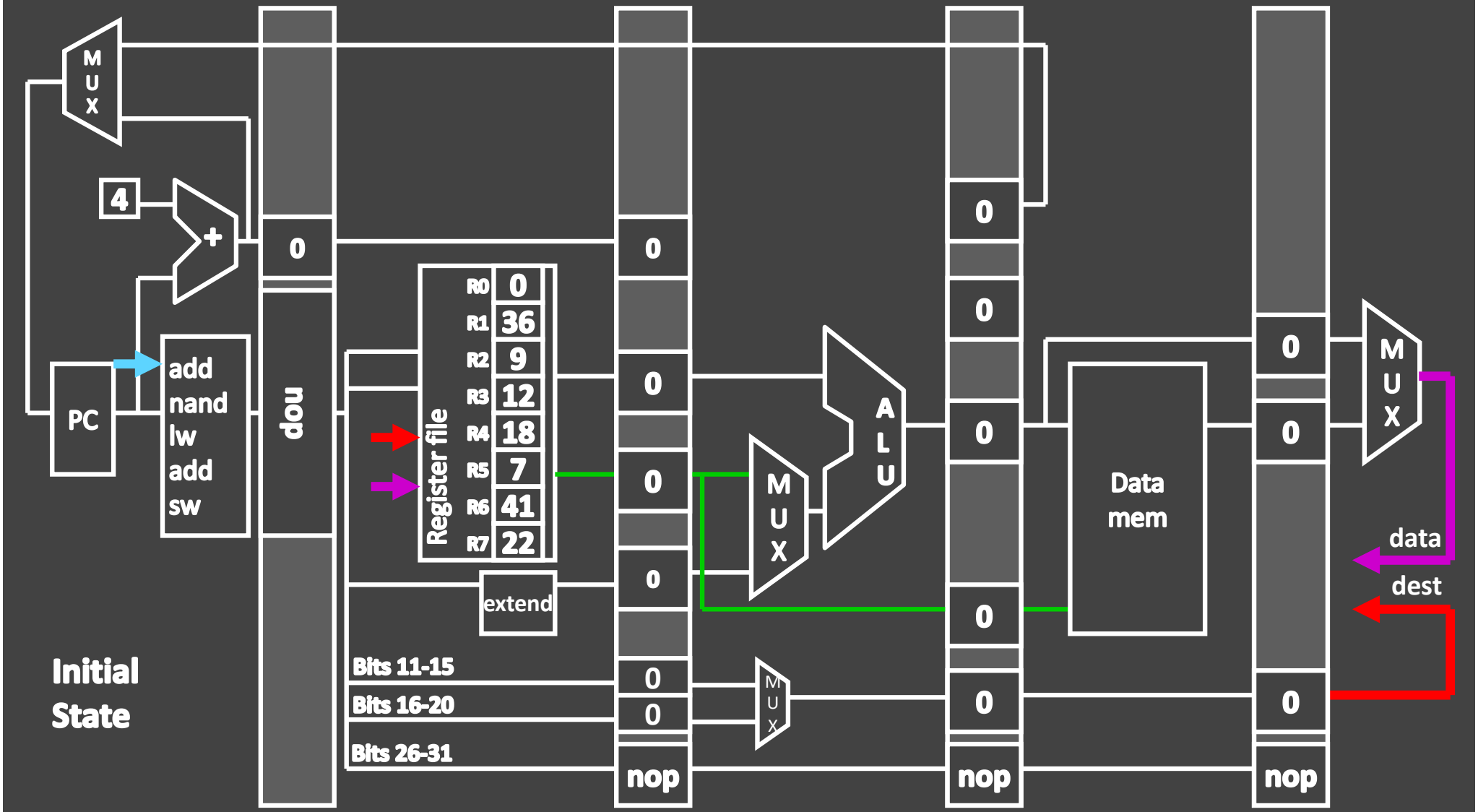
- Structural
- Data Hazards
- Control Hazards

# Example: : Sample Code (Simple)

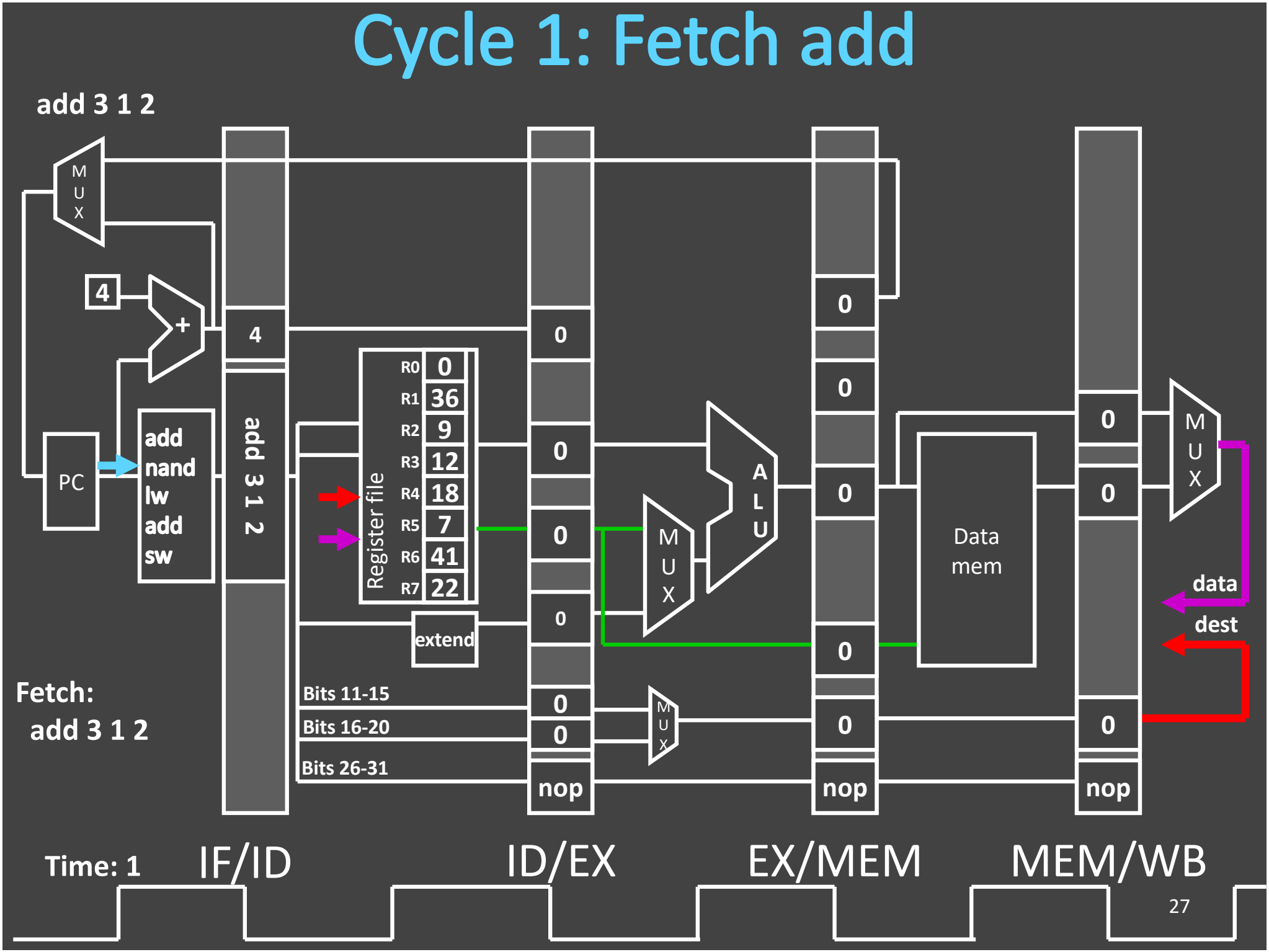
```
add     r3 ← r1, r2
nand    r6 ← r4, r5
lw      r4 ← 20(r2)
add     r5 ← r2, r5
sw      r7 → 12(r3)
```

Assume 8-register machine

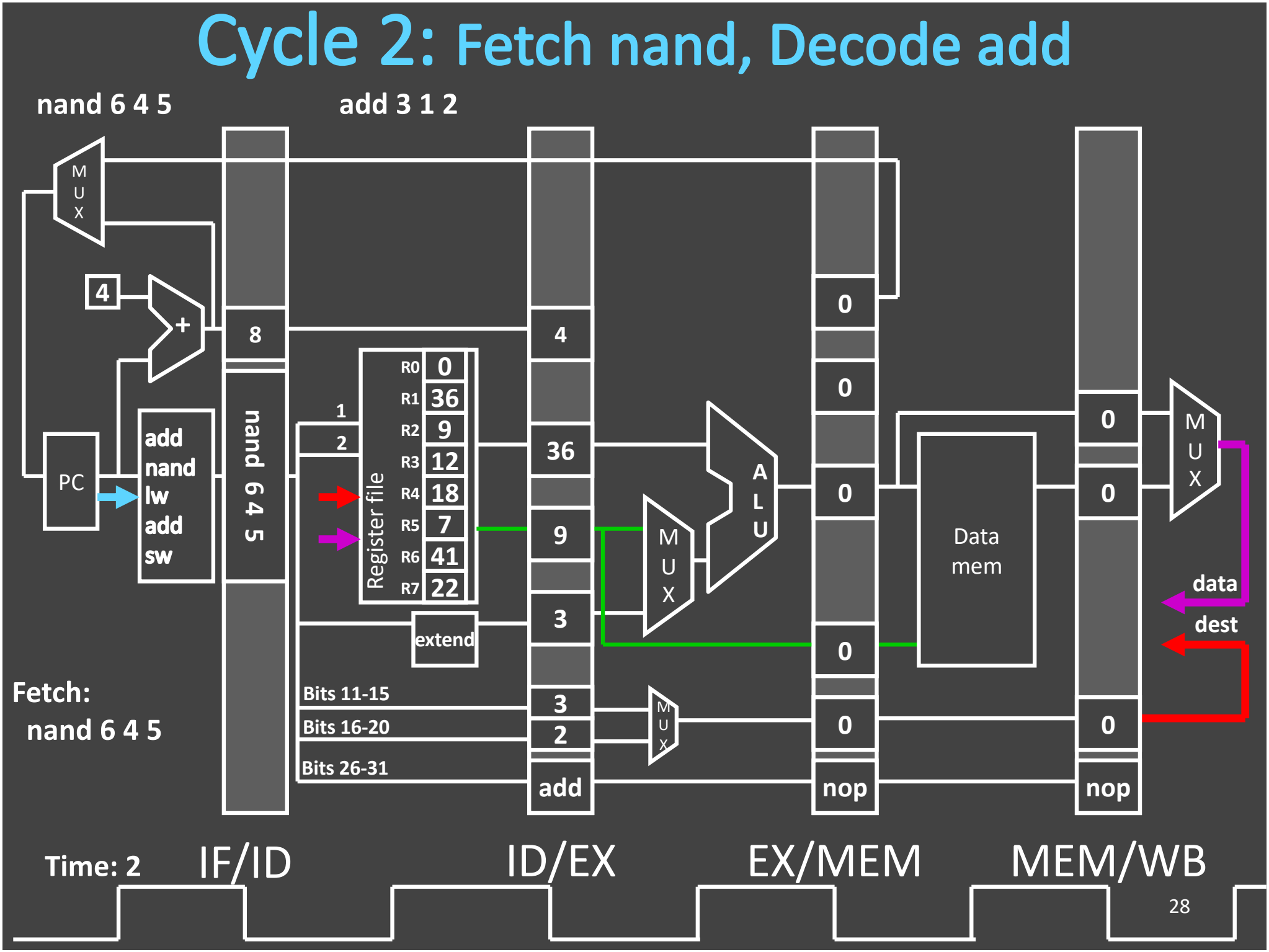
# Example: Start State @ Cycle 0



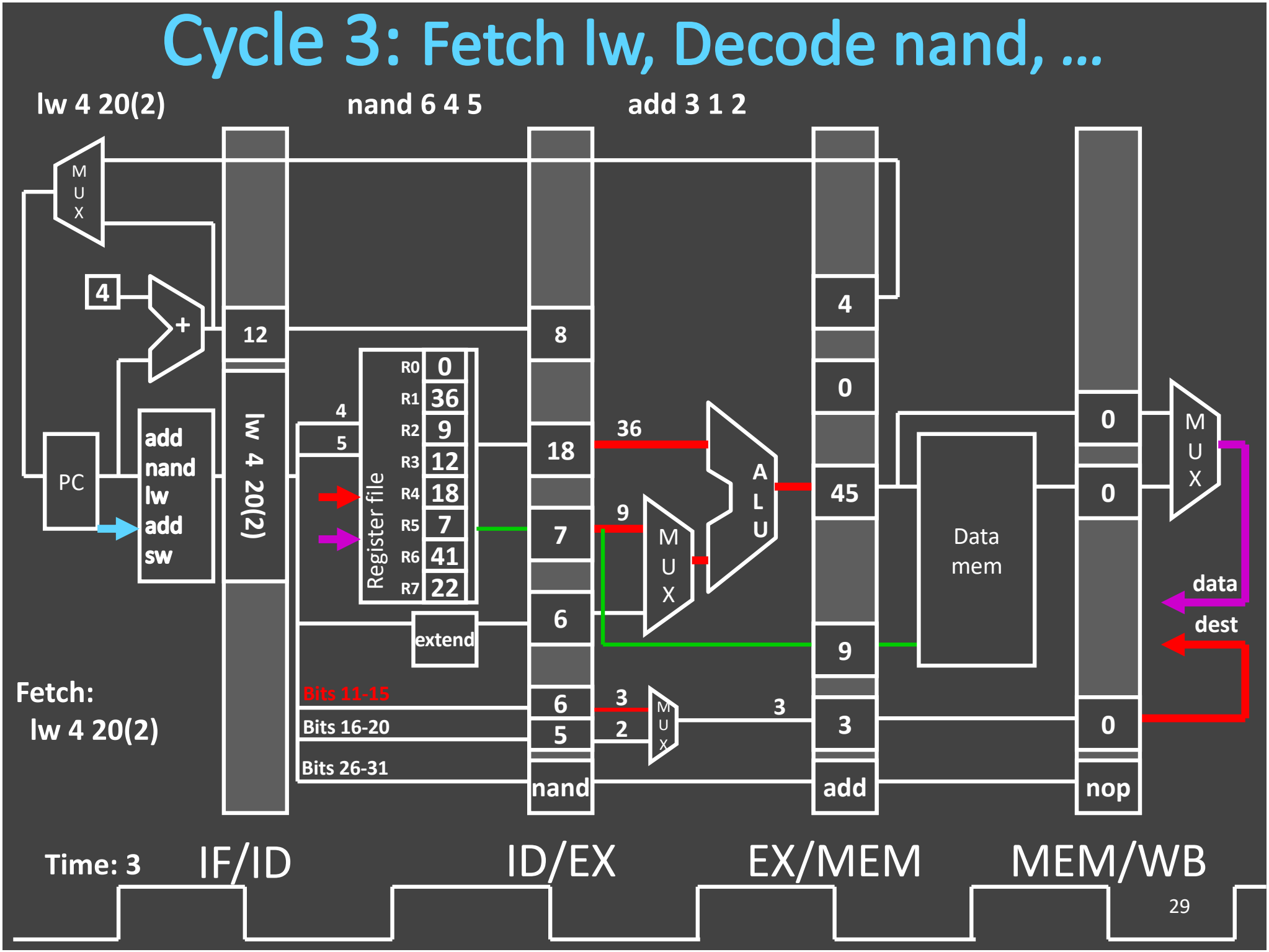
# Cycle 1: Fetch add



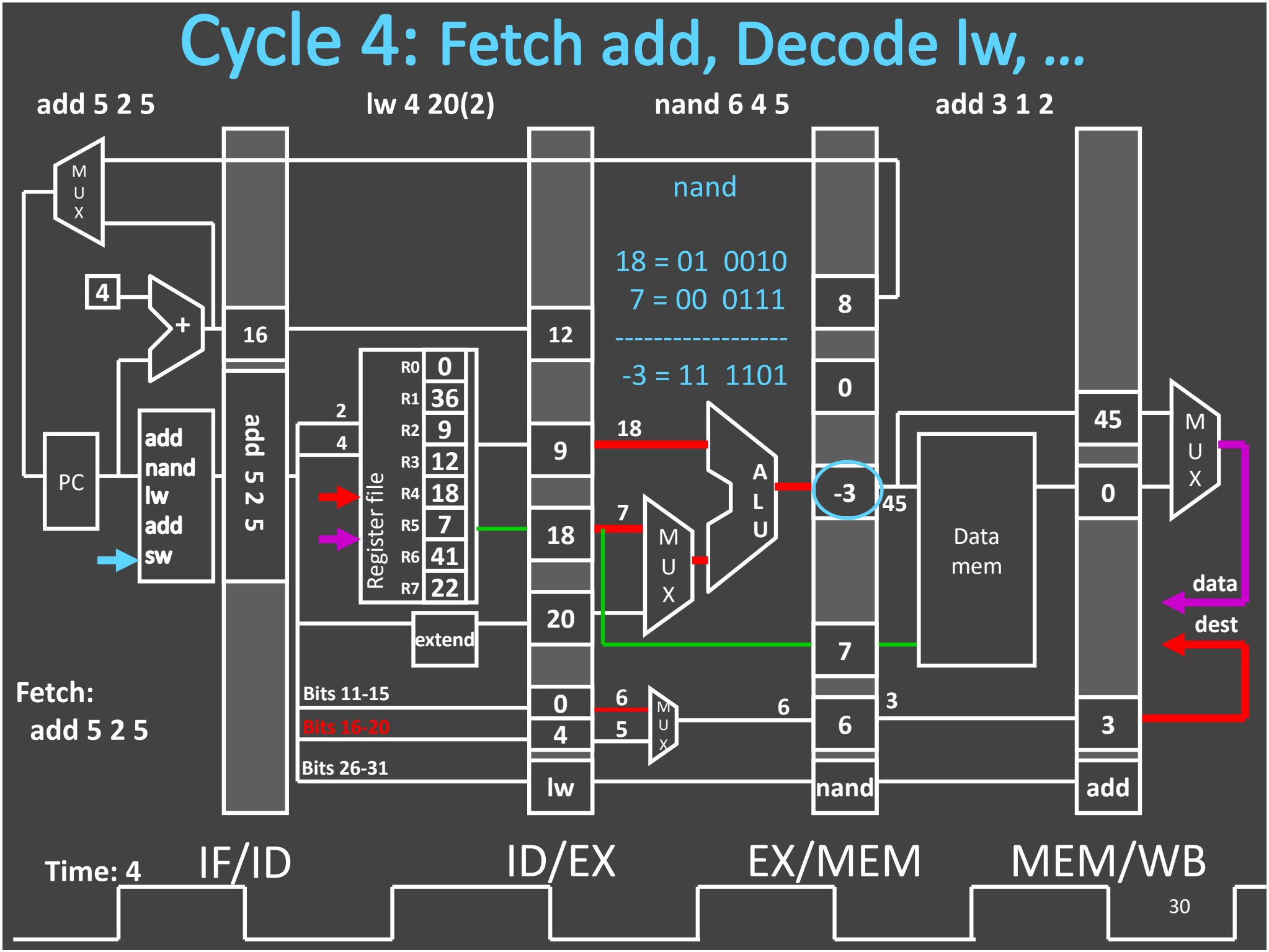
# Cycle 2: Fetch nand, Decode add



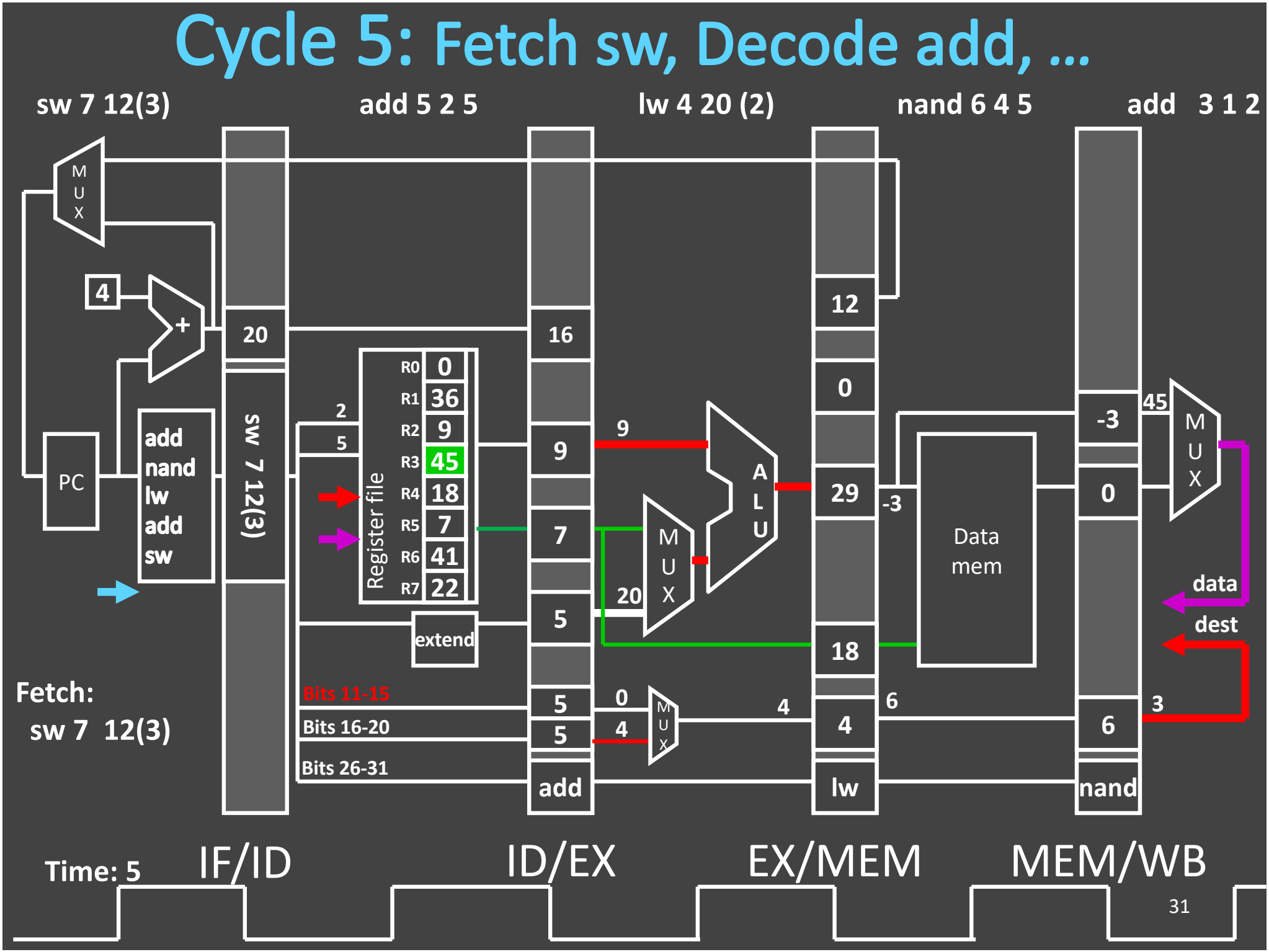
# Cycle 3: Fetch lw, Decode nand, …



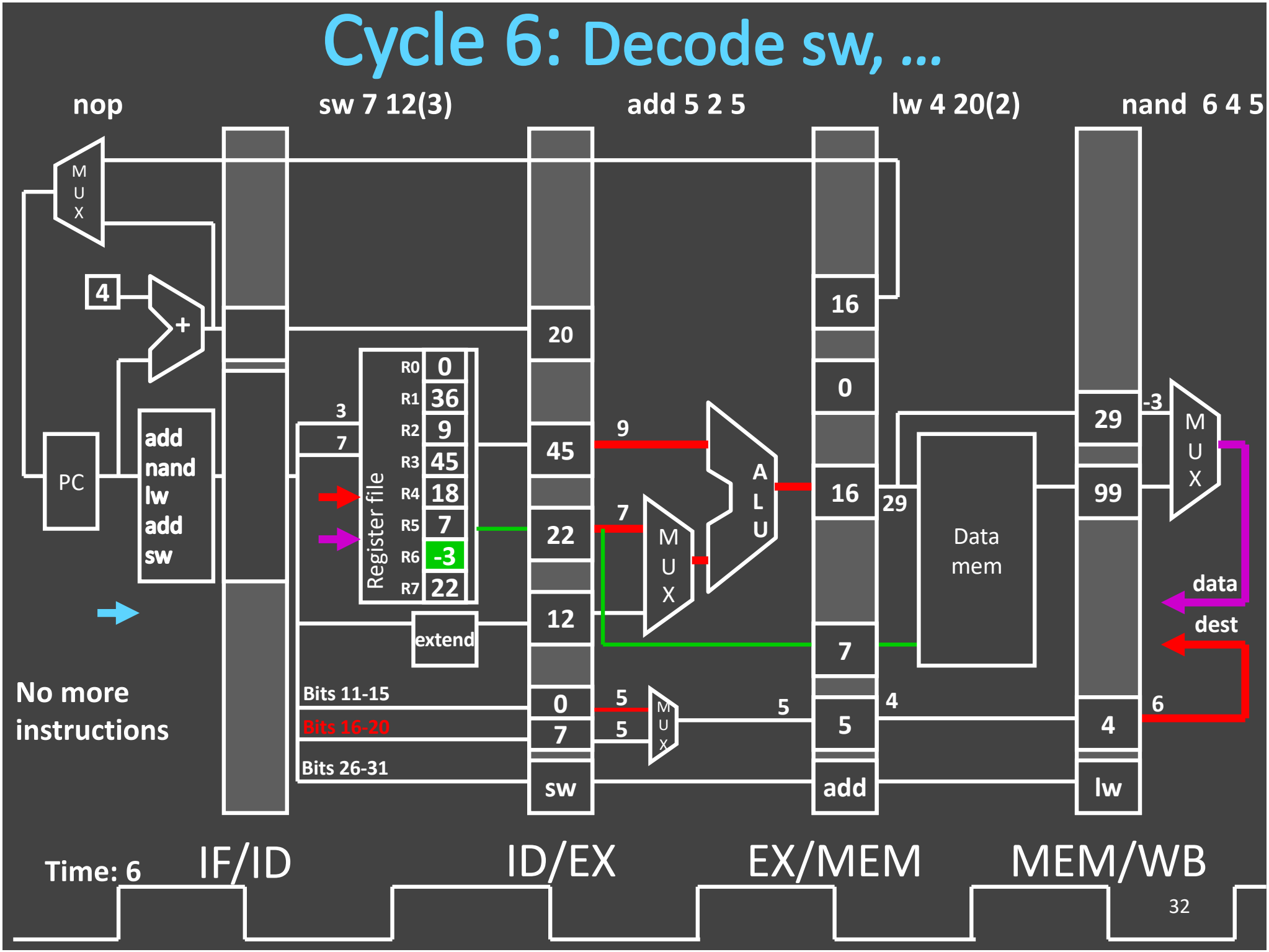
# Cycle 4: Fetch add, Decode lw, …



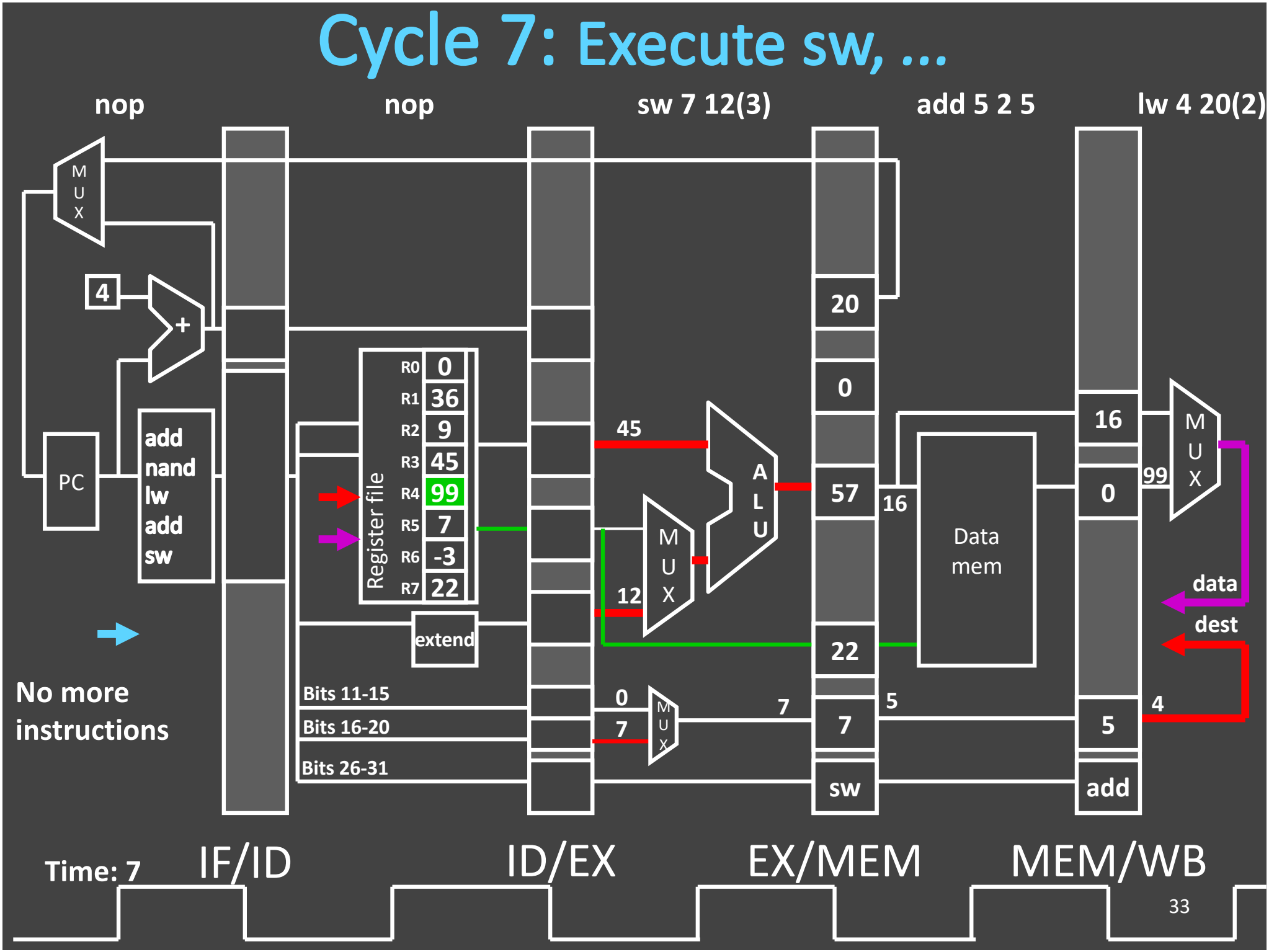
# Cycle 5: Fetch sw, Decode add, …



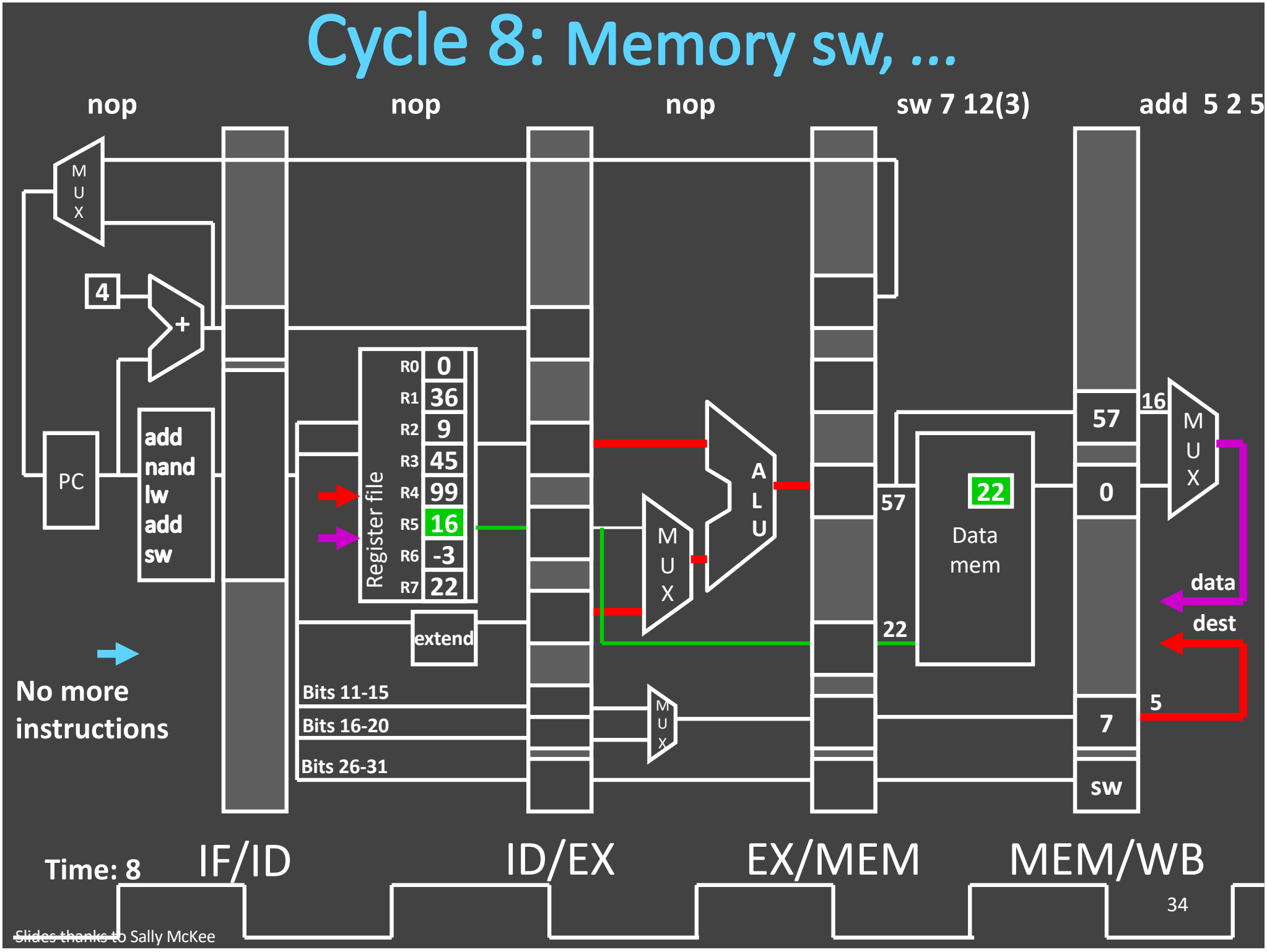
# Cycle 6: Decode sw, …



# Cycle 7: Execute sw, ...



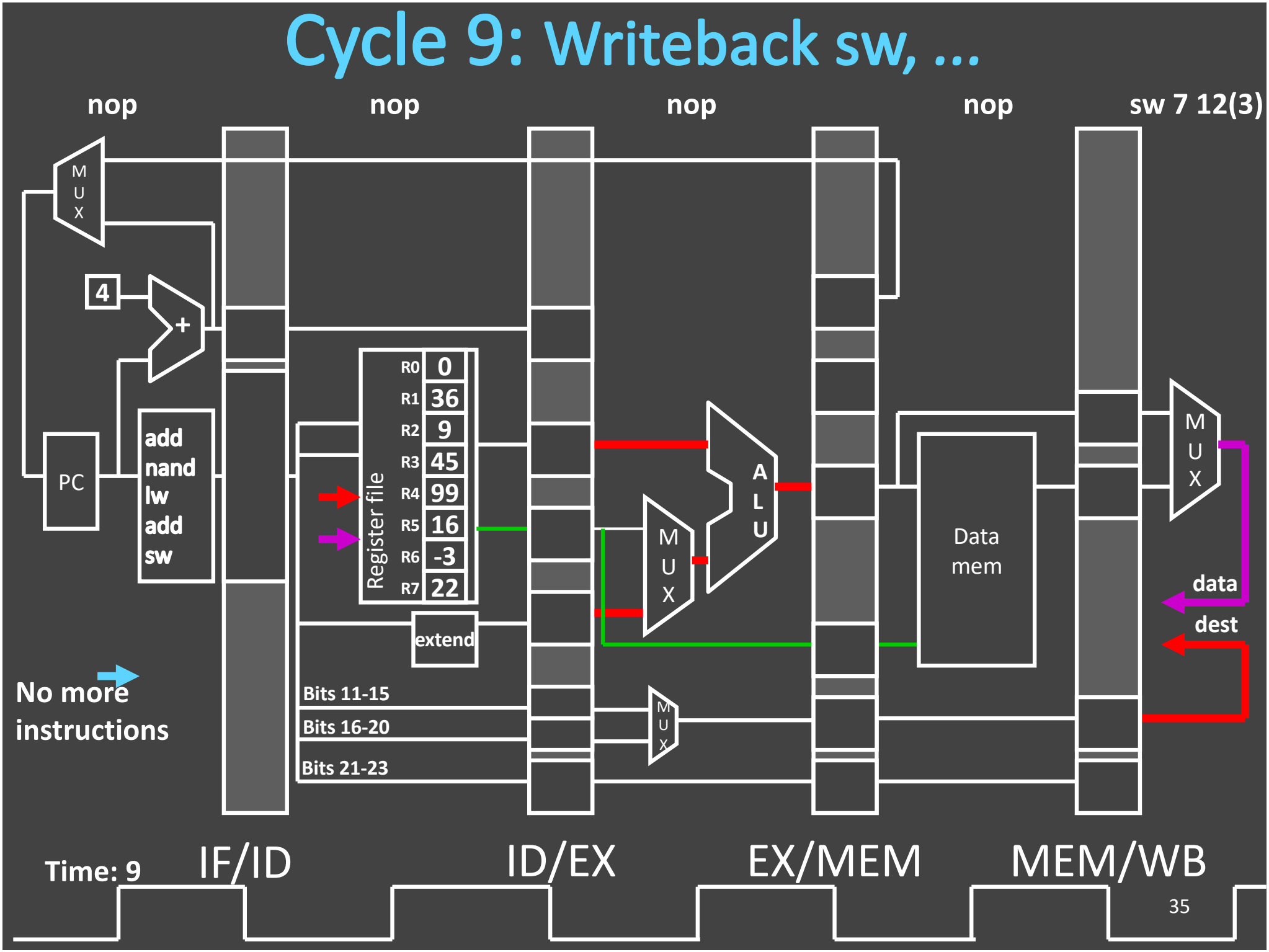
# Cycle 8: Memory sw, ...






Slides thanks to Sally McKee

# Cycle 9: Writeback sw, ...



# iClicker Question

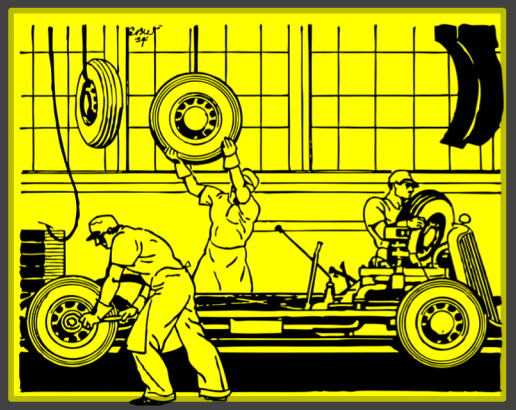
Pipelining is great because:

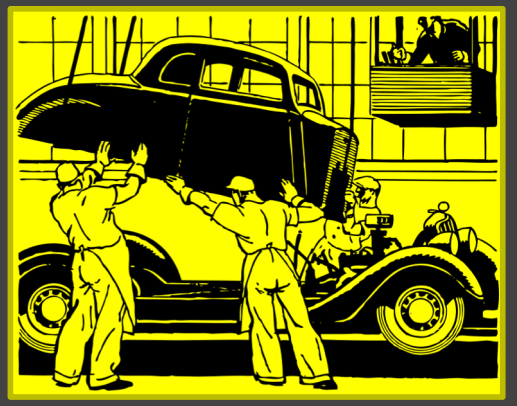
A. You can fetch and decode the same instruction at the same time.
B. You can fetch two instructions at the same time.
C. You can fetch one instruction while decoding another.
D. Instructions only need to visit the pipeline stages that they require.
E. C and D

# 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)

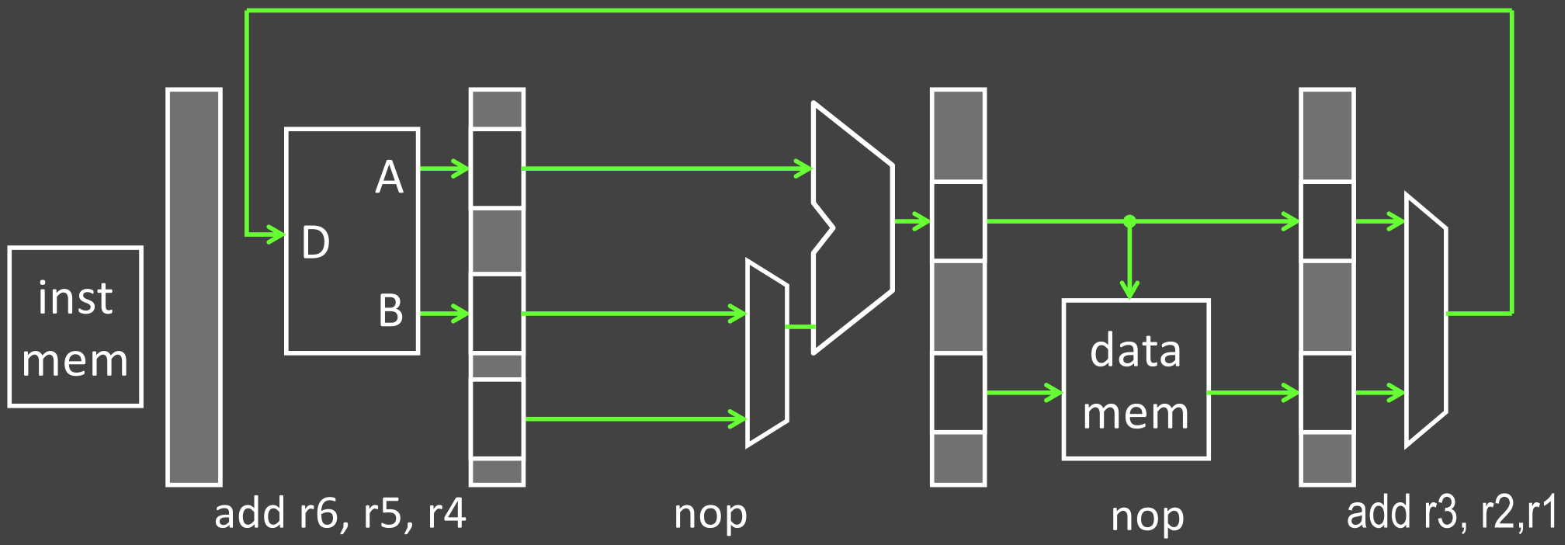
2. Data hazards

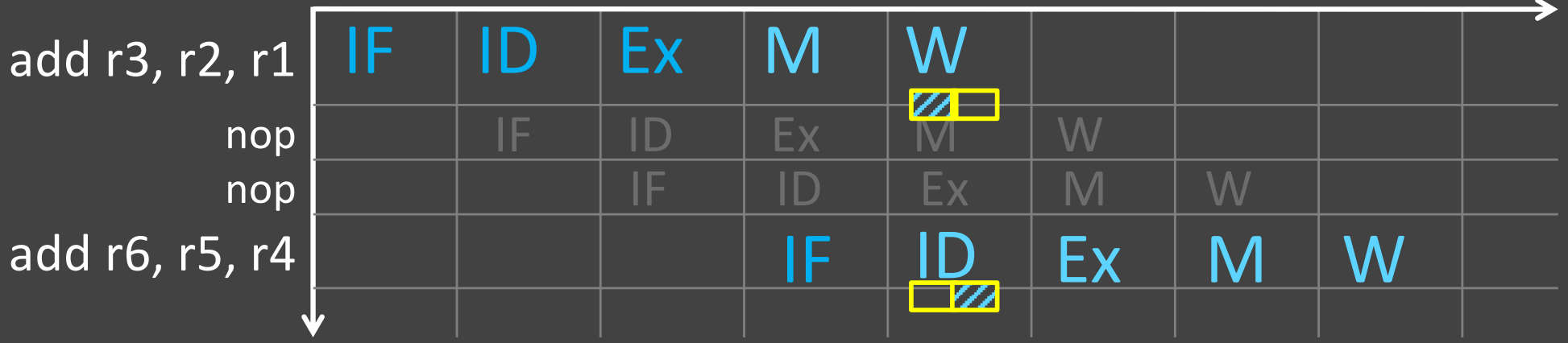
   Instruction output needed before it's available

3. Control hazards

   Next instruction PC unknown at time of Fetch

# Resolving Register File Structural Hazard




**Problem:** Need to read from and write to Register File at the same time

**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

# iClicker Question

## Data Hazards

- register file (RF) reads occur in stage 2 (ID)
- RF writes occur in stage 5 (WB)
- RF written in ½ half, read in second ½ half of cycle
- Processor is built exactly as we've seen up until this slide.

```
x10: add r3 ← r1, r2
x14: sub r5 ← r3, r4
```

1. Is there a dependence?
2. Is there a hazard?

A) Yes
B) No
C) Cannot tell with the information given.

# iClicker Follow-up

Which of the following statements is true?

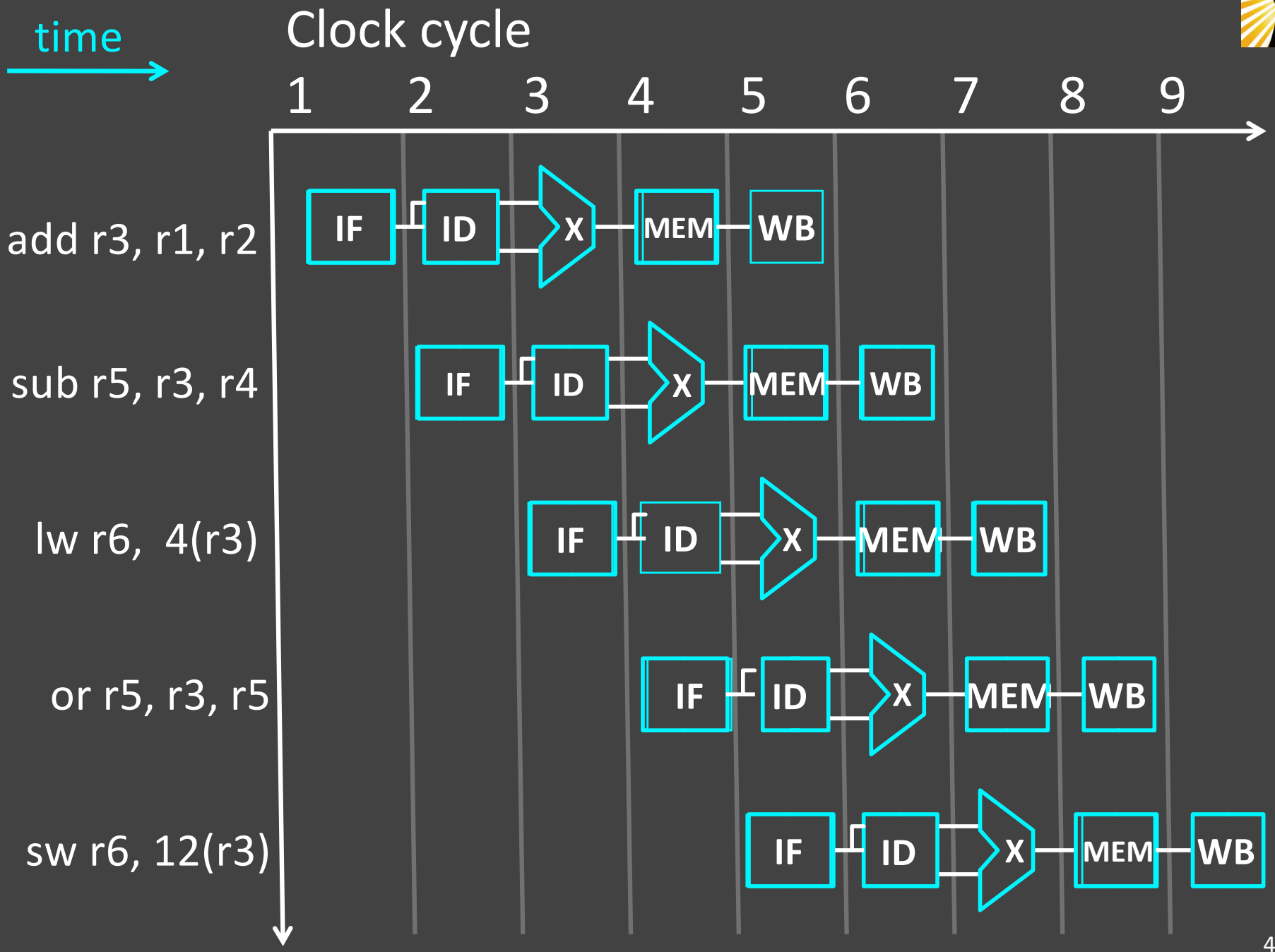
A. Whether there is a data dependence between two instructions depends on the machine the program is running on.

B. Whether there is a data hazard between two instructions depends on the machine the program is running on.

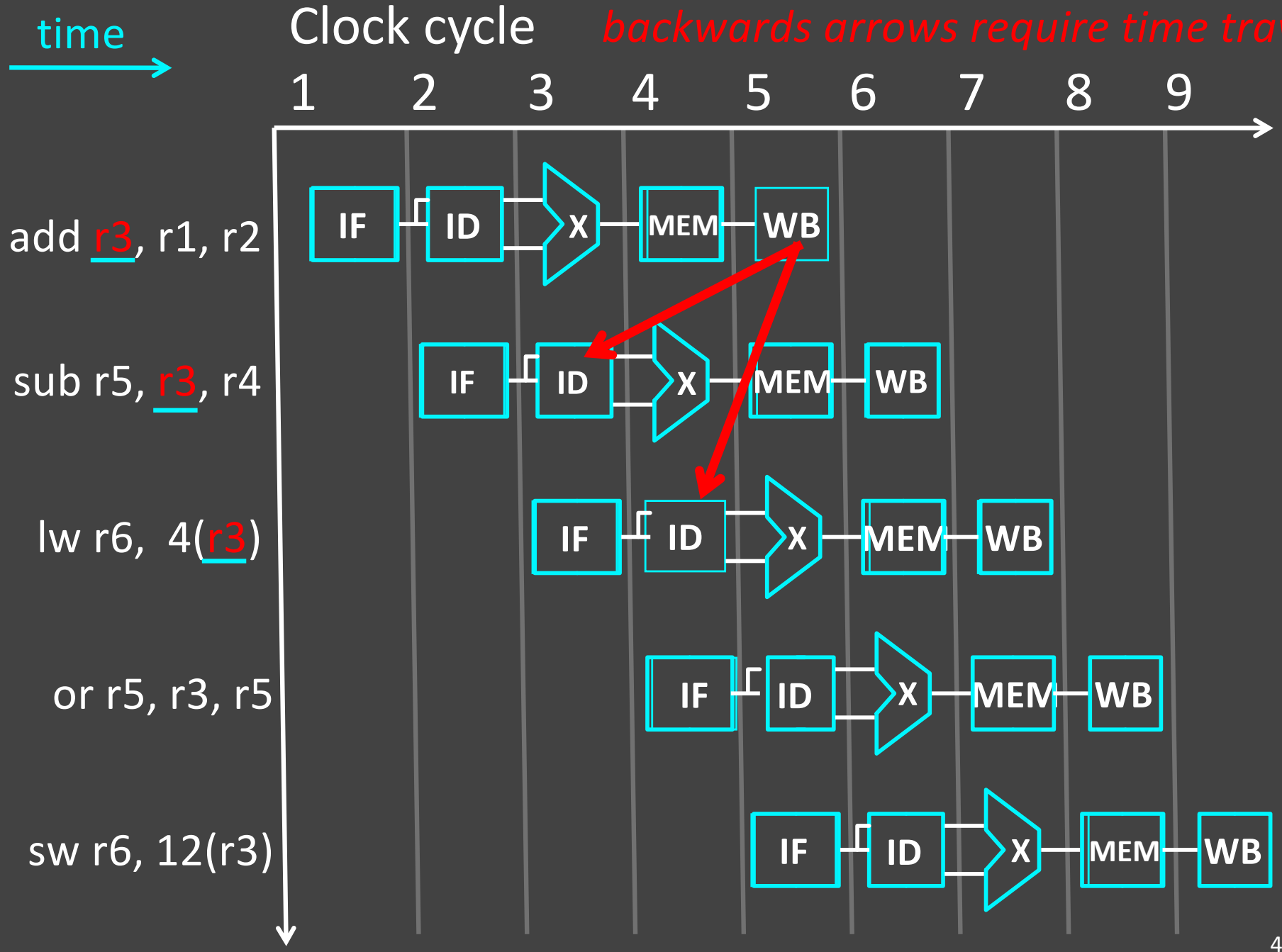
C. Both A & B

D. Neither A nor B

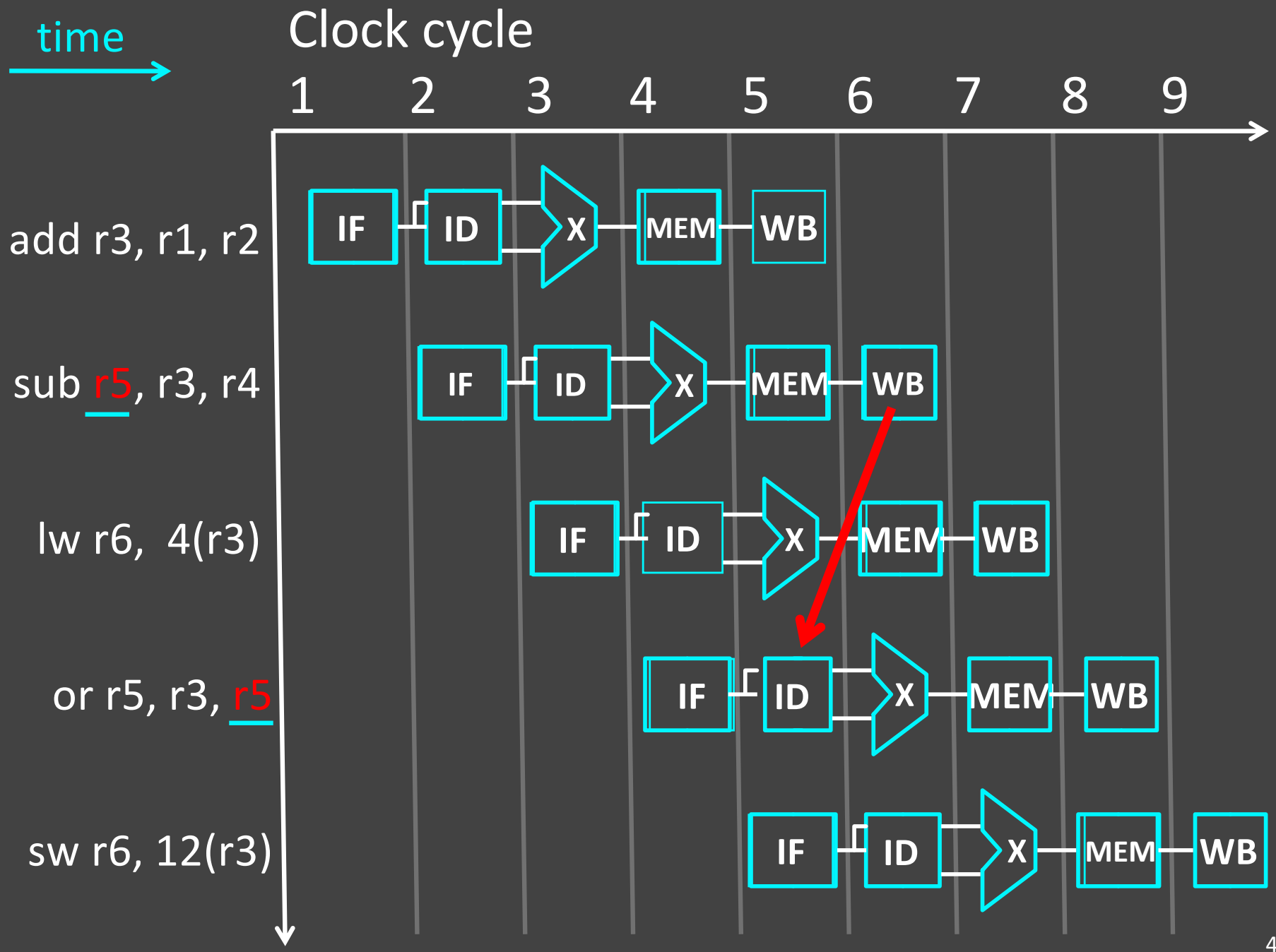
# Where are the Data Hazards?



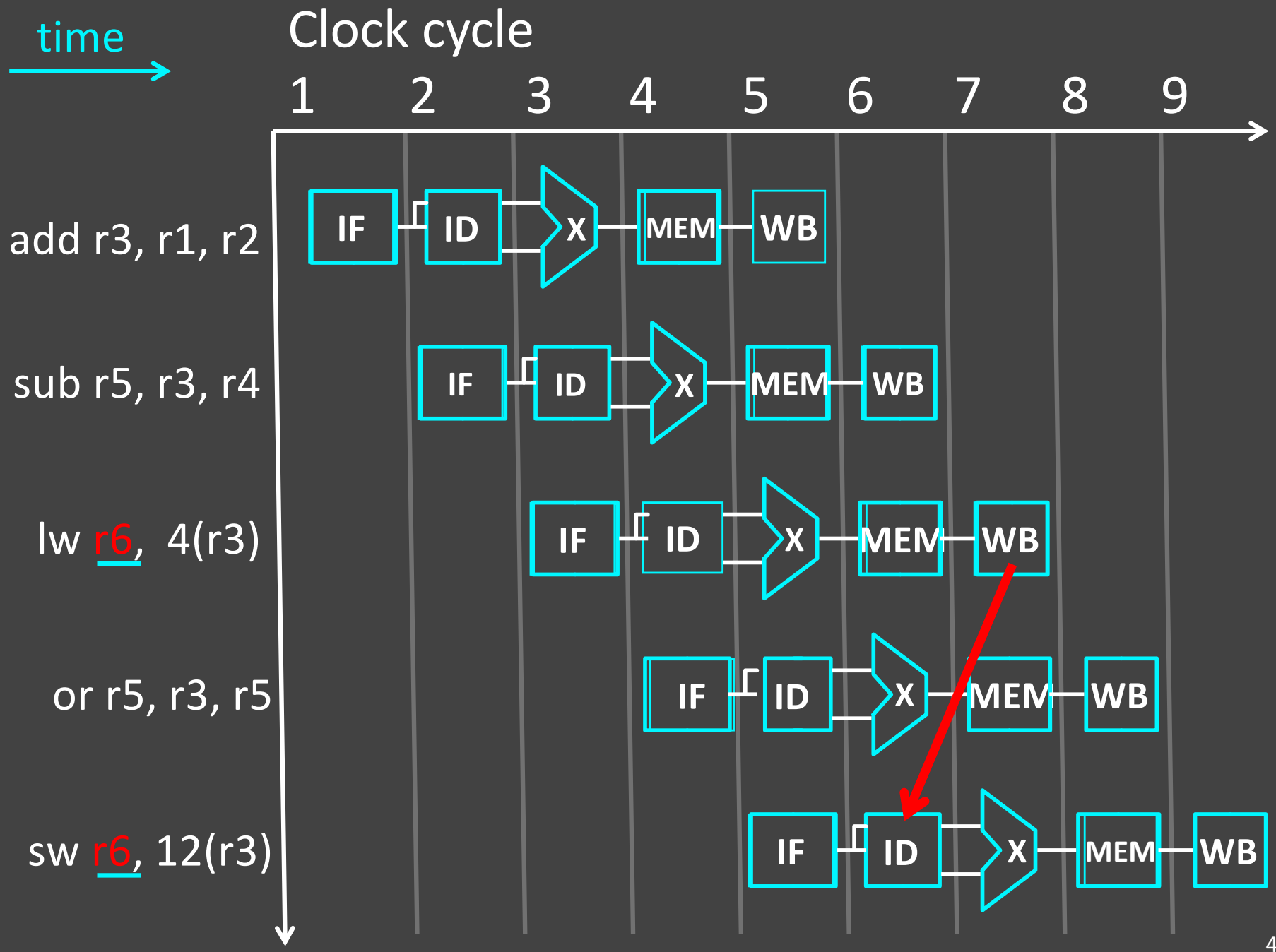
# Visualizing Data Hazards (1)



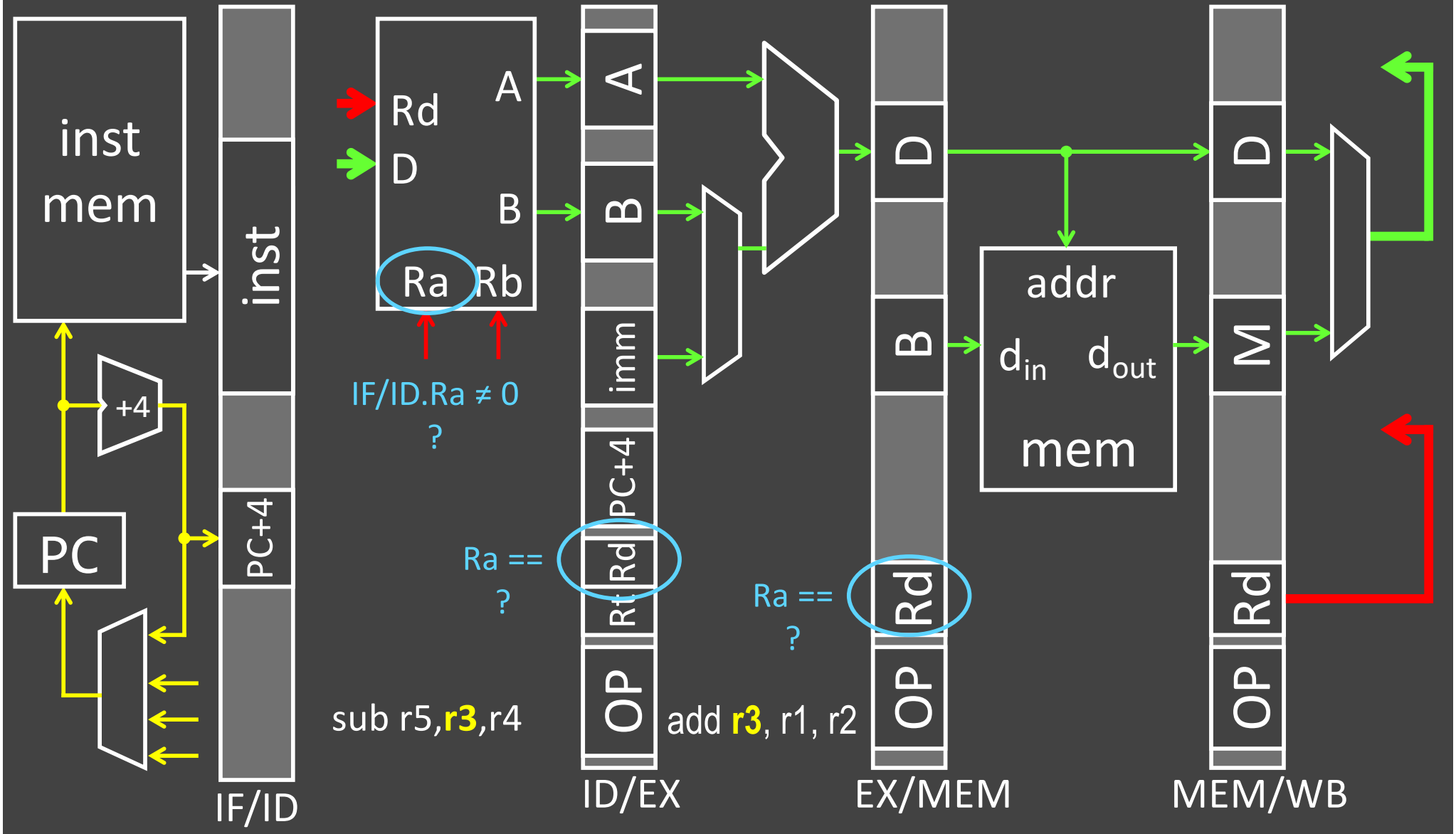
# Visualizing Data Hazards (2)



# Visualizing Data Hazards (3)



# Detecting Data Hazards




Problem = (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

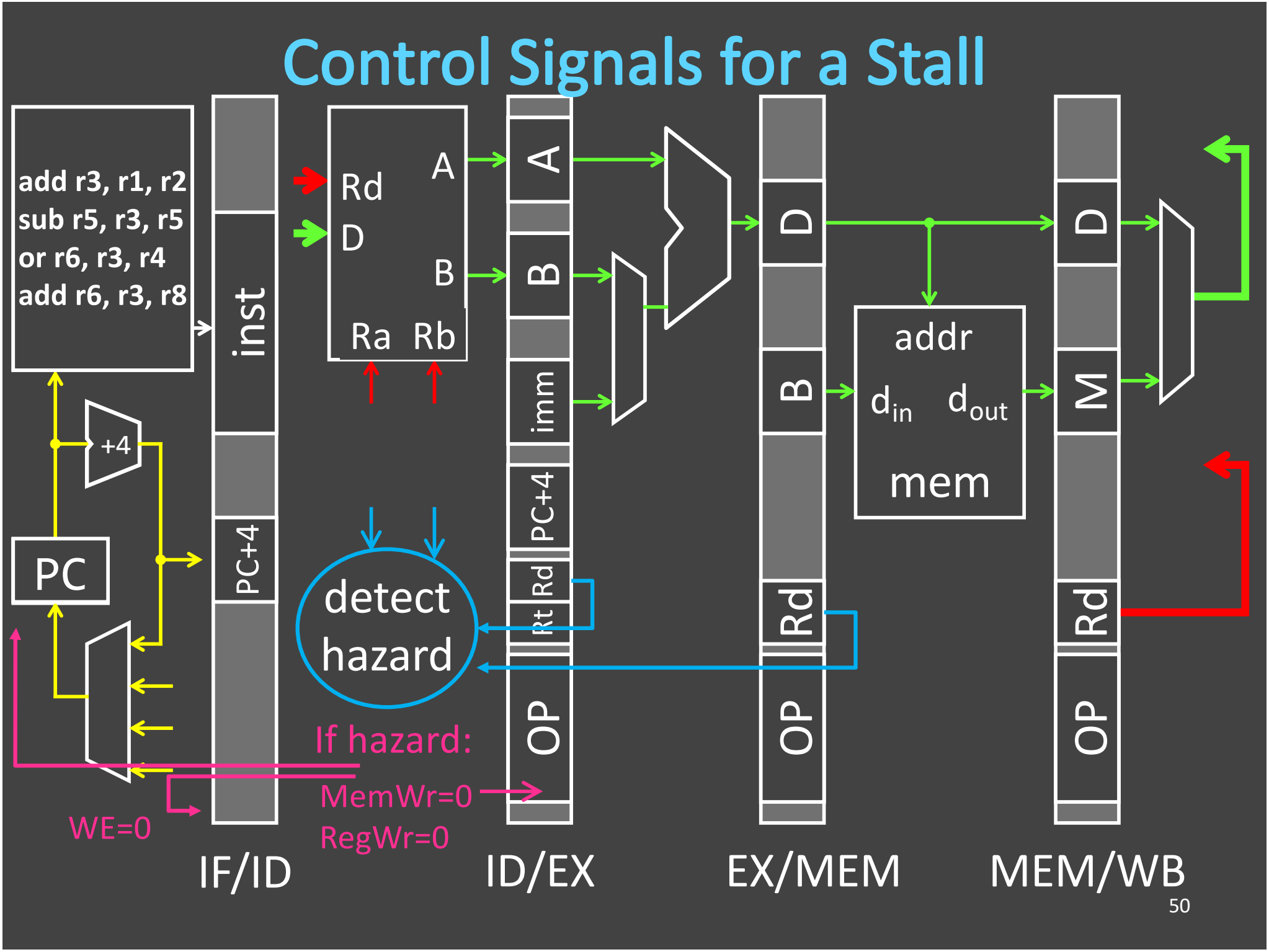
     *(Only works if value actually exists already)*

# Stalling

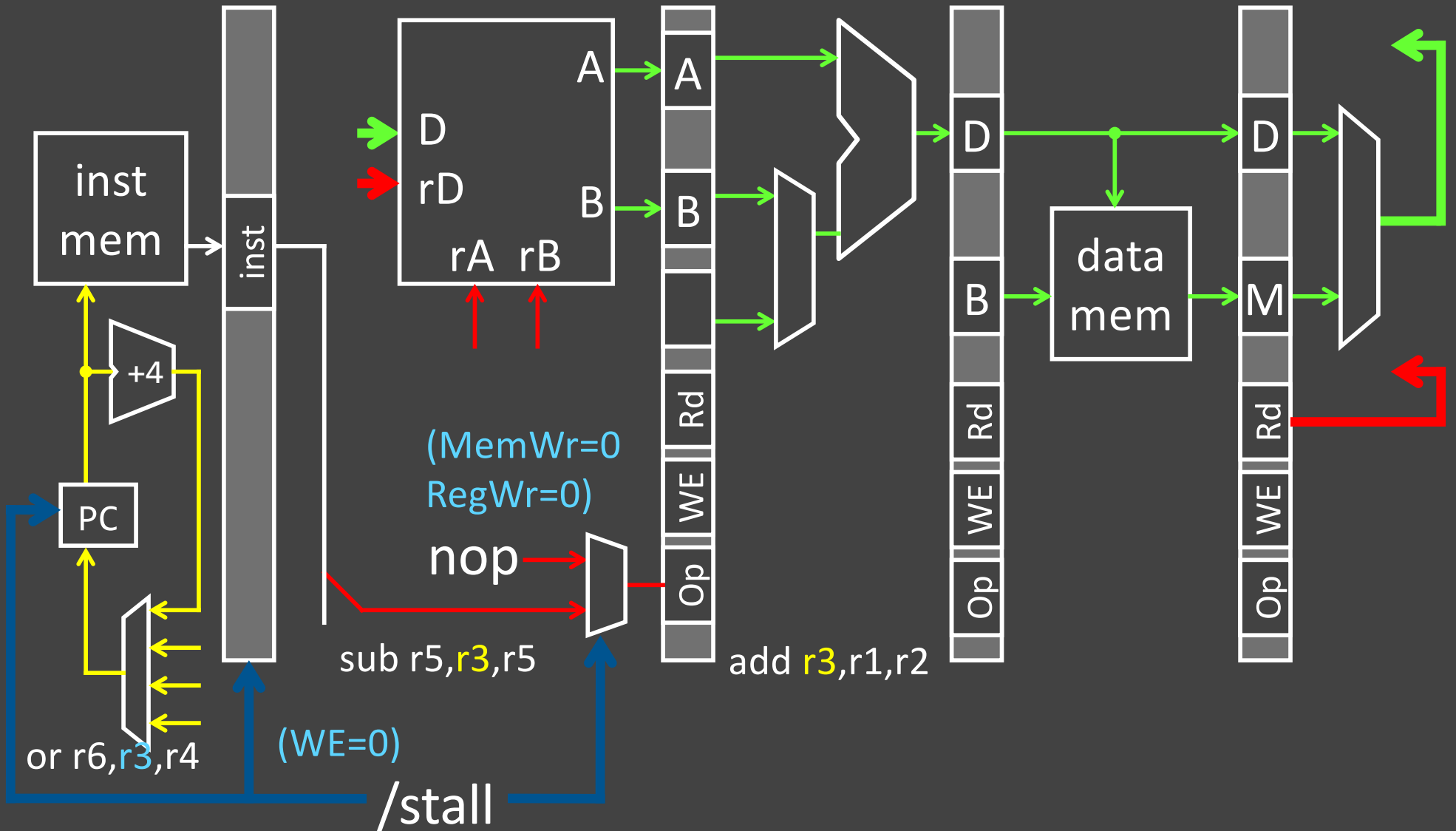
How to stall an instruction in ID stage

- 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

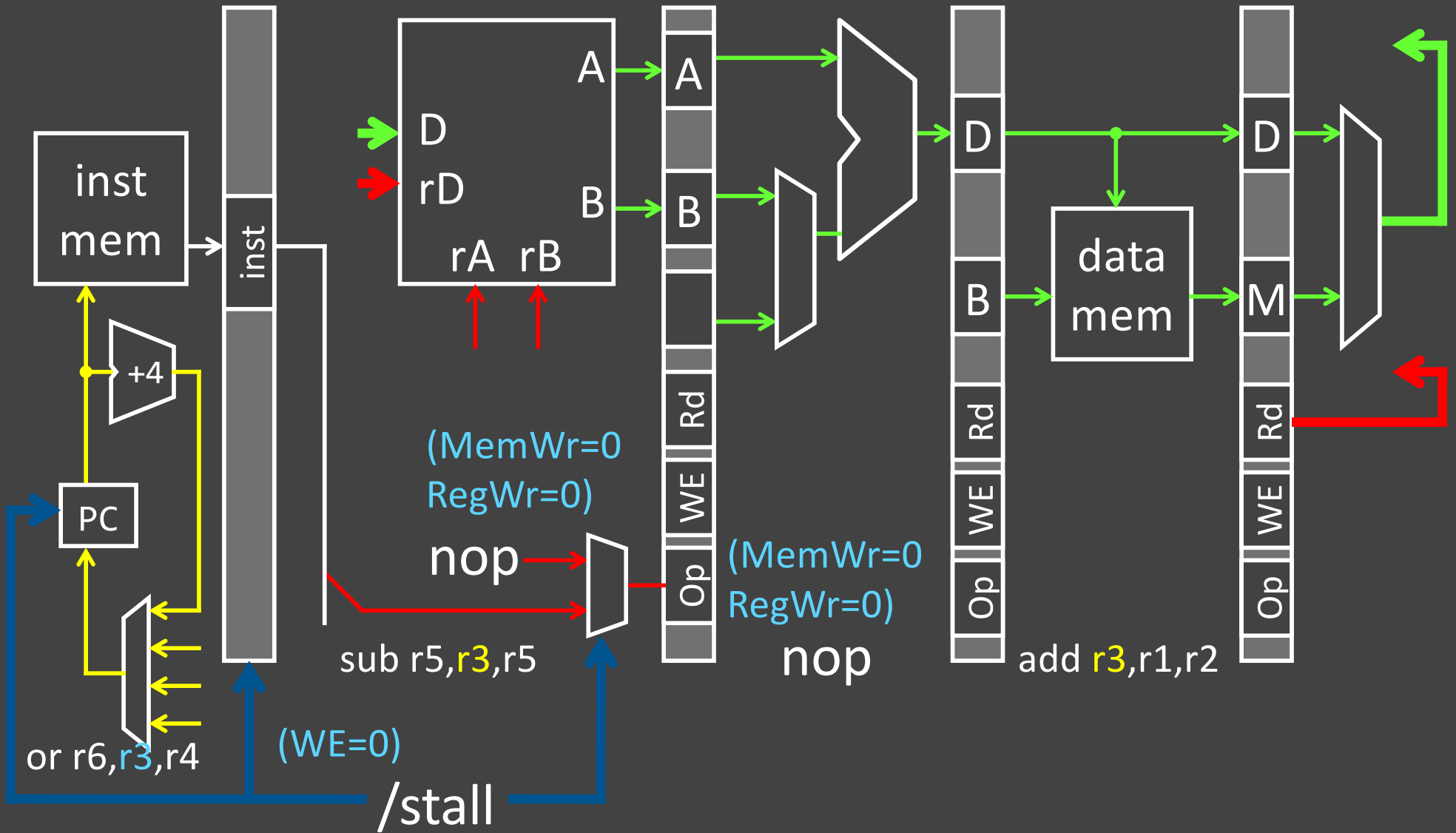


# Detecting the Hazard



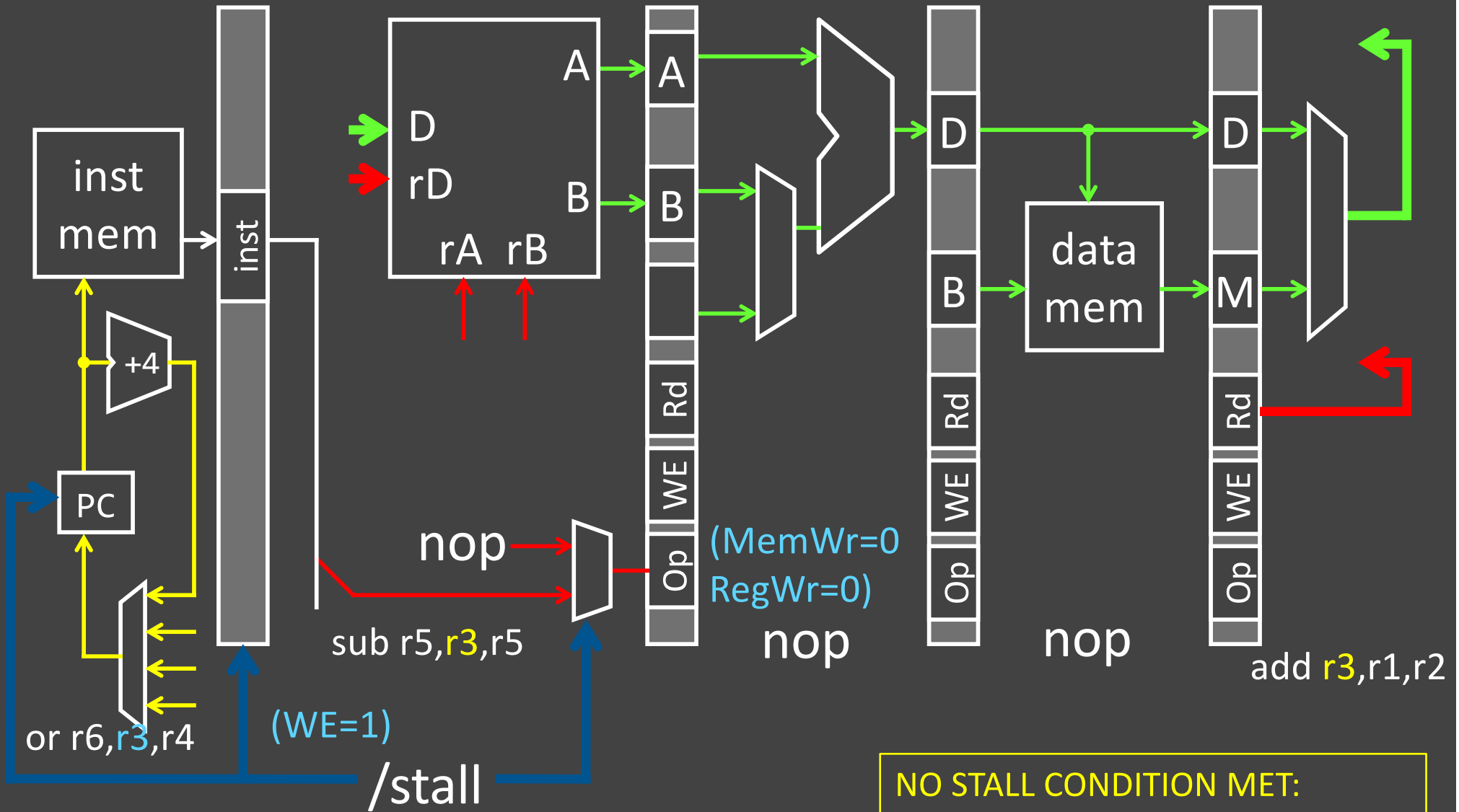

NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd ← STALL CONDITION MET
IF/ID.rA==Ex/M.Rd))

# 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

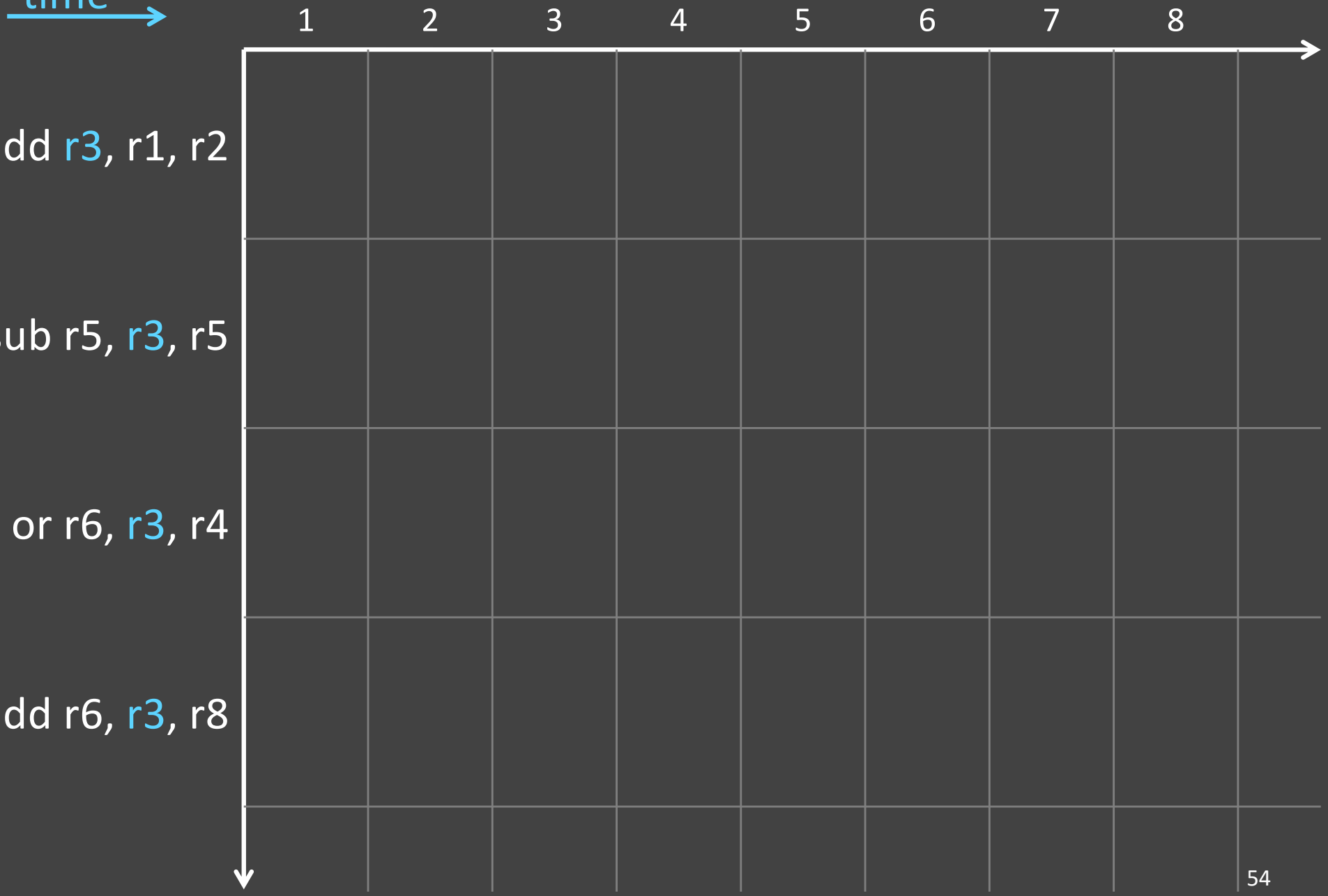
# Second Stall Cycle (nop in X, MEM)



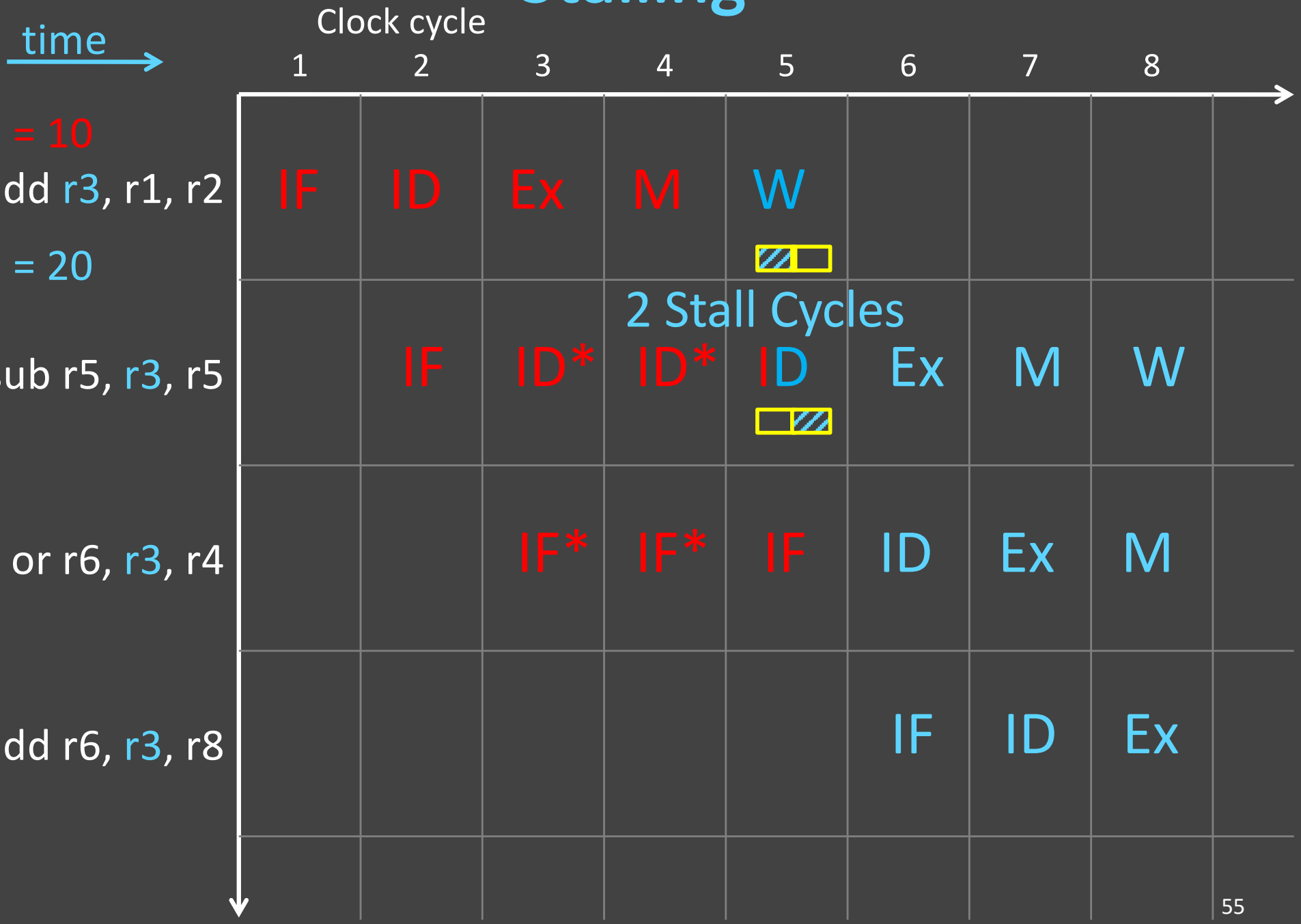

NOP = If(IF/ID.rA ≠ 0 &&
(IF/ID.rA==ID/Ex.Rd
IF/ID.rA==Ex/M.Rd))

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

# Stalling


time
Clock cycle
1
2
3
4
5
6
7
8
r3 = 10
add r3, r1, r2
IF
ID
Ex
M
W
r3 = 20
2 Stall Cycles
sub r5, r3, r5
IF
ID*
ID*
ID
Ex
M
W
or r6, r3, r4
IF*
IF*
IF
ID
Ex
M
add r6, r3, r8
IF
ID
Ex

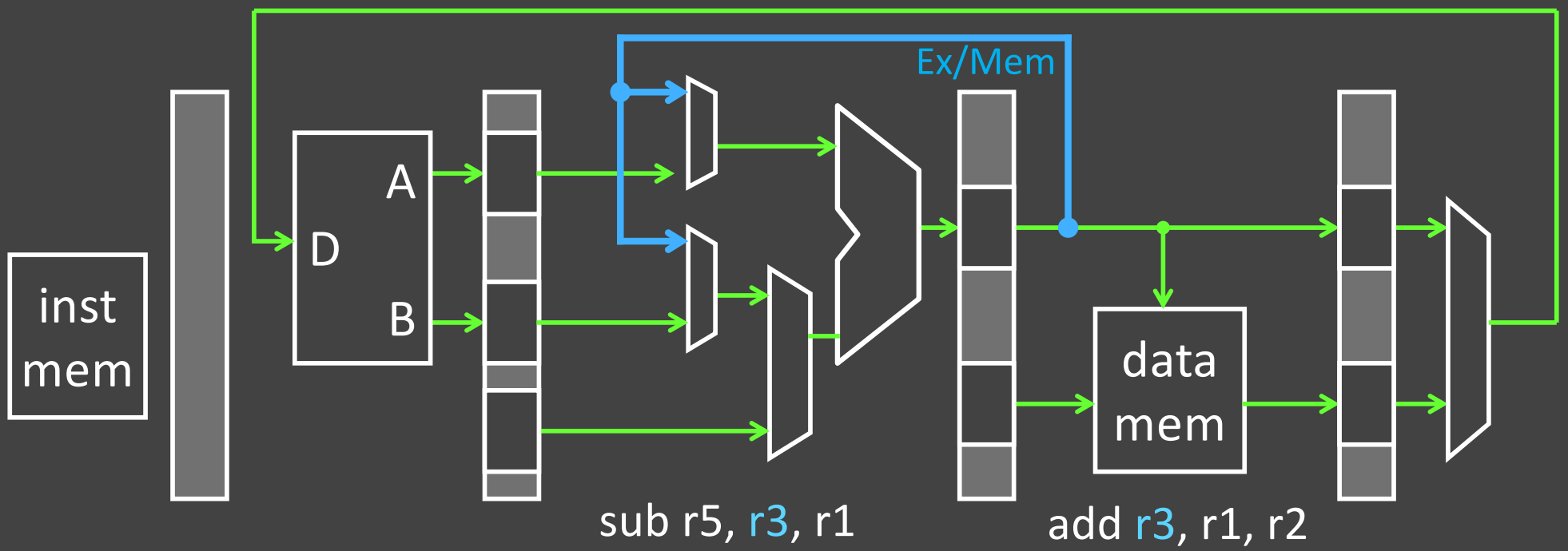
# Possible Responses to Data Hazards

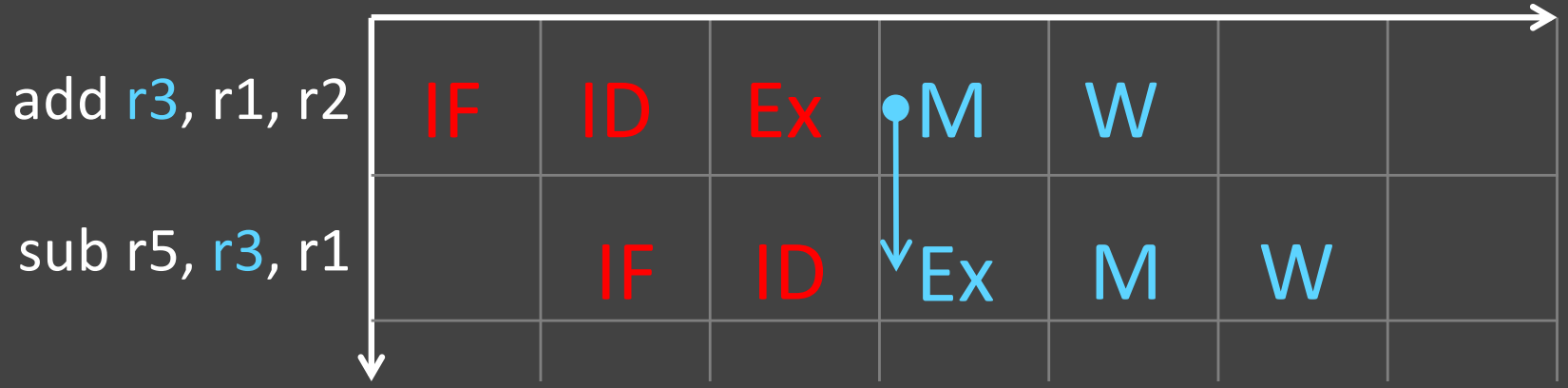
1. Do Nothing
   - Change the ISA to match implementation
   - "Compiler: don't create code with data hazards!"

     (*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 1: MEM → EX

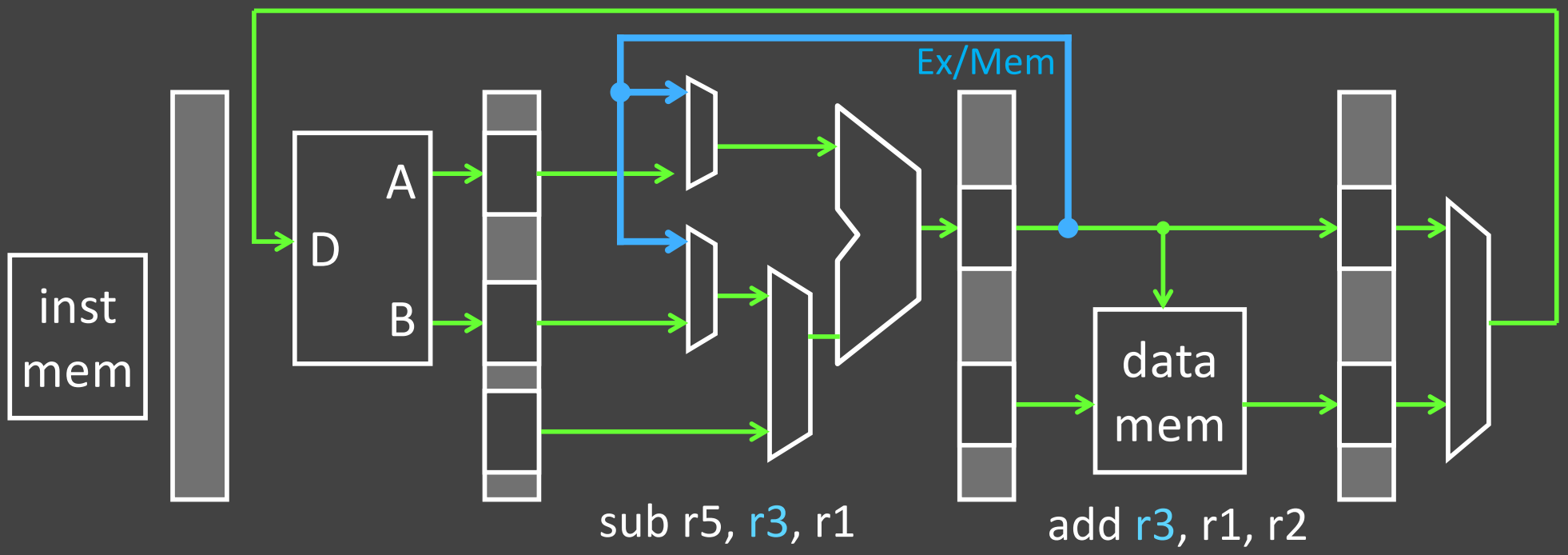



Problem: EX needs ALU result that is in MEM stage

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

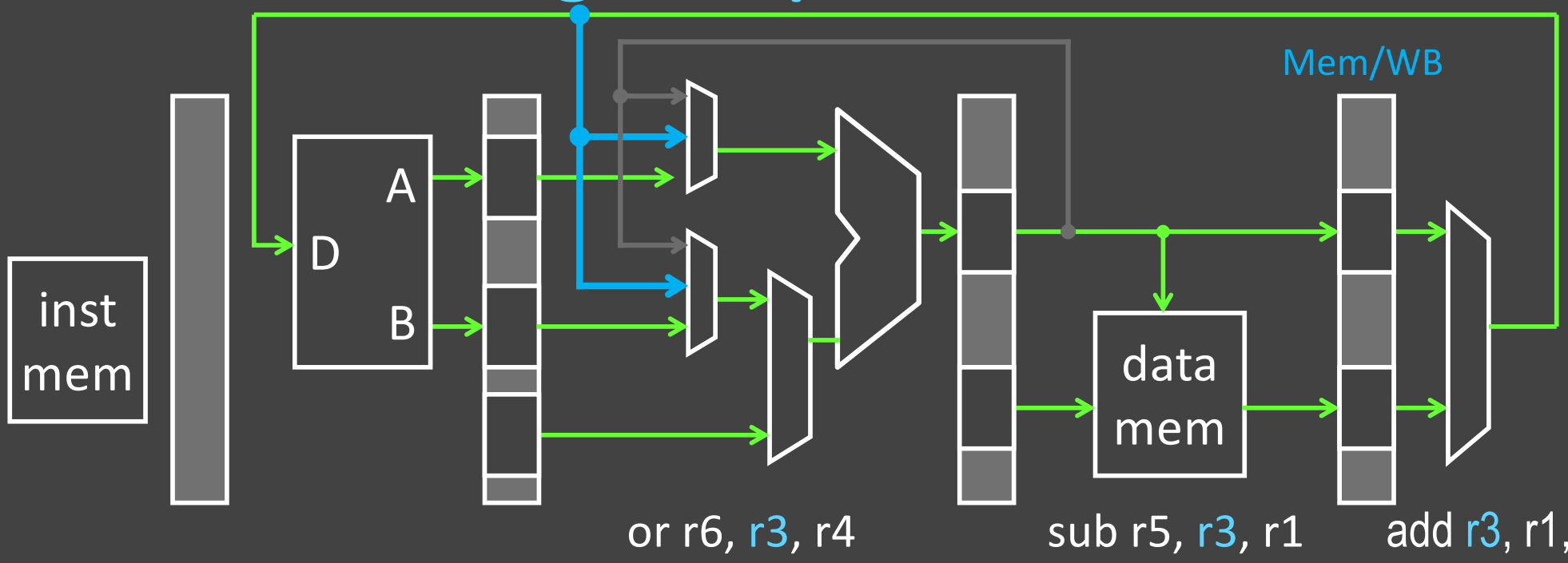
# Forwarding Datapath 1: MEM → EX

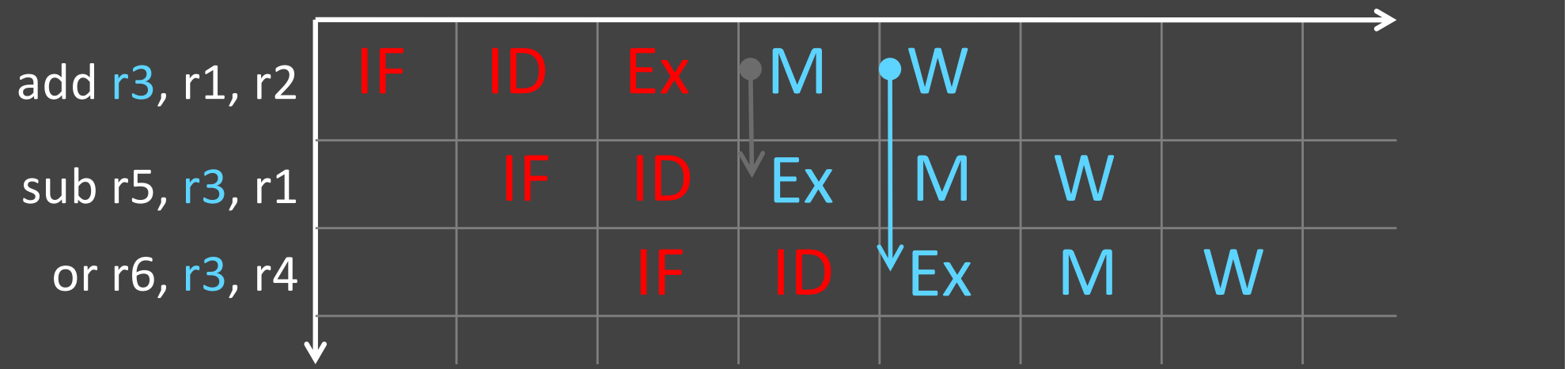



Detection Logic in Ex Stage:

forward = (Ex/M.WE && EX/M.Rd != 0 &&
ID/Ex.Ra == Ex/M.Rd)
|| (same for Rb)

# Forwarding Datapath 2: WB → EX

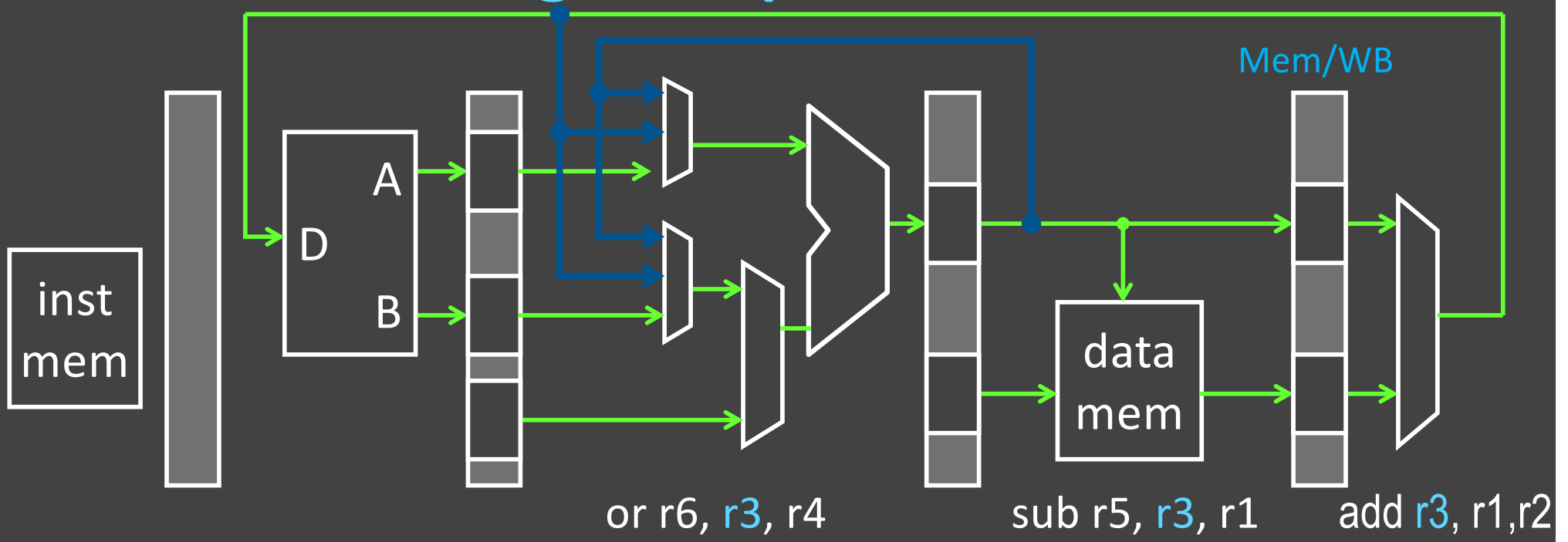



Problem: EX needs value being written by WB

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

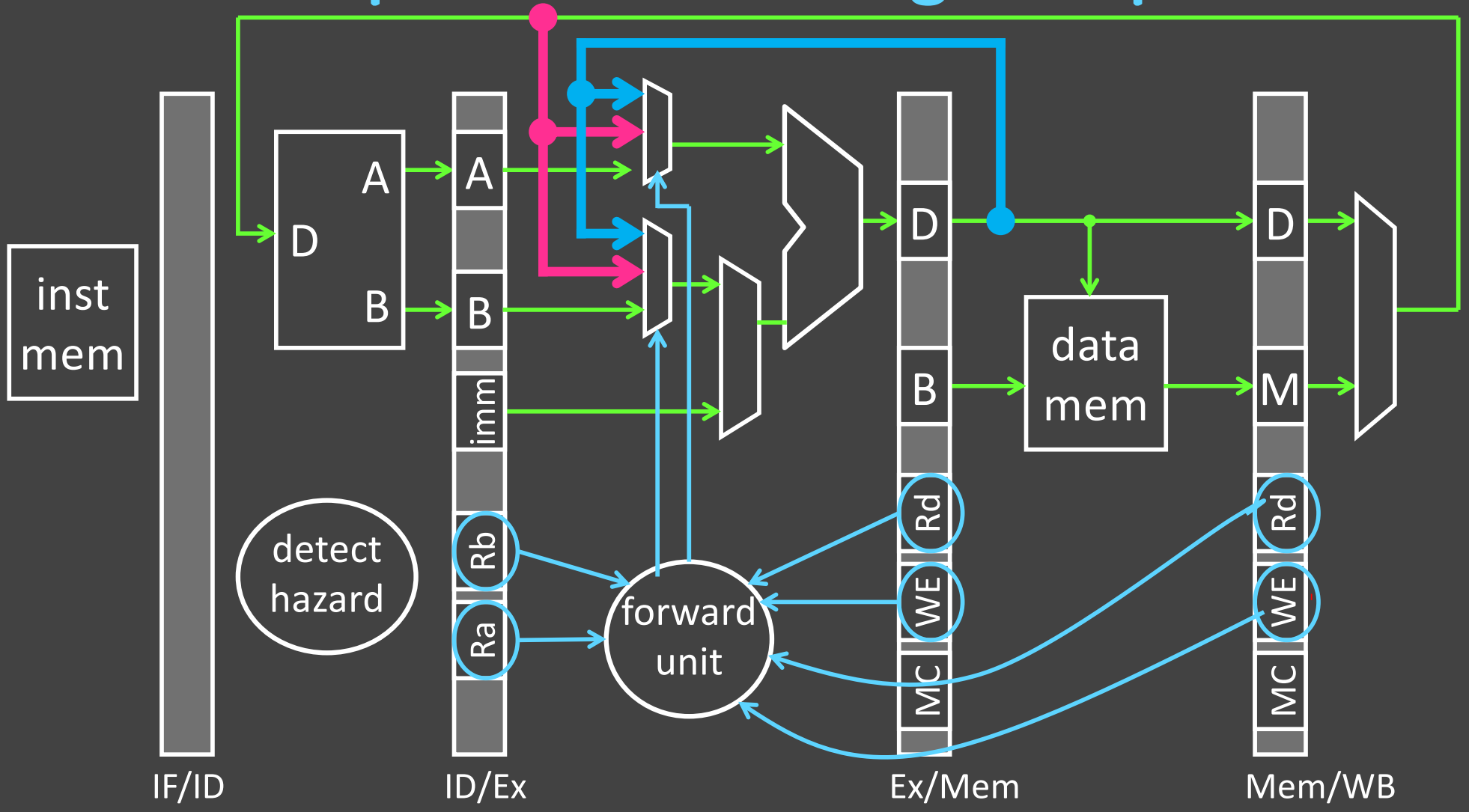
# Forwarding Datapath 2: WB → EX




Detection Logic:

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

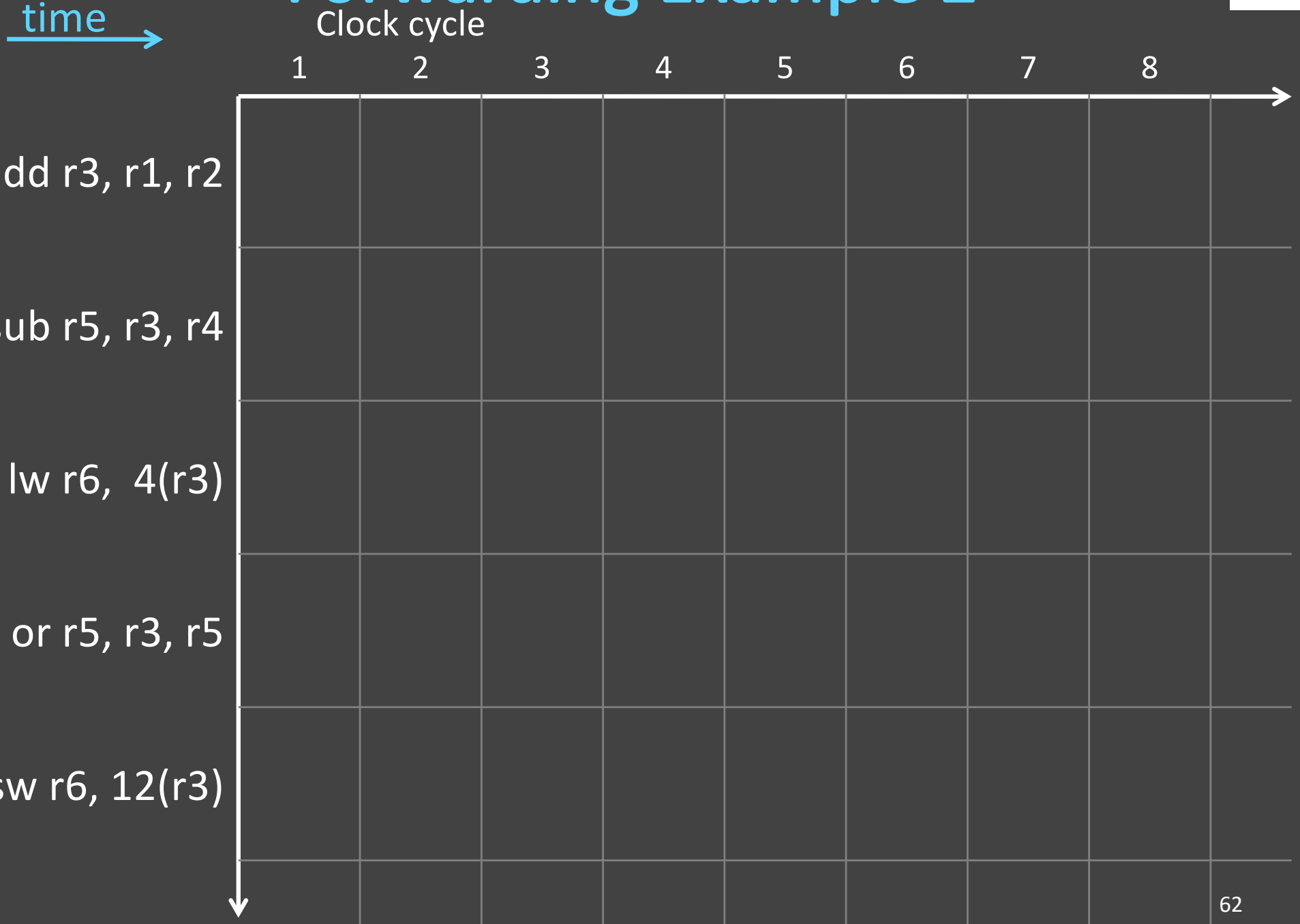
# Complete Forwarding Datapath



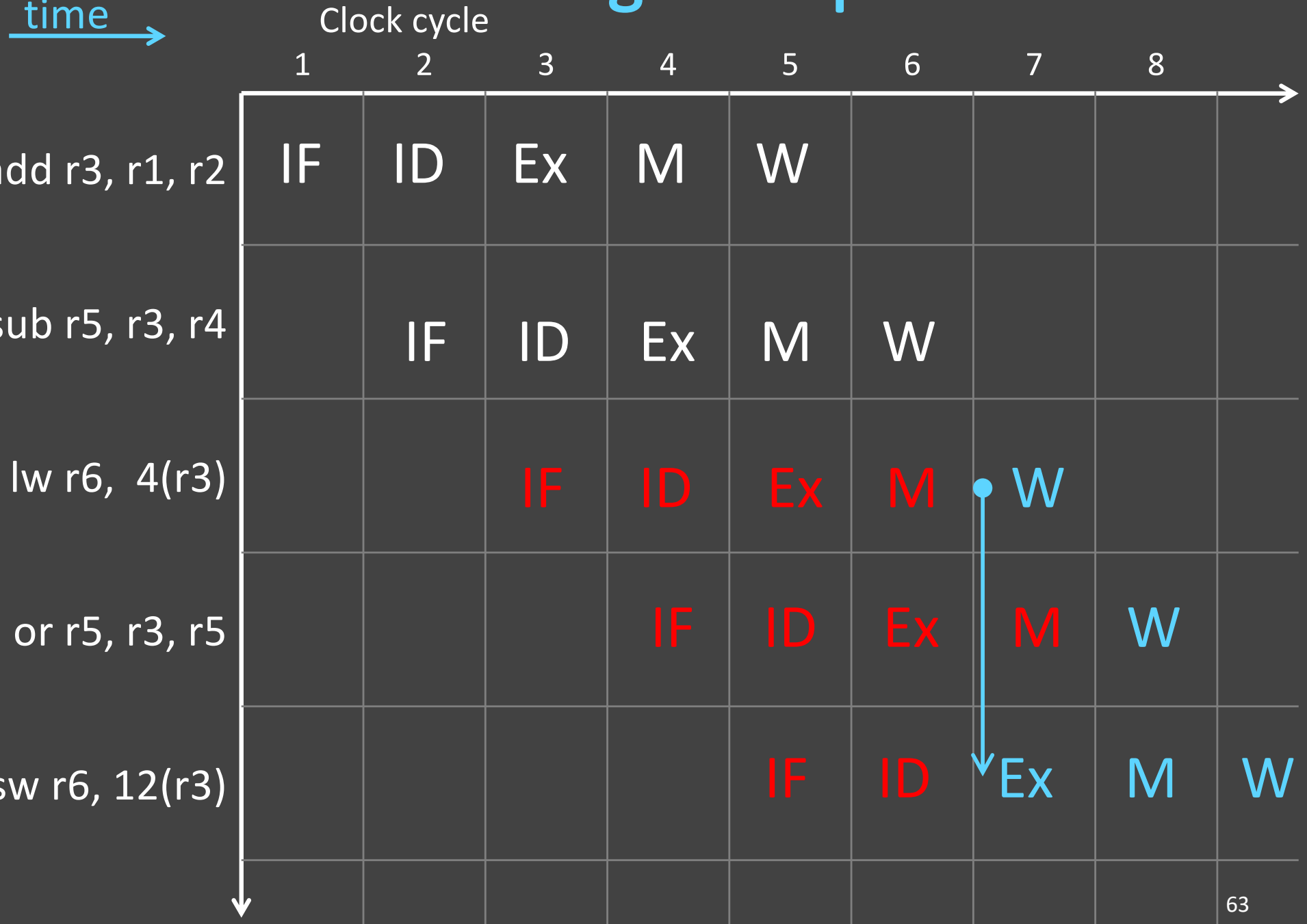

Two types of forwarding/bypass

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

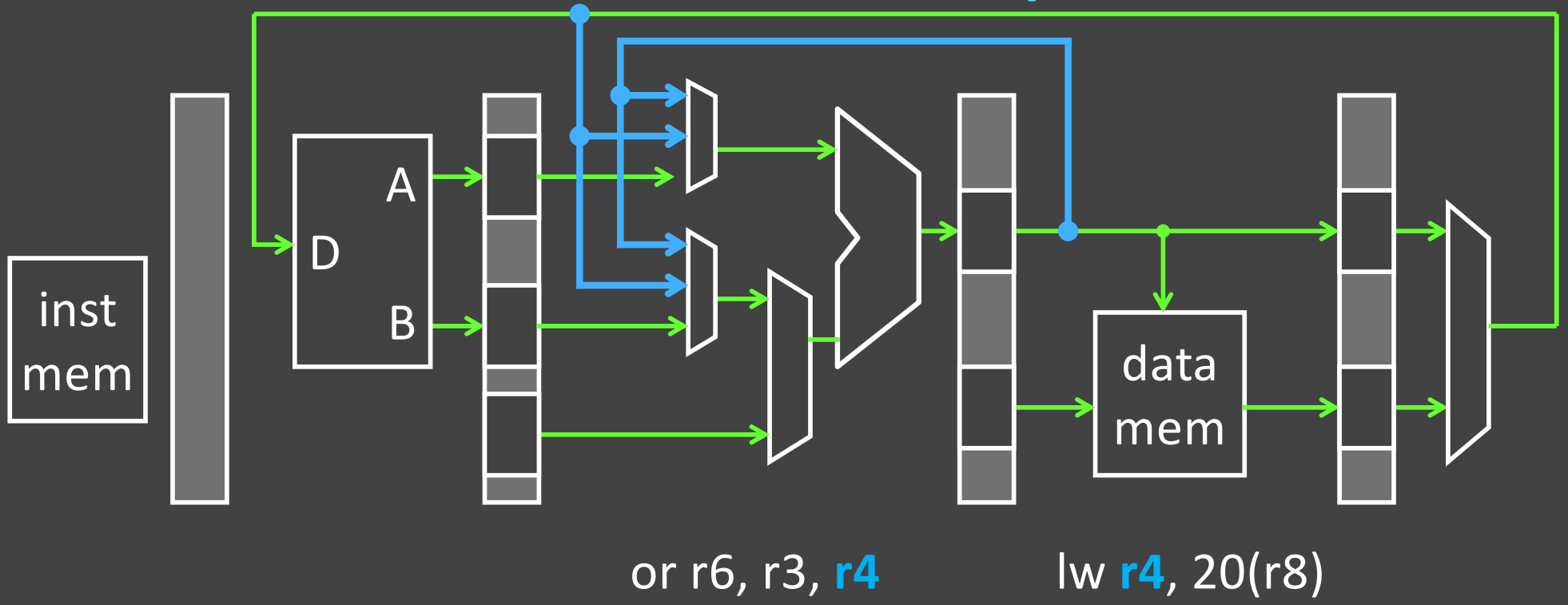
# Forwarding Example 2



# Forwarding Example 2



# Load-Use Hazard Explained



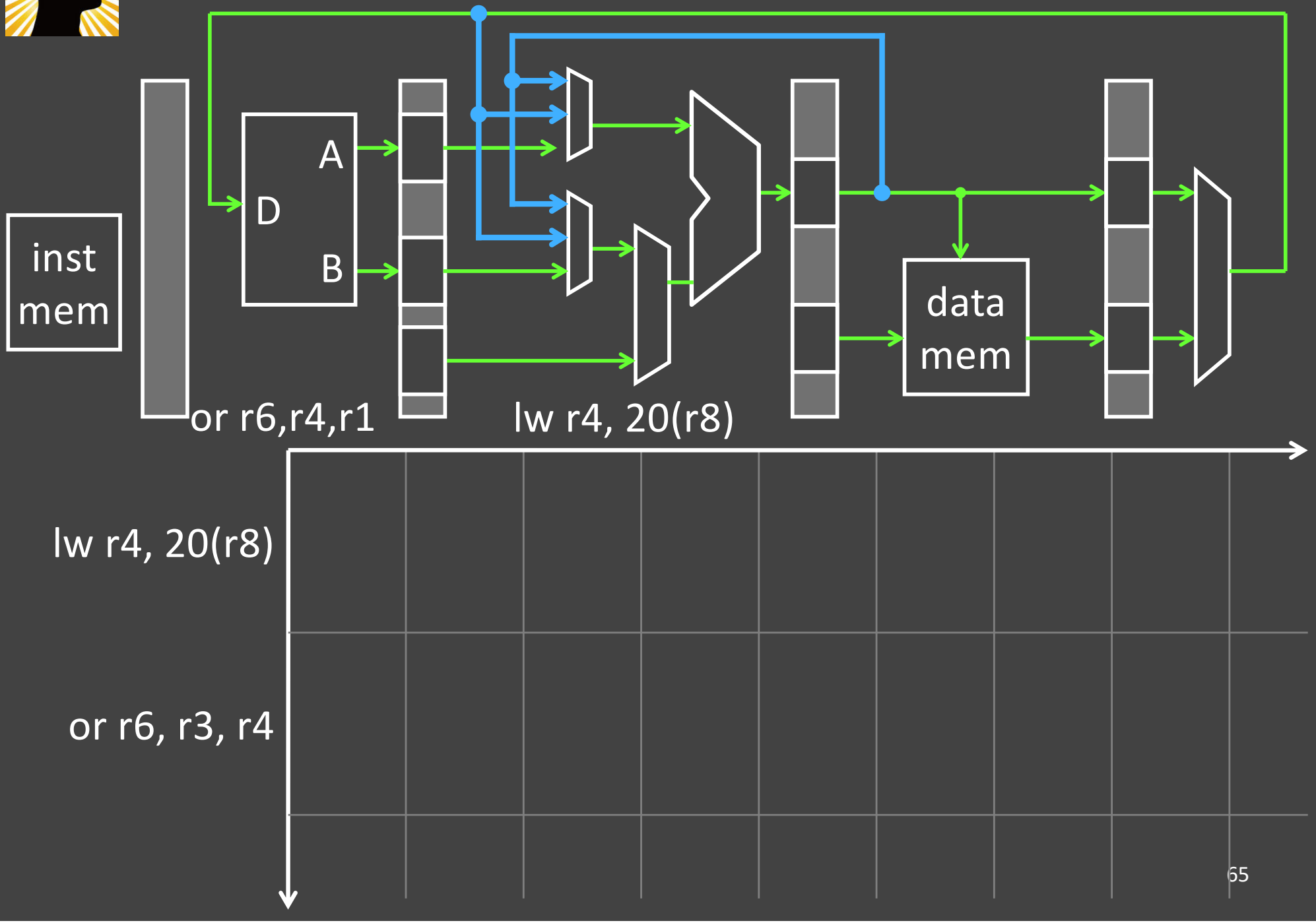

Data dependency after a load instruction:

- Value not available until *after* the M stage

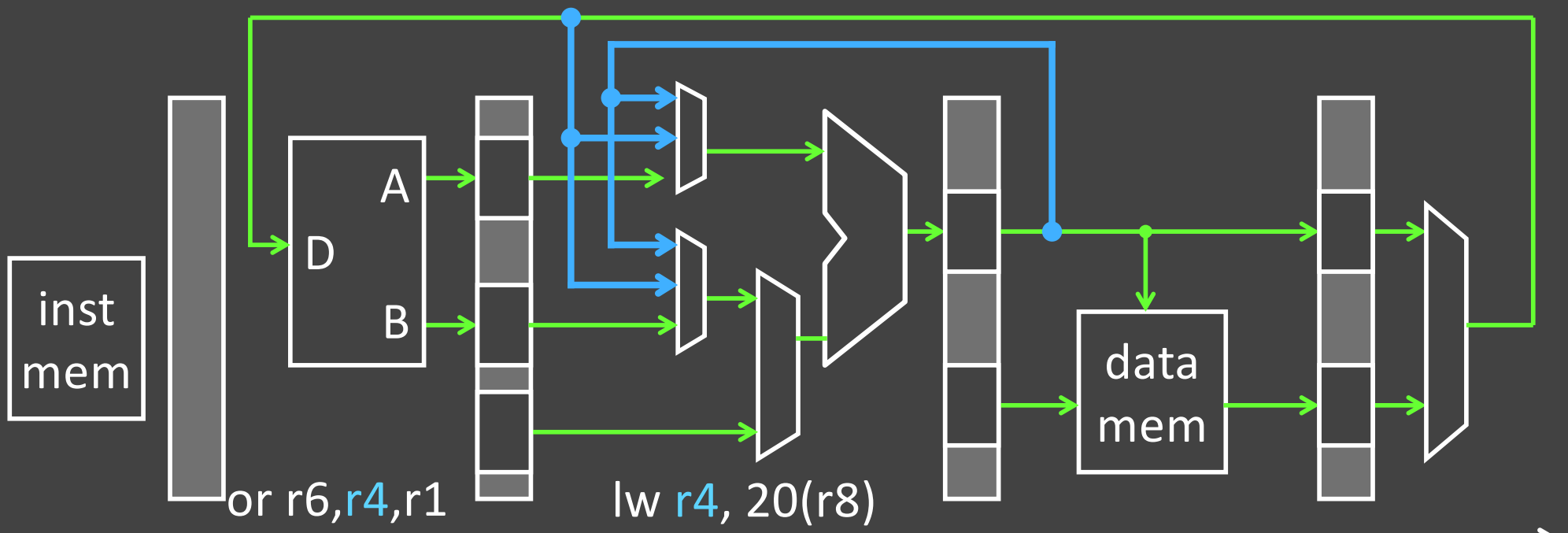
→Next instruction cannot proceed if dependent

*THE KILLER HAZARD*

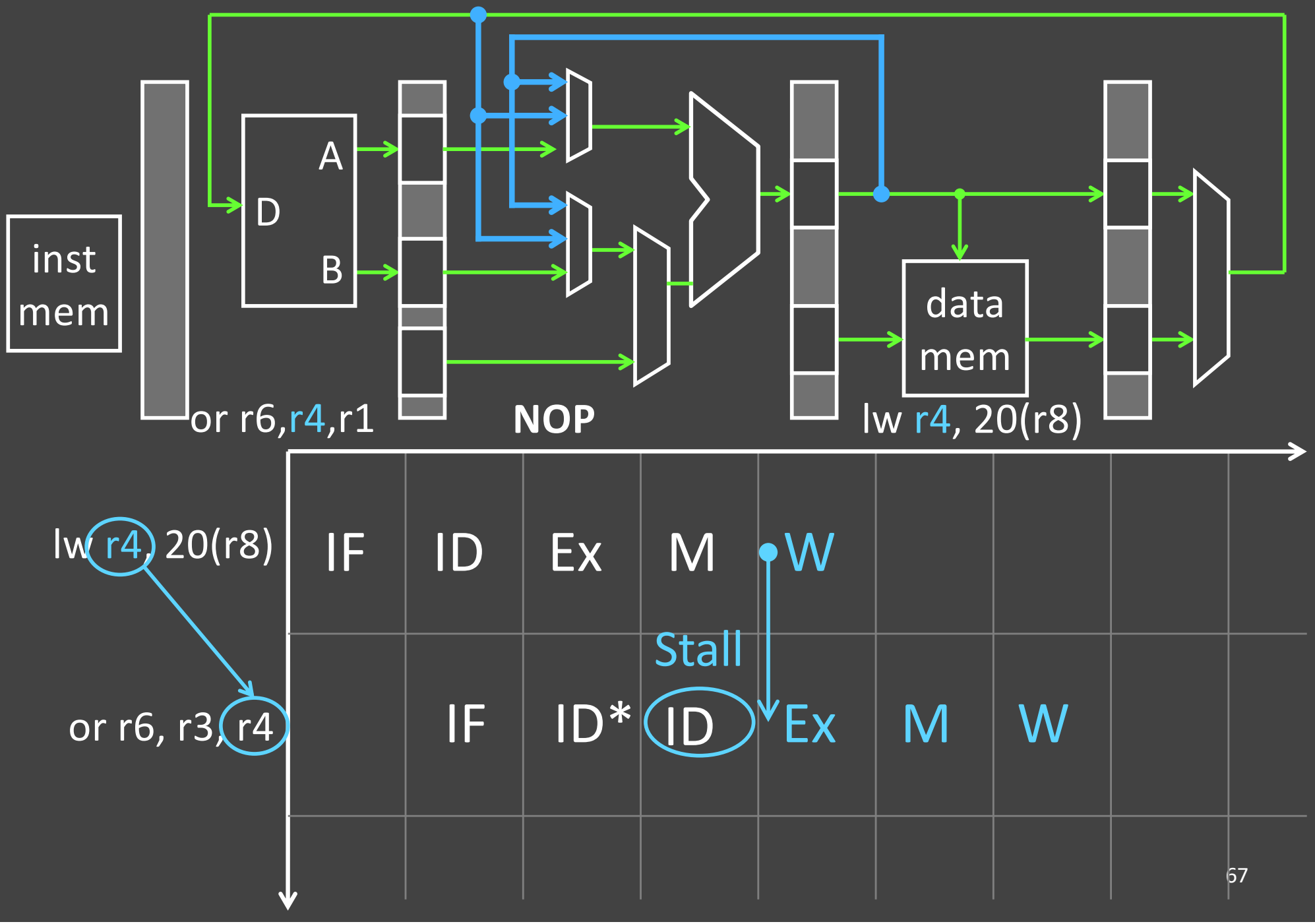
# Load-Use Stall



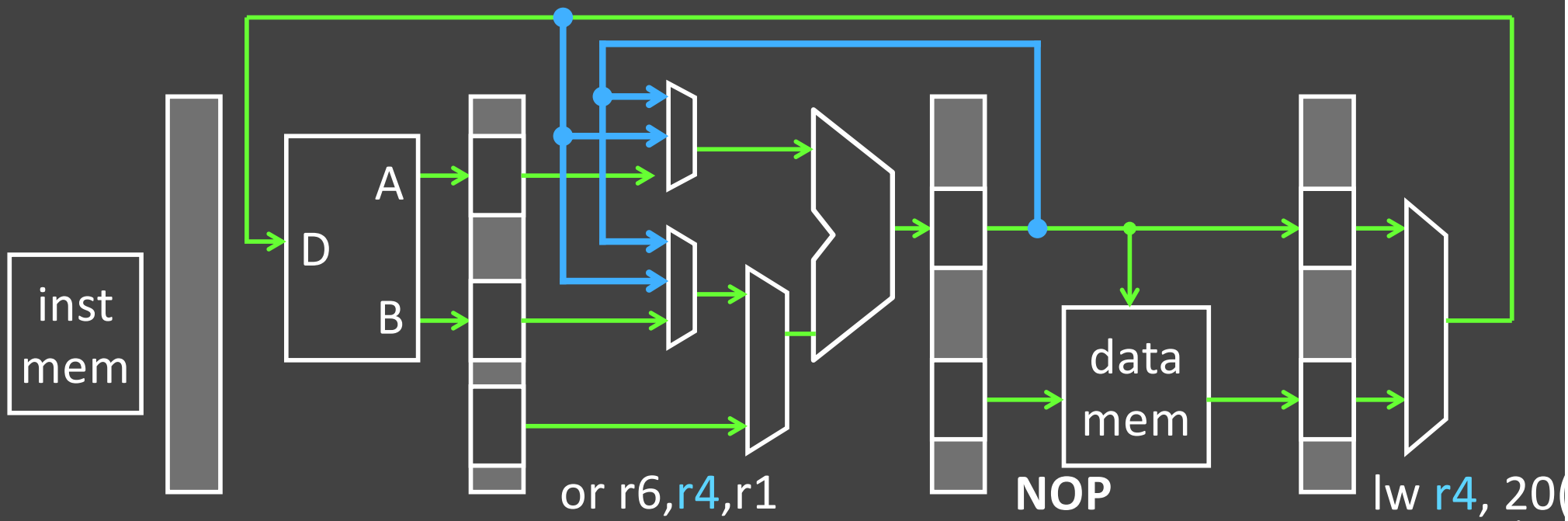
# Load-Use Stall (1)



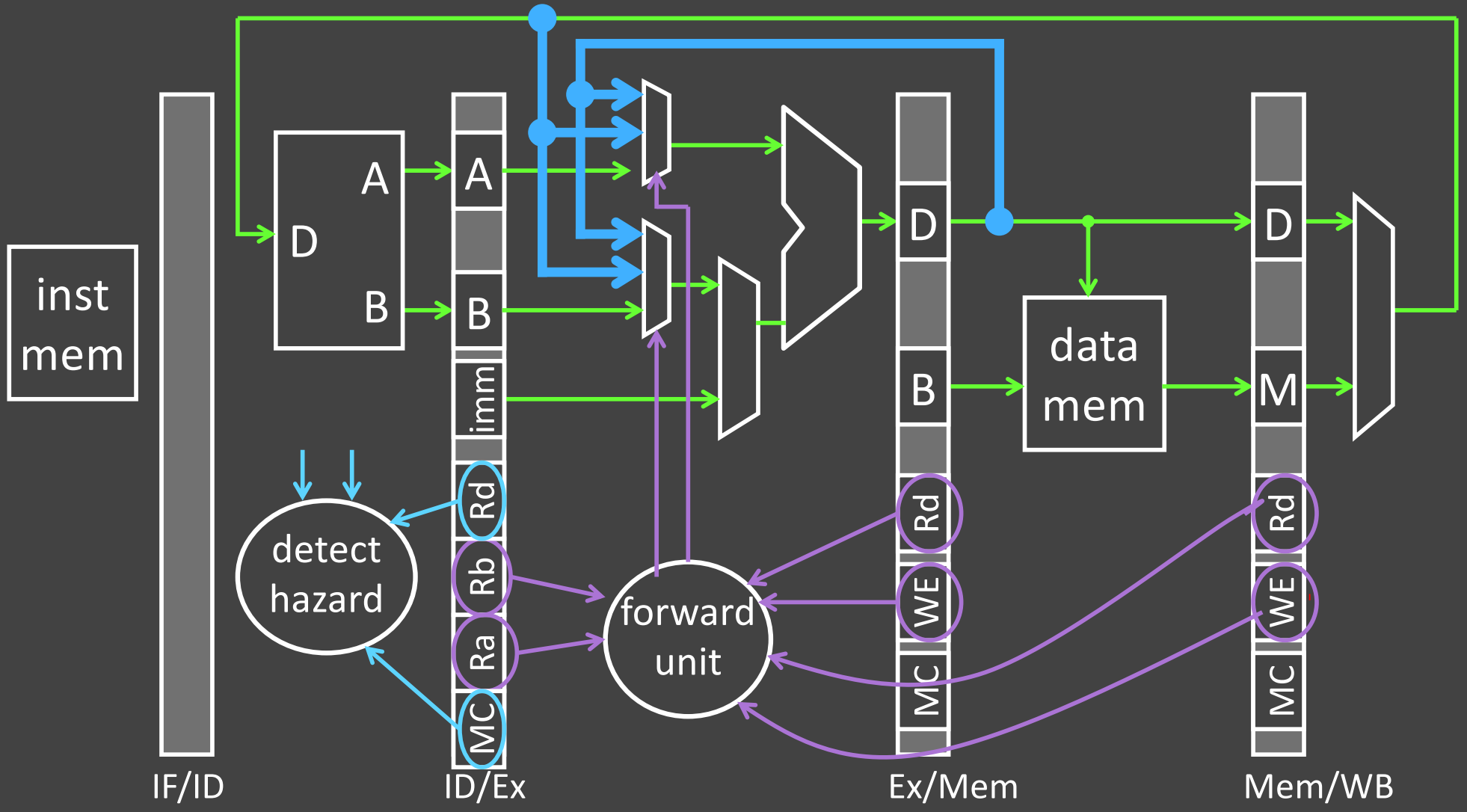

# Load-Use Stall (2)



# Load-Use Stall (3)

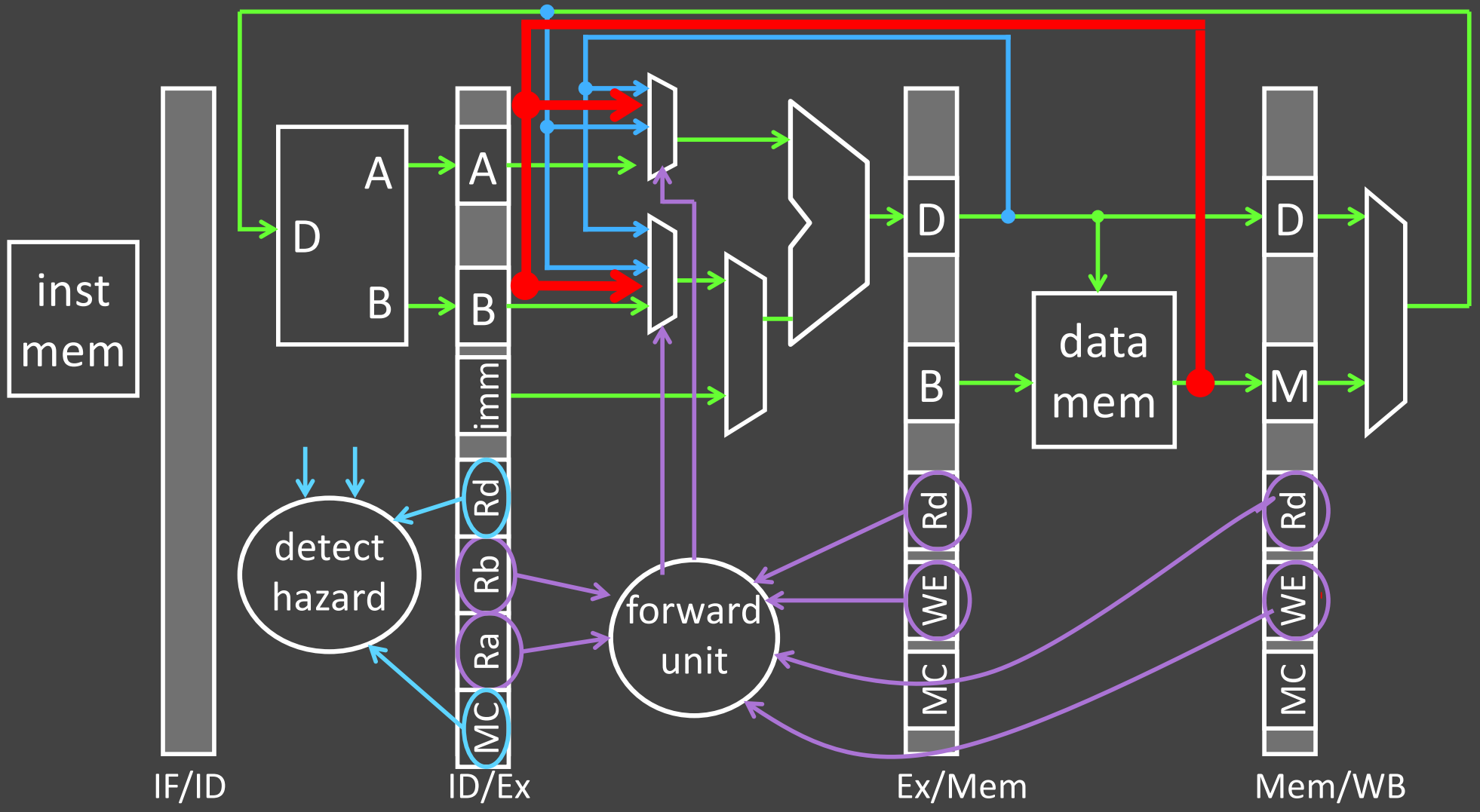



# Load-Use Detection




Stall = If(ID/Ex.MemRead &&
IF/ID.Ra == ID/Ex.Rd

# Incorrectly Resolving Load-Use Hazards




Most frequent 3410 **non-solution** to load-use hazards

**Why is this "solution" so so so so so so awful?**

# iClicker Question

Forwarding values directly from Memory to the Execute stage without storing them in a register first:

A. Does not remove the need to stall.
B. Adds one too many possible inputs to the ALU.
C. Will cause the pipeline register to have the wrong value.
D. Halves the frequency of the processor.
E. Both A & D

# Resolving Load-Use Hazards

Two MIPS Solutions:

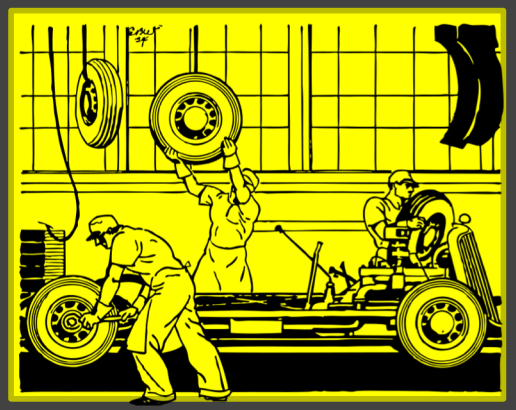
- MIPS 2000/3000: delay slot
  - ISA says results of loads are not available until one cycle later
  - Assembler inserts nop, or reorders to fill delay slot

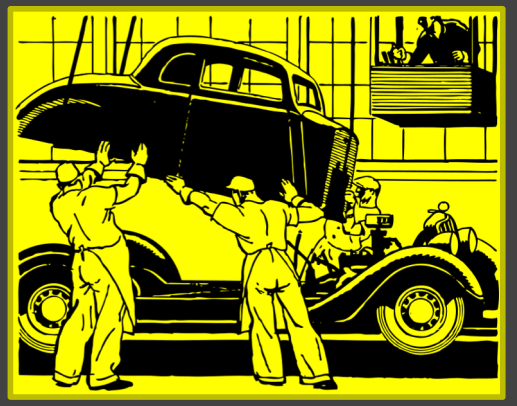
- MIPS 4000 onwards: stall
  - But really, programmer/compiler reorders to avoid stalling in the load delay slot

# Agenda

5-stage Pipeline

- Implementation
- Working Example





Hazards

- Structural
- Data Hazards
- Control Hazards

# A bit of Context

```
for (i = 0; i < max; i++) {
    n += 2;
}
i = 7;
n--;
```

```
r1: i
r2: n
r3: max
```

*Simplification: assume max > 0*

```
x10         addi r1, r0, 0      # i=0
x14 Loop:   addi r2, r2, 2      # n += 2
x18         addi r1, r1, 1      # i++
x1C         blt  r1, r3, Loop   # i<max?
x20         addi r1, r0, 7      # i = 7
x24         subi r2, r2, 1      # n--
```

# 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

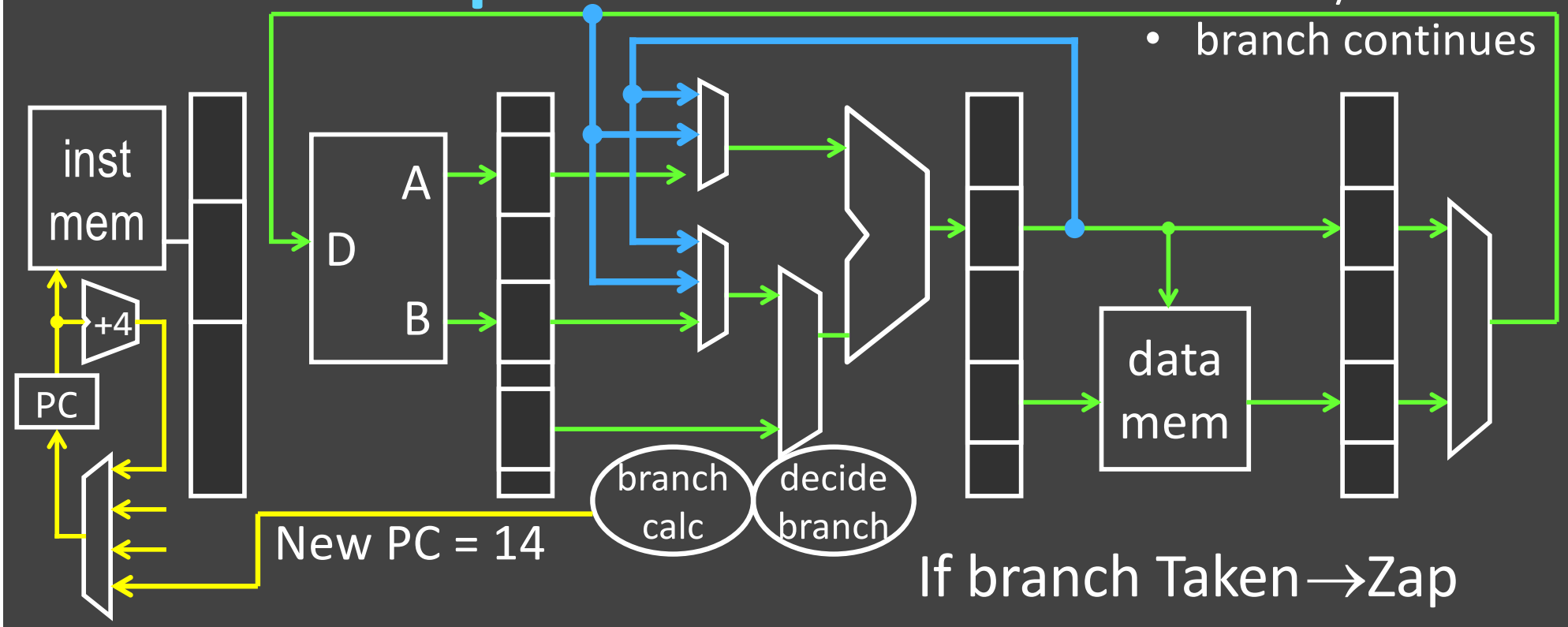
```
x1C   blt  r1, r3, Loop
x20   addi r1, r0, 7
x24   subi r2, r2, 1
```

Branch ***not*** taken?
No Problem!

Branch taken?
Just fetched 2 addi's
→ **Zap & Flush**

# Zap & Flush

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



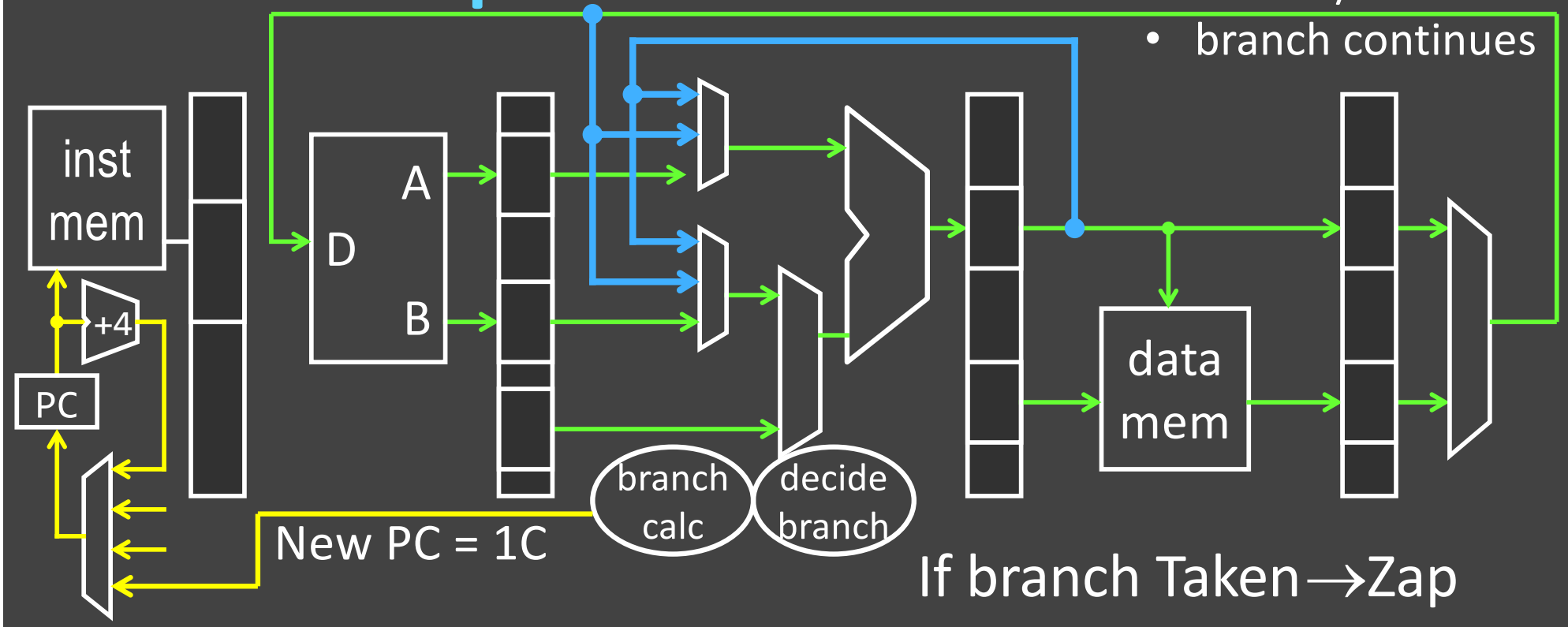

If branch Taken→Zap

```
1C blt r1,r3,L
20 addi r1,r0,7
24 subi r2,r2,1
14 L:addi r2,r2,2
```

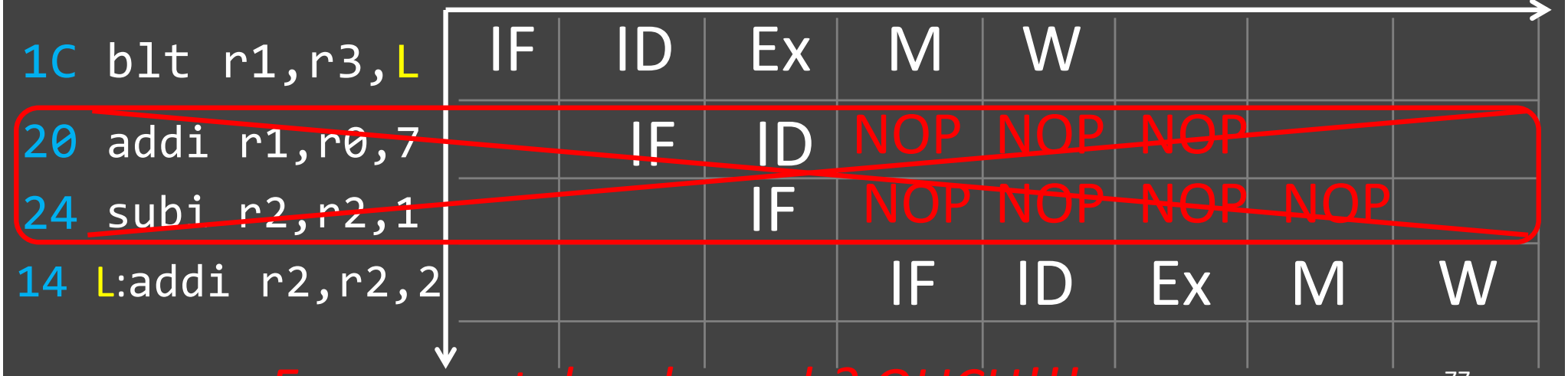
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| IF | ID | Ex | M | W | | | |
| | IF | ID | NOP | NOP | NOP | | |
| | | IF | NOP | NOP | NOP | NOP | |
| | | | IF | ID | Ex | M | W |

# Zap & Flush

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




If branch Taken→Zap




*For every taken branch? OUCH!!!*

# Branch Performance

Back of the envelope calculation

- **Branch: 20%,** load: 20%, store: 10%, other: 50%
- Say, **75% of branches are taken**

CPI = 1 + 20% * 75% * 2 =

1 + **0.20 * 0.75 * 2** = 1.3

- **Branches cause 30% slowdown**
  - Even worse with deeper pipelines

How do we reduce slowdown?

# Reducing the cost of control hazard

1. Delay Slot
   - MIPS ISA: 1 insn after ctrl insn ***always*** executed
     - Whether branch taken or not
   - Your MIPS assembly should do this
2. Resolve Branch at Decode
   - 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

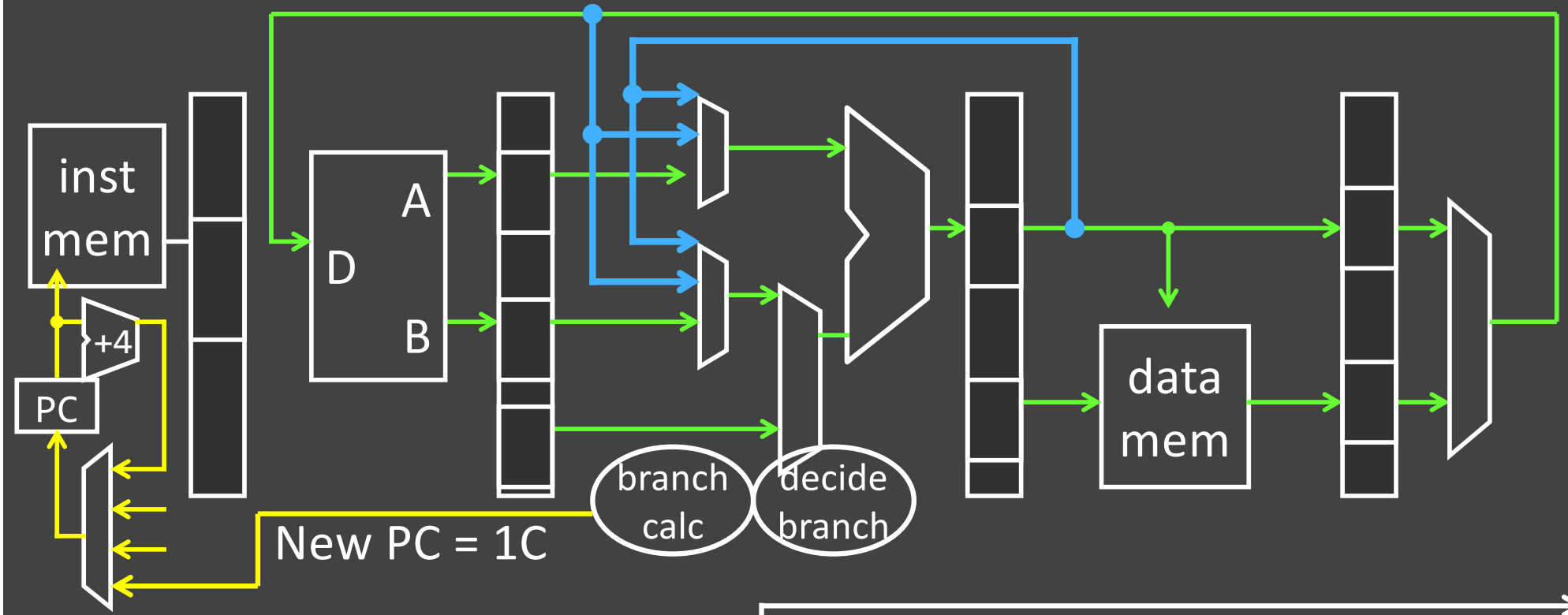
# Solution #1: Delay Slot

```
for (i = 0; i < max; i++) {
    n += 2;
}
i = 7;
n--;
```

*i → r1*
*Assume:*
*n → r2*
*max → r3*

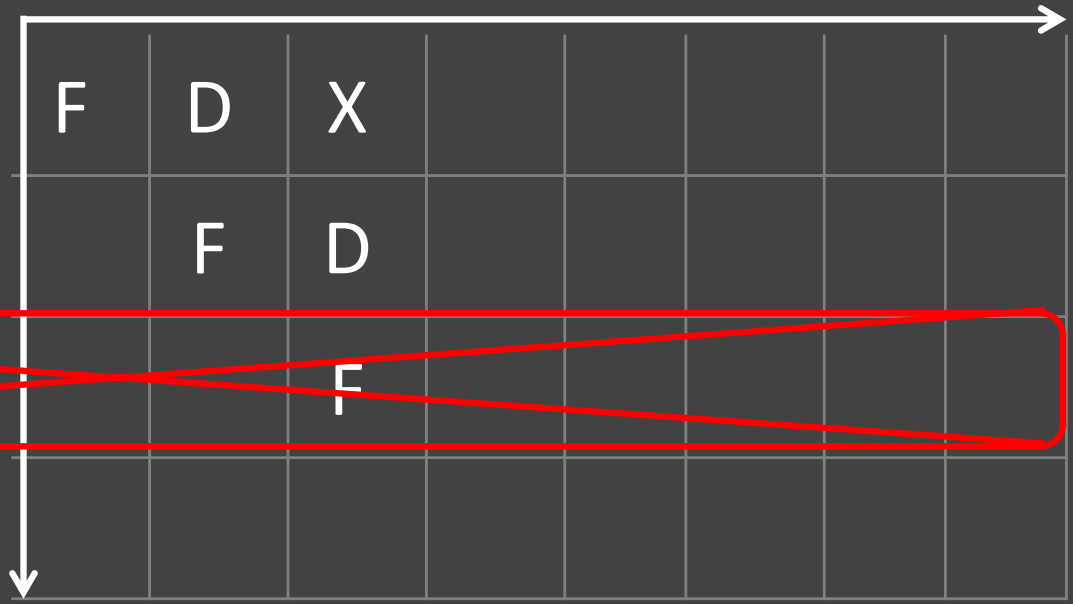
```
x10         addi r1, r0, 0      # i=0
x14 Loop:   addi r2, r2, 2      # n x+= 2
x18         addi r1, r1, 1      # i++
x1C         blt  r1, r3, Loop   # i<max?
x20         nop
x24         addi r1, r0, 7      # i = 7
x28         subi r2, r2, 1      # n++
```

# Delay Slot in Action




```
1C  blt r1, r3, Loop
20  nop
24  addi r1, r0, 7
```

zap!

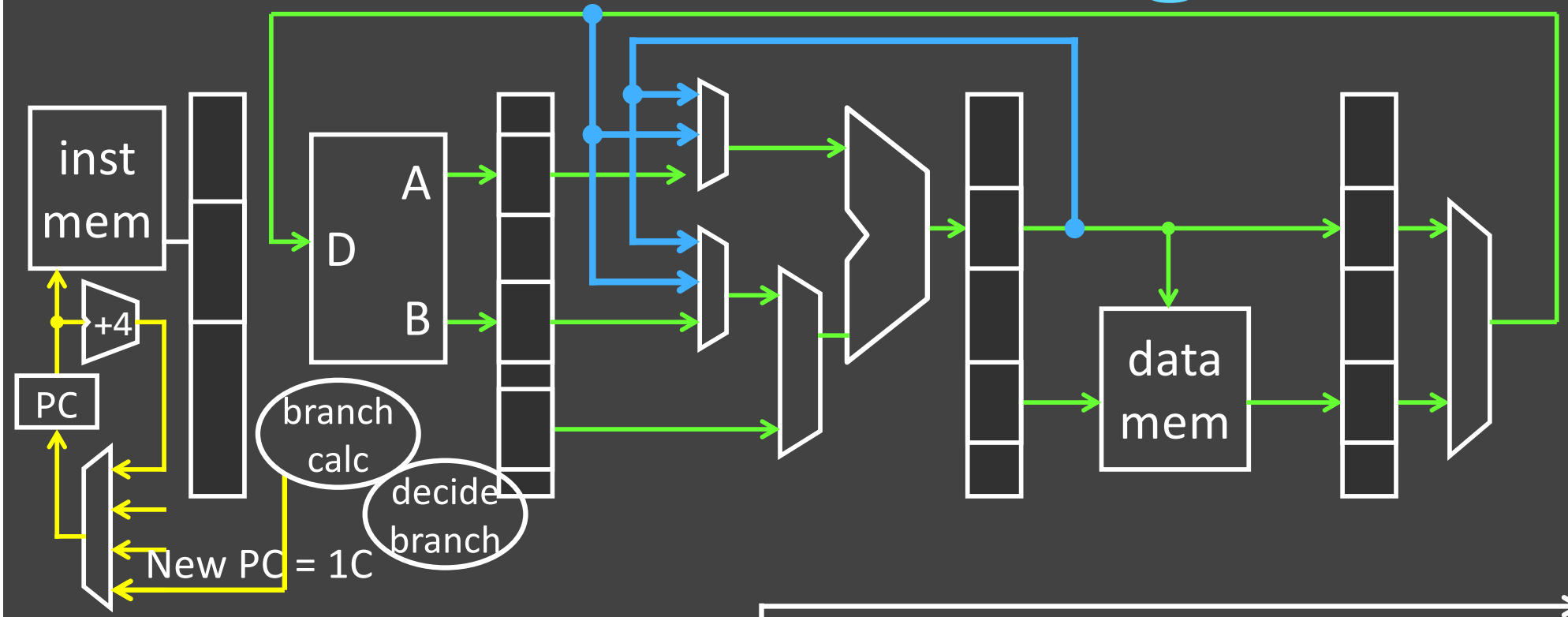


# iClicker Question

A delay slot complicates the design of a processor.

A. True
B. False
C. Cannot tell from the information given
D. I don't know
E. I think E is an awesome answer.

# Soln #2: Resolve Branches @ Decode




```
1C  blt r1, r3, Loop
20  nop
14  Loop:addi r2,r2,2
```

*No Zapping!*

# Branch Performance

Back of the envelope calculation

- **Branch: 20%,** load: 20%, store: 10%, other: 50%
- Say, **75% of branches are taken**

What is the CPI with resolution @ decode?

CPI = 1 + 20% * 75% * 1 =

1 + **0.20 * 0.75 * 1** = 1.15

– **30% slowdown → 15% slowdown**

# iClicker Question

Resolving branches at decode could slow down the clock frequency of the processor.

A. True
B. False
C. Cannot tell from the information given
D. I don’t know
E. I think E is an awesome answer.

# iClicker Question

Because MIPS has a delay slot, the instruction after any control instruction must always be a nop.

A. True
B. False
C. Cannot tell from the information given
D. I don't know
E. I think E is an awesome answer.

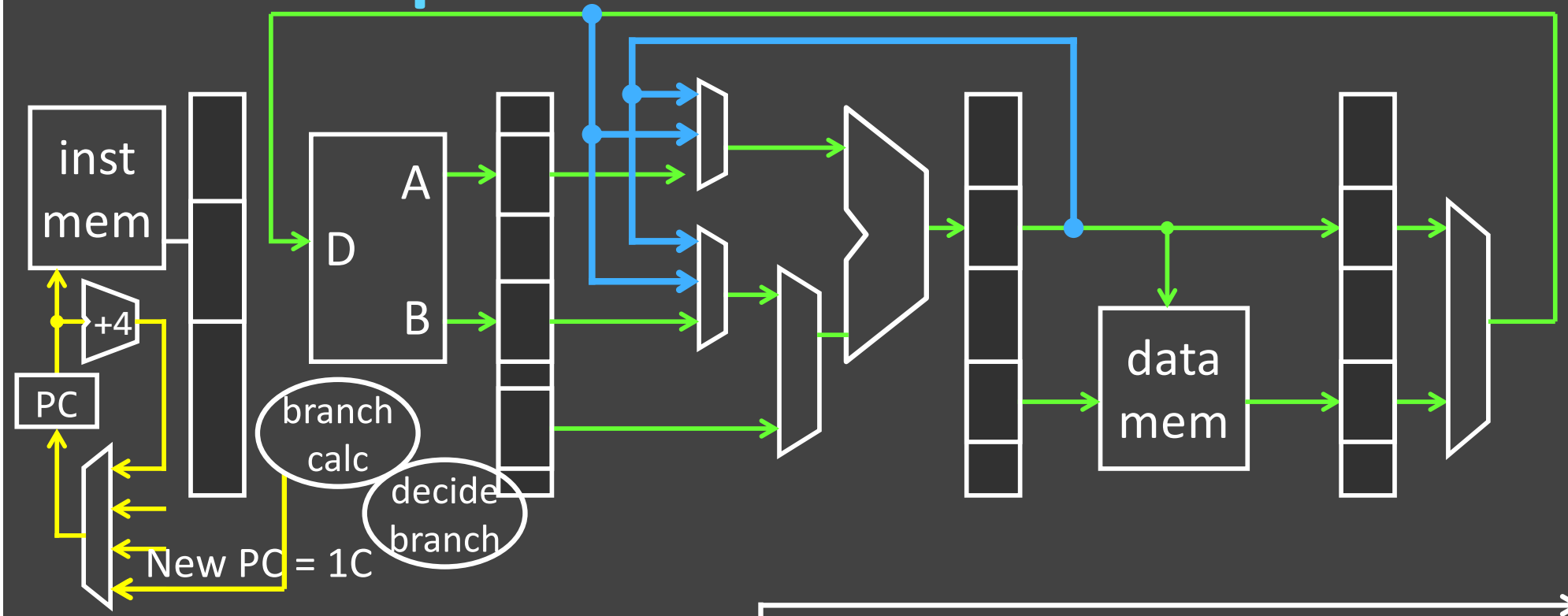
# Optimization: Fill the Delay Slot

```
x10          addi r1, r0, 0      # i=0
x14 Loop:    addi r2, r2, 2      # n += 2
x18          addi r1, r1, 1      # i++
x1C          blt  r1, r3, Loop   # i<max?
x20          nop
```

*Compiler transforms code*

```
x10          addi r1, r0, 0      # i=0
x14 Loop:    addi r1, r1, 1      # i++
x18          blt  r1, r3, Loop   # i<max?
x1C          addi r2, r2, 2      # n += 2
```

# Optimization In Action!




```
1C  blt r1, r3, Loop
20  addi r2,r2,2
14  Loop:addi r1,r1,1
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| F | D | X | | | | | |
| | F | D | | | | | |
| | | F | | | | | |
| | | | | | | | |

*No Nop or Zapping!*

# 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

# Branch Prediction Performance

Parameters

- **Branch: 20%,** load: 20%, store: 10%, other: 50%
- 75% of branches are taken

Dynamic branch prediction

- Branches predicted with 95% accuracy

What is the CPI with resolution @ decode?

- CPI = 1 + 20% * 5% * 2 = **1.02**

# 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.