



Pipelining & Performance

CS 3410: Computer System Organization and Programming

Spring 2025



Today's Goals



Consider what impacts
processor performance

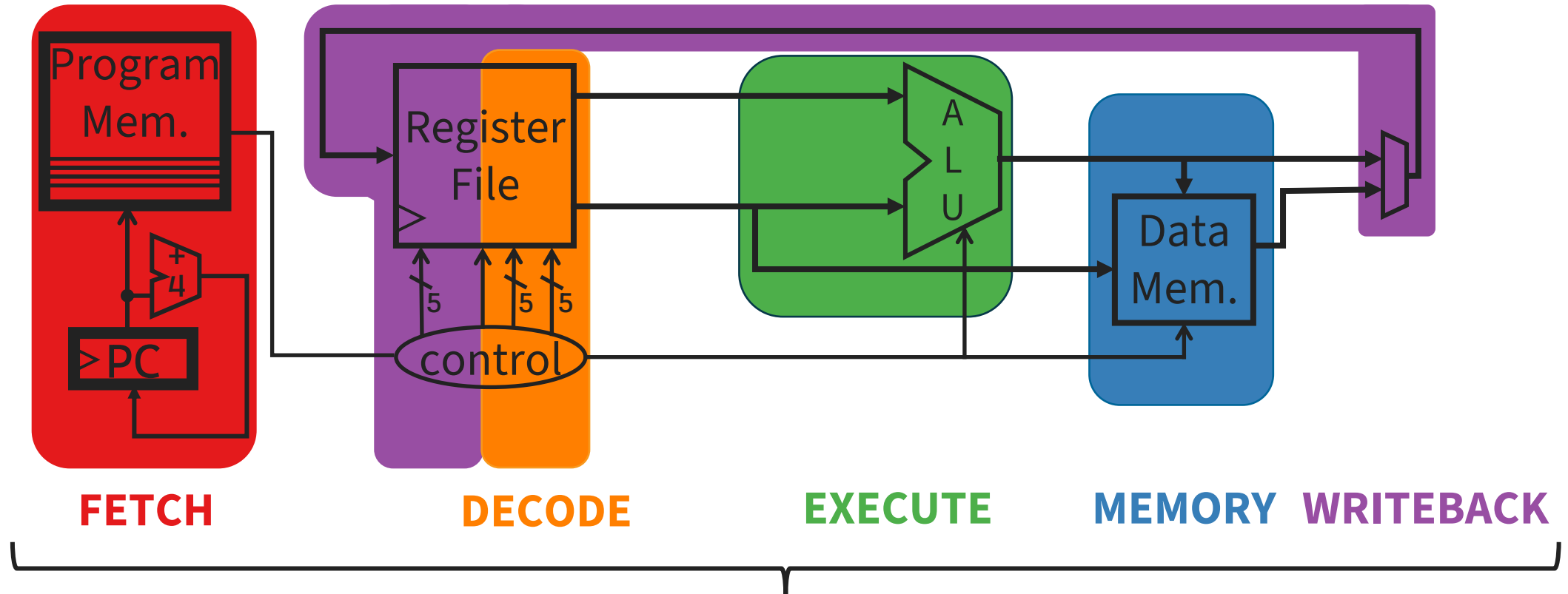
How to quantitatively estimate performance
Analyze performance / behavior with
diagrams



How to design processors with
better performance

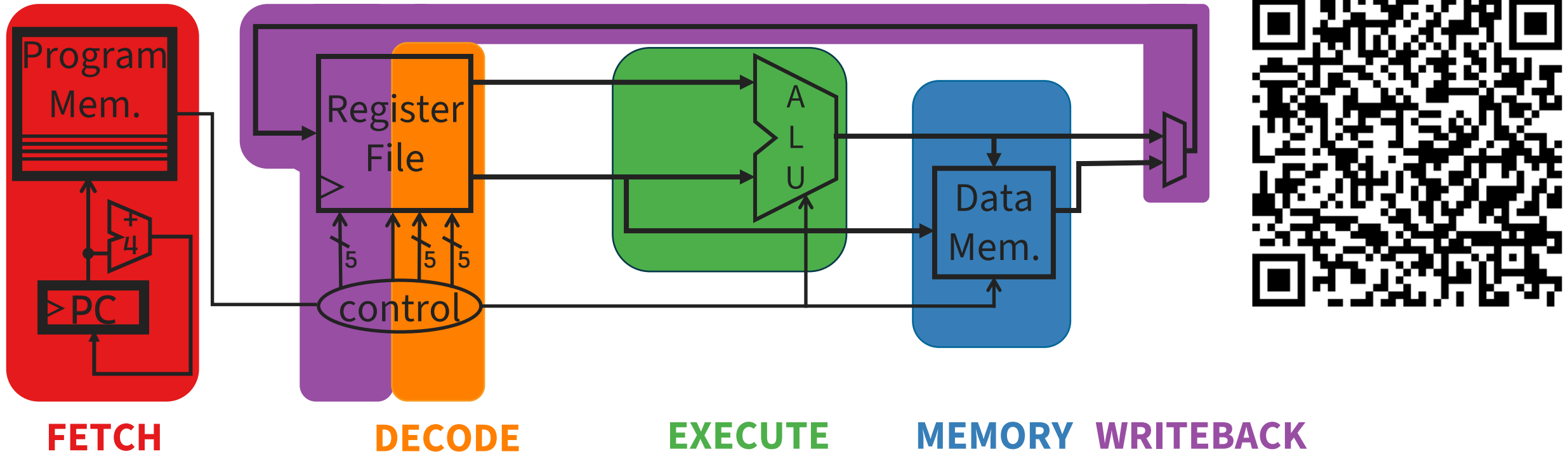
Single-cycle CPU
Multi-cycle CPU
Pipelined CPU

Single-Cycle RISC-V Datapath



Clock frequency must be **slow enough** for the very **slowest** instruction to complete in **1 cycle**

PollEverywhere



Which instruction, on average, will take the longest?

A: `add x9, x10, x11`

B: `lw x9, 0(x17)`

C: `addi x9, x10, -42`

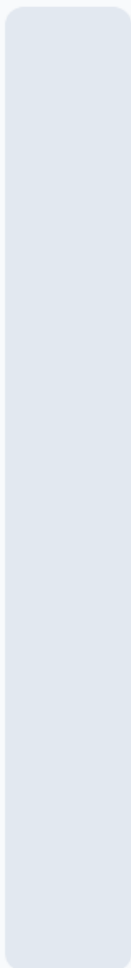
D: `sw x9, 0(x17)`

E: `beq x9, x10, EXIT`

Which instruction, on average, will take the longest?

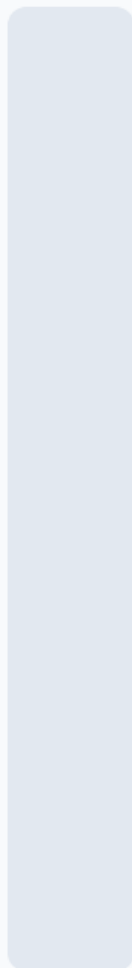


0%



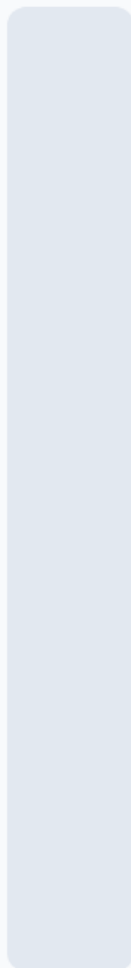
add x9, x10, x11

0%



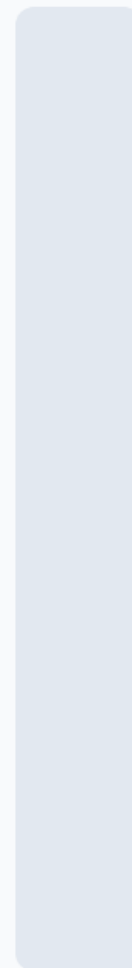
lw x9, 0(x17)

0%



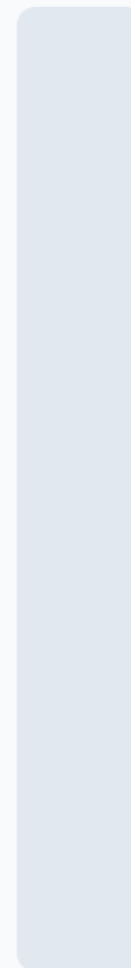
addi x9, x10, -42

0%



sw x9, 0(x17)

0%



beq x9, x19, EXIT

Ever been to Chipotle?



That's more like it!



Iron Law of Processor Performance

How do we make **a processor** that runs **programs faster**?

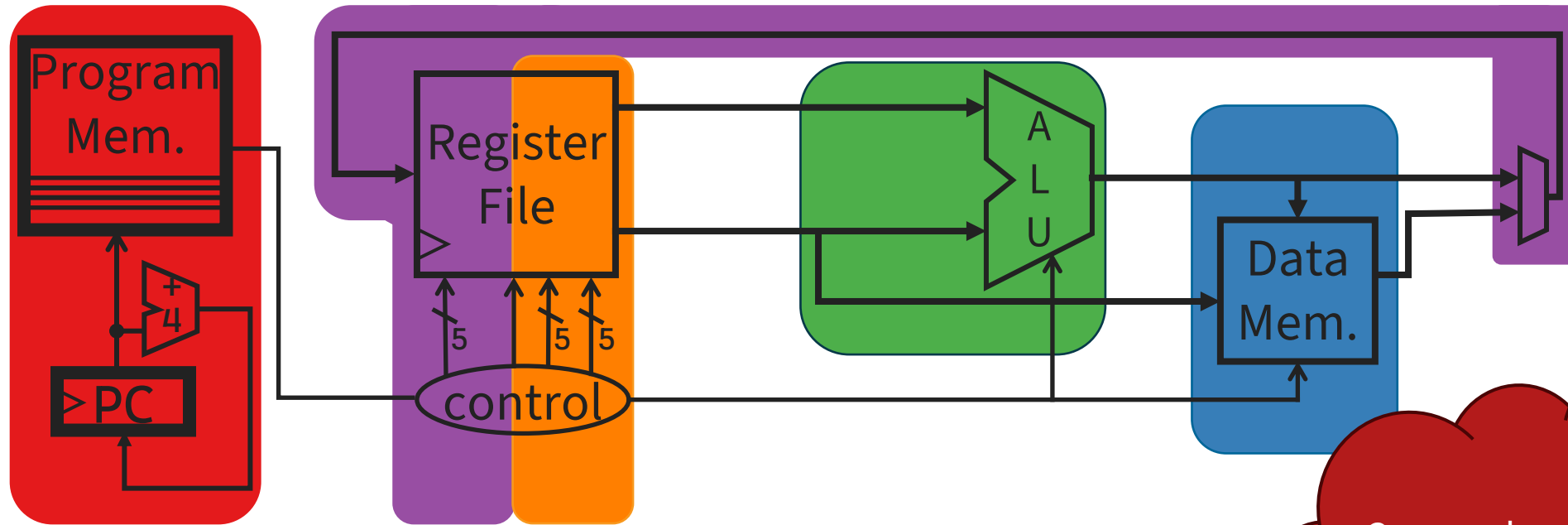
$$\frac{\textit{time}}{\textit{program}} = \frac{\textit{instructions}}{\textit{program}} \times \frac{\textit{cycles}}{\textit{instruction}} \times \frac{\textit{time}}{\textit{cycle}}$$

CPI **Clock
Period**

TODAY: tradeoff between CPI and clock period!



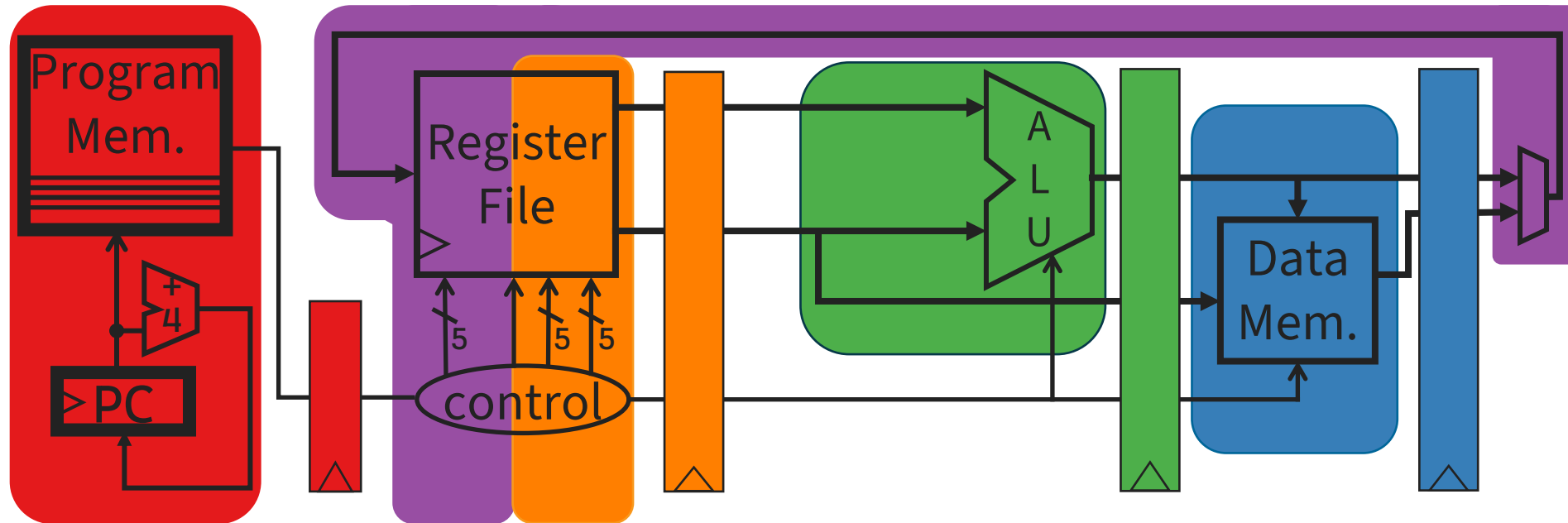
First Step: Shorten Clock Period



- Single-cycle processors have **long** clock periods
- Limited by **slowest** instruction

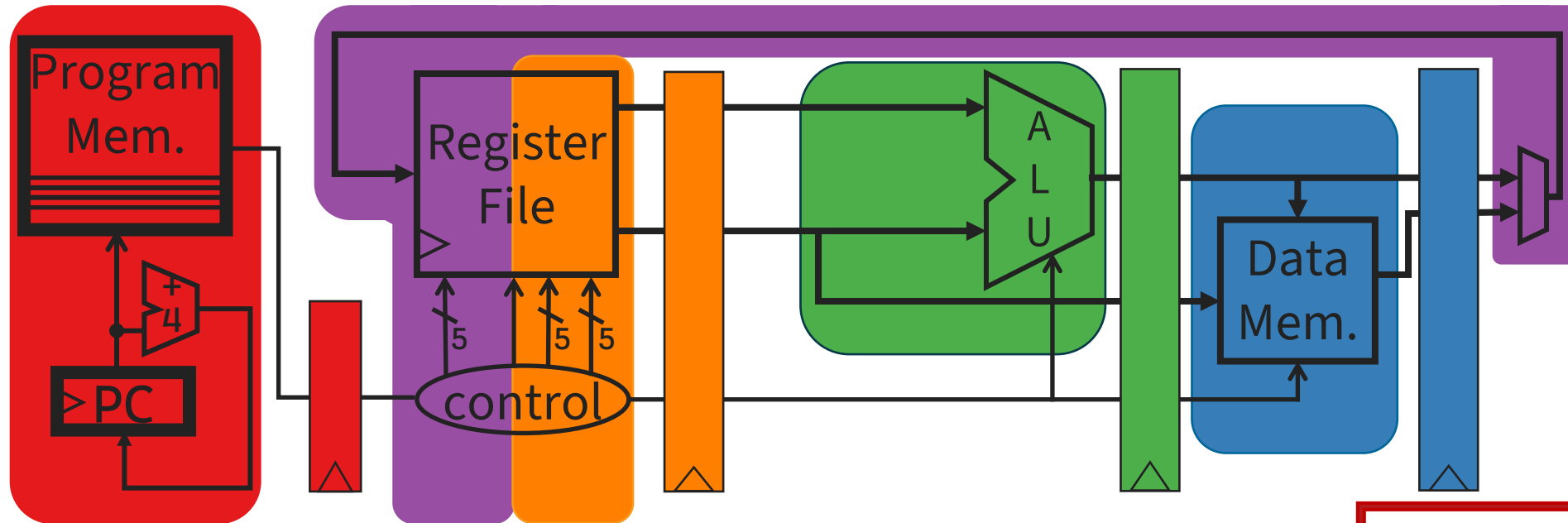
Can we decouple the clock period from instruction latency?

First Step: Multi-Cycle RISC-V Datapath



- Break datapath into **multiple cycles** (here 5)
- Add **registers** to store results at the end of each cycle
- Fetch, decode, and execute **1 instruction** over **multiple cycles**
- Allows instructions to take **different numbers of cycles**
- Opposite of single-cycle: short clock period, **high CPI**

First Step: Multi-Cycle RISC-V Datapath



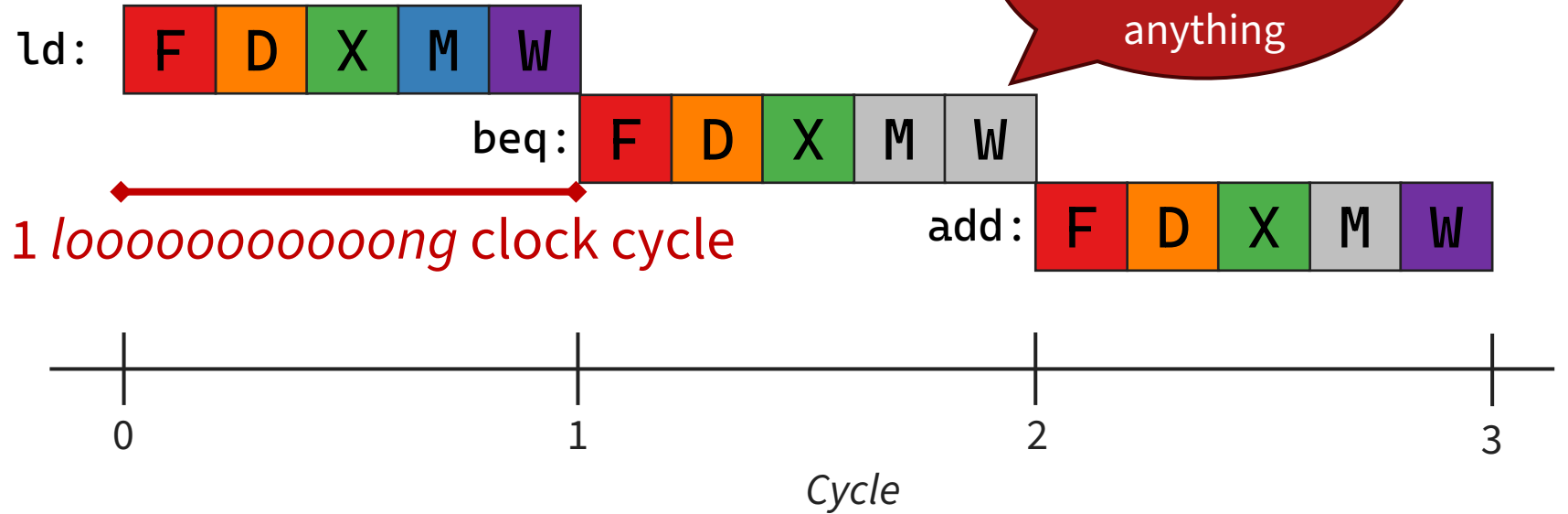
- Break datapath into **multiple cycles** (here 5)
- Add **registers** to store results at the end of each cycle
- Fetch, decode, and execute **1 instruction** over **multiple cycles**
- Allows instructions to take **different numbers of cycles**
- Opposite of single-cycle: short clock period, **high CPI**

Representation

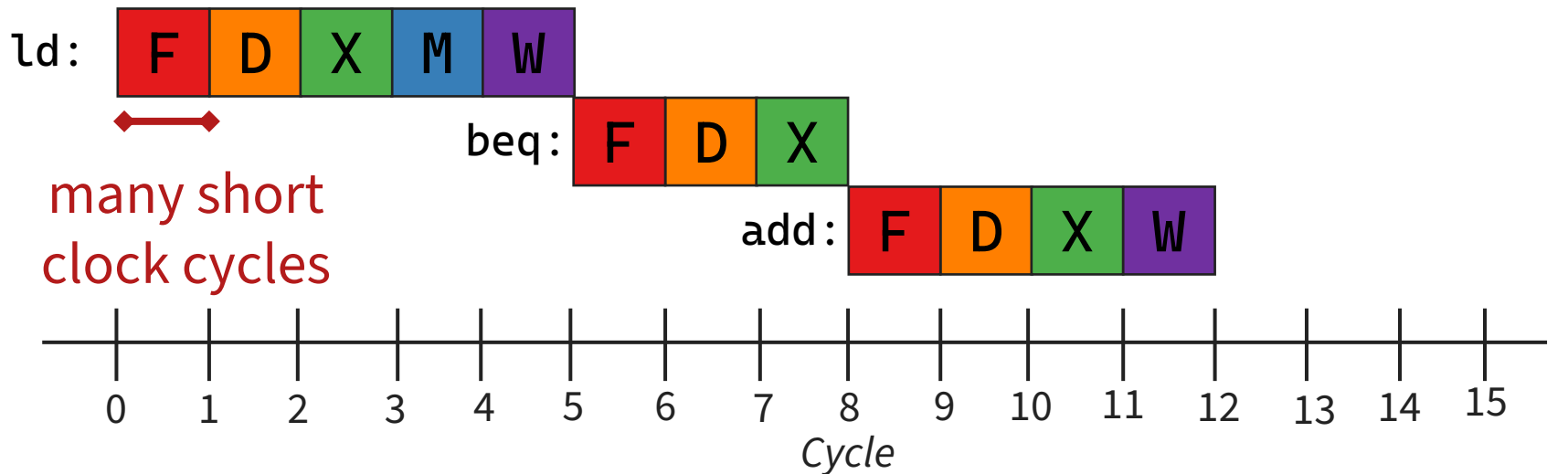


Single-Cycle vs. Multi-Cycle

Single-cycle



Multi-cycle



Single- vs. Multi-cycle Performance

Metric	Single Cycle	Multi Cycle
--------	--------------	-------------



Single- vs. Multi-cycle Performance

Metric	Single Cycle	Multi Cycle
Clock Period (time / cycle)	$F + D + X + M + WB$	$\text{MAX}(F, D, X, M, WB) + \epsilon$

ϵ is the overhead of accessing stage registers

Single- vs. Multi-cycle Performance

Metric	Single Cycle	Multi Cycle
Clock Period (time / cycle)	$F + D + X + M + WB$	$\text{MAX}(F, D, X, M, WB) + \epsilon$
Cycles Per Instruction (CPI)	1	?? (It depends!)

Use Average CPI – Depends on what programs (workloads) you run!

- E.g.: Branch: 20% (**3** cycles), Load: 20% (**5** cycles), ALU: 60% (**4** cycles)
 - $\text{CPI} = 0.2 * 3 + 0.2 * 5 + 0.6 * 4 = 4$
- *Caveat*: calculation ignores many effects
 - Back-of-the-receipt arguments only (i.e., it's a rough estimate)



Single- vs. Multi-cycle Performance

Metric	Single Cycle	Multi Cycle
Clock Period (time / cycle)	$F + D + X + M + WB$	$\text{MAX}(F, D, X, M, WB) + \epsilon$
Cycles Per Instruction (CPI)	1	(It depends!)
Performance (time / instruction)	Multiply down to see who wins!	



Single- vs. Multi-cycle Performance

Metric	Single Cycle	Multi Cycle
Clock Period (time / cycle)	900 ns	205 ns
Cycles Per Instruction (CPI)	1	4
Performance (time / instruction)	900 ns	820 ns

- Some concrete numbers:

- **Stage latency:** **F** = 170ns, **D** = 180ns, **X** = 200ns, **M** = 200ns, **W** = 150ns , Register = 5ns
- **Branch:** 20% (**3** cycles), **Load:** 20% (**5** cycles), **ALU:** 60% (**4** cycles)



Which of the following statements is true?



A multi-cycle CPU is **always** faster than a single-cycle CPU

Multi-cycle CPUs are **always** more efficient than single-cycle CPUs (i.e., less ti...

Adding more stages **always** makes the clock period shorter

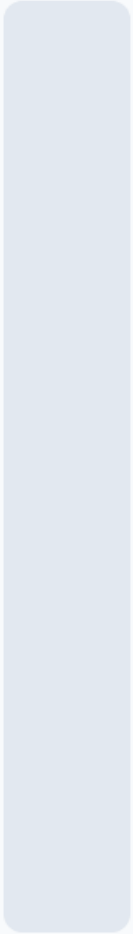
Multi-cycle CPUs have more complex **control logic** than single-cycle CPUs

B & D

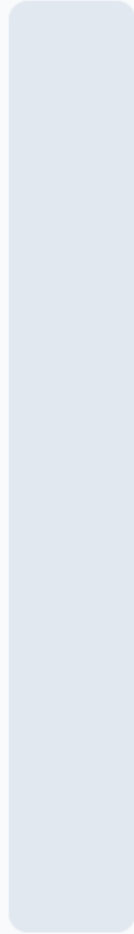
Which of the following statements is true?



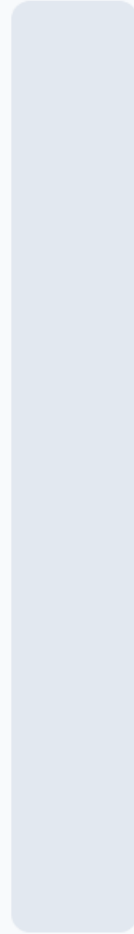
0%



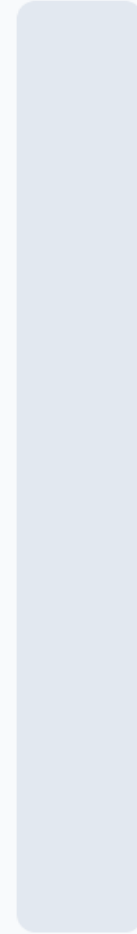
0%



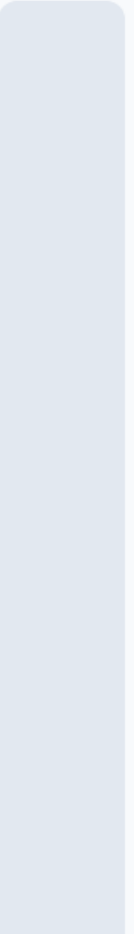
0%



0%



0%



A multi-cycle CPU is **always** faster than a single-cycle CPU

Multi-cycle CPUs are **always** more efficient than single-cycle CPUs (i.e., less time spent waiting/doing nothing)

Adding more stages **always** makes the clock period shorter

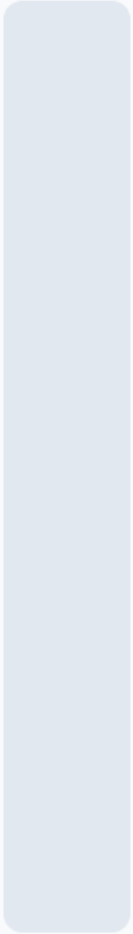
Multi-cycle CPUs have more complex **control logic** than single-cycle CPUs

B & D

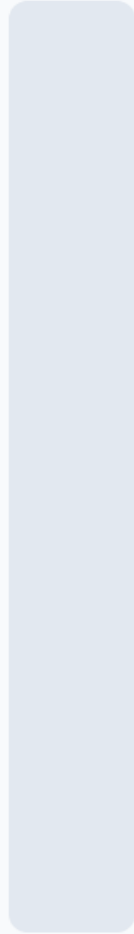
Which of the following statements is true?



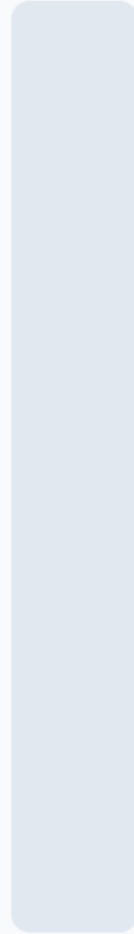
0%



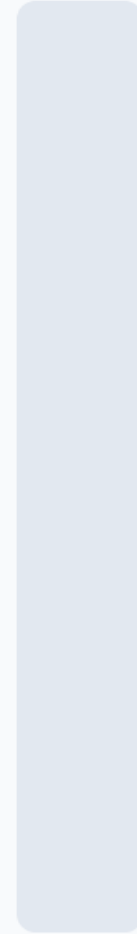
0%



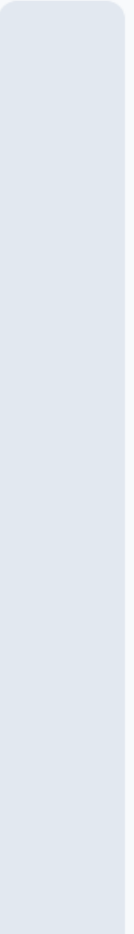
0%



0%



0%



A multi-cycle CPU is **always** faster than a single-cycle CPU

Multi-cycle CPUs are **always** more efficient than single-cycle CPUs (i.e., less time spent waiting/doing nothing)

Adding more stages **always** makes the clock period shorter

Multi-cycle CPUs have more complex **control logic** than single-cycle CPUs

B & D

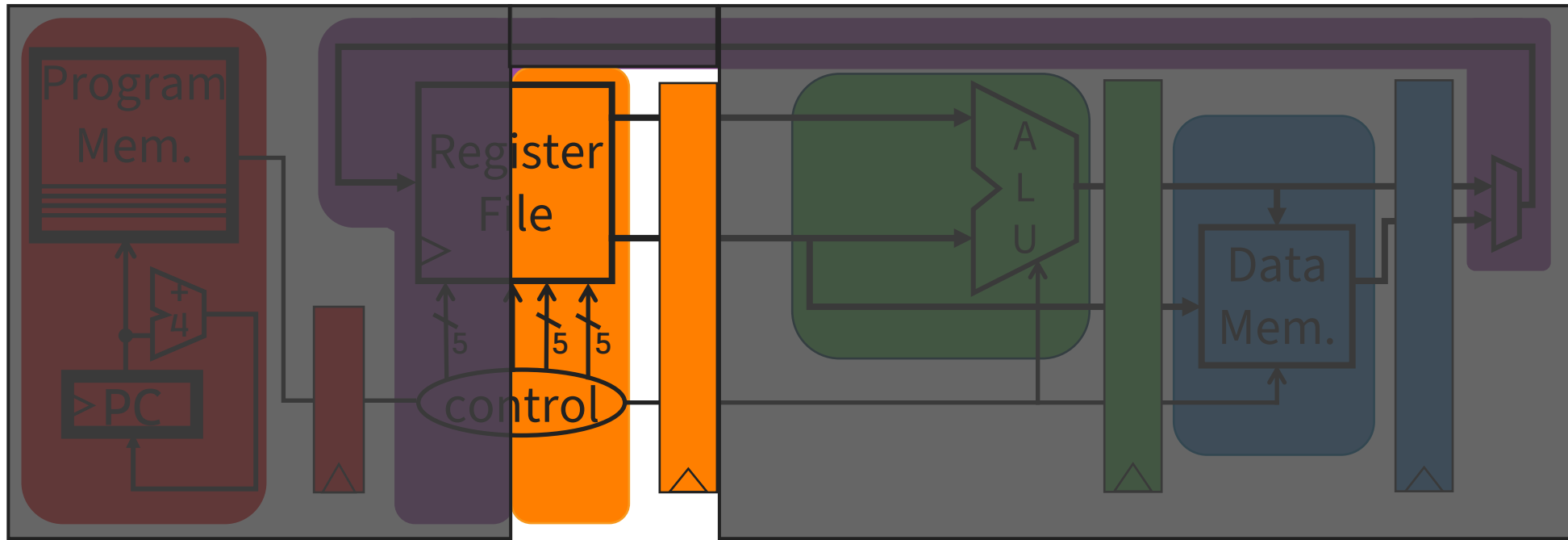
Is multi-cycle better?

“When you see a good move,
look for a better one.”

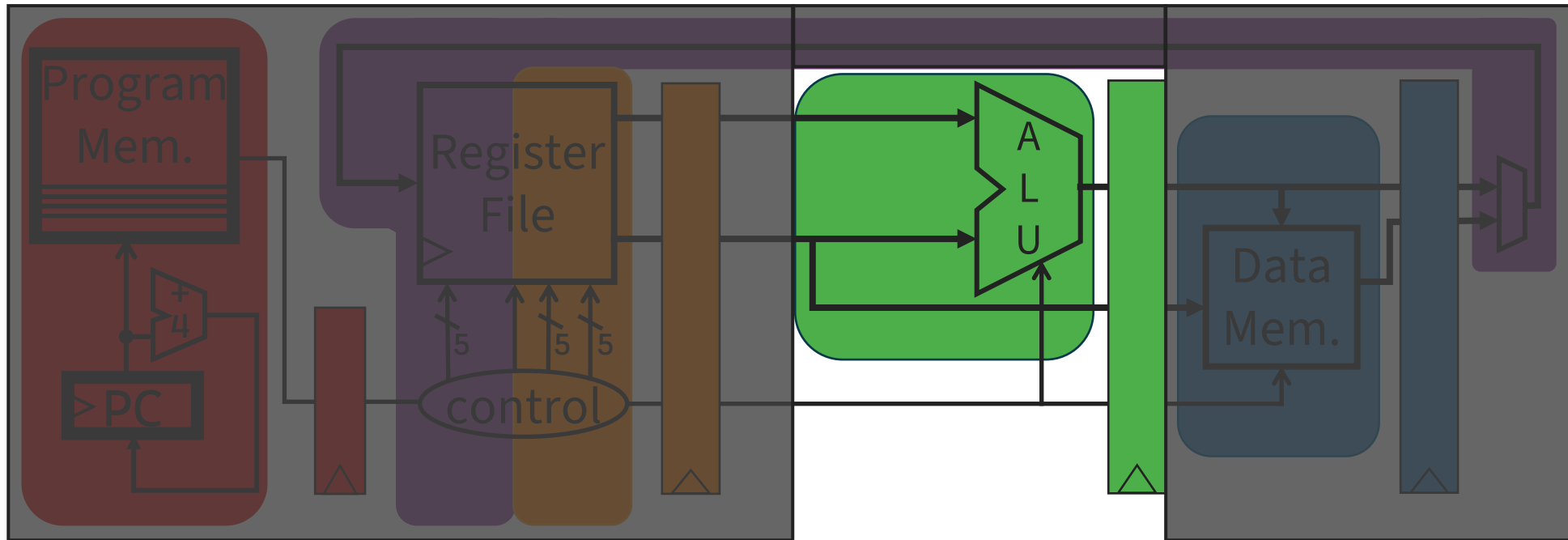


-Emanuel Lasker

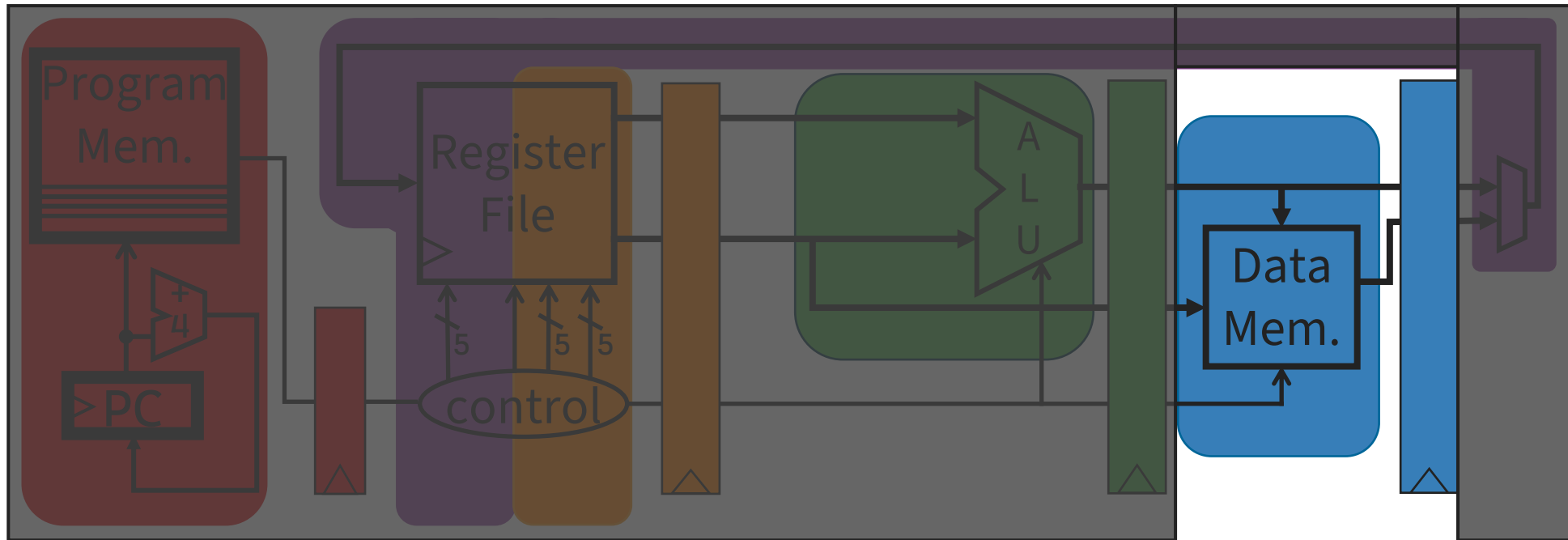
Improving Multi-Cycle Pipeline



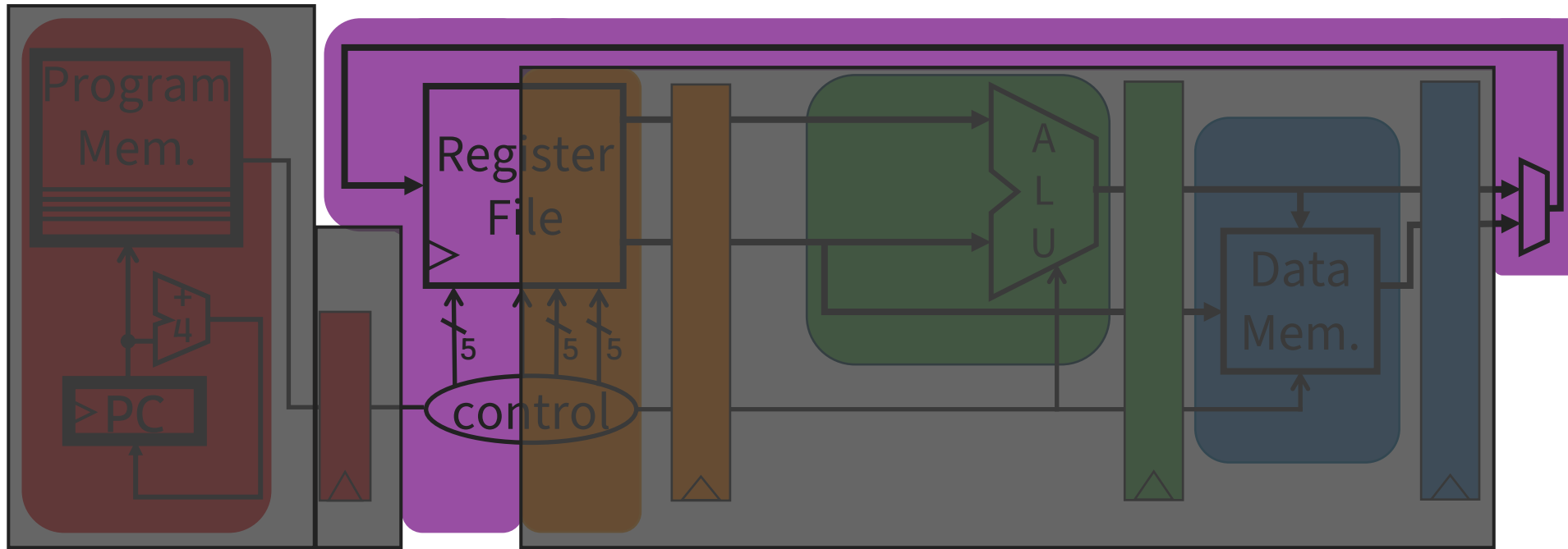
Improving Multi-Cycle Pipeline



Improving Multi-Cycle Pipeline



Improving Multi-Cycle Pipeline



Only **one stage** of the CPU is active per cycle!

Pipelining

*An implementation technique in which multiple instructions are **overlapped** in execution.*

Pipelining Example: Laundry

- Doing 1 load of laundry requires the sequence:

- Wash



20 min

- Dry



30 min

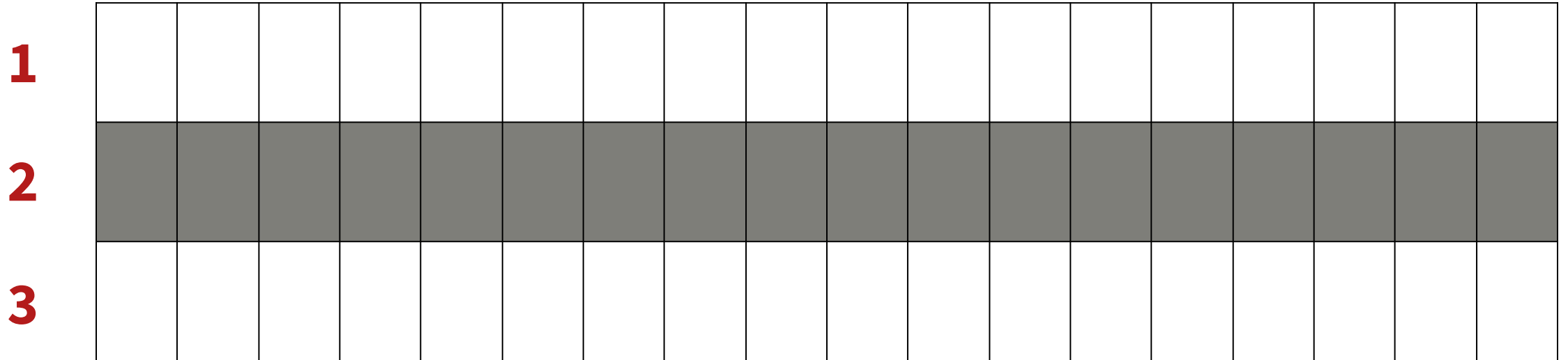
- Fold



10 min

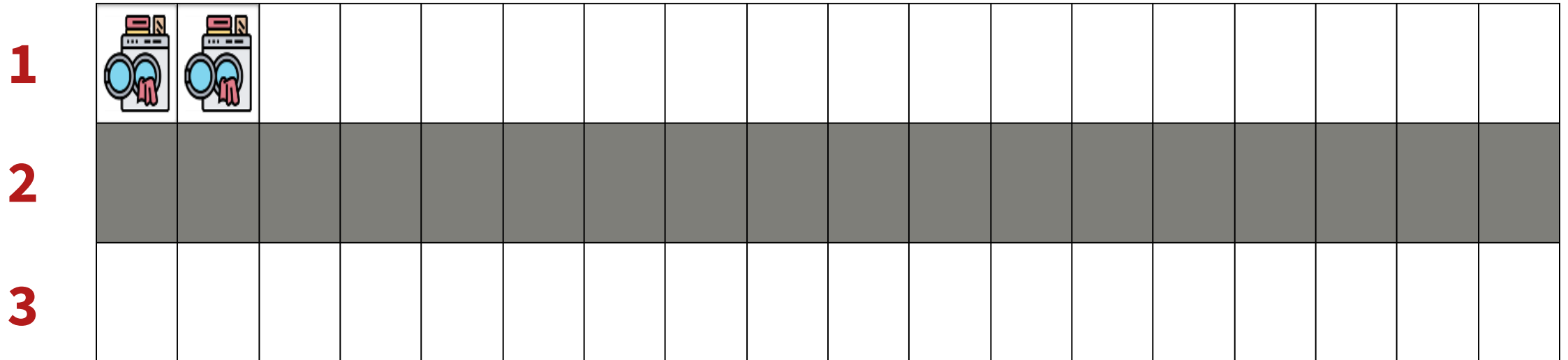
Laundry Example

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180







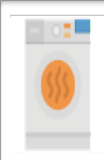
Laundry Example – Serial Case

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180







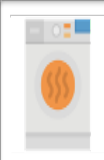

Laundry Example – Serial Case

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180

1																		
2																		
3																		

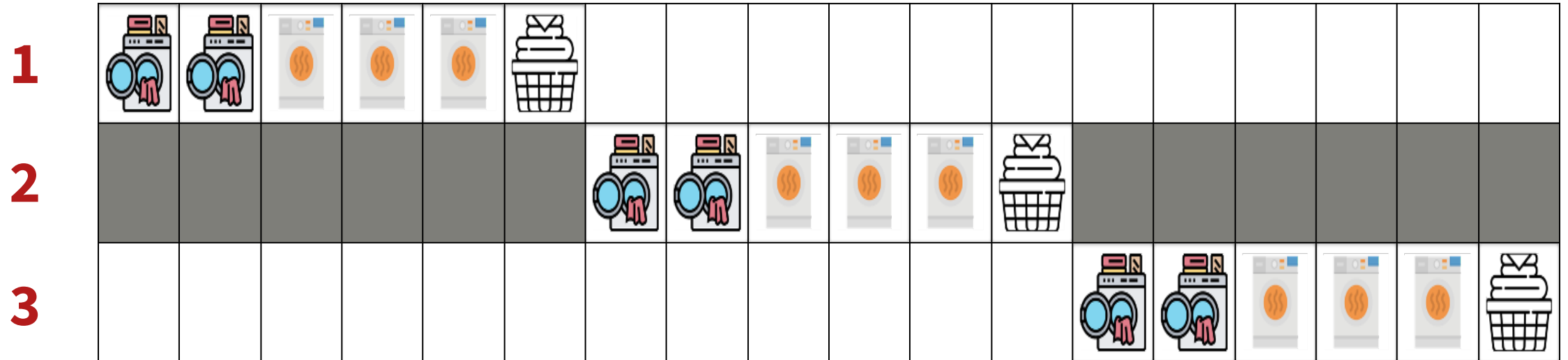
Laundry Example – Serial Case

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180

1																		
2																		
3																		

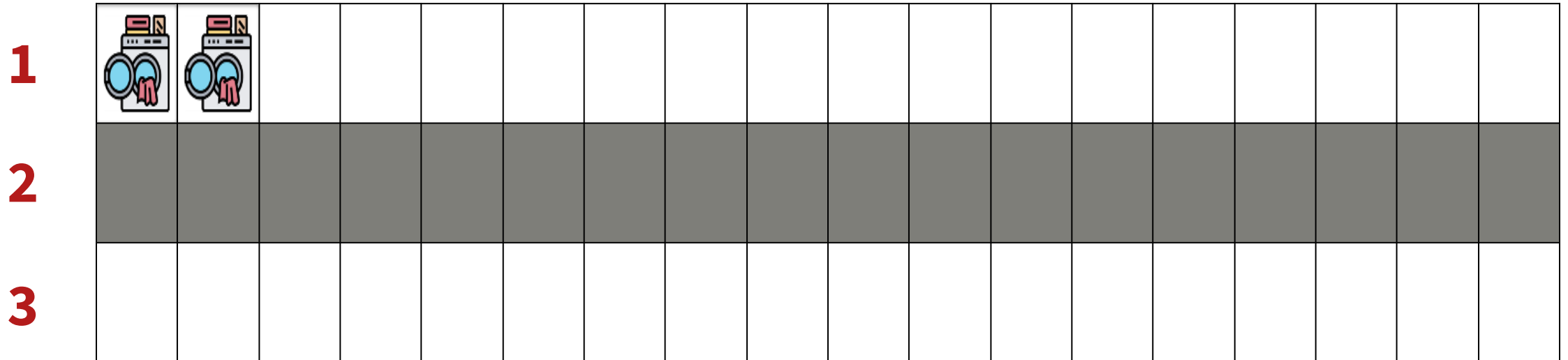
Laundry Example – Serial Case

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180



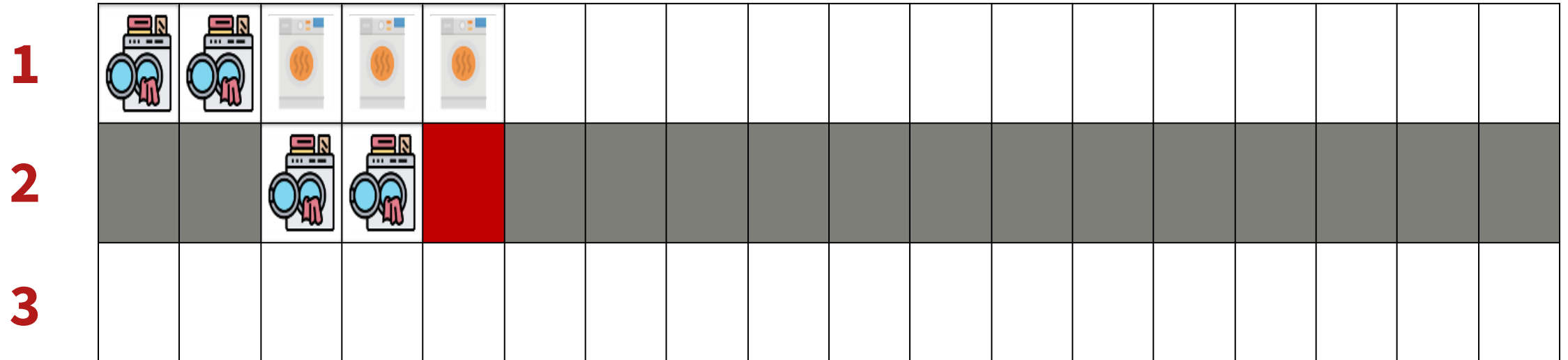
Laundry Example – But with Pipelining!

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180



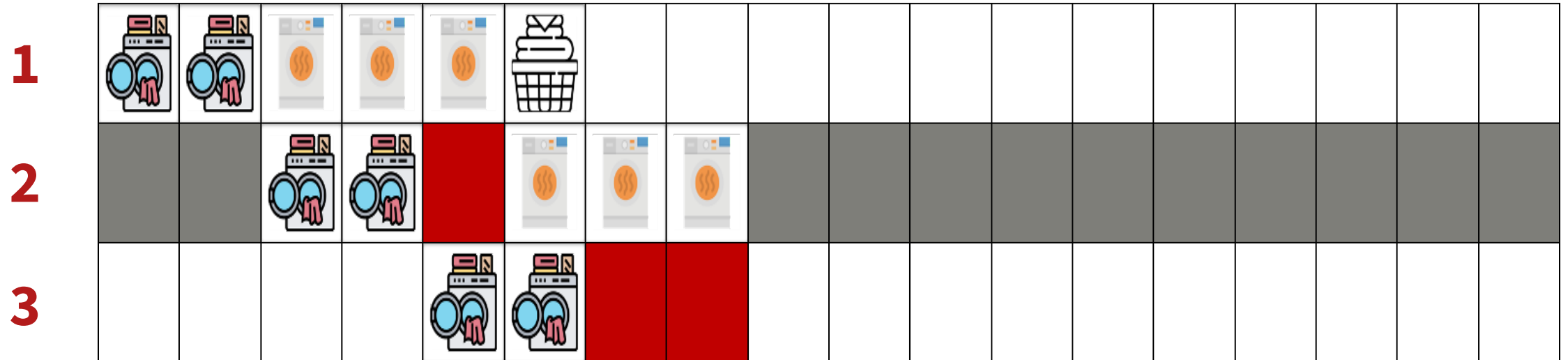
Laundry Example – But with Pipelining!

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180



Laundry Example – But with Pipelining!

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180



Laundry Example – But with Pipelining!

Load # 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180



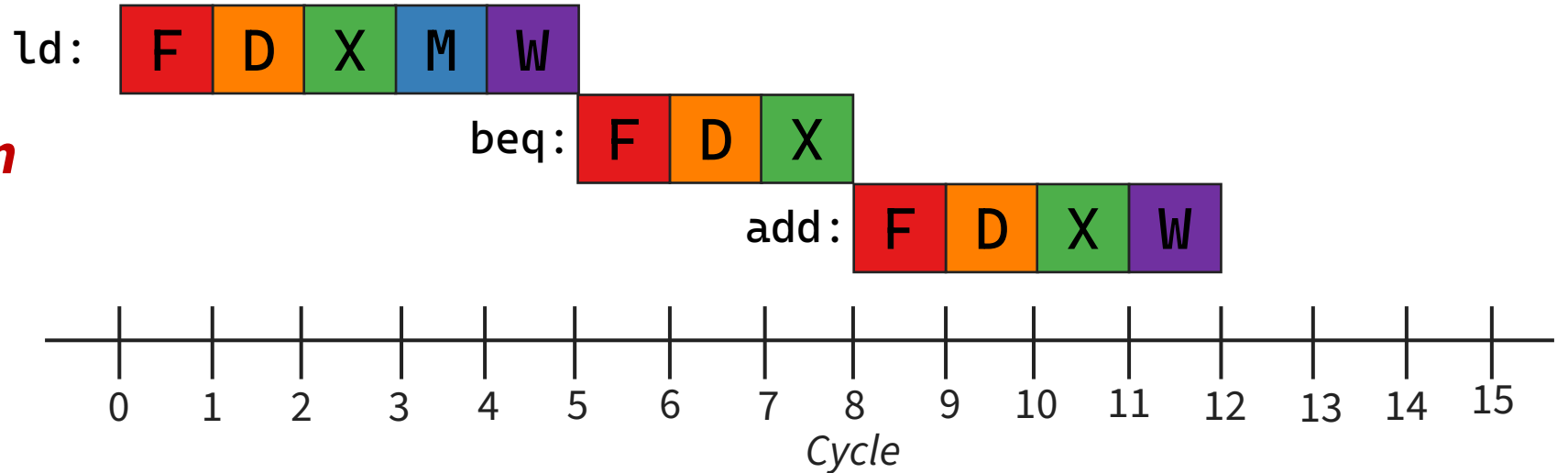
Which resources are in use at a given time

Which resources a given task uses over time.

Multi-Cycle → Pipelined

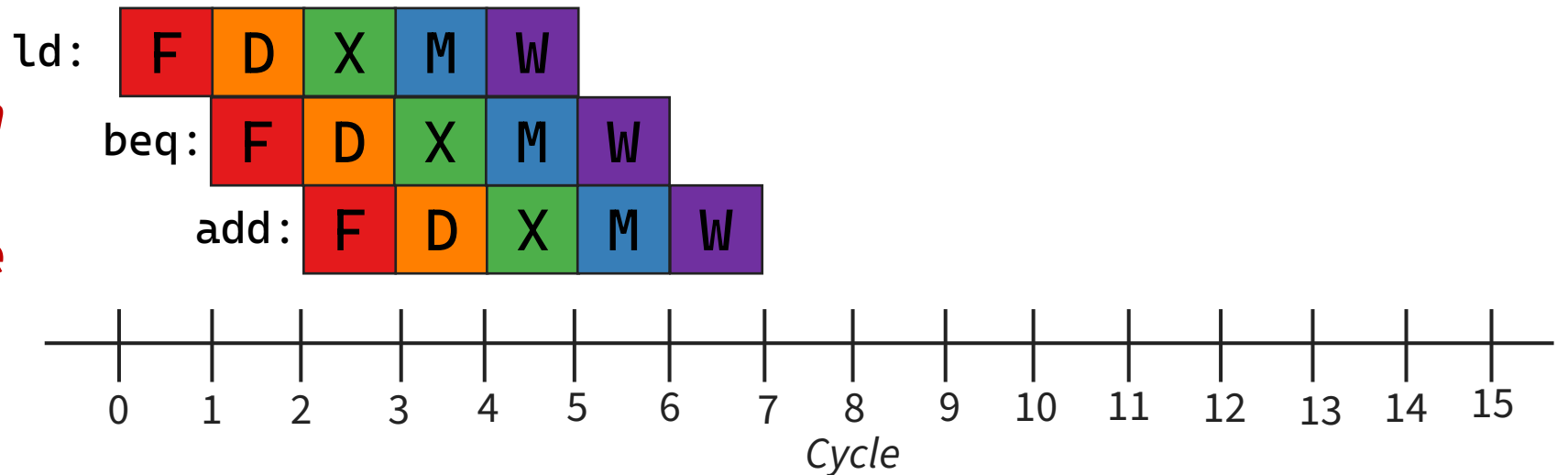
Multi-cycle

Each instruction takes n **short cycles** based on the work that **needs** to be done



Pipelined

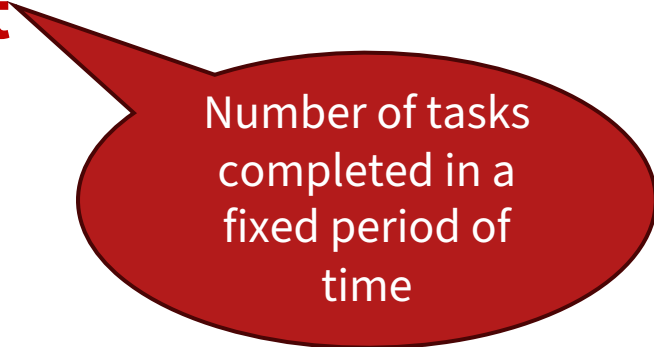
Each instruction takes n **short cycles** no matter what, but runs **multiple** instructions **in parallel**



Principles of Pipelining

Break datapath into **multiple cycles** (5 for our RISC-V example)

- Parallel execution increases **throughput**
- Balanced pipeline very important
 - Slowest stage determines clock rate
 - Imbalance kills performance



Number of tasks completed in a fixed period of time

Add **pipeline registers (flip-flops)** for isolation

- Stage *begins* by reading values **from** previous register
- Stage *ends* by writing values **to** next register

Pipeline Stages

Stage	Functionality	Values of Interest (to be latched)



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



p

n

$< \frac{p}{n}$

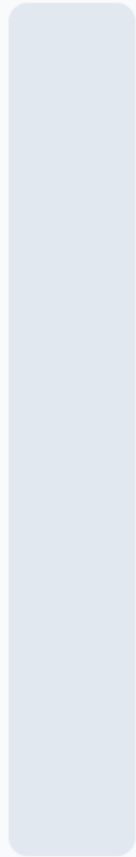
$\frac{p}{n}$

$> \frac{p}{n}$

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

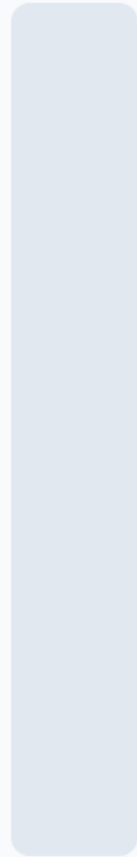


0%



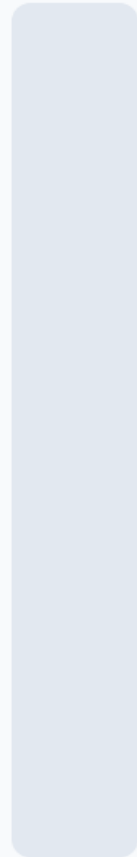
p

0%



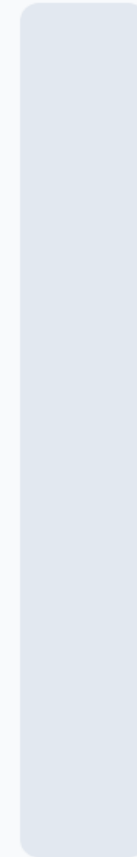
n

0%



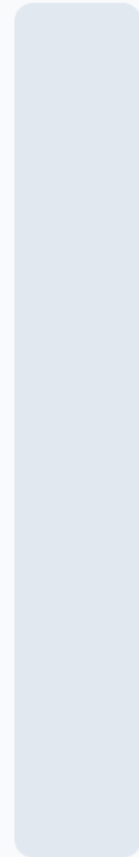
$< \frac{p}{n}$

0%



$\frac{p}{n}$

0%

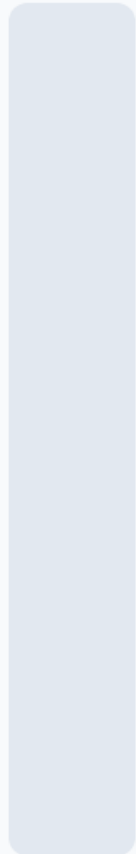


$> \frac{p}{n}$

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

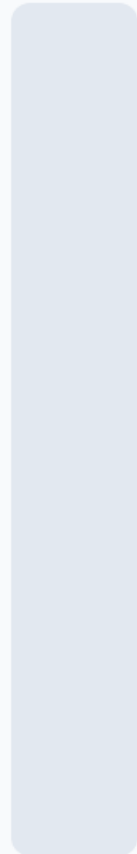


0%



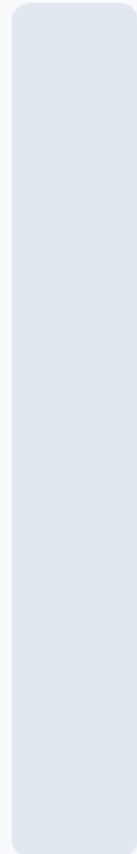
p

0%



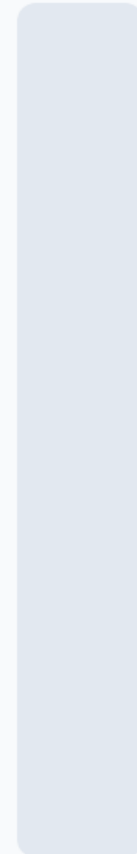
n

0%



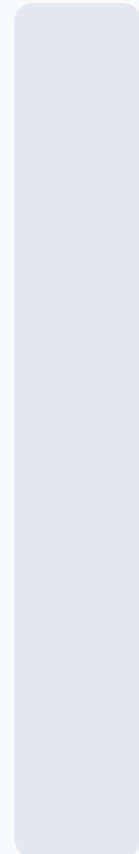
$< \frac{p}{n}$

0%



$\frac{p}{n}$

0%



$> \frac{p}{n}$

RISC-V is *Designed* for Pipelining

Instructions same length (32 bits)

- easy to fetch
- easy to decode

Few instruction formats

- Easy to decode
- Easy to route bits between stages
- Can read a register source before even knowing what the instruction is!

Memory accessed through `lw` and `sw` only

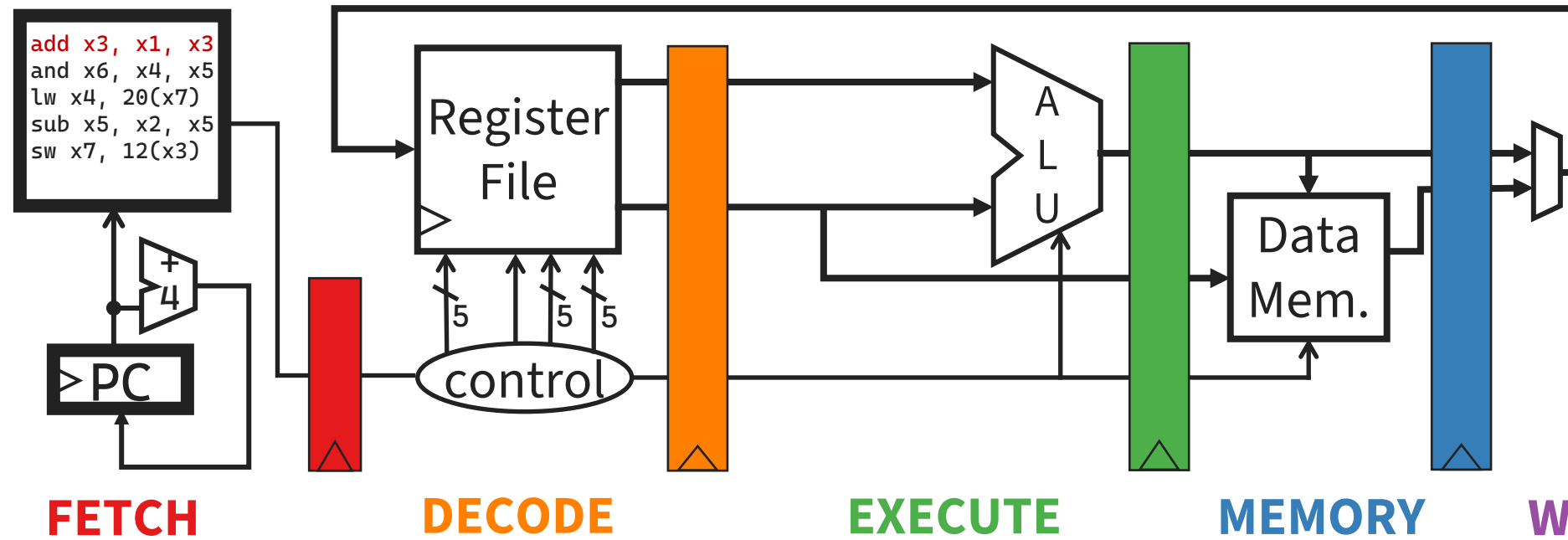
- Access memory after ALU

funct	rs2	rs1	funct3	rd	op
imm		rs1	funct3	rd	op
imm	rs2	rs1	funct3	imm	op
imm				rd	op

RISC-V Pipelining in Action!

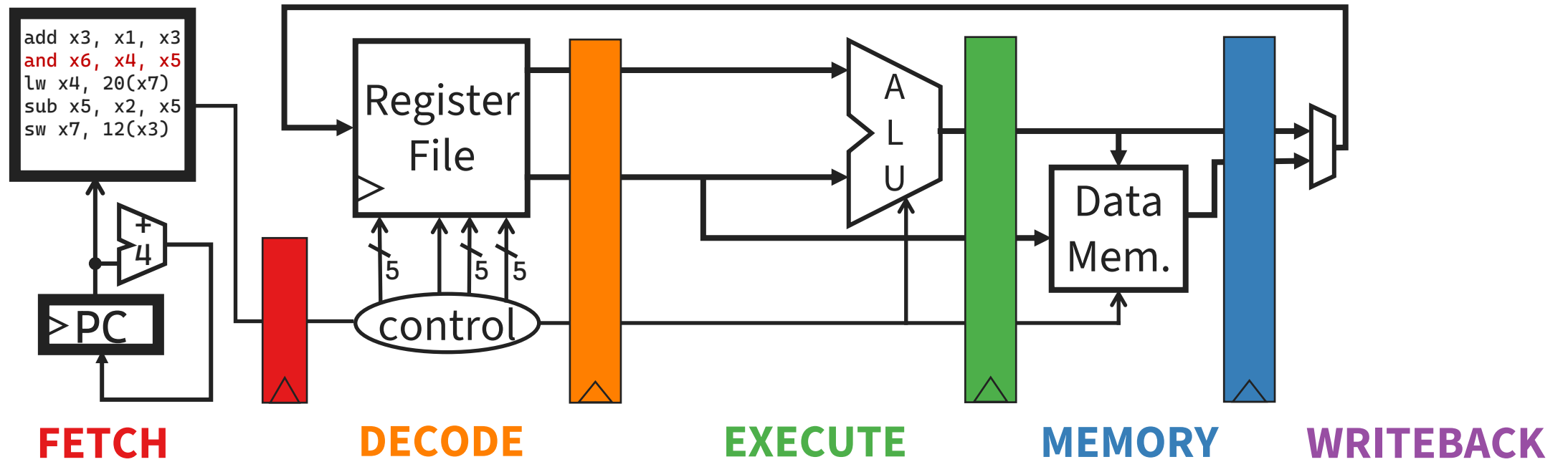
```
add    x3, x1, x3
and    x6, x4, x5
lw     x4, 20(x7)
sub    x5, x2, x5
sw     x7, 12(x3)
```

Pipelining in Action (1)



add x3, x1, x3

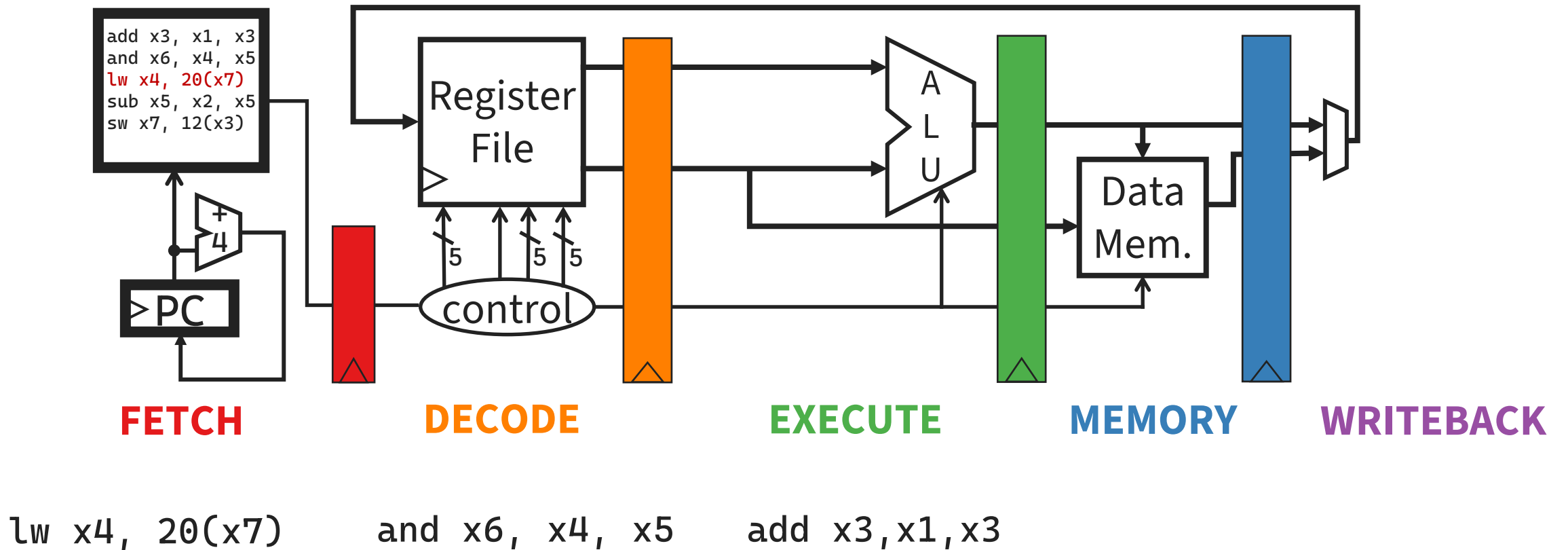
Pipelining in Action (2)



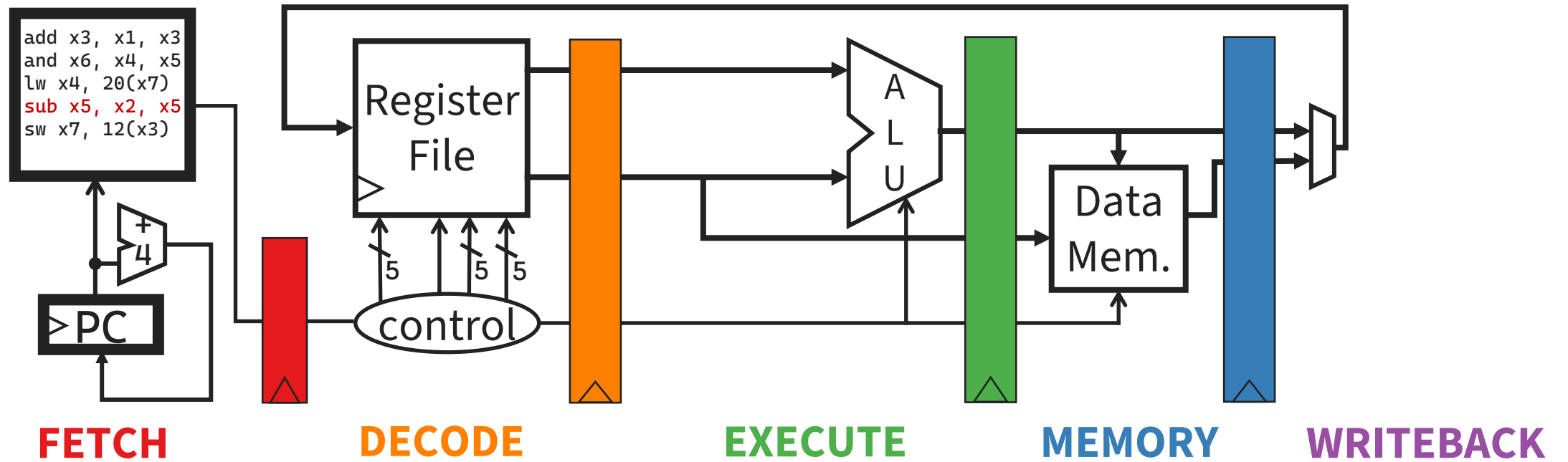
and x6, x4, x5

add x3, x1, x3

Pipelining in Action (3)

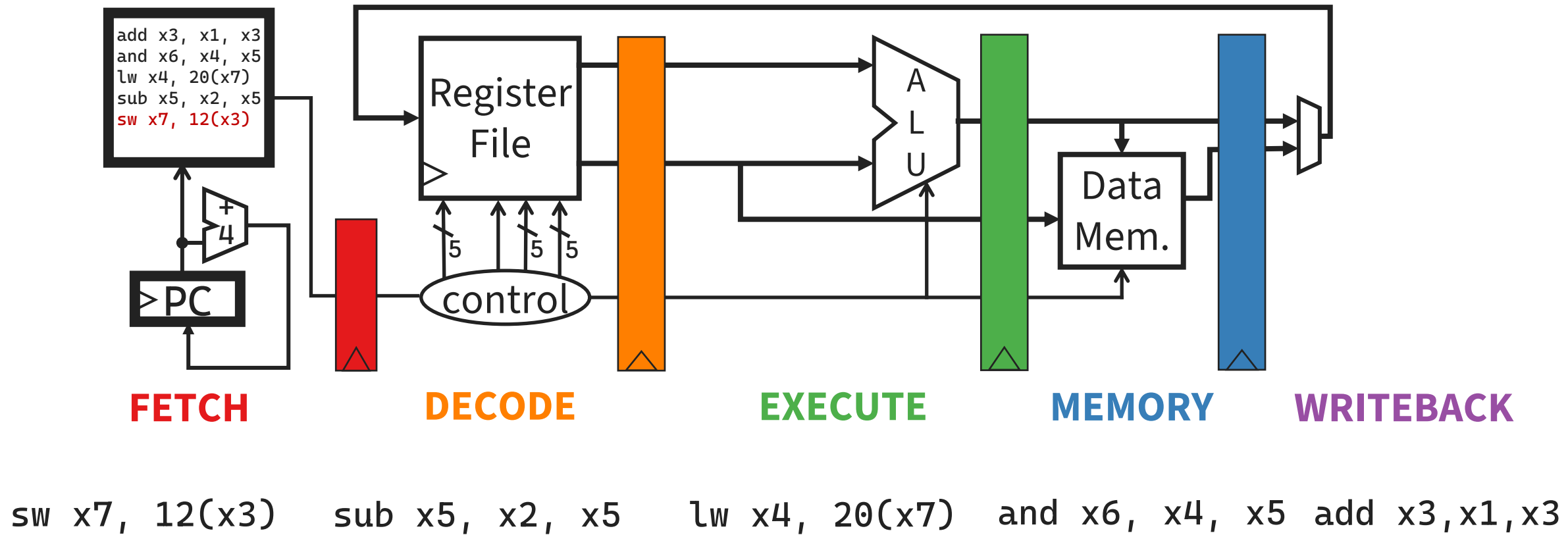


Pipelining in Action (4)



sub x5, x2, x5 lw x4, 20(x7) and x6, x4, x5 add x3, x1, x3

Pipelining in Action (5)



Interface vs. Implementation

Pipelining is a powerful technique to mask latencies and increase throughput

- Logically, instructions execute one at a time
- Physically, instructions execute in parallel

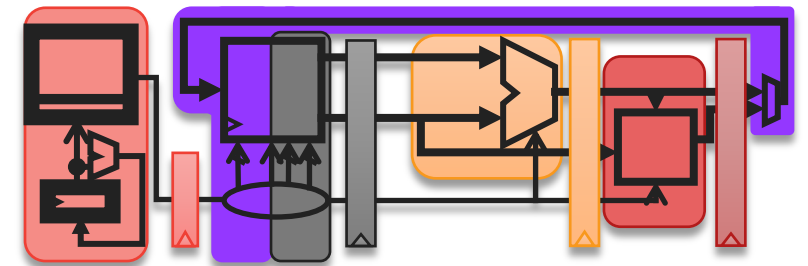
Abstraction promotes decoupling

- Interface (ISA) vs. implementation (Pipeline)

Compiler
Thinks About:

```
1. add    x3, x1, x3
2. and    x6, x4, x5
3. lw     x4, 20(x7)
4. sub    x5, x2, x5
5. sw     x7, 12(x3)
```

Architect
Builds:



Pipelining is great because...



You can fetch and decode the same instruction at the same time.

You can fetch two instructions at the same time.

You can fetch one instruction while decoding another.

Instructions only need to visit the pipeline stages that they require.

C & D

Pipelining is great because...



You can fetch and decode the same instruction at the same time.

0%

You can fetch two instructions at the same time.

0%

You can fetch one instruction while decoding another.

0%

Instructions only need to visit the pipeline stages that they require.

0%

C & D

0%

Pipelining is great because...



You can fetch and decode the same instruction at the same time.

0%

You can fetch two instructions at the same time.

0%

You can fetch one instruction while decoding another.

0%

Instructions only need to visit the pipeline stages that they require.

0%

C & D

0%

CPU Performance

Metric	Single Cycle	Multi Cycle	Pipelined
Clock Period (time / cycle)	$F + D + X + M + W$	$\text{MAX}(F, D, X, M, W) + \epsilon$	$\text{MAX}(F, D, X, M, W) + \epsilon$
Cycles Per Instruction (CPI)	1	(It depends!)	1
Performance (time / instruction)	Multiply down to see who wins!		

Pipelining is the best of both worlds!!

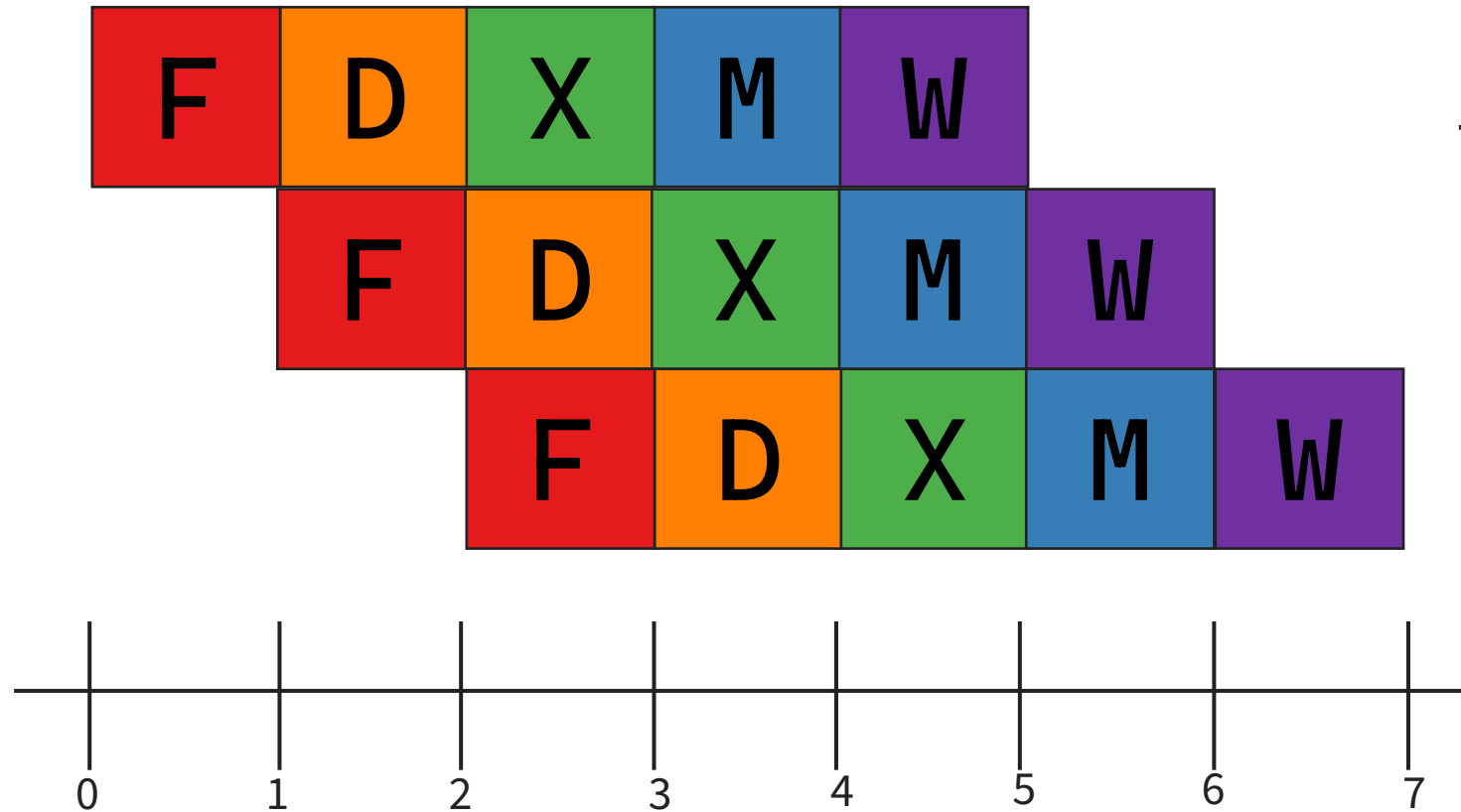


Pipeline Diagrams



Pipeline Diagrams

What two instruction sequence would not function correctly given this pipelined processor?

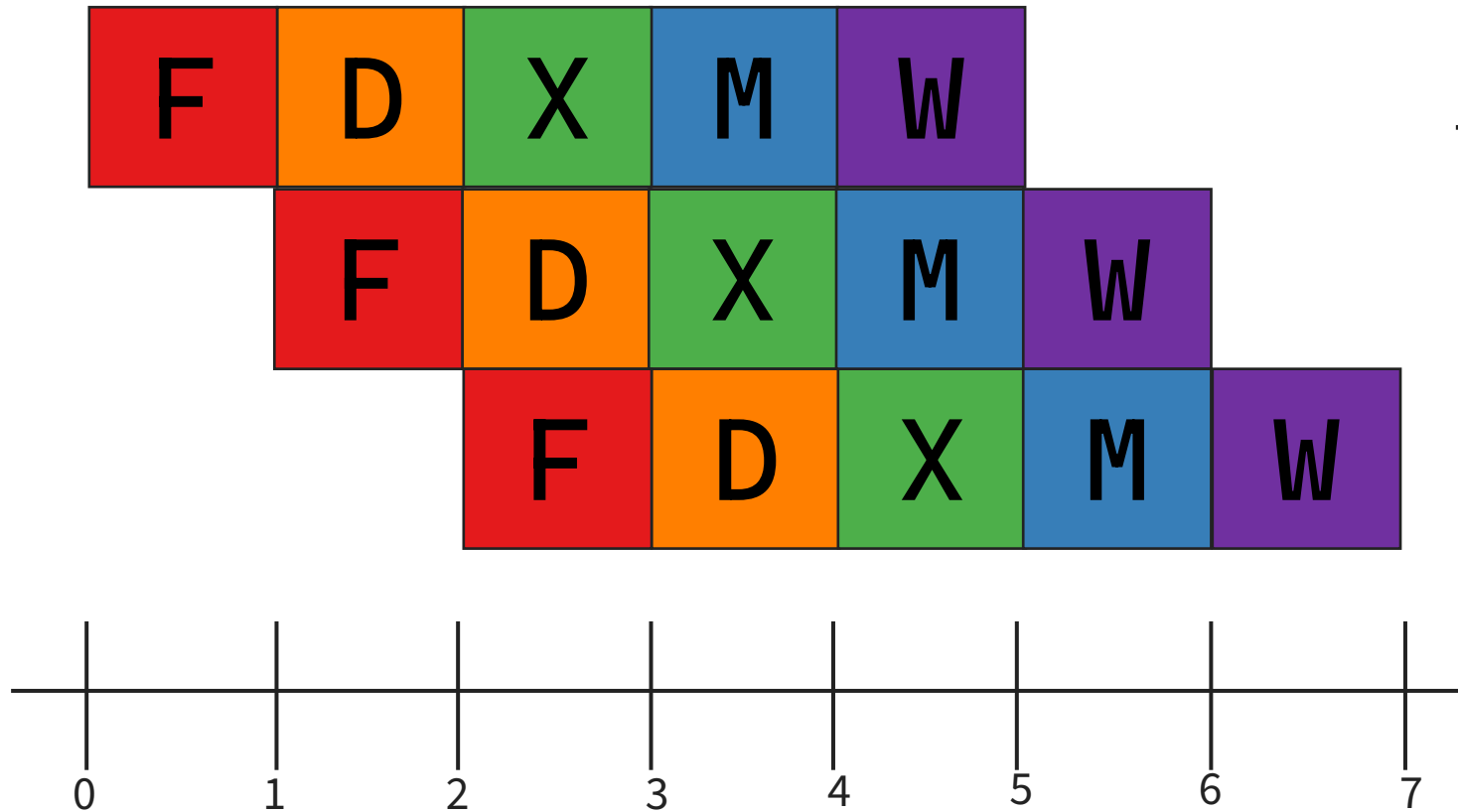


Pipeline Diagrams

What two instruction sequence would not function correctly given this pipelined processor?

```
lui x1, imm  
addi x2, x1, 3
```

```
j LABEL  
<any instruction>
```



Today's Goals



Consider what impacts
processor performance

How to quantitatively estimate performance
Analyze performance / behavior with
diagrams



How to design processors with
better performance

Single-cycle CPU
Multi-cycle CPU
Pipelined CPU