COMP2300-COMP6300-ENGN2219
Computer Organization & Program Execution

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

Australian National University

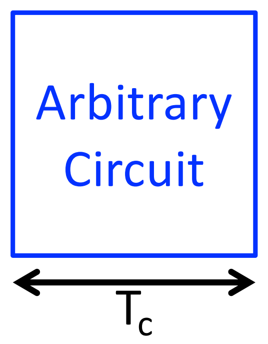
Recall: Temporal Parallelism

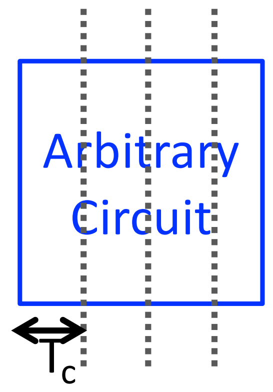
- **Temporal Parallelism (pipelining):**
  - Break down a circuit into stages
  - Each task passes through all stages
  - Multiple tasks are spread through stages

Recall: Automotive Pipeline

Recall: Pipelining

- If a task of latency L is broken into N stages, and all stages are of equal length, then the throughput is N/L



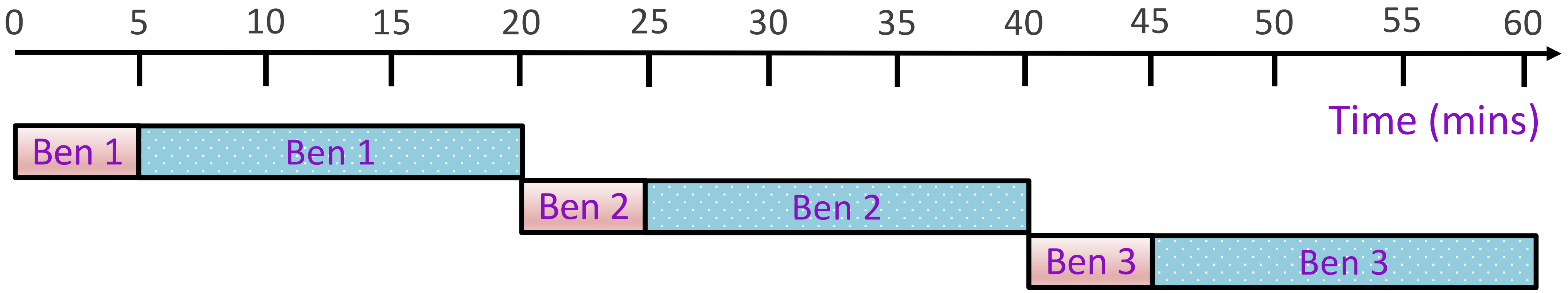


- The challenge of pipelining is to find stages of equal length
- Let’s go back to baking cookies

Recall: Cookie Parallelism

- Ben and Jon are making cookies. Let's study the latency and throughput of rolling and baking many cookie trays with
  - No parallelism
  - Spatial parallelism
  - Pipelining
  - Spatial parallelism + pipelining

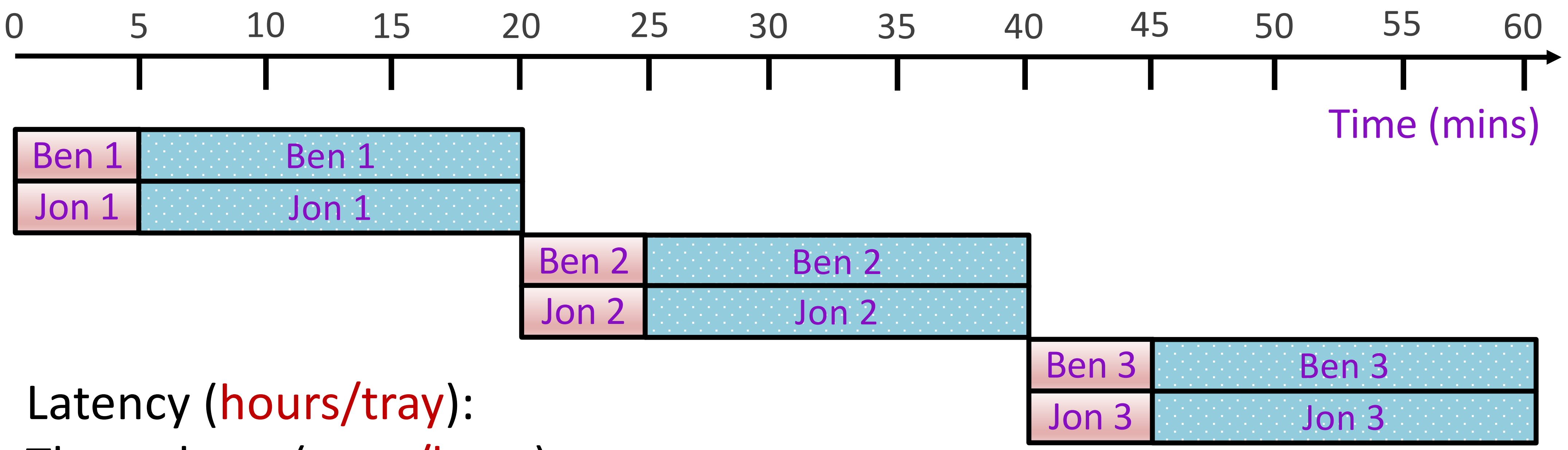
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

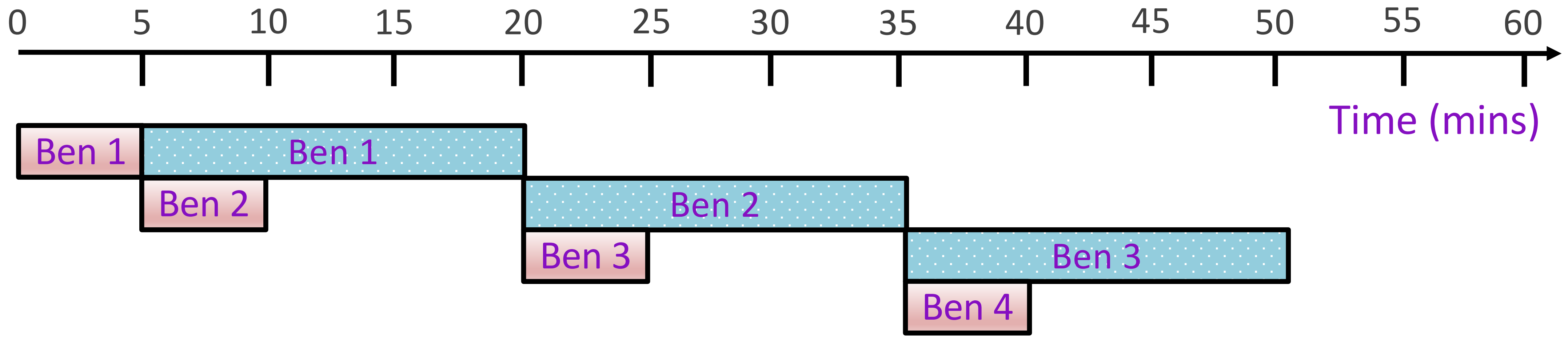


Latency (hours/tray):

Throughput (trays/hour):

Note: Jon owns a tray and oven (hardware duplication)

Pipelining (Ben Only)

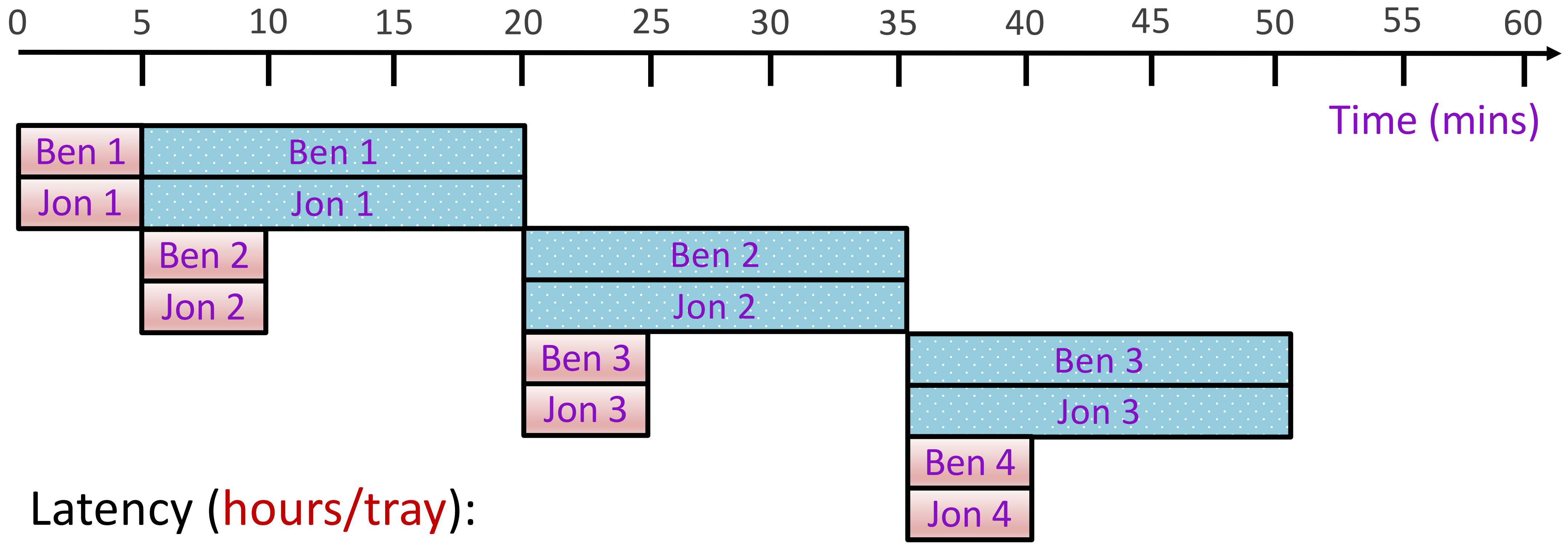


Latency (hours/tray):

Throughput (trays/hour):

Note: Ben decides not to waste a separate tray and oven

Spatial + Temporal Parallelism



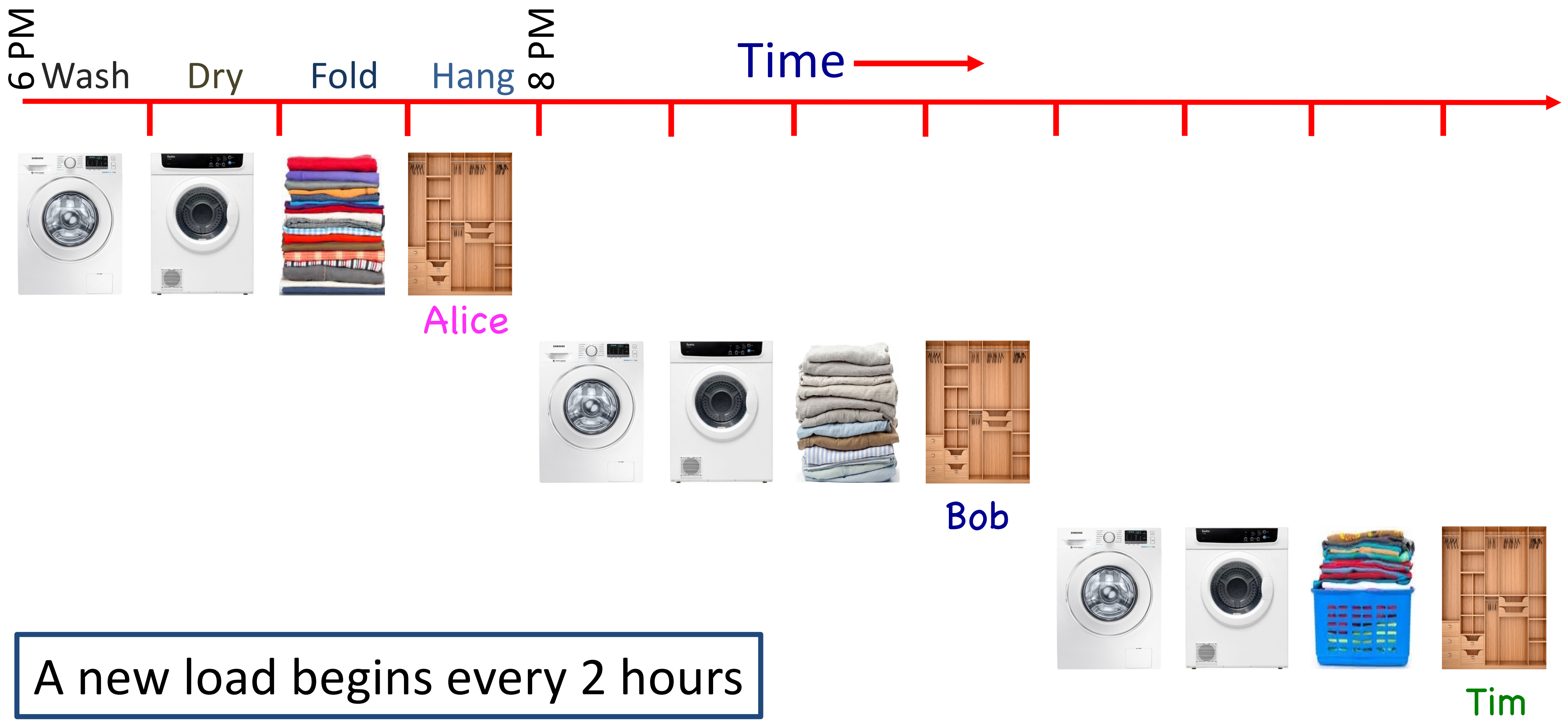
Latency (hours/tray):

Throughput (trays/hour):

Answers Explained

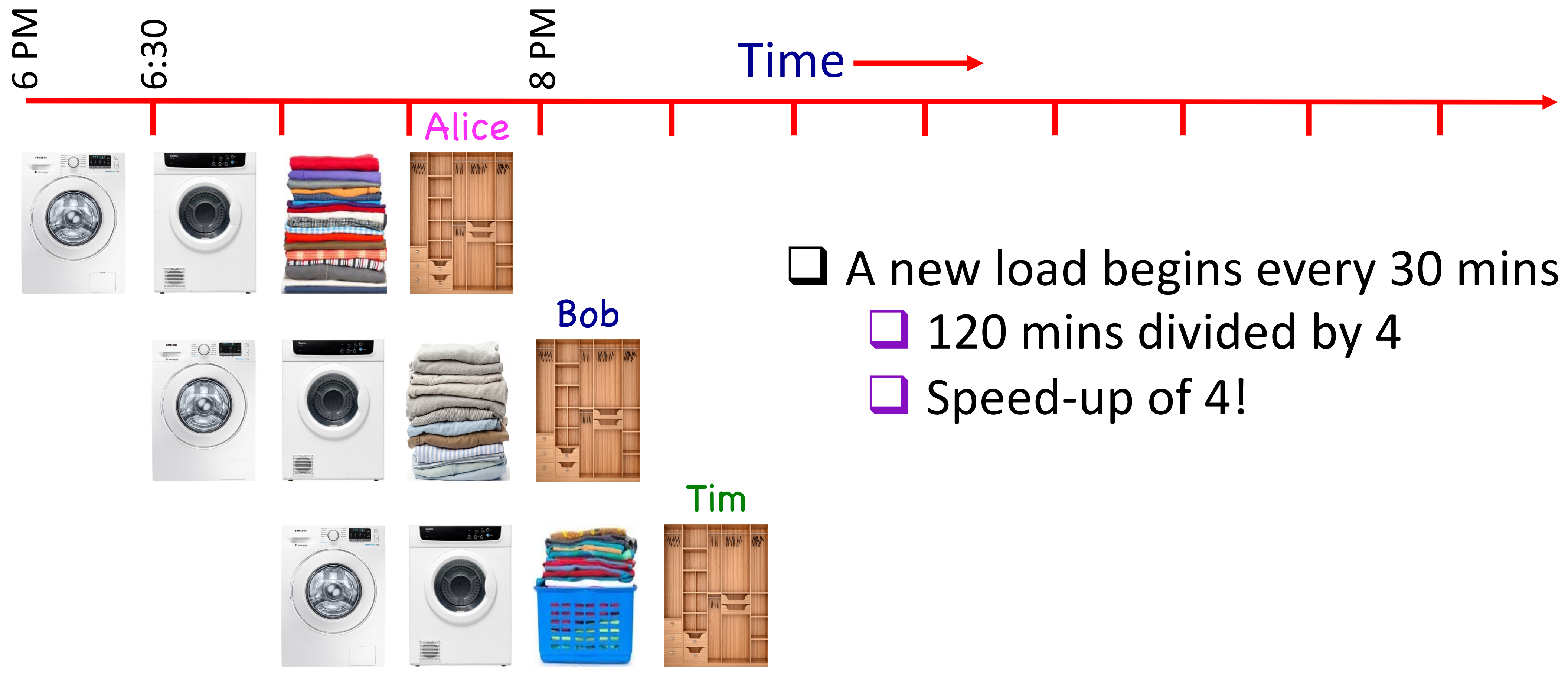
- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a single tray remains unchanged
  - Throughput: Ben puts a new tray in the oven every 15 minutes, so the throughput is 4 trays per hour
  - Note that in the first hour, Ben loses 5 minutes to fill the pipeline
- **Spatial parallelism + pipelining**
  - Latency remains unchanged
  - Throughput: Ben & Jon combo puts two trays in the oven every 15 minutes, so the throughput is 8 trays per hour

Sequential Laundry



A new load begins every 2 hours

Pipelined Laundry



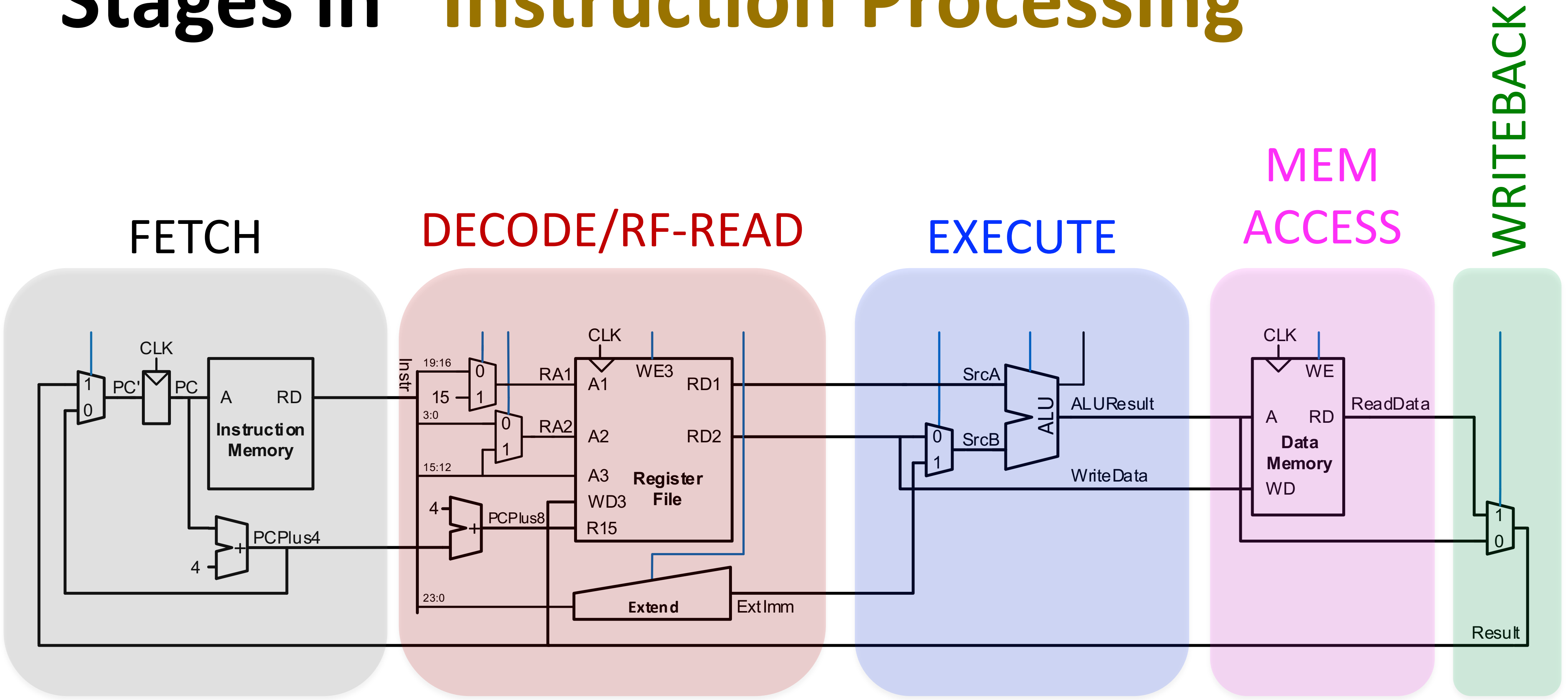
- A new load begins every 30 mins
  - 120 mins divided by 4
  - Speed-up of 4!

Recall: Pipelining Circuits

- Divide a large combinational circuit into shorter stages
- Insert registers between the stages
  - The outputs of one stage are copied into a register and communicated to the next stage
- Run the **pipelined** circuit at a **higher** clock frequency
  - Each clock cycle, data flows through the pipeline from left to the right
  - Multiple tasks can be spread across the pipeline

Pipelined Microarchitecture

Stages in "Instruction Processing"



Pipelined Microarchitecture: Key Idea

- Multiple instructions (up to 5) can be in the pipeline in any cycle
- Each instruction can be in a different stage
  - Idea is for **"maximizing utilization"** of hardware resources
- Stages must be isolated from one another using pipelined register (non-arch. registers). Referred to as "PPR"
- The work of a stage should be preserved in a PPR each cycle

Key Idea (Continued)

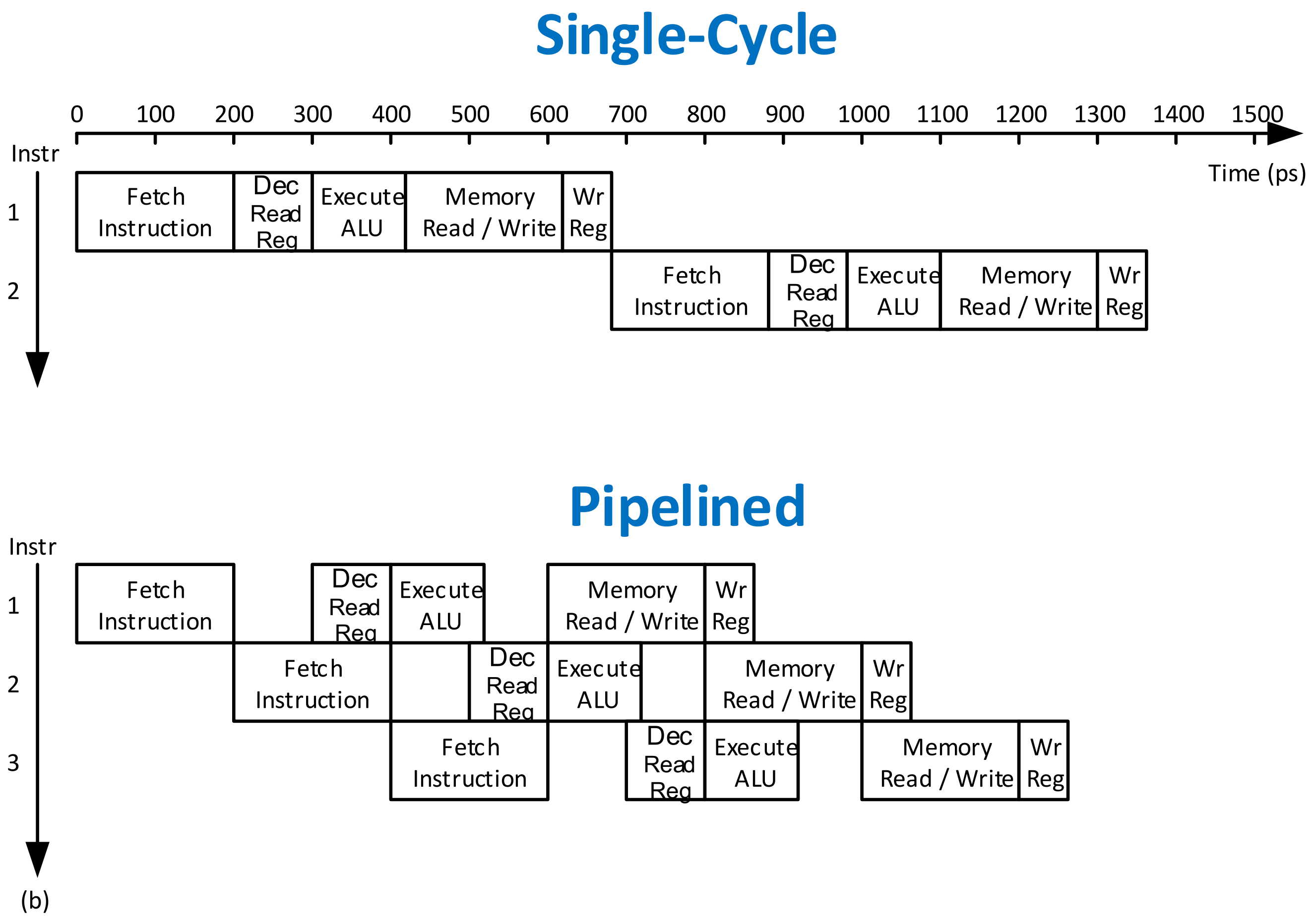
- The work of a stage should be preserved in a PPR each cycle
- PPR acts as a source of data the next stage needs in a subsequent cycle
- If any subsequent stage down the pipeline needs data from an earlier stage it must be passed through the PPRs
  - .... Things don’t always go smoothly as we shall see!

Timing Diagrams

- To visualize the execution of many instructions in a pipeline we can use timing diagrams where:
  - Time is on the horizontal axis
  - Instructions are on the vertical axis

Timing Diagrams

Assumption of logic element delays from Table 7.5 of textbook



Performance Analysis

- In the previous slide, what is the execution time and instructions per second (IPS) for the single-cycle microarchitecture?
  - 1.47 Billion Instructions per Second

- What about the pipelined microarchitecture?
  - The length of the pipeline stage is set by the slowest stage to be 200 ps
  - 1 instruction per 200 ps
  - 5 billion instructions per second

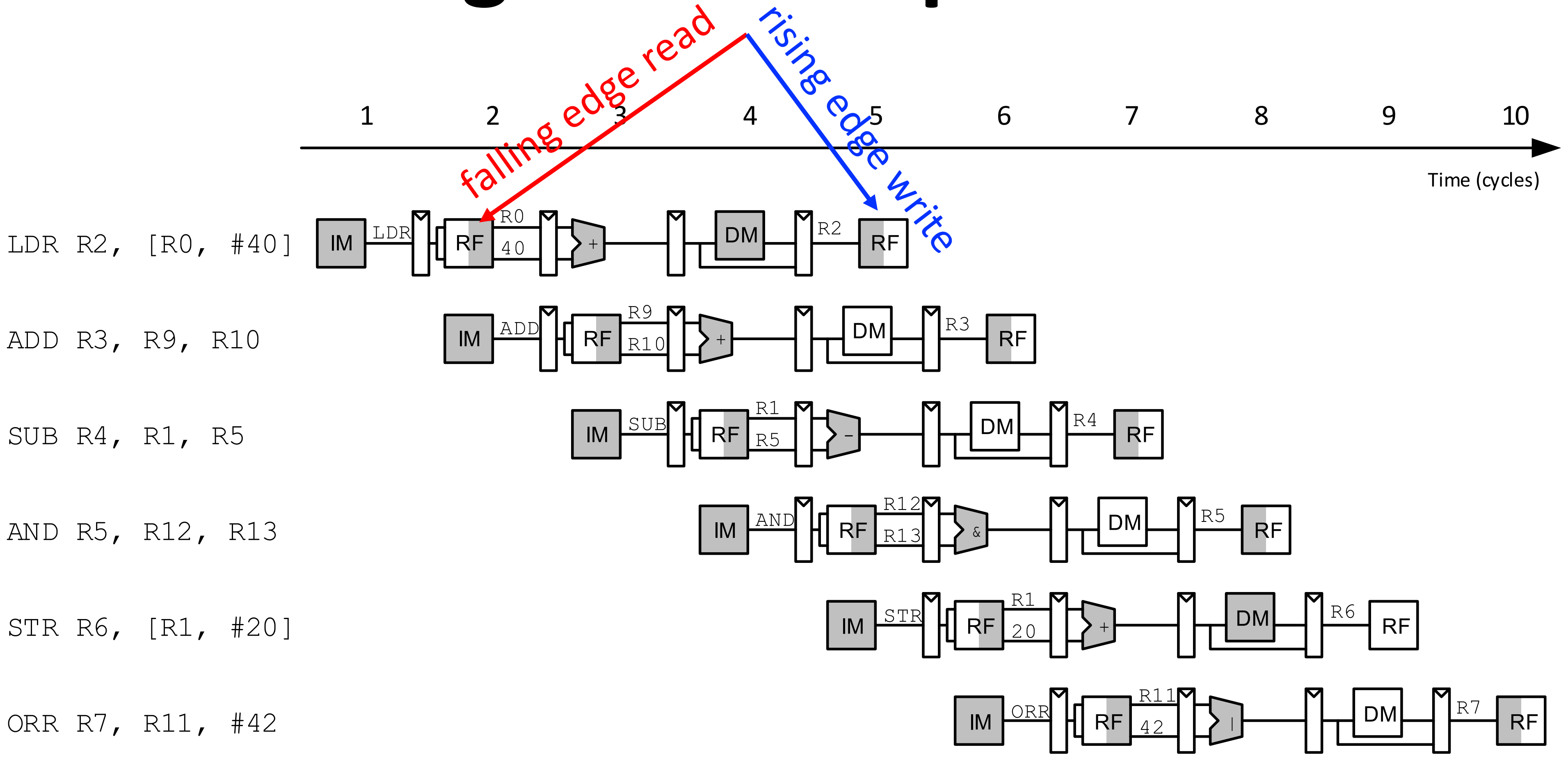
Instruction Latency with Pipelining

- Pipelining does not help to reduce the latency of a single instruction

- Latency of a single instruction increases
  - Sequencing overhead of pipeline registers
  - Clock cycle time decided by slowest pipeline stage (internal fragmentation due to imbalanced stages)

- Pipelining helps increase the throughput of an entire workload
  - Workload = Number of instructions
  - Workload must be **"sufficiently"** large

Abstract Diagrams of Pipelined uArch



Key idea: **In one cycle, an instruction writeback can be visible to a younger instruction's reg read**

Simplified View of Pipelining

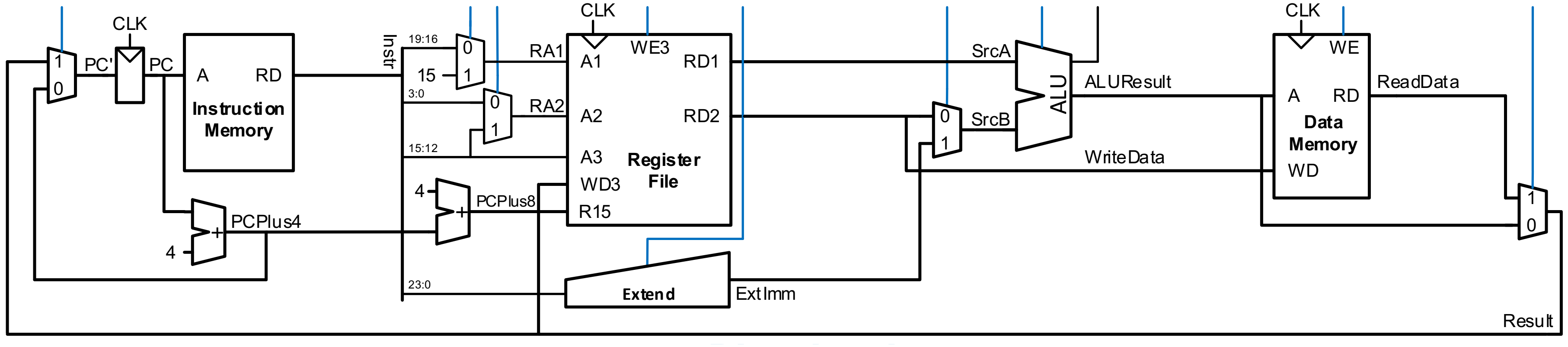
	C1	C2	C3	C4	C5	C6	C7	C8		
I1	F	D	E	M	W					
I2		F	D	E	M	W				
I3			F	D	E	M	W			
I4				F	D	E	M	W		
↓					F	D	E	M	W	
Insts						F	D	E	M	W

Let’s complete the picture

- Start with the single-cycle microarchitecture

- And insert pipeline registers

Single-Cycle

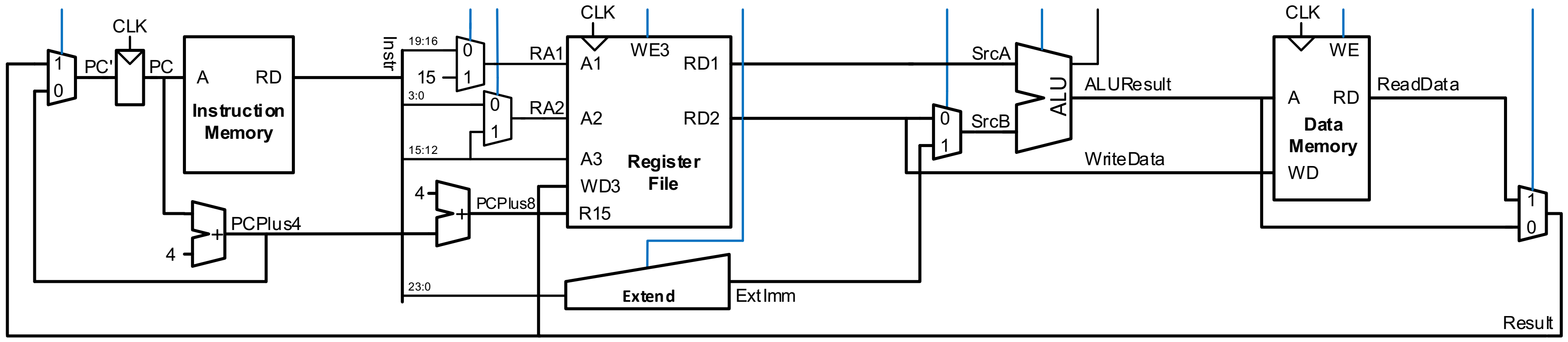


- Once we insert pipeline registers, we would need to pass the results of one stage to the next stage via the pipeline registers

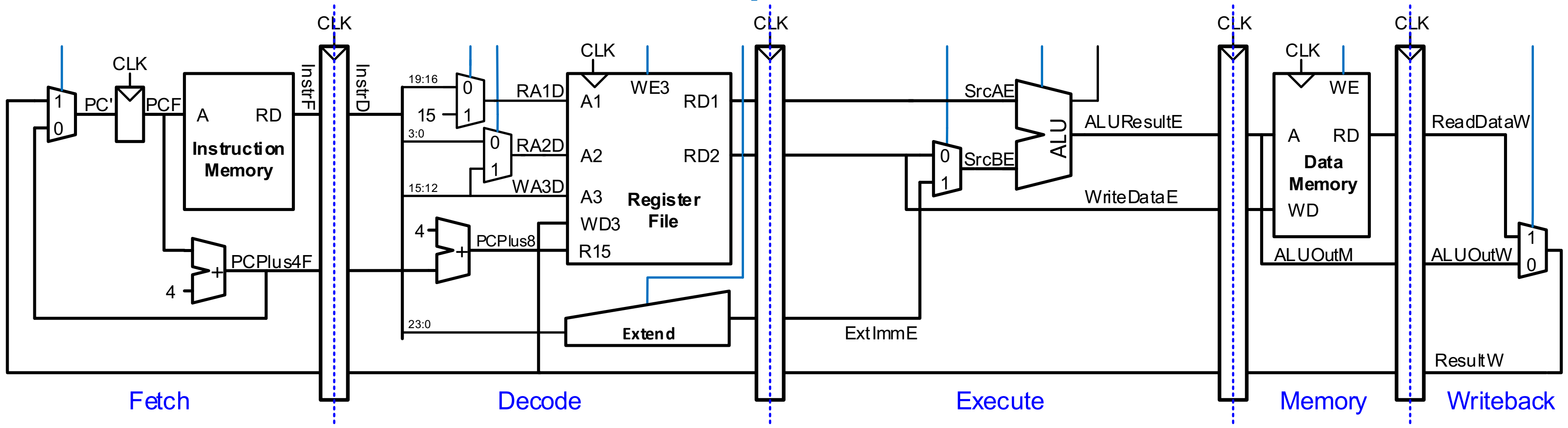
- What is the outcome of the FETCH stage?

Pipeline Microarchitecture

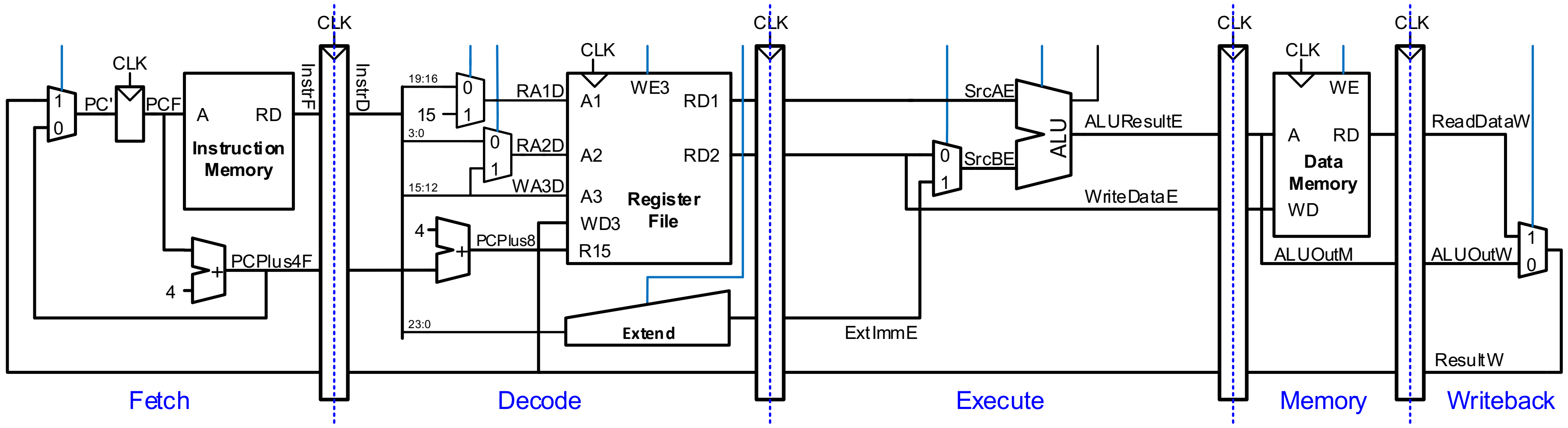
Single-Cycle



Pipelined



Pipeline Microarchitecture



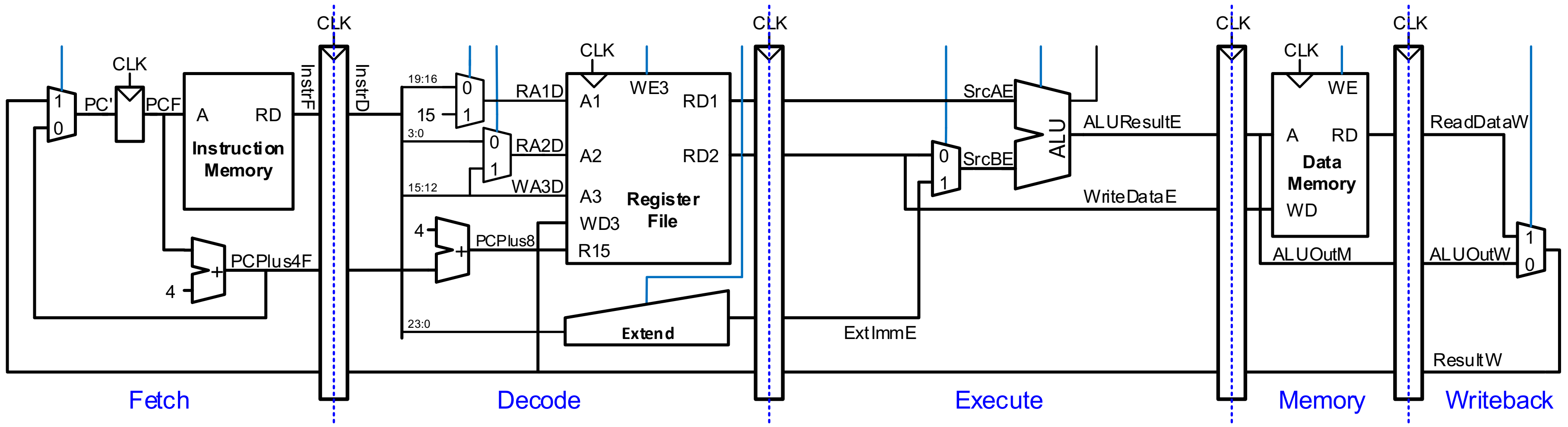
- Stages and their boundaries are indicated in blue
- Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside

Pipeline Operation

- Consider the example instruction sequence

```
I1:  ADD  R0,  R5,  #10
I2:  ADD  R1,  R5,  #10
I3:  ADD  R2,  R5,  #10
I4:  STR  R0,  [R7, #4]
I5:  STR  R1,  [R7, #8]
I6:  STR  R2,  [R7, #12]
```

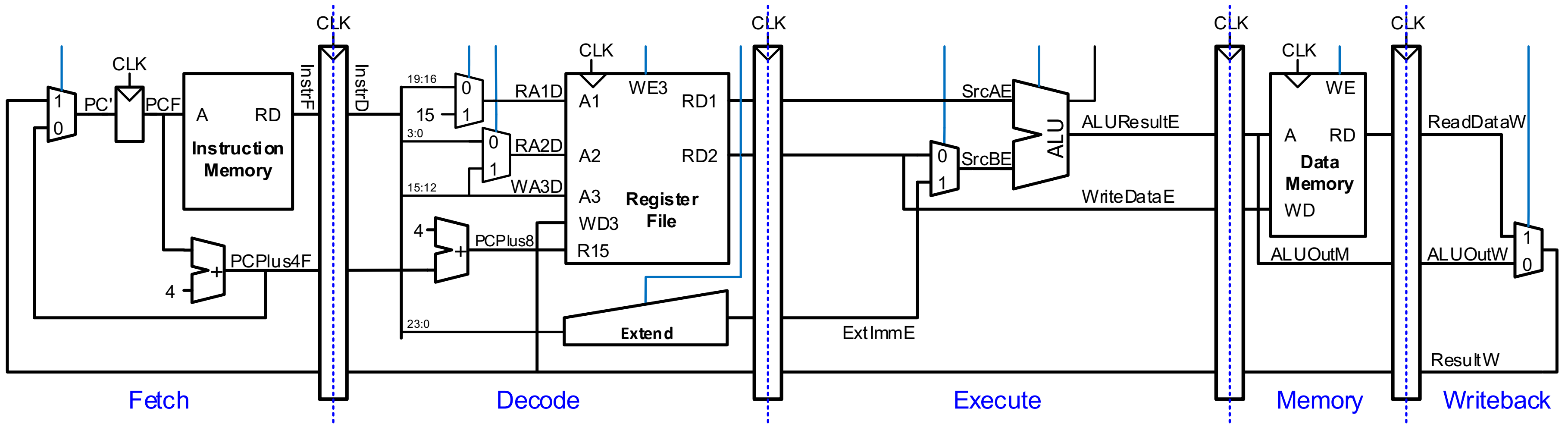
Pipeline Operation: Cycle 1



I1

- Is the pipeline fully utilized? NO

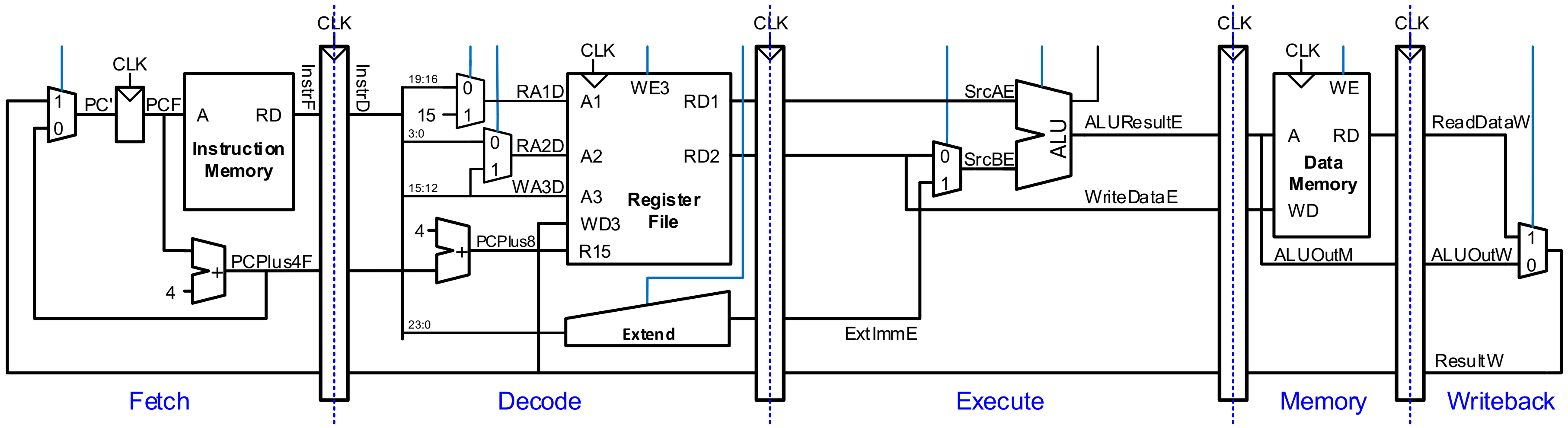
Pipeline Operation: Cycle 2



I2 I1

- ❑ Is the pipeline fully utilized? NO

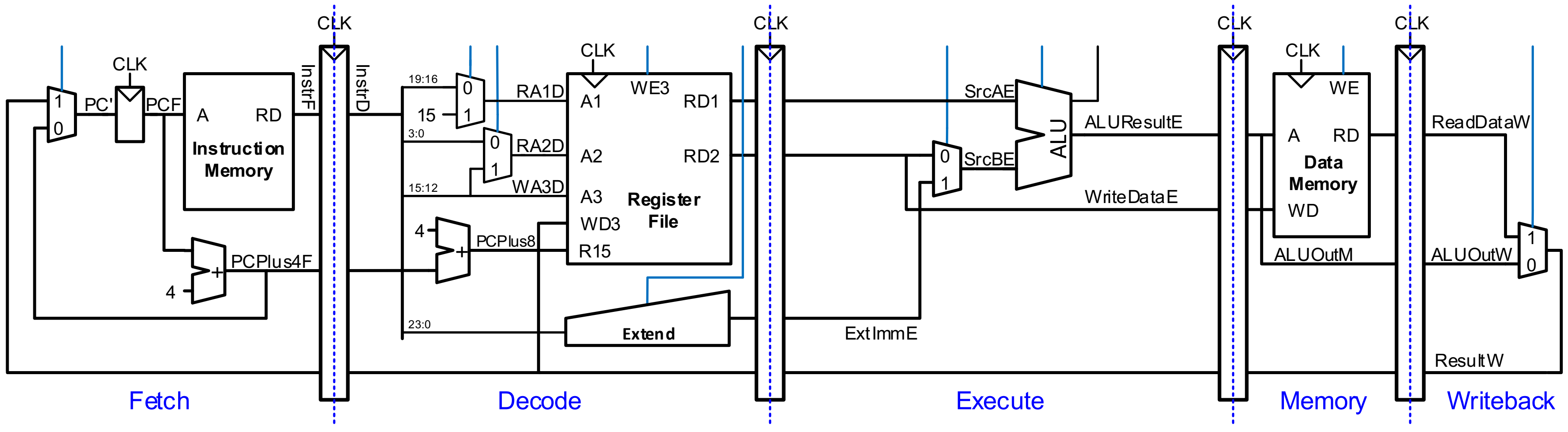
Pipeline Operation: Cycle 3



I3 I2 I1

❑ Is the pipeline fully utilized? NO

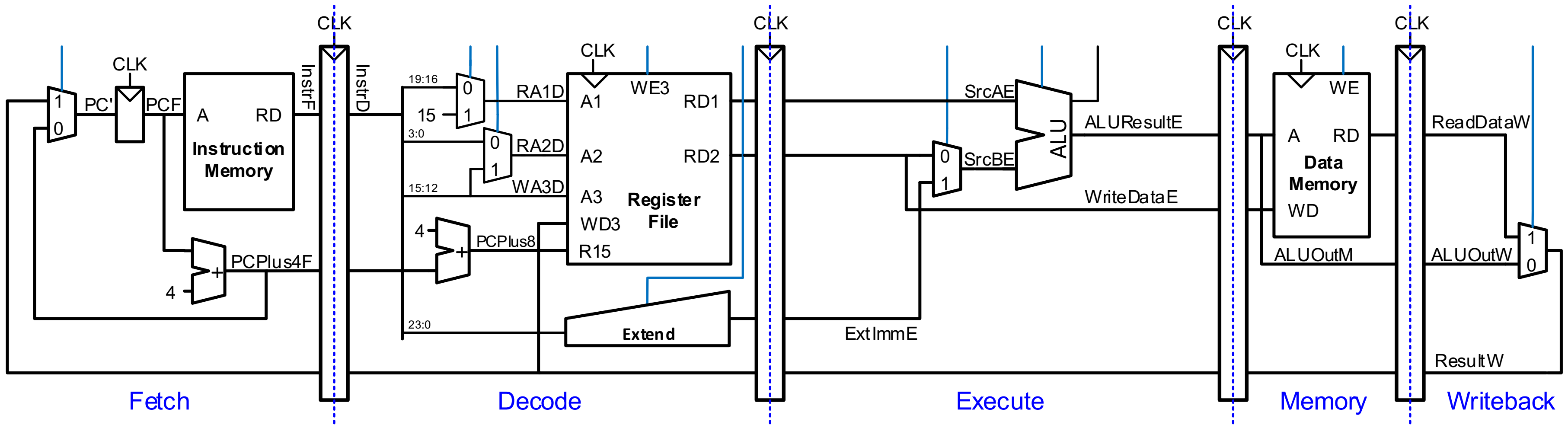
Pipeline Operation: Cycle 4



I4 I3 I2 I1

❑ Is the pipeline fully utilized? NO

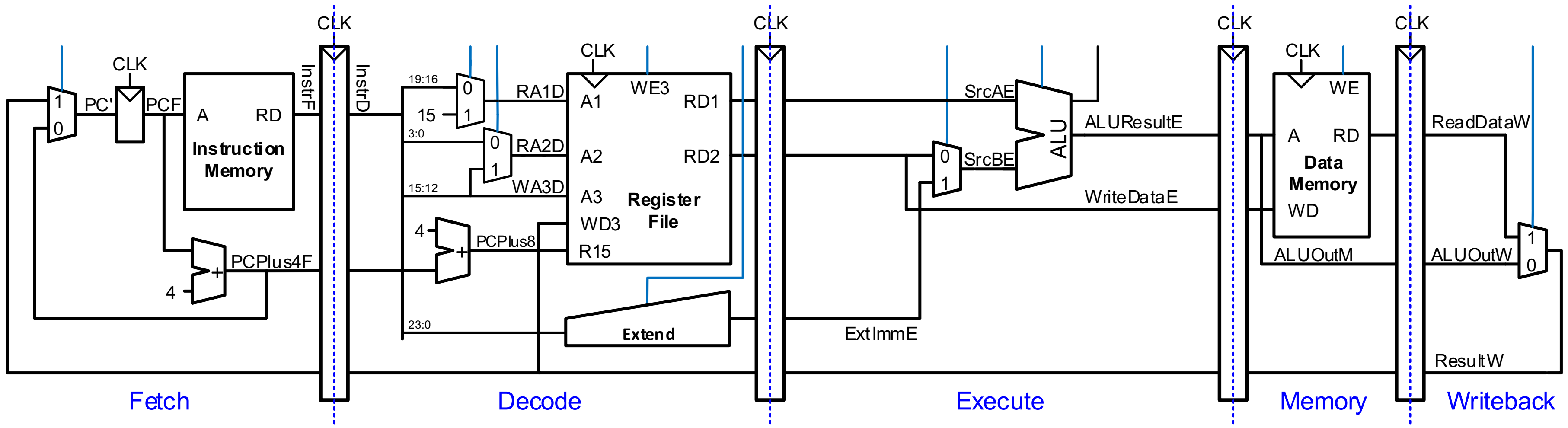
Pipeline Operation: Cycle 5



I5 I4 I3 I2 I1

- ❑ Is the pipeline fully utilized? YES

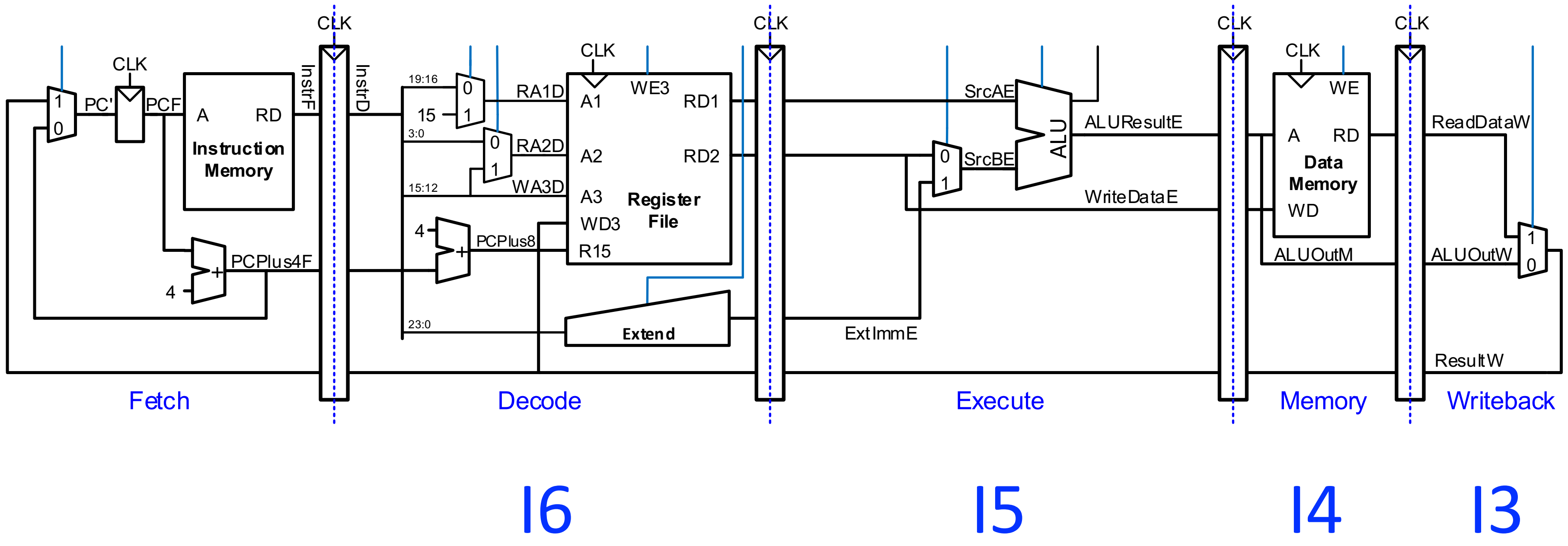
Pipeline Operation: Cycle 6



I6 I5 I4 I3 I2

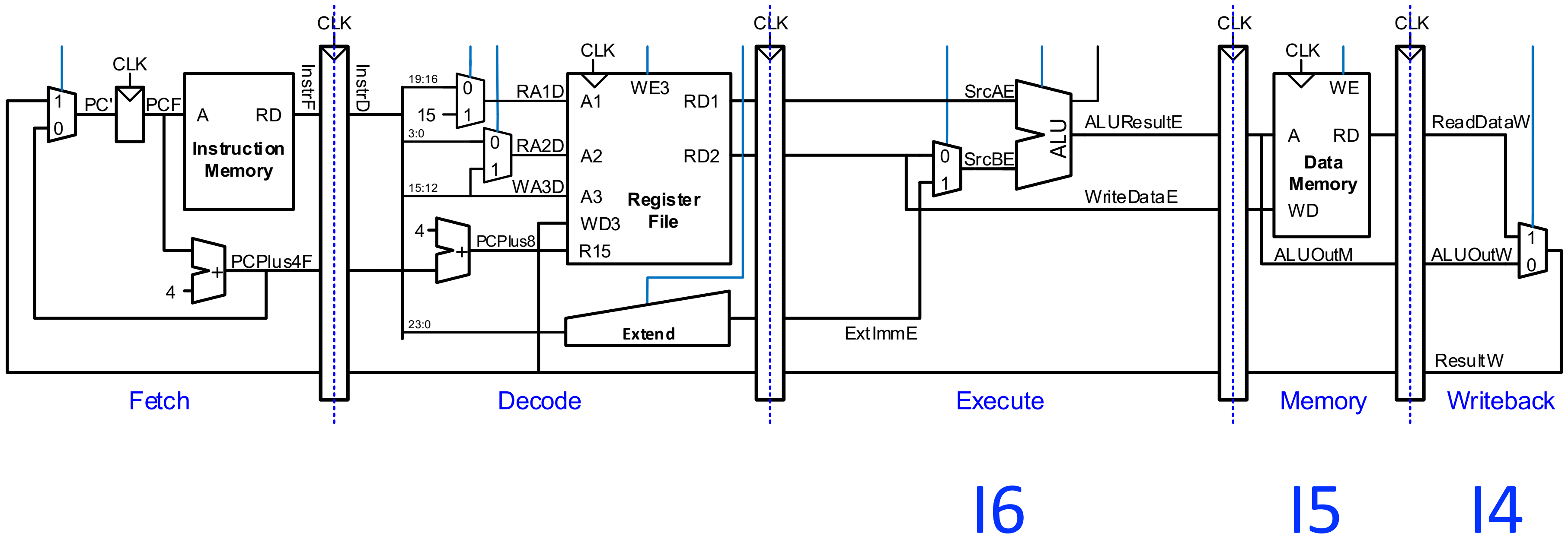
- Is the pipeline fully utilized? YES

Pipeline Operation: Cycle 7



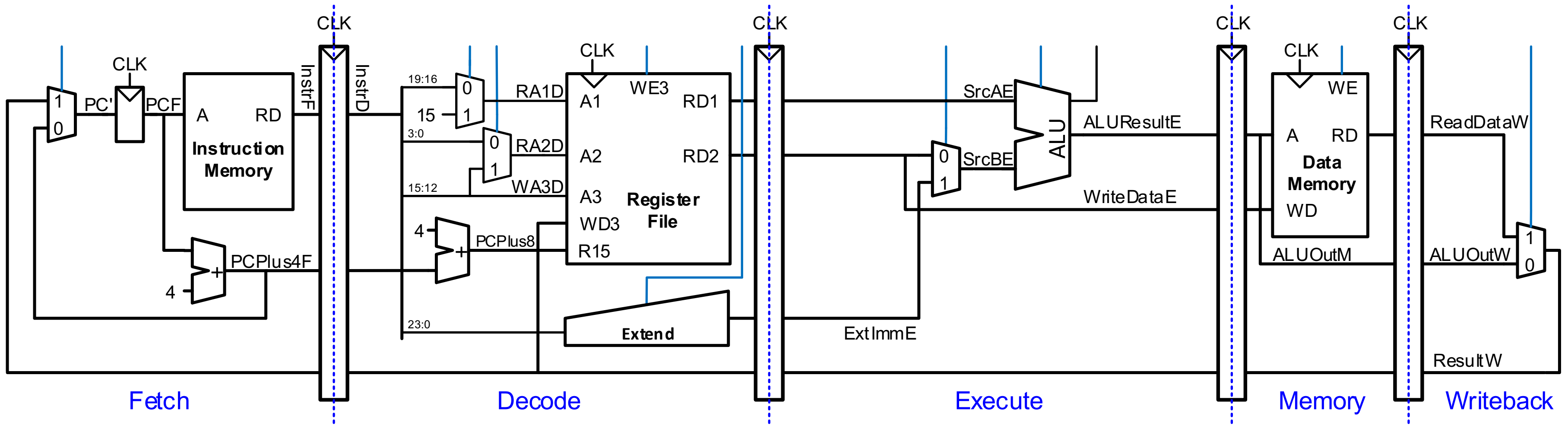
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 8



❑ Is the pipeline fully utilized? NO

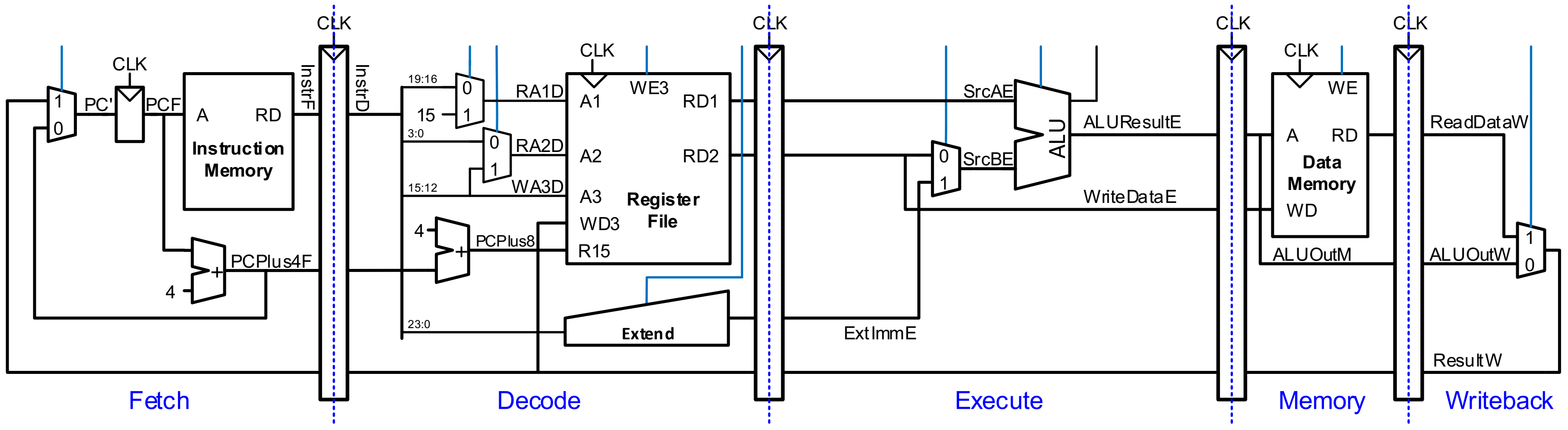
Pipeline Operation: Cycle 9



I6 I5

- ❑ Is the pipeline fully utilized? NO

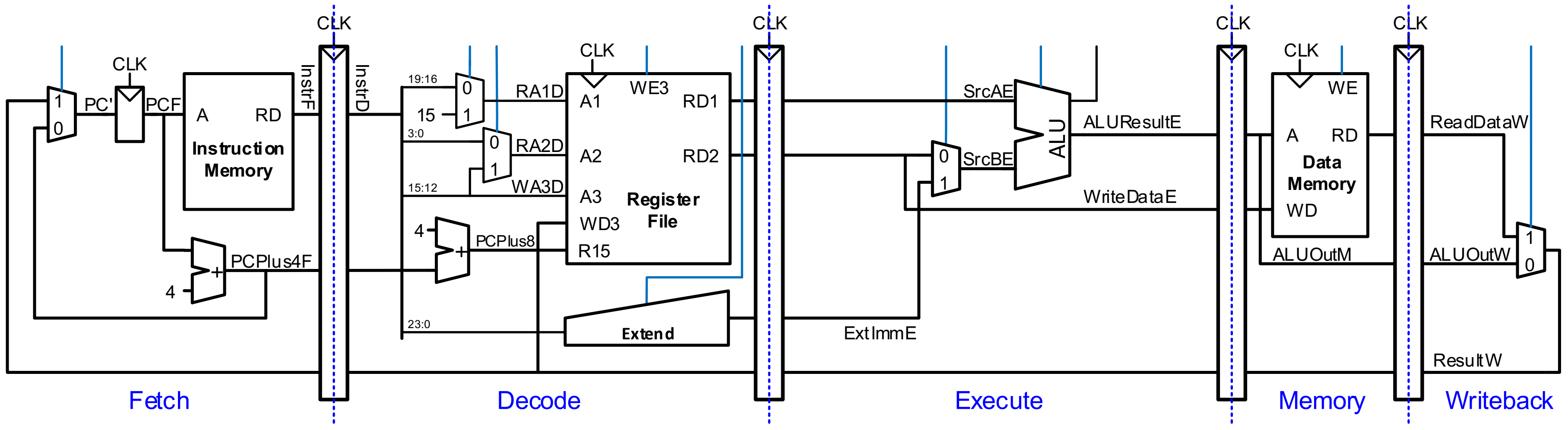
Pipeline Operation: Cycle 10



I6

- Is the pipeline fully utilized? NO

Pipeline Operation



- ❑ No more instructions to execute

Performance Analysis

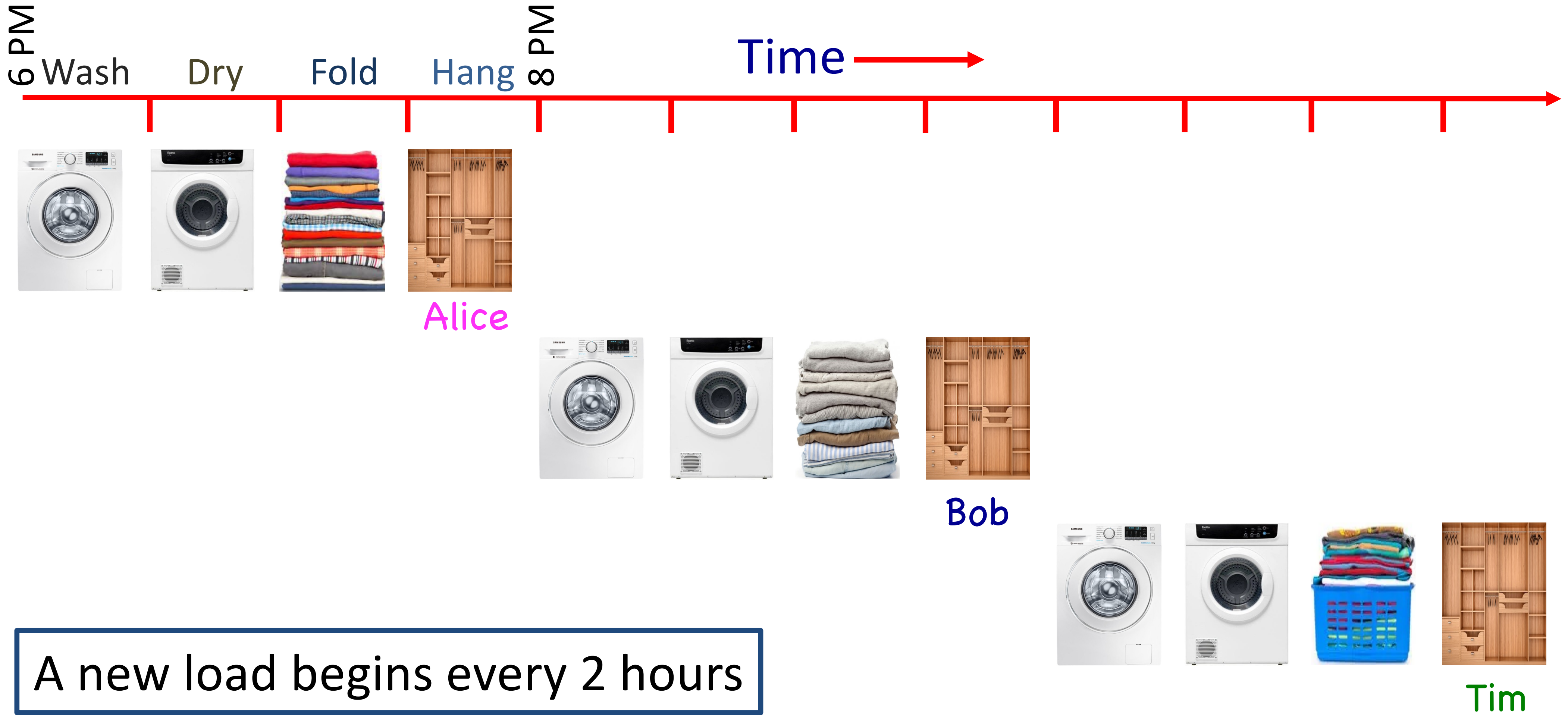
- The **6 instructions** took **10 cycles** to finish execution

- Cycles per Instruction (CPI) is : 10/6 = 1.66
  - Conversely, instruction per cycle (IPC) is: 0.6

- **Ideally,** we want the IPC to be close to 1
  - **One instruction finished every cycle**

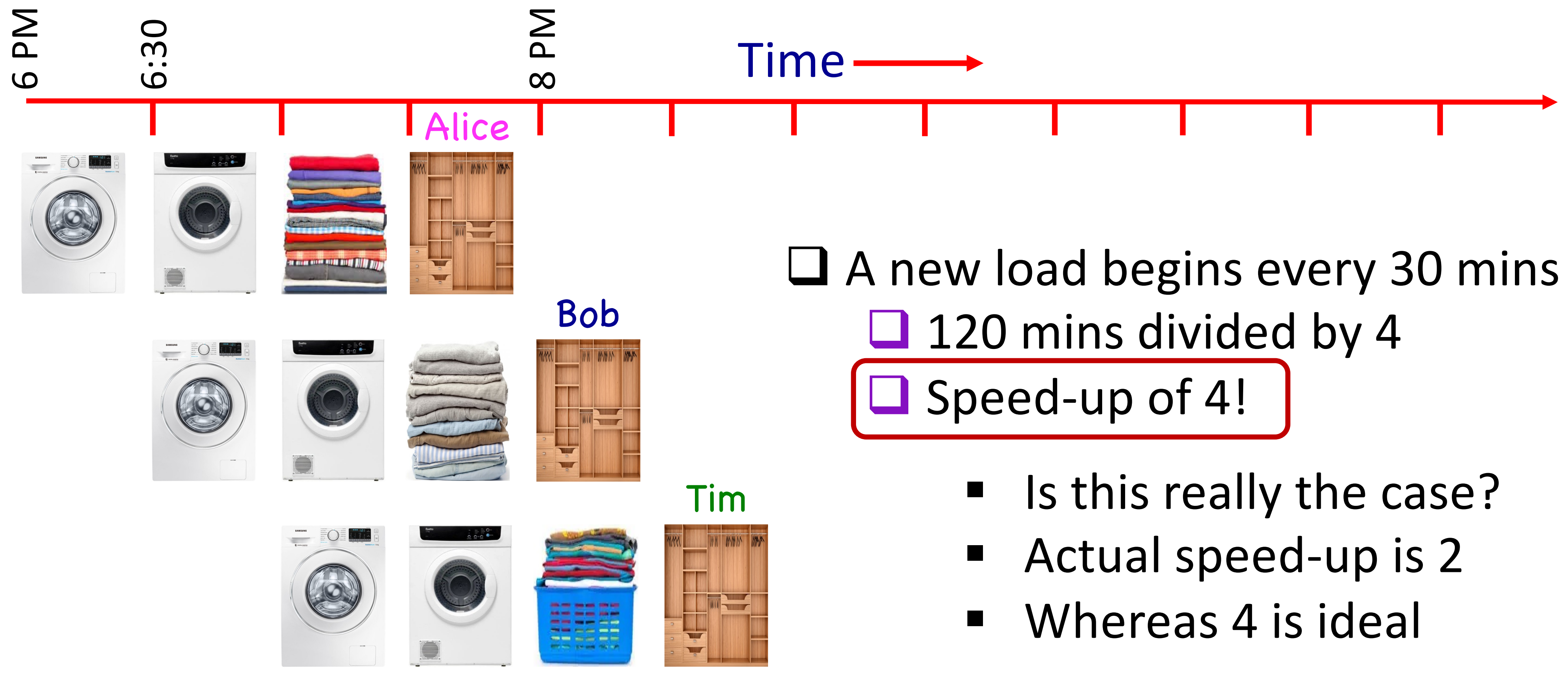
- Why is IPC less than 1?
  - It takes some time to fill and some time to drain the pipeline
    - During this time pipeline is operating below its potential

Sequential Laundry



A new load begins every 2 hours

Recall: Pipelined Laundry



- A new load begins every 30 mins
  - 120 mins divided by 4
  - Speed-up of 4!
    - Is this really the case?
    - Actual speed-up is 2
    - Whereas 4 is ideal

Pipeline Idealism vs. Reality

- **Pipeline fill time:** The time it takes to fill the pipeline and make it operate at maximum efficiency
- **Pipeline drain time:** The time that is wasted when there is no more work to do in the pipeline
- The two factors limit the pipeline from delivering ideal speed-up
  - In the case when the amount of work is small relative to the number of stages in the pipeline
- Let’s revisit the previous example

Performance Analysis

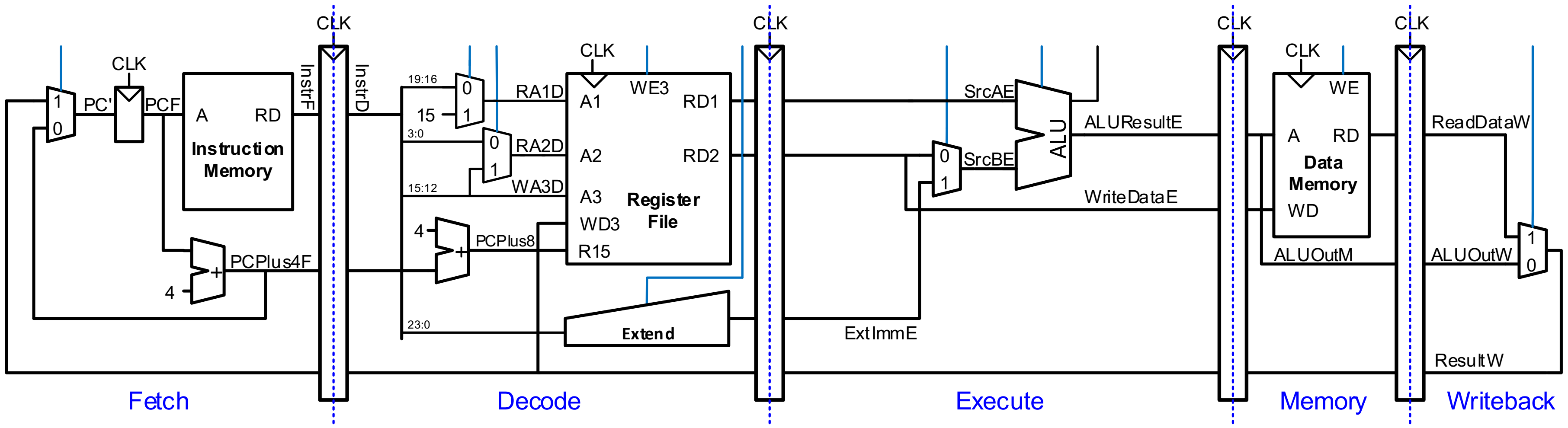
- The **6 instructions** took **10 cycles** to finish execution

- Cycles per Instruction (CPI) is : 10/6 = 1.66
  - Conversely, instruction per cycle (IPC) is: 0.6

- What if we have 1 billion instructions instead of 6?
  - CPI = (4 + 1000000000)/1000000000 = ~1

- Computer programs execute billions of instructions, so the overhead of filling/draining is amortized

Bug in Pipelined Hardware!



- There is a "hardware bug" in the pipelined microarchitecture
  - Can you spot it?