COMP2300-COMP6300-ENGN2219
Computer Organization &
Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian
National
University

First 44 slides from week 10 on pipelining included here for your review/convenience

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

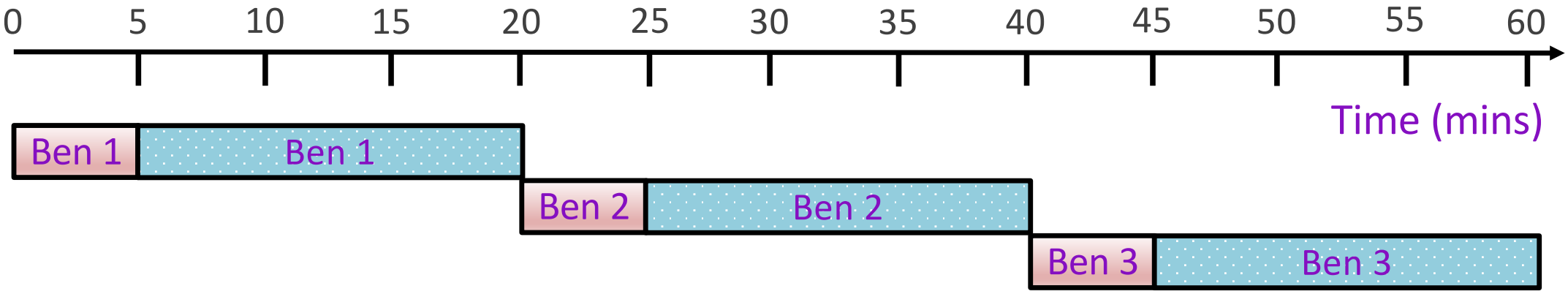
- The challenge of pipelining is to find stages of equal length

- 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

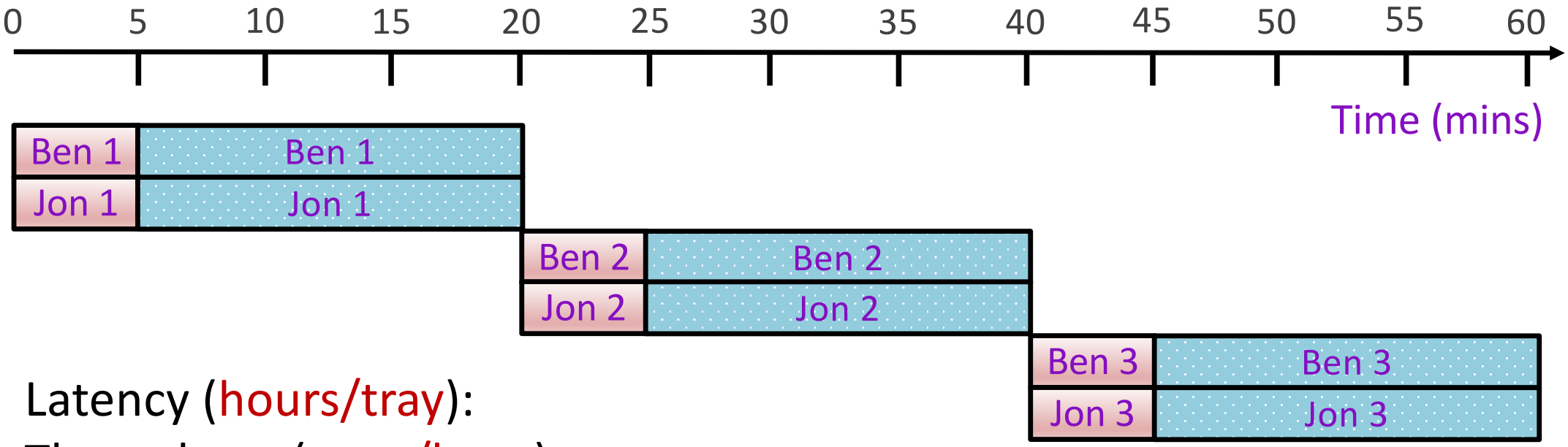
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

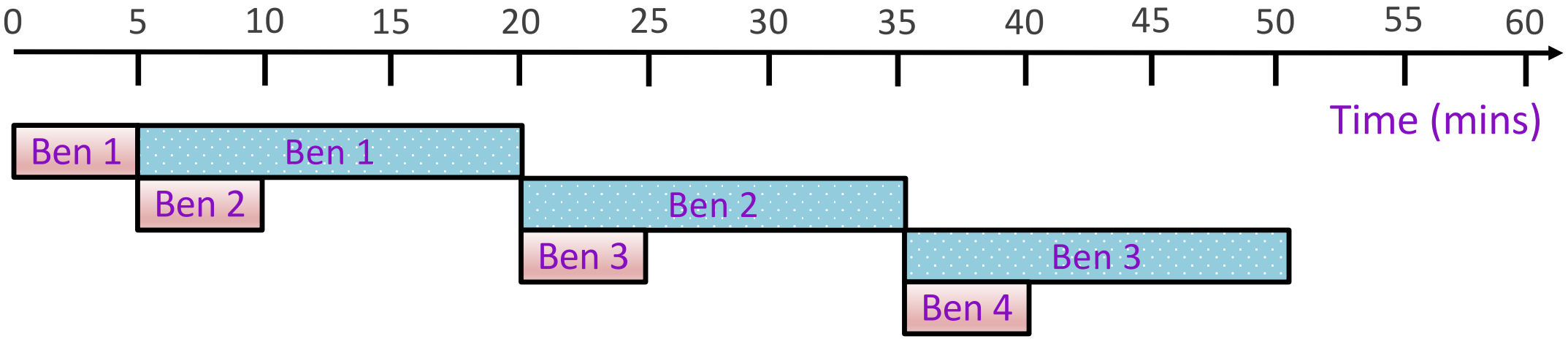


Latency (hours/tray):

Throughput (trays/hour):

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

Pipelining (Ben Only)

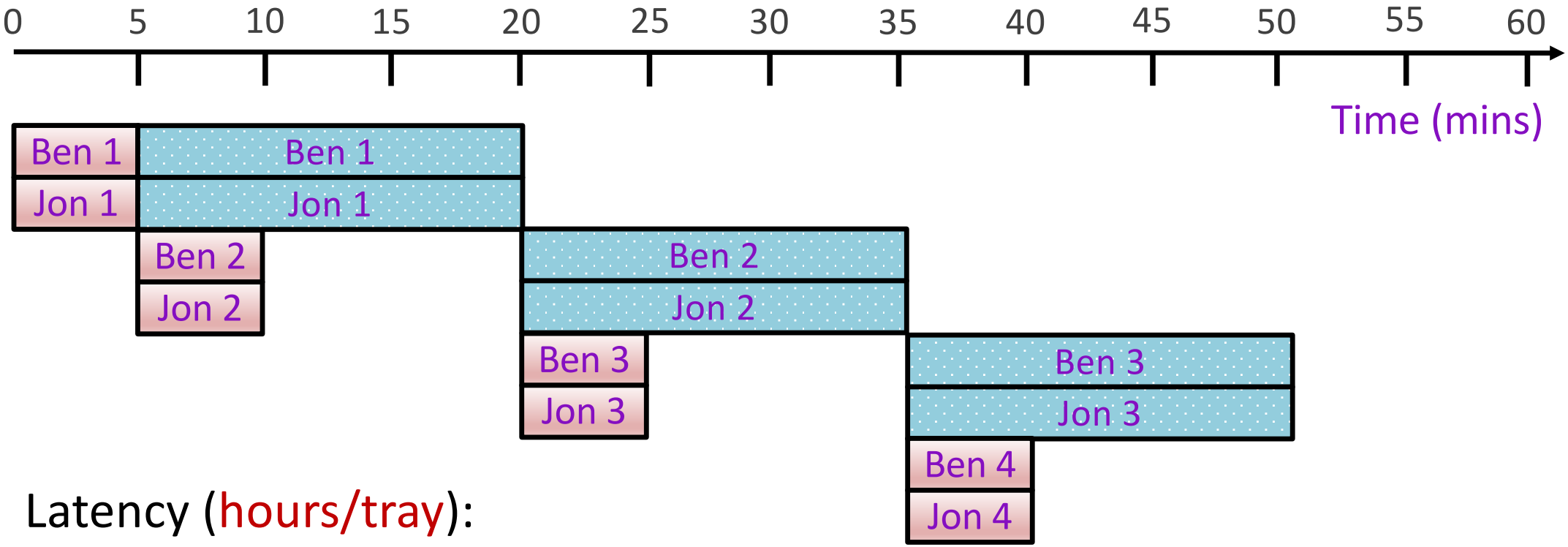


Latency (hours/tray):

Throughput (trays/hour):

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

Spatial + Temporal Parallelism



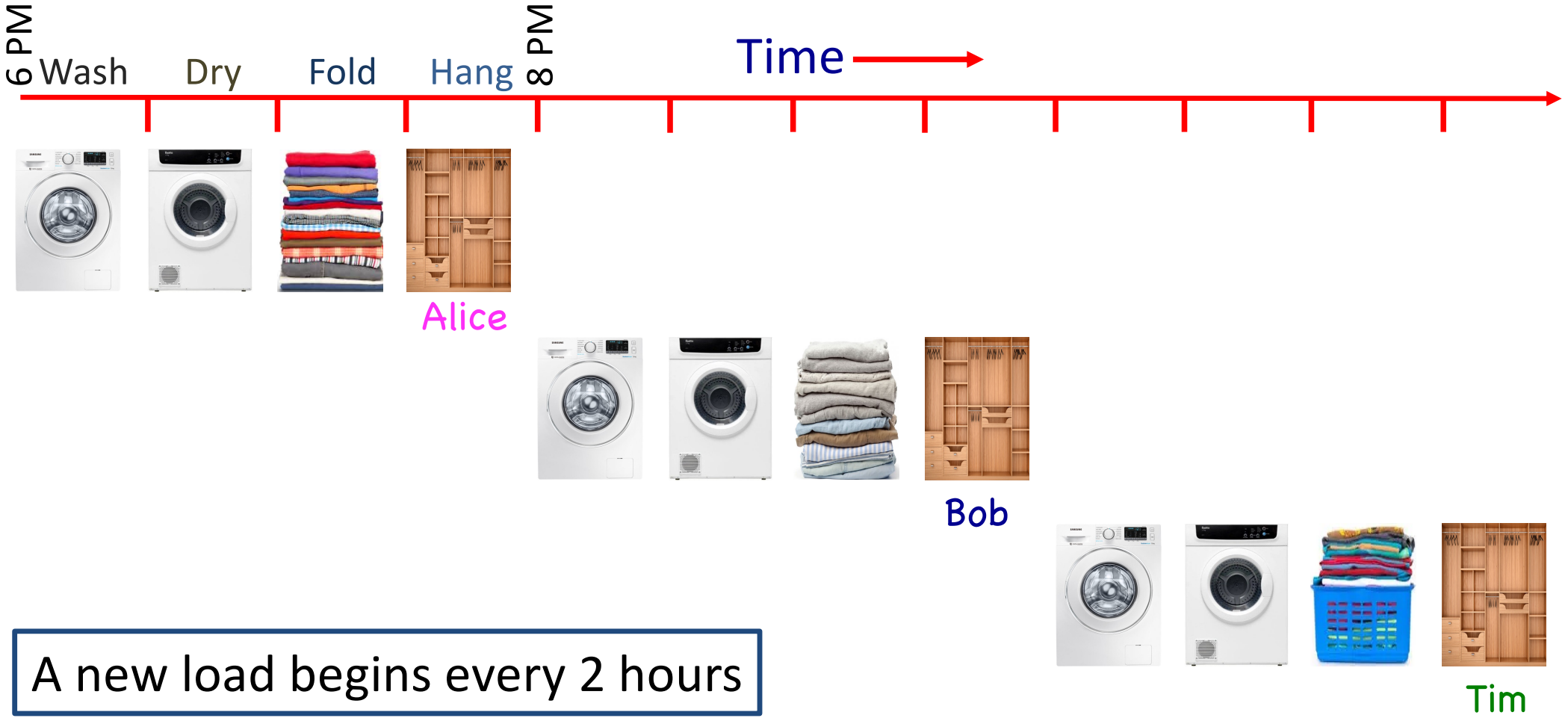
Latency (hours/tray):

Throughput (trays/hour):

Answers Explained

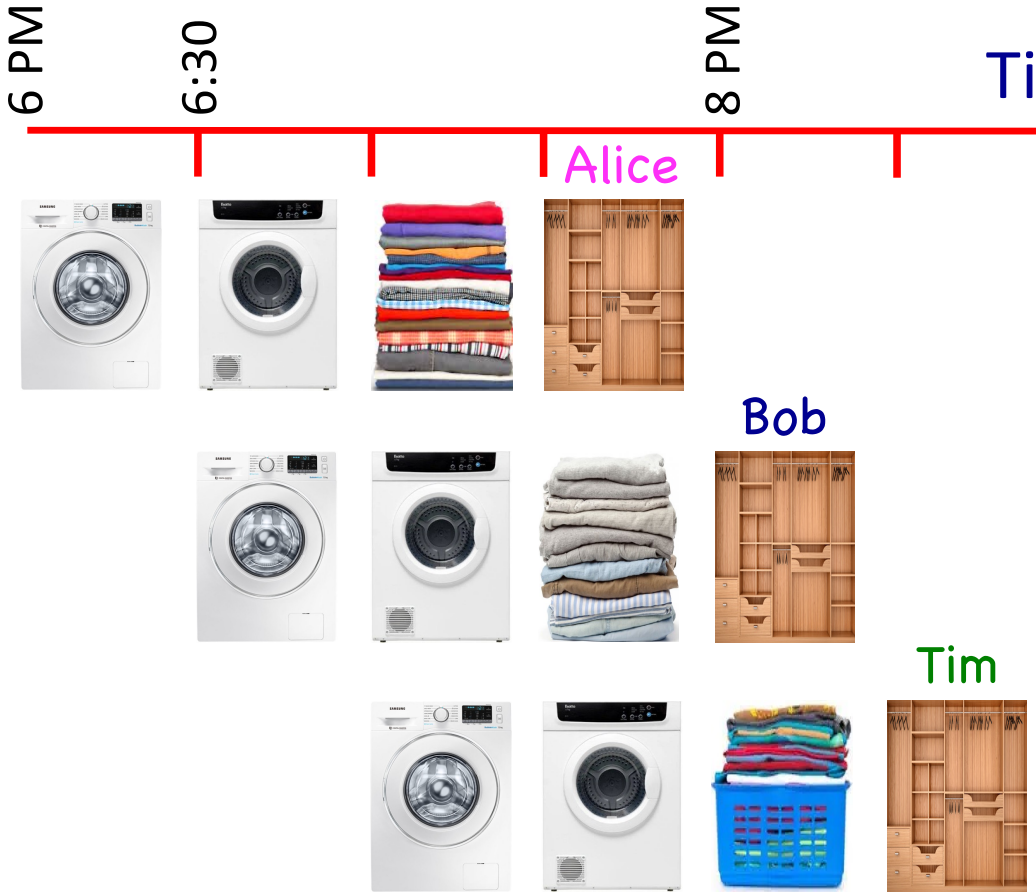
- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a 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



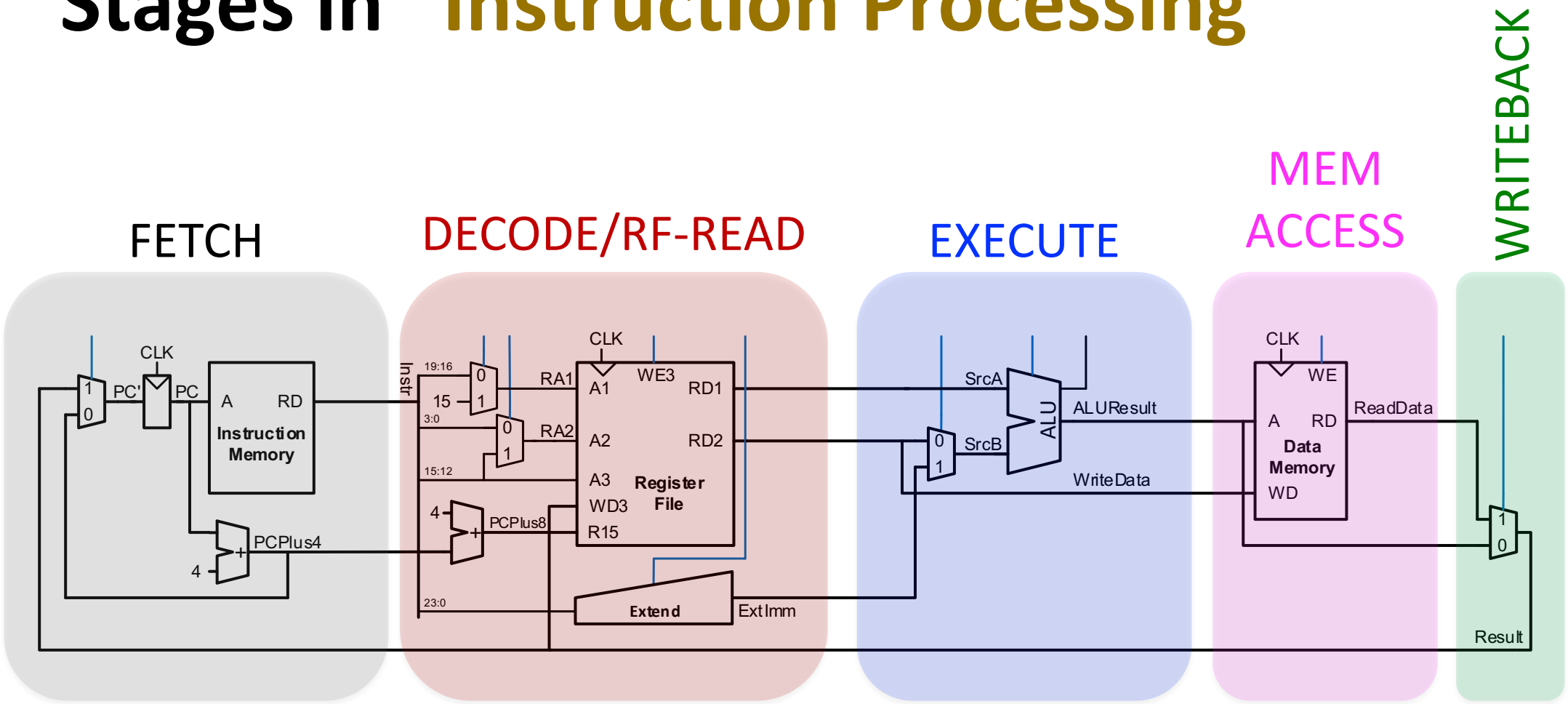
- A new load begins every 30 mins
  - 120 mins divided by 4
  - Speed-up of 4!

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!

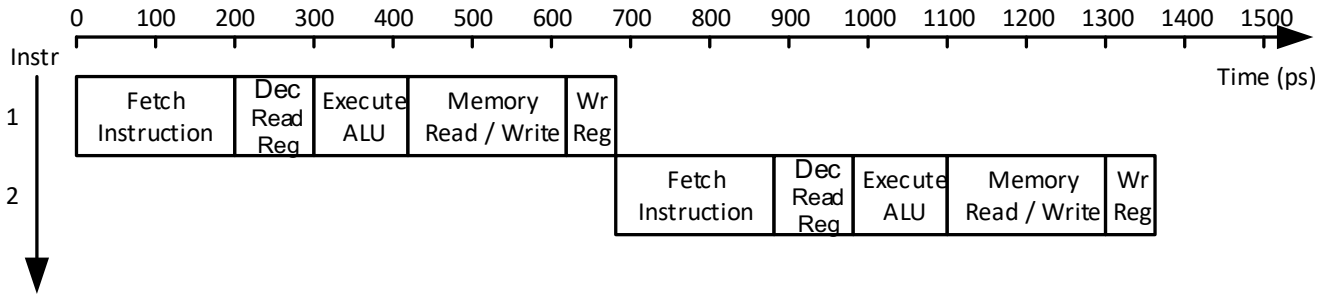
Timing Diagrams

- 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

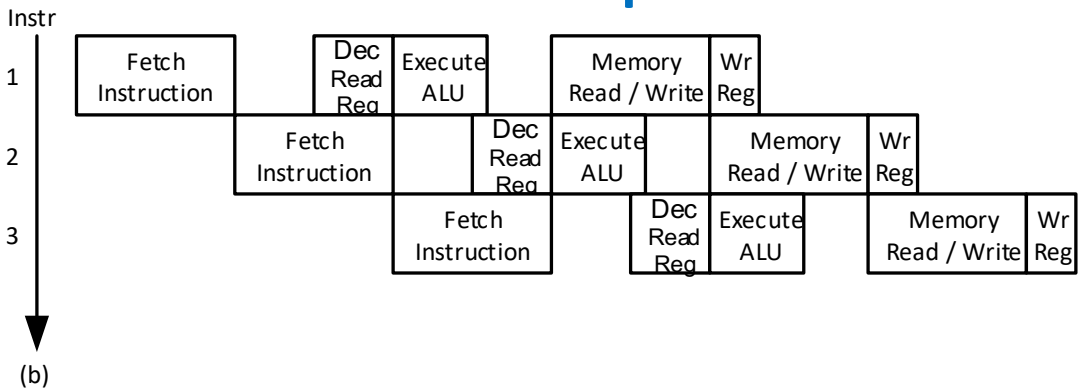
Timing Diagrams

Assumption of logic element delays from Table 7.5 of textbook

Single-Cycle



Pipelined



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

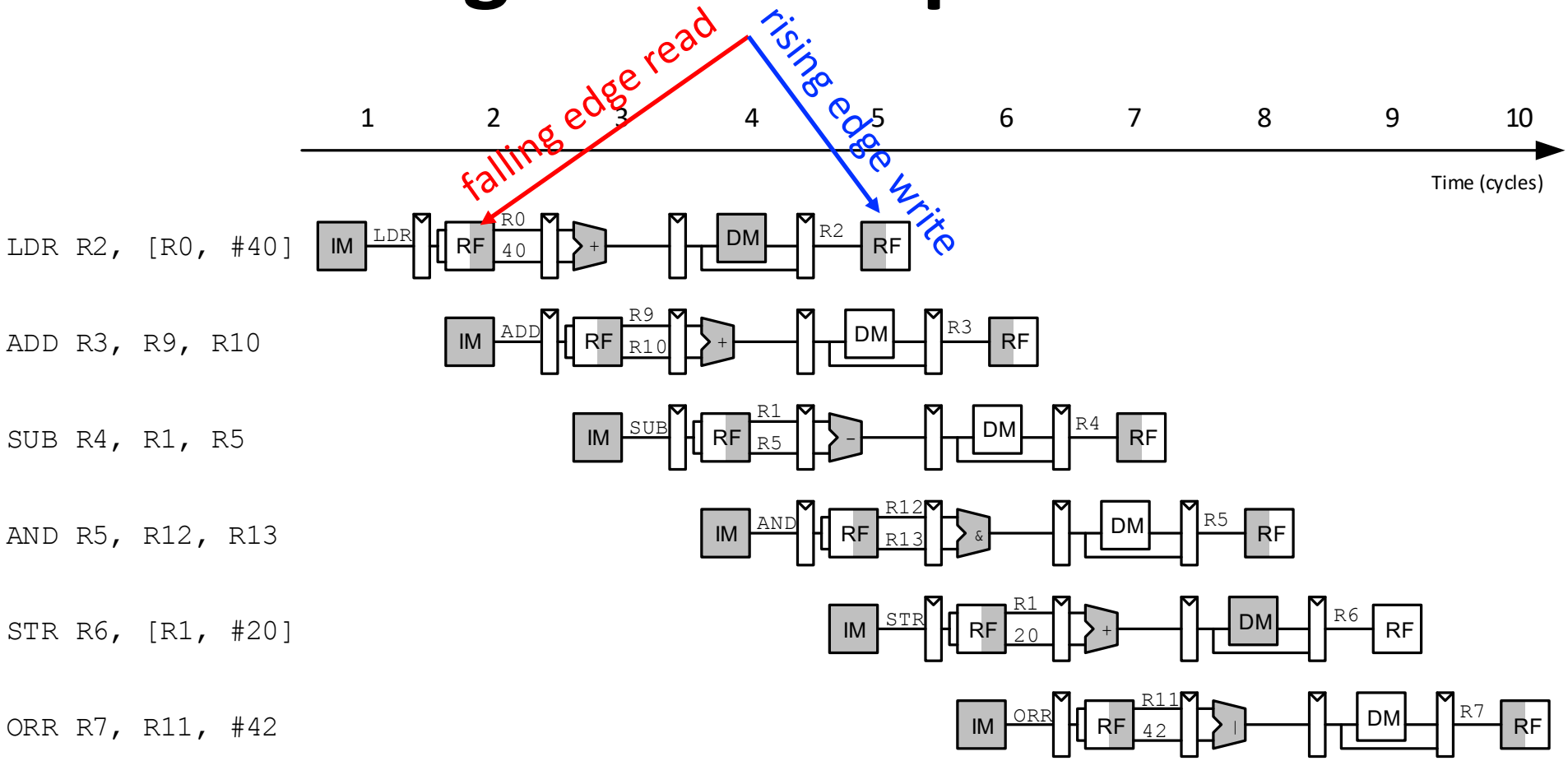
Instruction Latency with Pipelining

- 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)

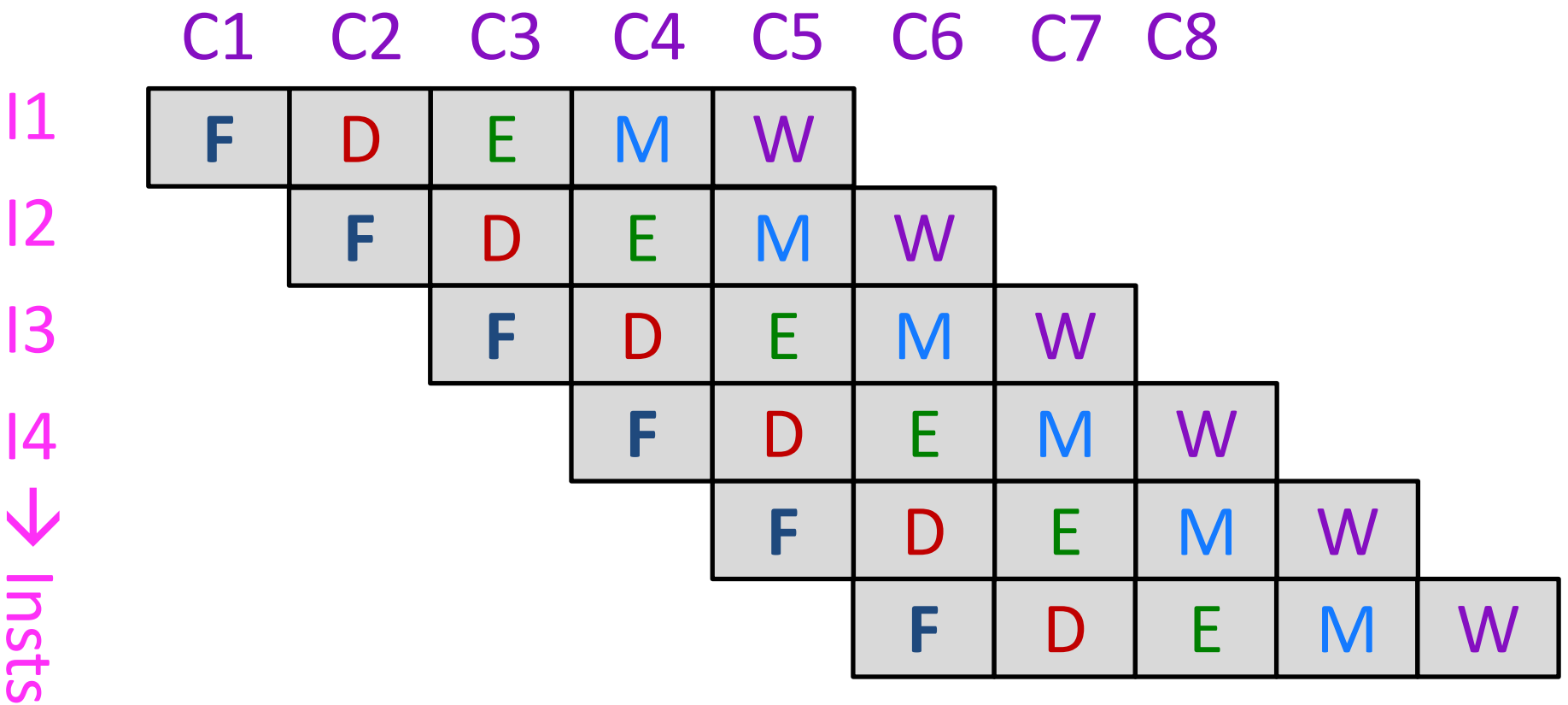
- Pipelining helps increase the throughput of an entire workload
  - Workload = Number of instructions
  - Workload must be **"sufficiently"** large

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

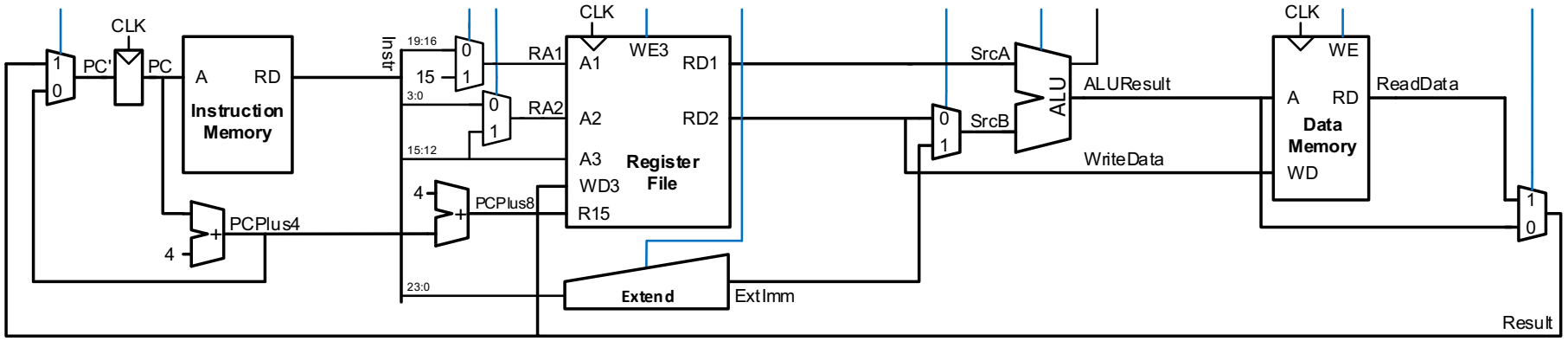


Let’s complete the picture

- Start with the single-cycle microarchitecture

- And insert pipeline registers

Single-Cycle

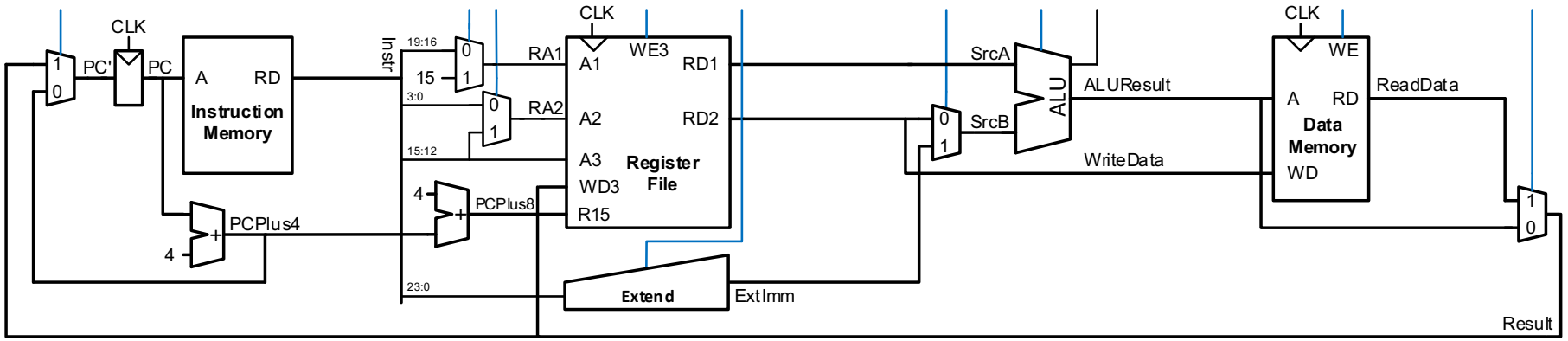


- Once we insert pipeline registers, we would need to pass the results of one stage to the next stage via the pipeline registers

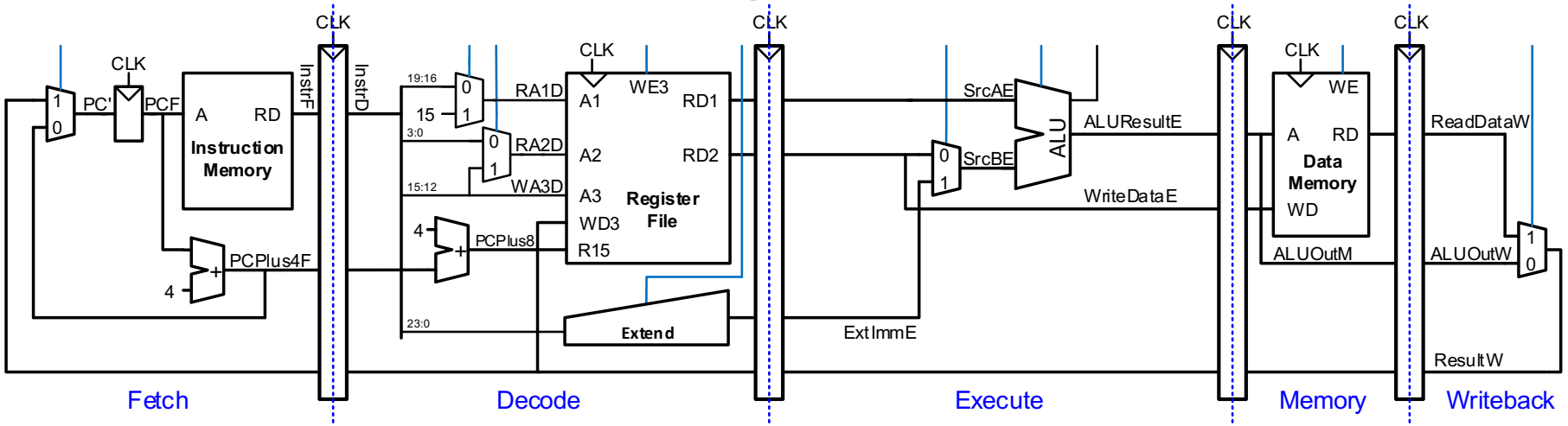
- What is the outcome of the FETCH stage?

Pipeline Microarchitecture

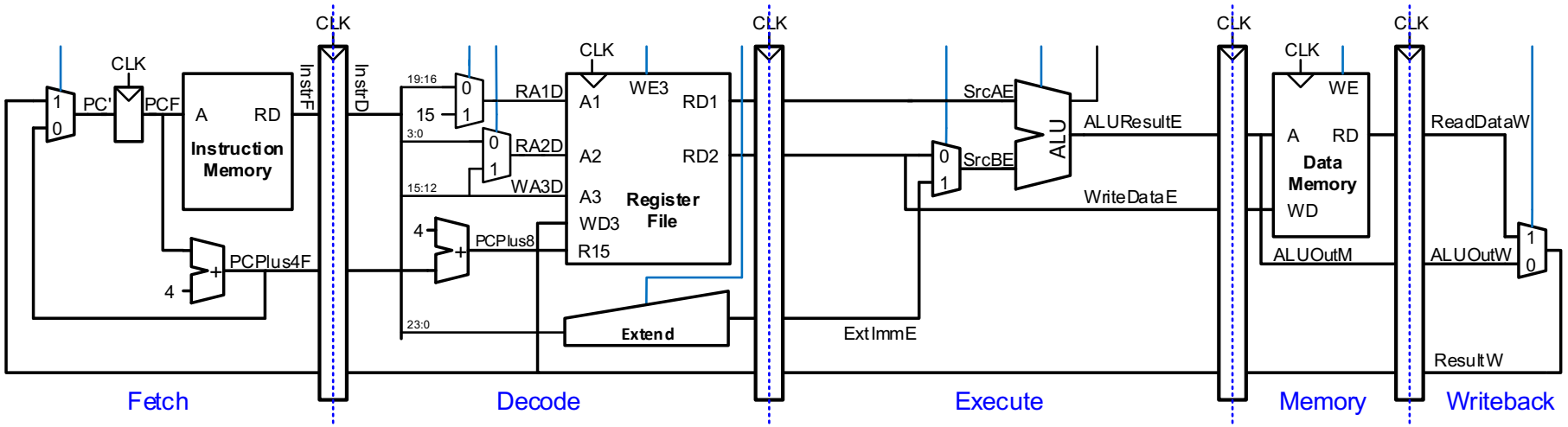
Single-Cycle



Pipelined



Pipeline Microarchitecture



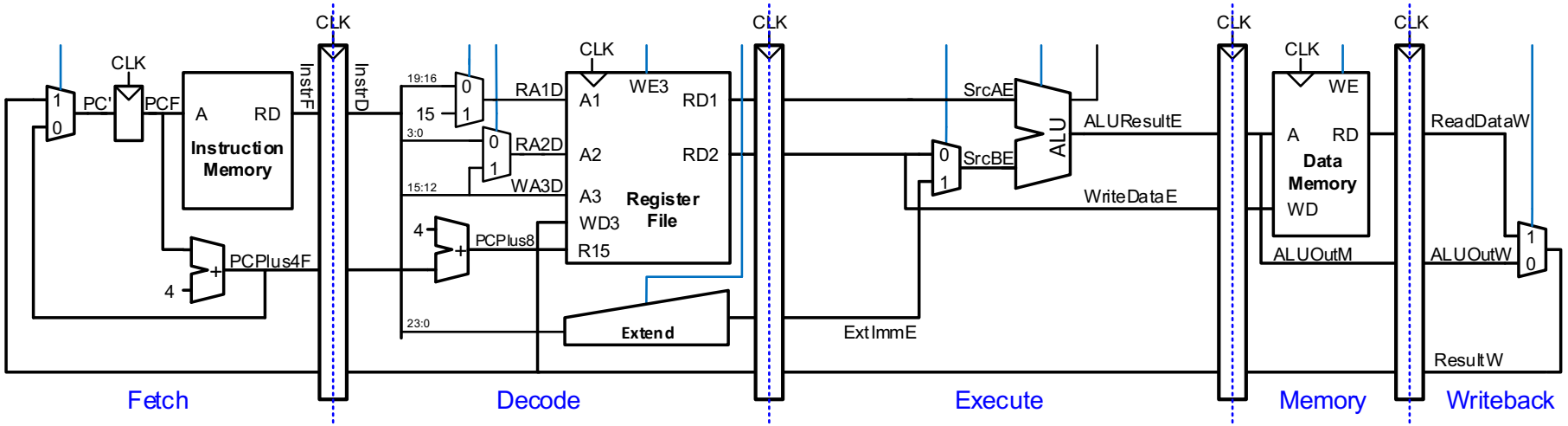
- Stages and their boundaries are indicated in blue
- Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside

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]
```

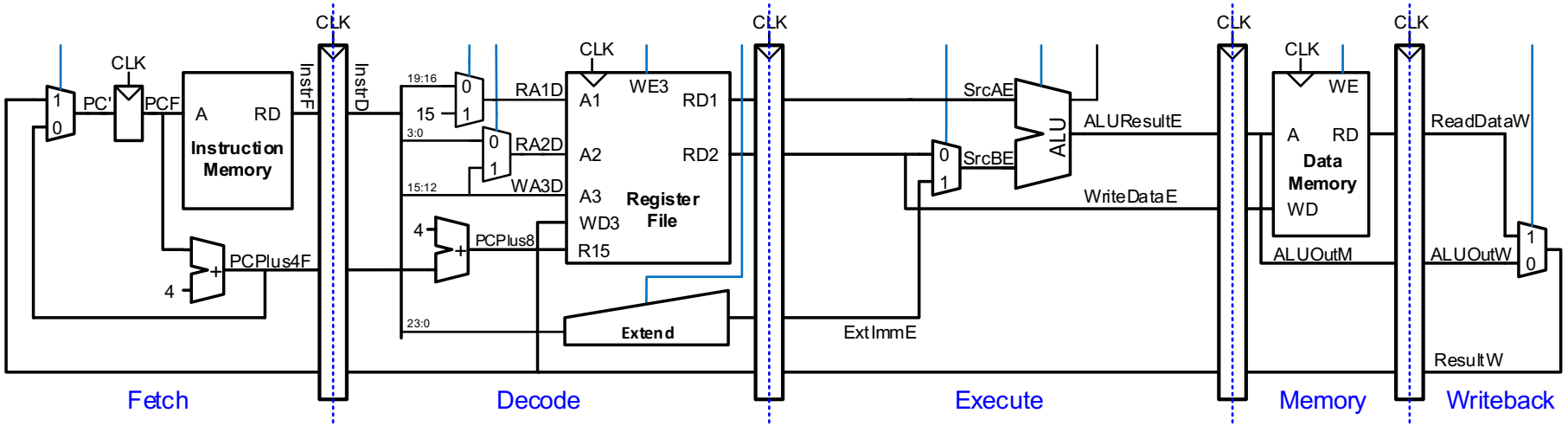
Pipeline Operation: Cycle 1



I1

- Is the pipeline fully utilized? NO

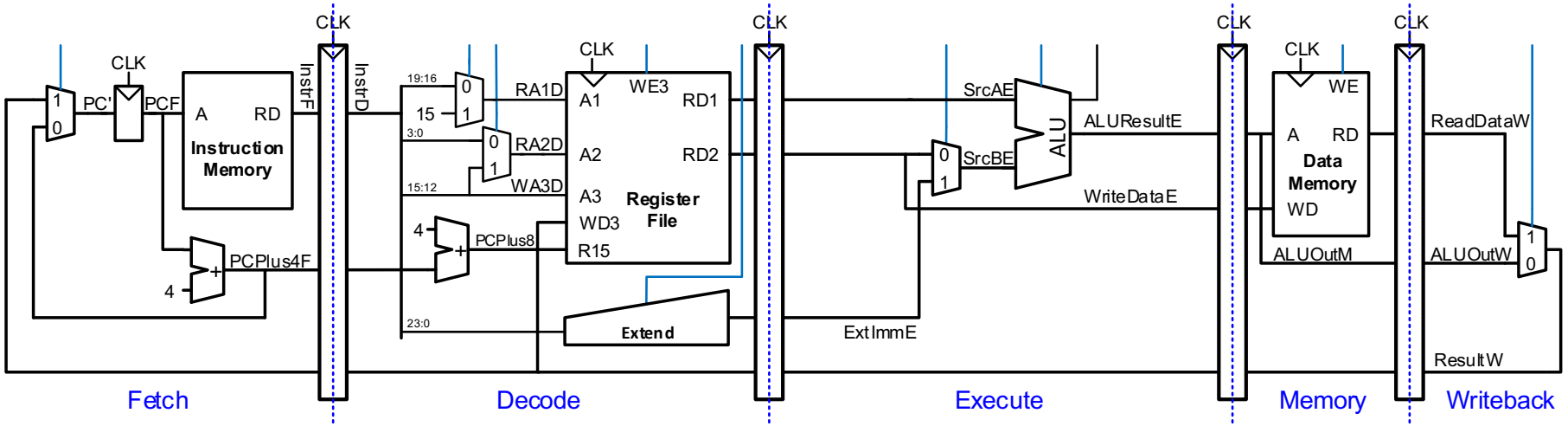
Pipeline Operation: Cycle 2



I2 I1

❑ Is the pipeline fully utilized? NO

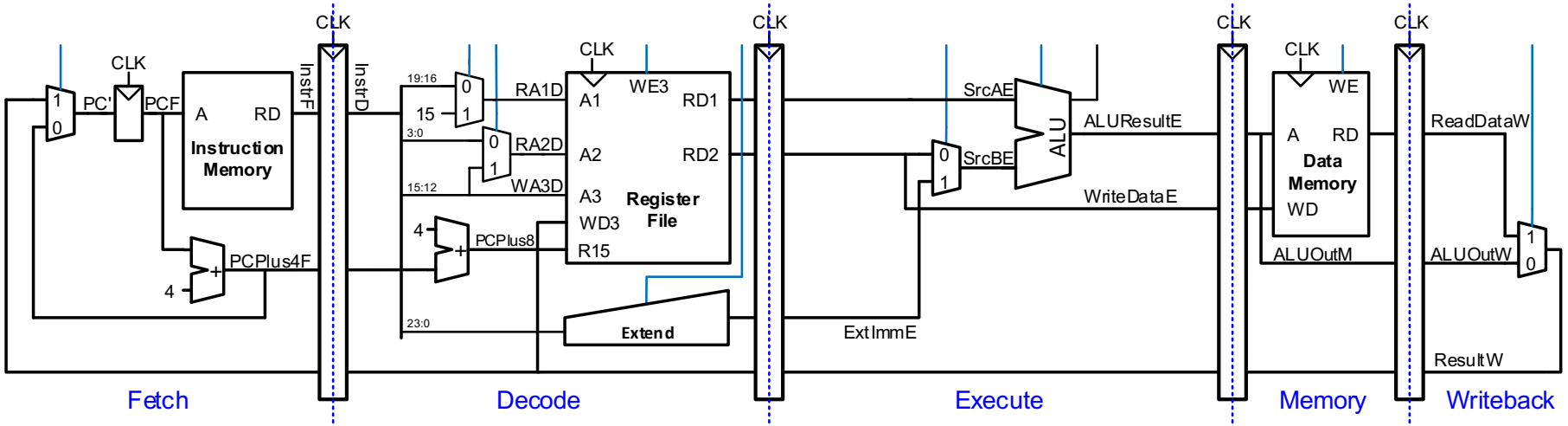
Pipeline Operation: Cycle 3



I3 I2 I1

- Is the pipeline fully utilized? NO

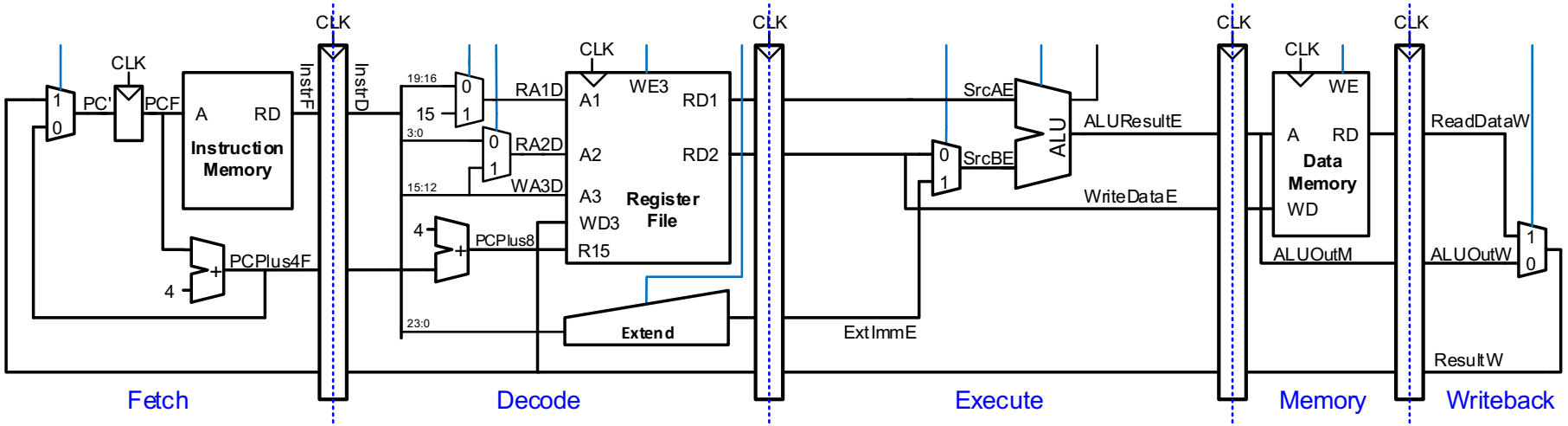
Pipeline Operation: Cycle 4



I4 I3 I2 I1

- Is the pipeline fully utilized? NO

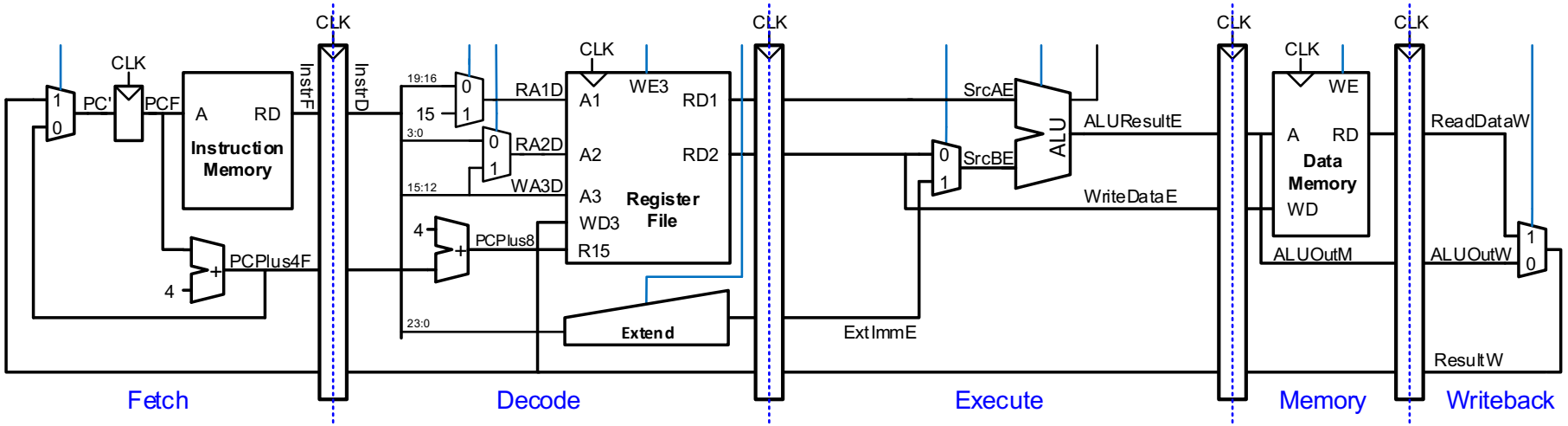
Pipeline Operation: Cycle 5



I5 I4 I3 I2 I1

- Is the pipeline fully utilized? YES

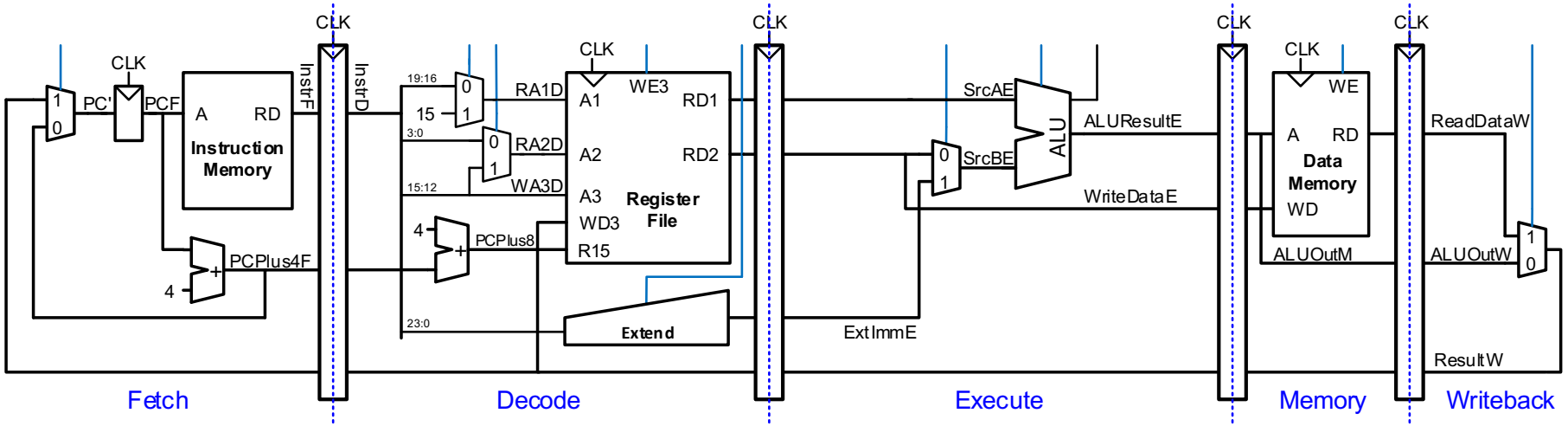
Pipeline Operation: Cycle 6



I6 I5 I4 I3 I2

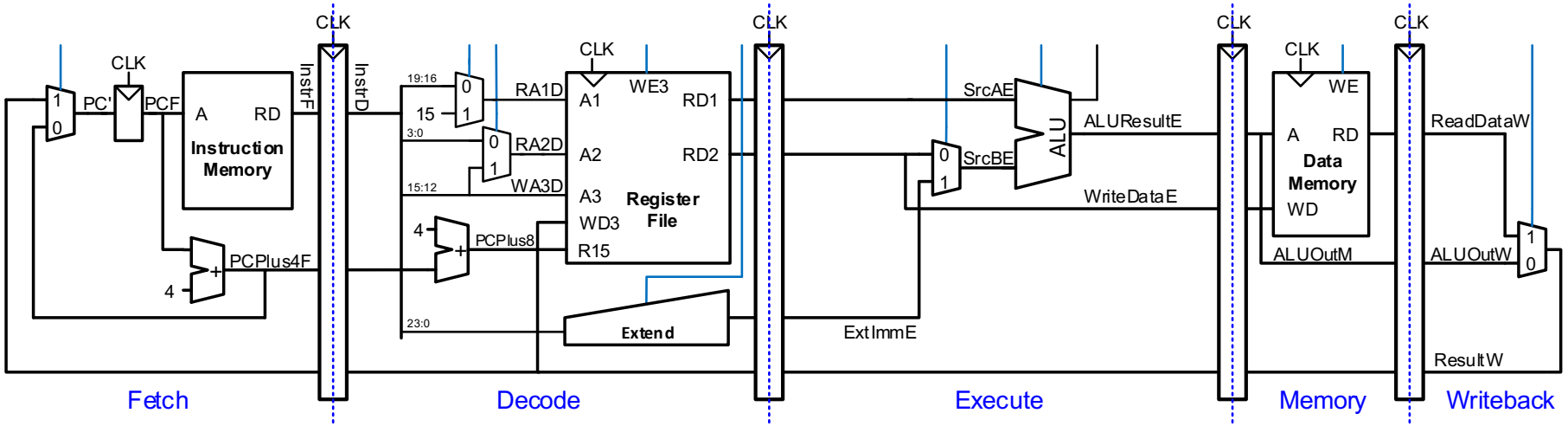
- Is the pipeline fully utilized? YES

Pipeline Operation: Cycle 7



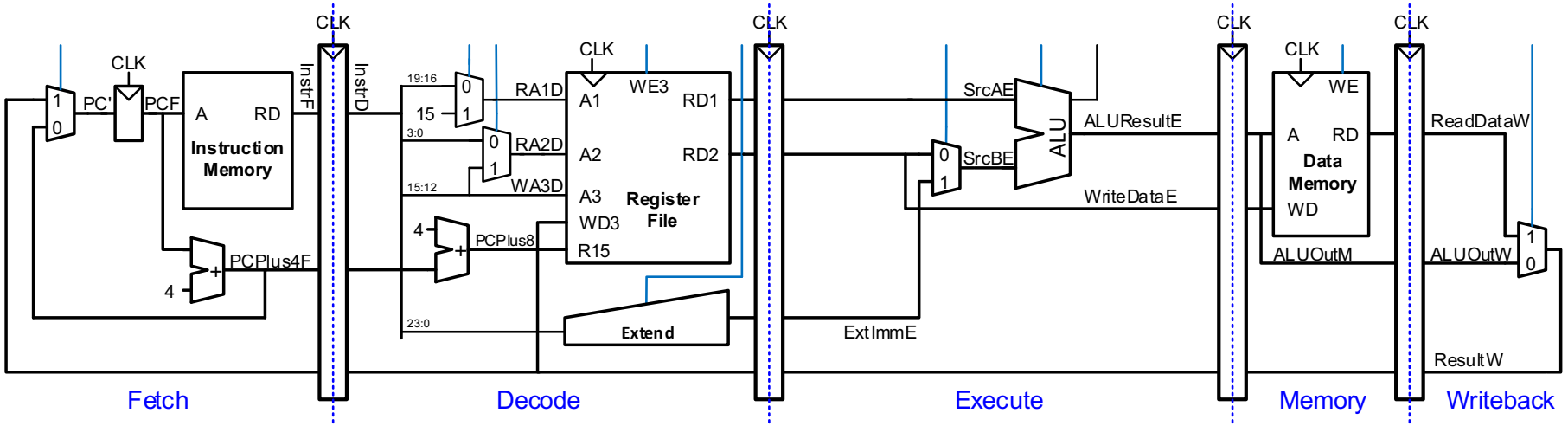
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 8



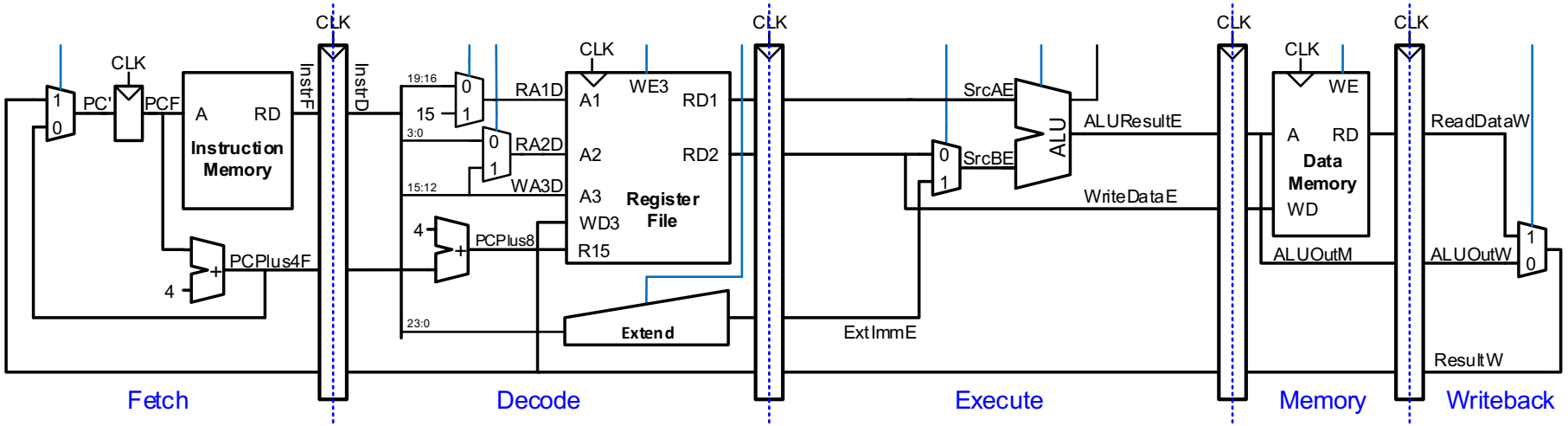
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 9



❑ Is the pipeline fully utilized? NO

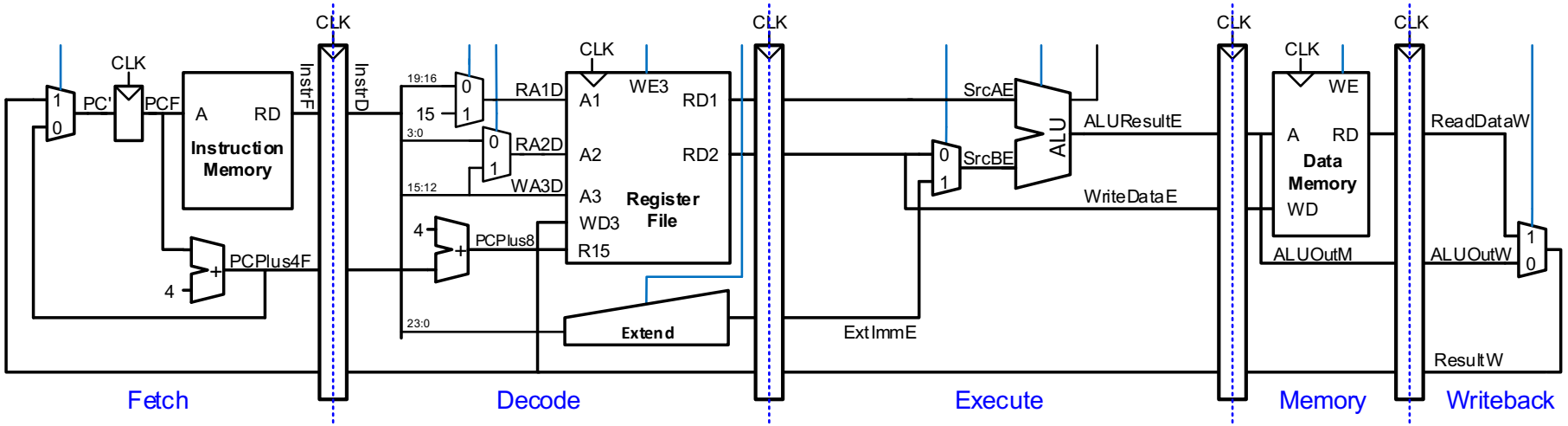
Pipeline Operation: Cycle 10



I6

- Is the pipeline fully utilized? NO

Pipeline Operation



- No more instructions to execute

Performance Analysis

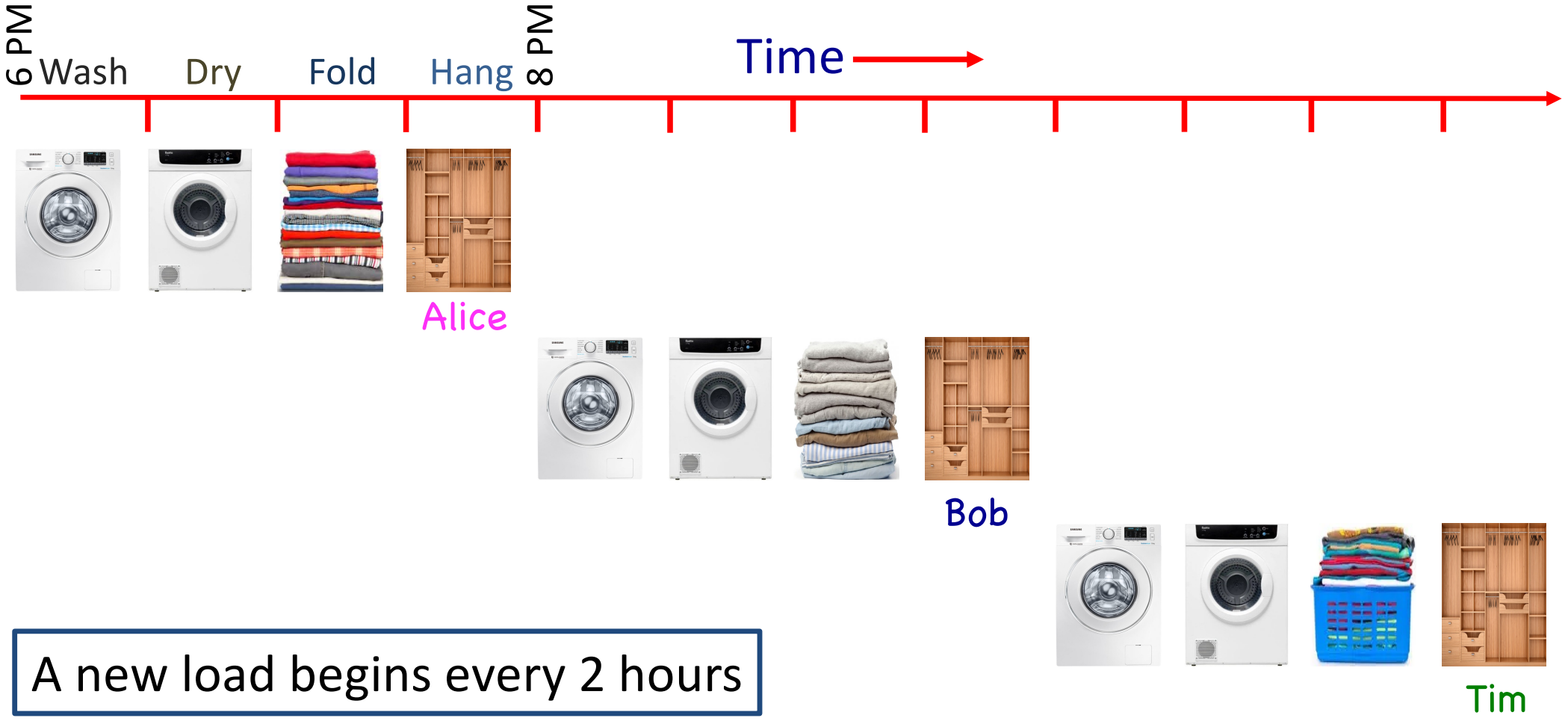
- The **6 instructions** took **10 cycles** to finish execution

- 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**

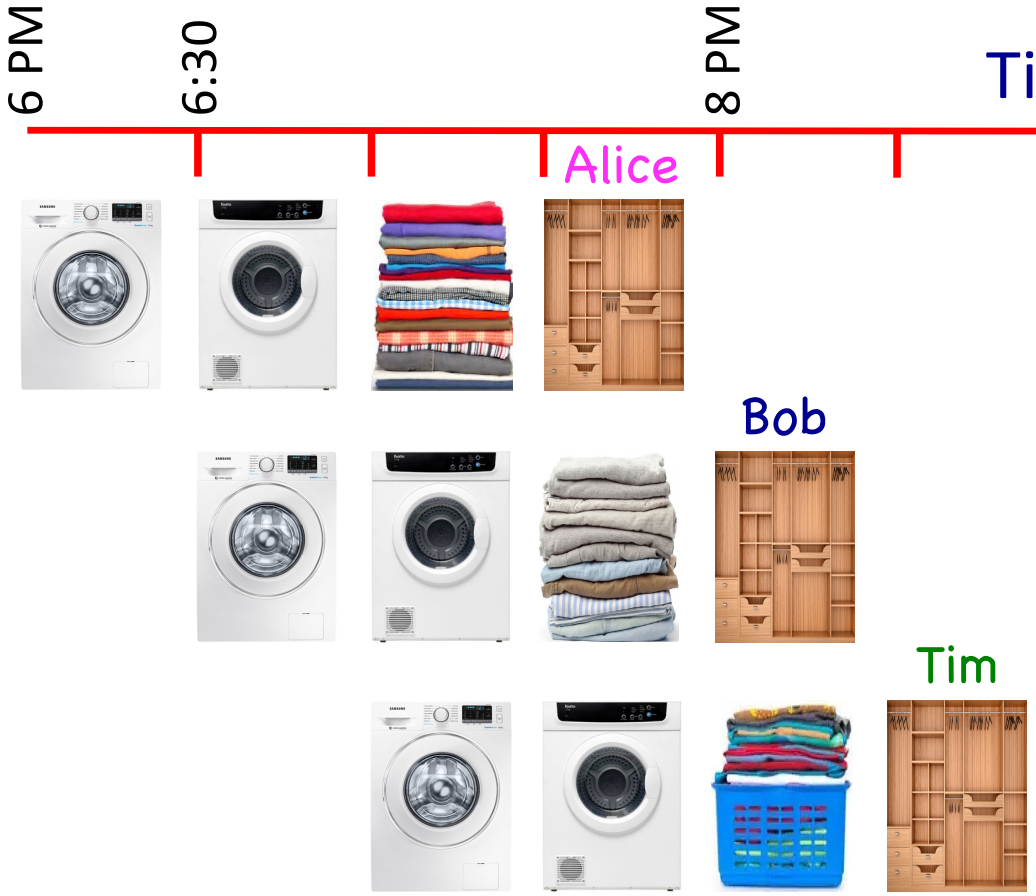
- Why is IPC less than 1?
  - It takes some time to fill and some time to drain the pipeline
    - During this time pipeline is operating below its potential

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

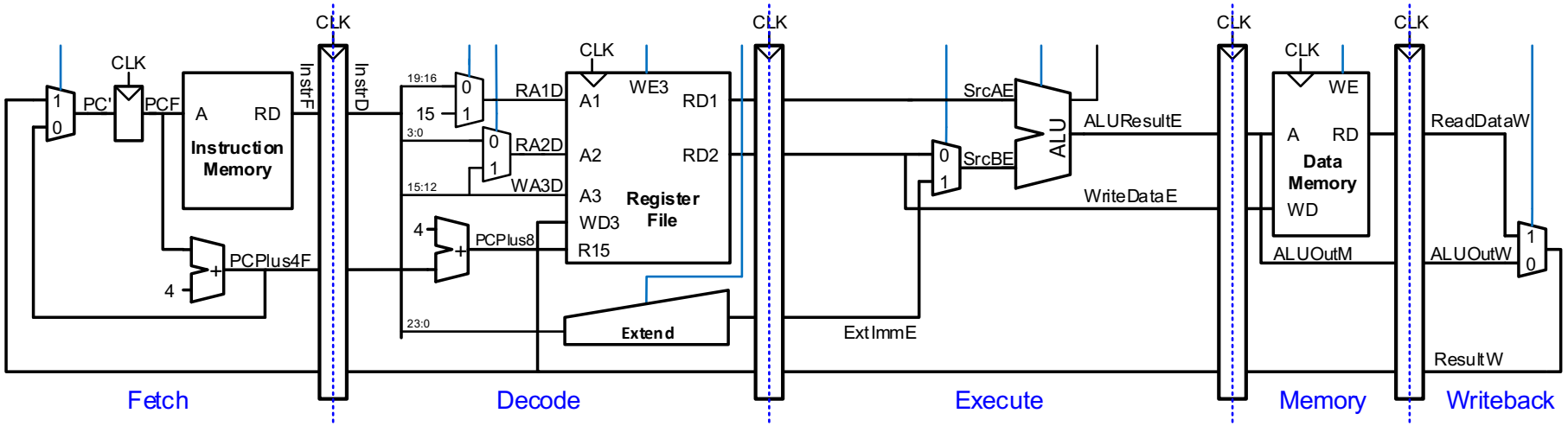
- The two factors limit the pipeline from delivering ideal speed-up
  - In the case when the amount of work is small relative to the number of stages in the pipeline

- 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

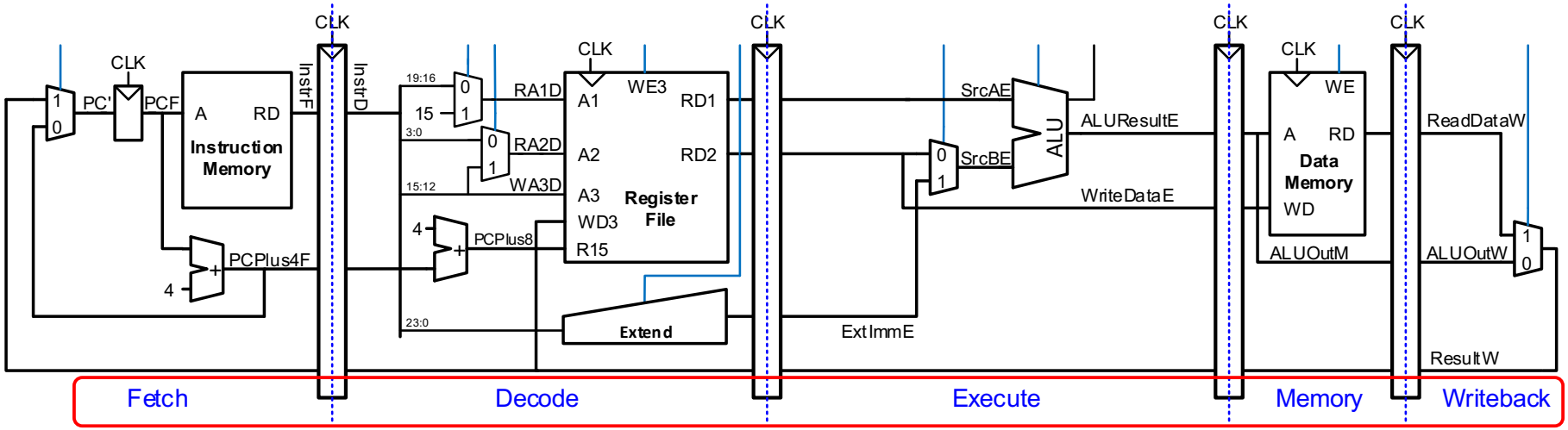
Bug in Pipelined Hardware!



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

We were Here.

Recap: Pipelined Data



- **From Fetch to Decode:** Instructions and `PC + 4`
- **From Decode to Execute:** Two register values and extended immediate
- **From Execute to Memory:** `ALUResultE` and `WriteDataE`
  - `WriteDataE` is one of the registers read from the `RF`, and `M` stage may need it for writing to memory in the case of an `STR` instruction
- **From Memory to Writeback:** Output of ALU (`ALUOutM`) and data read from memory (`ALUOutW`)

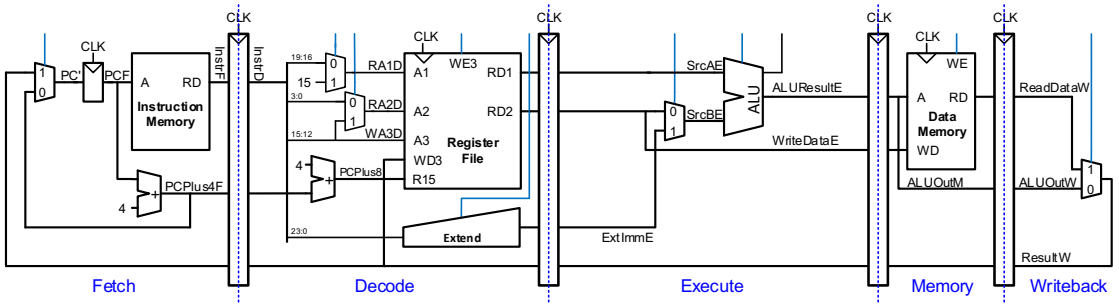
- **Think:** What is the width of each pipeline register?

Recap: Stages

- Fetch (F)
- Decode/RF-Read (D or DE/DEC or RF)
- Execute (E or EX)
- Memory (M or MEM)
- Writeback (W or WB)

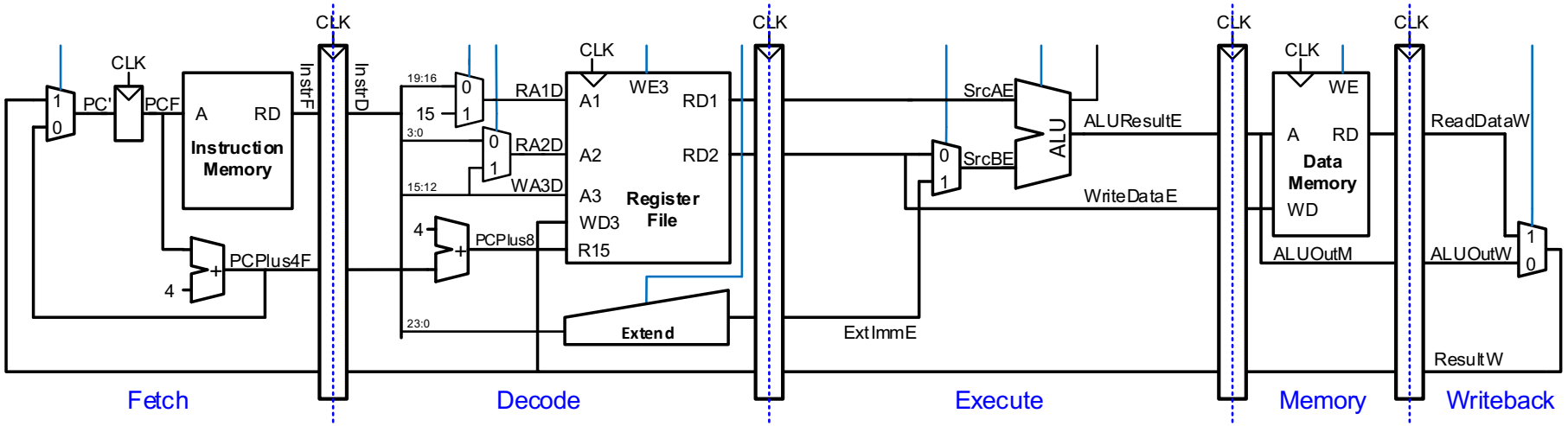
Recap: Pipeline Register Names

- PC is often referred to as the Fetch PPR
- B/w Fetch and Decode: Decode PPR
- B/w Decode and Execute: Execute PPR
- Similarly, Memory PPR
- Writeback PPR



Bug in Pipelined Hardware!

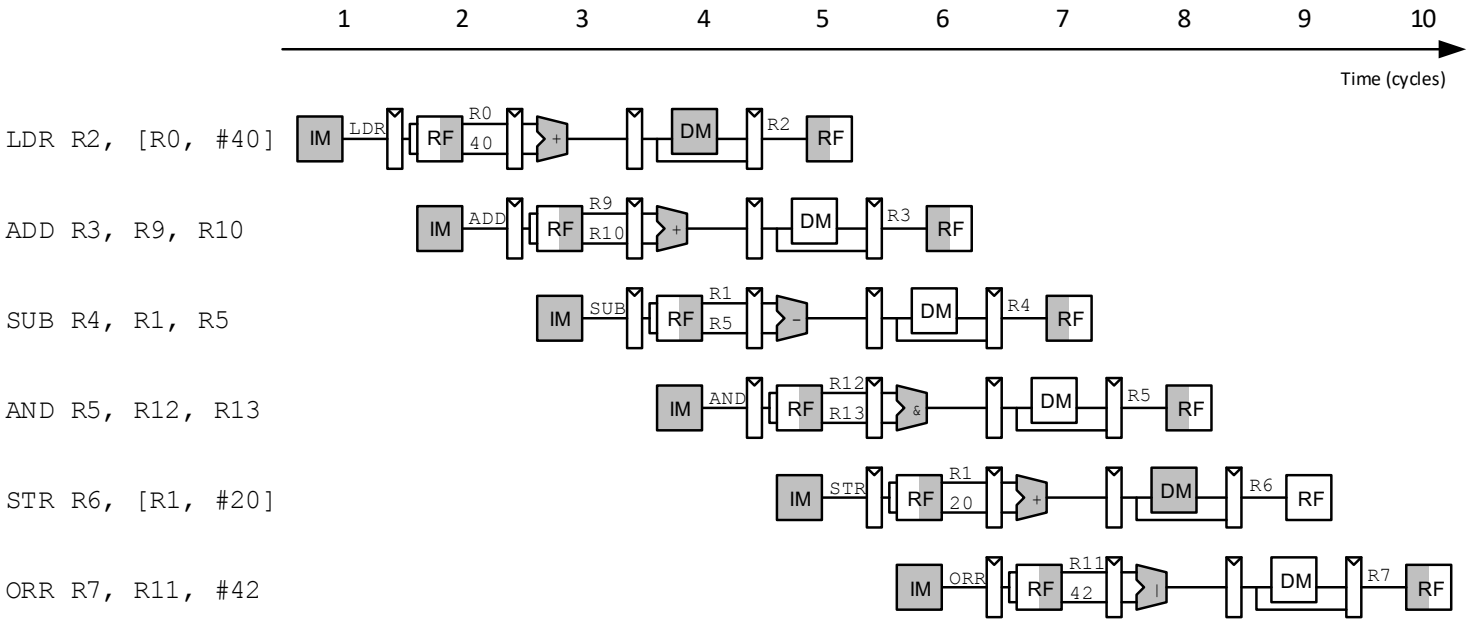
- The error is in the register file write logic that operates in the writeback stage



- The data value comes from ResultW, a Writeback stage signal
- But the write address comes from $InstrD_{15:12}$ (WA3D), a Decode stage signal
- Without correction, during cycle 5, the result of the instruction in the writeback stage would be incorrectly written to a different destination register

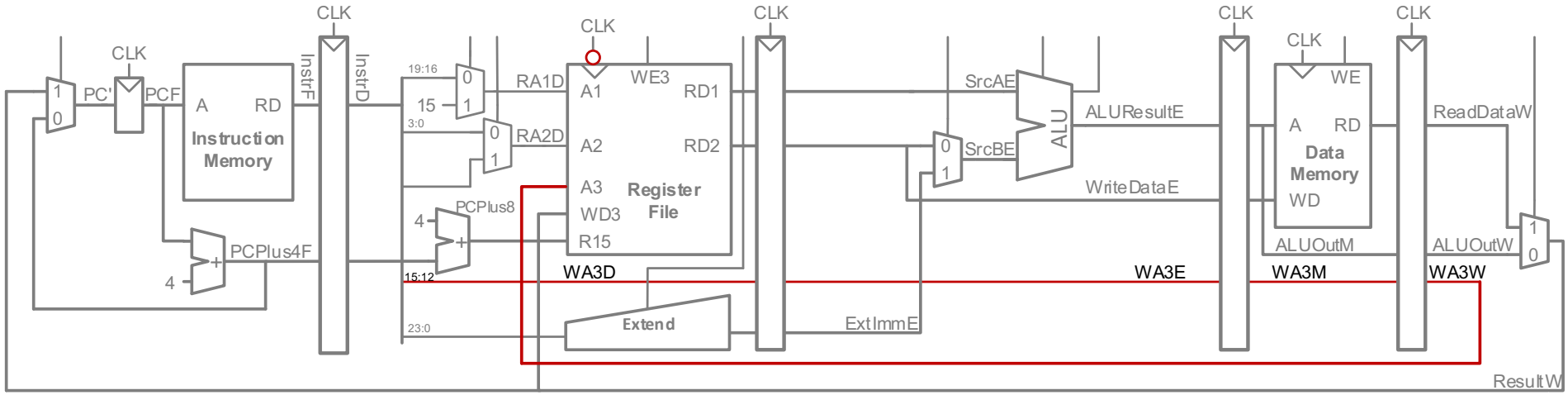
Bug in Pipelined Hardware!

- Without correction, during cycle 5, the result of the `LDR` instruction would be incorrectly written to `R5` instead of `R2`



Corrected Pipelined Datapath

- Here is the corrected pipelined datapath

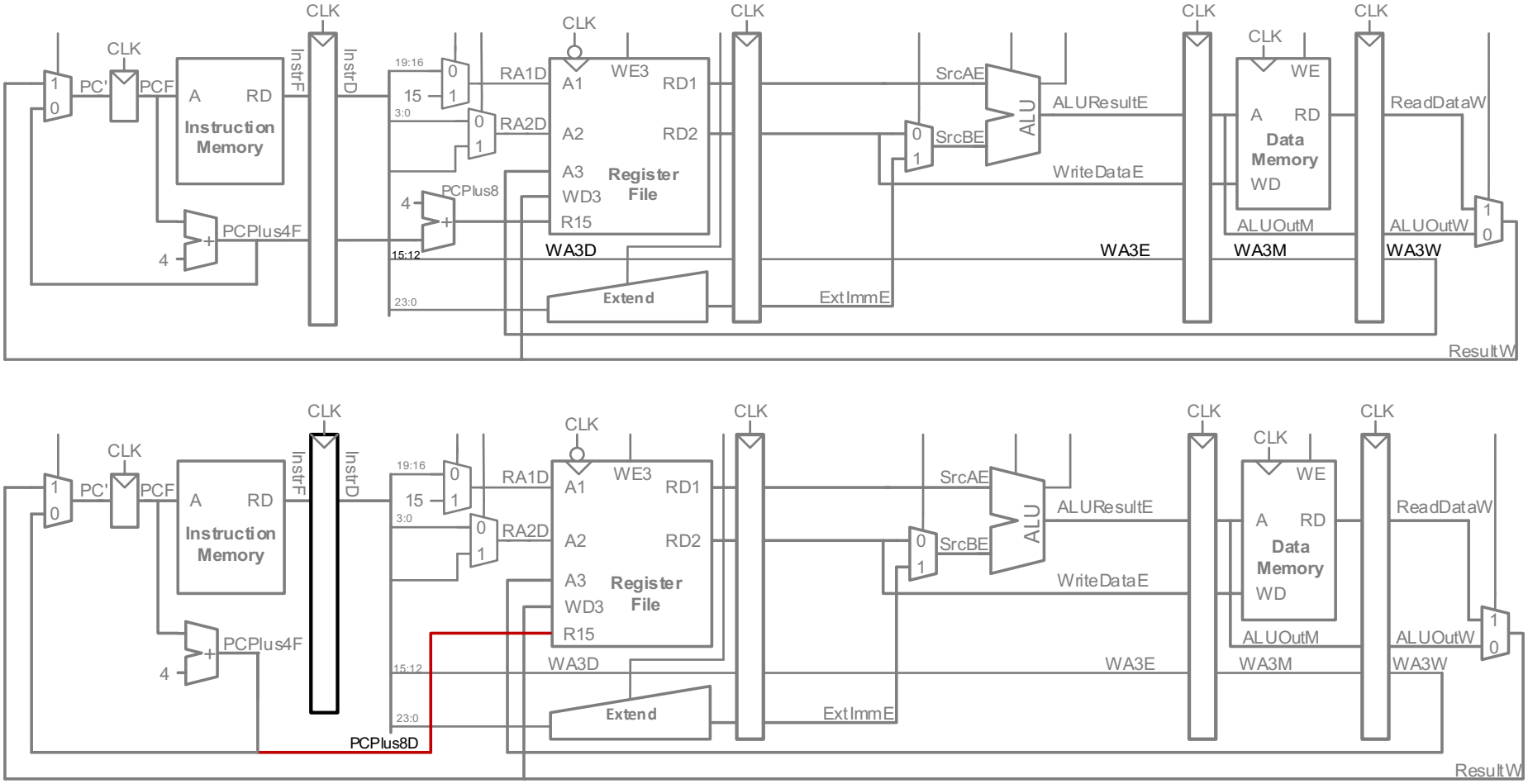


- The `WA3` signal is now pipelined along through the Execution, Memory, and Writeback stages so it remains sync with the rest of the instruction

- `WA3W` and `ResultW` are fed back together to the register file in the Writeback stage

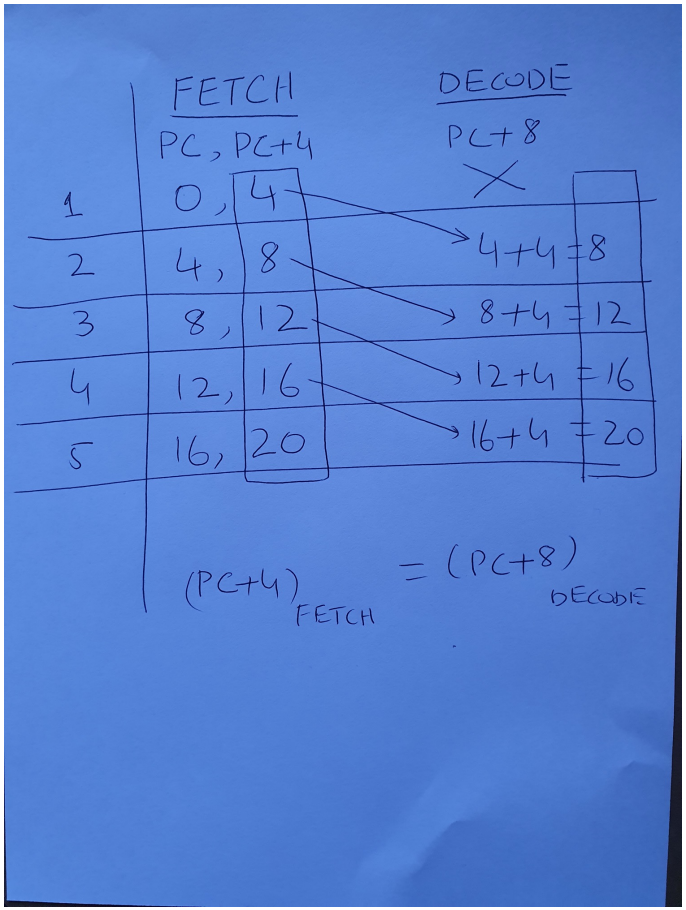
Optimized Pipelined Datapath

- Remove adder by using PCPlus4F after PC has been updated to PC+4



Optimized Pipelined Datapath

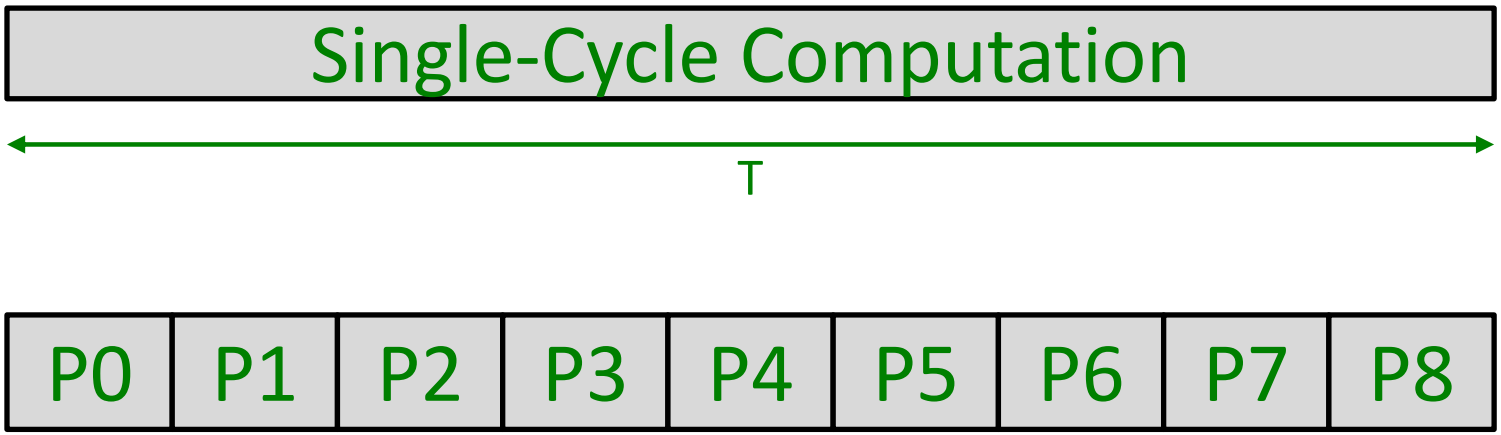
- Why the optimization works?



Balanced Pipeline (1)

- Let's revisit what hinders achieving ideal pipeline speedup
- Ideally, we want that the computation to be pipelined can be evenly partitioned into k uniform-latency subcomputations
  - If the latency (clocking period) of the original computation is T, then the clocking period of the pipelined computation is T/k
- Why is this a problem?
- Is the ARM pipelined microarchitecture balanced?
- What can we do to make it more balanced?

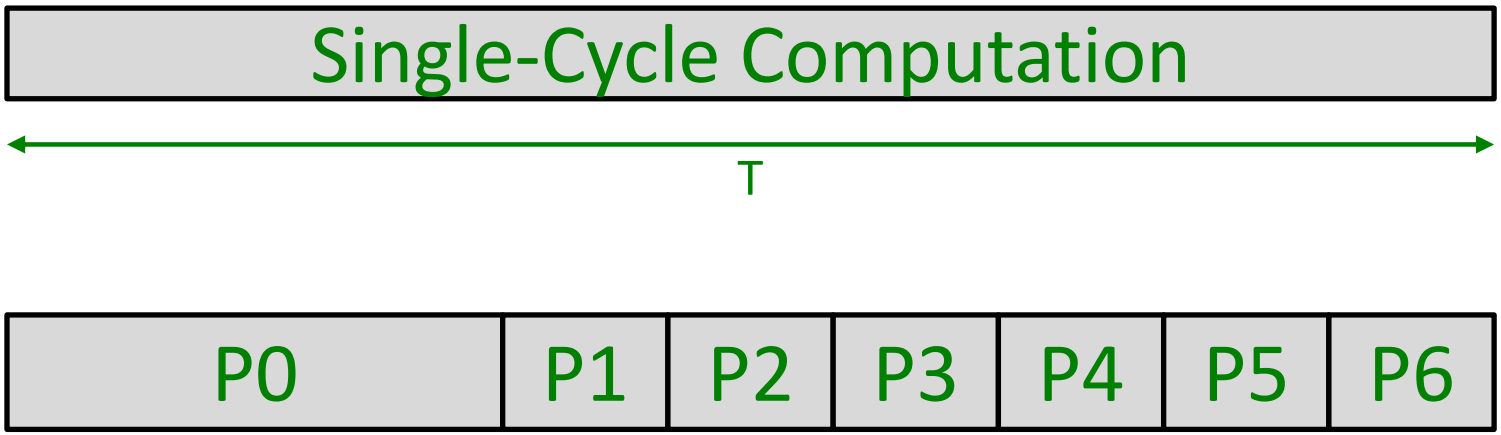
Balanced Pipeline (Example 1)



- Are the **8 stages** dividing the original computation sufficiently balanced?
  - Yes, the ideal speed-up is 8 (ignoring sequencing overheads)

- Unbalanced workload partitioning reduces speed-up (next example)

Balanced Pipeline (Example 2)



- Are the **8 stages** dividing the original computation sufficiently balanced?
  - **NO**, the cycle time (frequency) should account for the **slowest stage** (worst-case stage delay)
  - Each stage must incur the same latency as P0. **Latency of computation is very high compared to the original computation**

Control Unit for Pipelined uArch

- Same control signals as the single-cycle processor
  - Therefore, uses the same control unit

- The control unit examines the `Op` and `Funct` fields of the instruction in the Decode stage to produce the control signals

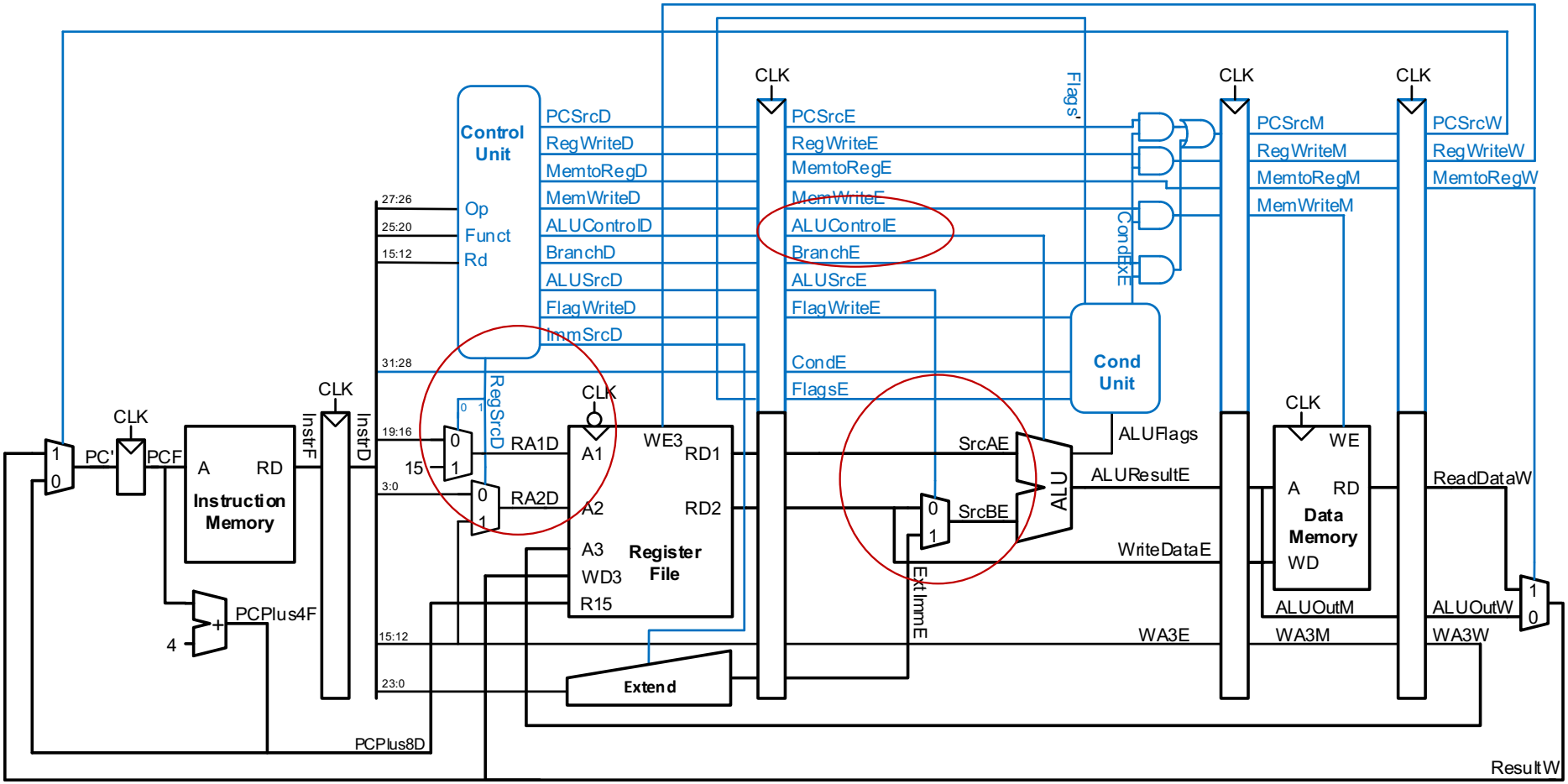
- These control signals must be pipelined along with the data

- **Remember:** The control unit also examines the `Rd` field (back flow)

- Special treatment for `RegWrite` and `WA3` (backward flow)

Pipelined Processor Control

- No need to send the circled signals to the next stage because they are no longer needed



Pipeline Hazards

- When multiple instructions are handled concurrently there is a danger of hazard

- Hazards are a part of real life

- Some **coping strategies:** Get around, precaution, mitigate harm after

Pipeline Hazards (Three Types)

- Structural hazard
  - When two instructions want to use the same resource
  - Memory for instructions (F) and data (M)
  - Register file is accessed in two different stages (**what are those?**)
- Data hazard
  - When a dependent instruction wants the result of an earlier instruction
- Control hazard
  - When a PC-changing instruction is in the pipeline (**why is this a hazard?**)

Hazard Mitigation

- Hardware for concurrent instruction execution must deal with hazards

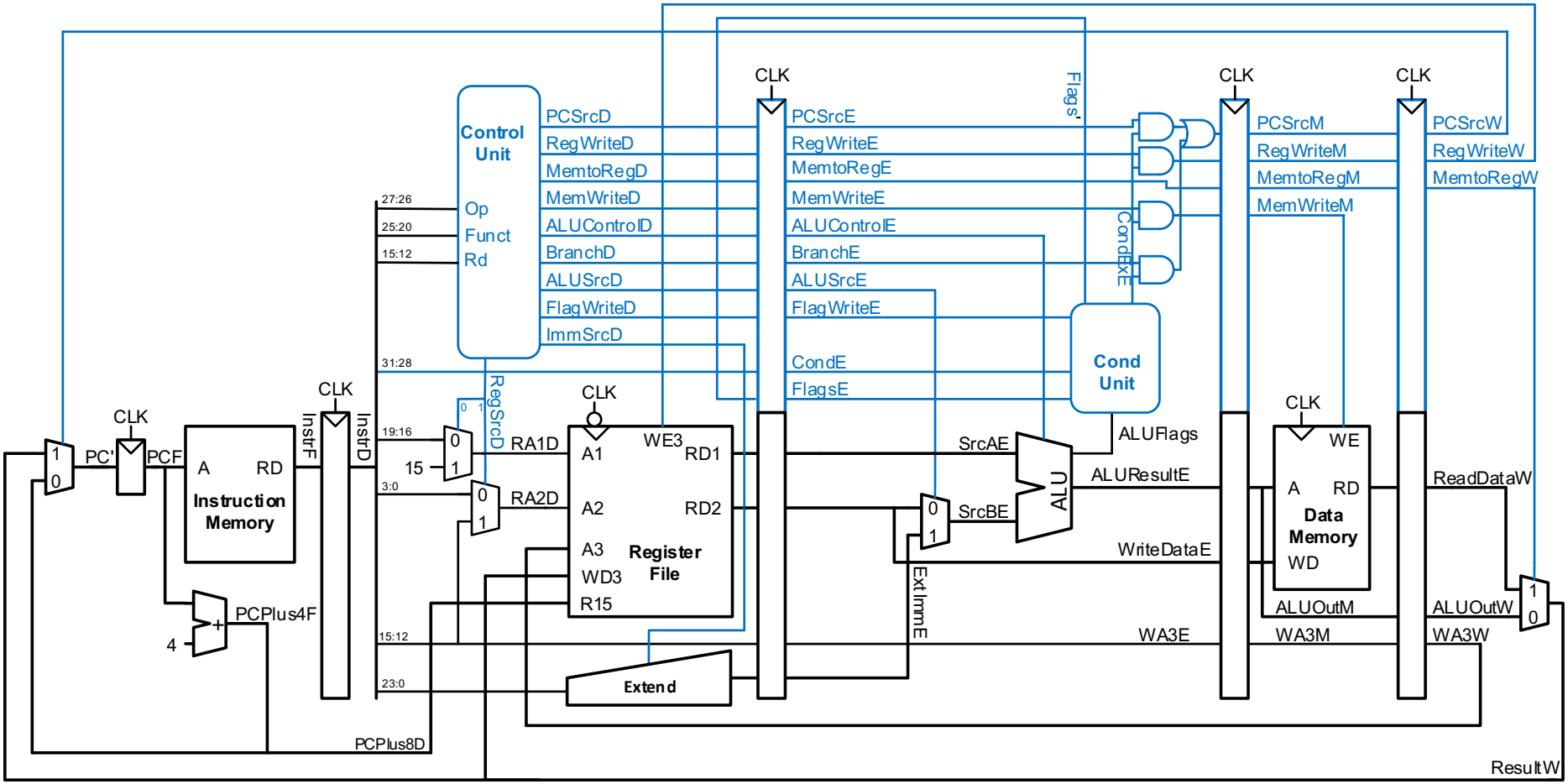
- From the processor's perspective:
  - Different solutions with different tradeoffs
    - Architectural state requires "serious" recovery
    - Architectural state is **untouched**, and hazard avoided
    - Lots of logic dedicated to hazard avoidance
    - Be very defensive (**stall the world** until hazard is gone)
    - Power, energy, latency are all considerations

Pipeline Hazards (Another View)

- Instructions and data generally flow from left to right
- Right-to-left flow affect future instructions and leads to hazards
- Writeback stage places the result into the register file (potential for data hazard)
- Selection of next `PC`, choice of `PC + 4` or branch target address
  - Also backward flow and a hazard: control hazard

Pipeline Hazards (Another View)

- Identify backward flows (control and data)



Recall: Data Dependences (Week 7)

- In Von Neumann model, instructions depend on each other for data
- **Data (True) Dependence:** One instruction **produces** a results that the subsequent instruction **consumes**
- **One can visualize a sequential program as instruction flow or data flow**
- Data dependence implies the two instructions must execute in program order
- They cannot be executed simultaneously (in parallel)
- We will discuss two more types of dependences
  - False dependences
  - Control dependences

Read-After-Write Hazards

- True dependences lead to read-after-write hazards

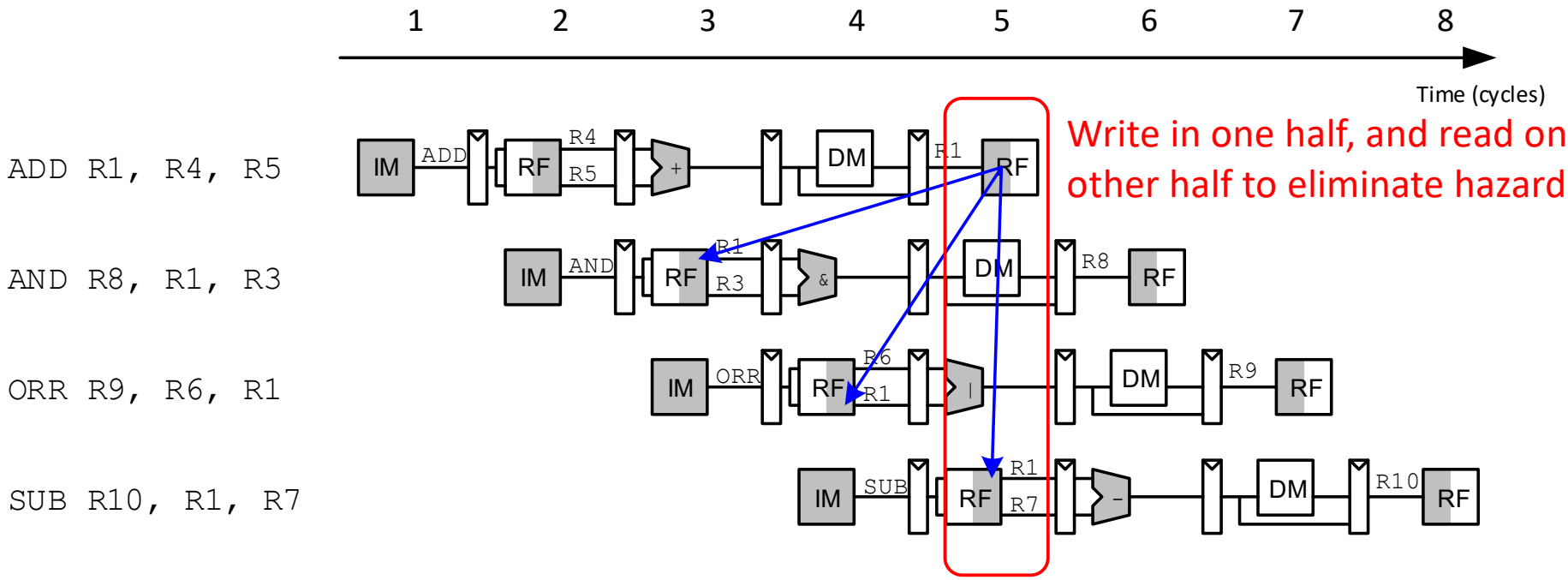
- **Think:** These hazards are not possible in a single-cycle microarchitecture

- **Two Very Important points to remember:**
  - True dependencies are a property of the program (programmer's intention is expressed by way of them)

  - Hazards are a property of microarchitecture
    - A dependency may or may not lead to a hazard

Pipeline Hazards (Example)

- Look at the instructions on the left. There are three data hazards



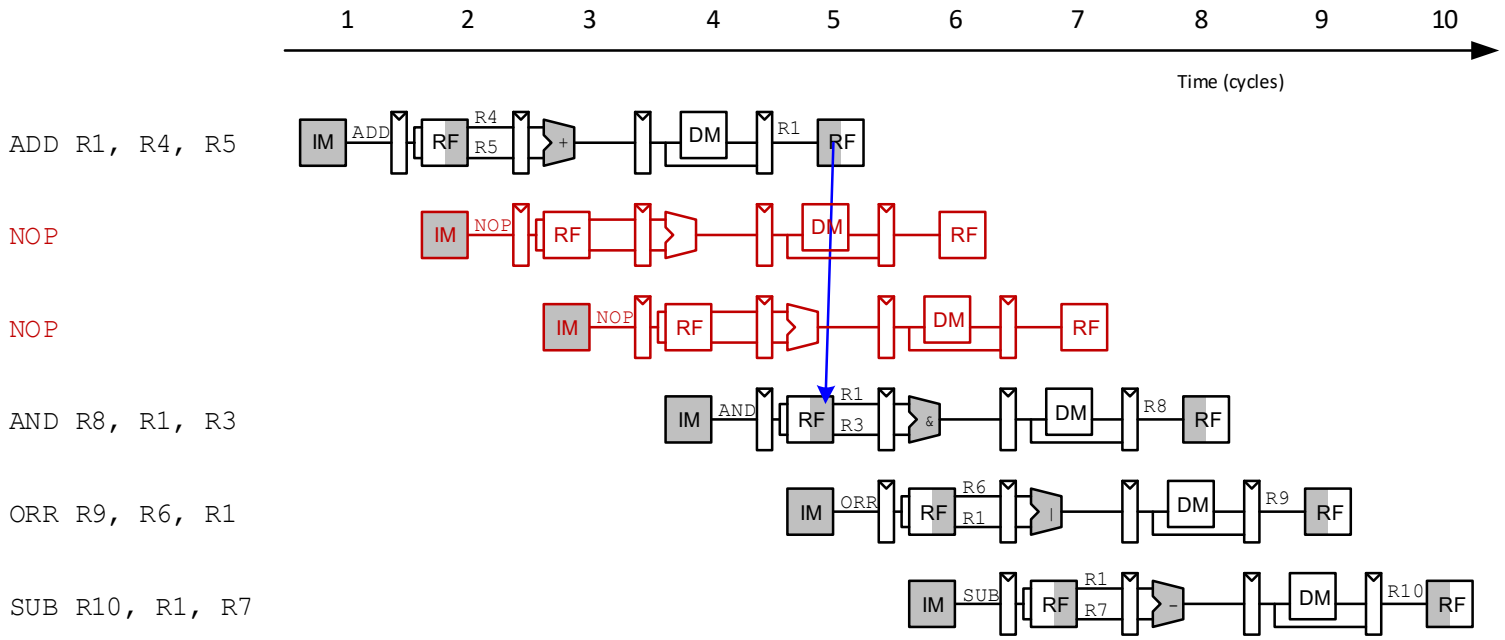
- Use a clever register read/write policy to eliminate one hazard
  - What can we do about the remaining two hazards?

Solution # 1: Software Interlocking

- Insert `NOPS` in code at compile time
  - NOP is an instruction that does nothing
- Rearrange code at compile time
- Programming is complicated
- Speed is degraded

Example: Software Interlocking

- Insert enough NOPS for results to be ready
- Or move dependent useful instructions forward



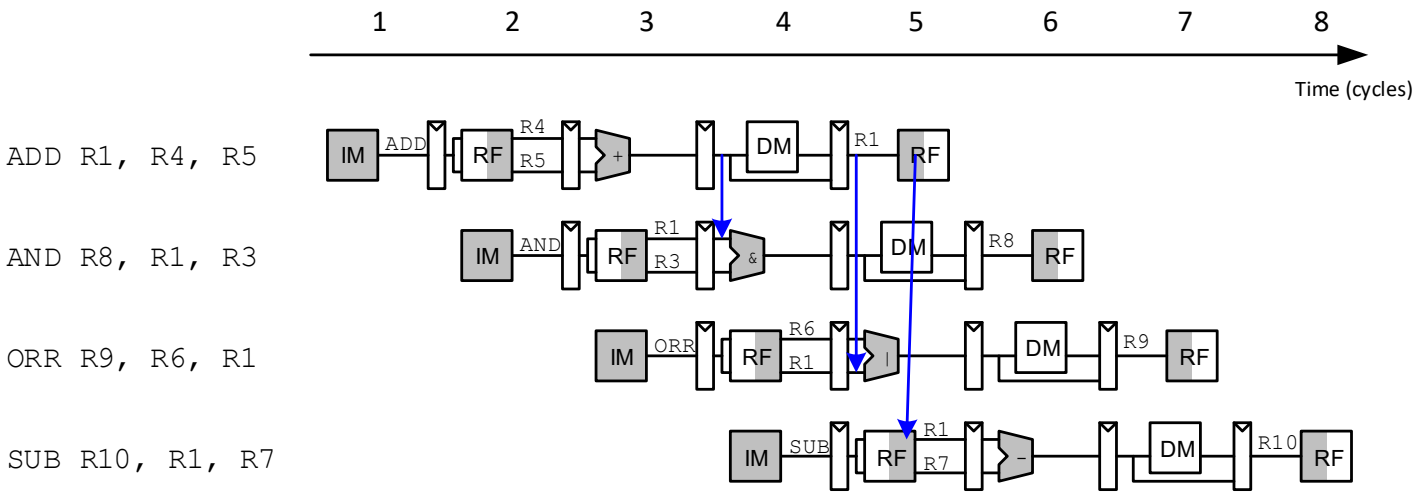
Solution # 2: Forwarding or Bypassing

- Some data hazards can be solved by **forwarding** or **bypassing**

- Extra hardware to send the result from the Memory or Writeback stage to a dependent instruction in the Execute stage
  - Key realization: We can bypass the register file and get results early from pipeline register

- Requires adding muxes in front of the ALU to select the operand from one of the many sources (RF, Memory PPR, Writeback PPR)

Forwarding Example

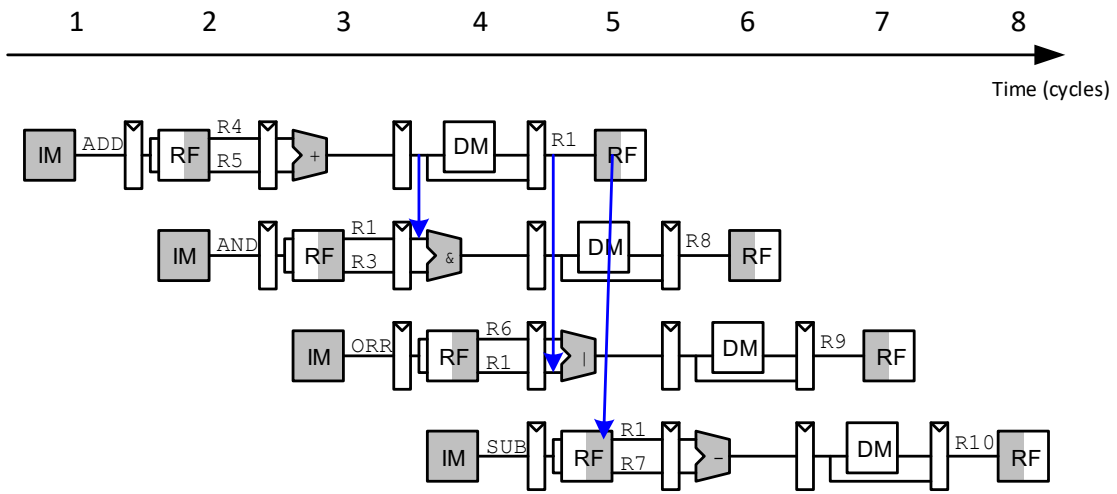


- Sum from the ADD instruction is computed by ALU in **cycle 3** and is not strictly needed by the AND instruction until the ALU uses it in **cycle 4**
- No need to wait for the results to appear in register file

Forwarding Example

Next 2 younger "dependent" instructions expose a **hazard**

```
ADD R1, R4, R5
AND R8, R1, R3
ORR R9, R6, R1
SUB R10, R1, R7
```



- When is forwarding necessary?
  - Check if register read in EX stage **matches** register written in MEM or WB stage
  - If so, forward result

Necessary Conditions for Forwarding

- When an instruction in Execute stage has a source register that **matches** the destination register of an instruction in Memory or Writeback stage

- Let's write **equations** for generating control signals that indicate whether to forward or not

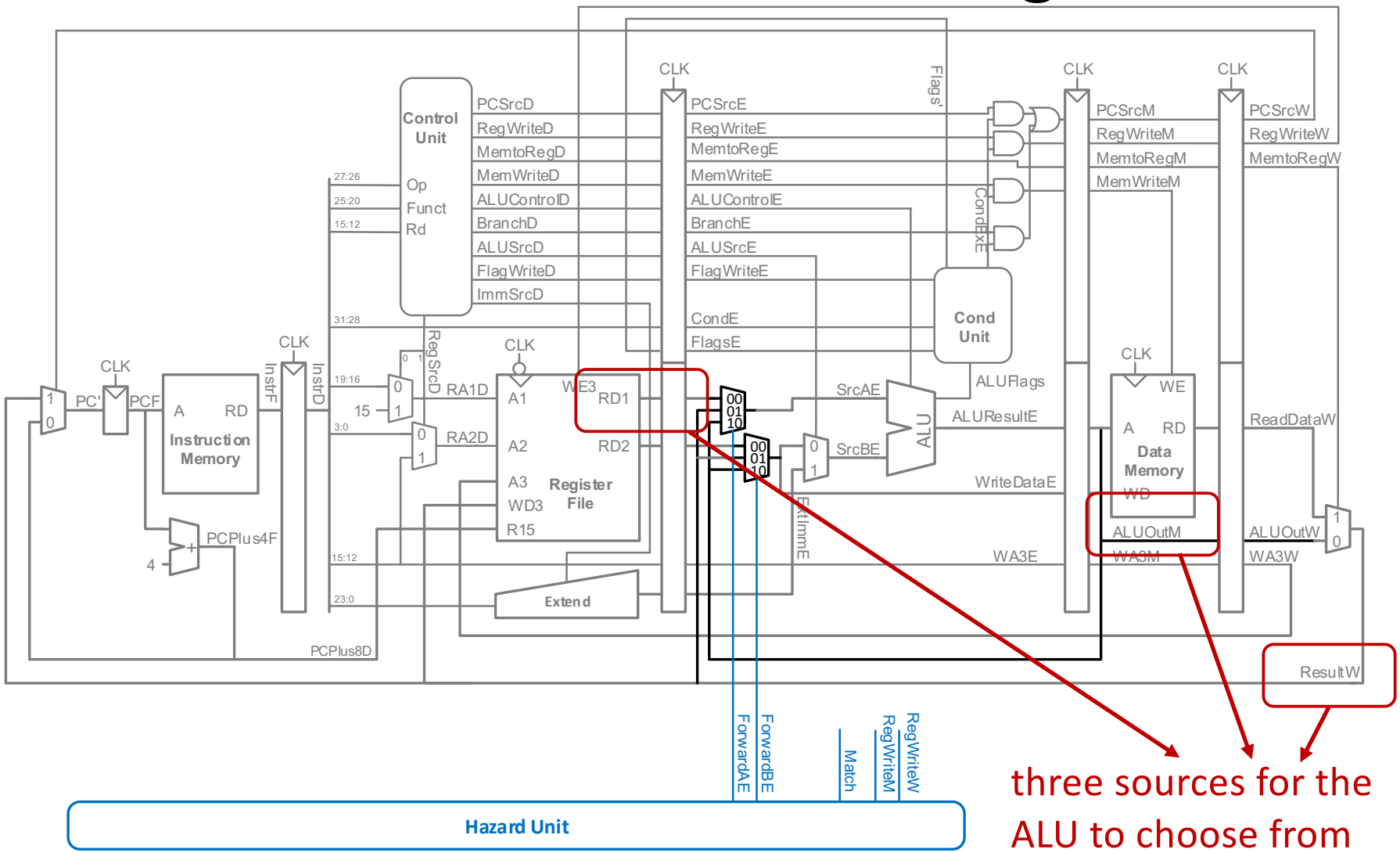
Necessary Conditions for Forwarding

- **Execute** stage register matches **Memory** stage register?
  Match_1E_M = (RA1E == WA3M)
  Match_2E_M = (RA2E == WA3M)
- **Execute** stage register matches **Writeback** stage register?
  Match_1E_W = (RA1E == WA3W)
  Match_2E_W = (RA2E == WA3W)
- If it matches, forward result:

```
if      (Match_1E_M • RegWriteM)    ForwardAE = 10;
else if (Match_1E_W • RegWriteW)    ForwardAE = 01;
else                                ForwardAE = 00;
```

ForwardBE same but with Match2E

Pipelined Processor with Forwarding



three sources for the ALU to choose from

Coming Attractions

- Some data hazards cannot be solved with forwarding
  - We need to solve such hazards by stalling the pipeline
  - Think: LDR followed by ADD with a true dependency
  - In literature known as Load-Use Hazard

- Control hazards can also be solved with stalls but there is a clever alternative

- Key BIG Idea in This Lecture: Branch Prediction to resolve control hazards!

Load-Use Hazard

- **Recall:** Execution of Load has a two-cycle latency (E + M)

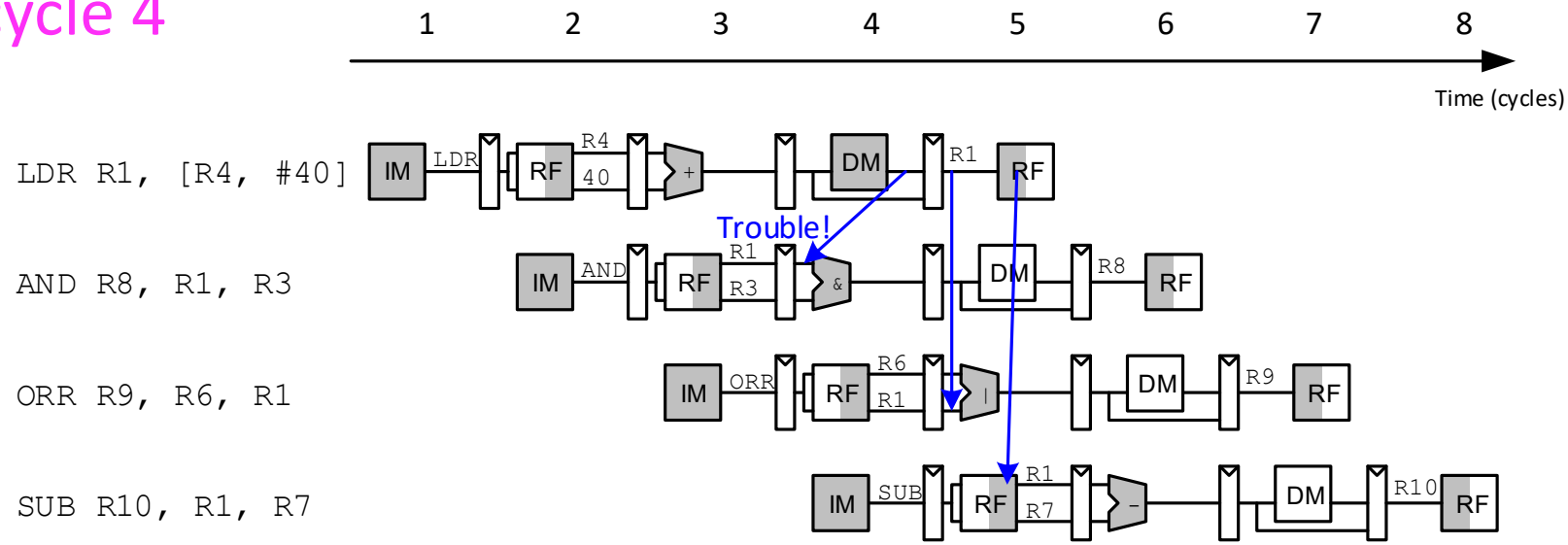
- `LDR` does not finish reading data until the end of the **MEM stage**
  - The result cannot be forwarded to the **EX-stage** of the next instruction
  - We call Load followed by its use a Load-Use hazard

- Load-Use hazard cannot be solved with forwarding

- **Solution:** stalling the pipeline until the data is available

Load-Use Hazard

- The `LDR` instruction received data from memory at the end of cycle 4



- The `AND` instruction needs that data at the beginning of cycle 4
- We cannot go backward in time and fix things up!

Stalls to Resolve Load-Use Hazards

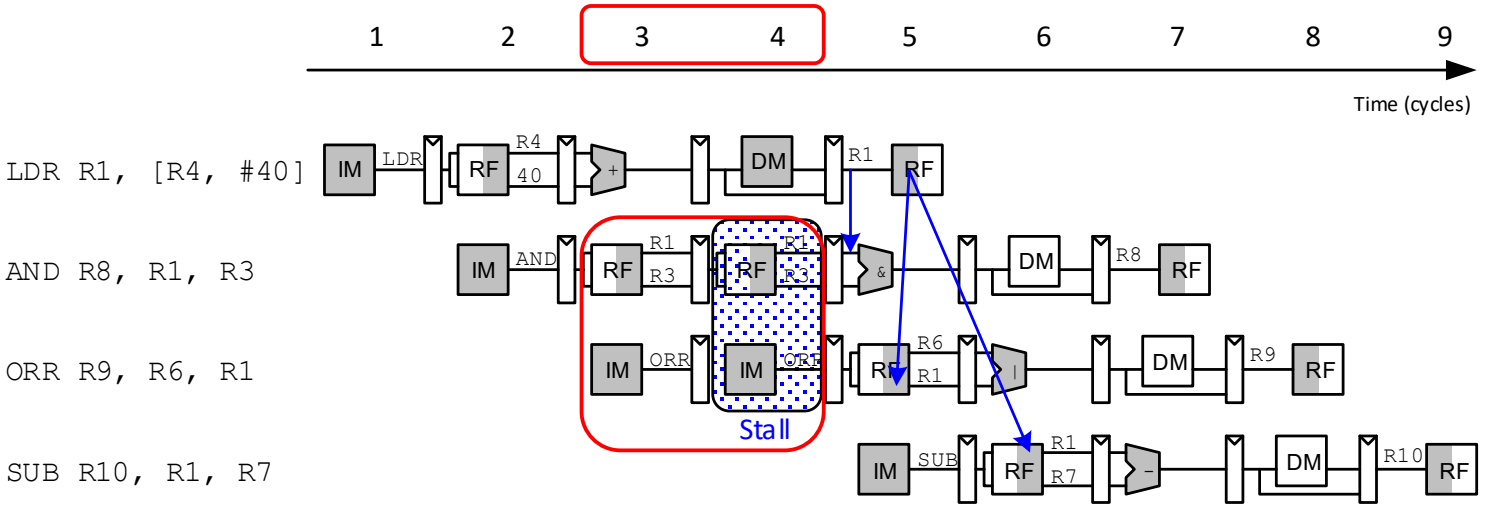
- The dependent instruction can be detected as the "user" of `LDR` in the **Decode stage**

- **Idea:** Stall the dependent instruction in the **Decode stage**

- Furthermore, the instruction immediately behind the "user" of `LDR` must remain in the **Fetch stage** because the **Decode stage** is **full**

Stalls to Resolve Load-Use Hazards

- Stall the dependent instruction (AND) in **Decode stage**



- AND remains in **Decode**, and ORR remains in **Fetch**
- **Cycle 5:** result forwarded from WB of LDR to EX of AND
  - The mechanism works!
- **Think:** If we stall AND, what does EX do in **cycle 4**? MEM in **cycle 5**?

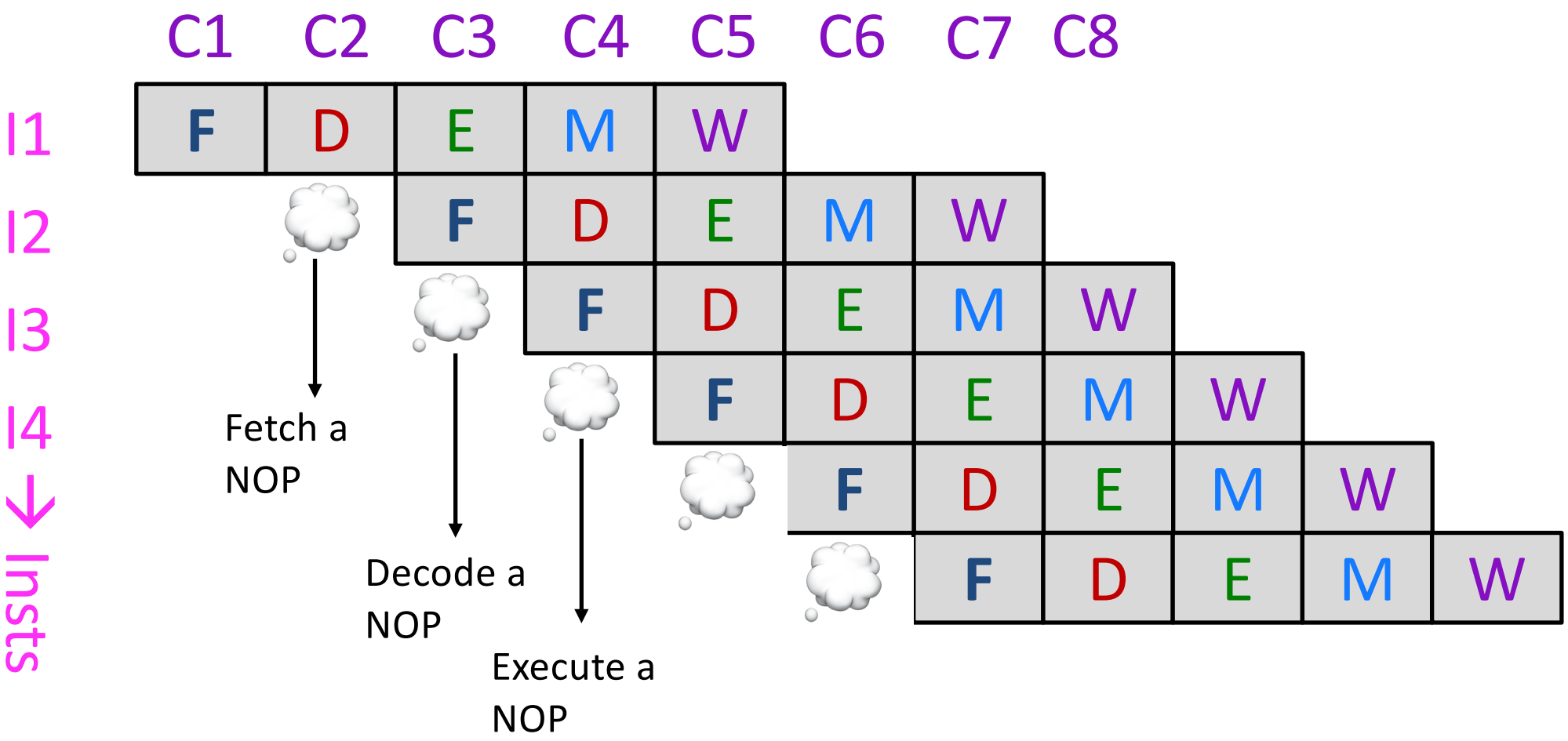
Pipeline Bubbles

- EX is unused in cycle 4
- MEM is unused in cycle 5
- WB is unused in cycle 6

- This used stage propagating through the pipeline is called a bubble

- It behaves like a NOP instruction

Visualizing Bubbles



Implementing Stalls

- **Stalling** a stage requires disabling the pipeline register, so that the contents do not change
  - Remember: All previous stages must also be stalled

- **Bubble** is introduced by clearing the pipeline register directly after the stalling stage
  - Prevents bogus information from propagating forward

- Forgetting to introduce a bubble may wrongly update the architectural state

- Stalls **degrade performance** so must be used only when needed

Logic to Compute Stalls and Flushes

- Is either source register in the **Decode stage** the same as the one being written in the **Execute stage**?

Match_12D_E = (*RA1D* == *WA3E*) + (*RA2D* == *WA3E*)

- Is a `LDR` in the **Execute stage** AND *Match_12D_E*?

ldrstall = *Match_12D_E* • *MemtoRegE*

StallF = *StallD* = *FlushE* = *ldrstall*

Pipelined CPU with Stalls to Solve Load-Use Hazard

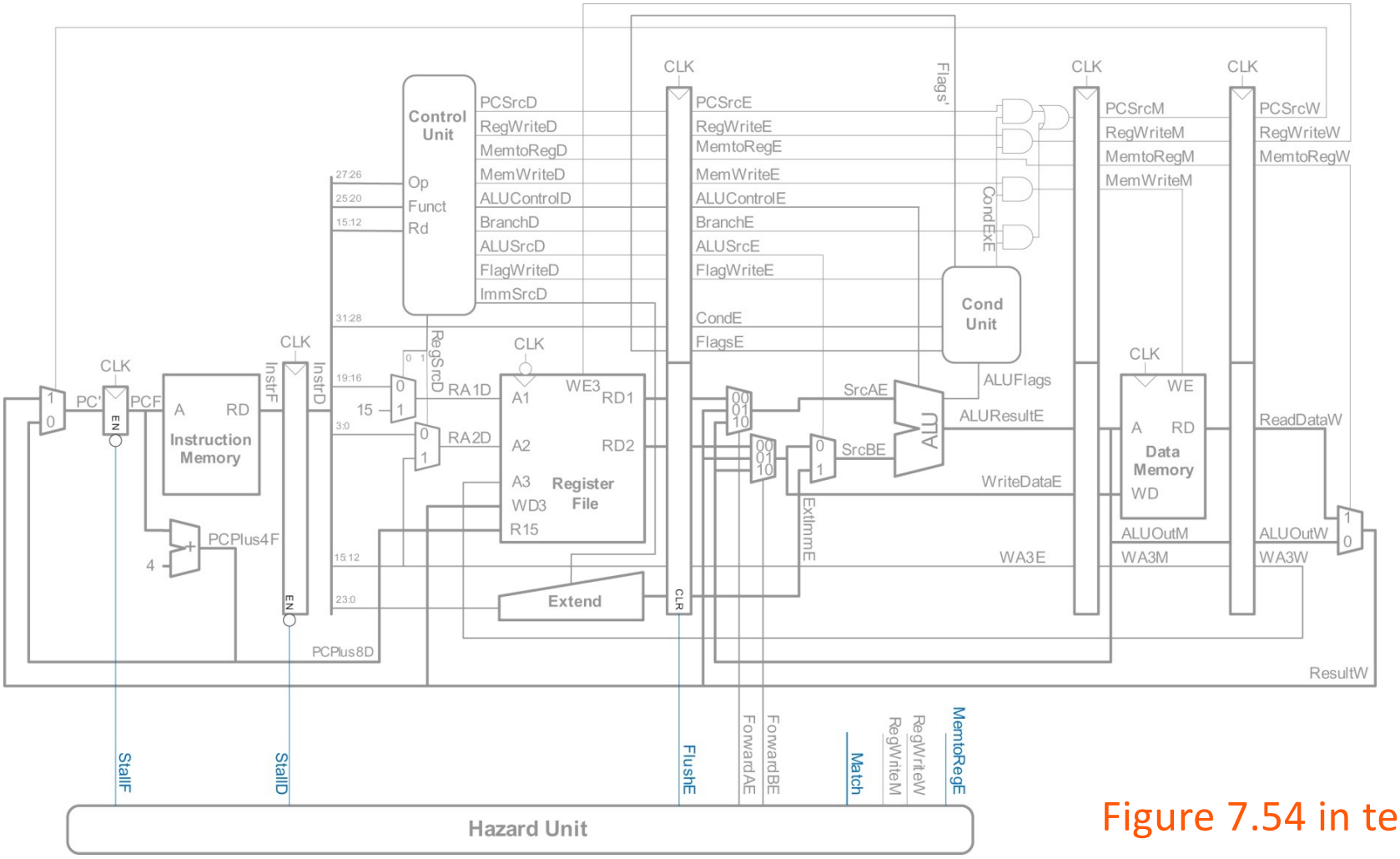


Figure 7.54 in textbook

Control Hazards

- Control hazards are due to changes to sequential control flow
  - Branch (`B`) instructions
  - Writes to `PC` (`R15`) by regular instructions

- The pipelined processor **does not know** which instruction to fetch next

- Branch decision **has not been made** by the time instruction is fetched

Solving Control Hazards

- There are two solutions

- Stall the pipeline on a branch instruction
  - Instruction is fetched in the first stage
  - Branch is resolved in the last (fifth) stage
  - Four stall cycles is a very high penalty for a branch

- Predict the branch outcome (aka. **branch prediction**)
  - If the outcome is correct, continue execution (zero penalty)
  - If the outcome is wrong (branch misprediction), clean up the pipeline, and restart from the correct target instruction
  - Branch misprediction penalty depends on when recovery is initiated

Simplest Branch Predictor

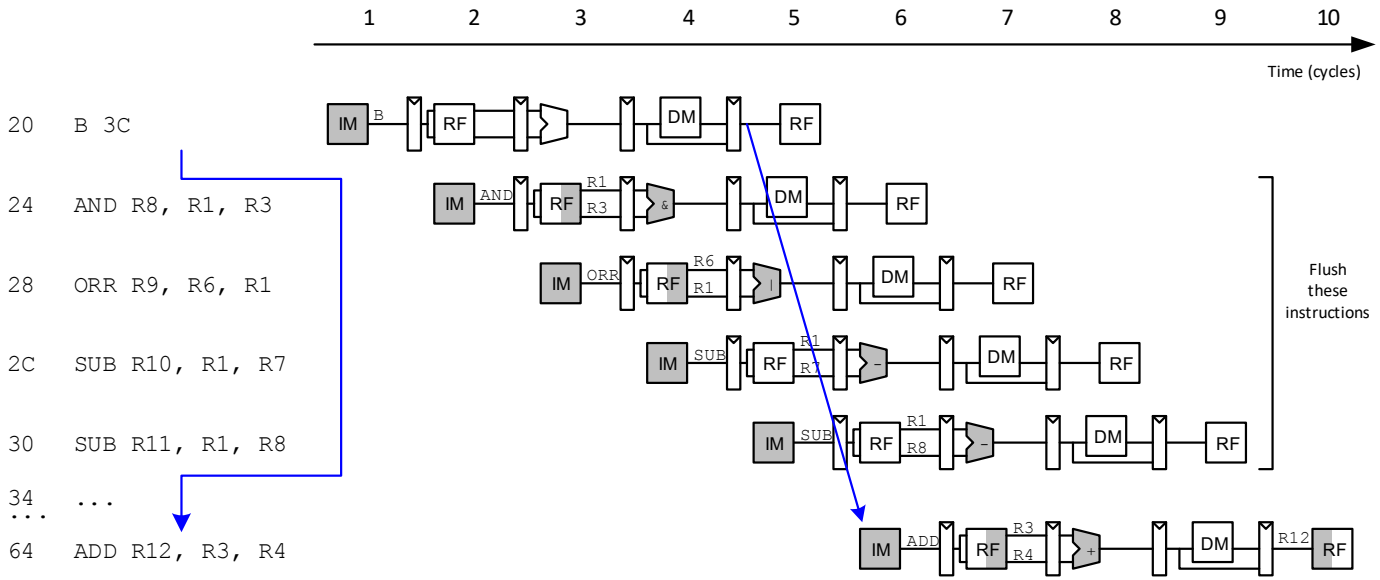
- Predict-always-untaken (eager)

- Could be predict-always-taken (lazy)
  - Drawback is CPU “always stalls” for four cycles
  - No other option but to stall (target address is not available)

- Both predictors above use a **static prediction policy**
- We will look at **dynamic branch prediction** later
  - Different predictions for *different executions* of same branch
  - Takes *recent branch behavior* into account
  - Also predicts the target address

Flushing when Branch is Taken

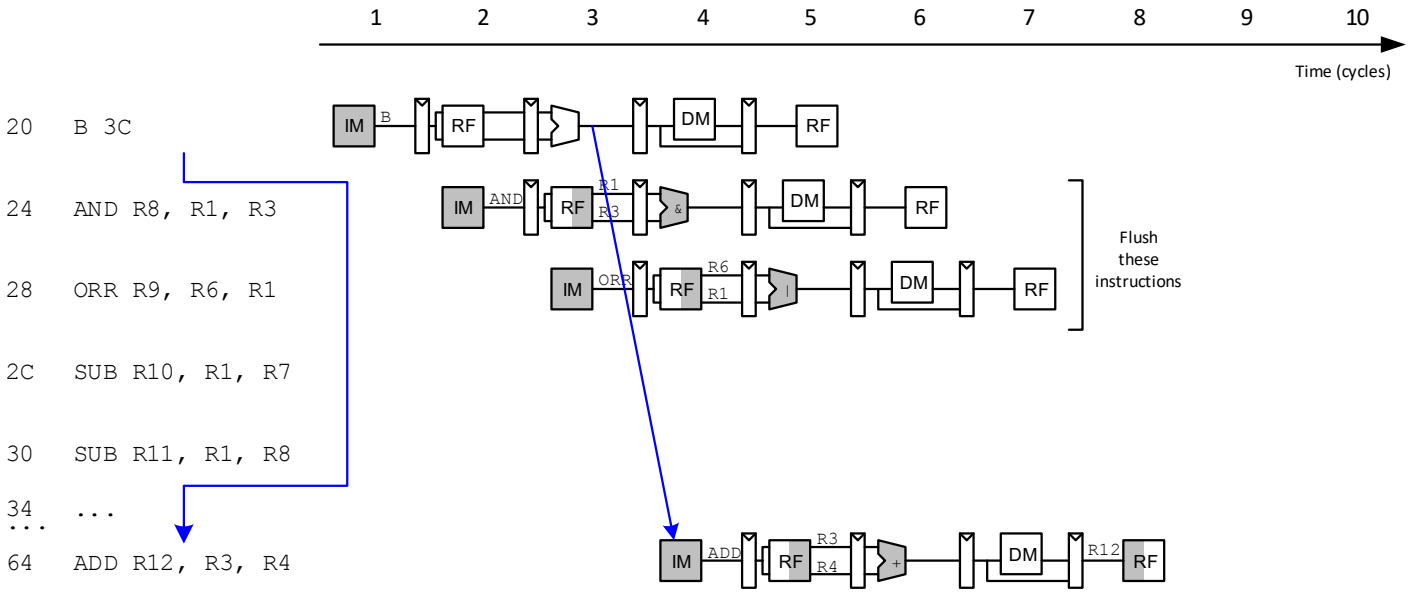
- Fetching the next instruction is an example of predict-always-untaken



- Number of instructions flushed when branch is taken (**4**)
- Misprediction penalty of 4 cycles or 4 wasted instructions
- May be reduced by predicting the branch early

Early Branch Resolution

- The **earliest stage** branch target is known is **EX**
- Update the PC in the next cycle to save two cycles



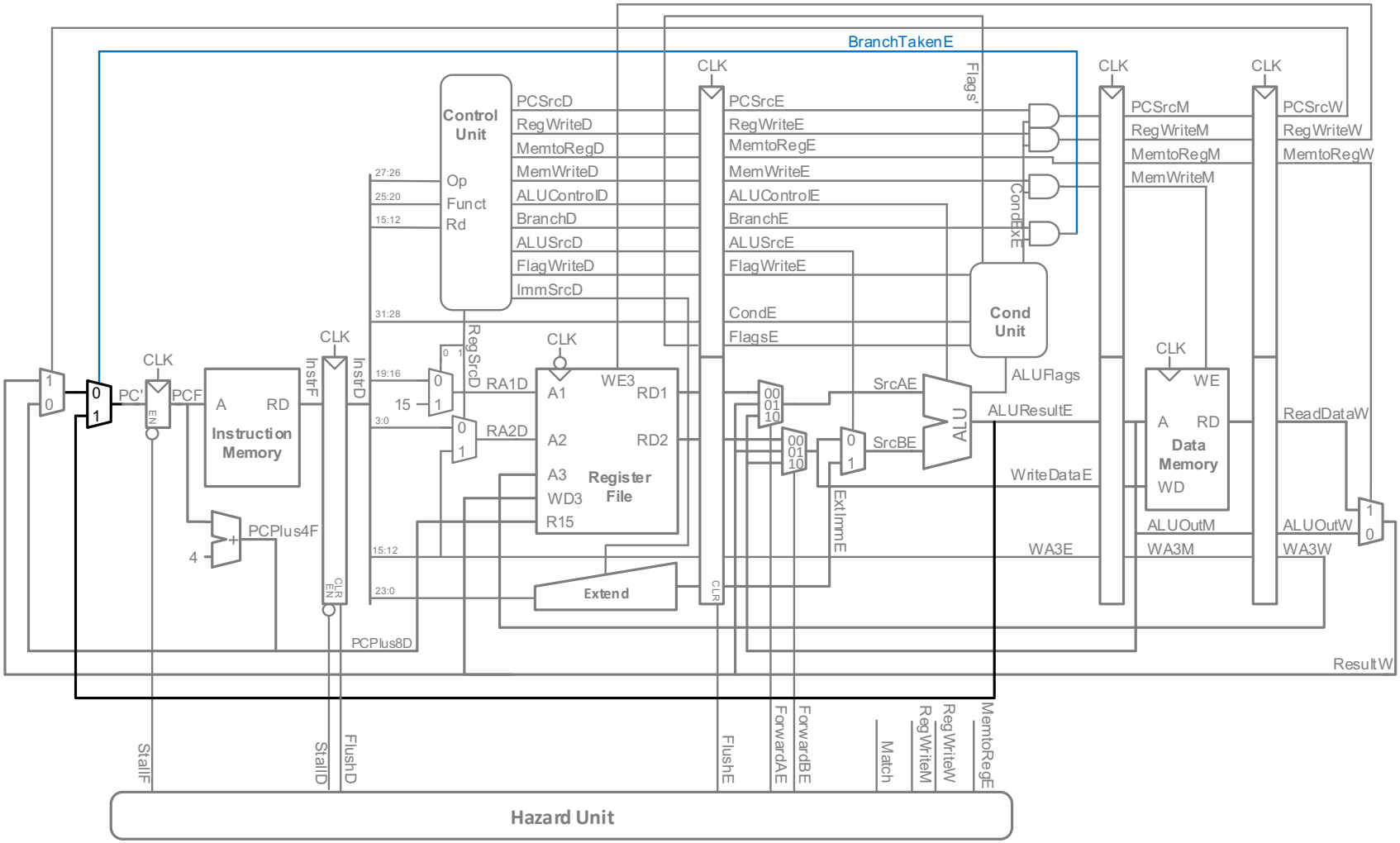
- Flush the two instructions in the **F** and **D** stages

Hardware Changes for Early Resolution

- **Idea:** Determine the branch target address (BTA) in the **EX-stage**
  - Branch misprediction penalty = 2 cycles

- **Hardware changes**
  - Add a branch multiplexer before `PC` register to select BTA from `ALUResultE`
  - Add `BranchTakenE` select signal for this multiplexer (only asserted if branch condition satisfied)
  - `PCSrcW` now only asserted for writes to `PC`

Pipelined Processor with Early BTA



Control Stalling Logic (page # 440 of H&H)

- ***PCWrPendingF* = 1** if write to *PC* in Decode, Execute or Memory

  PCWrPendingF = PCSrcD + PCSrcE + PCSrcM

  PC write is in progress in D, E, M

- **Stall Fetch** if *PCWrPendingF*

  StallF = ldrStallD + PCWrPendingF

  Stall fetch if LDR-Use hazard or PC write in D, E, or M

- **Flush Decode** if *PCWrPendingF* OR *PC* is written in Writeback OR branch is taken

  FlushD = PCWrPendingF + PCSrcW + BranchTakenE

- **Flush Execute** if branch is taken

  FlushE = ldrStallD + BranchTakenE

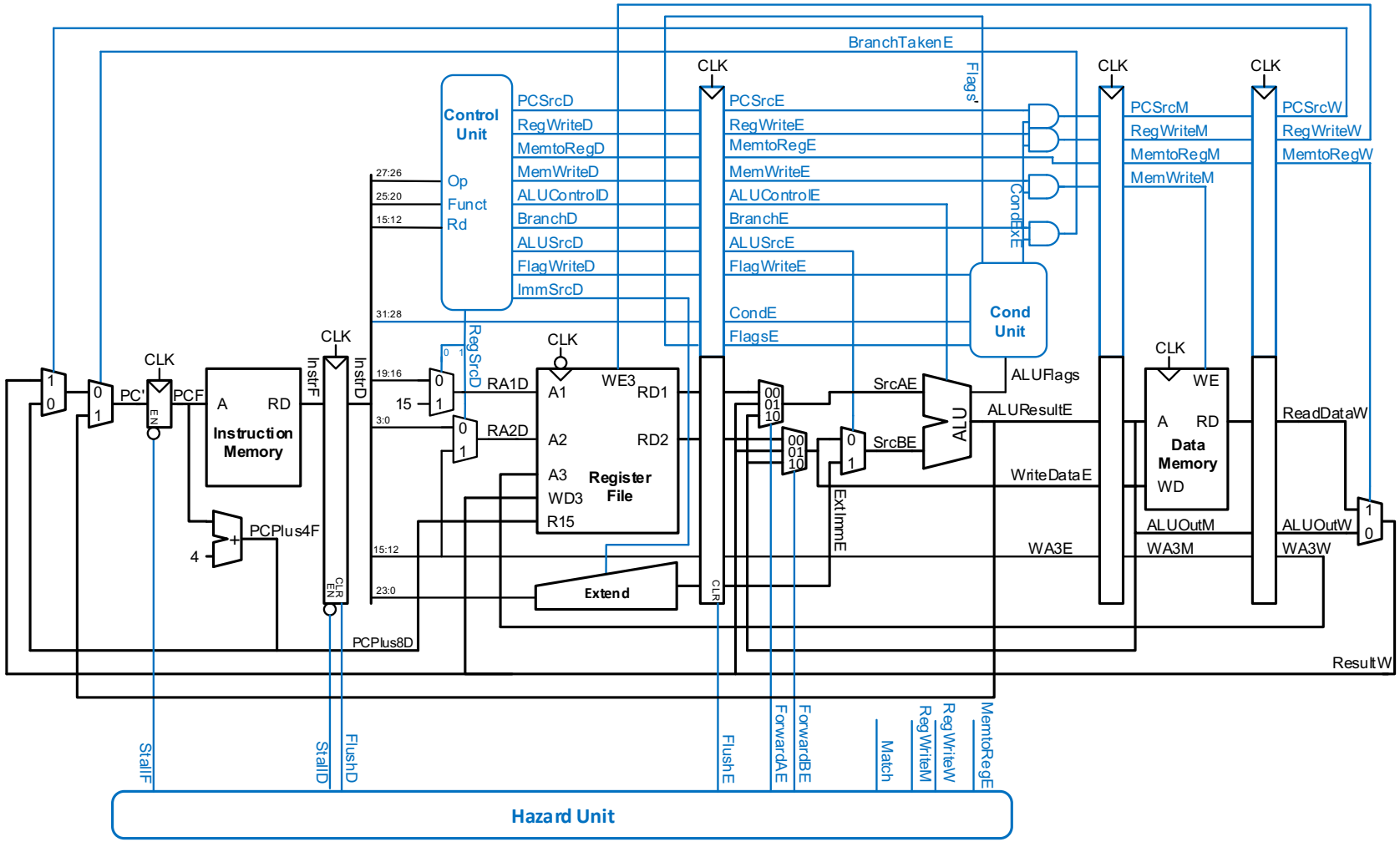
  Flush D if PC write in progress in D, E, M, or W, or branch taken in E

- **Stall Decode** if *ldrStallD* (as before)

  StallD = ldrStallD

  Stall Decode if LDR-Use hazard

ARM Processor with Full Hazard Handling

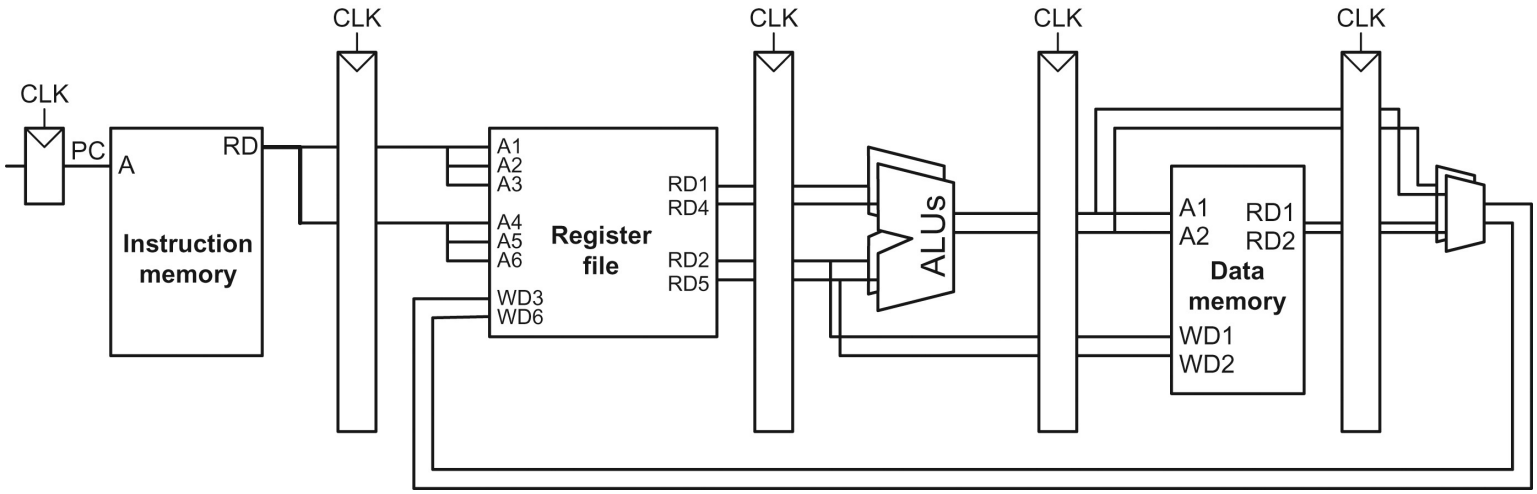


Superscalar Processor

- We have seen pipeline that execute instructions in program order
- **Next:** Increasing instruction throughput beyond this scalar pipeline that issues and finishes one instruction per cycle

Superscalar: Idea and Datapath

- Multiple copies of datapath hardware to execute instructions simultaneously
- **Example:** 2-way superscalar **fetches** and **executes** 2 instructions per cycle

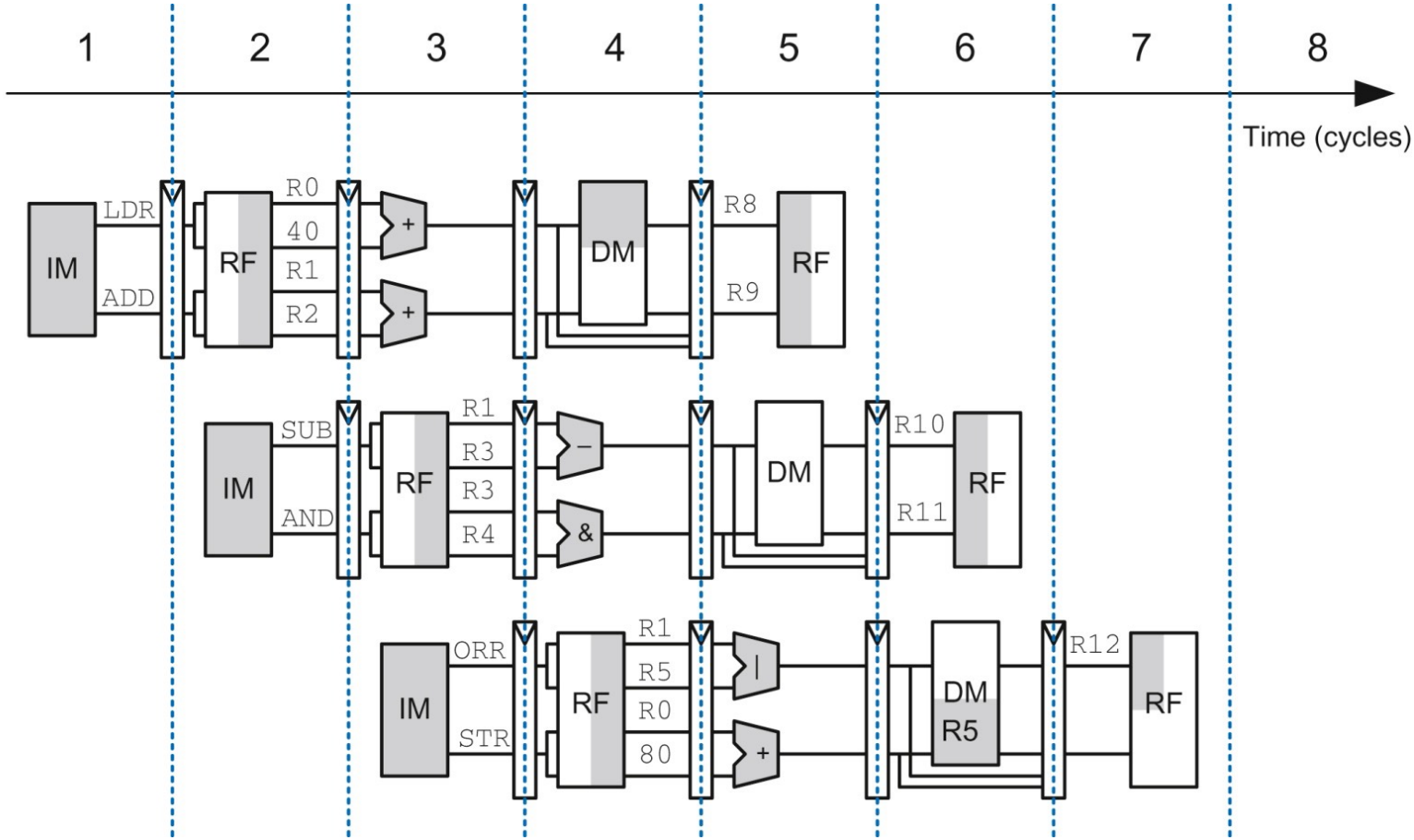


- Requires 6-ported register file (4 reads, 2 writes), 2 ALUs, 2-ported data memory
- Ideal CPI = 0.5 and IPC = 2
- Dependencies and hazards inhibit ideal IPC
- Above figure does not show forwarding and hazard detection logic

Superscalar: Pipeline Operation

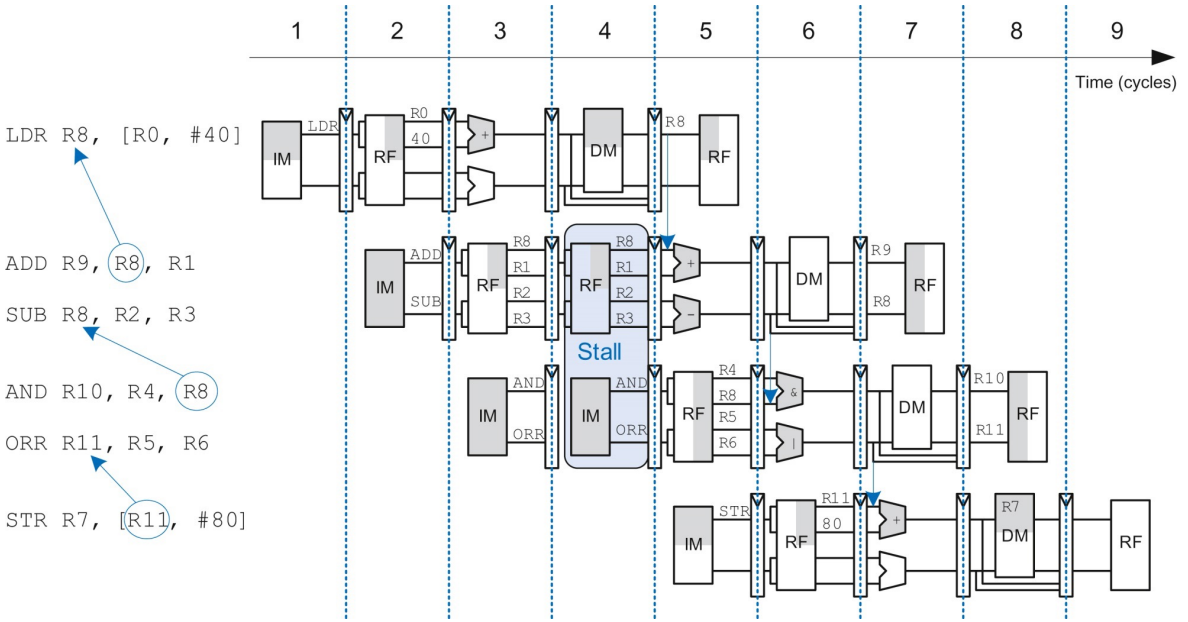
- Example program where IPC = 2 is possible

```
LDR R8, [R0,#40]
ADD R9, R1,R2

SUB R10, R1, R3
AND R11, R3, R4

ORR R12, R1, R5
STR R5,[R0,#80]
```



Superscalar: Impact of Dependencies

- Example of program with data dependences



- The program requires 5 cycles to **issue** six instructions with an IPC of 1.2

In-Order Superscalar: Tradeoffs

- Superscalar processors encompass spatial + temporal parallelism
  - Pipelining
  - Multiple execution units (or lanes)

- Three, four, and six—way superscalars are common

- Too many dependencies (data + control) in real programs
  - Hard to find many instructions to issue **(in order)** every cycle
    - Out-of-order processor unlocks this bottleneck

- Large number of execution units and complex forwarding and hazard detection logic costs area, power, and energy