COMP2300-COMP6300-ENGN2219
Computer Architecture

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

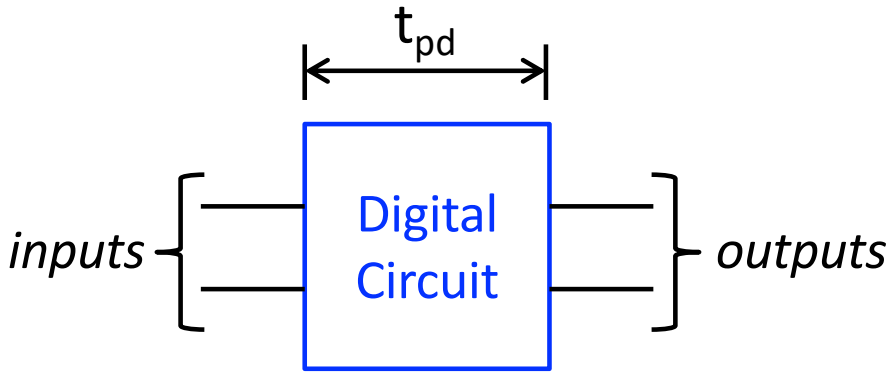
Australian
National
University

General Idea of

Pipelining

Speed of a Circuit

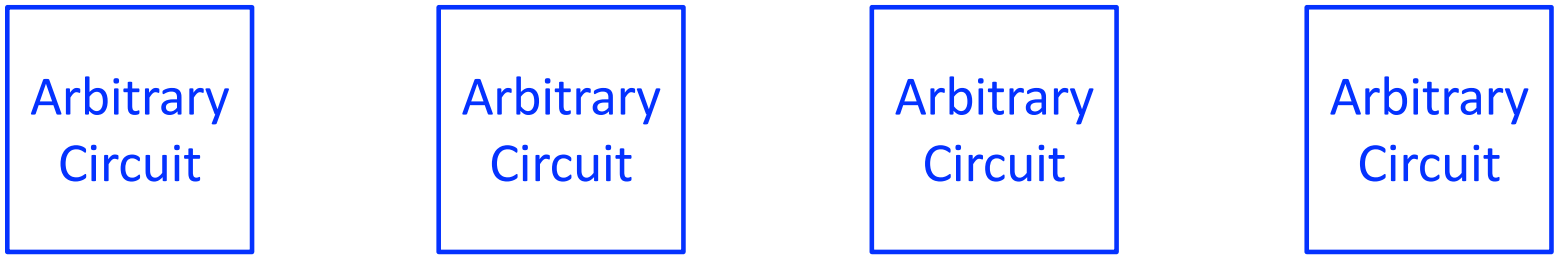
- A digital circuit processes a group of inputs (task) and produces a group of outputs



- **Latency:** Time required to produce one group of outputs
- **Throughput:** Number of input groups processed per unit of time
- **Parallelism** is a key technique for increasing throughput and processing several inputs at the same time

Spatial Parallelism

- **Spatial Parallelism:** Use multiple copies of hardware (circuit) to get multiple tasks done at the same time

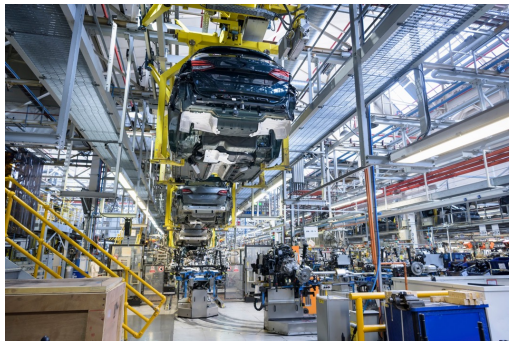


- Suppose a task has a latency of L second
  - **No spatial parallelism:** Throughput is 1/L (one task per L second)
  - **N copies of hardware:** Throughput is N/L (N tasks per L second)
  - Gain in throughput (speedup) = N

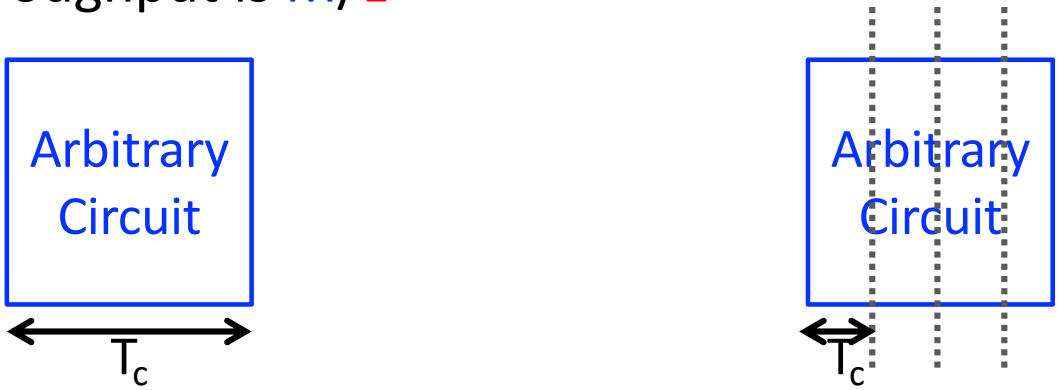
Spatial Parallelism does not reduce the latency of the circuit. We can finish more tasks per unit of time. But each task still takes L seconds

Pipelining

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



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



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

Cookie Parallelism

- What is the latency and throughput for a tray of cookies?

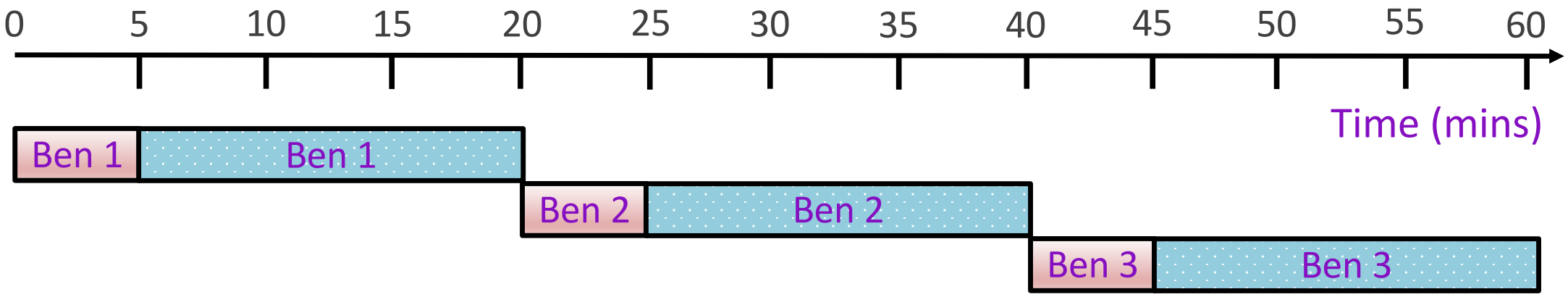
  - Step 1: **Roll** cookies (5 minutes)
  - Step 2: **Bake** in the oven (15 minutes)
    - Once cookies are baked, start another tray

- Latency (hours/tray):
- Throughput (trays/hour):

Cookie Parallelism: Scenarios

- 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

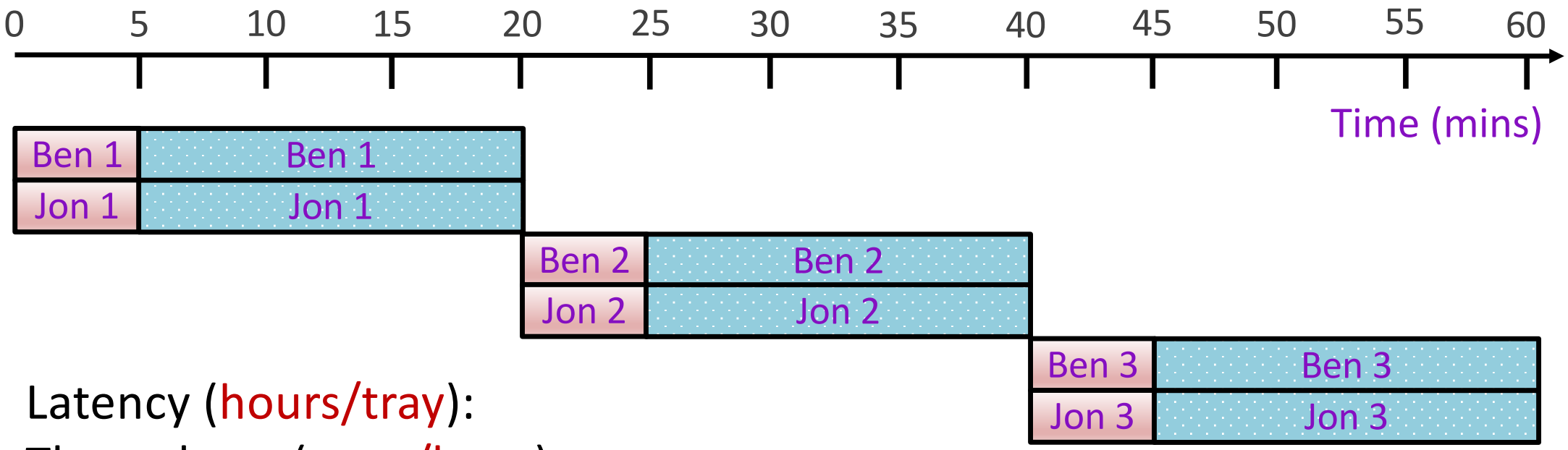
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

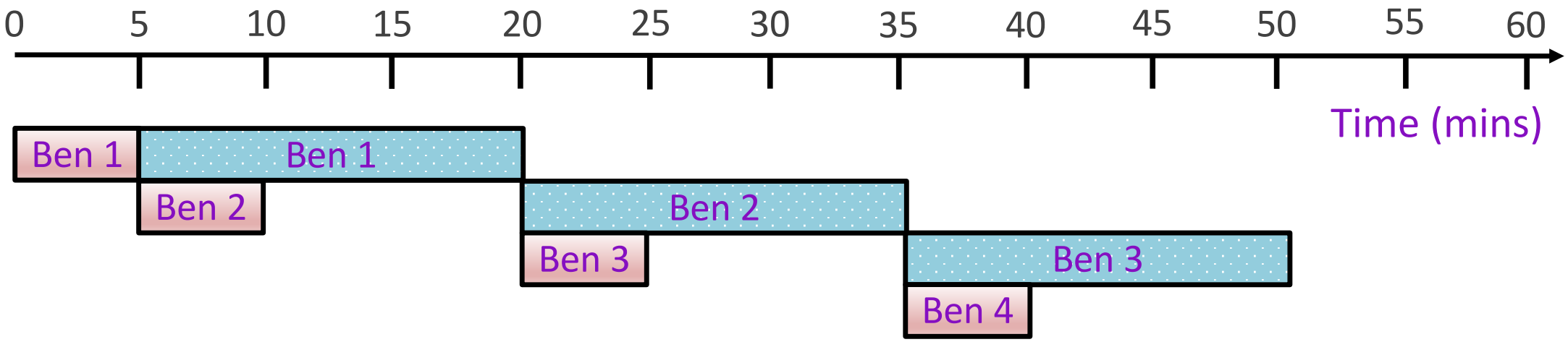


Latency (hours/tray):

Throughput (trays/hour):

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

Pipelining (Ben Only)

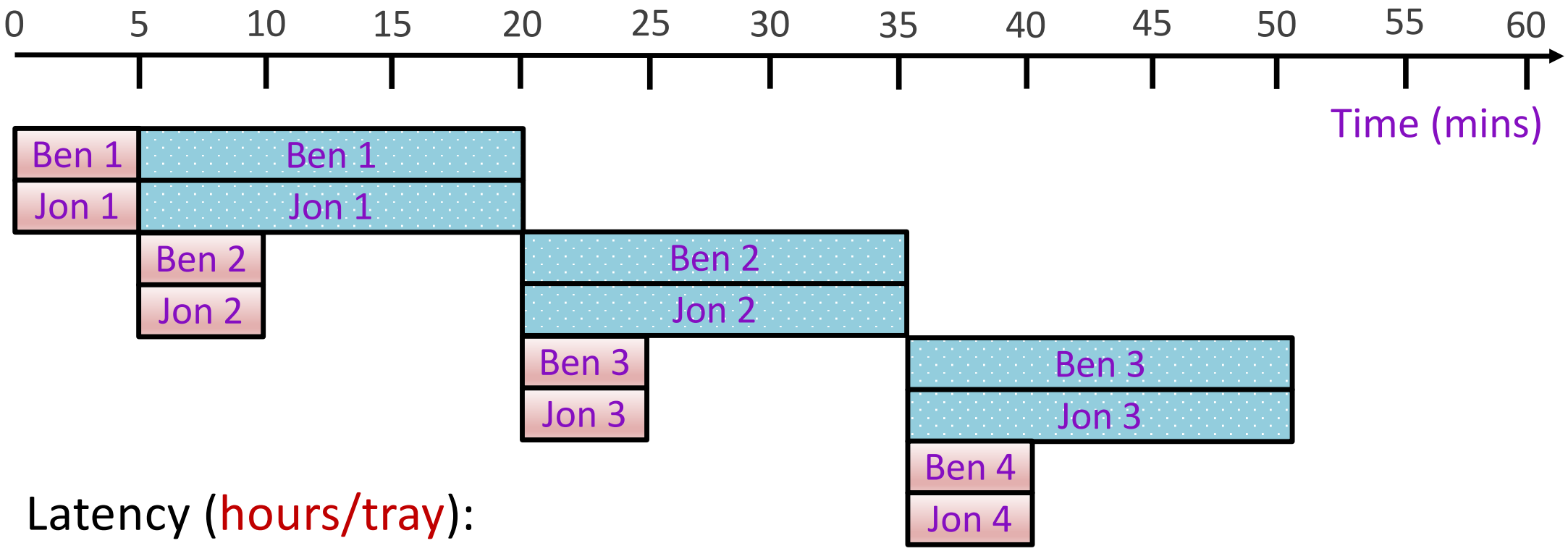


Latency (hours/tray):

Throughput (trays/hour):

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

Spatial + Temporal Parallelism



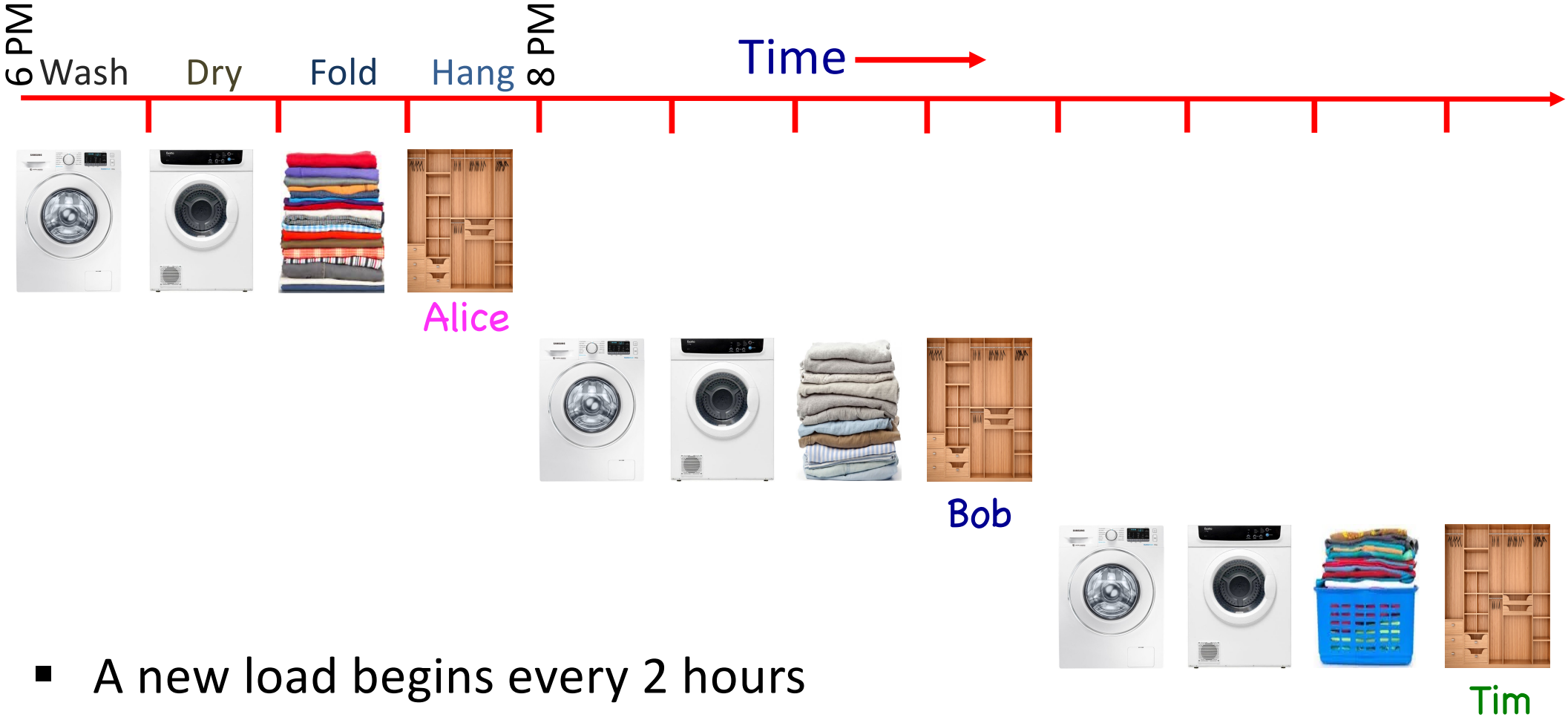
Latency (hours/tray):

Throughput (trays/hour):

Answers Explained

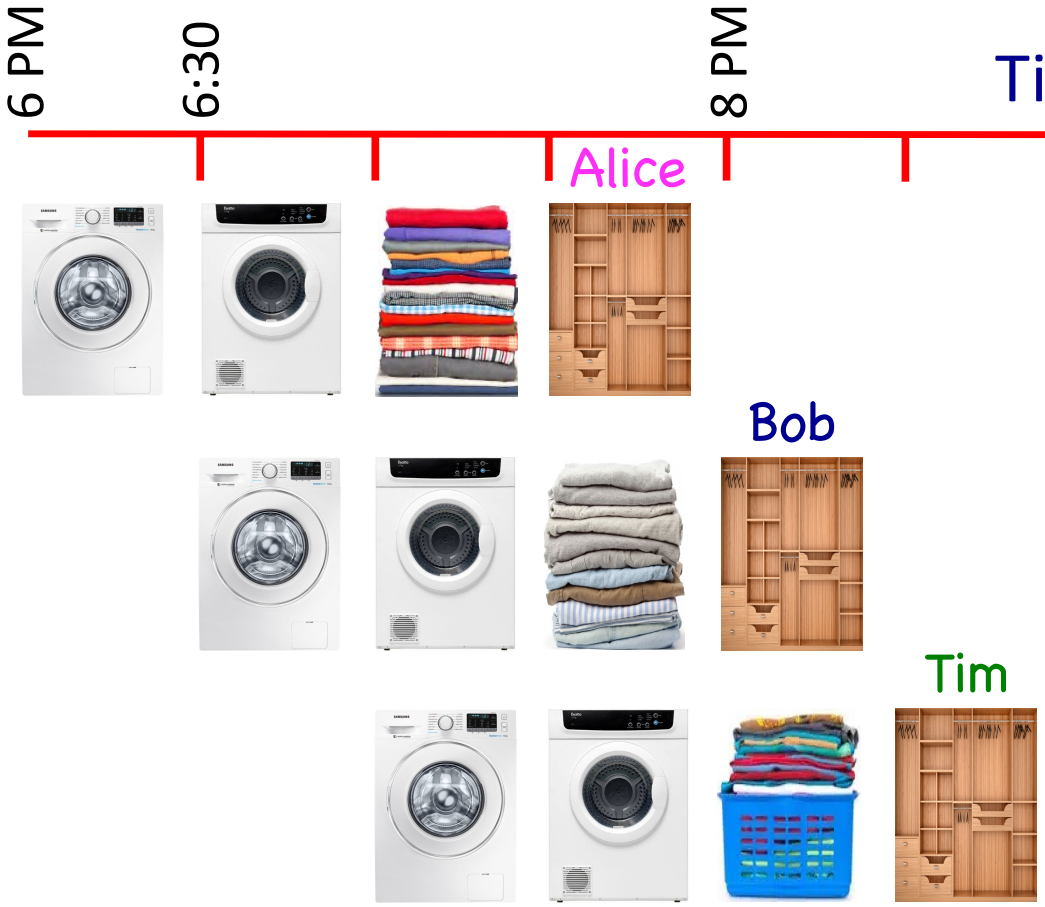
- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a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



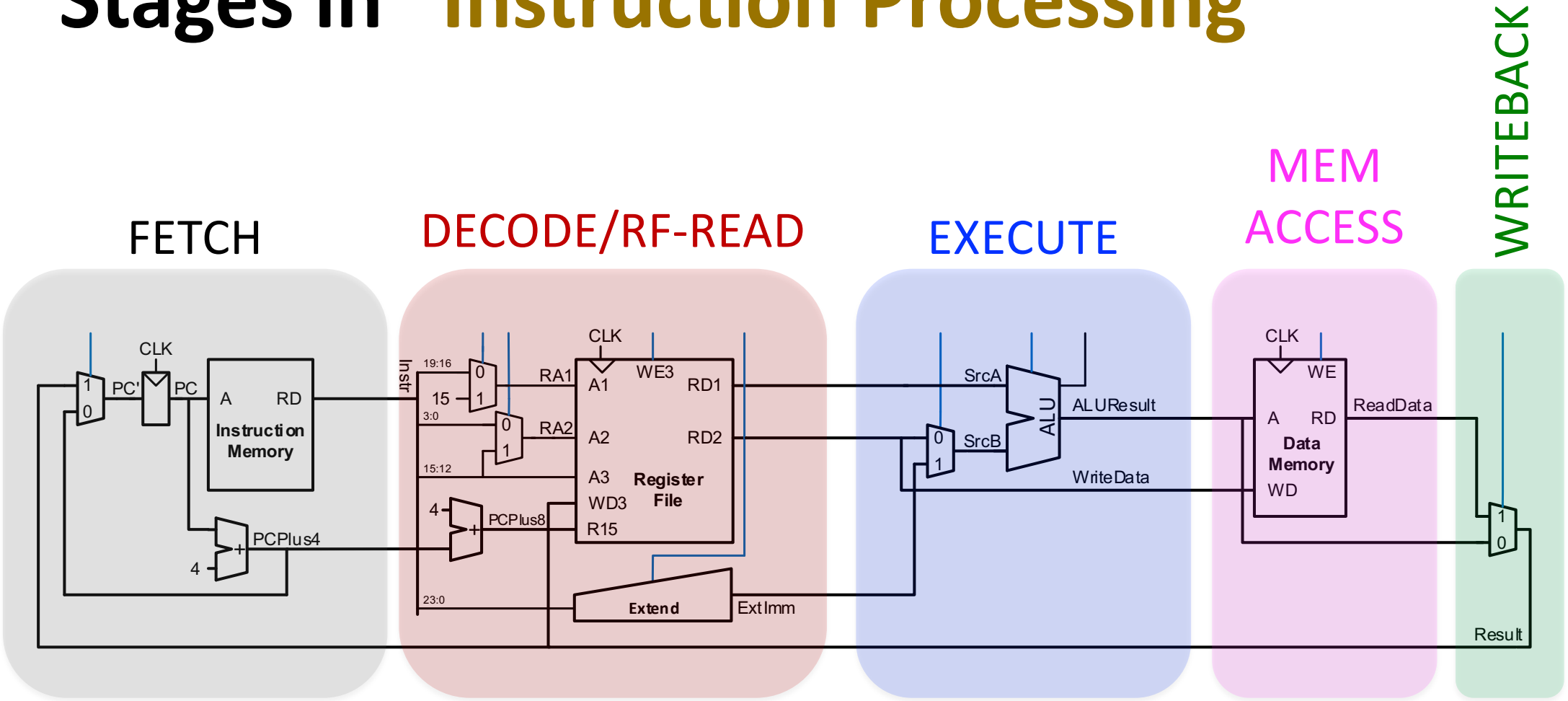
- A new load begins every 30 mins → speedup of 4

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)

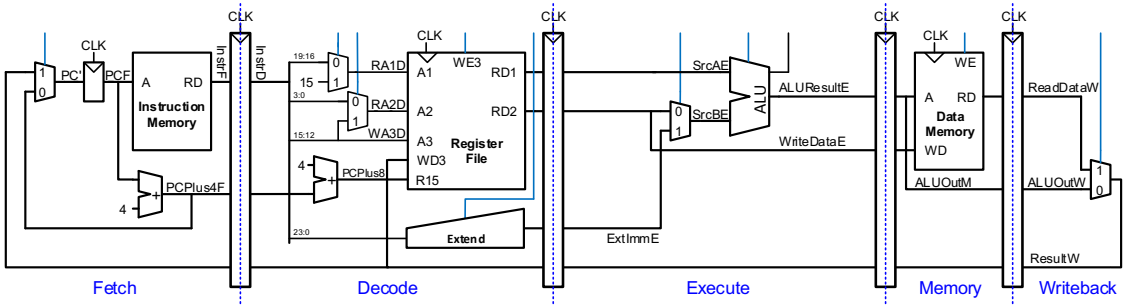
- The work of a stage should be preserved in a PPR each cycle
- PPR acts as a source of data the next stage needs in a subsequent cycle
- If any subsequent stage down the pipeline needs data from an earlier stage it must be passed through the PPRs
  - .... Things don't always go smoothly as we shall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

Stages

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

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

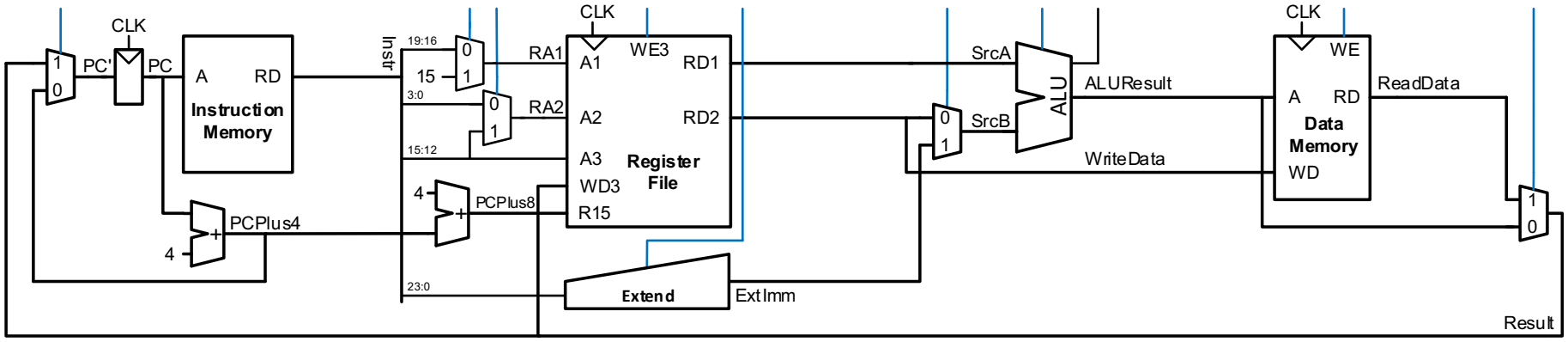


Let’s complete the picture

- Start with the single-cycle microarchitecture

- And insert pipeline registers

Single-Cycle

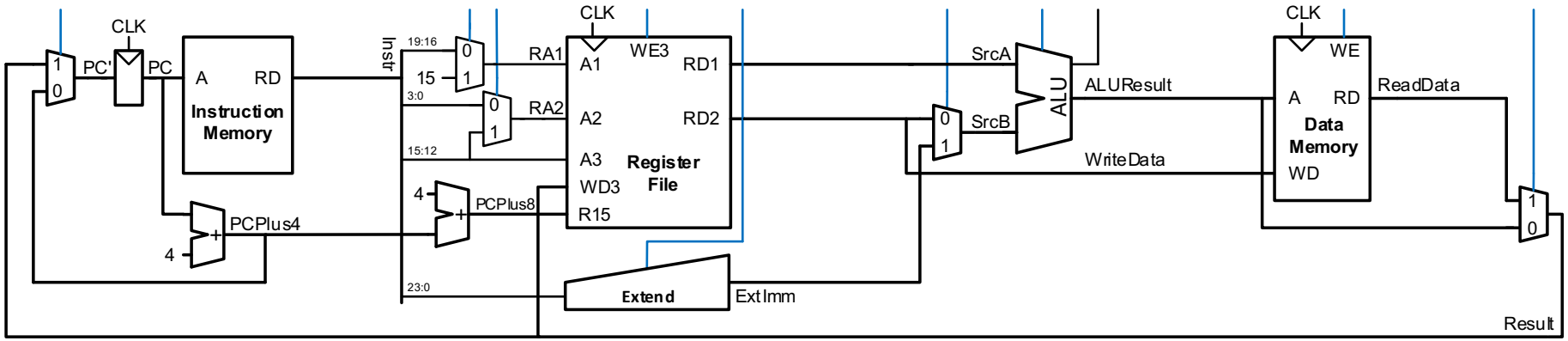


- Once we insert pipeline registers, we would need to pass the results of one stage to the next stage via the pipeline registers

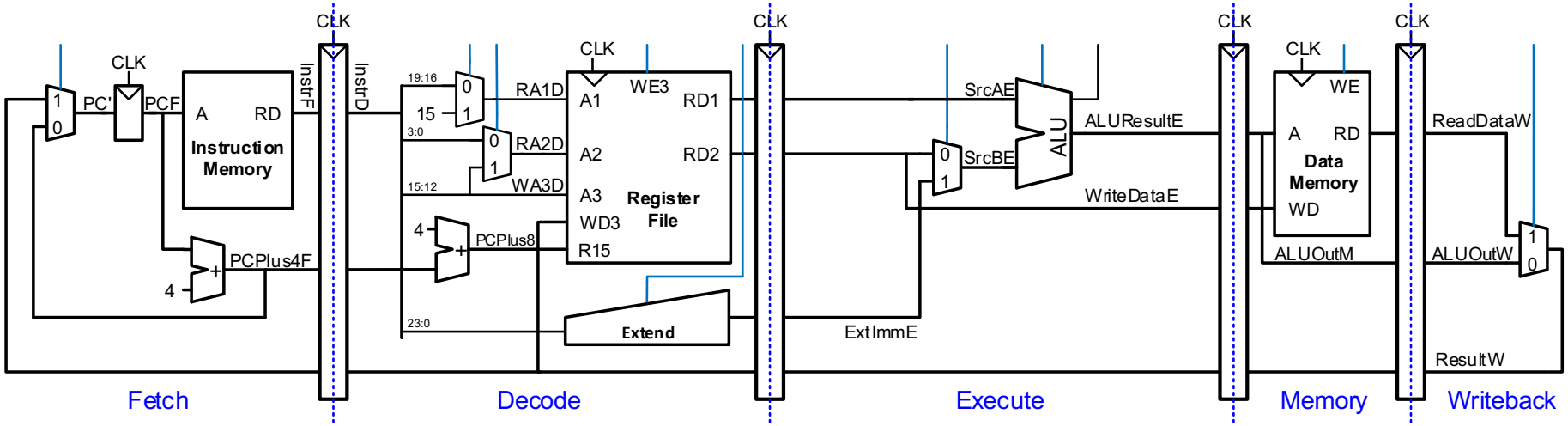
- What is the outcome of the FETCH stage?

Pipeline Microarchitecture

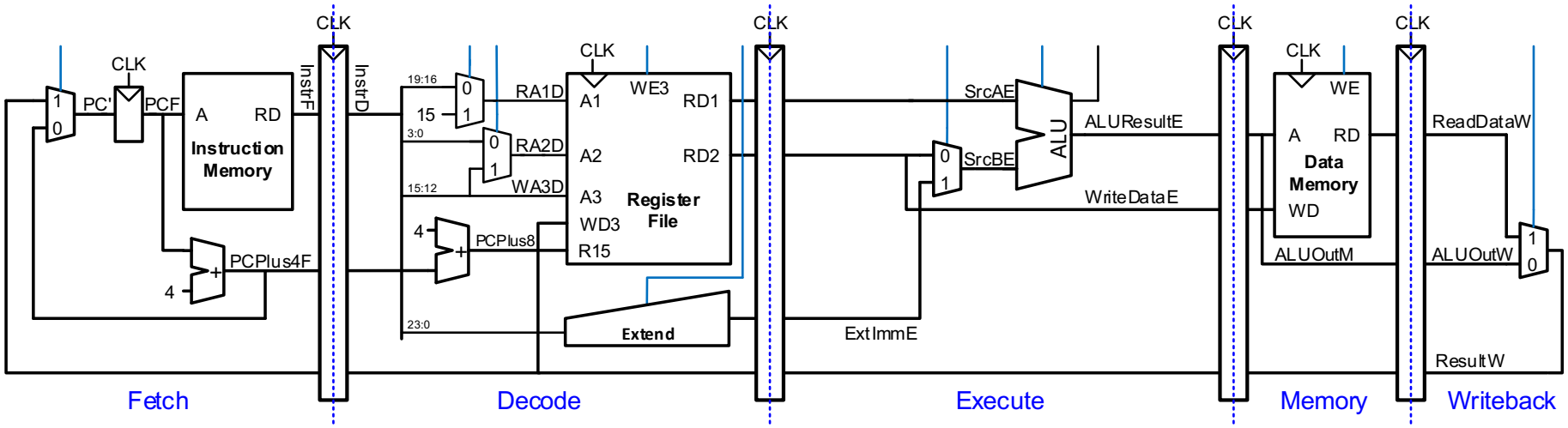
Single-Cycle



Pipelined



Pipeline Microarchitecture



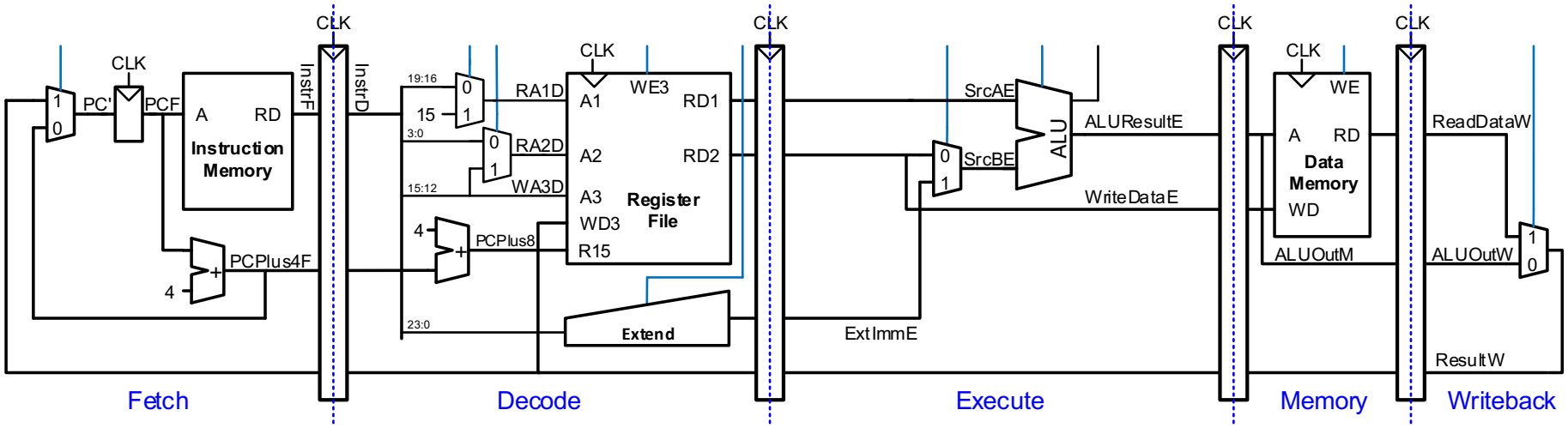
- Stages and their boundaries are indicated in blue
- Signals are given a suffix **(F, D, E, M, or W)** to indicate the stage in which they reside

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]
```

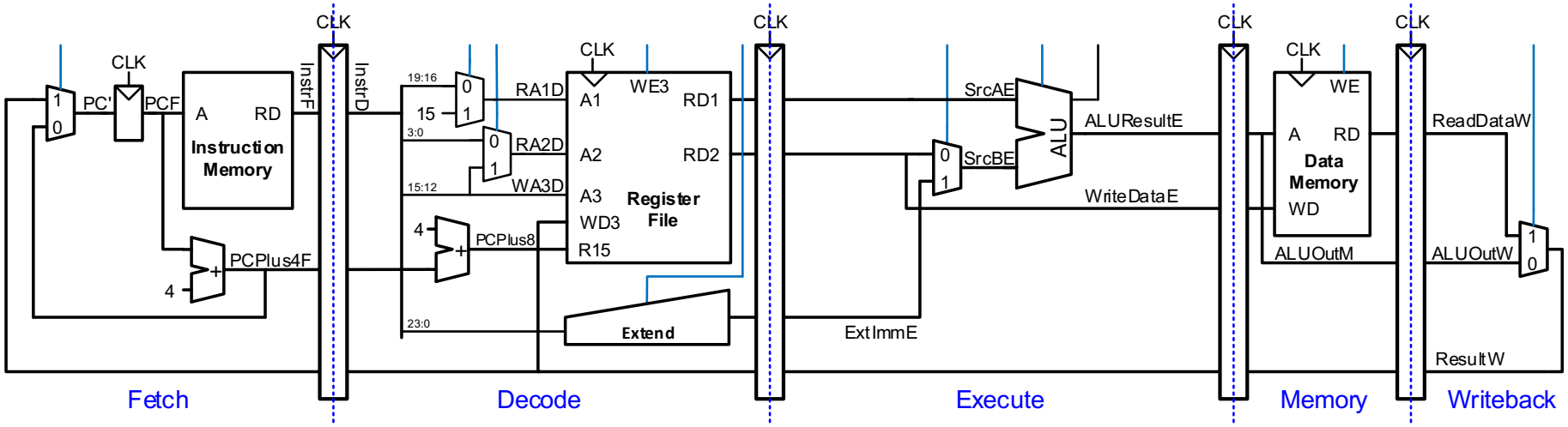
Pipeline Operation: Cycle 1



I1

- Is the pipeline fully utilized? NO

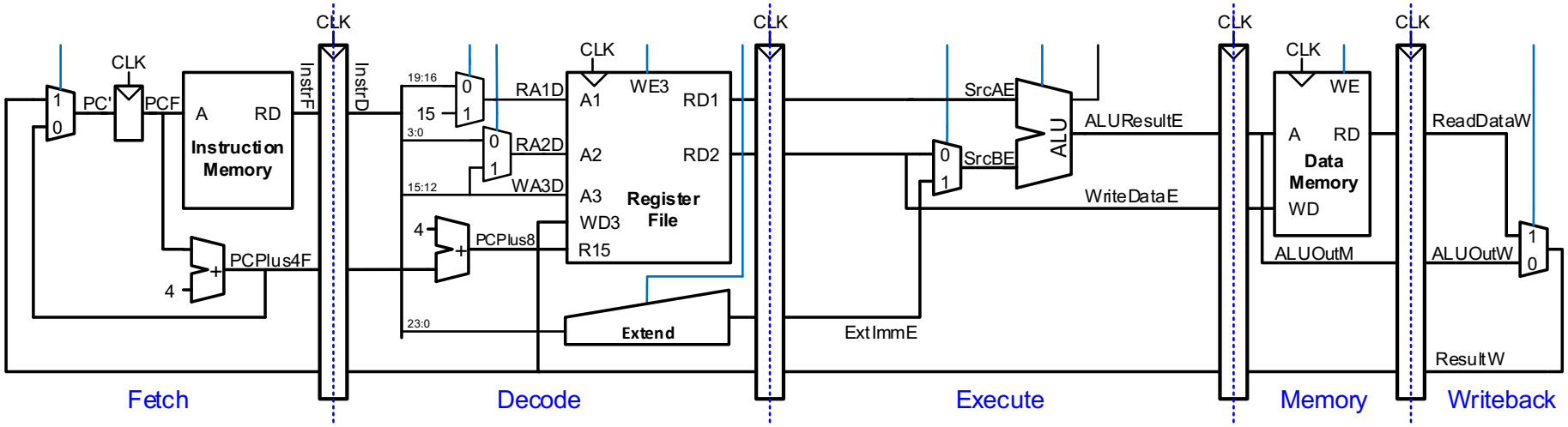
Pipeline Operation: Cycle 2



I2 I1

- Is the pipeline fully utilized? NO

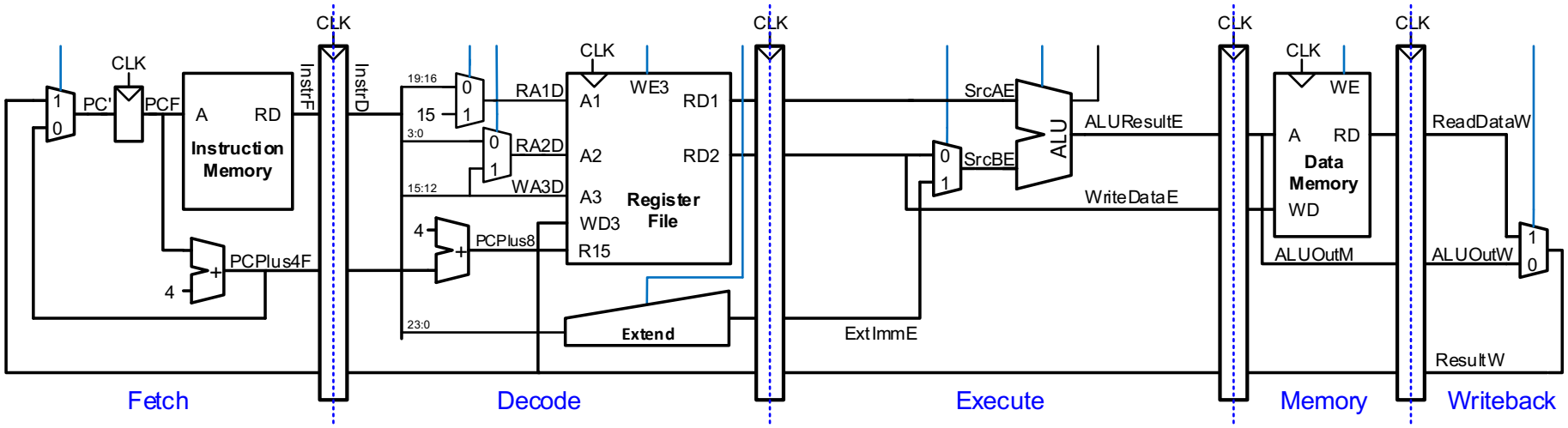
Pipeline Operation: Cycle 3



I3 I2 I1

- Is the pipeline fully utilized? NO

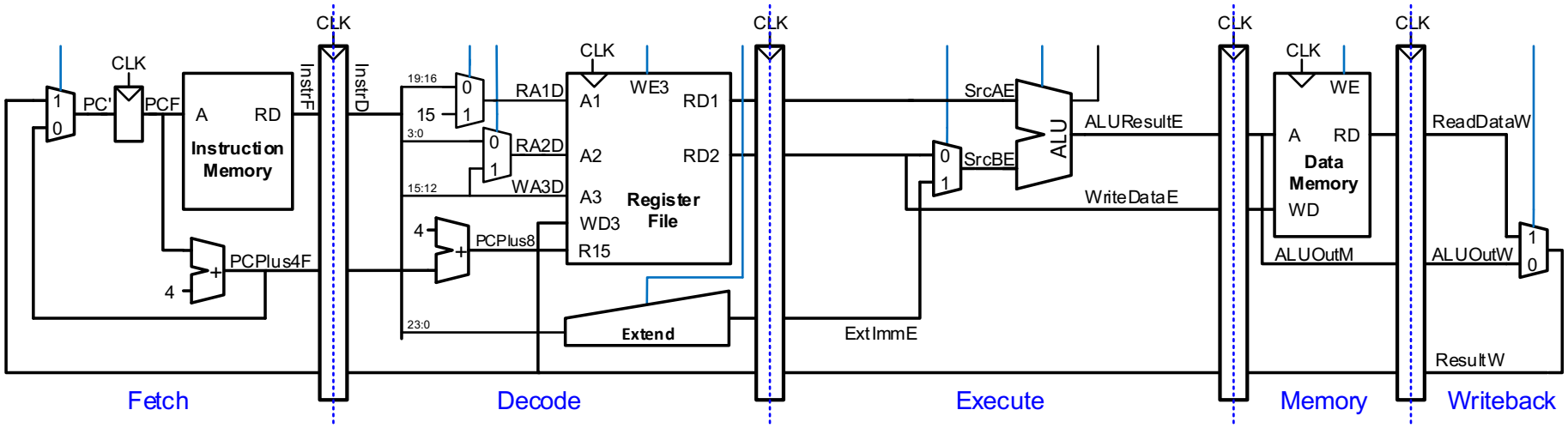
Pipeline Operation: Cycle 4



I4 I3 I2 I1

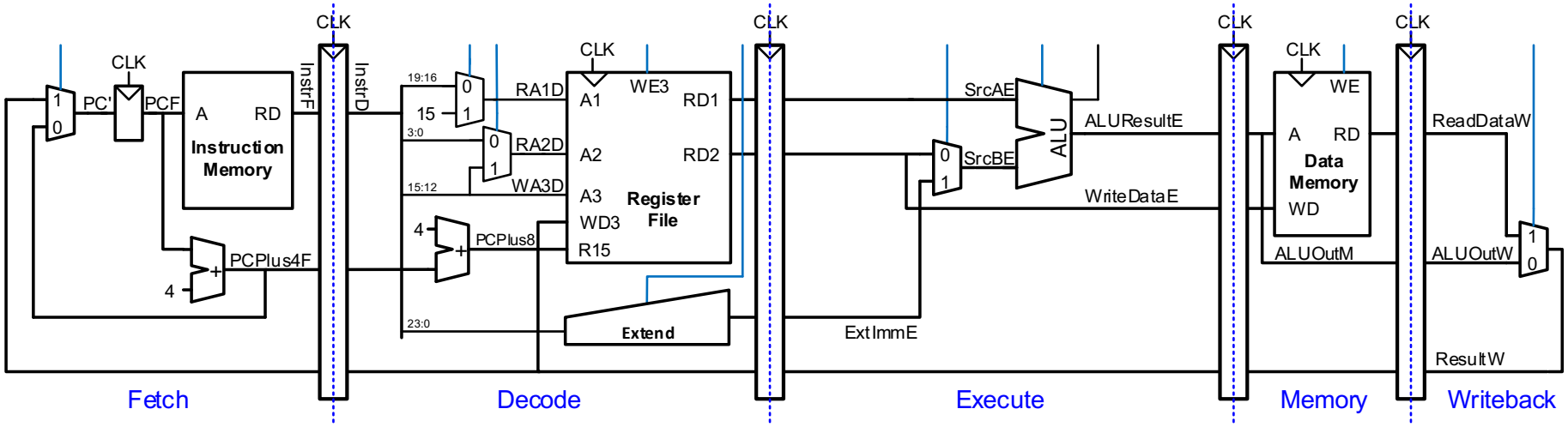
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 5



- Is the pipeline fully utilized? YES

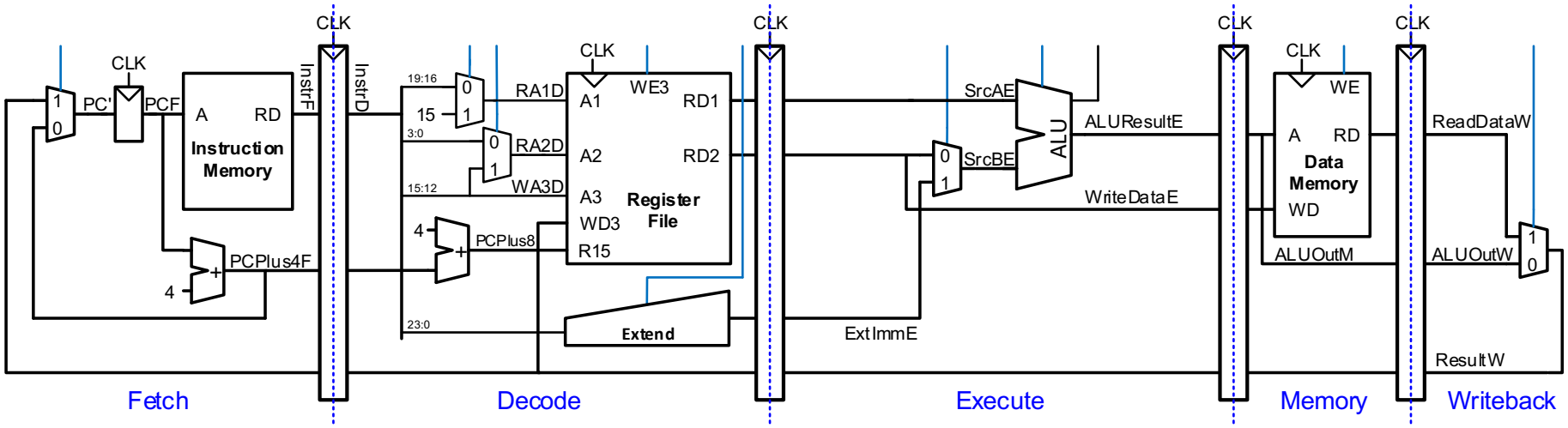
Pipeline Operation: Cycle 6



I6 I5 I4 I3 I2

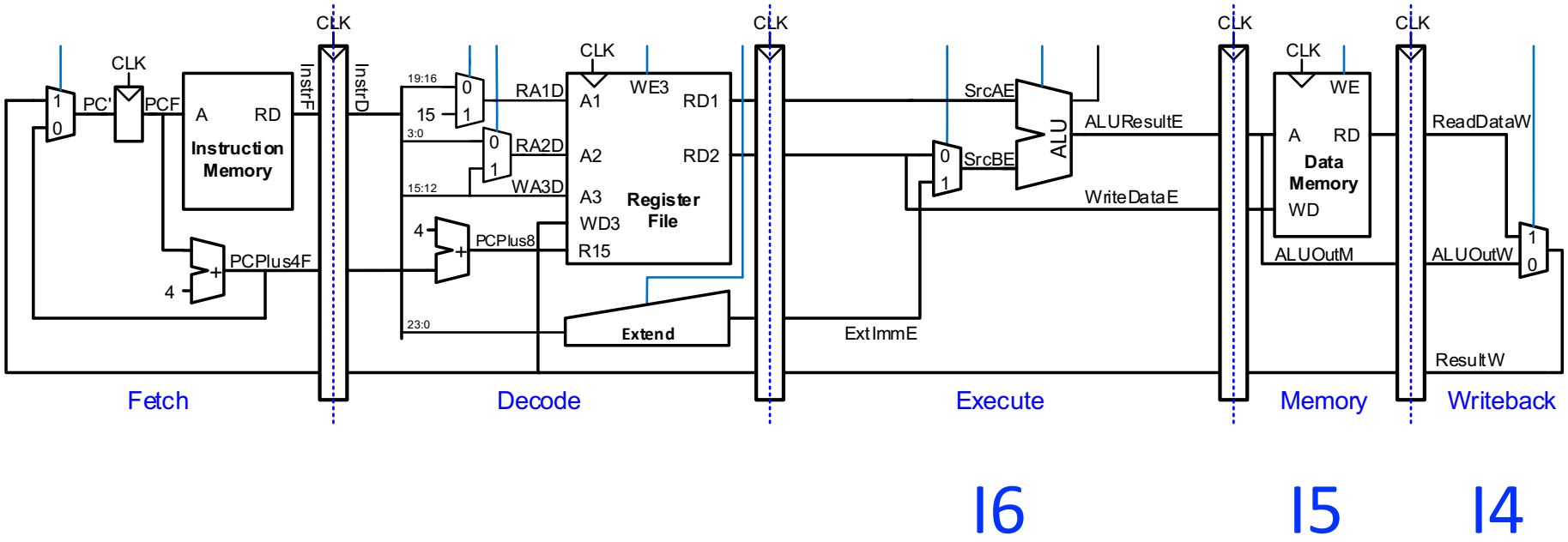
- Is the pipeline fully utilized? YES

Pipeline Operation: Cycle 7



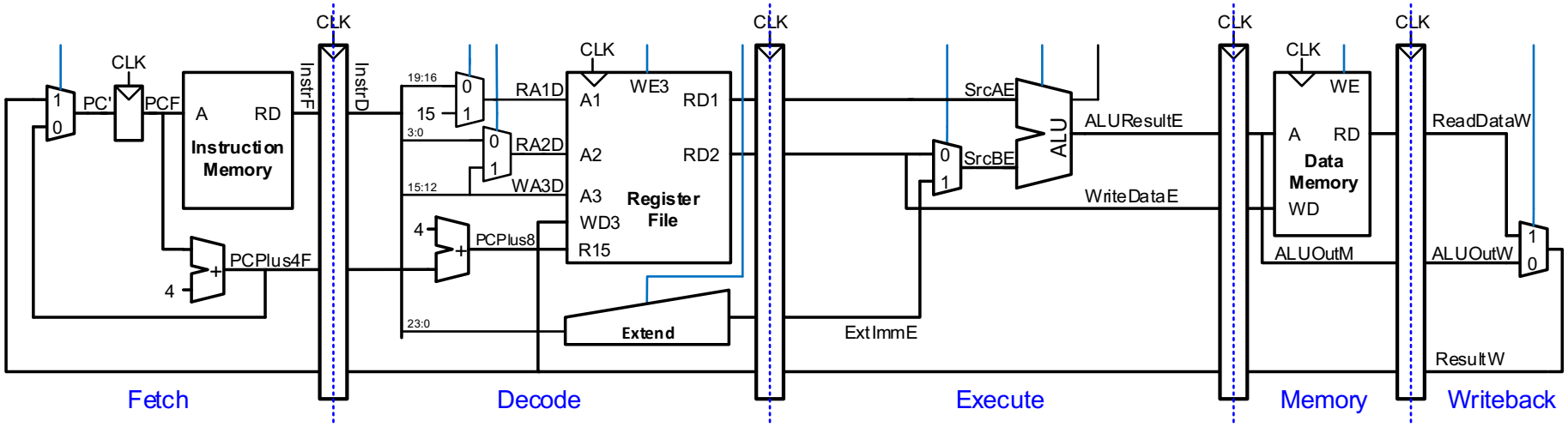
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 8



- Is the pipeline fully utilized? NO

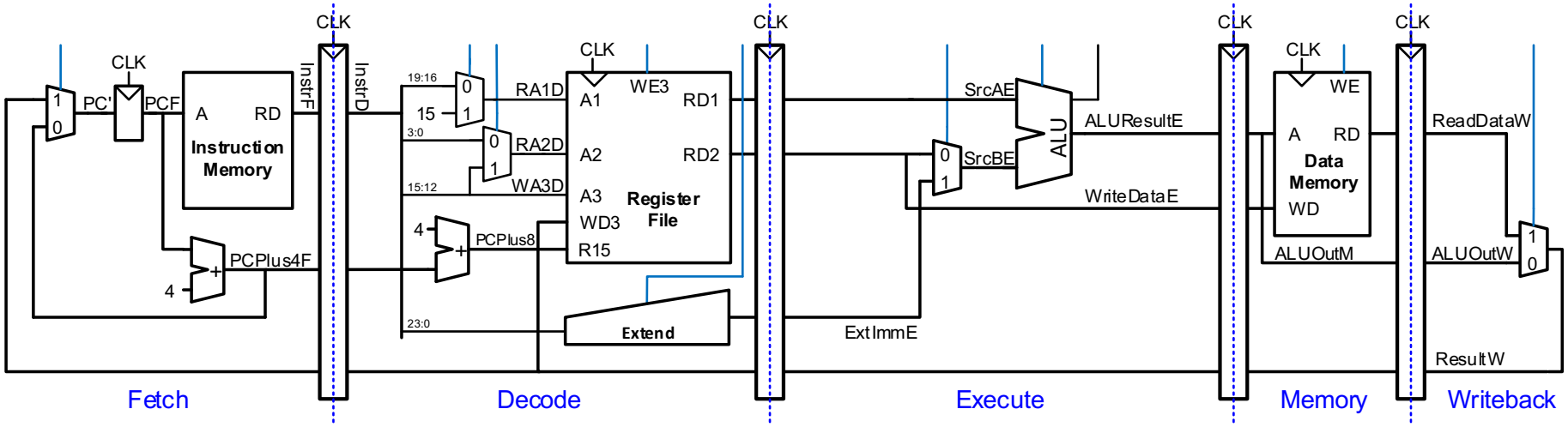
Pipeline Operation: Cycle 9



I6 I5

- Is the pipeline fully utilized? NO

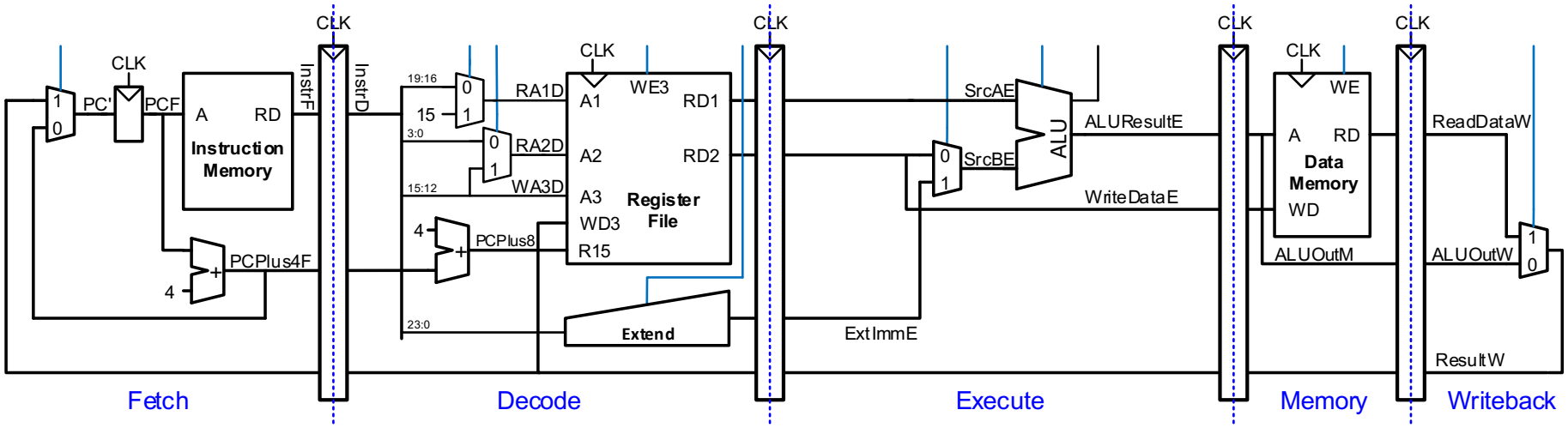
Pipeline Operation: Cycle 10



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 the throughput(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

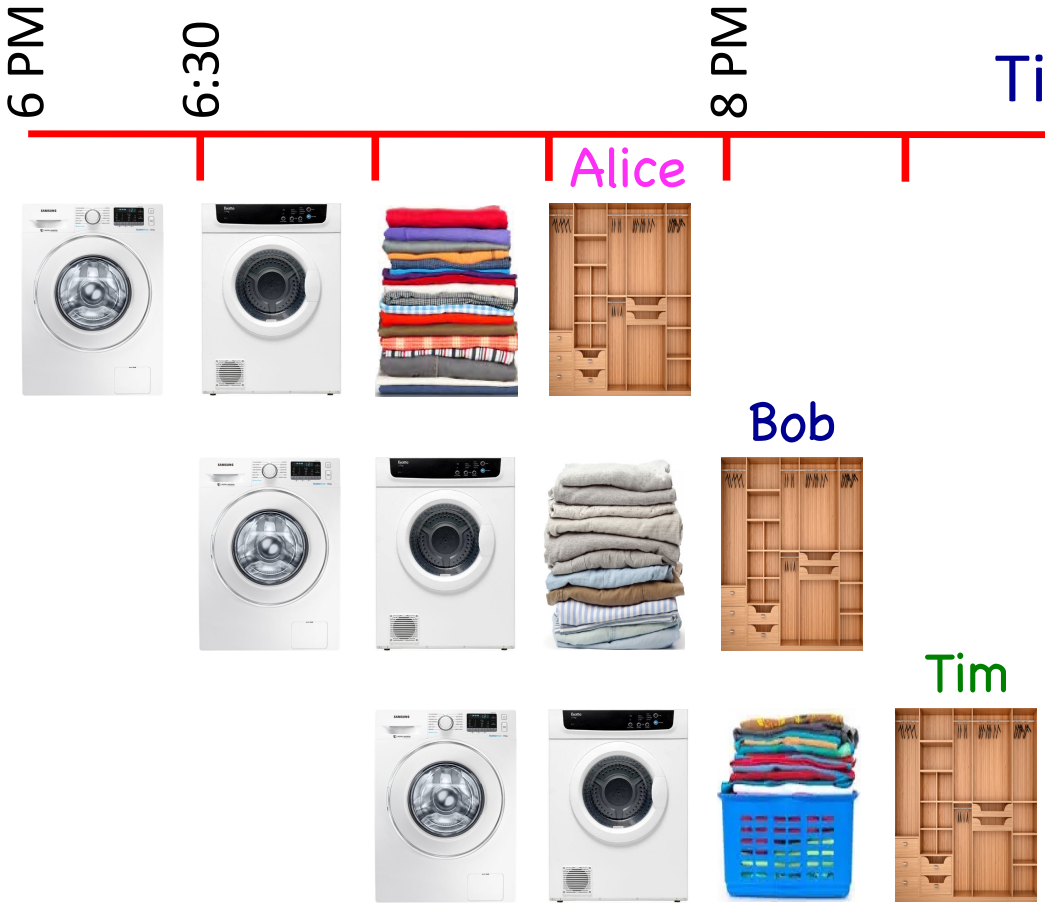
Pipeline Idealism vs. Reality

- **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

Recall: Pipelined Laundry

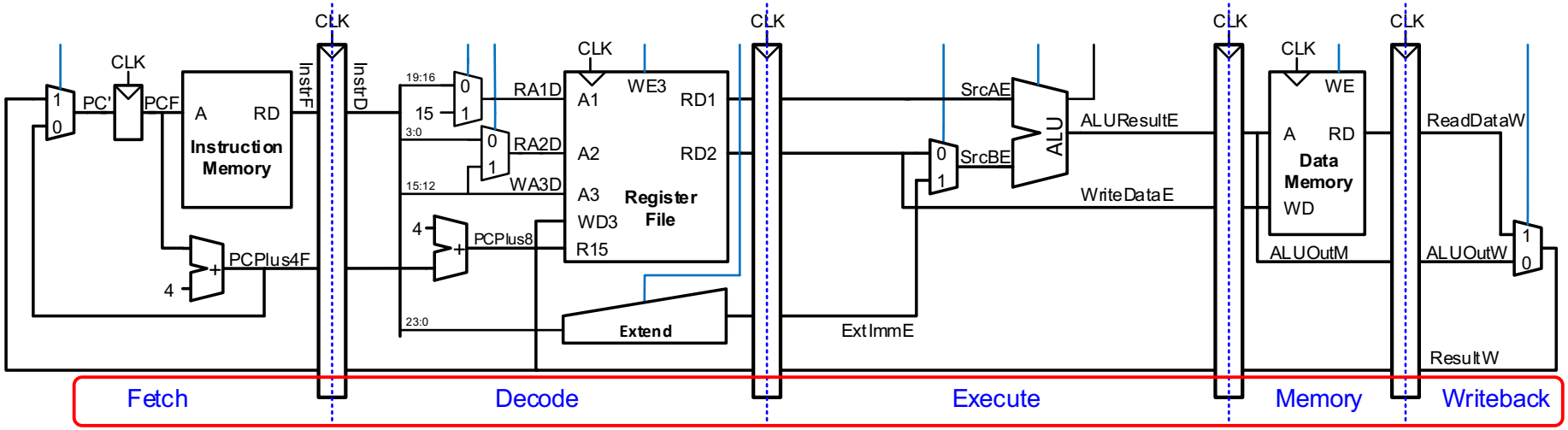


- Ideal speedup = 4, Actual speedup = 2

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

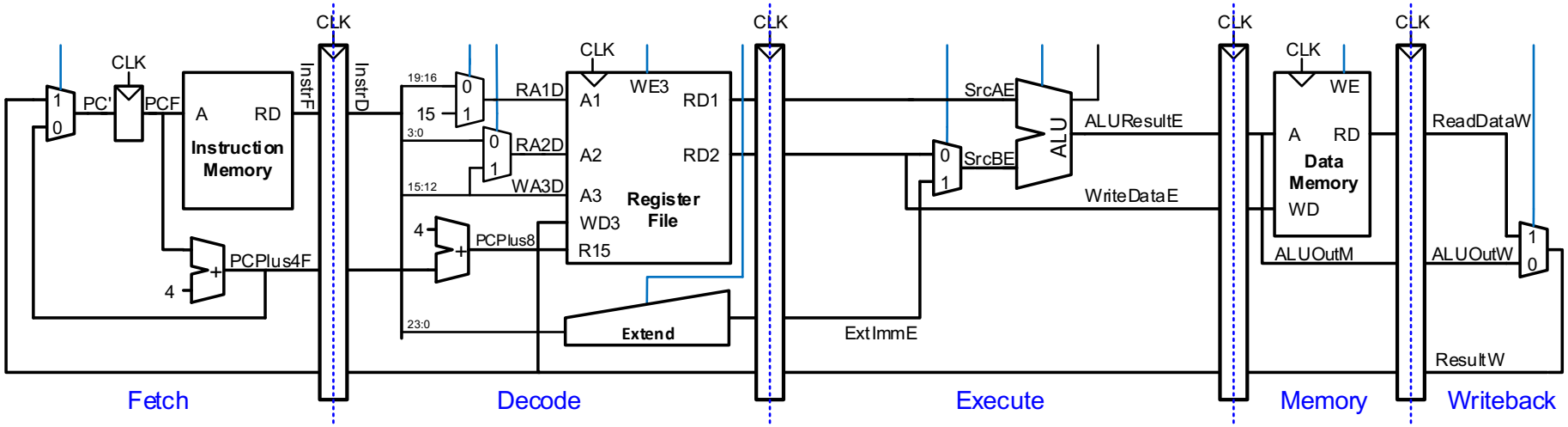
Pipelined Data



- **From Fetch to Decode:** Instruction 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?**

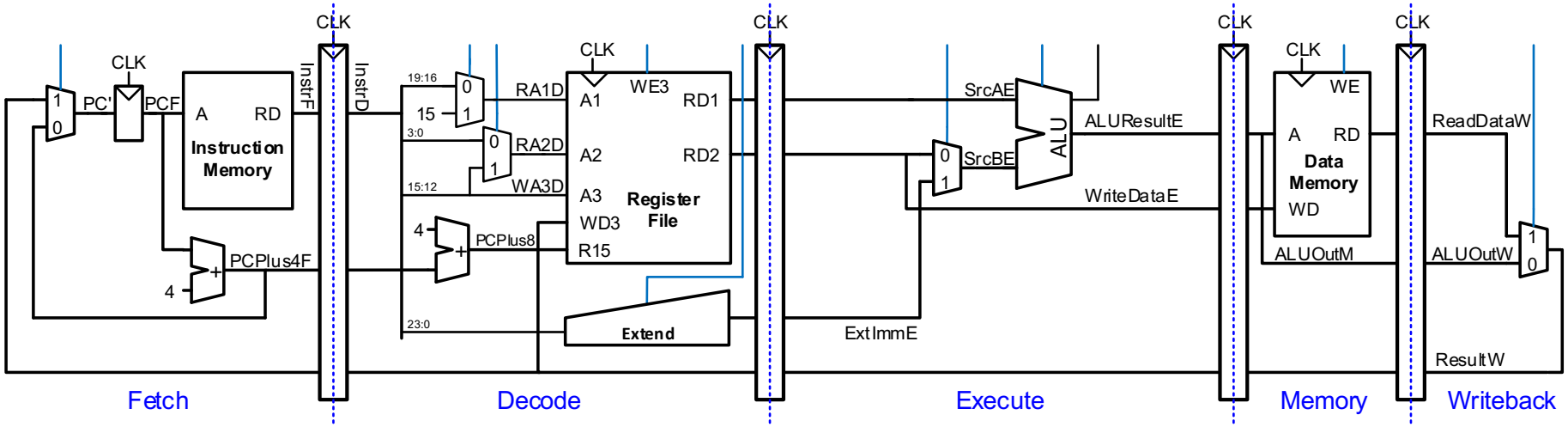
Bug in Pipelined Hardware!



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

Bug in Pipelined Hardware!

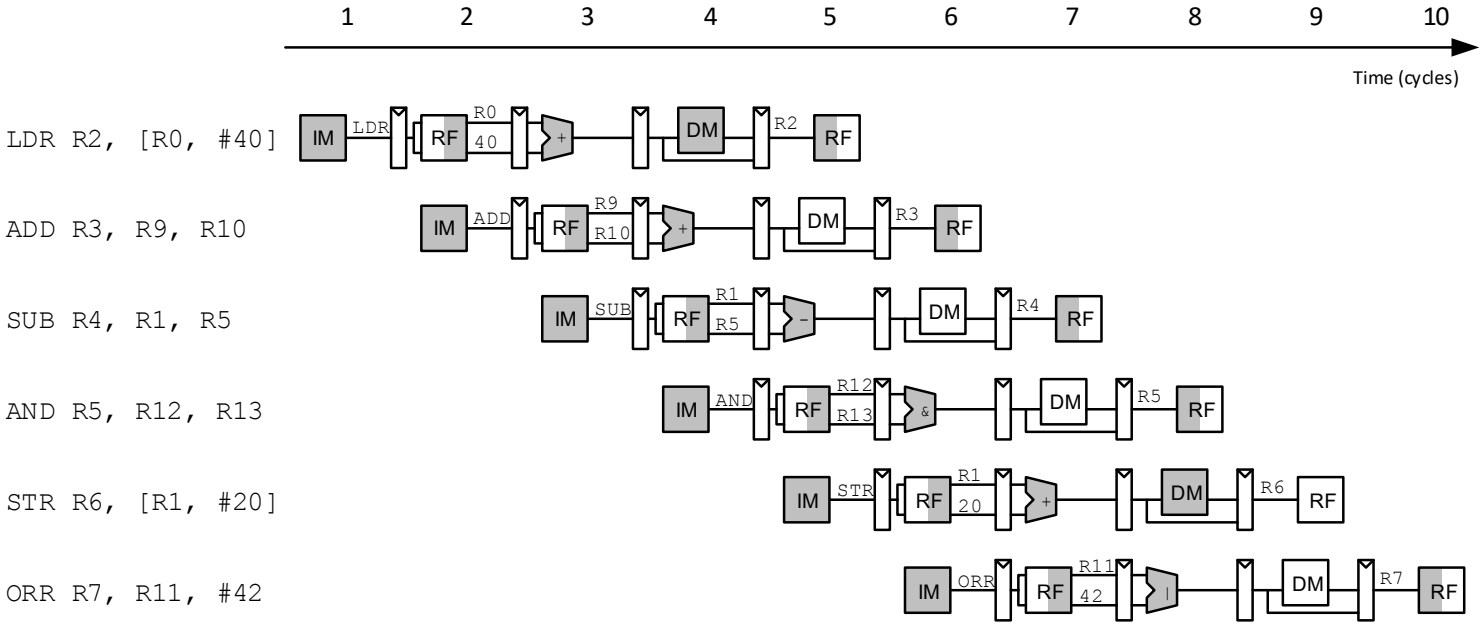
- The error is in the register file write logic that operates in the writeback stage



- 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

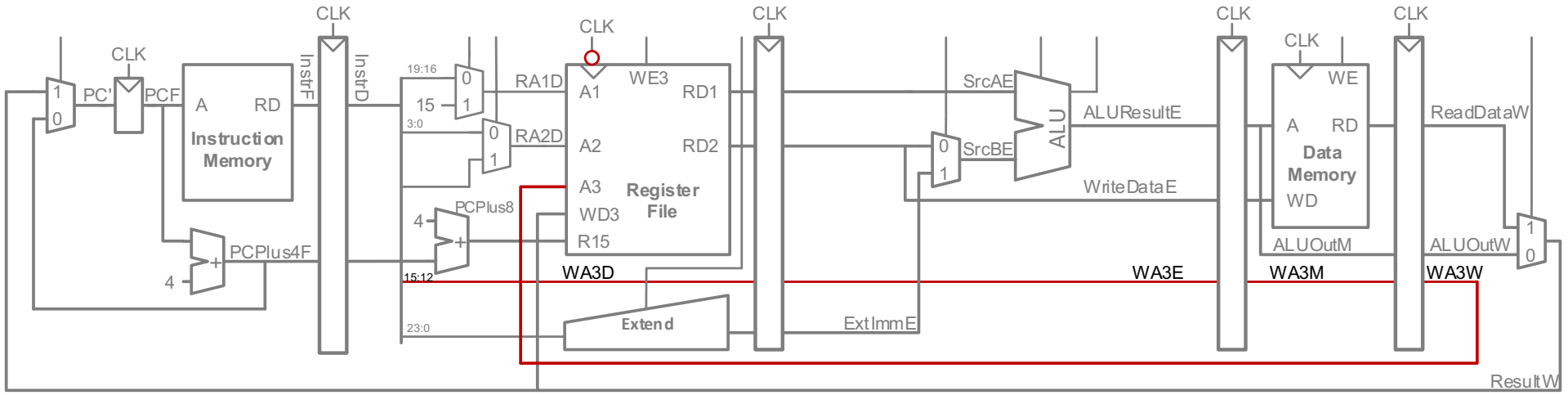
Bug in Pipelined Hardware!

- 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

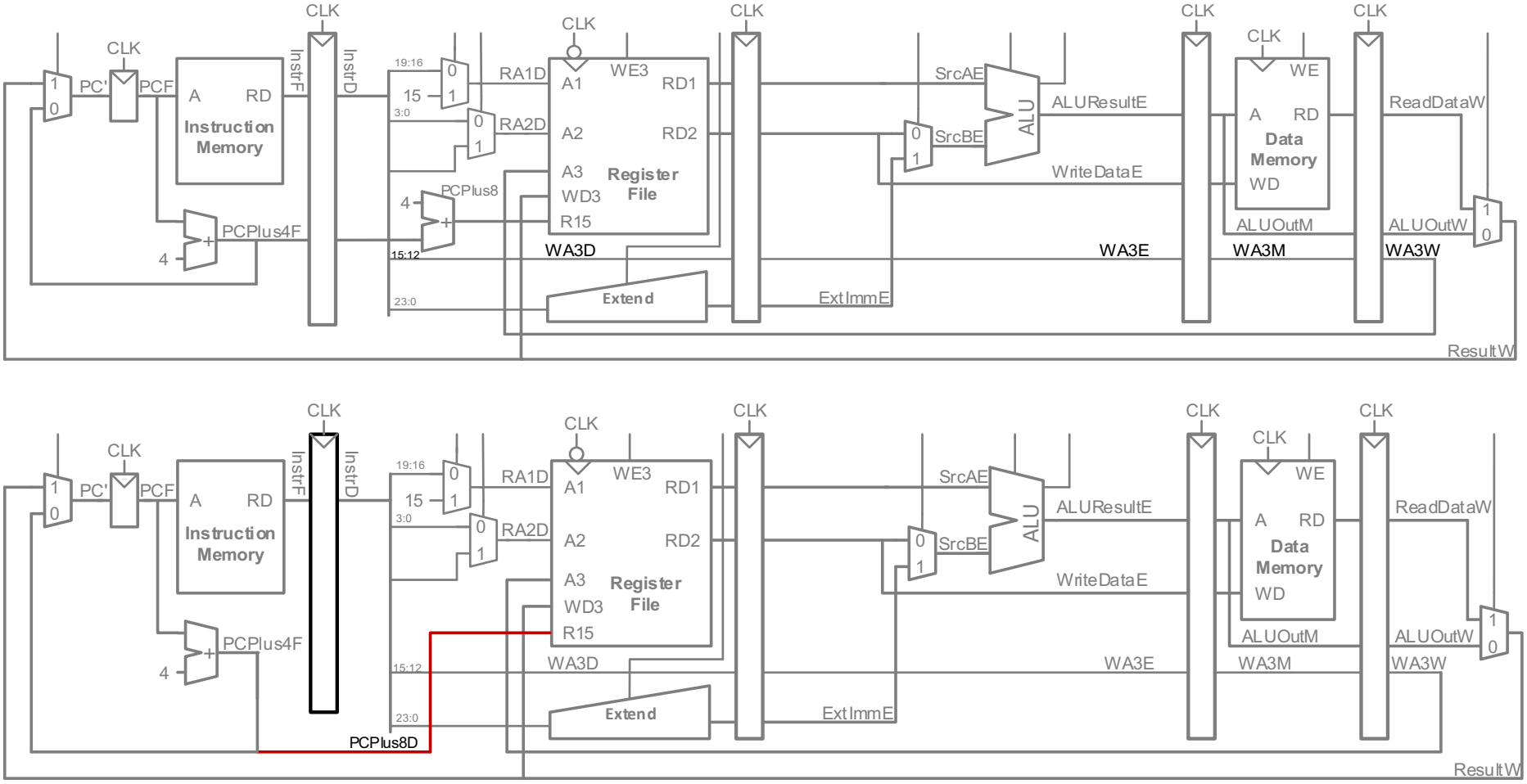


- The `WA3` signal is now pipelined along through the Execution, Memory, and Writeback stages so it remains sync with the rest of the instruction

- `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



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

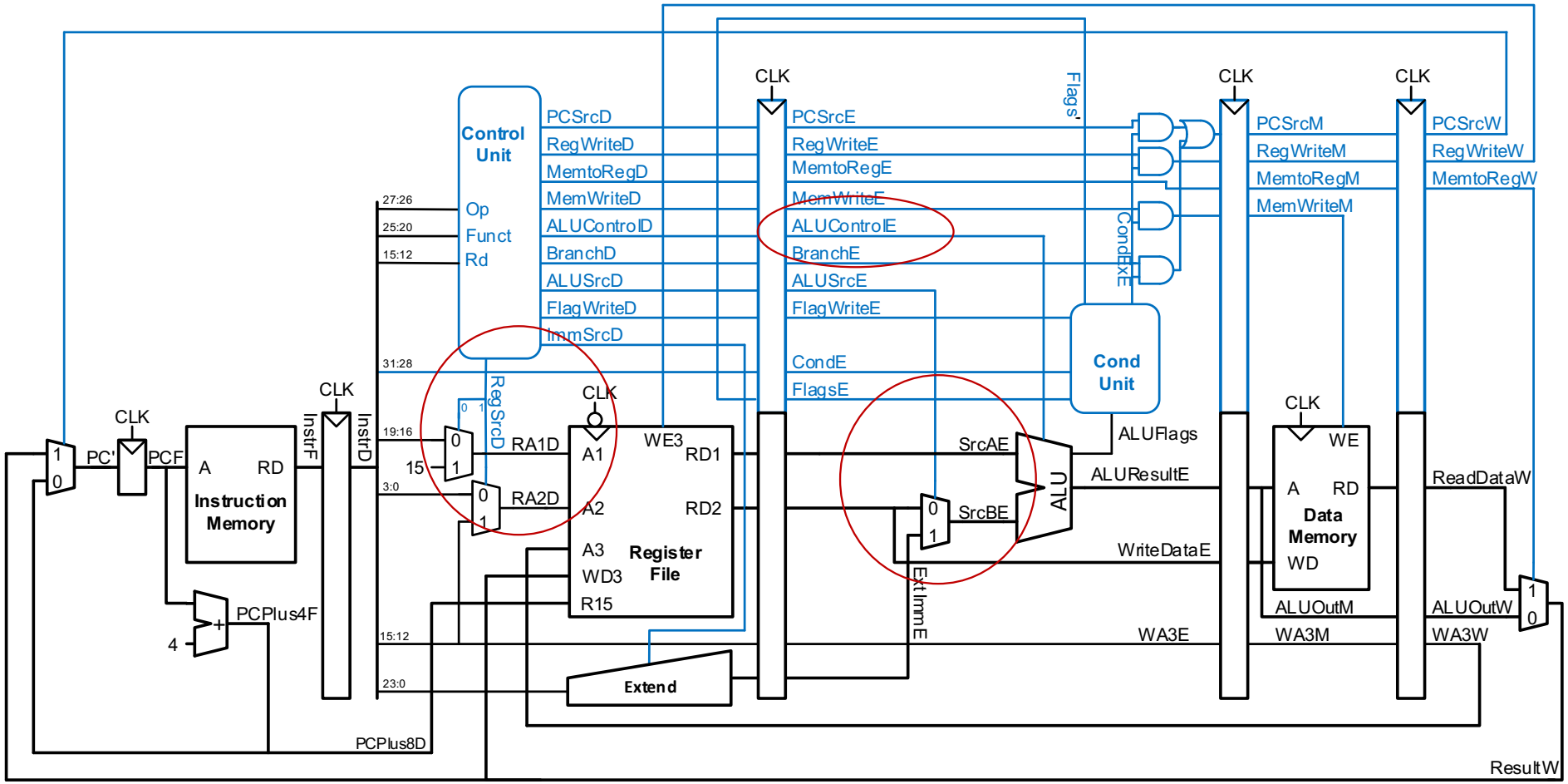
- These control signals must be pipelined along with the data

- **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



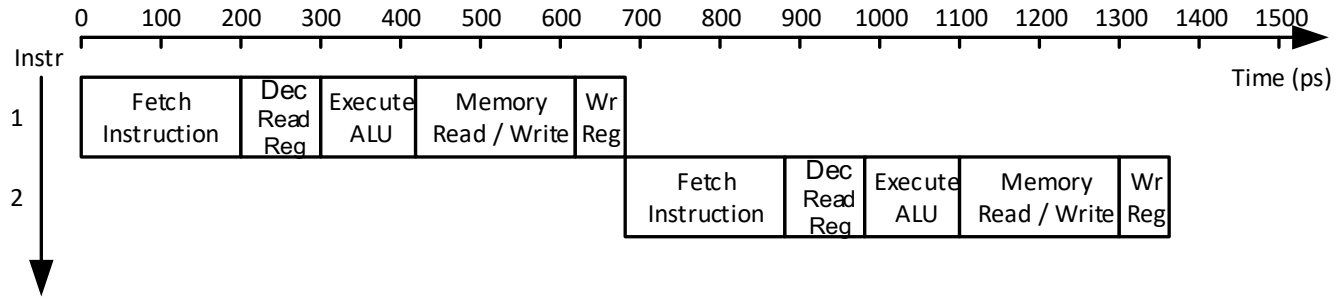
Timing Diagrams

- 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

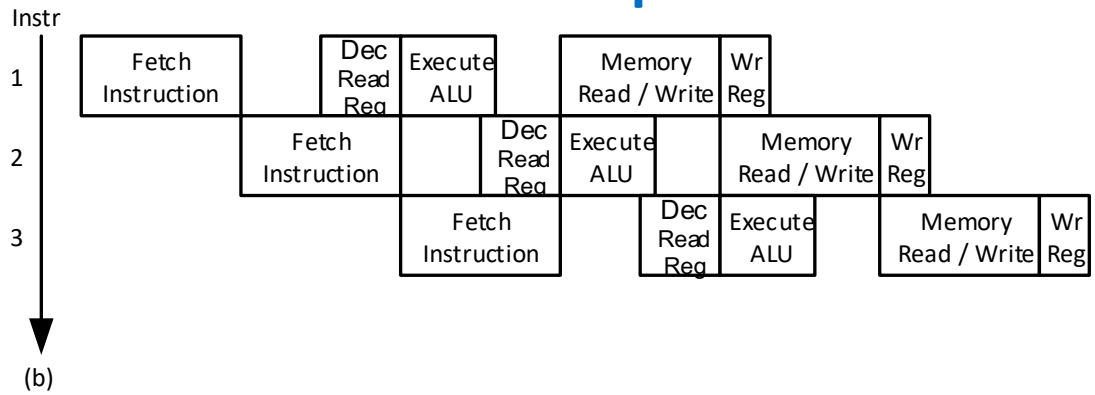
Timing Diagrams

Assumption of logic element delays from Table 7.5 of textbook

Single-Cycle



Pipelined



Performance Analysis

- In the previous slide, what is the throughout in terms of instructions per second (IPS) for single-cycle microarchitecture?
  - 1 instruction every 680 picoseconds
  - 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

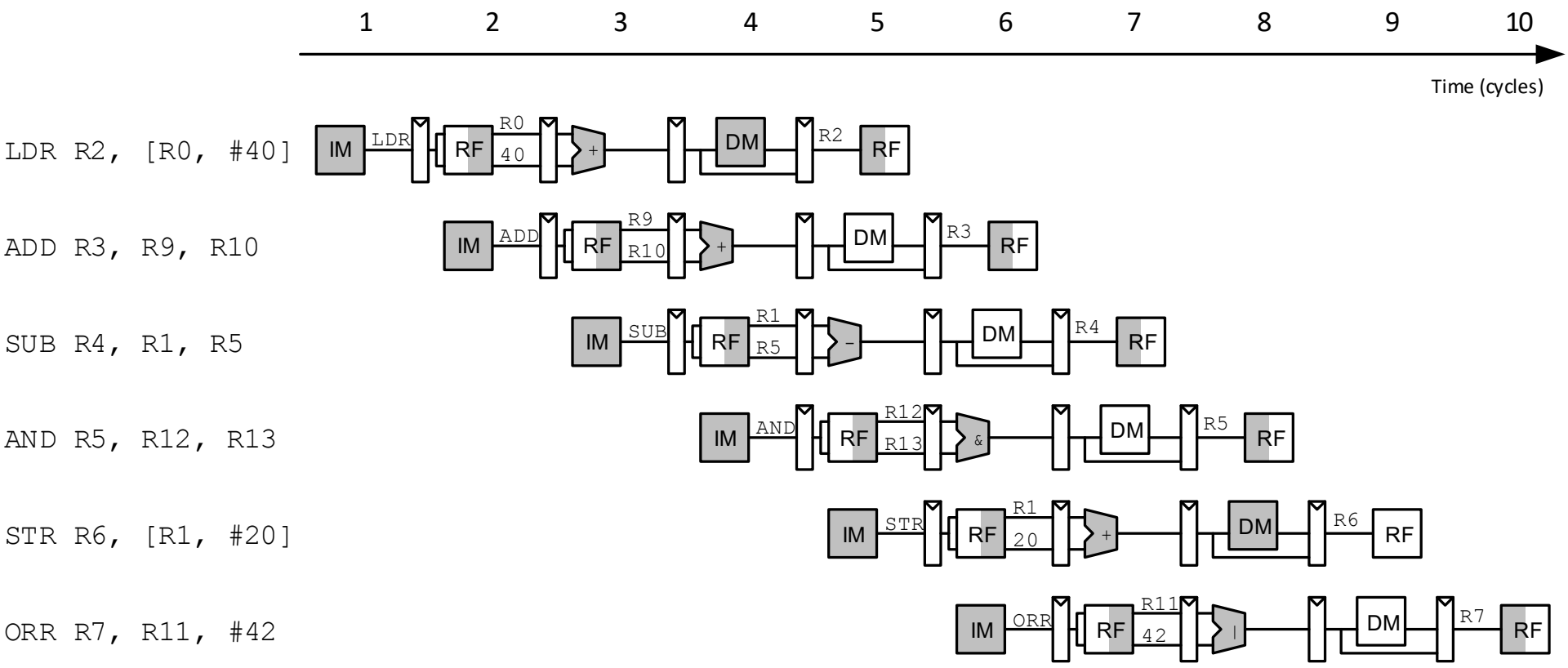
Instruction Latency with Pipelining

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

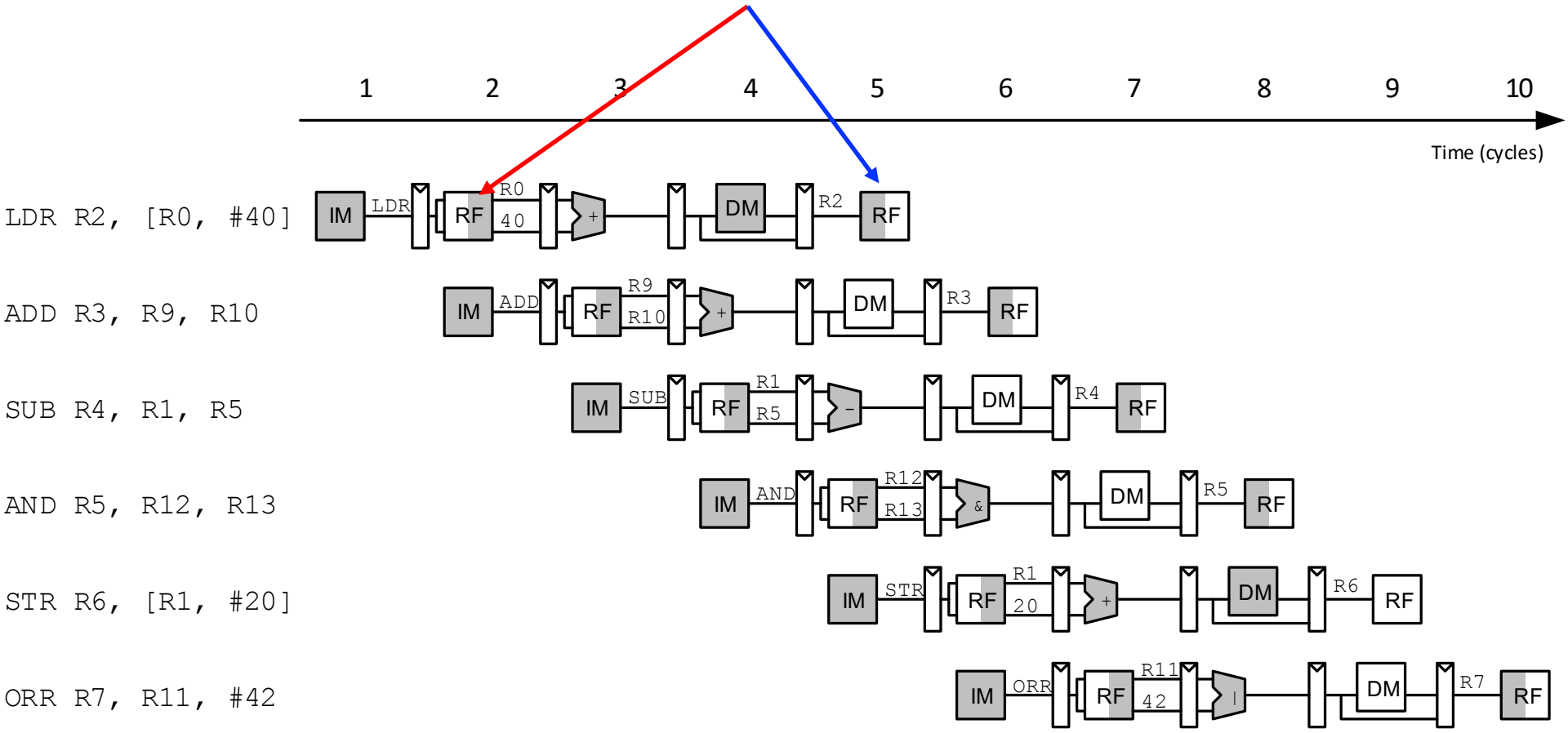
- **Latency of a single instruction increases**
  - Sequencing overhead of pipeline registers
  - Clock cycle time decided by slowest pipeline stage (internal fragmentation due to imbalanced stages)

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

Abstract Diagrams of Pipelined uArch

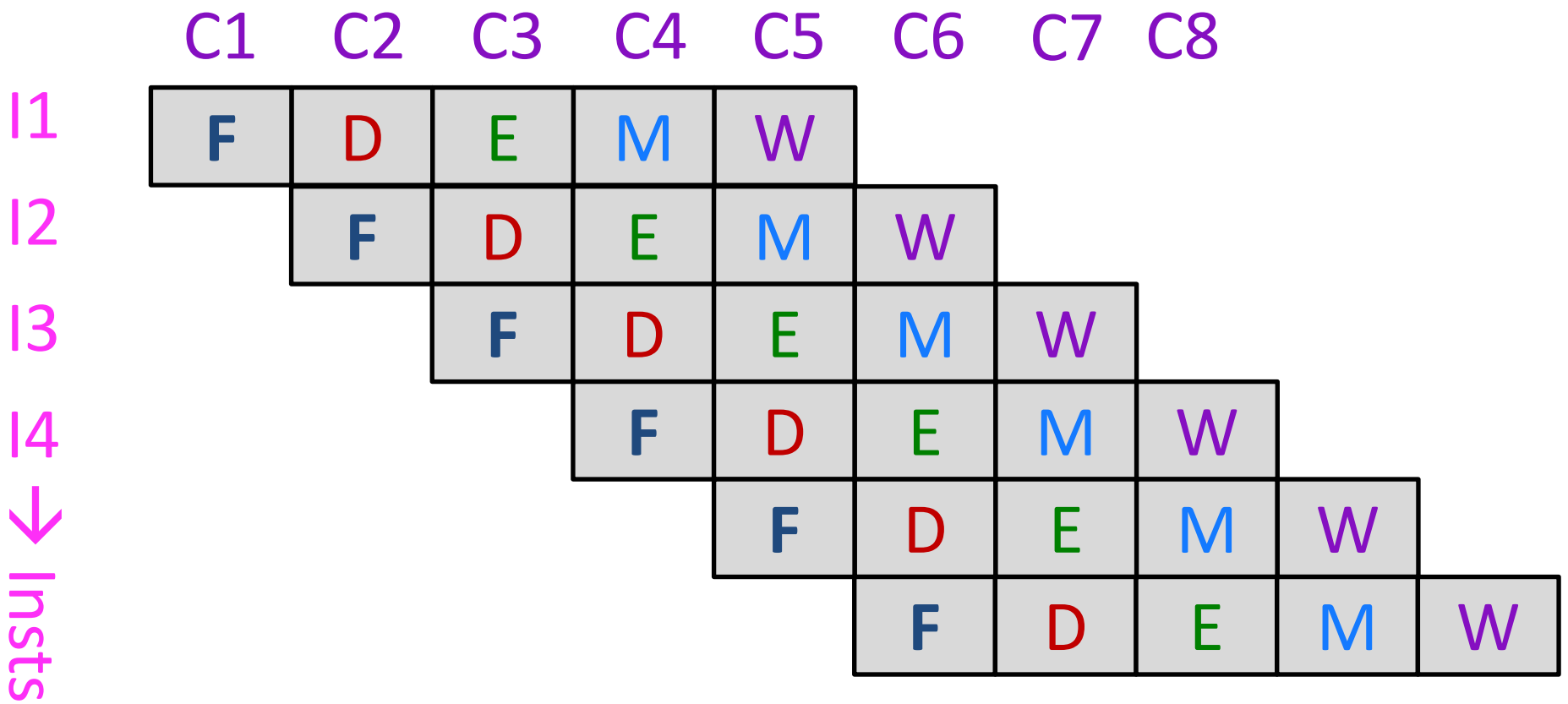


RF Read/Write in Pipelined uArch



Write in first half of clock cycle, read in the second half. In one cycle, an instruction's writeback can be visible to a younger instruction's reg read

Simplified View of Pipelining



Next: Pipeline Hazards

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

- Hazards are a part of real life

- We need to cope with hazards using extra hardware

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

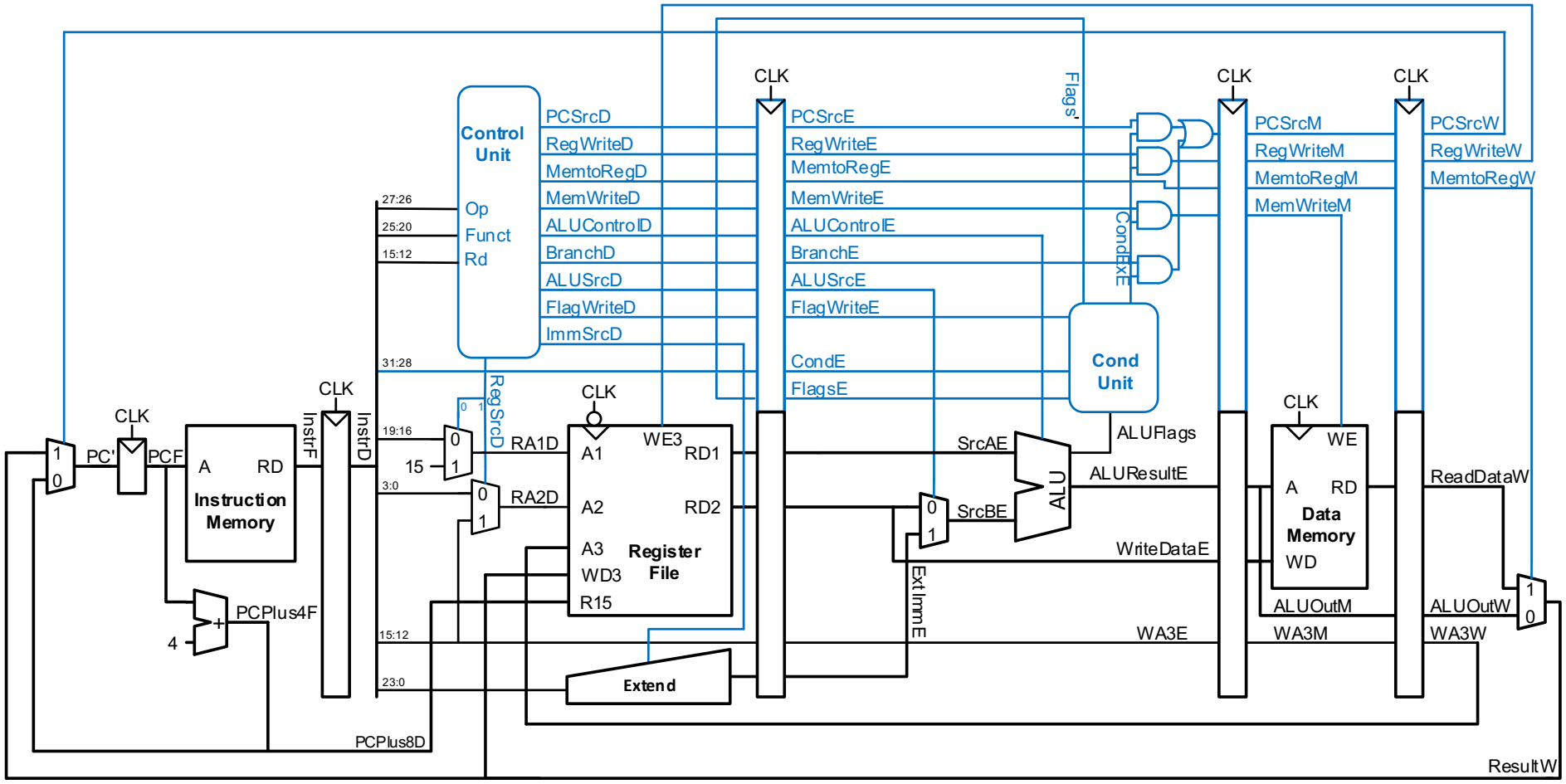
- From the processor's perspective:
  - Different solutions with different tradeoffs
    - Architectural state requires "serious" repair
    - Architectural state is **untouched**, and hazard avoided
    - Dedicated logic may be needed for hazard avoidance
    - Defensive mindset: **stall the CPU** until 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**

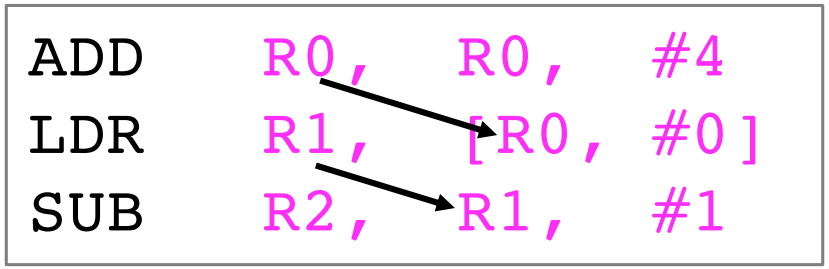
Pipeline Hazards (Another View)

- Identify backward flows (control and data)



Data Dependences

- In Von Neumann model, instructions depend on each other for data
- One type of dependence is called true dependence
- **Data (True) Dependence:** One instruction **produces** a result that the subsequent instruction **consumes**



Dependence b/w ADD & LDR
Dependence b/w LDR and SUB

```
ADD   R0,  R1,  #4
LDR   R2,  [R3, #0]
ADD   R4,  R5,  #1
```

NO dependences b/w instructions

- **Instruction chains with dependences need special care in pipelined uarch**

Read-After-Write Hazards

- True dependences lead to read-after-write hazards

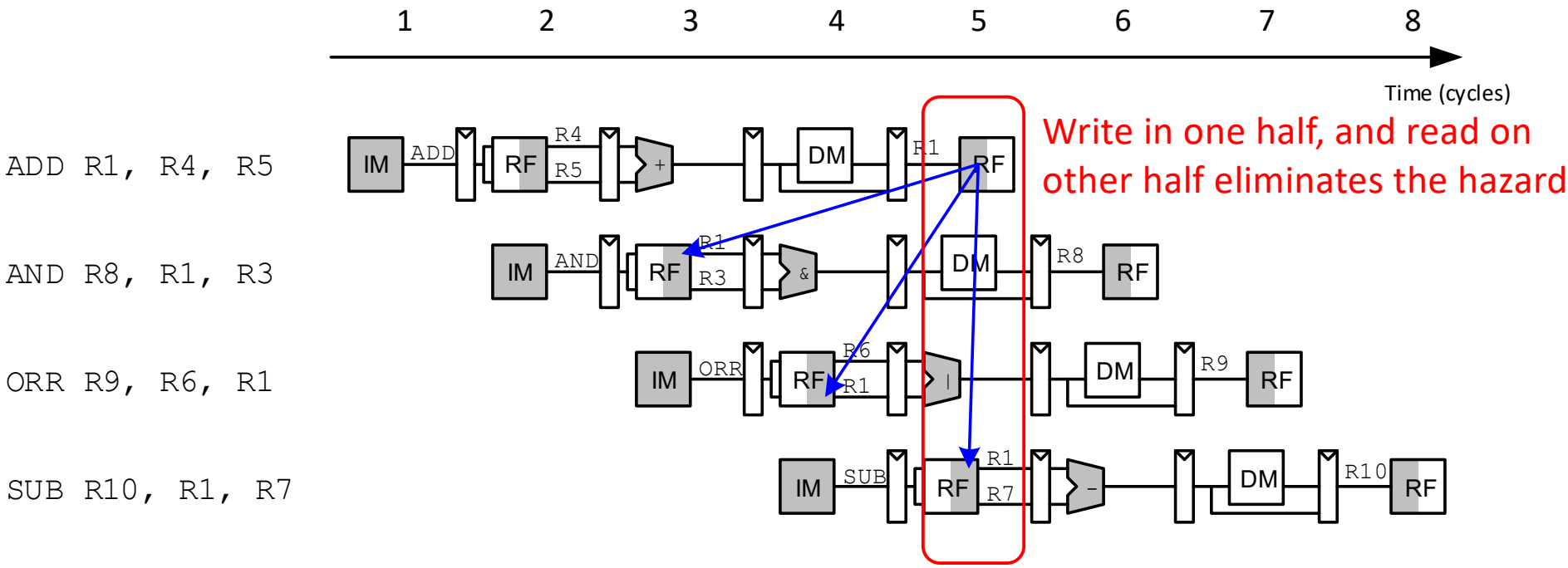
- These hazards are not possible in a single-cycle microarchitecture

- **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



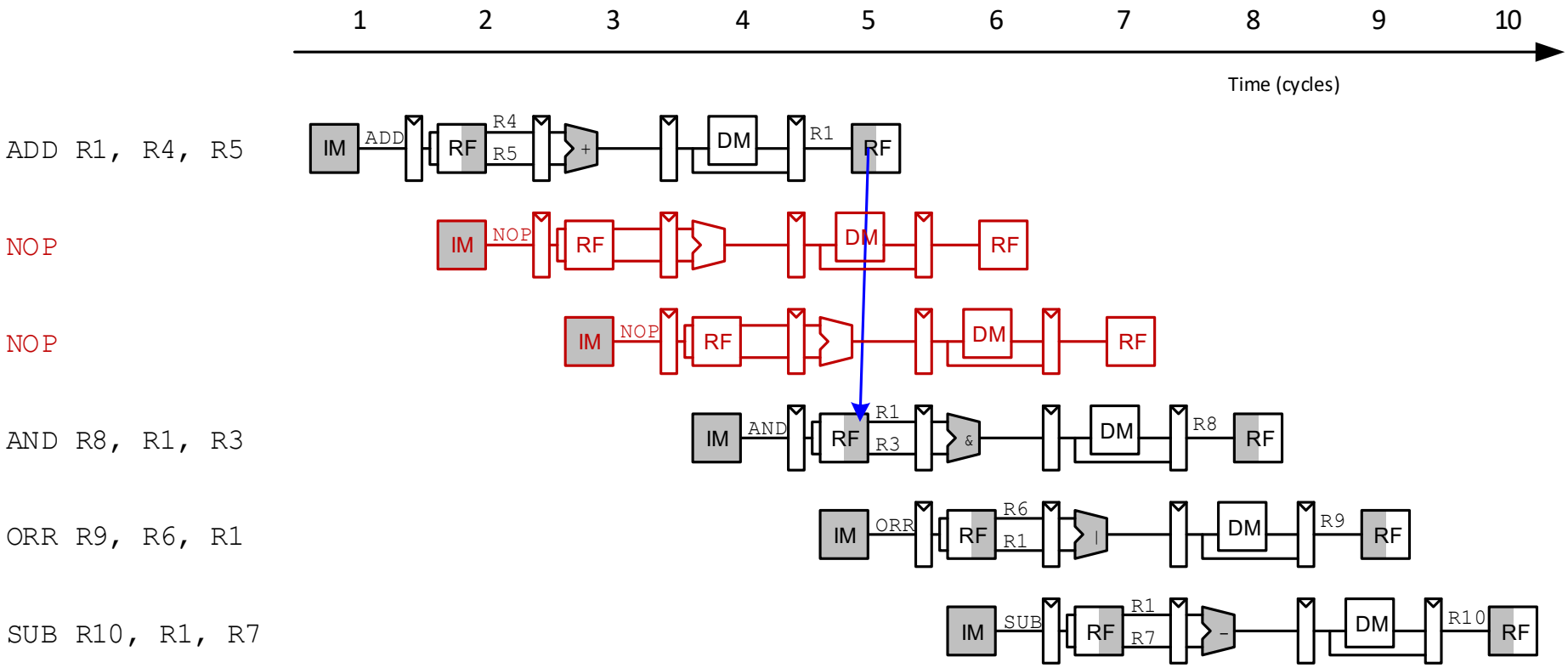
- Use a clever register read/write policy to eliminate one hazard
  - What can we do about the remaining two hazards?

Solution # 1: Software Interlocking

- Insert `NOPS` in code at **compile time**
  - NOP is an instruction that does nothing
  - **Idea:** Insert enough `NOPS` for results to be ready

- OR better, move independent useful instructions forward at **compile time**

Example: Software Interlocking



Solution # 1: Software Interlocking

- **Drawbacks of software interlocking**
  - Programming is complicated
  - Speed is degraded

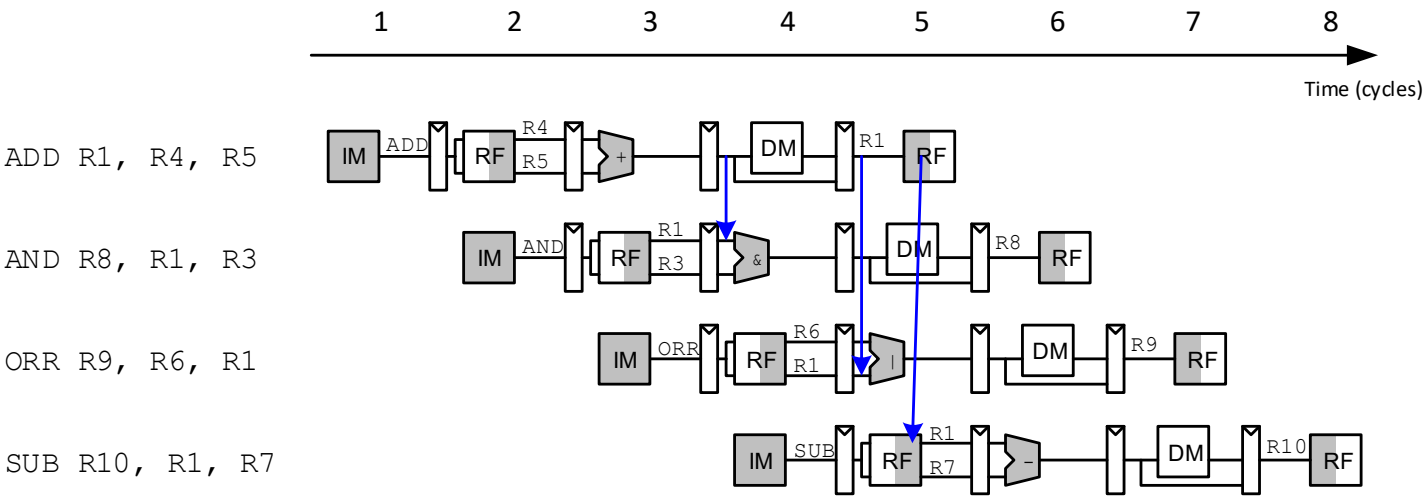
Solution # 2: Forwarding or Bypassing

- **Hardware solution:** Data hazards can be solved by **forwarding** or **bypassing** (except some special scenarios)

- Extra hardware to send result from the Memory or Writeback stage to a **"dependent"** instruction in Execute stage
  - **Key:** 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
  - (1) RF, (2) Memory PPR, (3) Writeback PPR

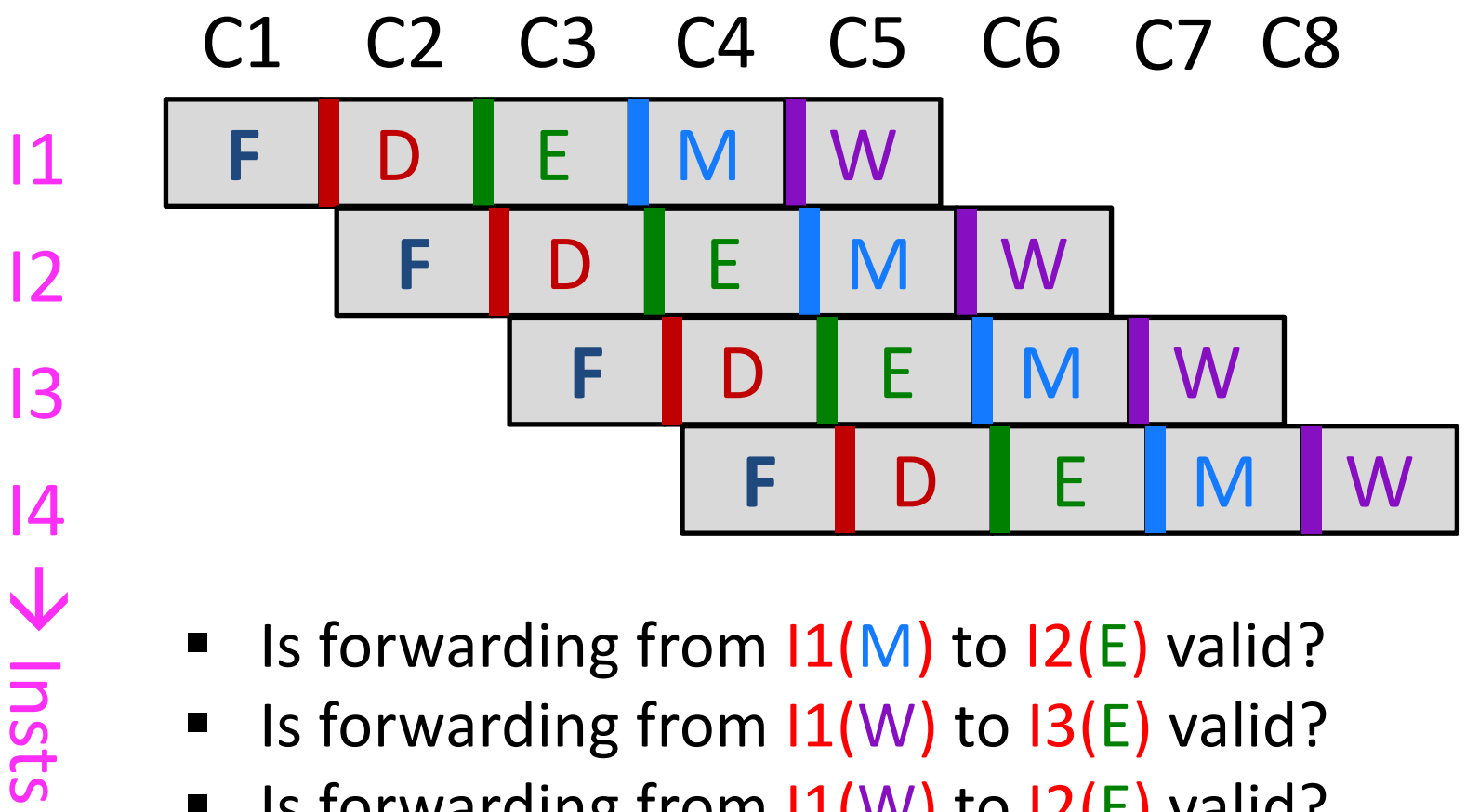
Why Forwarding Works?



- Sum from the `ADD` instruction is computed by ALU in **cycle 3** and is needed by the `AND` instruction in **cycle 4**

- No need to wait for the results to appear in register file

Forwarding Exercise



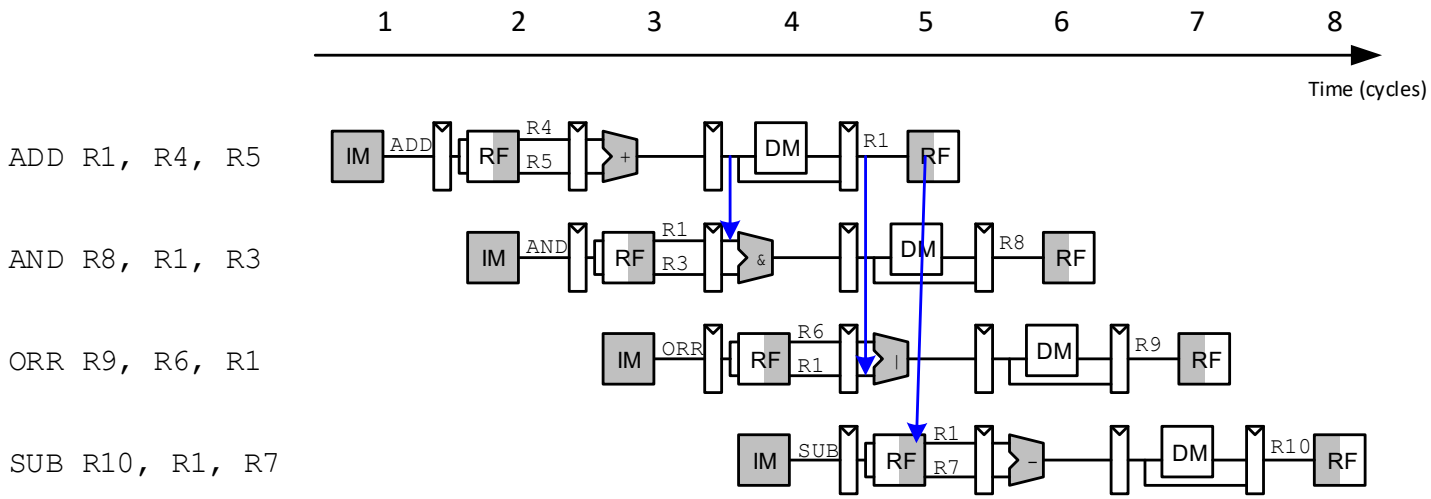
PPR Code

D
E
M
W

- Is forwarding from I1(M) to I2(E) valid?
- Is forwarding from I1(W) to I3(E) valid?
- Is forwarding from I1(W) to I2(E) valid?

Forwarding Example

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



- When is forwarding necessary?
  - Check if source register read in EX stage **matches** destination 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)

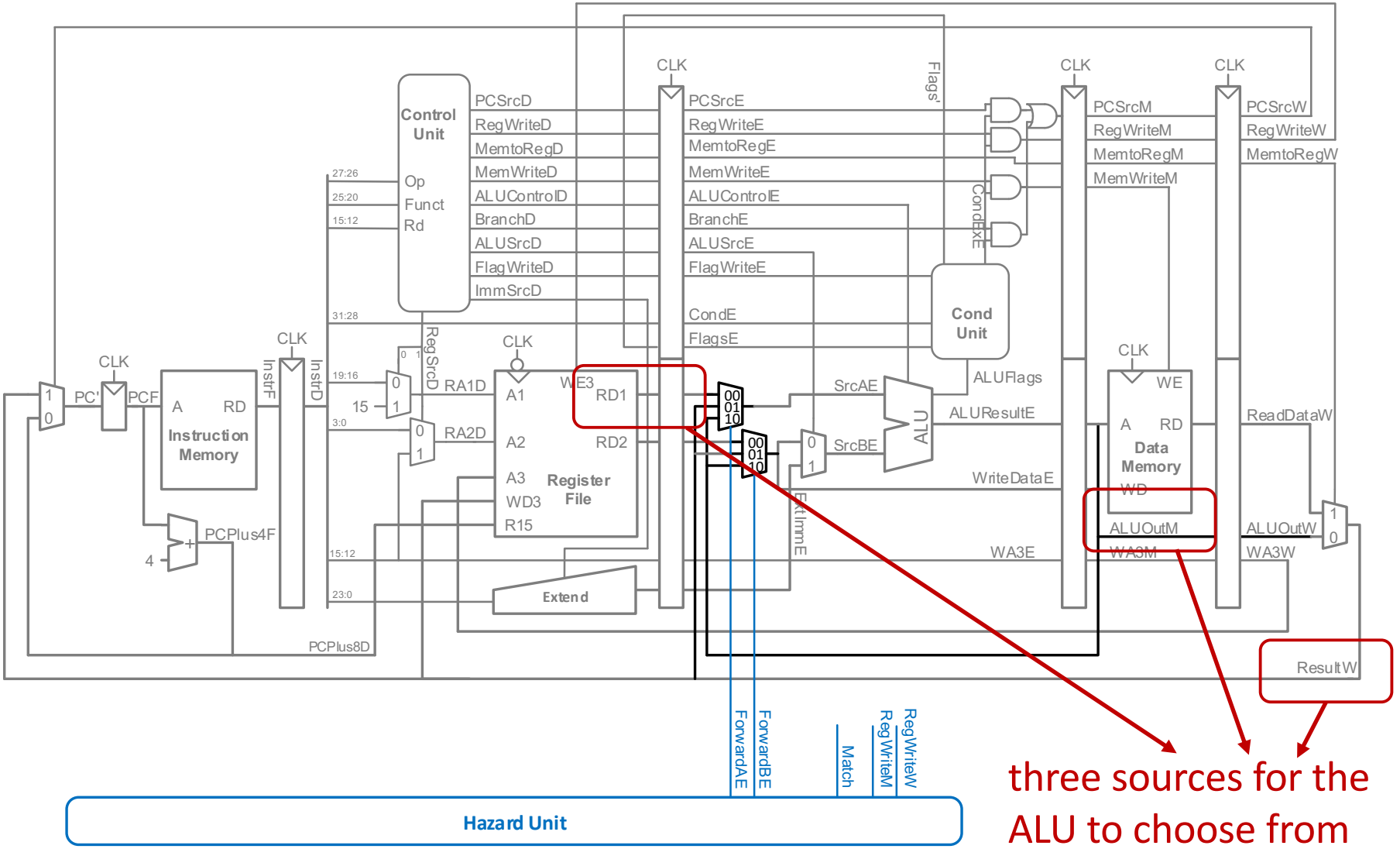
- **Execute** stage register matches **Writeback** stage register?
  Match_1E_W = (RA1E == WA3W)
  Match_2E_W = (RA2E == WA3W)

- 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

Load-Use Hazard

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

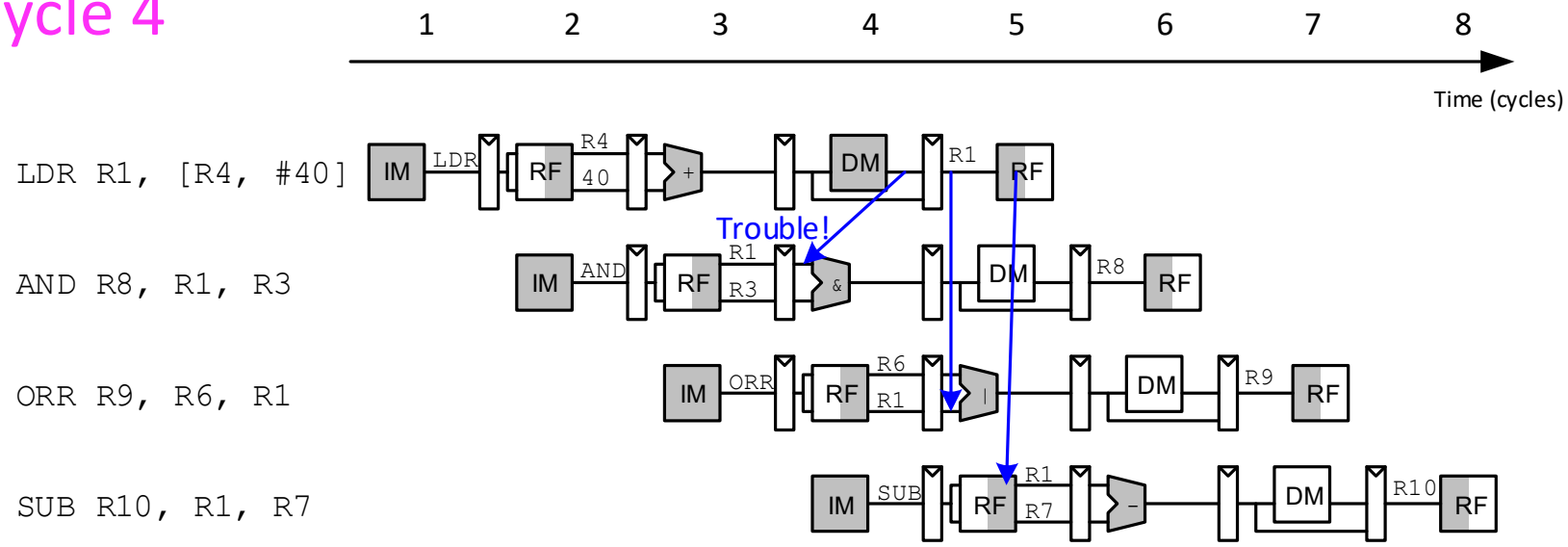
- `LDR` does not finish reading data until the end of the **MEM stage**
  - The result cannot be forwarded to the **EX-stage** of the next instruction
  - We call Load followed by its use a Load-Use hazard

- 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

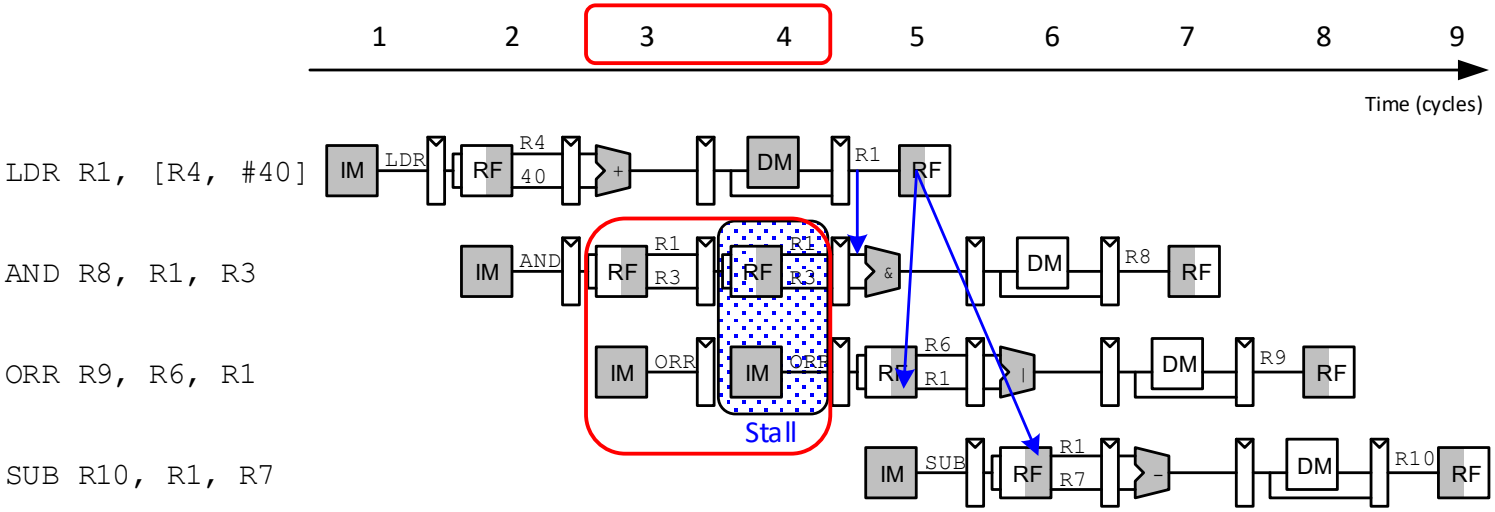
- The dependent instruction can be detected as the "user" of `LDR` after it has been decoded at the end of **Decode stage**

- **Idea:** Stall the dependent instruction in the **Decode stage for one cycle** (until LDR completes the memory read)

- 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

What does a stall look like?

- Stalling stage **X** does three things
  - Stalls stage X **(obviously)**
  - Stalls stage X – 1
  - Sends a bubble in stage X + 1

Let’s Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

Cycle #	Fetch	Decode	Execute	Memory	Writeback
1:					
2:					
3:					
4:					
5:					
6:					
7:					

Let’s Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

Cycle #	Fetch	Decode	Execute	Memory	Writeback
1:	i1				
2:					
3:					
4:					
5:					
6:					
7:					

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

Cycle #	Fetch	Decode	Execute	Memory	Writeback
1:	i1				
2:	i2	i1			
3:					
4:					
5:					
6:					
7:					

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i1 – i2**

Cycle #	Fetch	Decode	Execute	Memory	Writeback
1:	i1				
2:	i2	i1			
3:	i3	i2	i1		
4:					
5:					
6:					
7:					

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

	Cycle #	Fetch	Decode	Execute	Memory	Writeback
	1:	i1				
	2:	i2	i1			
	3:	i3	i2	i1		
(stall)	**4:**	i3	i2	00000000	i1	
	5:					
	6:					
	7:					

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

	Cycle #	Fetch	Decode	Execute	Memory	Writeback
	1:	i1				
	2:	i2	i1			
	3:	i3	i2	i1		
(stall)	**4:**	i3	i2	00000000	i1	
	5:	i4	i3	i2	00000000	i1
	6:					
	7:					

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

	Cycle #	Fetch	Decode	Execute	Memory	Writeback
	1:	i1				
	2:	i2	i1			
	3:	i3	i2	i1		
(stall)	**4:**	i3	i2	00000000	i1	
	5:	i4	i3	i2	00000000	i1
	6:	i5	i4	i3	i2	00000000
	7:					

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

	Cycle #	Fetch	Decode	Execute	Memory	Writeback
	1:	i1				
	2:	i2	i1			
	3:	i3	i2	i1		
(stall)	**4:**	i3	i2	00000000	i1	
	5:	i4	i3	i2	00000000	i1
	6:	i5	i4	i3	i2	00000000
	7:		i5	i4	i3	i2

Let's Visualize Stall in Decode stage

- i1 – i5 are five instructions. Load-use hazard between **i2 – i3**

	Cycle #	Fetch	Decode	Execute	Memory	Writeback
	1:	i1				
	2:	i2	i1			
	3:	i3	i2	i1		
(stall)	**4:**	i3	i2	000000 	i1	
	5:	i4	i3	i2	000000 	i1
	6:	i5	i4	i3	i2	000000 
	7:		i5	i4	i3	i2

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

Implementing Stalls

- **Stalling** a stage requires disabling the pipeline register, so that the contents do not change
  - 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 `LDR` in the **Execute stage** AND *Match_12D_E* is TRUE?

  ldrstall = *Match_12D_E* AND *MemtoRegE*

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

Pipelined CPU with Stalls to Solve Load-Use Hazard

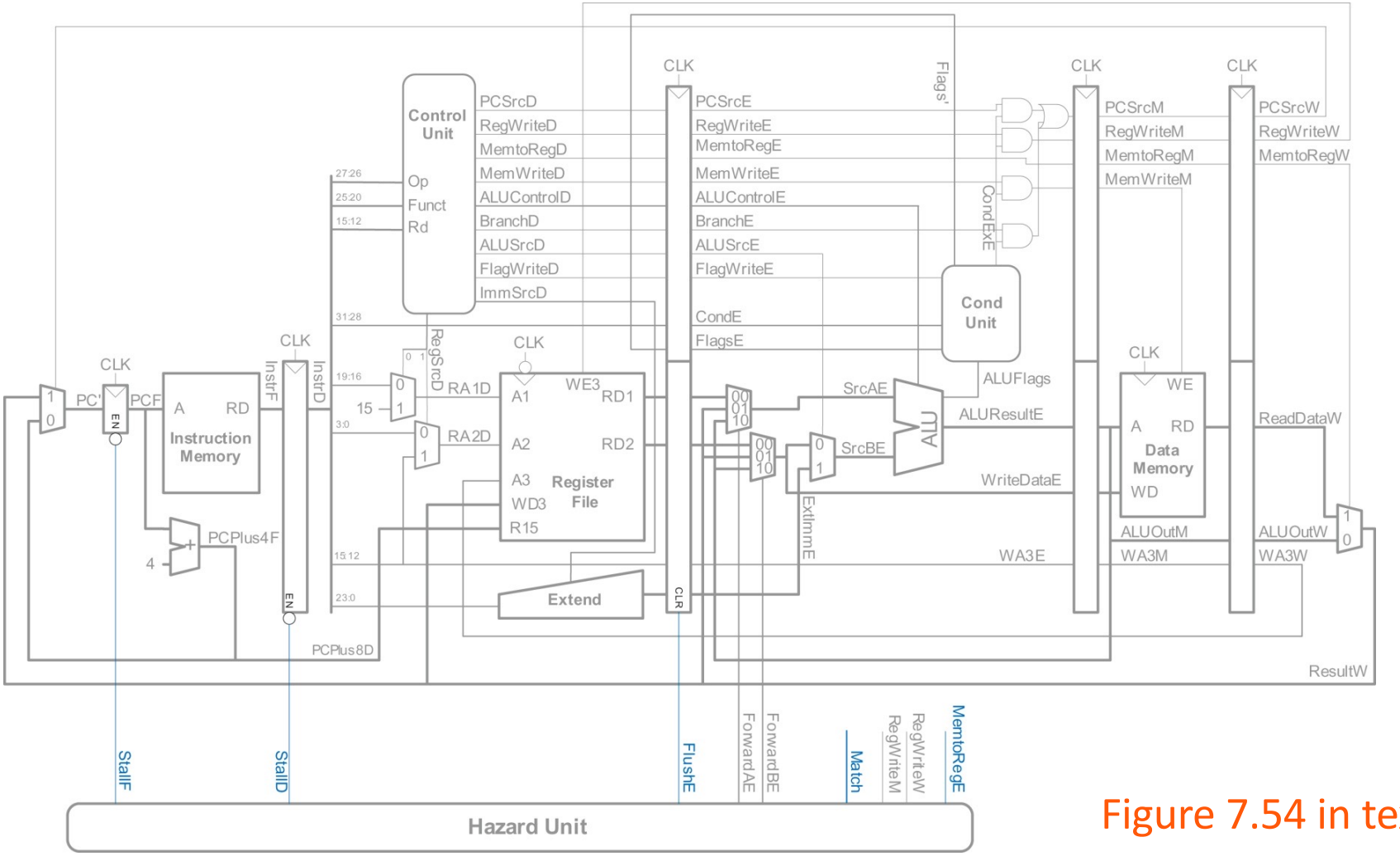


Figure 7.54 in textbook

Control Hazards

- Control hazards are due to changes in 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** when the instruction is fetched
  - But the PC register is incremented in the Fetch stage

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
  - Stall for **4** cycles – a very high penalty to pay for every branch instruction

- 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 (aka., recovery)
  - Branch misprediction penalty depends on when recovery is initiated

Simplest Branch Predictor

- Predict-always-untaken
  - Keep fetching the next sequential instructions

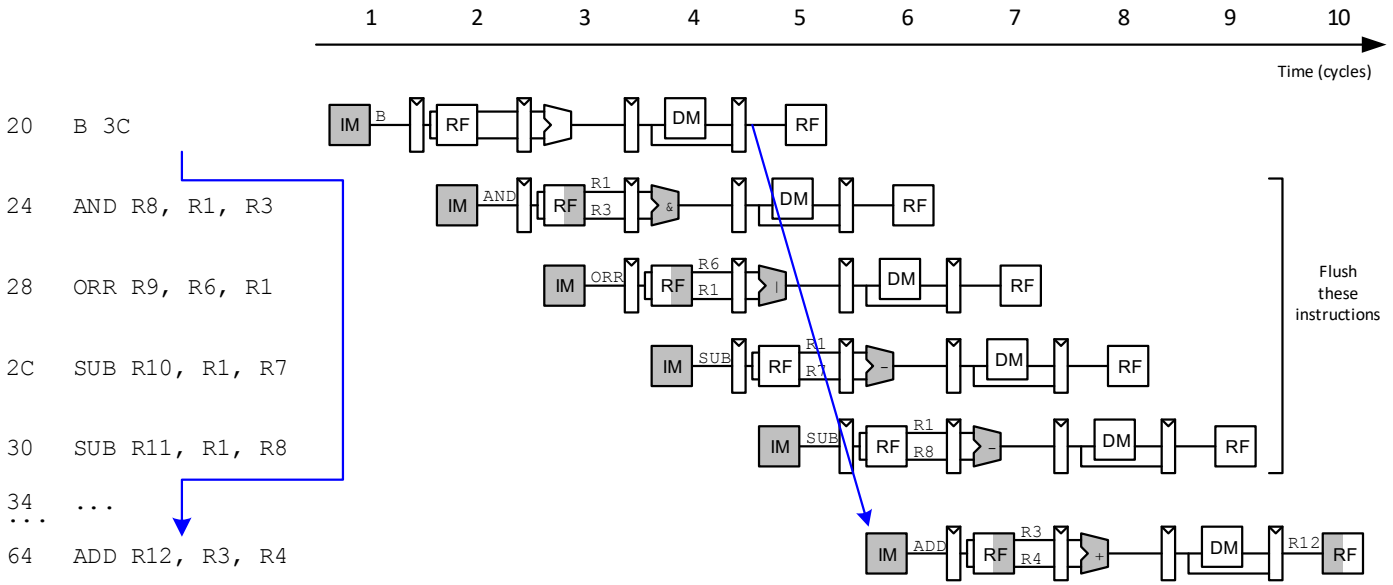
- Predict-always-taken
  - CPU **stalls** for four cycles because target address not available

- Both predictors above use a **static prediction policy**

- **Dynamic branch prediction**
  - Different predictions for different executions of same branch
  - Takes recent branch behavior into account

Predict-always-untaken: **Branch is Taken**

- predict-always-untaken seems reasonable if target is not known



- BUT, four instructions are flushed when branch is taken
- Misprediction penalty of 4 wasted cycles for taken branches
- **Idea: Predict the branch early**

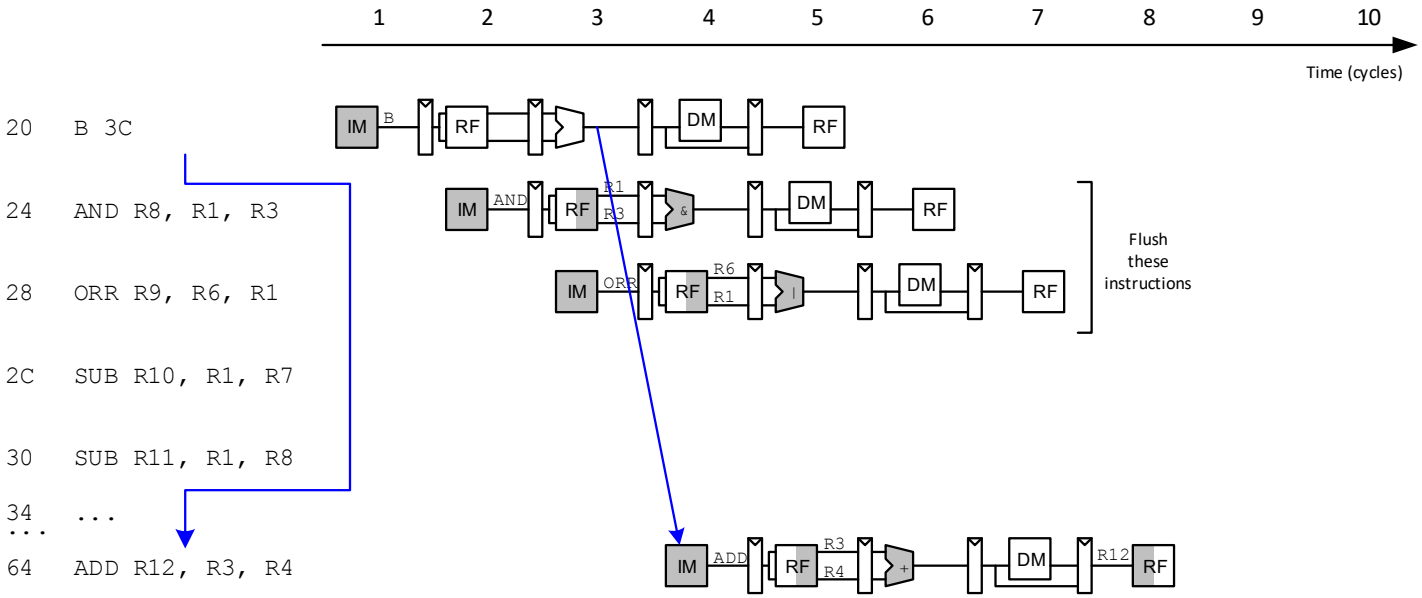
Static Branch Prediction: All Scenarios

- **Predict-always-untaken (Keep the pipeline busy)**
  - If prediction is correct, nothing to do
  - If prediction is incorrect, flush 4 instructions and repair the architectural state (i.e., update PC with correct target)

- **Predict-always-taken (Ok to waste slots in the pipeline)**
  - If prediction is correct, branch to the target inst., no harm
  - If prediction is incorrect, use incremented PC of next instruction (4 cycles are wasted)

Alternative: Early Branch Resolution

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



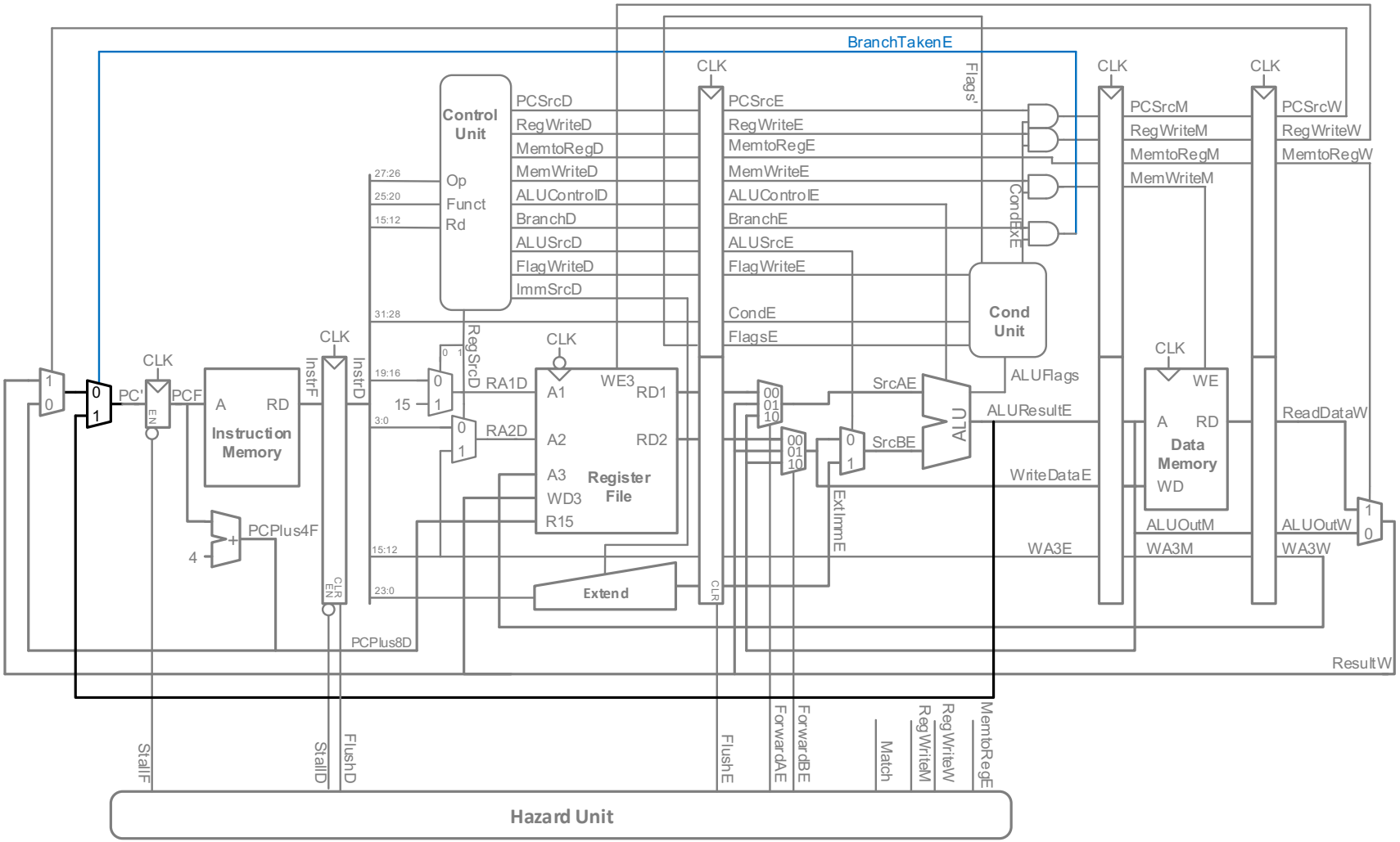
- Flush the two instructions in the **Fetch** and **Decode** stages

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)

Pipelined Processor Early Resolution



Flush Logic with Early Branch Resolution

- **Flush Decode** if branch is taken

  FlushD = BranchTakenE

- **Flush Execute** if branch is taken

  FlushE = BranchTakenE

Stall + Flush Logic with Early Branch Resolution + Load-Use Hazard

- **Stall Fetch** if *load-use hazard is discovered*
  StallF = ldrStallD
- **Flush Decode** if branch is taken
  FlushD = BranchTakenE
- **Flush Execute** if branch is taken
  FlushE = ldrStallD + BranchTakenE
- **Stall Decode** if *load-use hazard is discovered*
  StallD = ldrStallD

Optional: Writes to PC

- Writes to PC still stall the CPU for 4 cycles (contrast with **B** instruction)

- **Stall Fetch** if *PC write is discovered in Decode, Execute, or Memory*

  StallF = *PCSrcD* + *PCSrcE* + *PCSrcM*

- **Flush Decode** if *PC write is discovered in Decode, Execute, Memory, or Writeback*

  FlushD = PCSrcD + PCSrcE + PCSrcM + PCSrcW

Flush and Stall Logic for Writes to PC

- Explaining the logic for StallF control signal
  - Cycle #1: PC-changing instruction (**I**) is fetched
  - Cycle #2: **I** is decoded and PCSrcD is asserted
  - Cycle #3: **I** is executed and PCSrcE is asserted
  - Cycle #4: **I** is in M stage and PCSrcM is asserted
  - Cycle #5: PCSrcW is asserted, and new PC is written to the ResultW bus
- PC is a register so will be updated in the next clock cycle (cycle # 6)
- In cycle #5, StallF is asserted, so that the next cycle the PC register is set up properly to capture the new value of instruction address (ResultW)
- In the first four cycles, StallF is deasserted to not cause a change to PC

Flush and Stall Logic for Writes to PC

- Explaining the logic for FlushD control signal
  - Cycle #1: PC-changing instruction (**I**) is fetched
  - Cycle #2: **I** is decoded and PCSrcD is asserted
  - Cycle #3: **I** is executed and PCSrcE is asserted
  - Cycle #4: **I** is in M stage and PCSrcM is asserted
  - Cycle #5: PCSrcW is asserted, and new PC is written to the ResultW bus
- If we keep FlushD asserted during cycle 5, then at the beginning of cycle # 6 when rising edge arrives, register will still read all zeroes
- In cycle # 6, FlushD is released so in cycle # 7, when the correct instruction advances to the Decode register, the instruction is captured at the edge of the clock (in cycle # 7)

Full Control Stalling Logic (page # 440)

- ***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

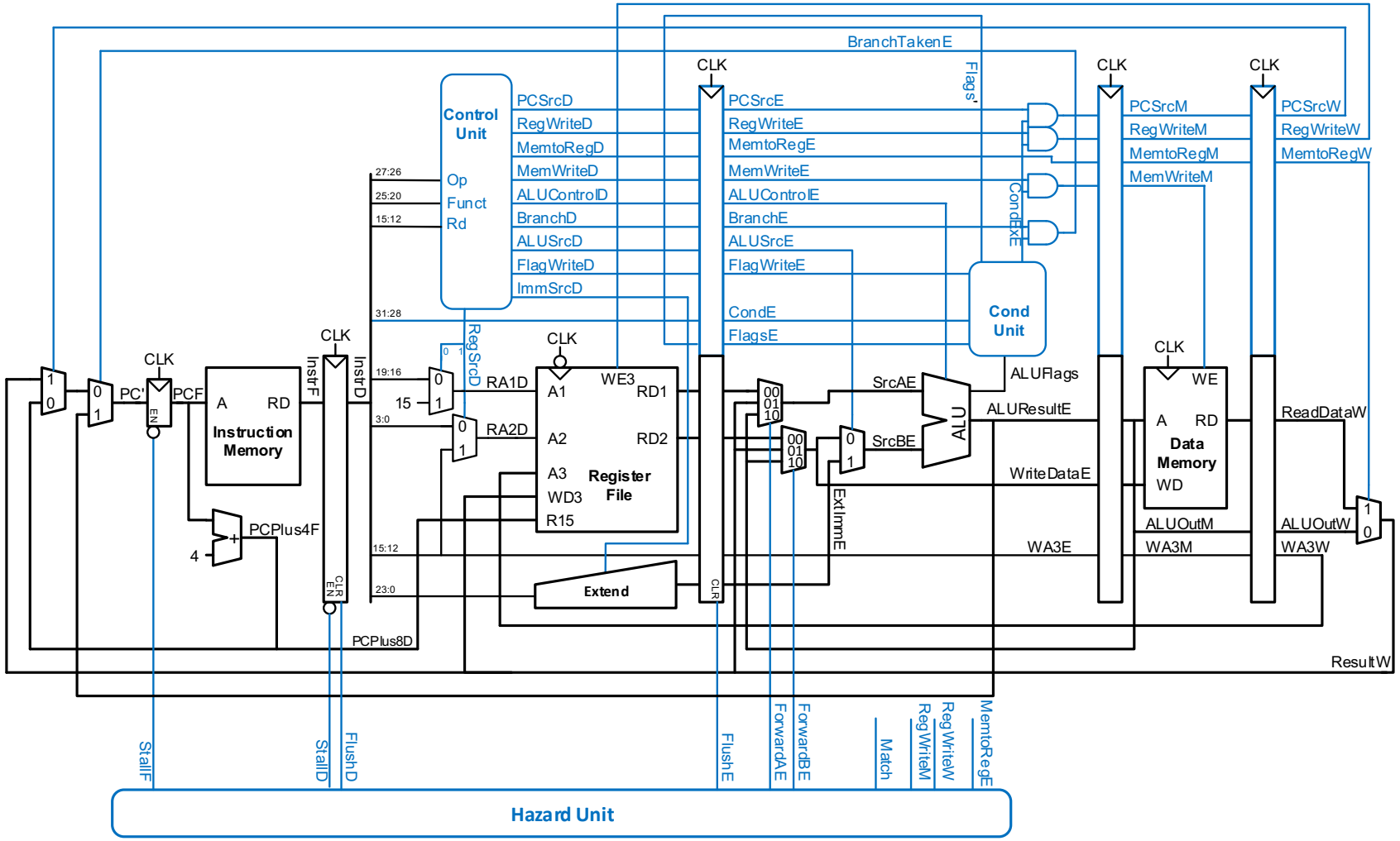
  Flush D if PC write in progress in D, E, M, or W, or branch taken in E

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

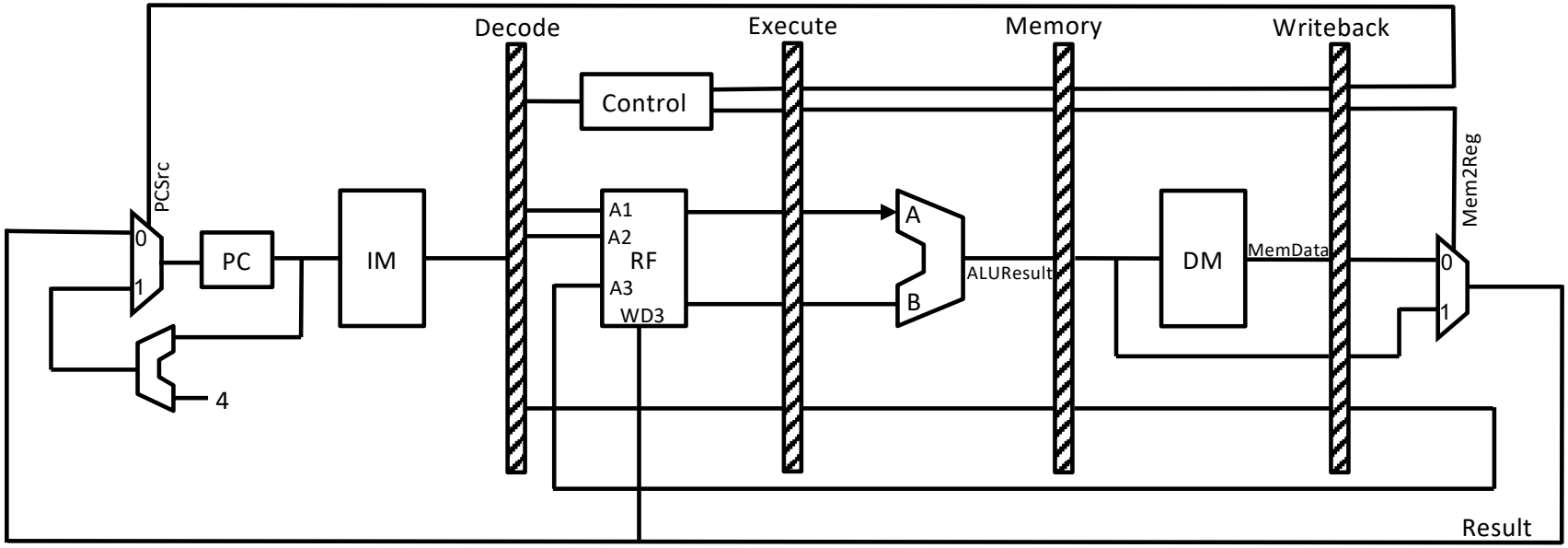
  StallD = ldrStallD

  Stall Decode if LDR-Use hazard

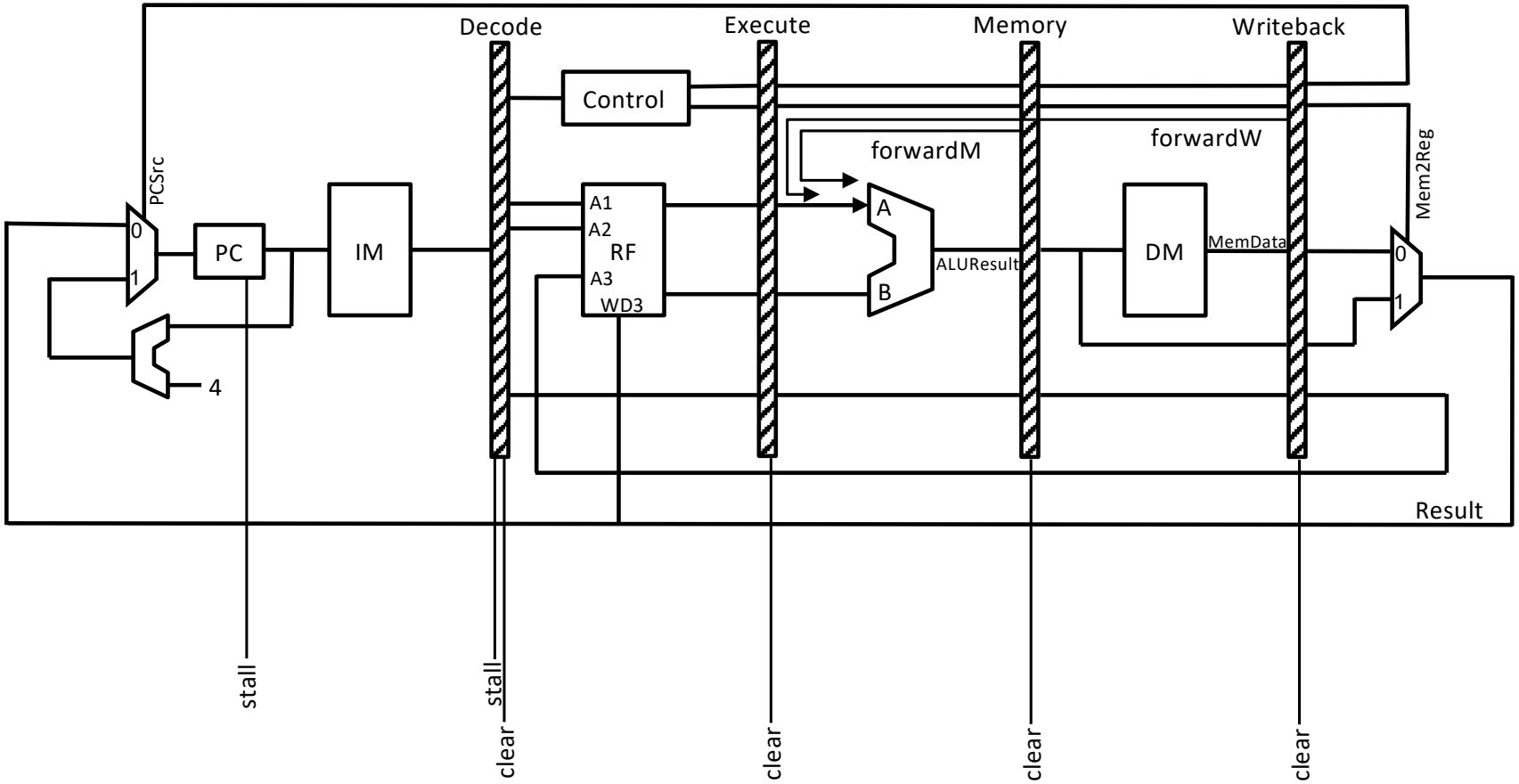
ARM Processor with Full Hazard Handling



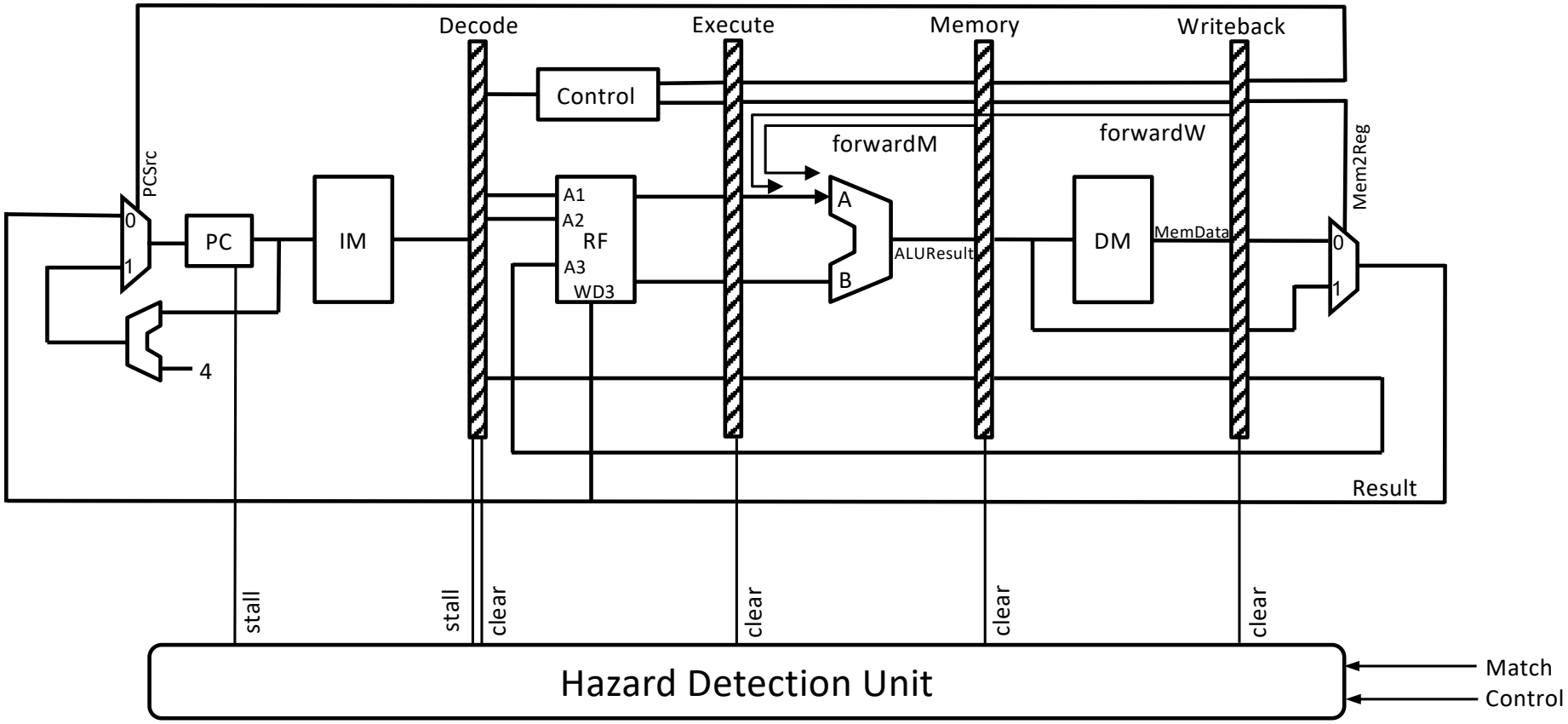
Simplified View of Pipeline



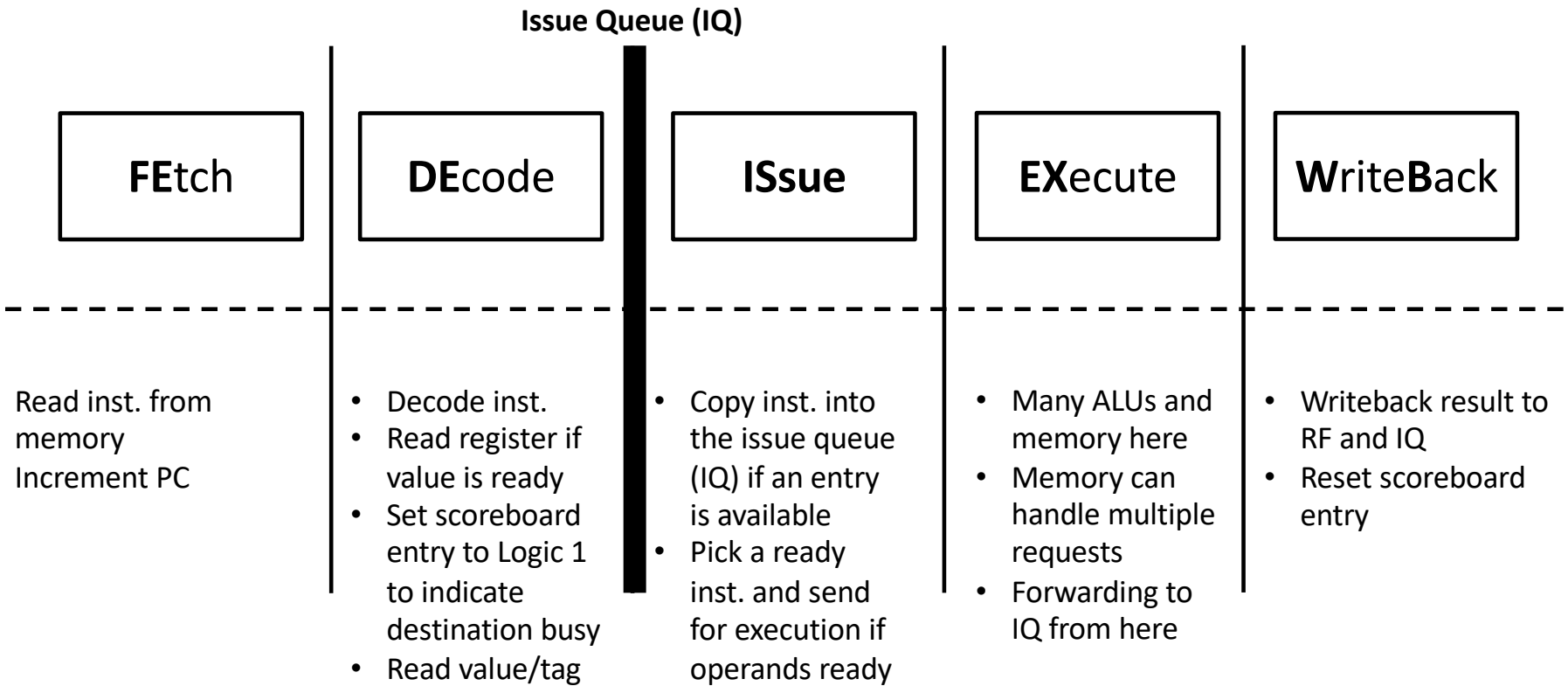
Simplified View of Pipeline



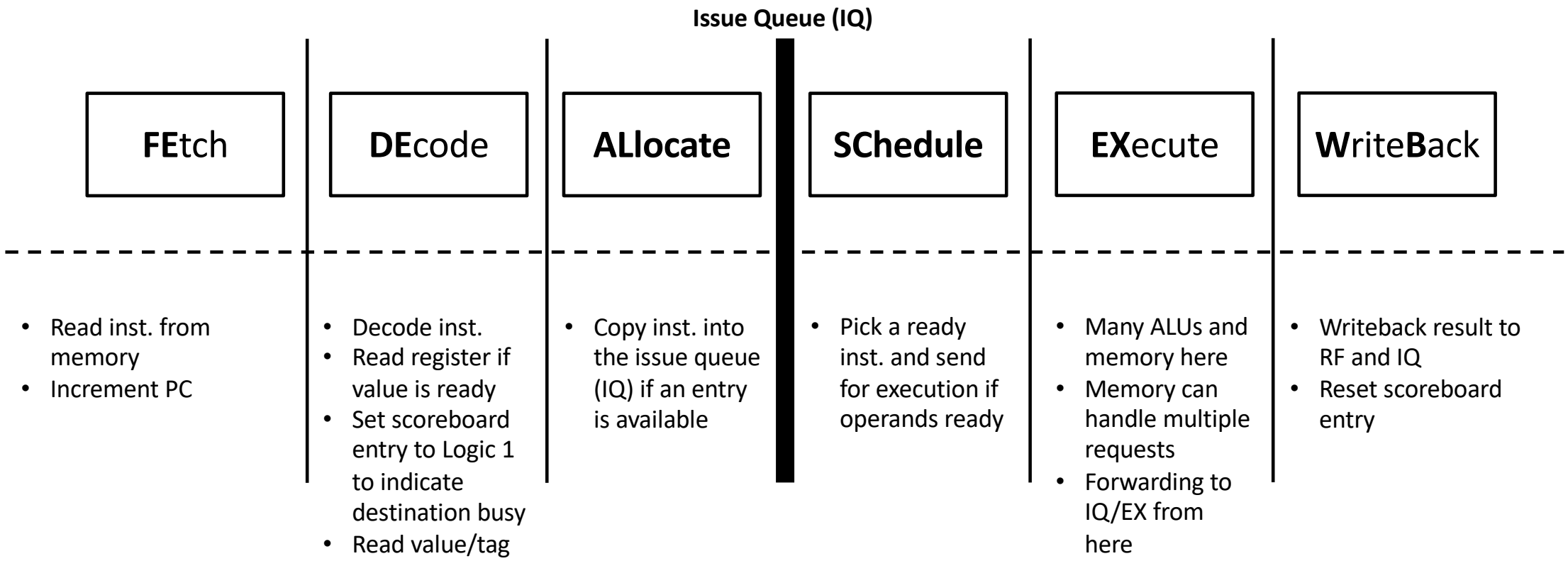
Simplified View of Pipeline



Simplified View of OOO Pipeline



Simplified View of OOO Pipeline



When to Forward?

- **Read-after-write** hazard between two instructions where the first or "older" instruction is not a load

```
ADD R0, R1, R2
SUB R4, R0, #1


MUL R12, R2, R3
ADD R0, R12, #1
```

When to Stall?

- **Load-use hazard**
  - Stall the Decode and Fetch stages when the "use" is discovered

- **PC-changing instructions**
  - Possible but not implemented for complexity reasons
  - Stall Fetch for four cycles

When to Flush?

- **Load-use hazard**
  - Flush the Execute pipeline register

- **PC-changing instructions**
  - Keep flushing the Decode stage until the new instruction (branch target) is available in the Decode pipeline register

- **Branch instructions**
  - When branch is resolved early in the Execute stage, flush the pipeline registers in the Decode and Execute stages

How does the CPU Stall and Flush?

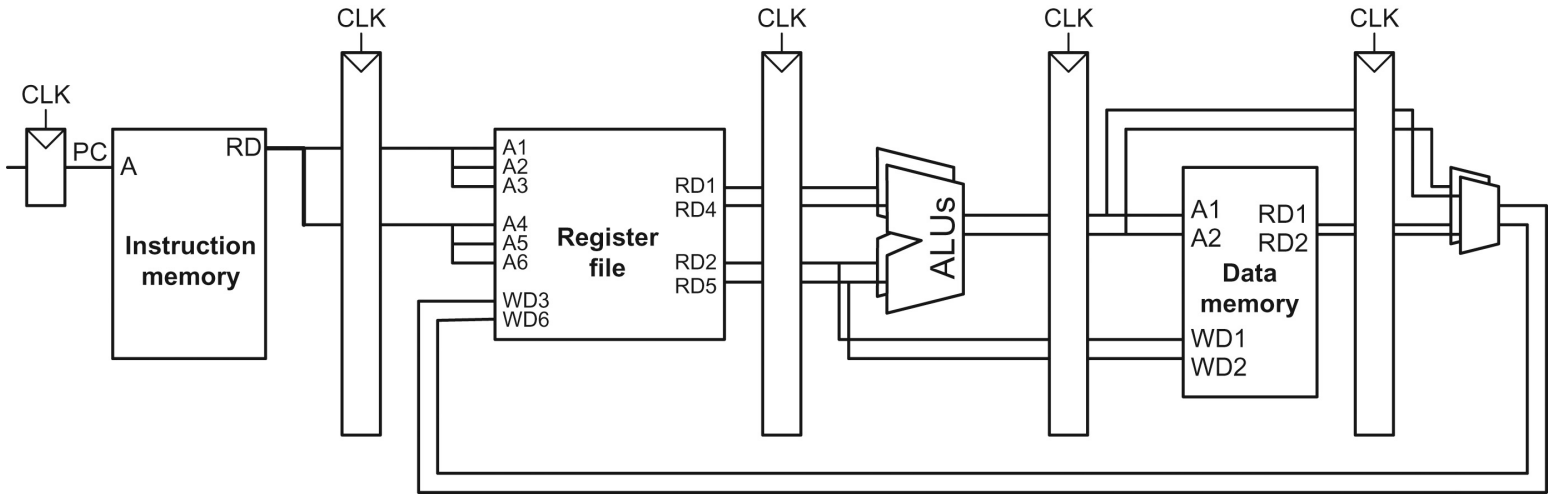
- **Stall**
  - Use Enable input to hold/retain the value stored in the pipeline register

- **Flush**
  - Use the Clear input to zero the register contents

Superscalar Processor

Superscalar: Idea and Datapath

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

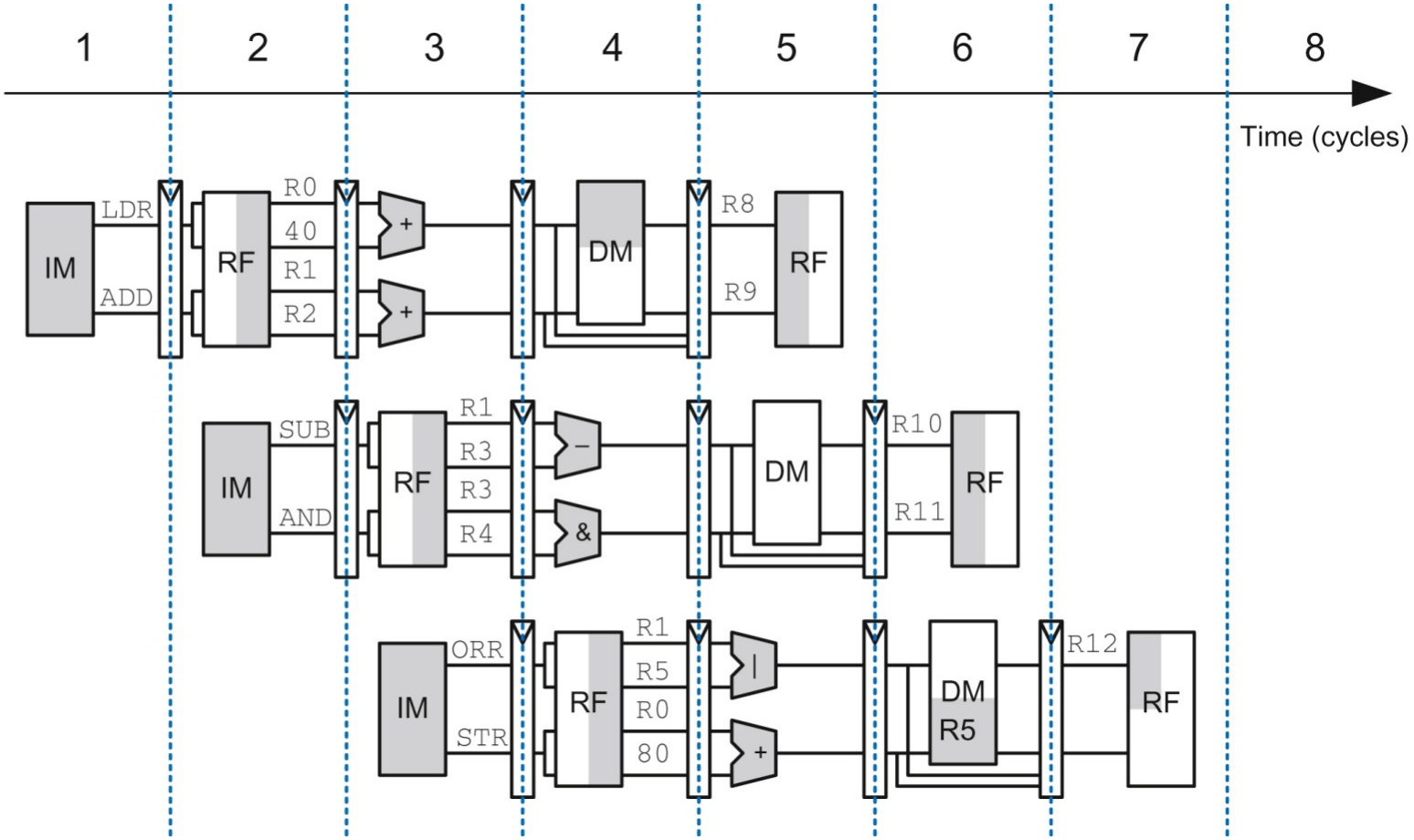


- Requires 6-ported register file (4 reads, 2 writes), 2 ALUs, 2-ported data memory
- Ideal CPI = 0.5 and IPC = 2
- Dependencies and hazards inhibit ideal IPC
- Above figure does not show forwarding and hazard detection logic

Superscalar: Pipeline Operation

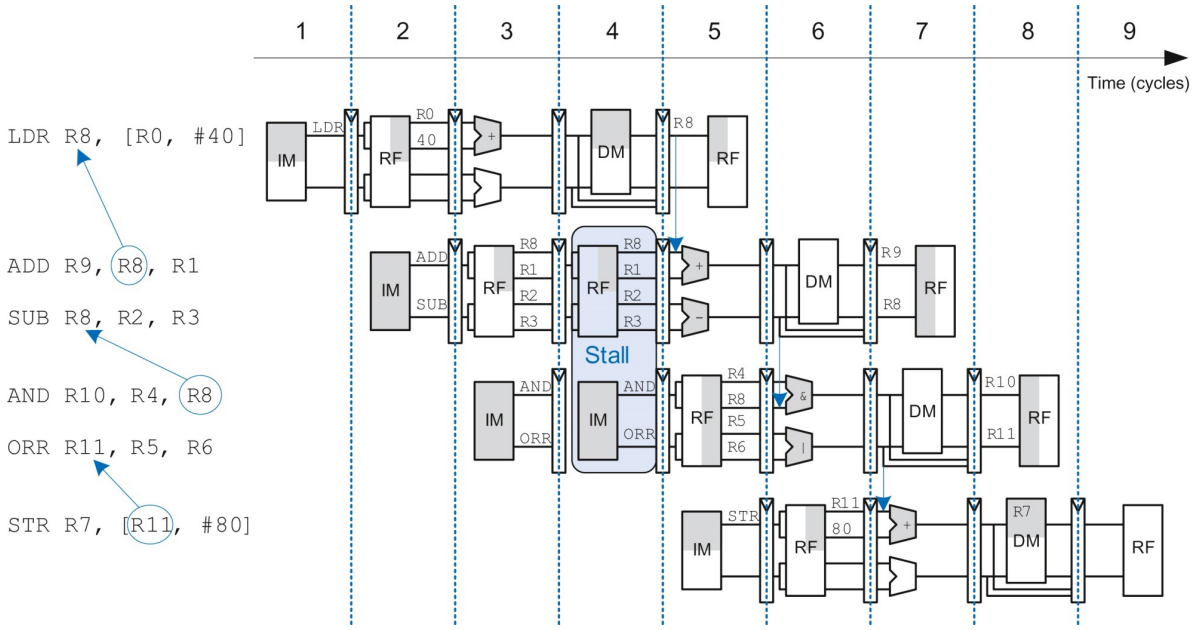
- Example program where IPC = 2 is possible

```
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



- CPU completes (on average) 6 instructions in 5 cycles (IPC of 1.2)
- Can also compute IPC from the fetch side (6 instructions are issued in five cycles)

In-Order Superscalar: Tradeoffs

- Superscalar processors encompass spatial + temporal parallelism
  - Two pipelined lanes in one CPU with duplicated resources
- 2-wide, 4-wide, and 6-wide superscalars are common (wide = way)
- Too many dependencies (data + control) in real programs
  - Hard to find many instructions to issue **(in order)** every cycle
  - Out-of-order CPUs unlock this bottleneck
- Large number of execution units and complex forwarding and hazard detection logic costs area, power, and energy

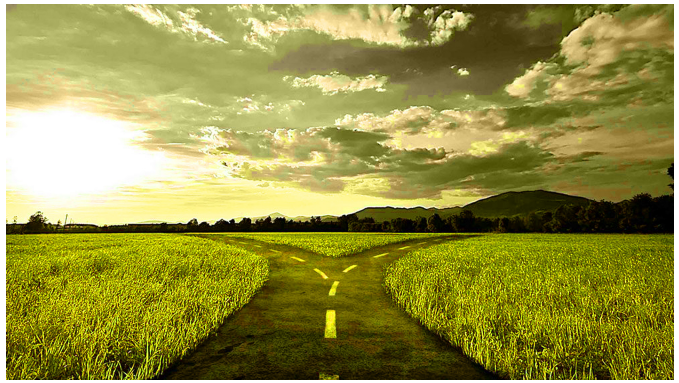
In-Order Superscalar: Role of Compiler

- **In-order** (superscalar) CPU: Instructions are executed in the exact order determined by the assembly programmer or compiler

- The compiler can change instruction order to maximize pipeline utilization

- Goal: Achieve maximum IPC (e.g., IPC of 2 for 2-wide superscalar CPUs)

Branch Prediction



Static Branch Prediction

- **Static (fixed) policy #1:** Always predict that the branch is **not taken**
- **Static policy # 2:** Always predict that the branch will be **taken**
- The cost of a branch misprediction (branch misprediction penalty) increases for superscalars
  - Effort to process **"wrong path"** instructions is wasted
- **We need more accurate branch predictors (>99% accuracy)**

Dynamic Branch Prediction

- Predict the outcome of a branch instruction (in fetch stage) based on the *recent behavior* of the branch

- **What do we need?**

  - Branch identification (`PC` uniquely identifies a branch)

  - Recent branch behavior (taken/untaken last time)

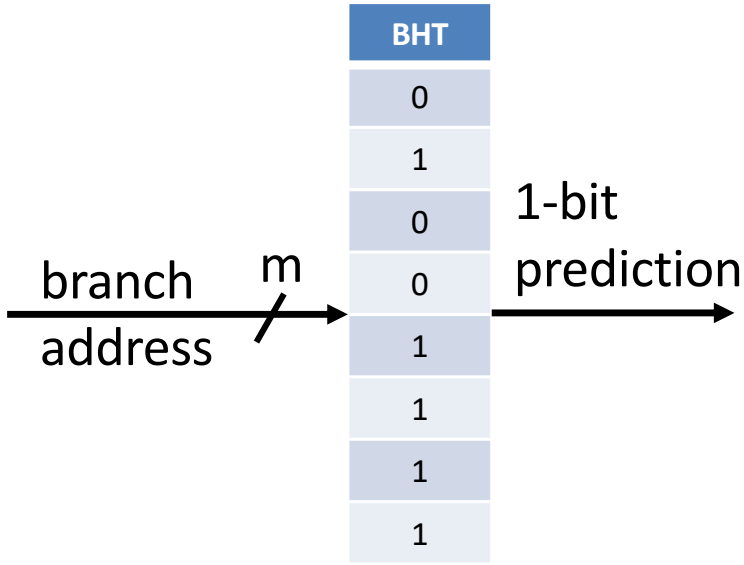
Branch Identification & Behavior

- **Branch identification**
  - Use the branch address in instruction memory
  - Can grab it from PC

- **Branch behavior**
  - Outcome of the **condition test** from ALU
  - Also need to store the **branch target** the last time the branch executed

One-Bit Predictor

- Branch History Table (**BHT**) or Branch Prediction Buffer
  - Small amount of memory indexed by low-order branch address bits
  - Store a single bit that says branch was recently taken or not



- Due to limited entries in the table, there are conflicts (aka. aliasing)

One-Bit Predictor: Operation

- **Placed in the Fetch stage**
  - Predicted untaken: Fetch the next instruction
  - Predicted taken: Compute the target address and fetch from target

- **Updates to the BHT**
  - Nothing to do if outcome matches prediction
  - If outcome does not match prediction
    - Flip the entry in the BHT
    - Flush the pipeline and update the PC

- **Questions**
  - Is correctness affected by misprediction?
  - Is performance affected by misprediction?

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV   R0,  #1
FOR
    CMP   R0,  #10
    BGE   DONE
    ADD   R0,  R0,  #1
    B FOR
DONE
```

- What is the prediction accuracy of a 1-bit branch predictor?

i =	1	2	3	4	5	6	7	8	9	10

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV   R0,  #1
FOR
    CMP   R0,  #10
    BGE   DONE
    ADD   R0,  R0,  #1
    B FOR
DONE
```

- What is the prediction accuracy of a 1-bit branch predictor?

i =	1	2	3	4	5	6	7	8	9	10
Actual Direction	NT	NT	NT	NT	NT	NT	NT	NT	NT	T

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV   R0,  #1
FOR
    CMP   R0,  #10
    BGE   DONE
    ADD   R0,  R0,  #1
    B FOR
DONE
```

- What is the prediction accuracy of a 1-bit branch predictor?

i =	1	2	3	4	5	6	7	8	9	10
Actual Direction	NT	NT	NT	NT	NT	NT	NT	NT	NT	T
Current State/Prediction										
New State										

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV   R0,  #1
FOR
    CMP   R0,  #10
    BGE   DONE
    ADD   R0,  R0,  #1
    B FOR
DONE
```

- What is the prediction accuracy of a 1-bit branch predictor?

i =	1	2	3	4	5	6	7	8	9	10
Actual Direction	NT	NT	NT	NT	NT	NT	NT	NT	NT	T
Current State/Prediction	T	NT	NT	NT	NT	NT	NT	NT	NT	NT
New State										

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV   R0,  #1
FOR
    CMP   R0,  #10
    BGE   DONE
    ADD   R0,  R0,  #1
    B FOR
DONE
```

- What is the prediction accuracy of a 1-bit branch predictor?

i =	1	2	3	4	5	6	7	8	9	10
Actual Direction	NT	NT	NT	NT	NT	NT	NT	NT	NT	T
Current State/Prediction	T	NT	NT	NT	NT	NT	NT	NT	NT	NT
New State	NT	NT	NT	NT	NT	NT	NT	NT	NT	T

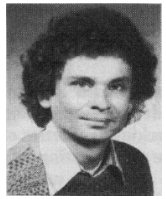
Anomalous Decision

- Accuracy of one-bit predictor is 80% for a branch that is NOT TAKEN **90%** of the time

- **Anomaly:** When branches that are strongly biased toward one direction suddenly takes a different path or direction

- A 1-bit predictor is "thrown off" by a single **anomolous** decision

Smith's Algorithm

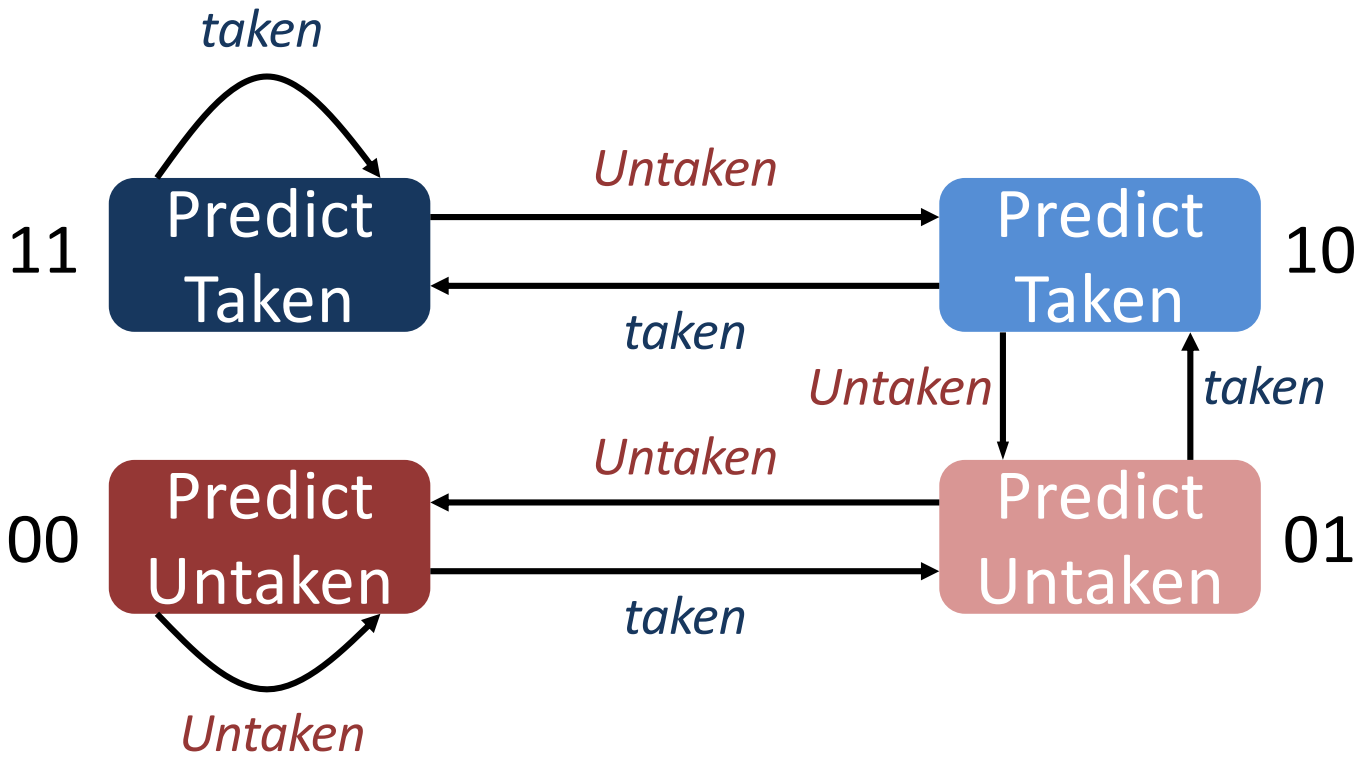


- **1979:** *James E. Smith* notices the performance pathology of 1-bit predictor at loop termination
  - Proposes Smith's branch prediction algorithm
  - **Key insight:** Add hysterisis (inertia) to the predictor's state
- The same outcome must occur multiple times to reach the strong states
- A saturating counter maps the outcomes of several recent branches on to a **counter** with different states

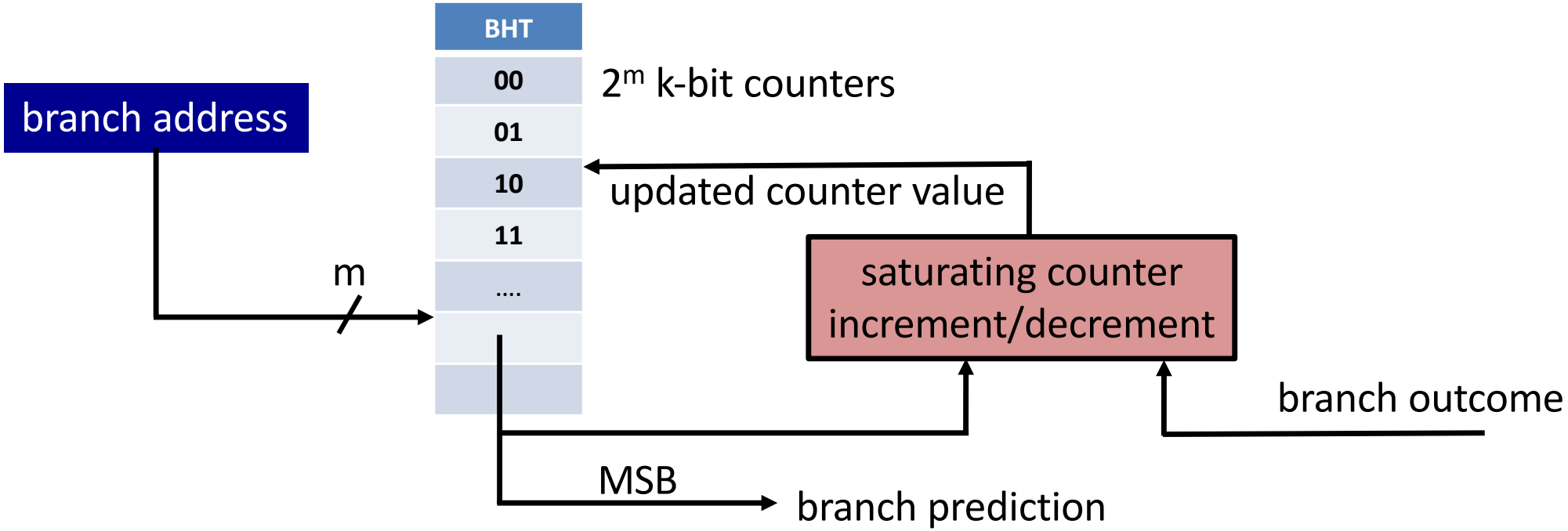
k = 2

- **Four states**
  - Strongly not-taken (SN or **0**0)
  - Weakly not-taken (WN or **0**1)
  - Weakly taken (WT or **1**0)
  - Strongly taken (ST or **1**1)

Smith's Algorithm



Smith's Predictor Hardware (k = 2)



Accuracy of Smith's Predictor

Below: Accuracy of **Smith$_1$ (1-bit counter)** and **Smith$_2$ (2-bit counter)** on a sequence of branches with sudden shifts in branch behavior

Branch Direction
1
1
0
1
1
1

Accuracy of Smith's Predictor

Below: Accuracy of **Smith$_1$ (1-bit counter)** and **Smith$_2$ (2-bit counter)** on a sequence of branches with sudden shifts in branch behavior

Branch Direction	Smith$_1$	
	State	Prediction
1	1	1
1	1	1
0	1	1 (misprediction)
1	0	0 (misprediction)
1	1	1
1	1	1

Accuracy of Smith's Predictor

Below: Accuracy of **Smith$_1$ (1-bit counter)** and **Smith$_2$ (2-bit counter)** on a sequence of branches with sudden shifts in branch behavior

Branch Direction
1
1
0
1
1
1

Smith$_2$	
State	**Prediction**
11	1
11	1
11	1 (misprediction)
10	1
11	1
11	1

Accuracy of Smith's Predictor

Below: Accuracy of **Smith$_1$ (1-bit counter)** and **Smith$_2$ (2-bit counter)** on a sequence of branches with sudden shifts in branch behavior

Branch Direction	Smith$_1$		Smith$_2$	
	State	Prediction	State	Prediction
1	1	1	11	1
1	1	1	11	1
0	1	1 (misprediction)	11	1 (misprediction)
1	0	0 (misprediction)	10	1
1	1	1	11	1
1	1	1	11	1

Branch Target Buffer (BTB)

- Buffer = A small memory for storing “some” information

- Recall the **CPU needs to know** in the fetch stage
  - Branch direction
  - Branch target address

- BTB stores the target addresses for taken branches

- Does not make sense to search the BTB for targets of untaken branches

Operation with BTB

- Branch is predicted to be taken
  - Get target address from BTB
- Branch is predicted untaken
  - `PC = PC + 4`
- If the prediction is correct: **continue** normal execution
- If the prediction is incorrect: **flush** all pipeline stages containing instructions from the mispredicted path

Correlating Branch Predictors

- In real programs, the outcome of one branch often depends on the behavior of other branches

```
aa = 0; bb = 0;
if (cond1)
        aa = 1;
if (aa == bb)
        {…}
```

- Traditional (non-correlating) predictors that rely only on the outcome of a single branch fail to capture these relationships

- **Correlating branch predictors improve accuracy by using**
  - Local history (past behavior of the same branch)
  - Global history (outcomes of recent branches across the program)
  - Branch address (to distinguish branches with similar histories)

A Lot More to Say on Branch Prediction!

- **Important component of a modern processor**
  - Especially superscalar and out-of-order processors
  - Prediction accuracy above 99%

- **State of art:** Deep neural networks, machine learning approaches

- **COMP4045:** Students implement and compare state of the art branch predictors in a C++ simulator

Real pipelines have caches and real memory latencies!

- **Each memory access costs 100s of cycles (we assumed 1 cycle data memory access for simplicity)**
  - Cache hit cost 1– 4 cycles
  - Cache miss costs close to 100 cycles
  - Therefore, an **in-order** pipelined CPU can stall for many cycles on memory accesses

- **Next step**
  - Out-of-order CPU that continues doing useful work in the presence of long-latency memory accesses