COMP2300-COMP6300-ENGN2219
Computer Organization & Program Execution

Convener: Shoaib Akram

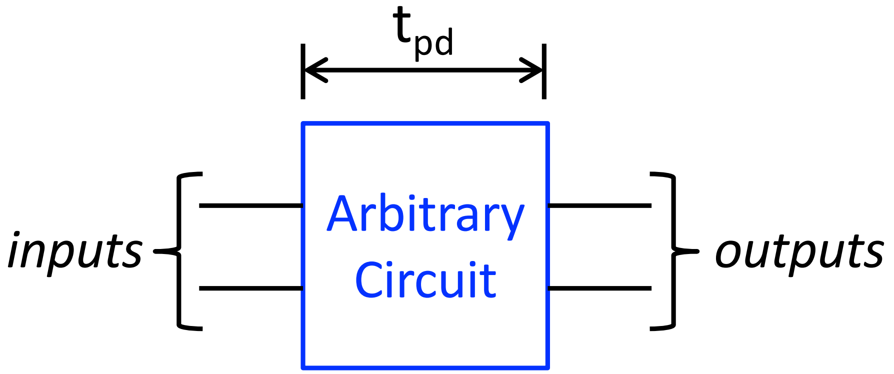
shoaib.akram@anu.edu.au

Australian National University

General Idea of Pipelining

Speed of a Circuit

- At a high level, an arbitrary digital circuit processes a group of inputs and produces a group of outputs



- We needs **metrics** to quantify the speed with which we can process inputs to produce outputs (i.e., the performance of a circuit)
  - **Latency:** The time required to produce one group of outputs once the inputs arrive (propagation delay, end-to-end latency)
  - **Throughput:** The number of input groups processed per unit of time

Example: Latency/Throughput

- 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

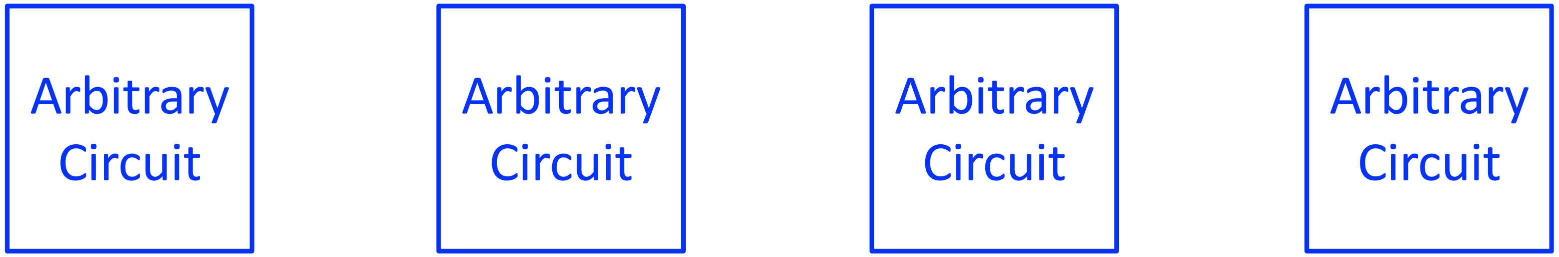
- Latency (hours/tray):
- Throughput (trays/hour):

Parallelism

- Many scenarios in the real-world requires us to increase the throughput of the digital system
  - # add operations per second (ALU)
  - # instructions per second (CPU)
- **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

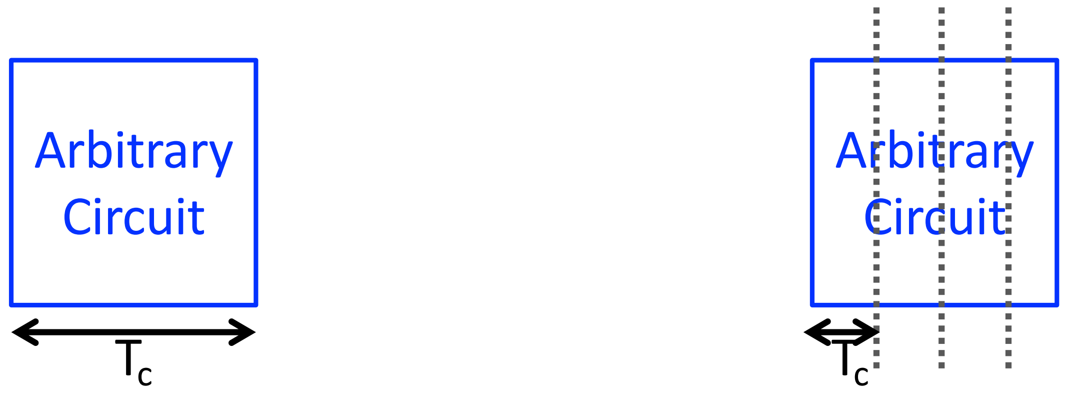
Temporal Parallelism

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

Automotive Pipeline

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

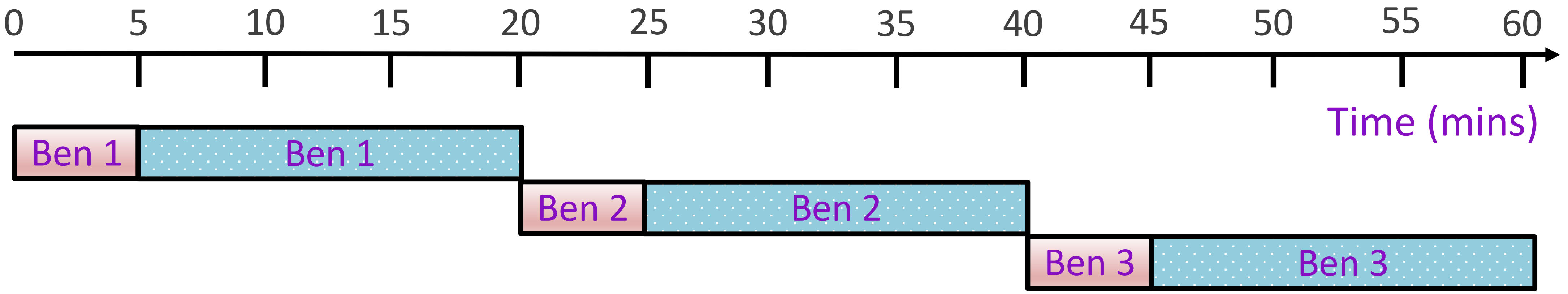


- The challenge of pipelining is to find stages of equal length
- Let’s go back to baking cookies

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

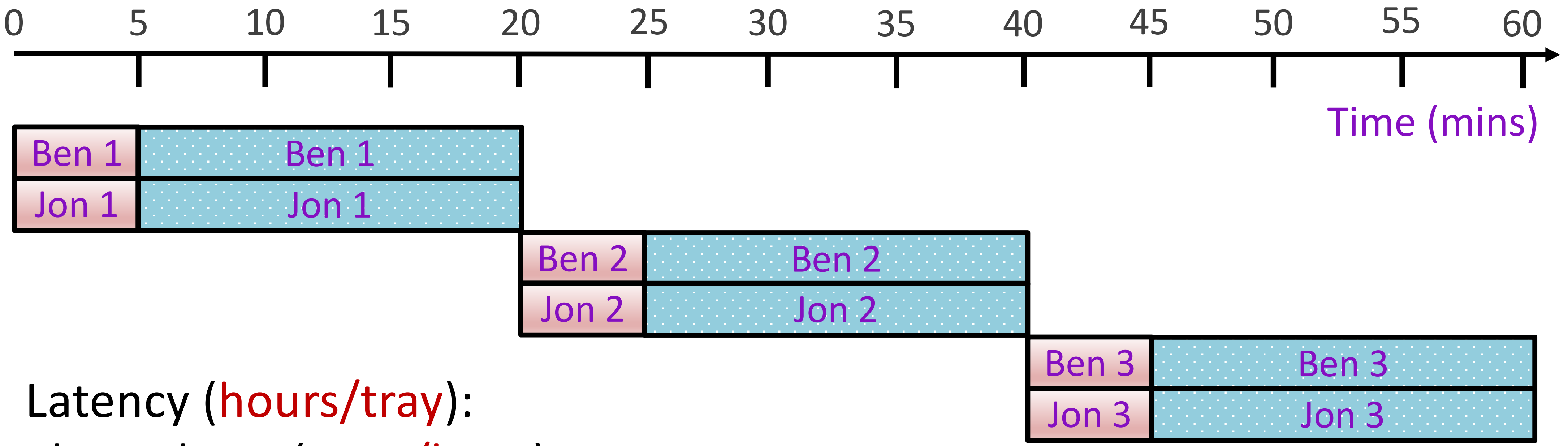
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

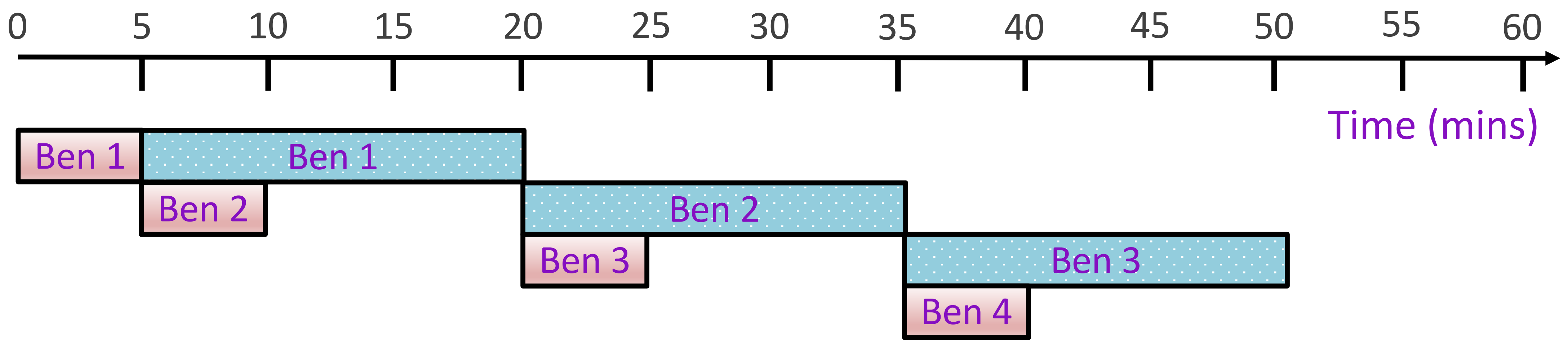


Latency (hours/tray):

Throughput (trays/hour):

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

Pipelining (Ben Only)

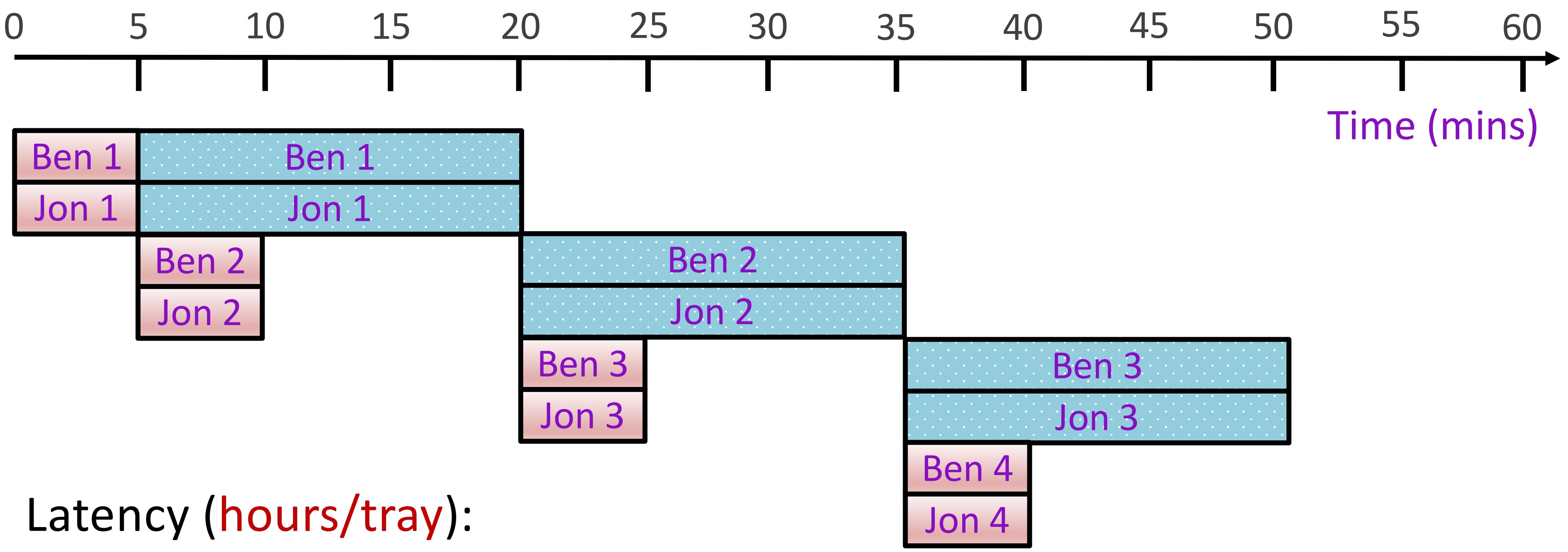


Latency (hours/tray):

Throughput (trays/hour):

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

Spatial + Temporal Parallelism



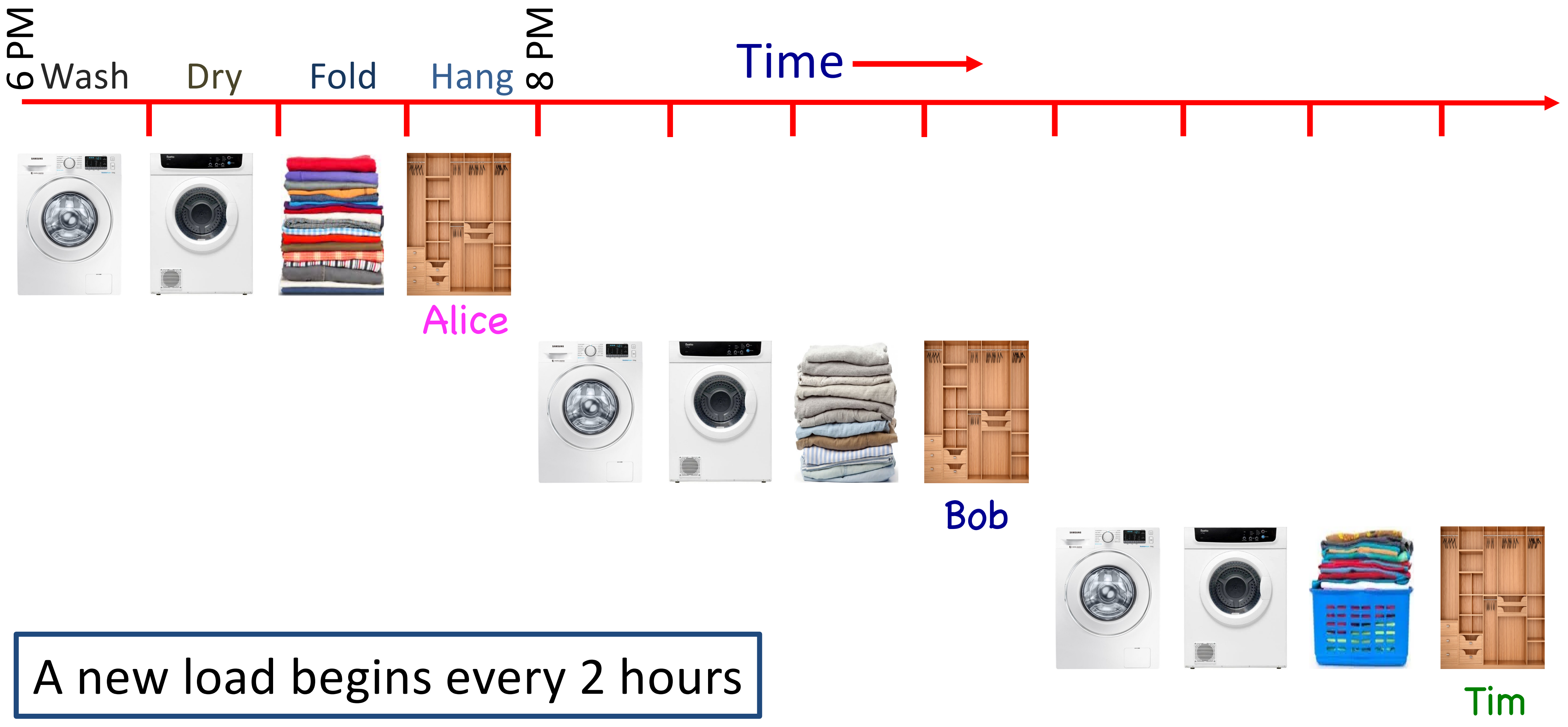
Latency (hours/tray):

Throughput (trays/hour):

Answers Explained

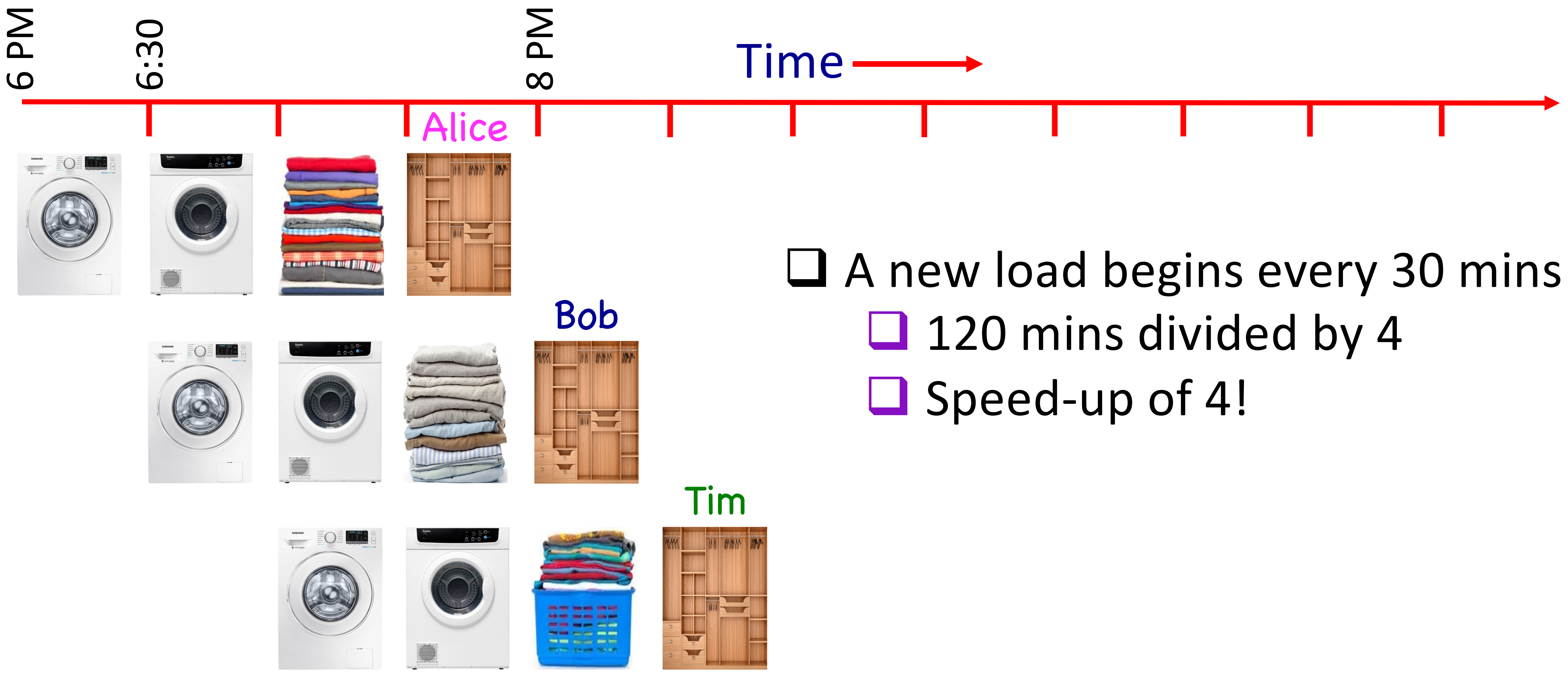
- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a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



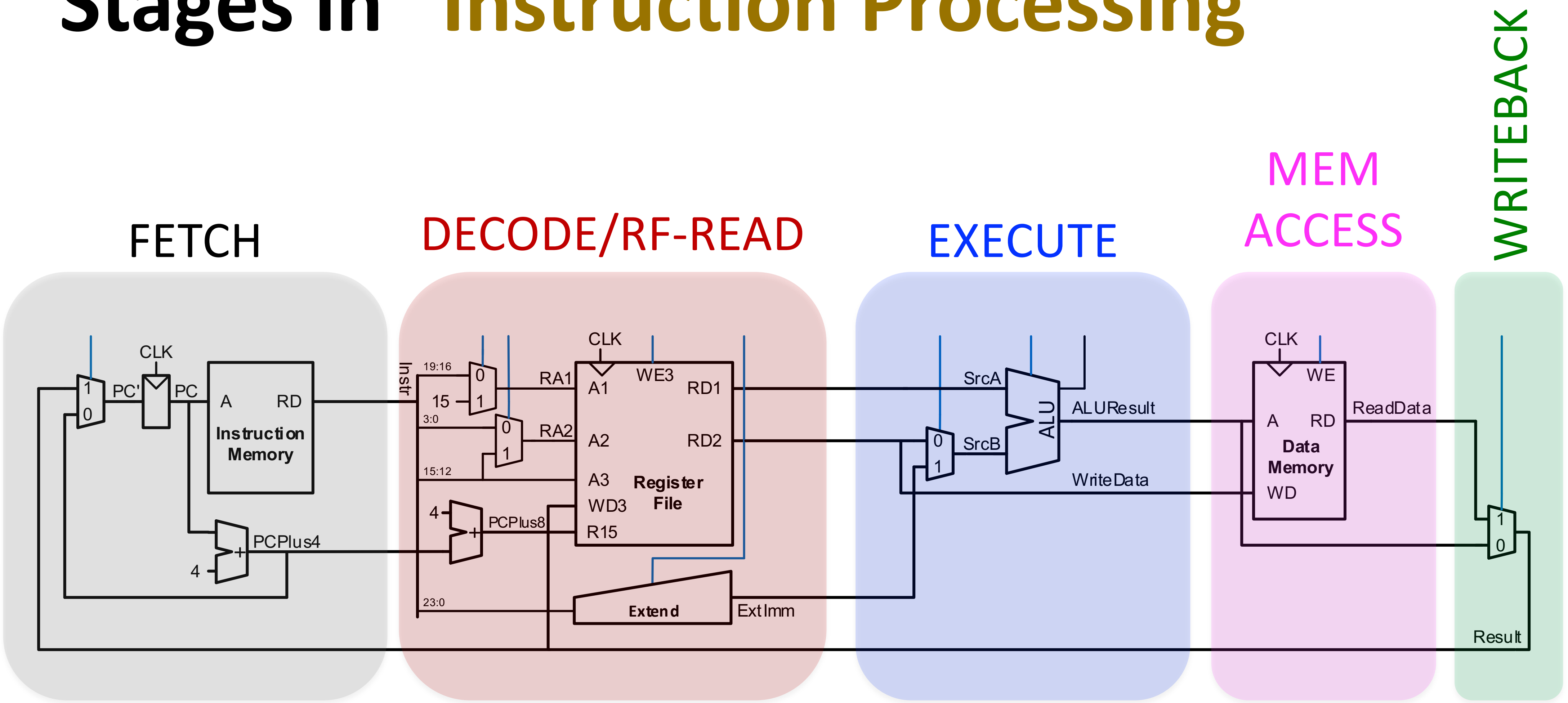
- ❑ A new load begins every 30 mins
  - ❑ 120 mins divided by 4
  - ❑ Speed-up of 4!

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)

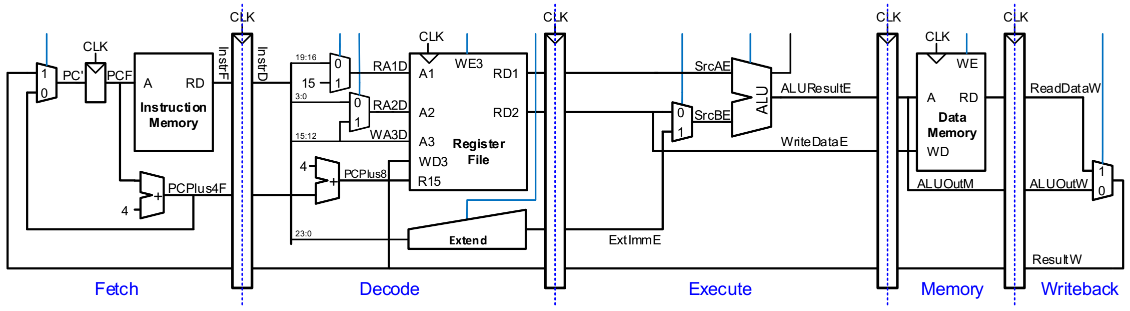
- The work of a stage should be preserved in a PPR each cycle
- PPR acts as a source of data the next stage needs in a subsequent cycle
- If any subsequent stage down the pipeline needs data from an earlier stage it must be passed through the PPRs
  - .... Things don’t always go smoothly as we shall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

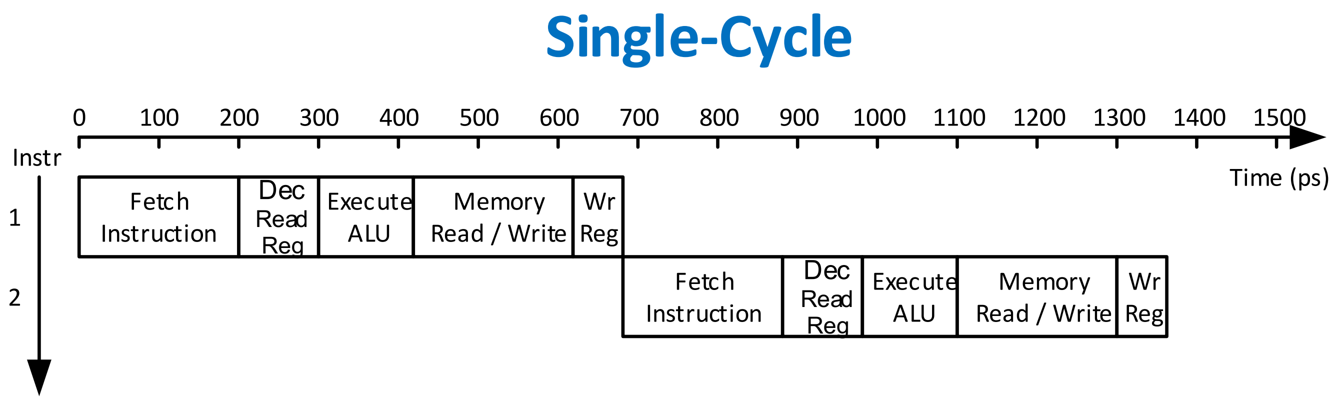


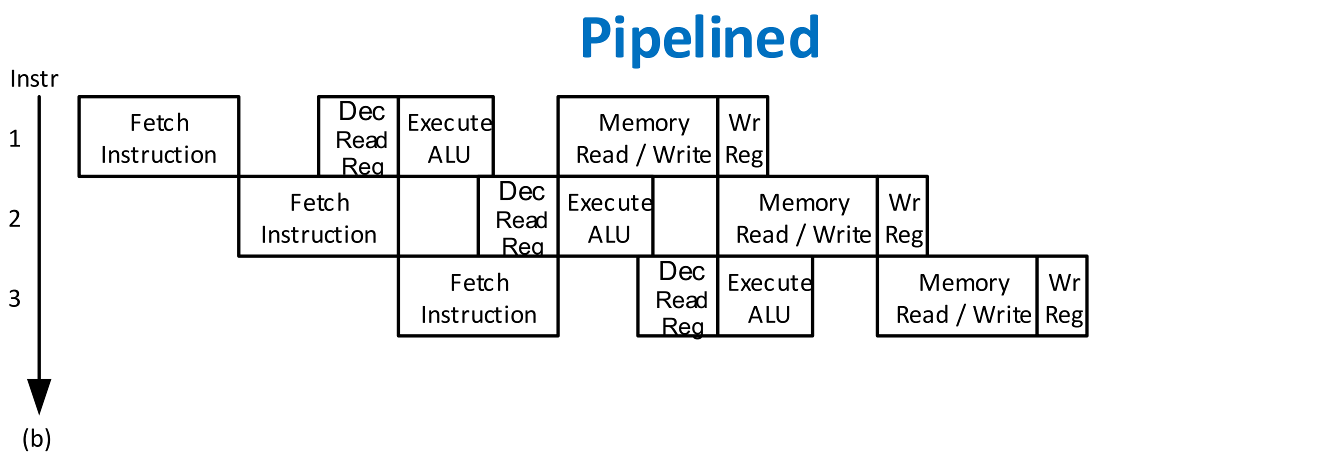
Timing Diagrams

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

Timing Diagrams

Assumption of logic element delays from Table 7.5 of textbook





Performance Analysis

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

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

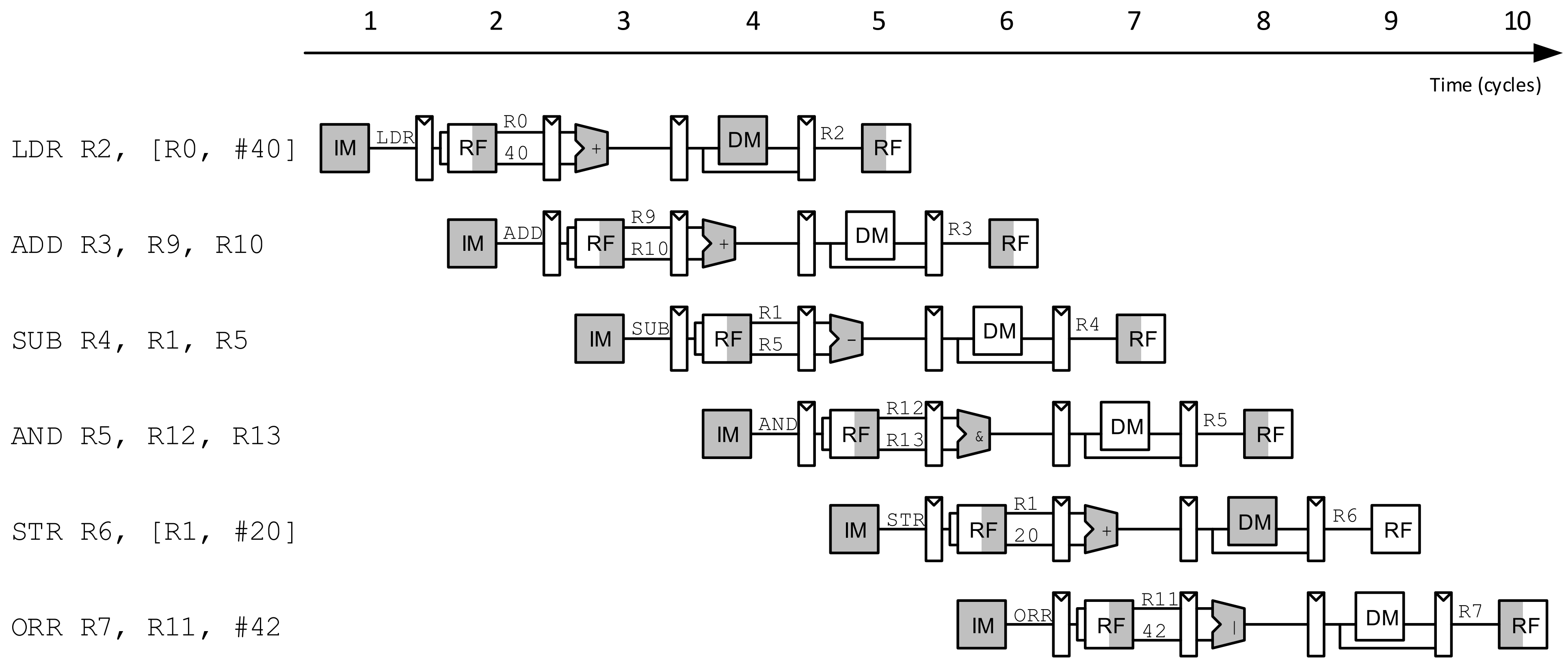
Instruction Latency with Pipelining

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

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

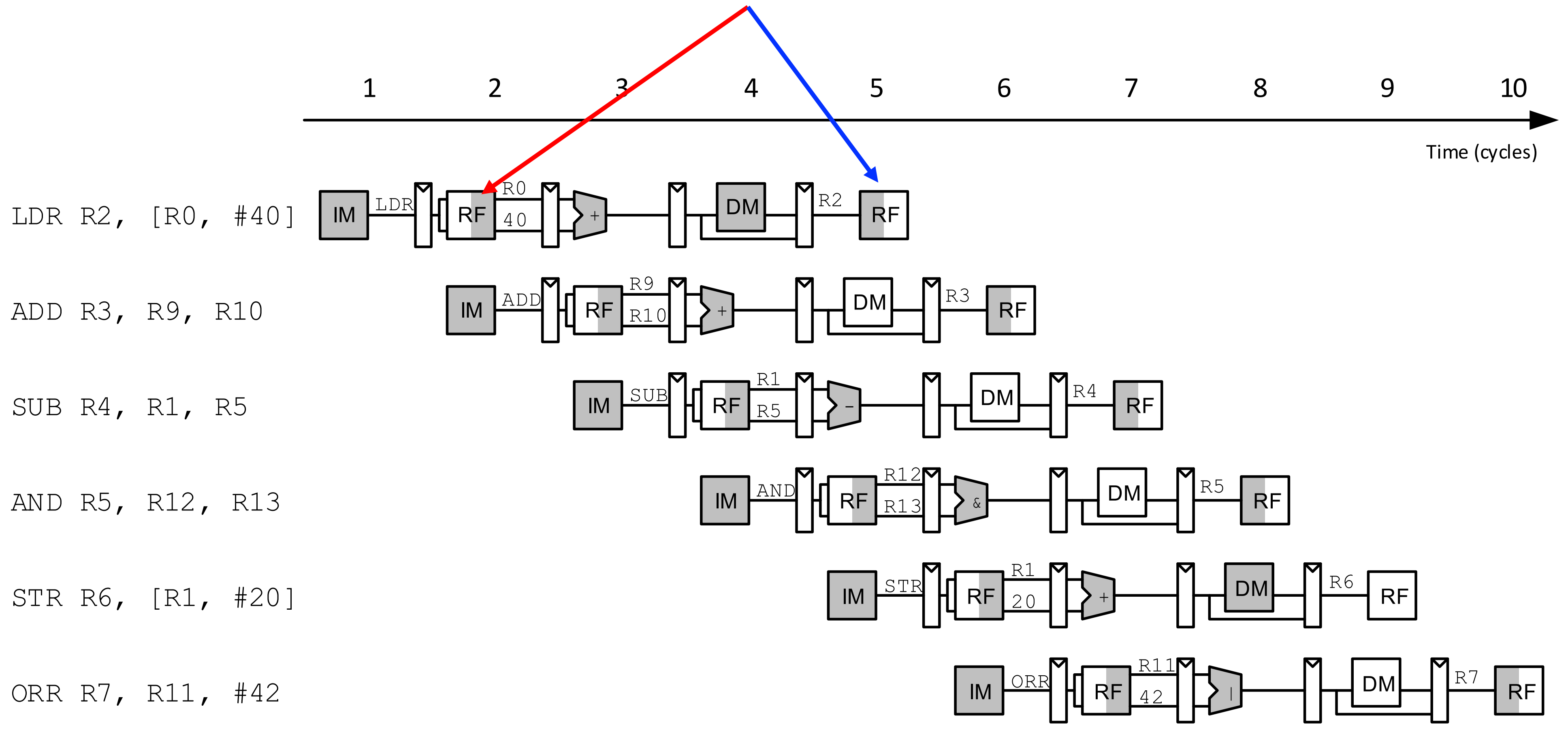
- Pipelining helps increase the throughput of an entire workload
  - Workload = Number of instructions
  - Workload must be **“sufficiently”** large

Abstract Diagrams of Pipelined uArch



```
LDR R2, [R0, #40]
ADD R3, R9, R10
SUB R4, R1, R5
AND R5, R12, R13
STR R6, [R1, #20]
ORR R7, R11, #42
```

RF Read/Write in Pipelined uArch



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

Simplified View of Pipelining

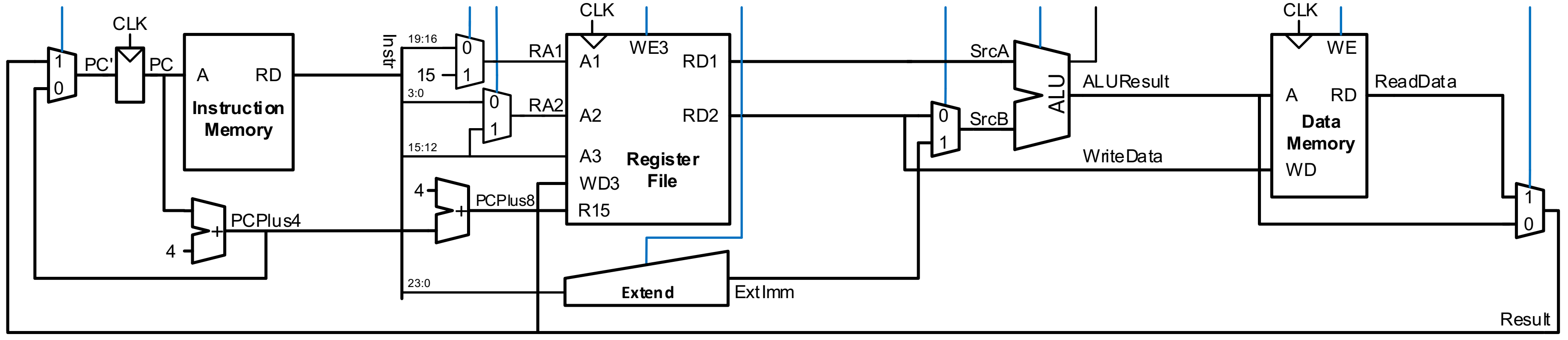
	C1	C2	C3	C4	C5	C6	C7	C8		
I1	F	D	E	M	W					
I2		F	D	E	M	W				
I3			F	D	E	M	W			
I4				F	D	E	M	W		
↓					F	D	E	M	W	
Insts						F	D	E	M	W

Let’s complete the picture

- Start with the single-cycle microarchitecture

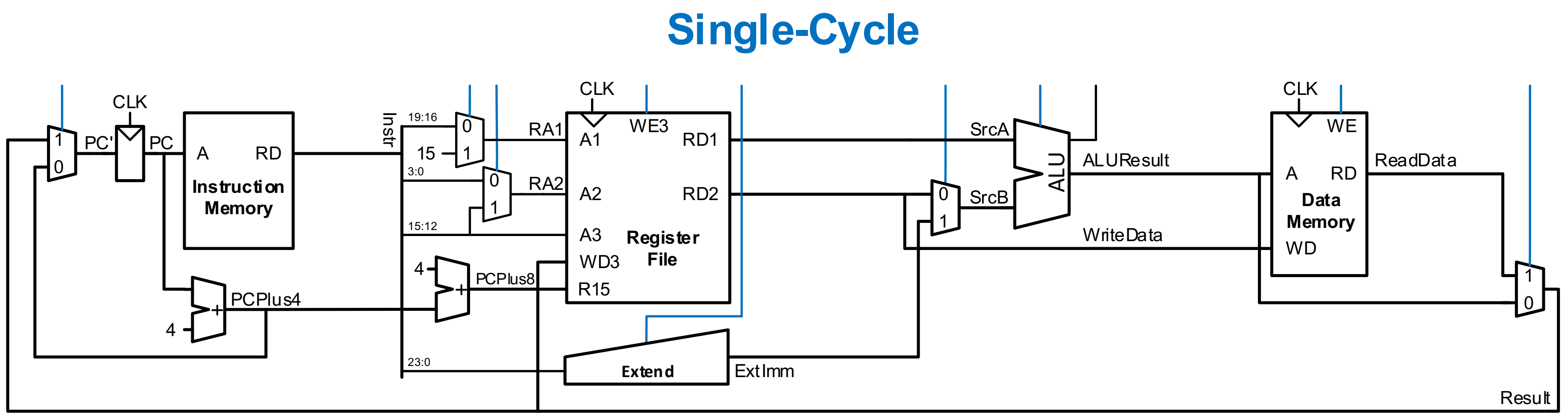
- And insert pipeline registers

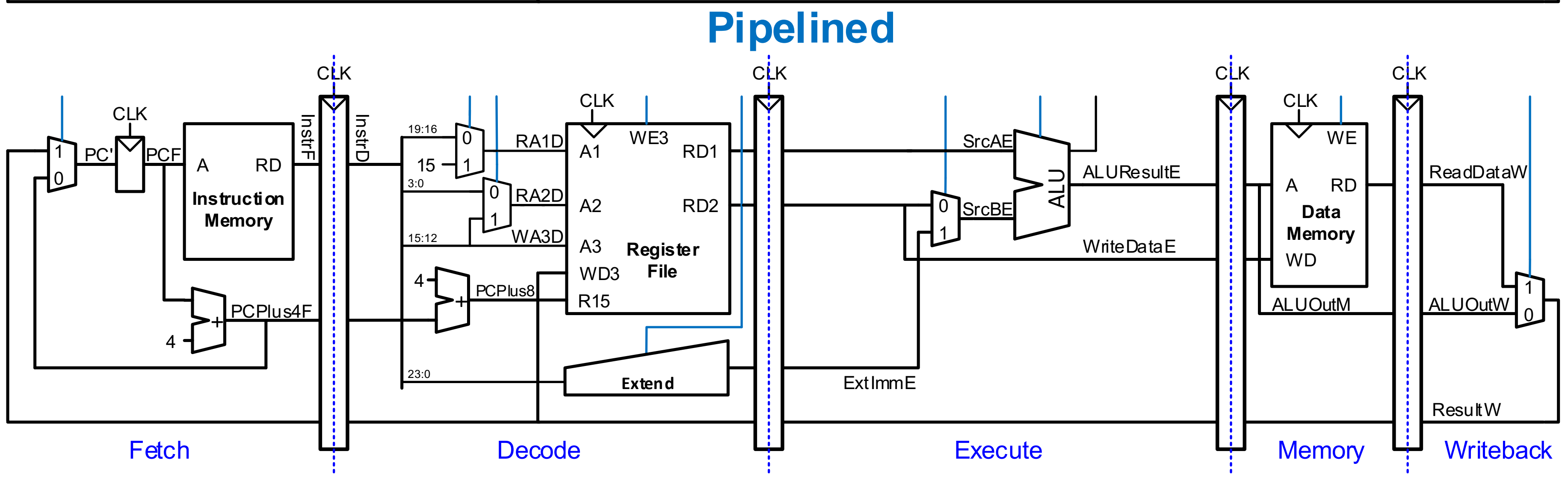
Single-Cycle



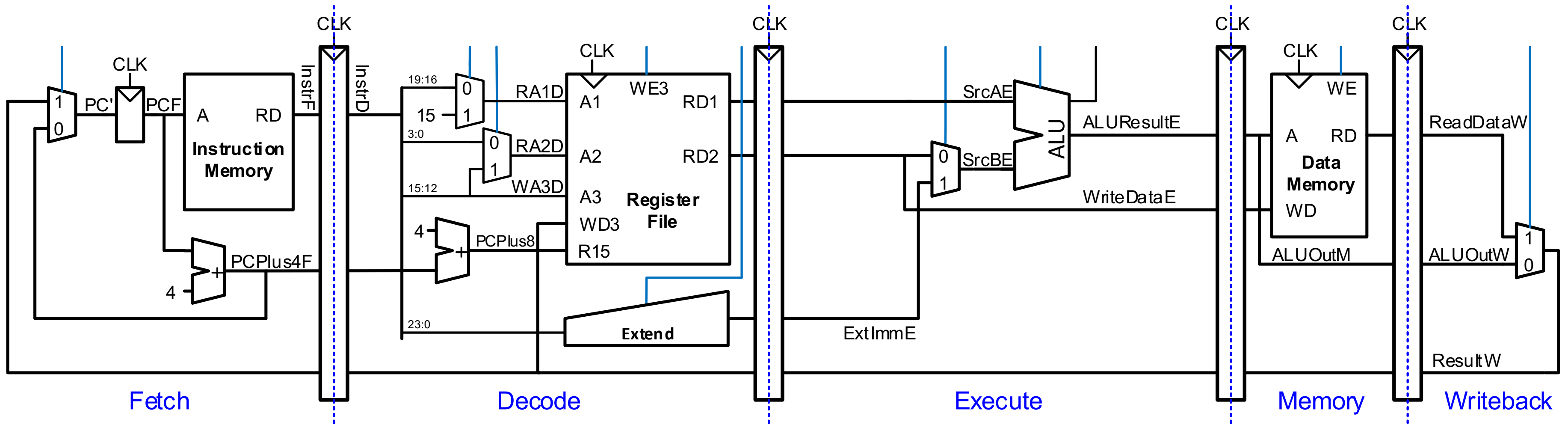
- Once we insert pipeline registers, we would need to pass the results of one stage to the next stage via the pipeline registers
- What is the outcome of the FETCH stage?

Pipeline Microarchitecture





Pipeline Microarchitecture



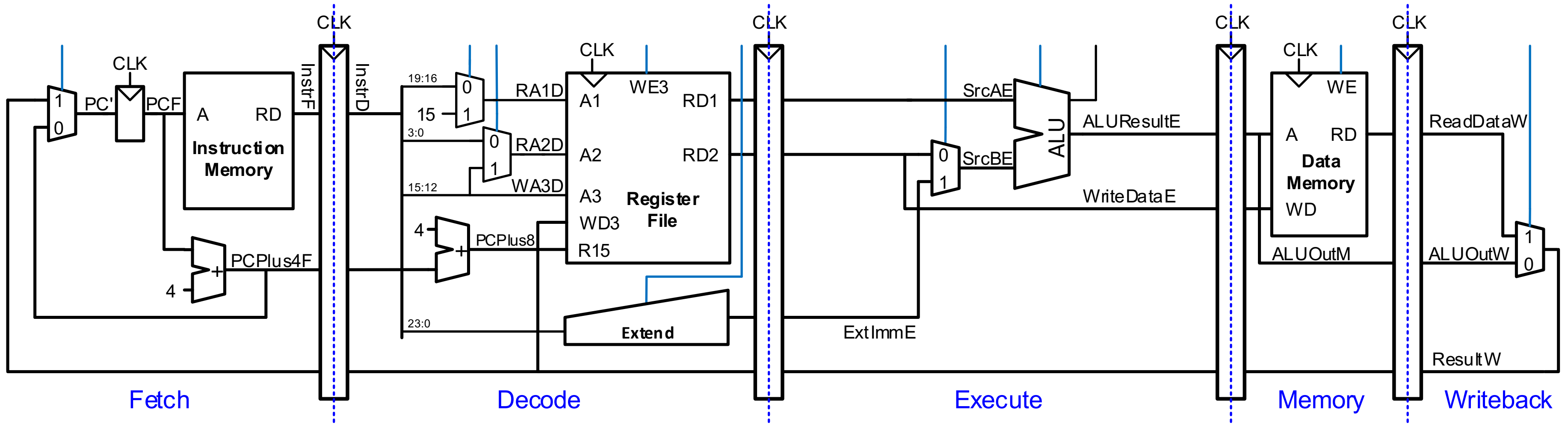
- Stages and their boundaries are indicated in blue
- Signals are given a suffix **(F, D, E, M, or W)** to indicate the stage in which they reside

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]
```

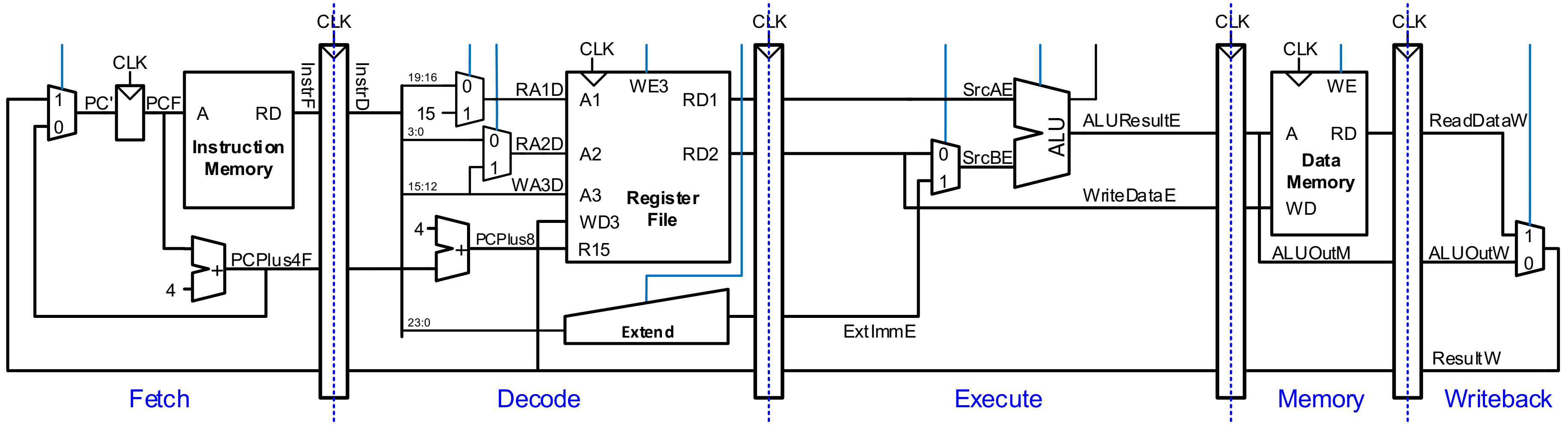
Pipeline Operation: Cycle 1



I1

- ❑ Is the pipeline fully utilized? NO

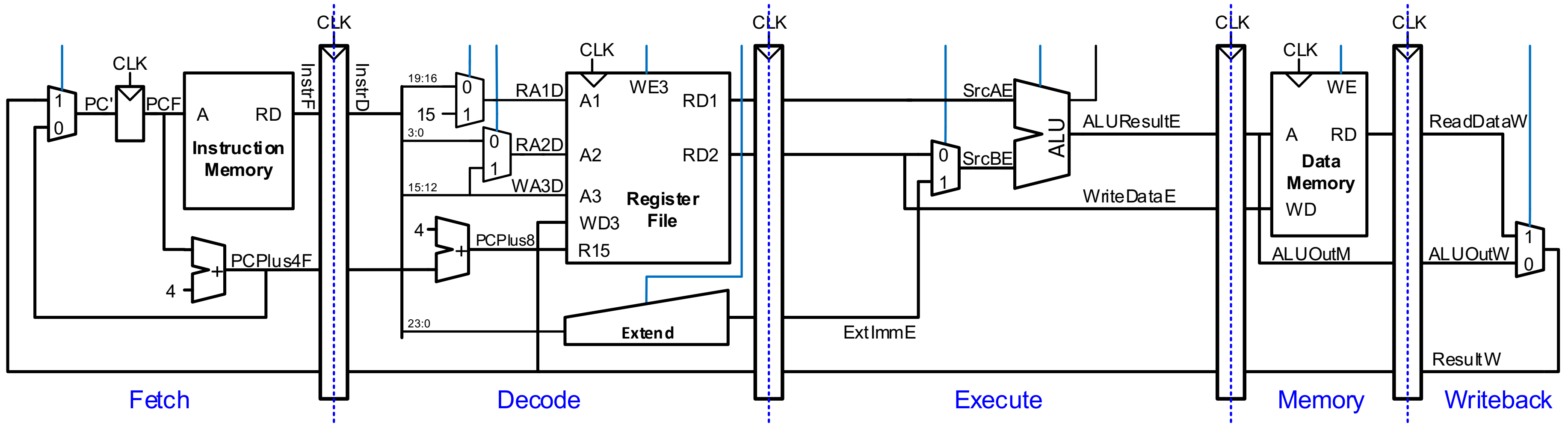
Pipeline Operation: Cycle 2



I2 I1

❑ Is the pipeline fully utilized? NO

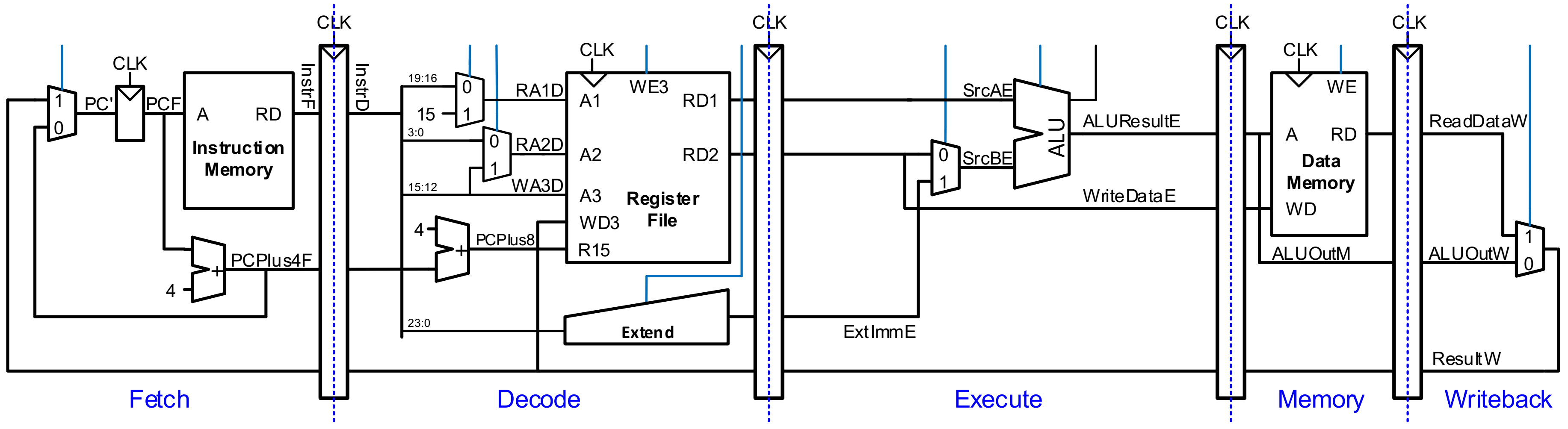
Pipeline Operation: Cycle 3



I3 I2 I1

- ❑ Is the pipeline fully utilized? NO

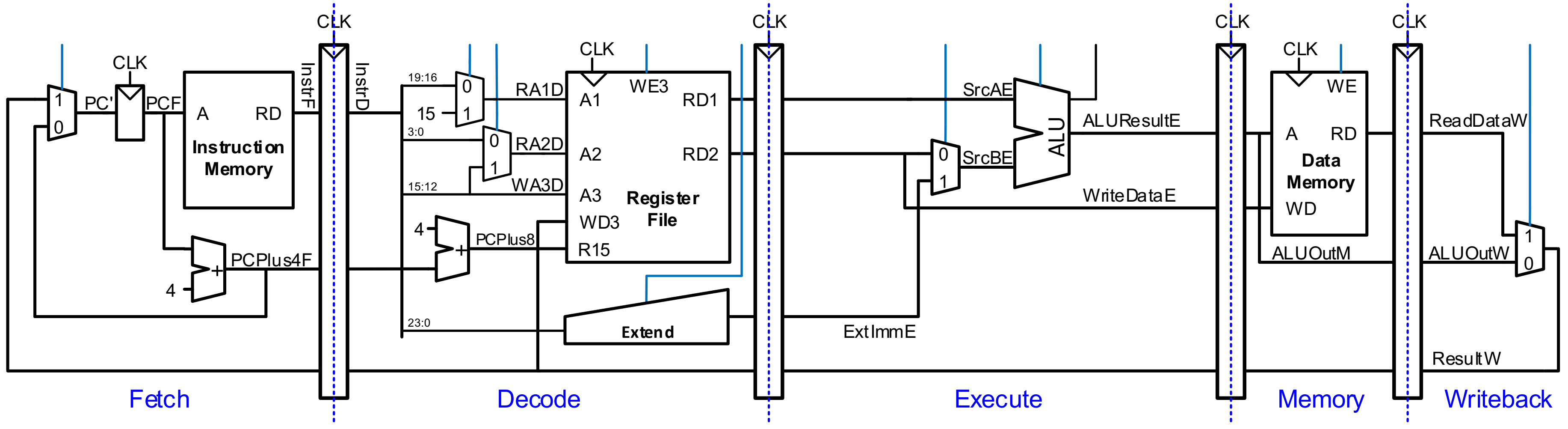
Pipeline Operation: Cycle 4



I4 I3 I2 I1

❑ Is the pipeline fully utilized? NO

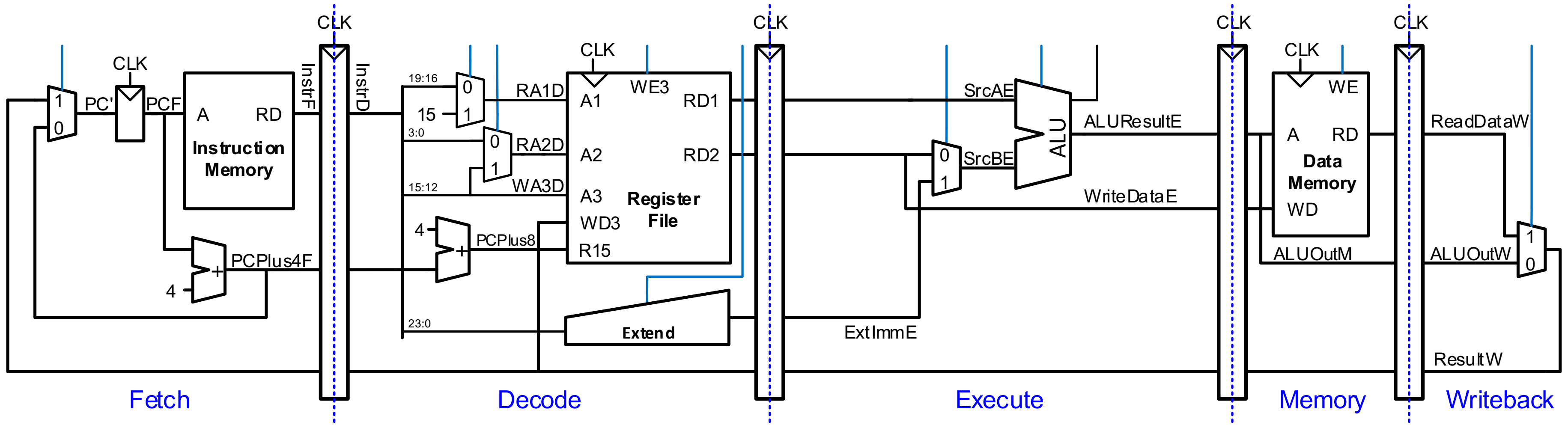
Pipeline Operation: Cycle 5



I5 I4 I3 I2 I1

- Is the pipeline fully utilized? YES

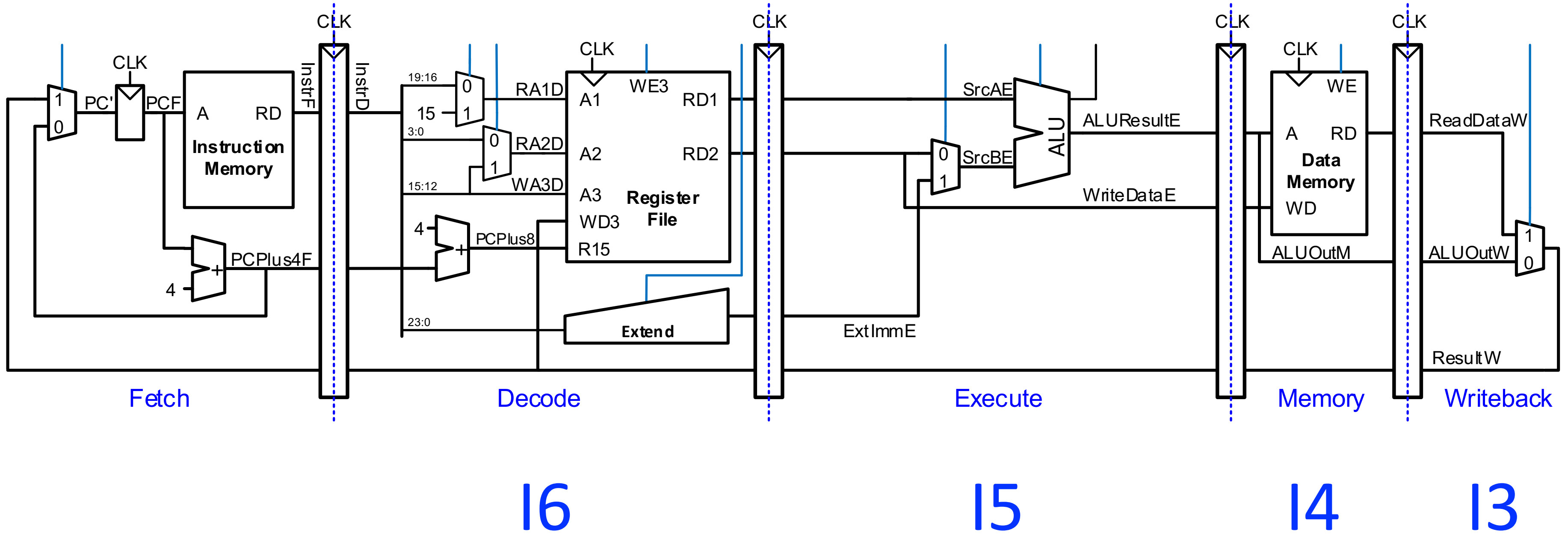
Pipeline Operation: Cycle 6



I6 I5 I4 I3 I2

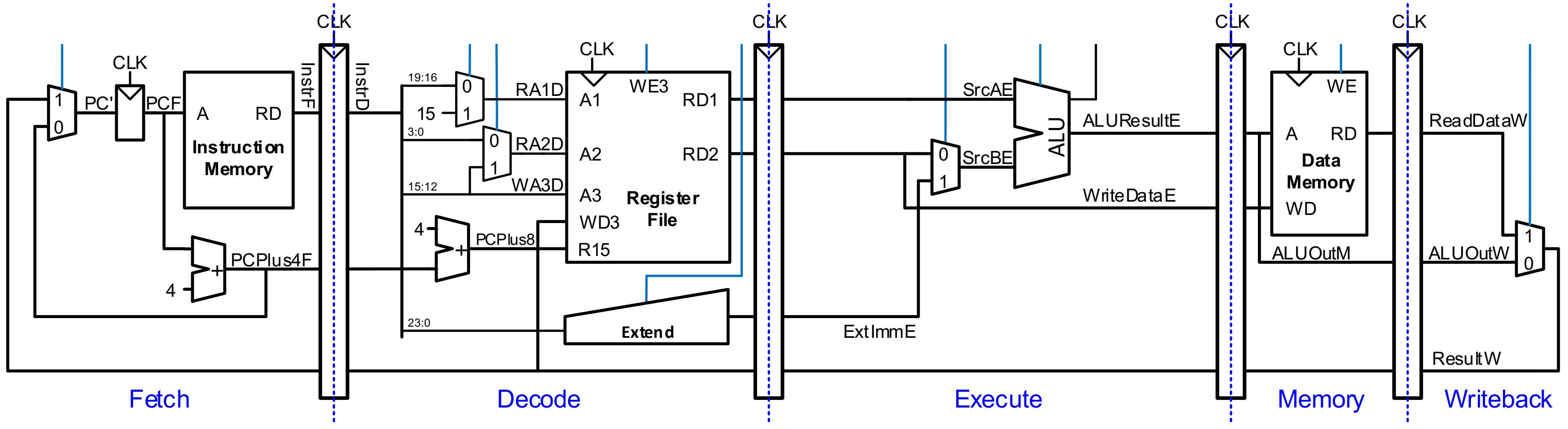
- ❑ Is the pipeline fully utilized? YES

Pipeline Operation: Cycle 7



❑ Is the pipeline fully utilized? NO

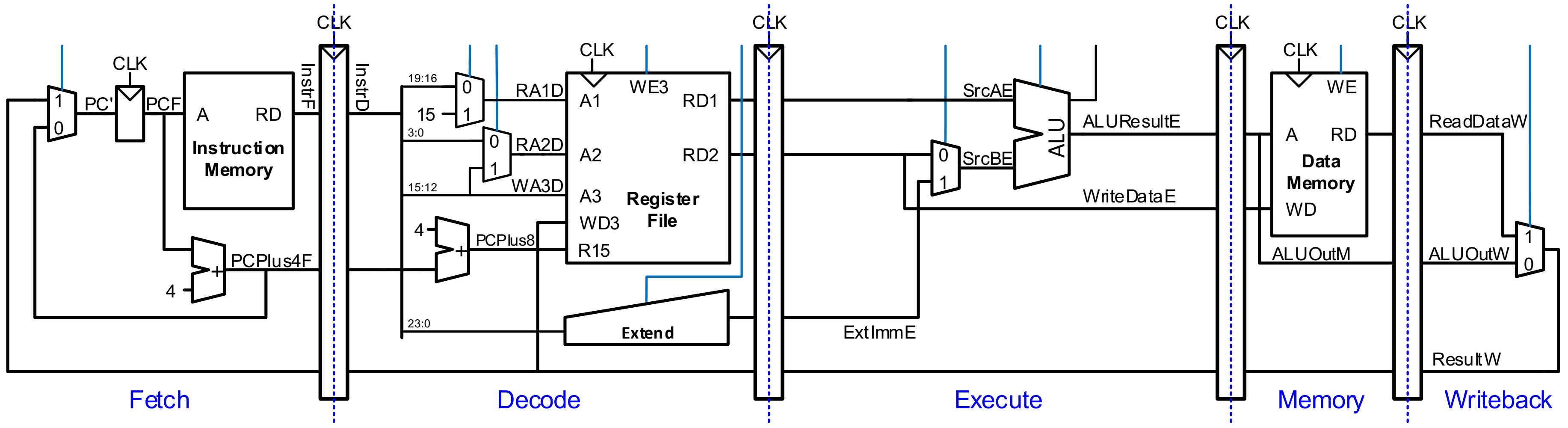
Pipeline Operation: Cycle 8



I6 I5 I4

- Is the pipeline fully utilized? NO

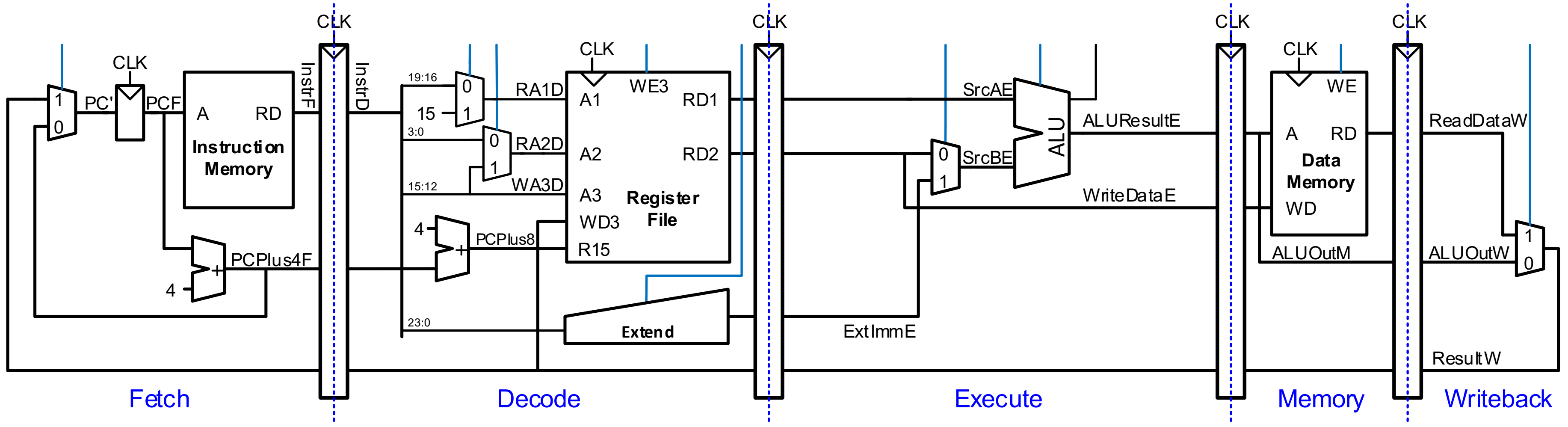
Pipeline Operation: Cycle 9



I6 I5

- Is the pipeline fully utilized? NO

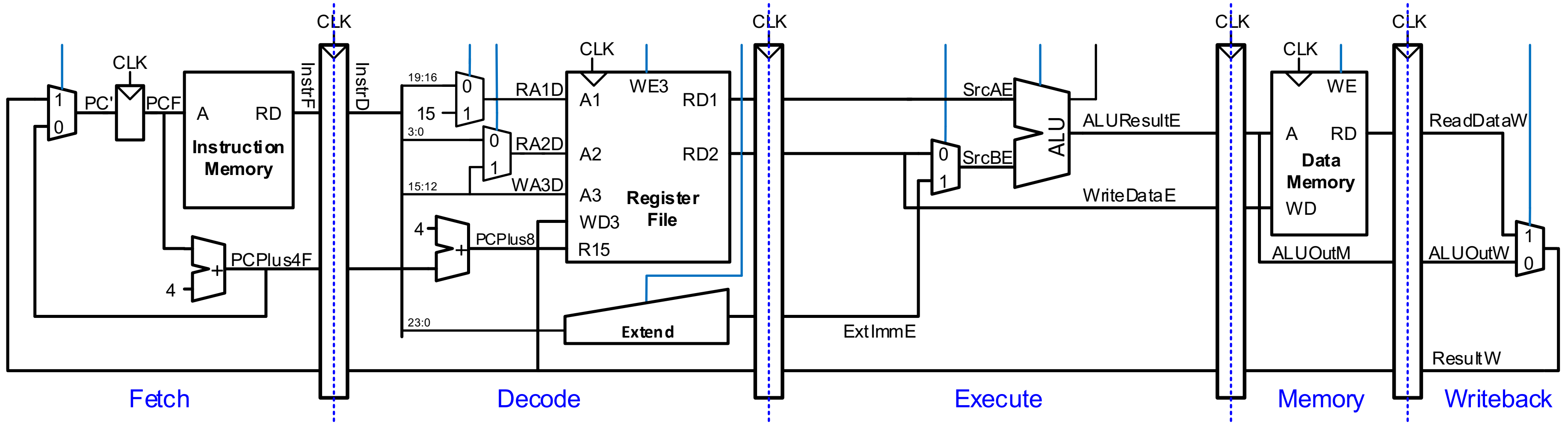
Pipeline Operation: Cycle 10



I6

- ❑ Is the pipeline fully utilized? NO

Pipeline Operation



❑ No more instructions to execute

Performance Analysis

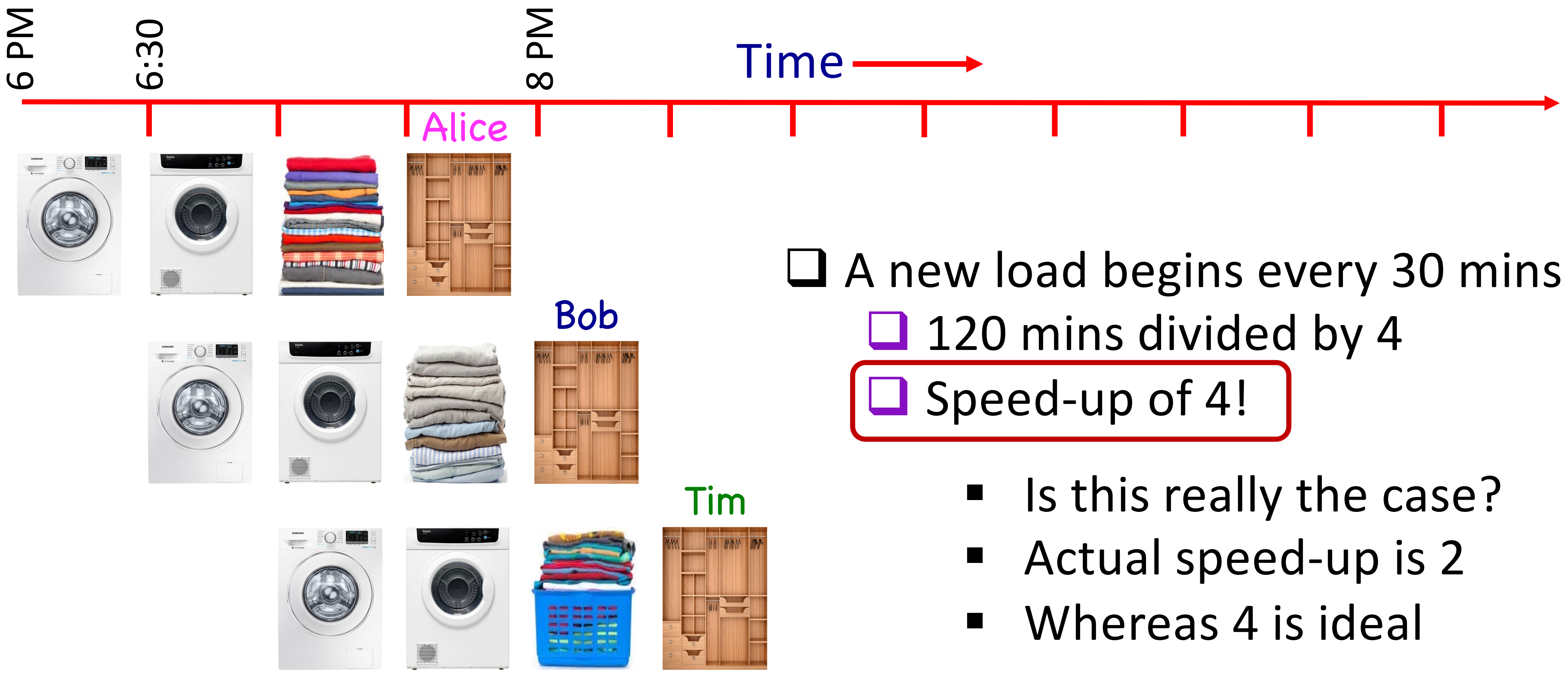
- The **6 instructions** took **10 cycles** to finish execution

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

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

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

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

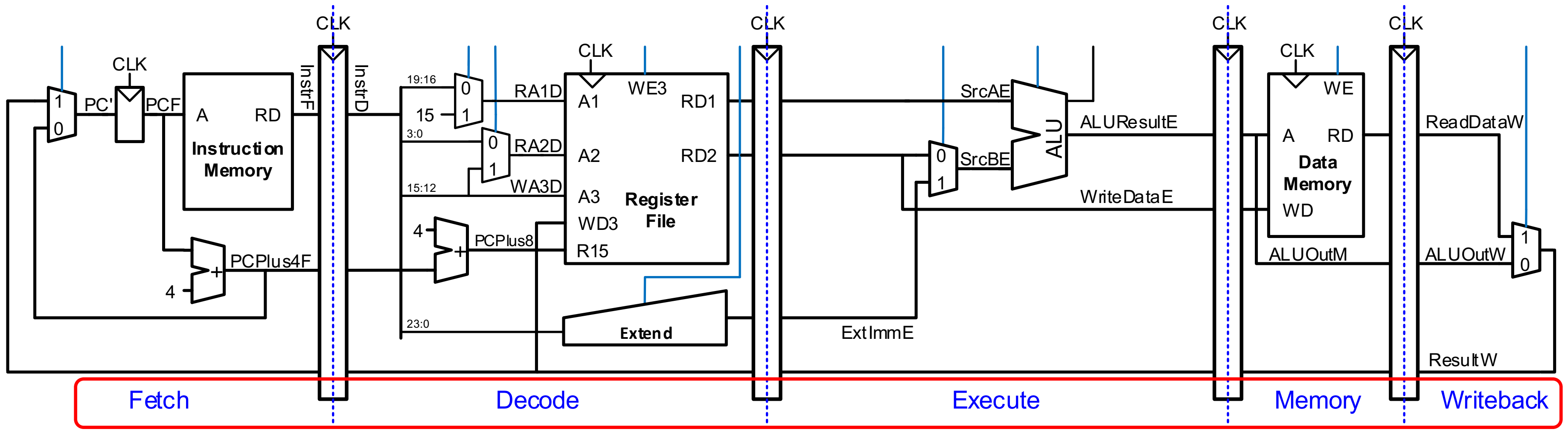
- The **6 instructions** took **10 cycles** to finish execution

- 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

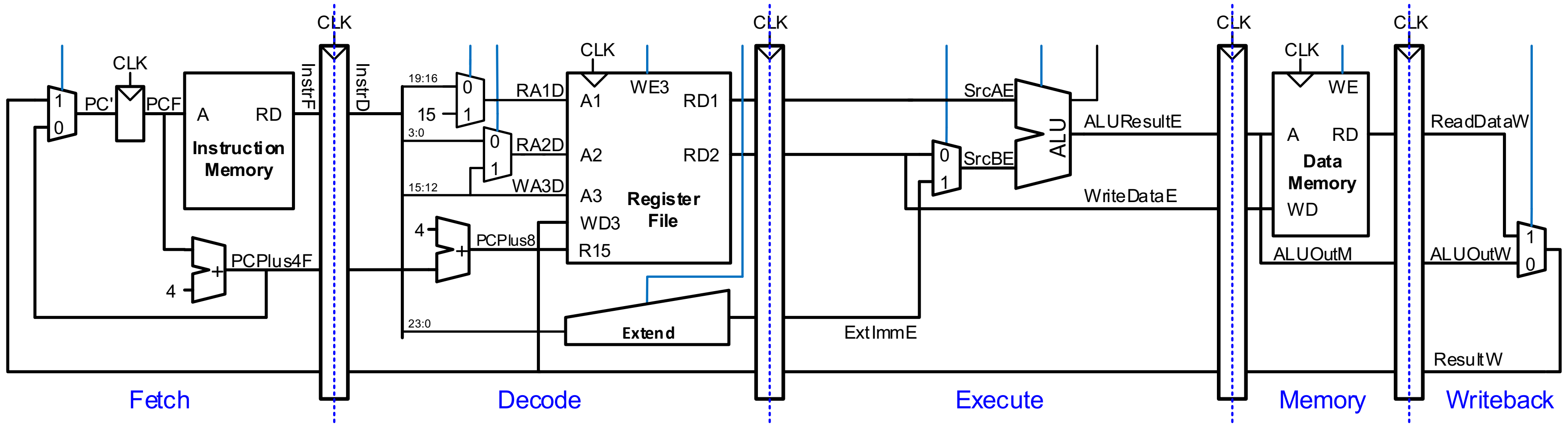
Pipelined Data



- **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?**

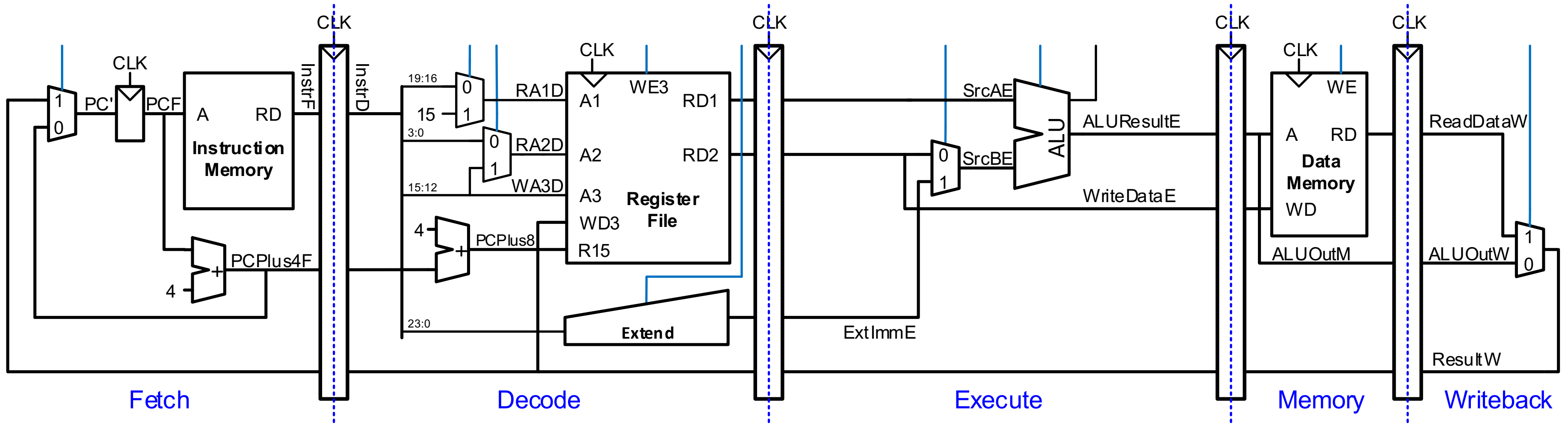
Bug in Pipelined Hardware!



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

Bug in Pipelined Hardware!

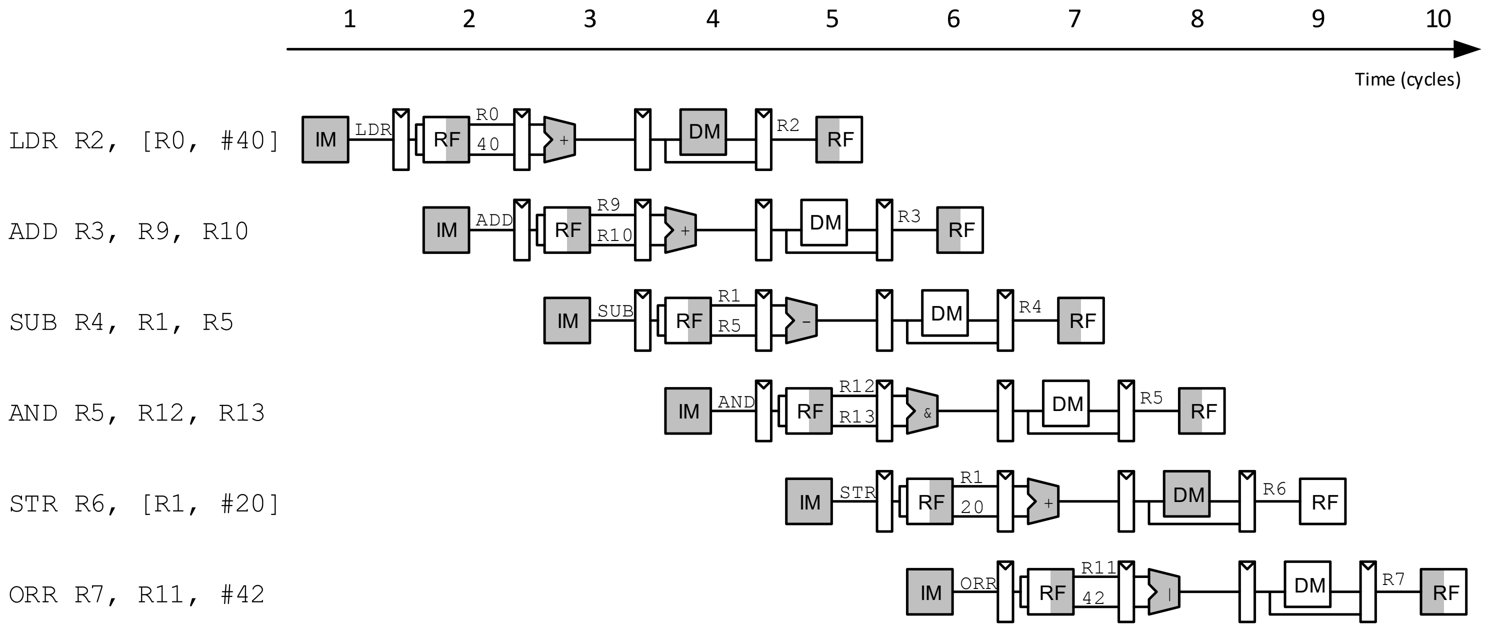
- The error is in the register file write logic that operates in the writeback stage



- The data value comes from ResultW, a Writeback stage signal
- But the write address comes from $\text{InstrD}_{15:12}$ (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!

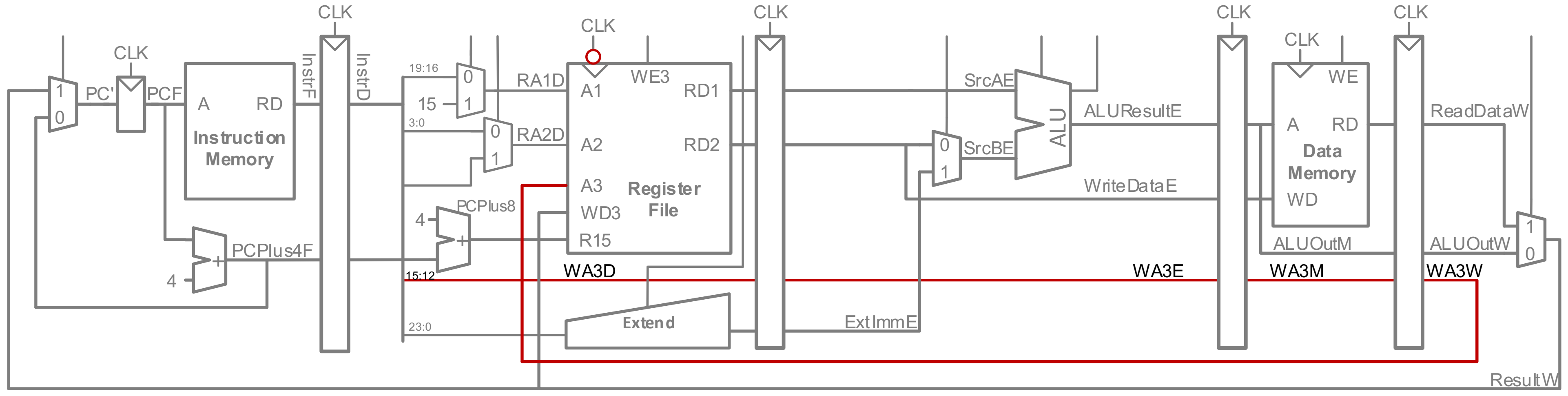
- Without correction, during cycle 5, the result of the `LDR` instruction would be incorrectly written to `R5` instead of `R2`



1 2 3 4 5 6 7 8 9 10
Time (cycles)
LDR R2, [R0, #40]
IM LDR RF R0 40 + DM R2 RF
ADD R3, R9, R10
IM ADD RF R9 R10 + DM R3 RF
SUB R4, R1, R5
IM SUB RF R1 R5 - DM R4 RF
AND R5, R12, R13
IM AND RF R12 R13 & DM R5 RF
STR R6, [R1, #20]
IM STR RF R1 20 + DM R6 RF
ORR R7, R11, #42
IM ORR RF R11 42 | DM R7 RF

Corrected Pipelined Datapath

- Here is the corrected pipelined datapath

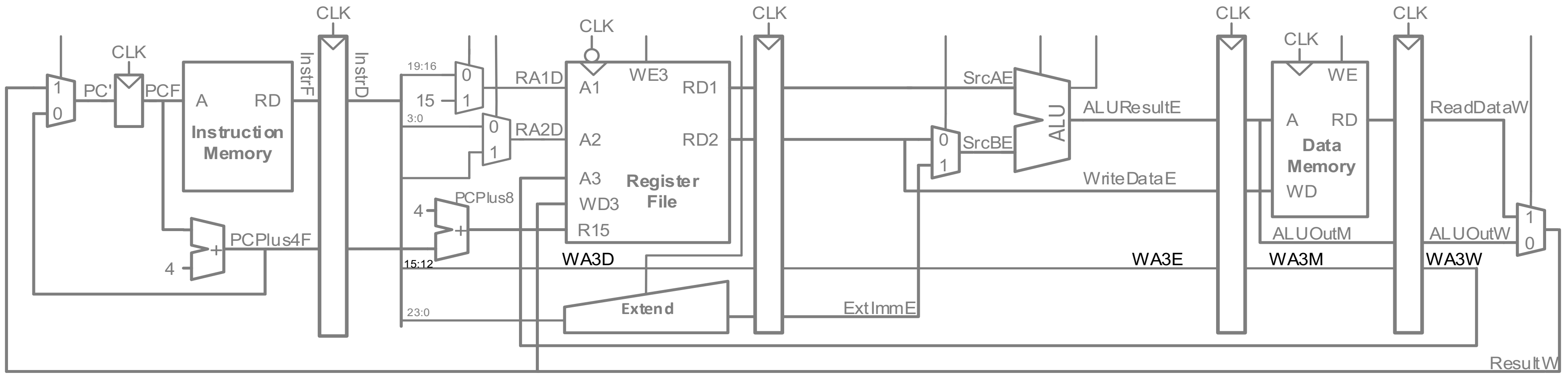


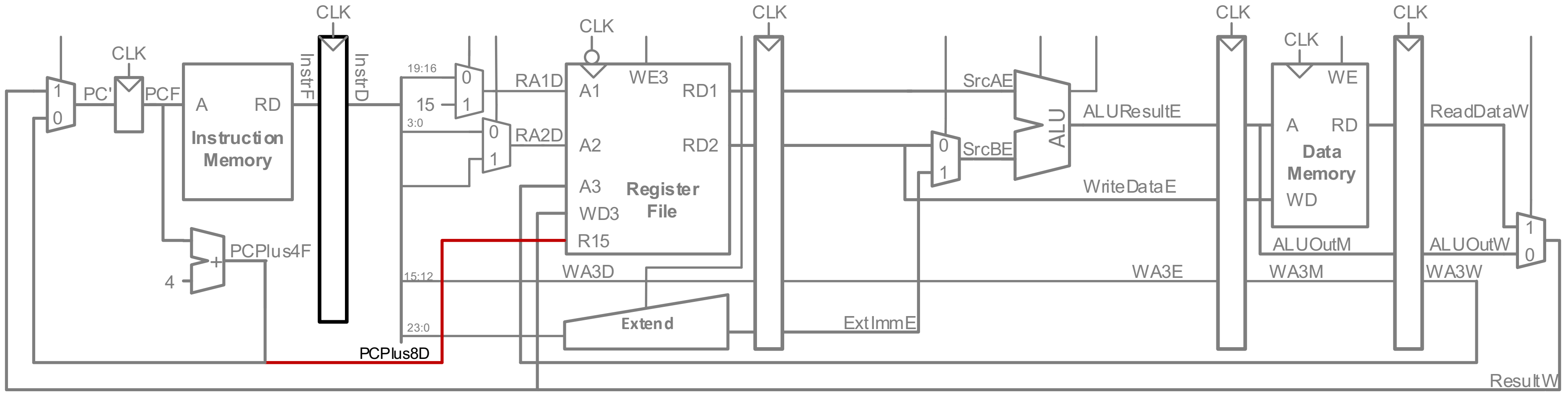
- The WA3 signal is now pipelined along through the Execution, Memory, and Writeback stages so it remains sync with the rest of the instruction

- 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

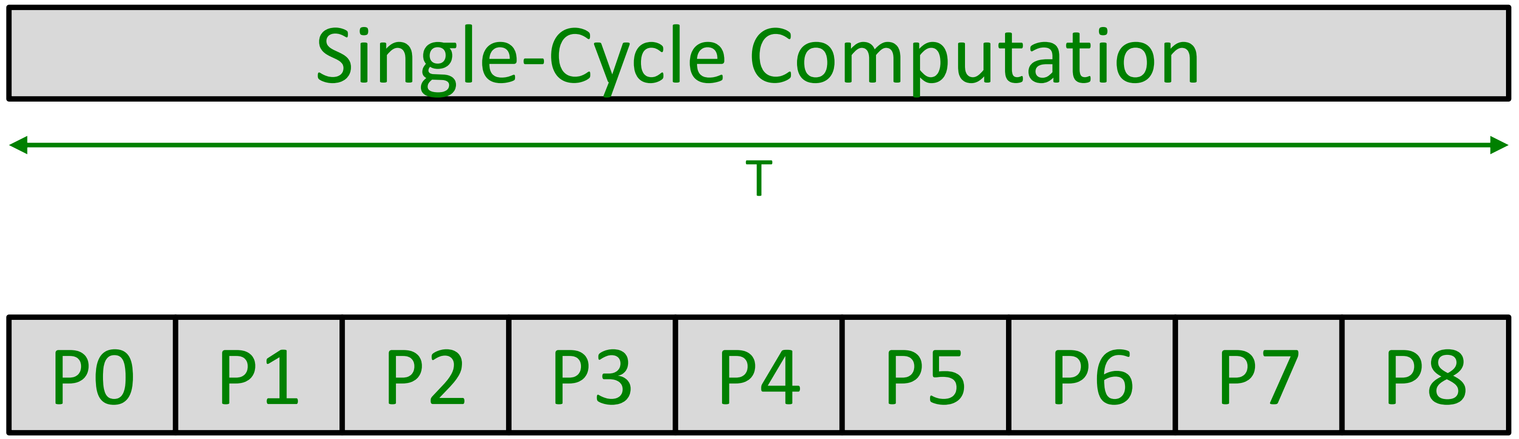




Balanced Pipeline (1)

- Let's revisit another factor that hinders ideal pipeline speedup
- Ideally, we want the computation to be pipelined, 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
- Balancing pipelines is challenging
  - Is the ARM pipelined microarchitecture balanced?
  - What can we do to make it more balanced?

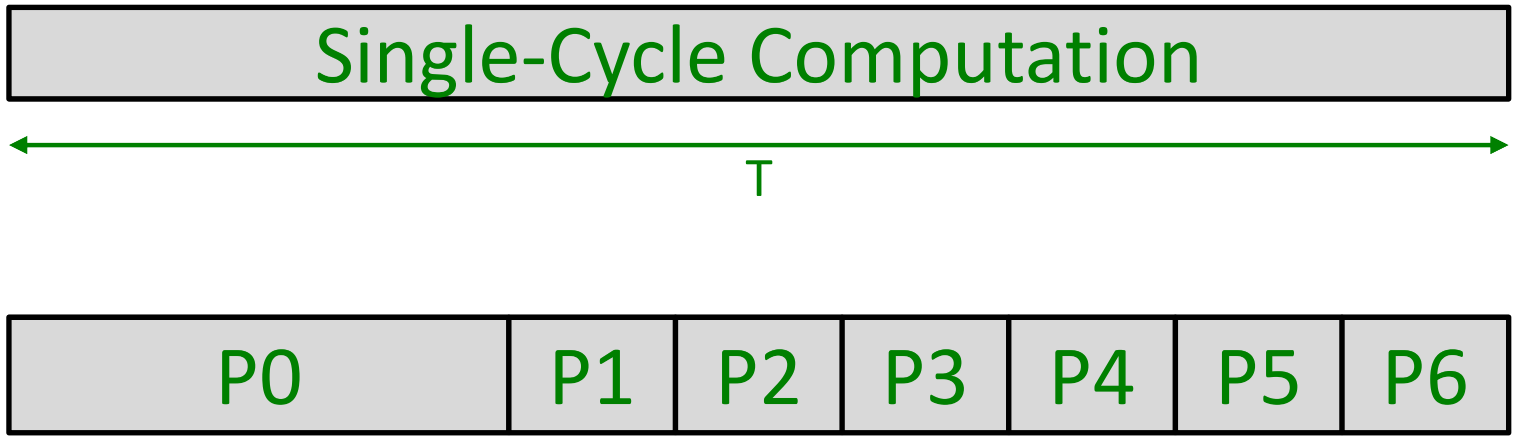
Balanced Pipeline (Example 1)



- 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

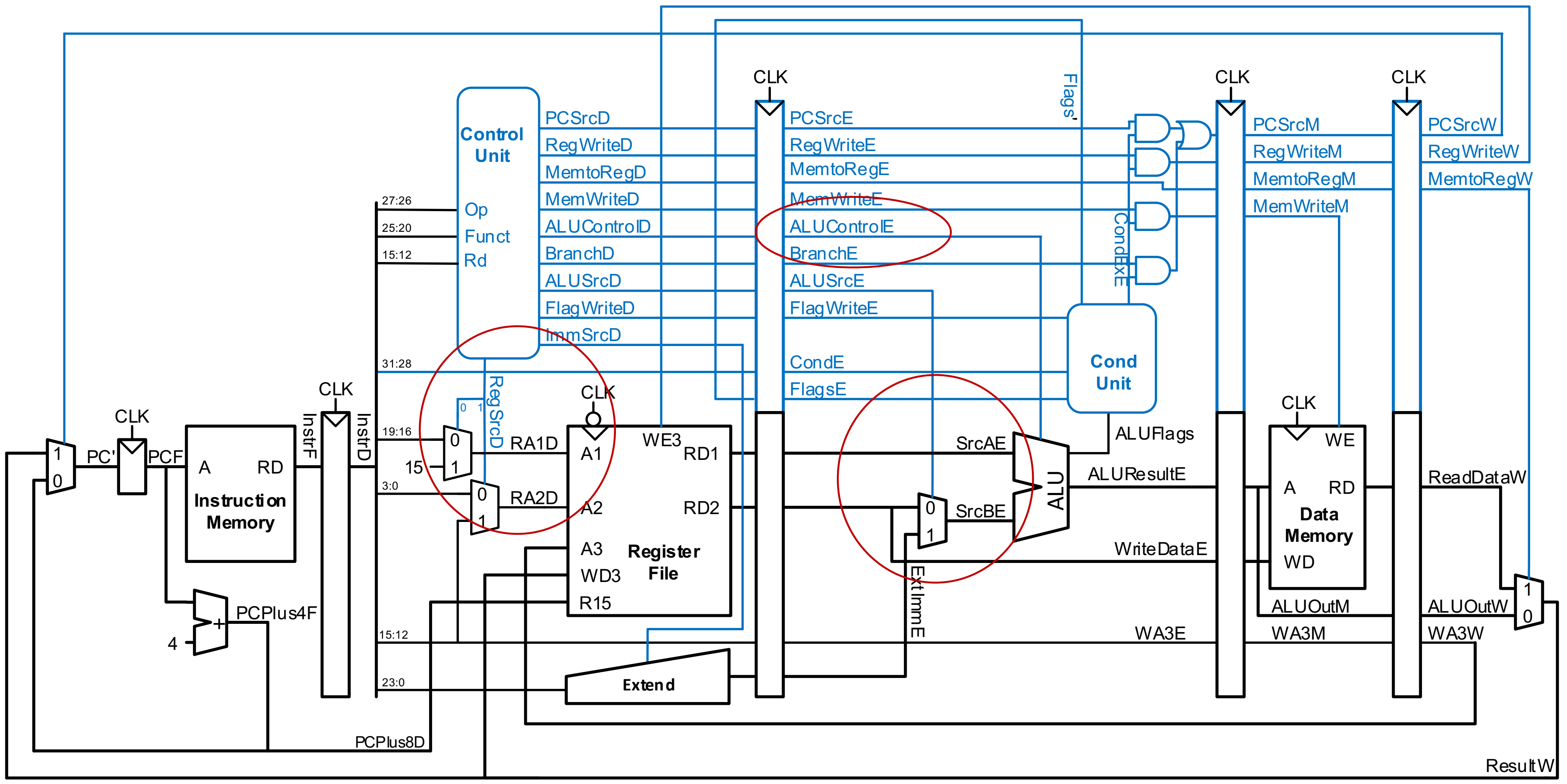
- These control signals must be pipelined along with the data

- **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

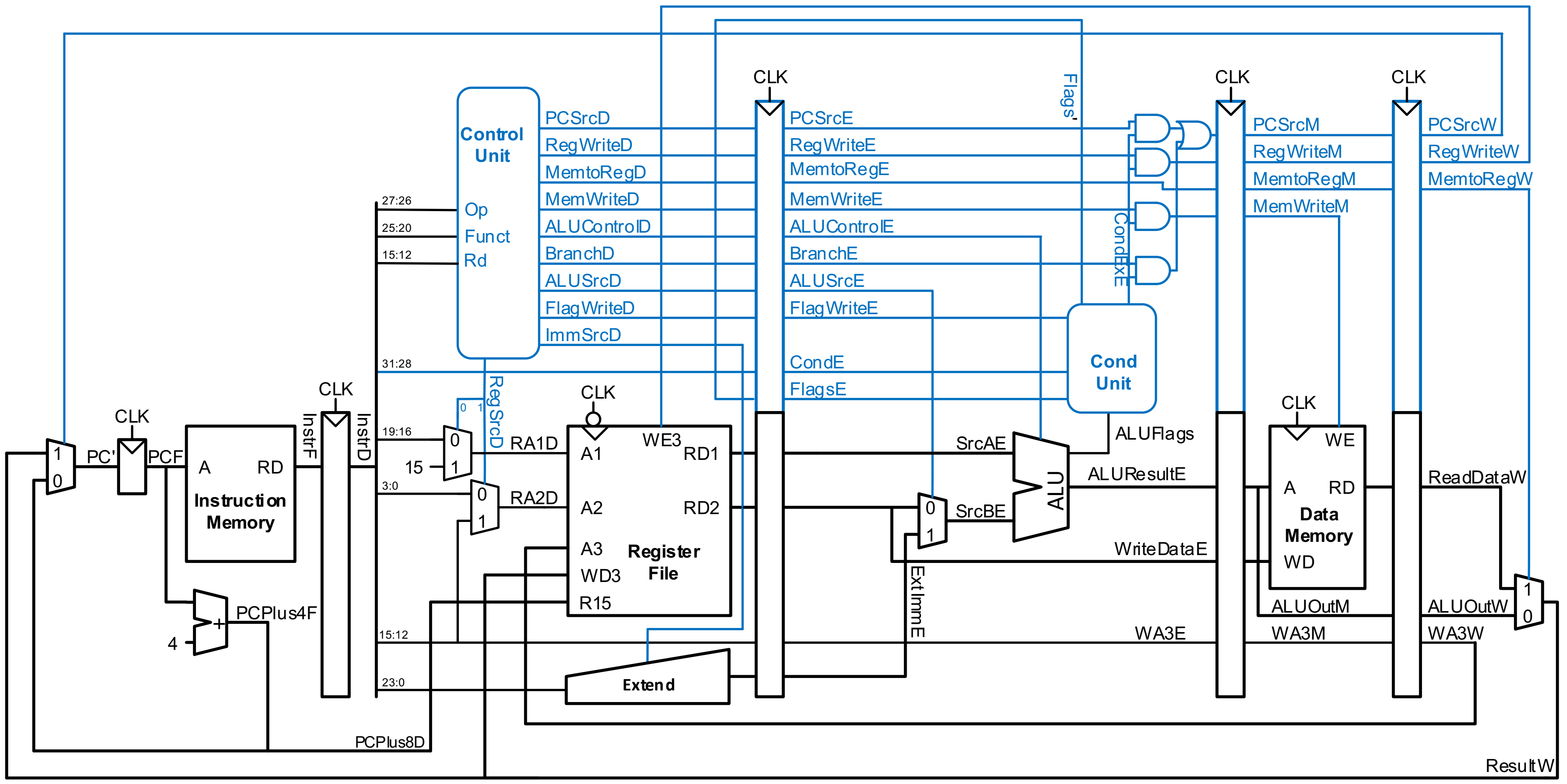
- From the processor's perspective:
  - Different solutions with different tradeoffs
    - Architectural state requires "serious" repair
    - Architectural state is **untouched**, and hazard avoided
    - Dedicated logic may be needed for hazard avoidance
    - Defensive mindset: **stall the CPU** until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

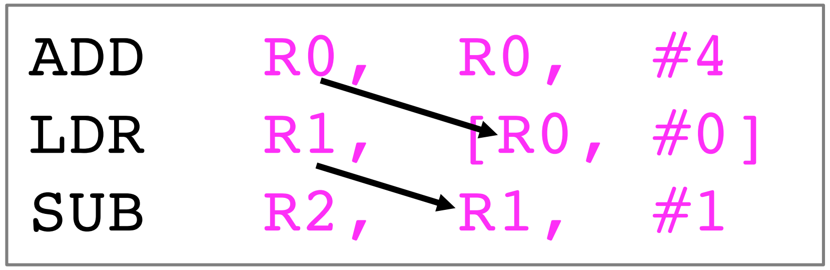
Pipeline Hazards (Another View)

- 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**



```
ADD    R0,   R0,   #4
LDR    R1,  [R0,  #0]
SUB    R2,   R1,   #1
```

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

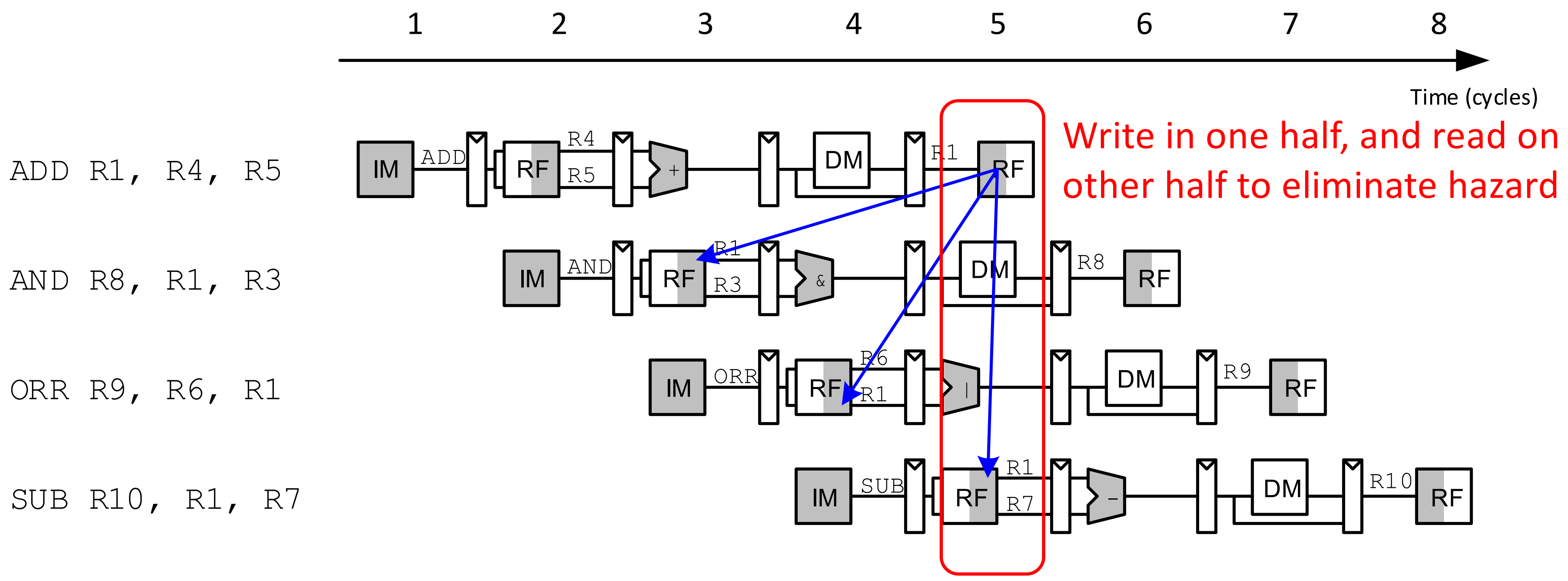
- **Think:** These hazards are not possible in a single-cycle microarchitecture

- **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



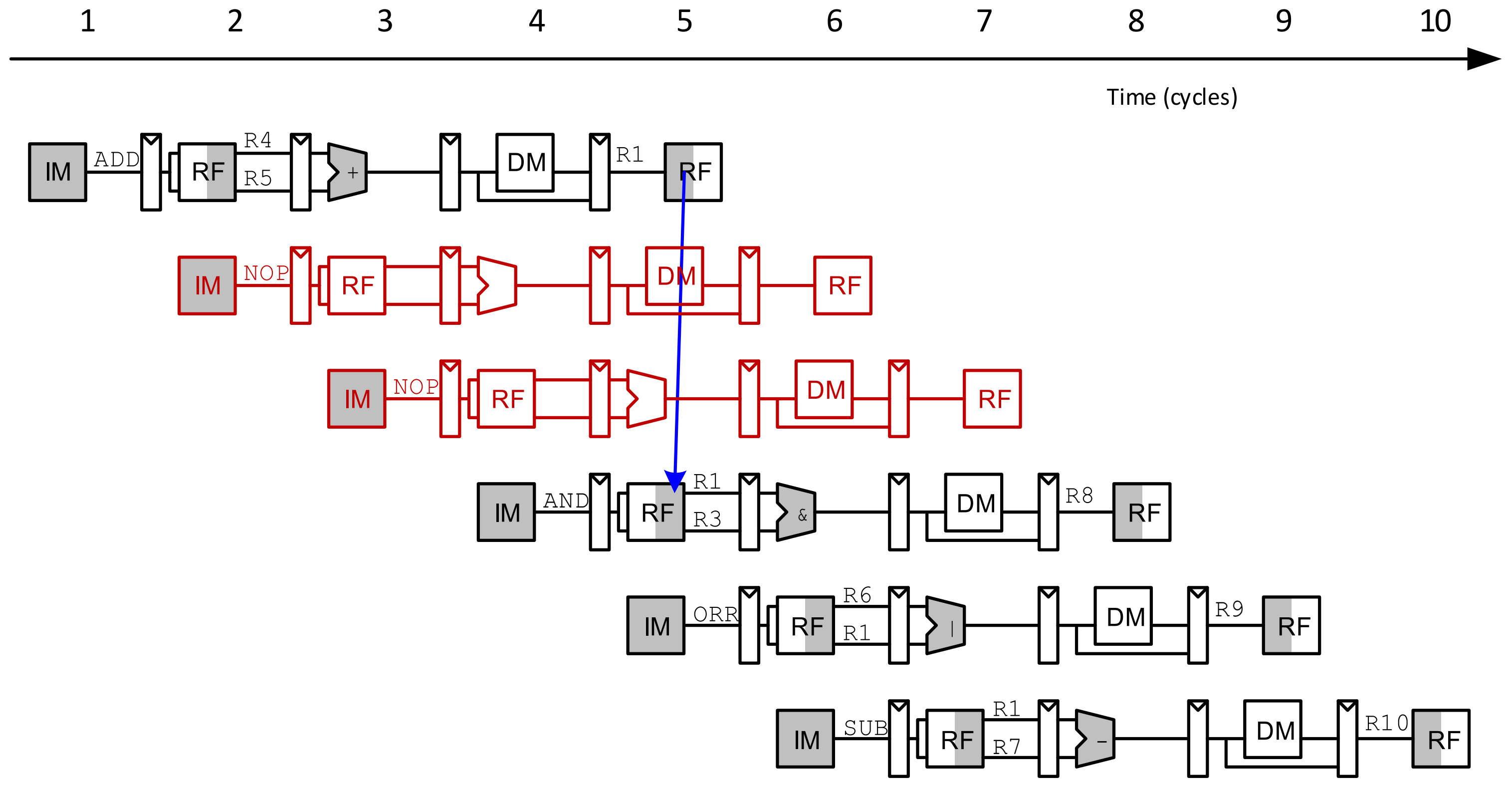
- Use a clever register read/write policy to eliminate one hazard
  - What can we do about the remaining two hazards?

Solution # 1: Software Interlocking

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

- Rearrange code at compile time
  - Move dependent useful instructions forward

Example: Software Interlocking



ADD R1, R4, R5

NOP

NOP

AND R8, R1, R3

ORR R9, R6, R1

SUB R10, R1, R7

Solution # 1: Software Interlocking

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

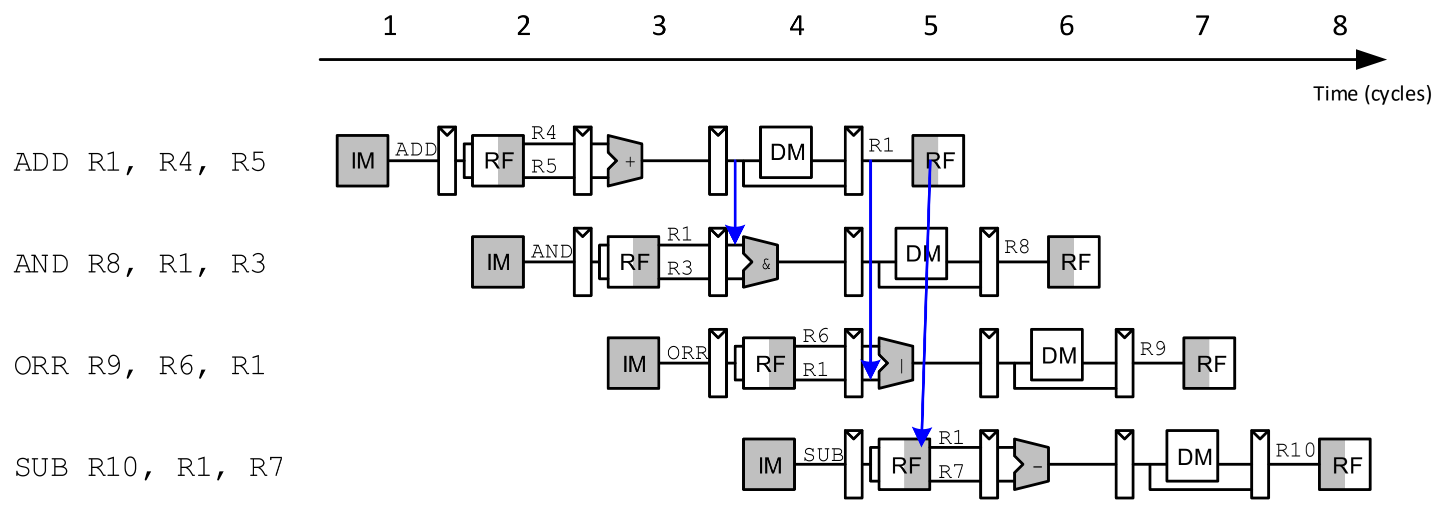
Solution # 2: Forwarding or Bypassing

- **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

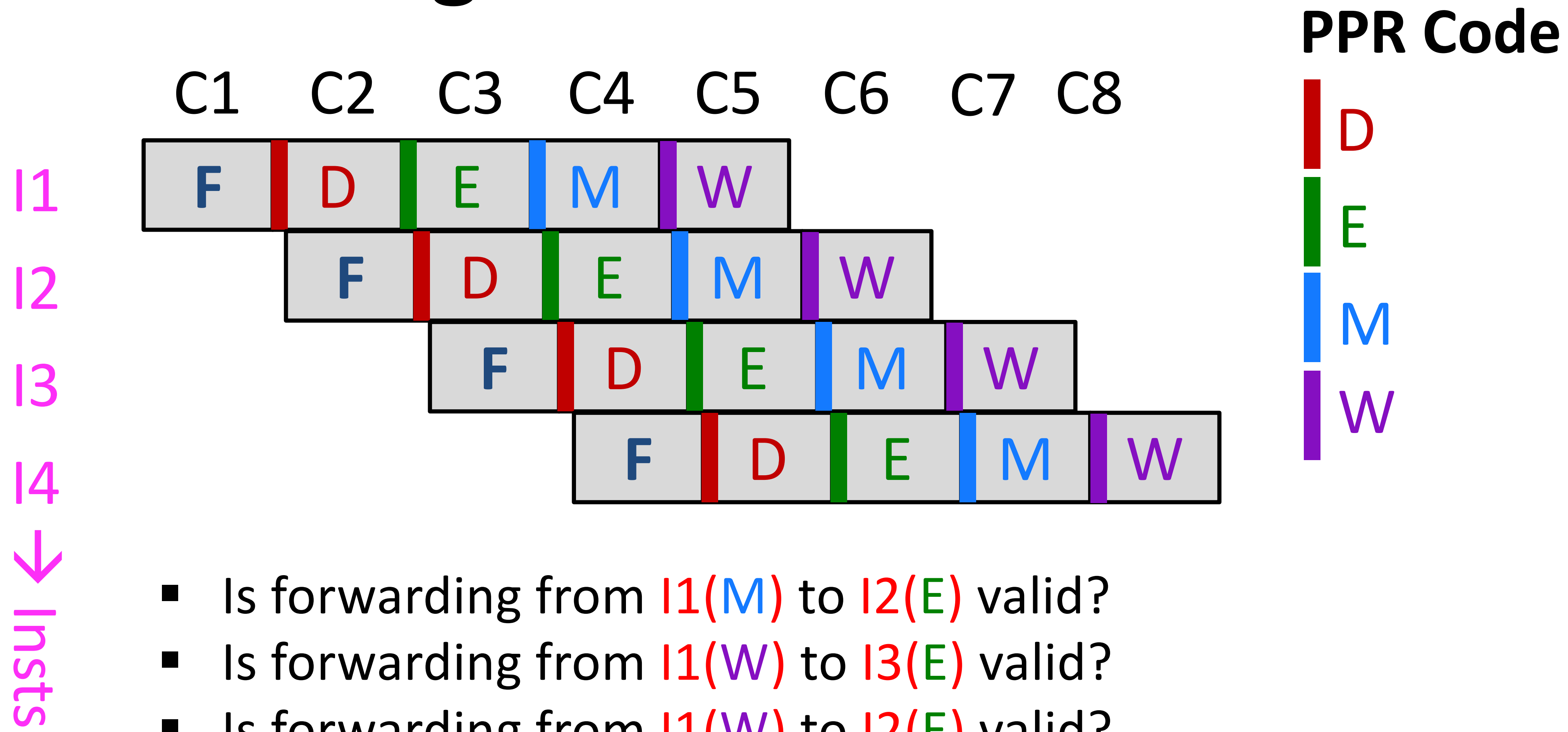
- Requires adding muxes in front of the ALU to select the operand from one of the many sources
  - (1) RF, (2) Memory PPR, (3) Writeback PPR

Why Forwarding Works?



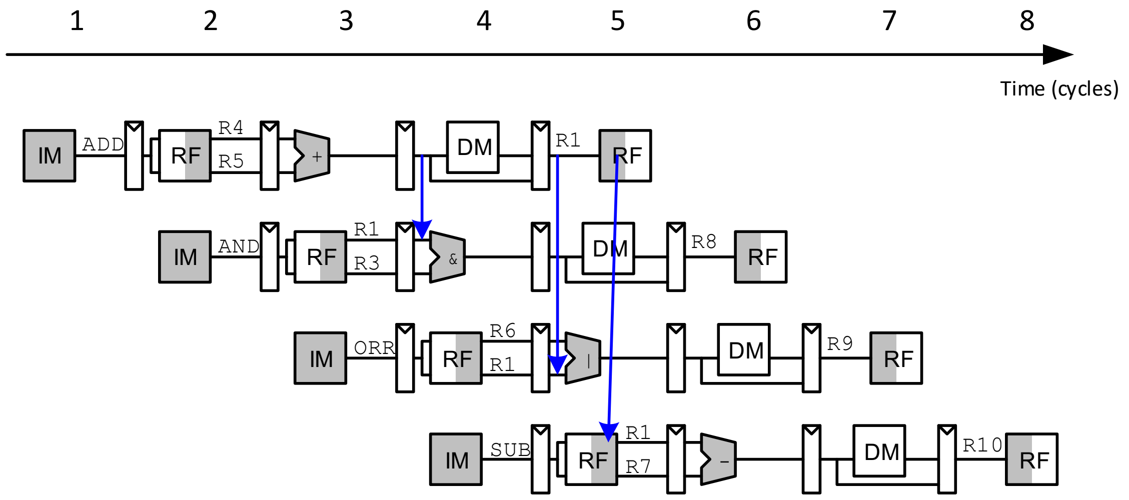
- Sum from the `ADD` instruction is computed by **ALU** in **cycle 3** and is needed by the `AND` instruction in **cycle 4**
- No need to wait for the results to appear in register file

Forwarding Exercise



- 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



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

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)

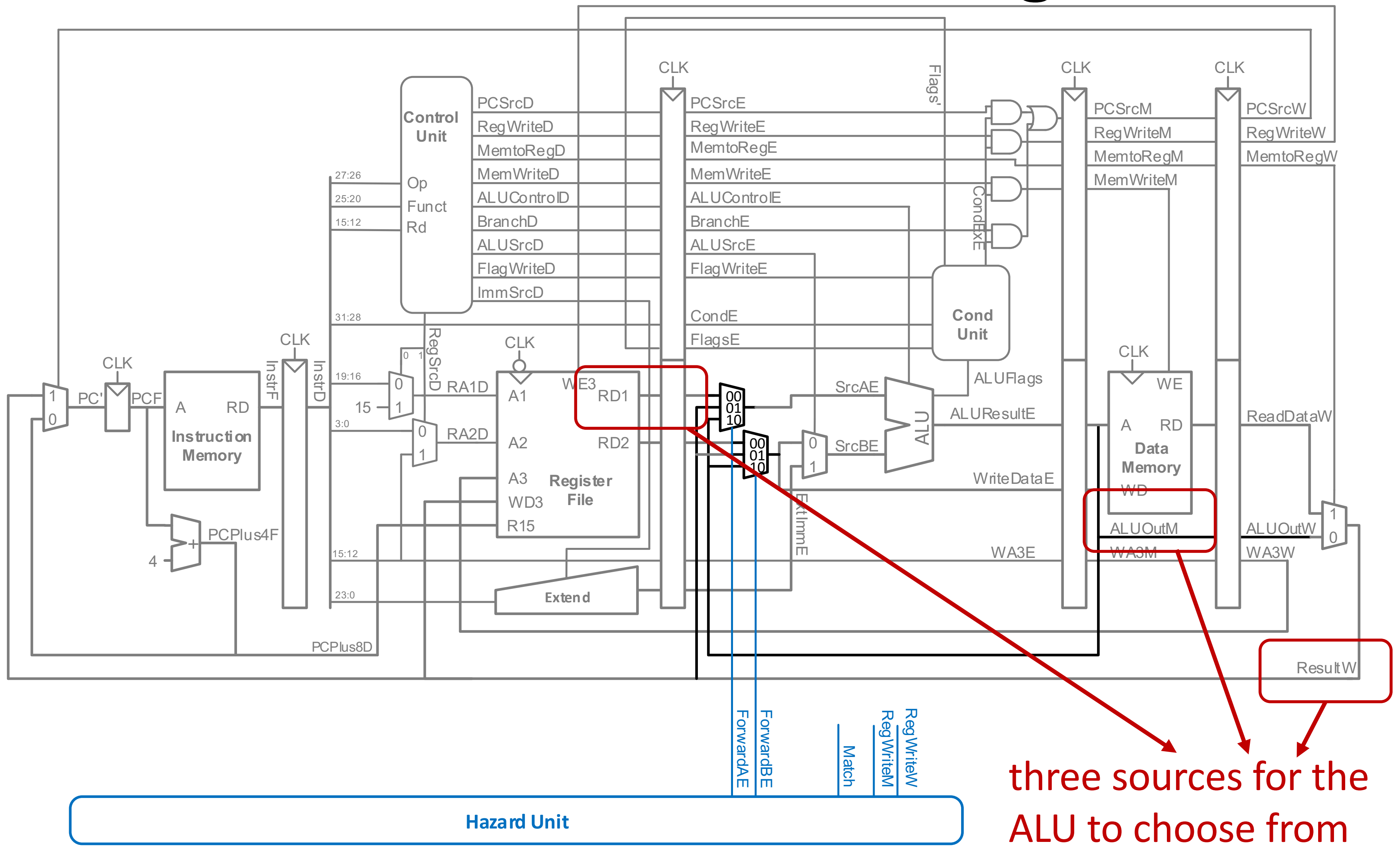
- **Execute** stage register matches **Writeback** stage register?
  Match_1E_W = (RA1E == WA3W)
  Match_2E_W = (RA2E == WA3W)

- 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



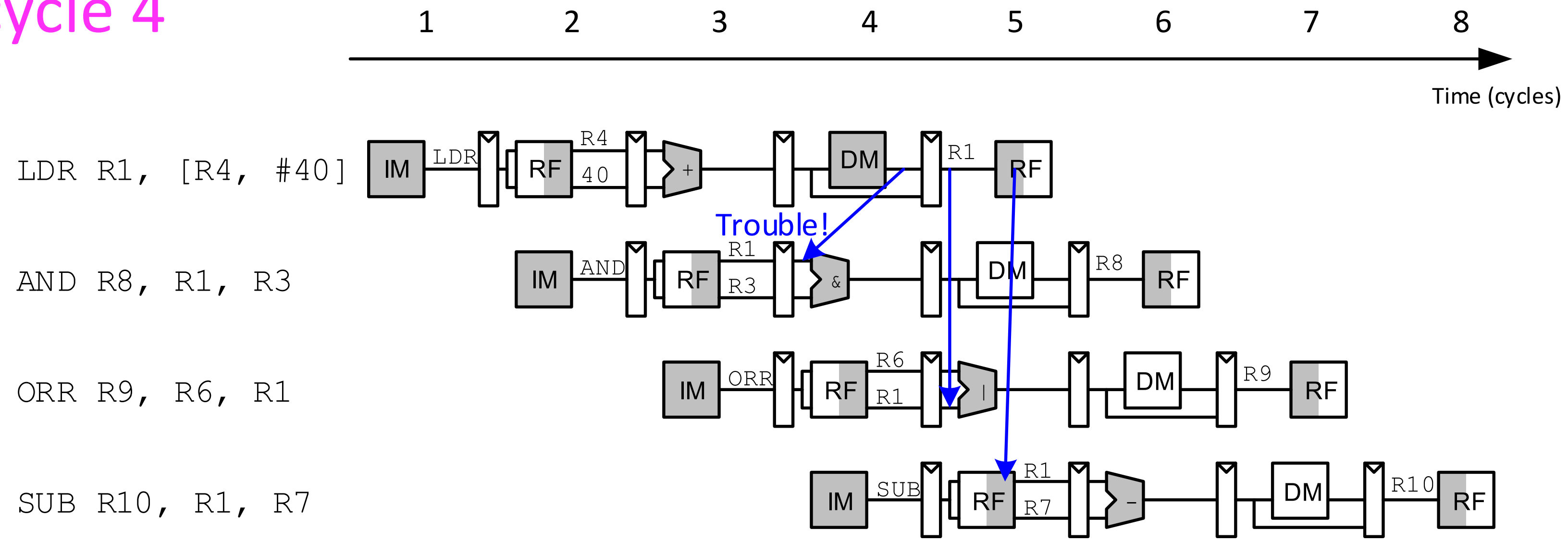
three sources for the ALU to choose from

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

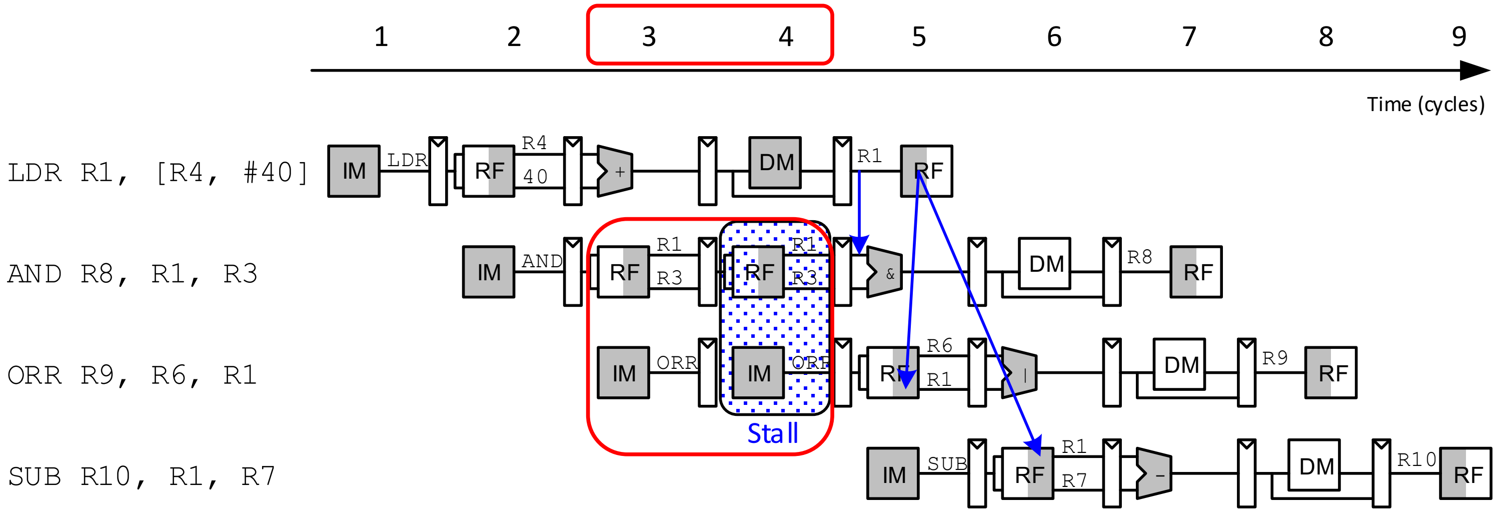
- The dependent instruction can be detected as the "user" of `LDR` after it has been decoded at the end of **Decode stage**

- **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

Stall in the Decode stage

Cycle #	Fetch	Decode	Execute	Memory	Writeback
1:	i1				
2:	i2	i1			
3:	**i3**	**i2**	i1		
4:	**i3 (Stall)**	**i2 (Stall)**	**Bubble**	i1	
5:	i4	i3	i2	**Bubble**	i1
6:	i5	i4	i3	i2	**Bubble**
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
  - 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

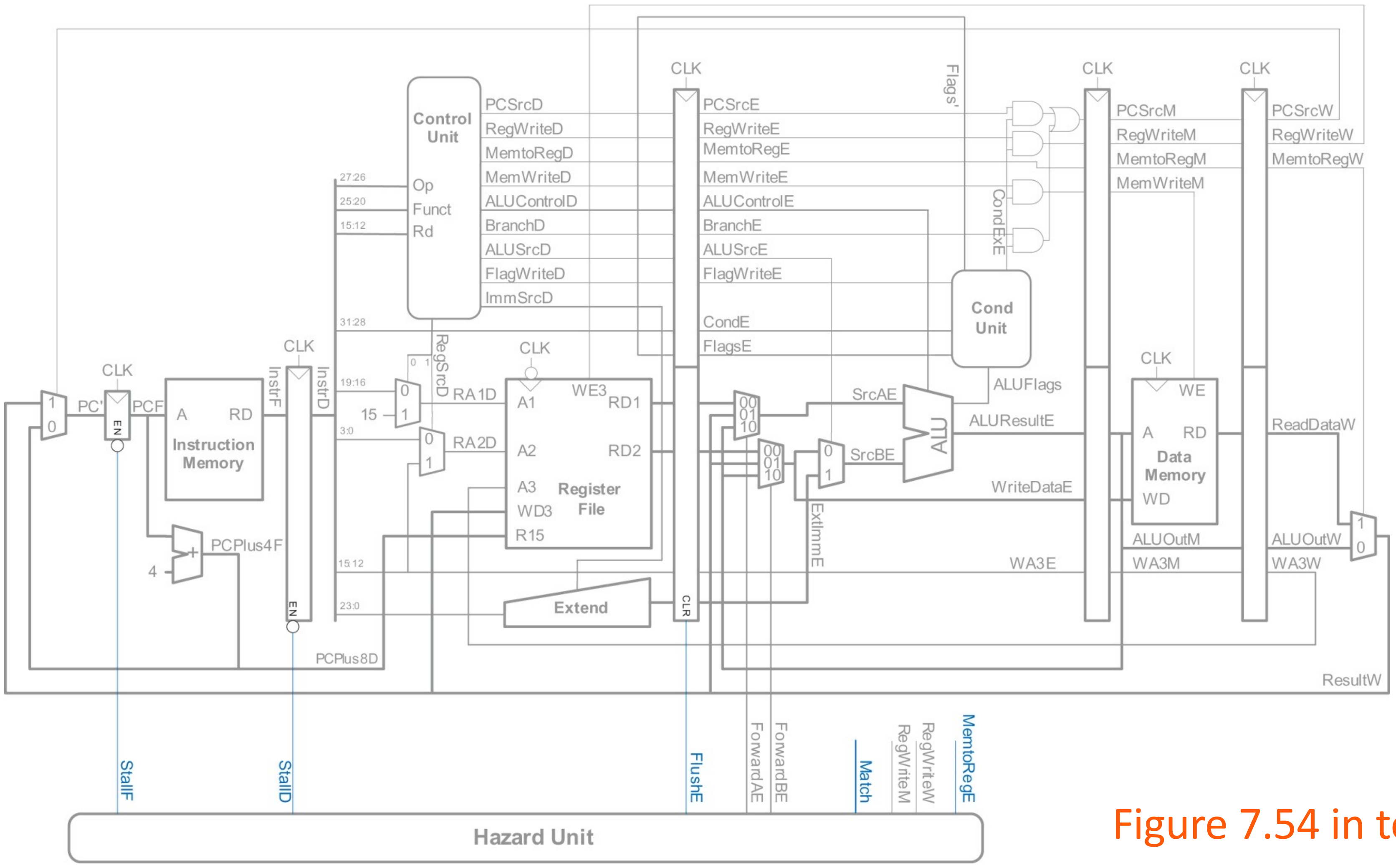


Figure 7.54 in textbook

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
  - Keep fetching the next sequential instructions

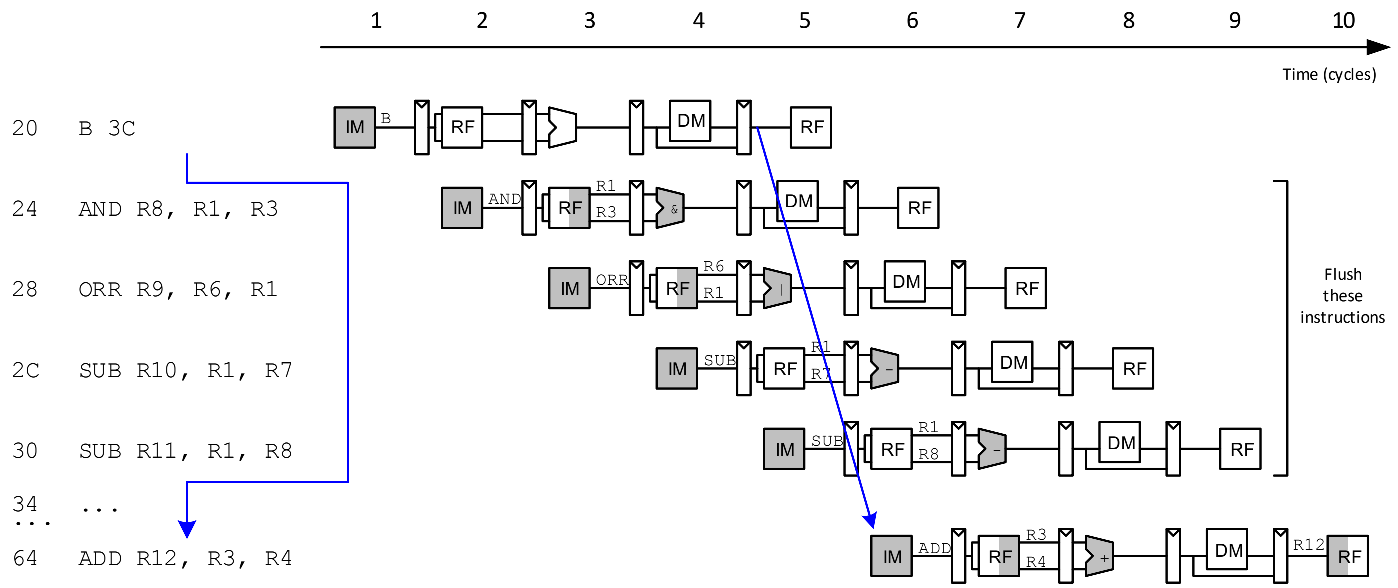
- Predict-always-taken
  - CPU **stalls** for four cycles because target address not available

- 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

Flushing when Branch is Taken

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

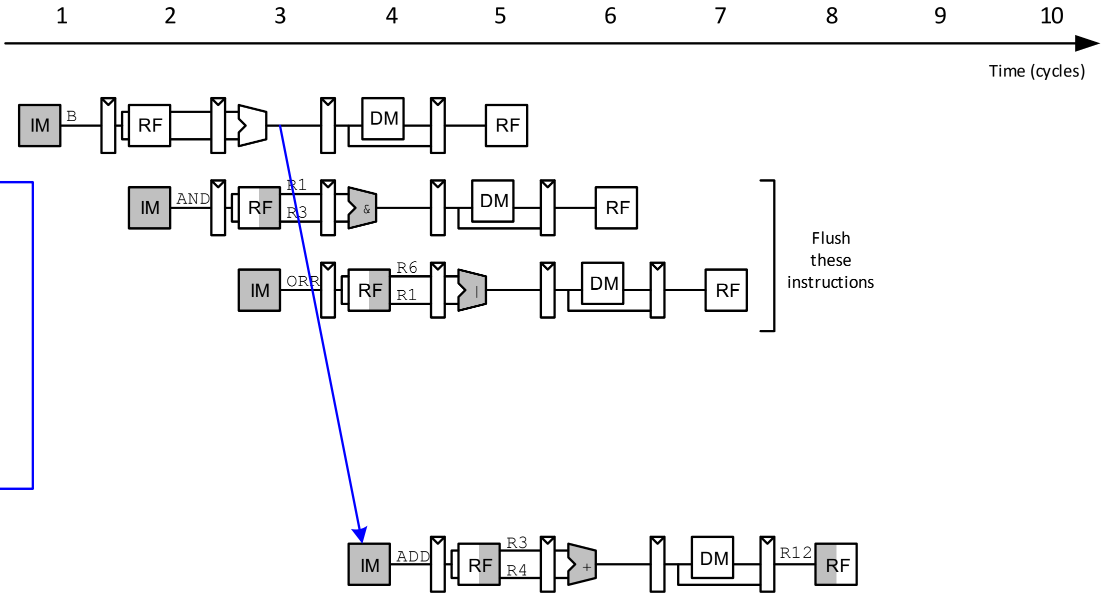


- Four instructions flushed when branch is taken
- Misprediction penalty of 4 wasted cycles for taken branches
- **Idea: Predict the branch early**

Early Branch Resolution

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

```
20   B 3C

24   AND R8, R1, R3

28   ORR R9, R6, R1

2C   SUB R10, R1, R7

30   SUB R11, R1, R8

34   ...
...
64   ADD R12, R3, R4
```



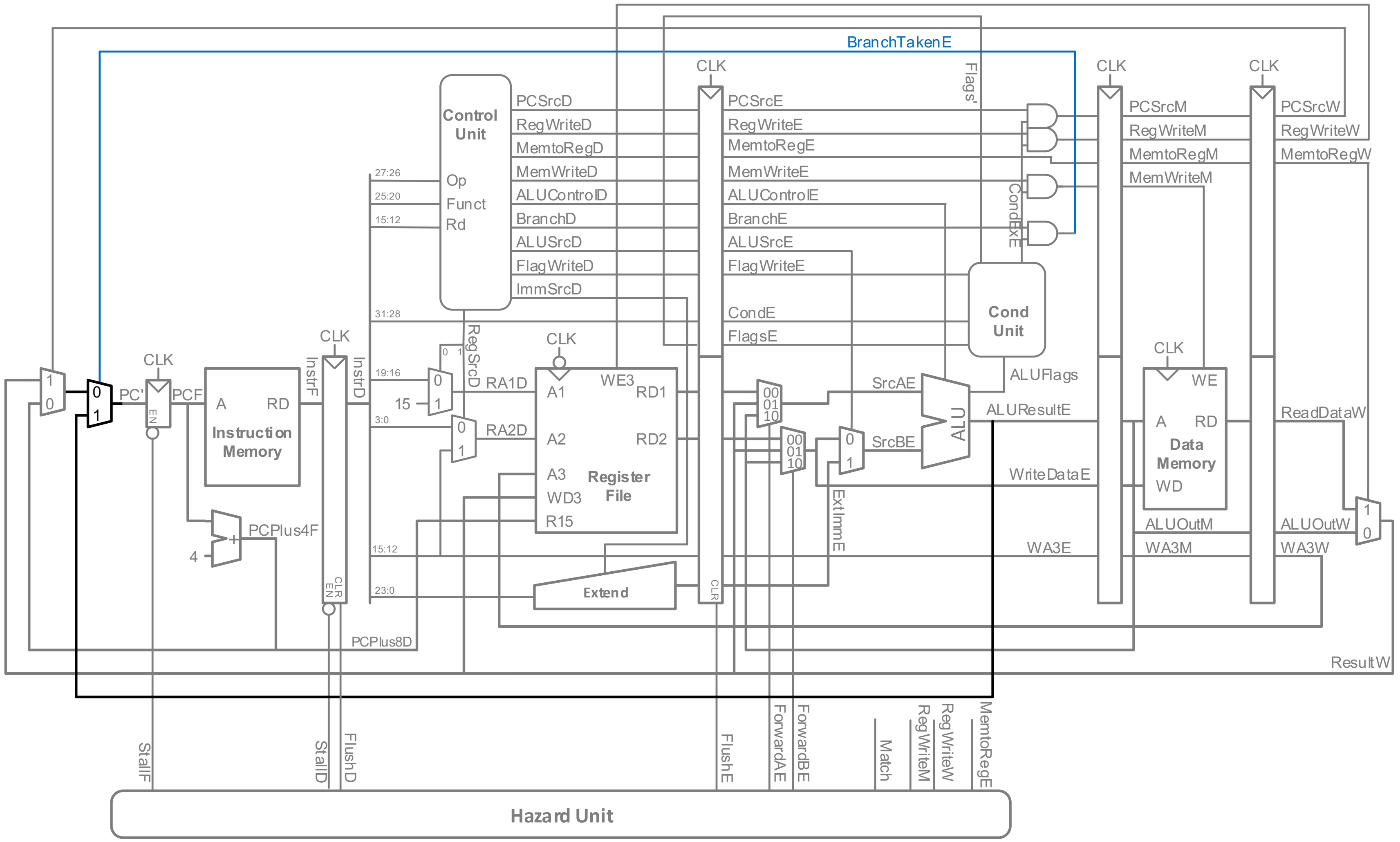
- Flush the two instructions in the **F** and **D** stages

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)

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

Flush and Stall Logic for Writes to PC

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

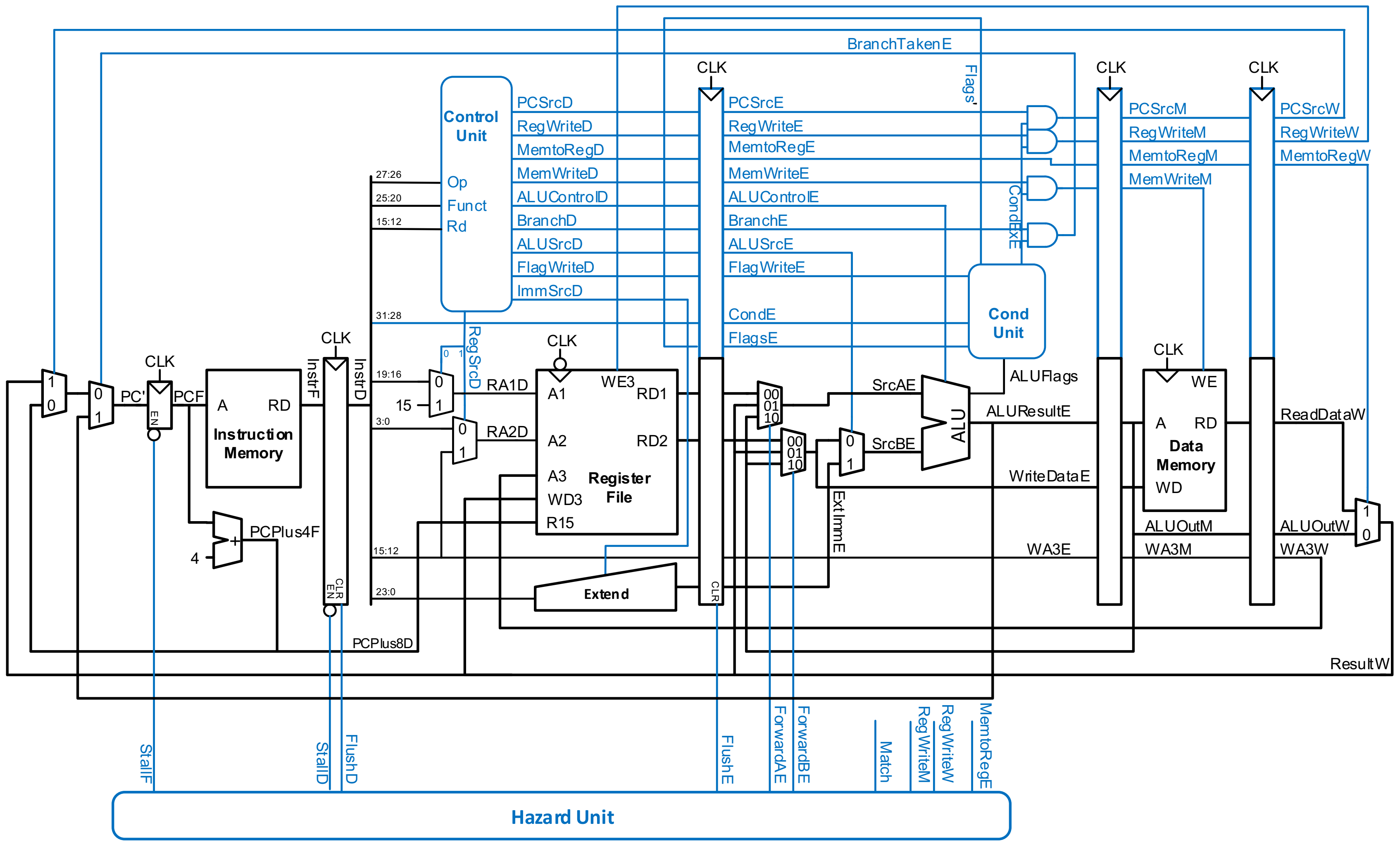
Flush D if PC write in progress in D, E, M, or W, or branch taken in E

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

  StallD = ldrStallD

Stall Decode if LDR-Use hazard

ARM Processor with Full Hazard Handling

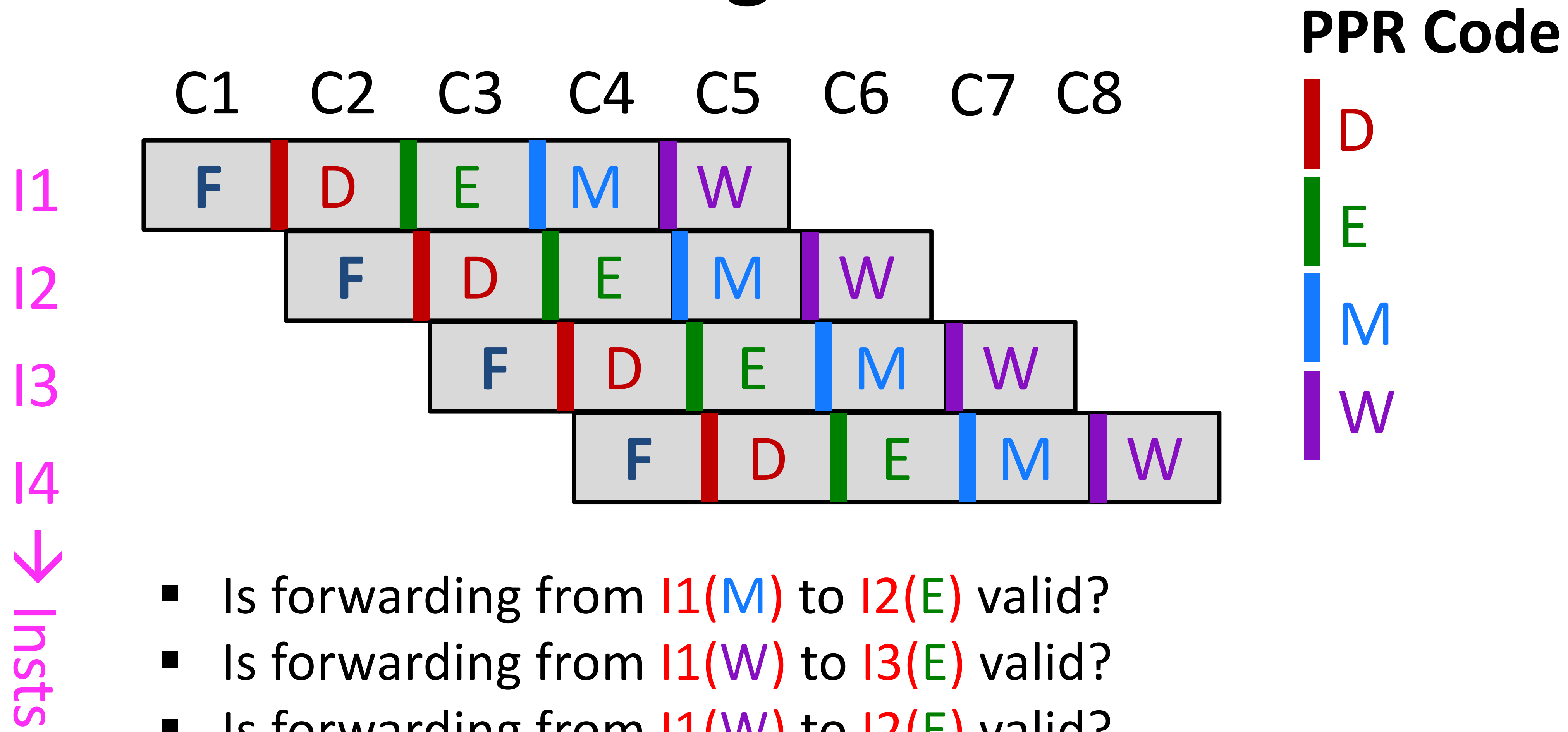


When to Forward?

- 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
```

Recall: Forwarding Exercise



- 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?

When to Stall?

- Load-use hazard
  - Stall the Decode and Fetch stages when the “use” is discovered

- PC-changing instructions
  - 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?

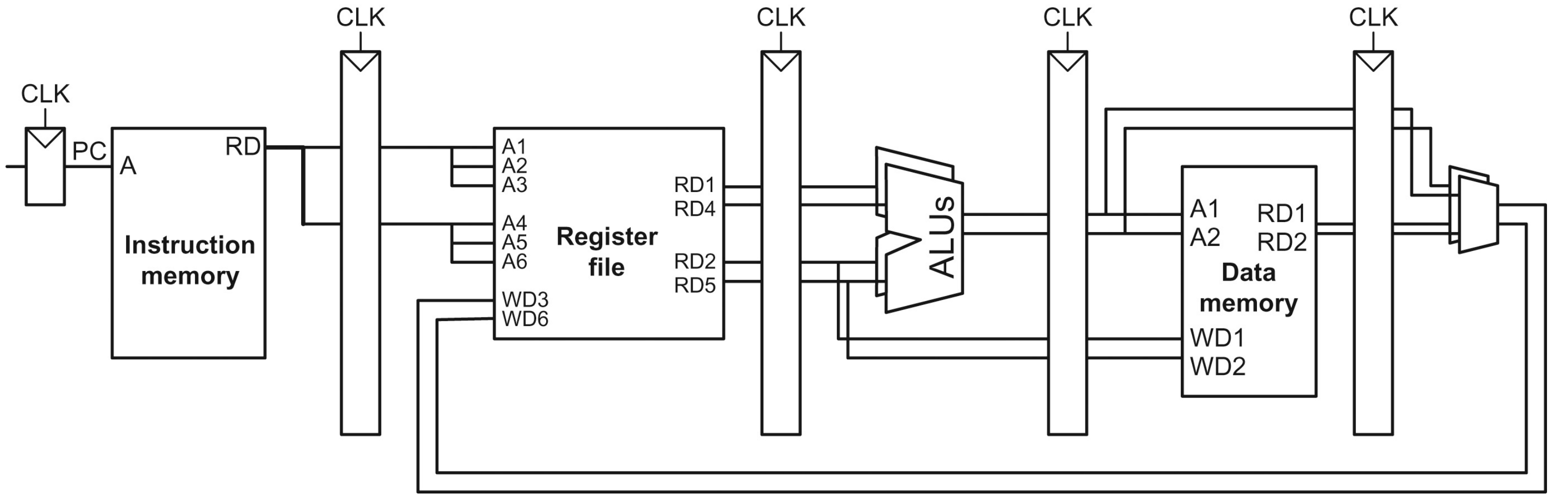
- Stall
  - Use Enable input to hold/retain the value stored in the pipeline register

- 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

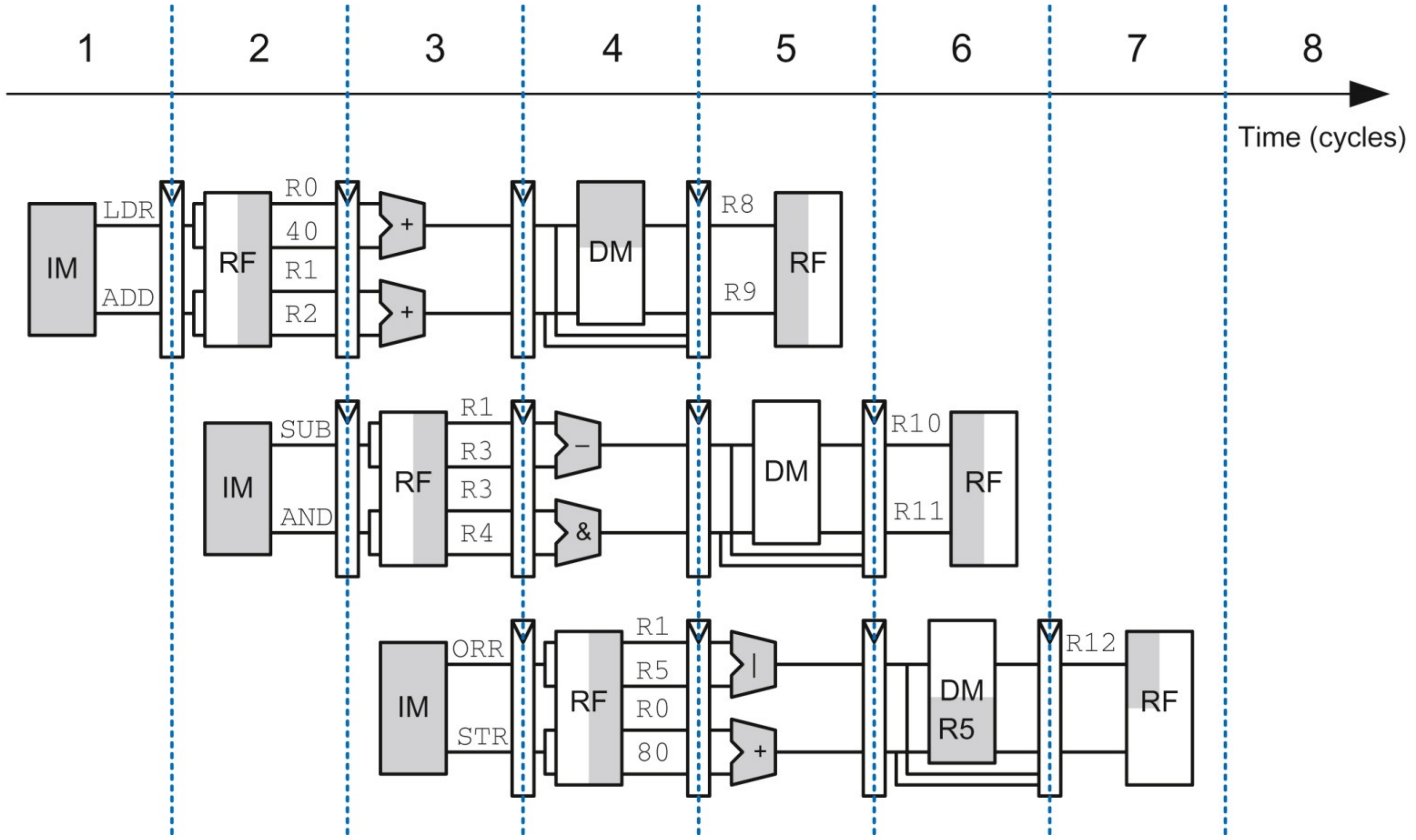


- Requires 6-ported register file (4 reads, 2 writes), 2 ALUs, 2-ported data memory
- Ideal CPI = 0.5 and IPC = 2
- Dependencies and hazards inhibit ideal IPC
- Above figure does not show forwarding and hazard detection logic

Superscalar: Pipeline Operation

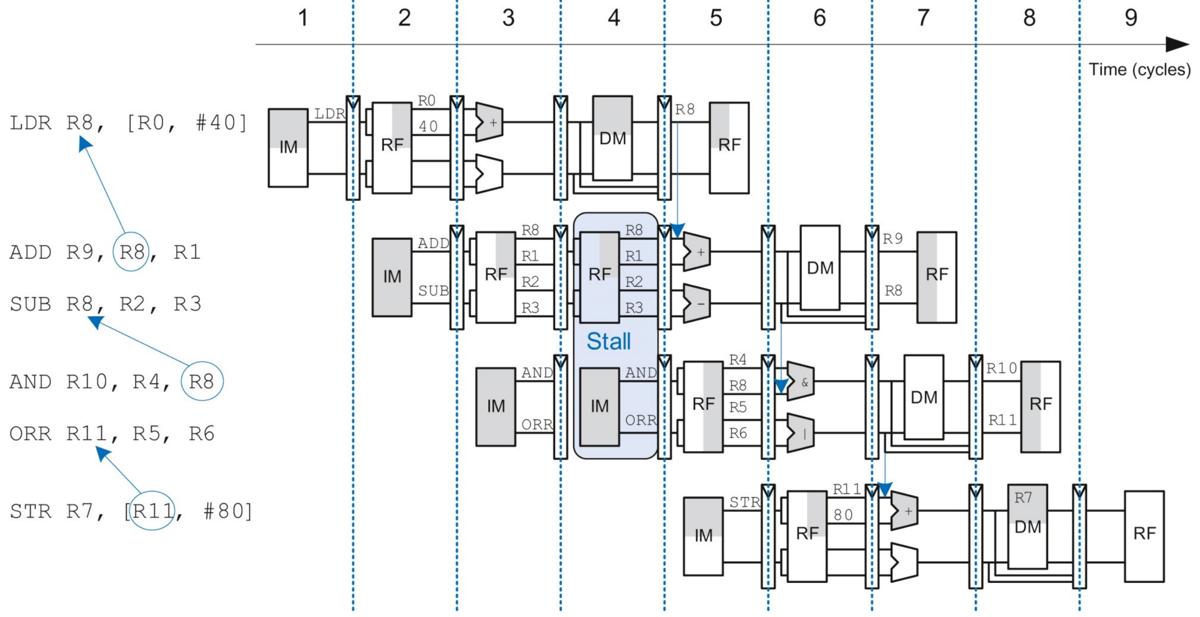
- Example program where IPC = 2 is possible

```
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



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

In-Order Superscalar: Tradeoffs

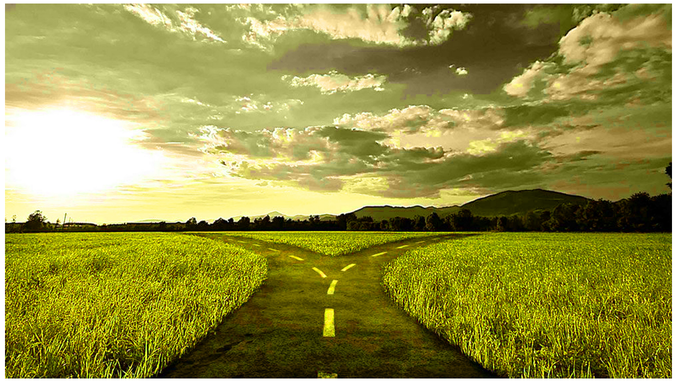
- Superscalar processors encompass spatial + temporal parallelism
  - Two pipelined lanes in one CPU with duplicated resources

- 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

Branch Prediction



Static Branch Prediction

- **Static 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)

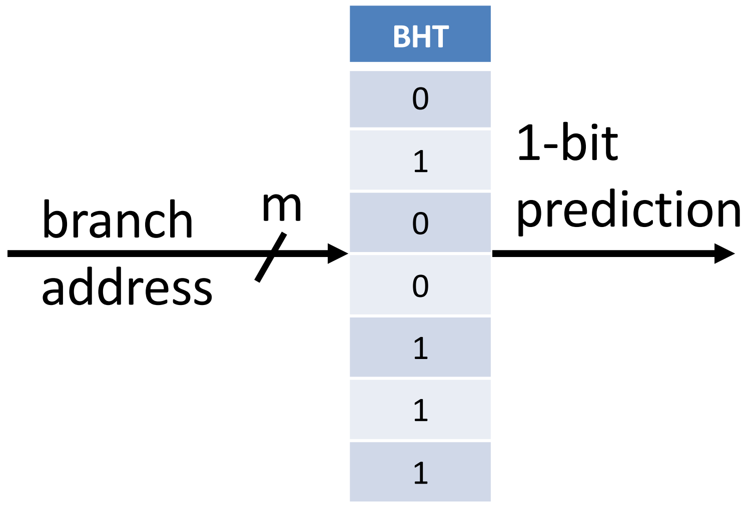
Branch Identification & Behavior

- **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
  - *A small amount of memory indexed by the low-order bits of branch address*
- **Key Idea:** *Store a single bit that says branch was recently taken or not*



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

Operation

- **Placement & Access:** Fetch stage

- Predicted untaken: Fetch PC + 4
- Predicted taken: Compute/predict target address and fetch from target

- Outcome matches prediction: **Noting to do**
- Outcome does not match prediction:
  - Flip the entry in the BHT
  - Flush the pipeline (EXECUTE, DECODE, FETCH), and update the PC

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

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV  R1,  #0
    MOV  R0,  #1
FOR
    CMP  R0,  #10
    BGE  DONE
    ADD  R1,  R1,  R0
    ADD  R1,  R1,  R0
    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  R1,  #0
    MOV  R0,  #1
FOR
    CMP  R0,  #10
    BGE  DONE
    ADD  R1,  R1,  R0
    ADD  R1,  R1,  R0
    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
Direction	NT	NT	NT	NT	NT	NT	NT	NT	NT	T

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
    MOV  R1,  #0
    MOV  R0,  #1
FOR
    CMP  R0,  #10
    BGE  DONE
    ADD  R1,  R1,  R0
    ADD  R1,  R1,  R0
    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
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  R1,  #0
    MOV  R0,  #1
FOR
    CMP  R0,  #10
    BGE  DONE
    ADD  R1,  R1,  R0
    ADD  R1,  R1,  R0
    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
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  R1,  #0
    MOV  R0,  #1
FOR
    CMP  R0,  #10
    BGE  DONE
    ADD  R1,  R1,  R0
    ADD  R1,  R1,  R0
    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
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

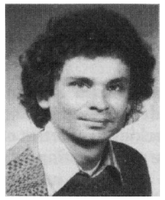
Anomalous Decision

- 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/direction

- A 1-bit predictor is “thrown off” by a single anomolous decision

Smith's Algorithm

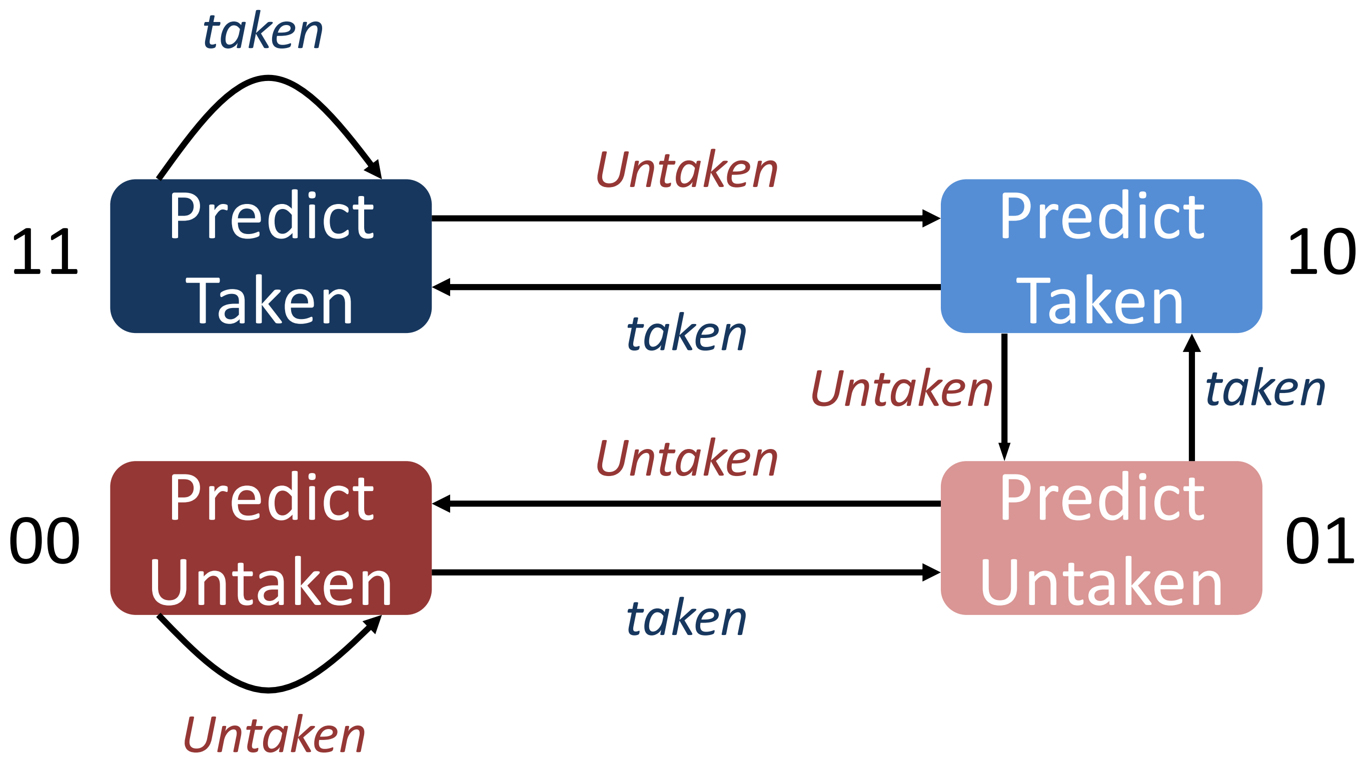


- **1979:** James E. Smith patents branch prediction at Control Data
  - Notices the performance pathology of 1-bit predictor at loop termination
  - **Key insight:** Add hysterisis (inertia) to the predictor's state
- The same outcome must occur multiple times to reach the strong states
- A saturating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

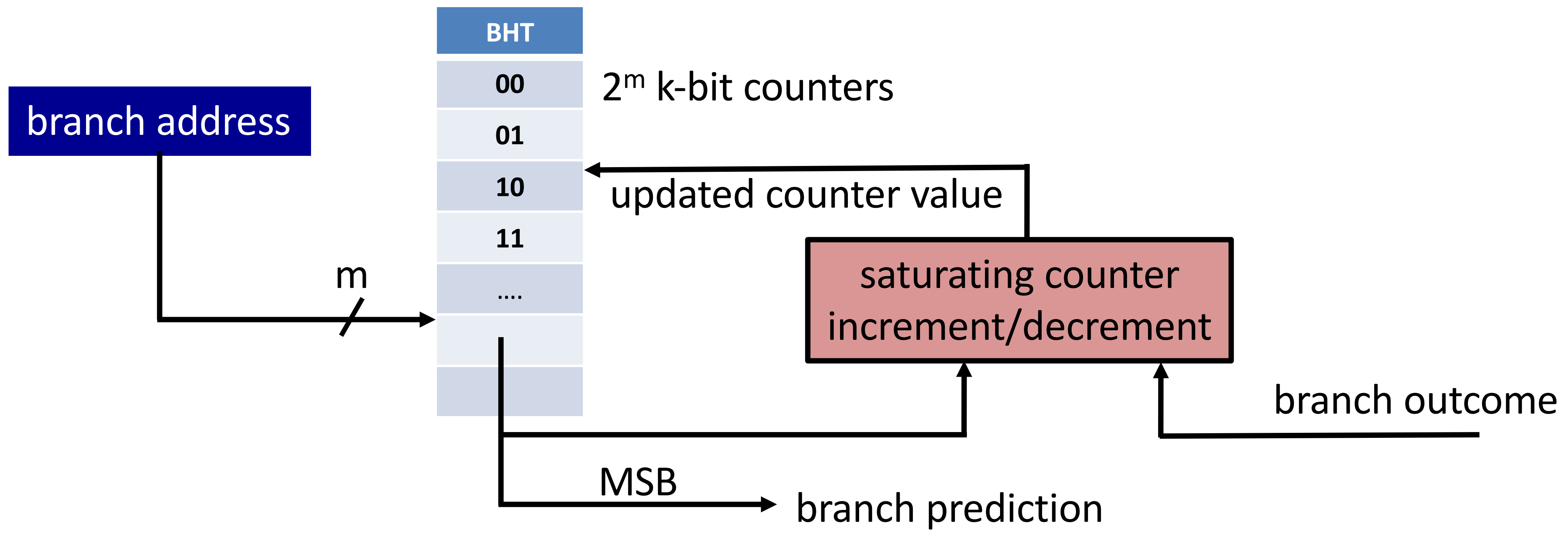
k = 2

- **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 a single **anomolous** decision

Branch Direction
1
1
0 anomaly
1 anomaly
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 a single **anomolous** decision

Branch Direction	**Smith$_1$**	
	State	**Prediction**
1	1	1
1	1	1
0 anomaly	1	1 (misprediction)
1 anomaly	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 a single **anomolous** decision

Branch Direction
1
1
0 anomaly
1 anomaly
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 a single **anomolous** decision

Branch Direction	Smith$_1$		Smith$_2$	
	State	**Prediction**	**State**	**Prediction**
1	1	1	11	1
1	1	1	11	1
0 anomaly	1	1 (misprediction)	11	1 (misprediction)
1 anomaly	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
  - Initiate pipeline flush (details are not in scope)

Correlating Branch Predictors

- In real programs, the behavior of one branch is correlated with that of another
- **Key drawback of previous predictors:** A predictor that uses the outcomes of only a single branch to predict the behavior of that branch does not capture correlation b/w branches
- Correlating branch predictors use branch history and branch address to predict the branch outcome
- **Correlating branch predictors** consider the local history of a branch and global history across all branches

```
if (aa == 2)
      aa = 0;
if (bb == 2)
      bb = 0
if (aa != bb)
      {...}
```

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

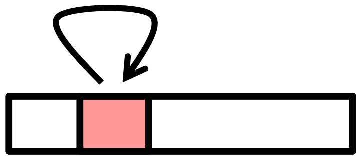
- **Random branches** are increasingly common (ML, NLP, GPT)

Locality and its Exploitation

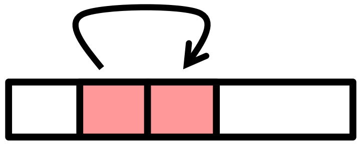
Locality

- **Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern). **Spatial locality**
  - Reference variable `sum` each iteration. **Temporal locality**
- **Instruction references**
  - Reference instructions in sequence. **Spatial locality**
  - Cycle through loop repeatedly. **Temporal locality**

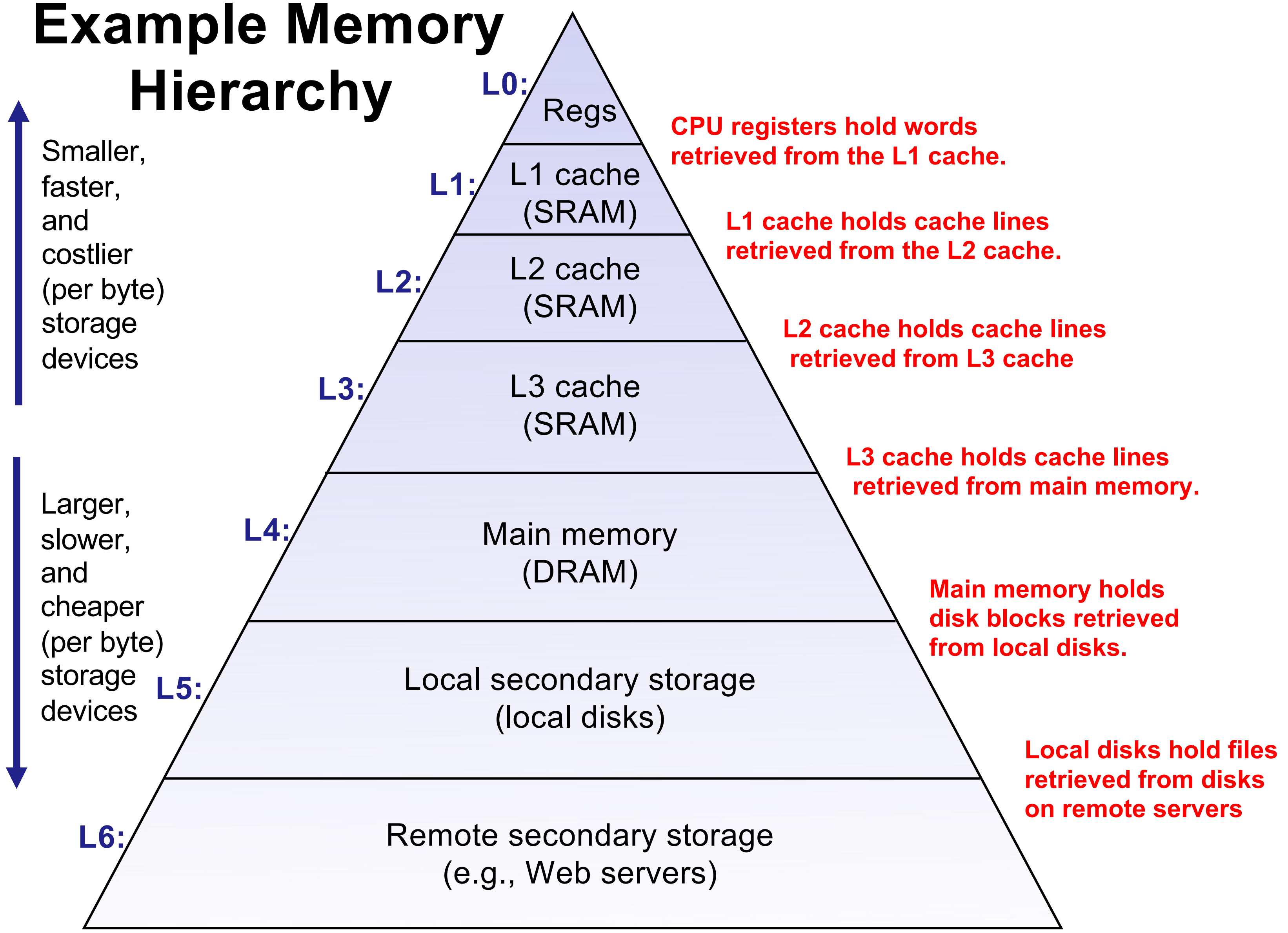
Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software:**
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- **These fundamental properties complement each other beautifully.**

- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy.**

Example Memory Hierarchy



Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers

Cache

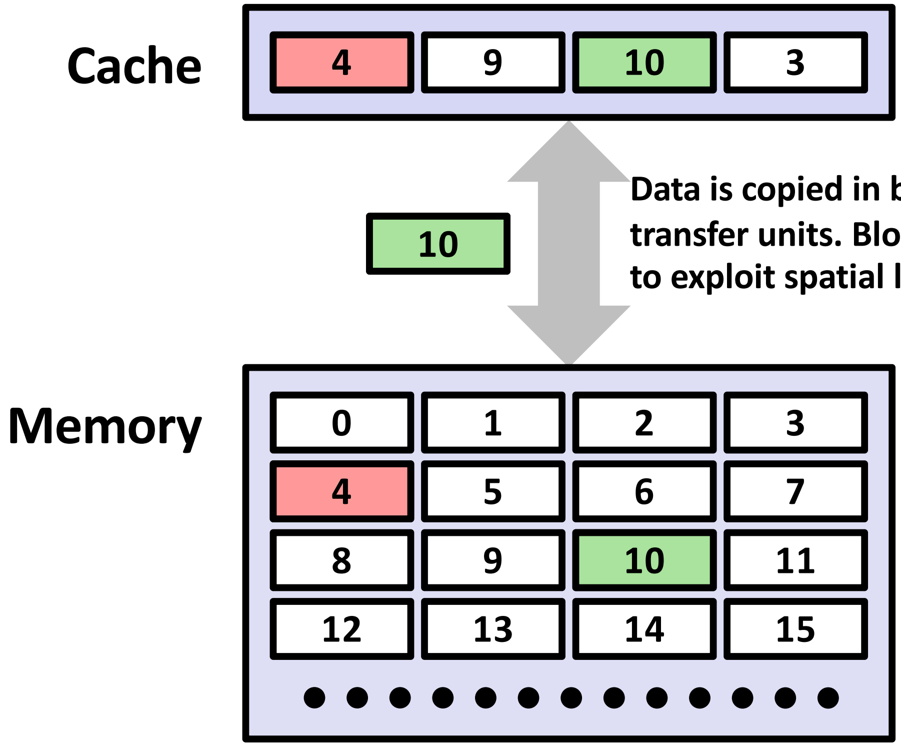
- ***Cache:*** **A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.**

- **Fundamental idea of a memory hierarchy:**
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.

- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

- ***Big Idea:*** **The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.**

General Cache Concepts

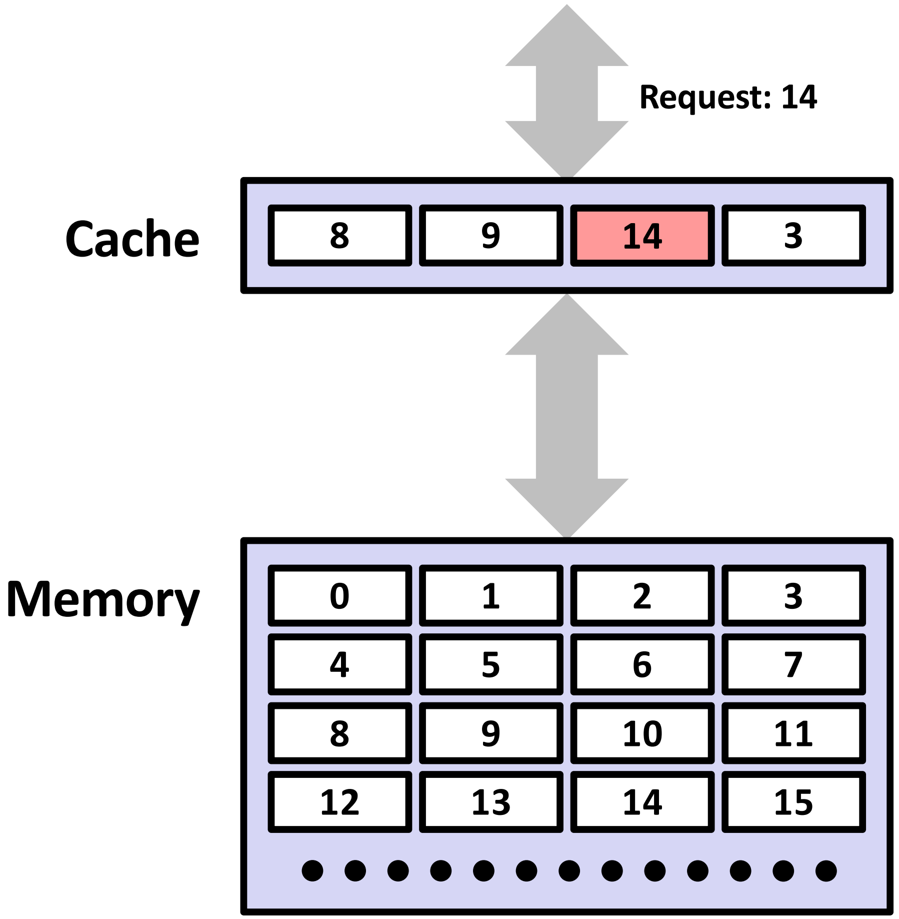


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units. Blocks are large to exploit spatial locality

Larger, slower, cheaper memory viewed as partitioned into “blocks”

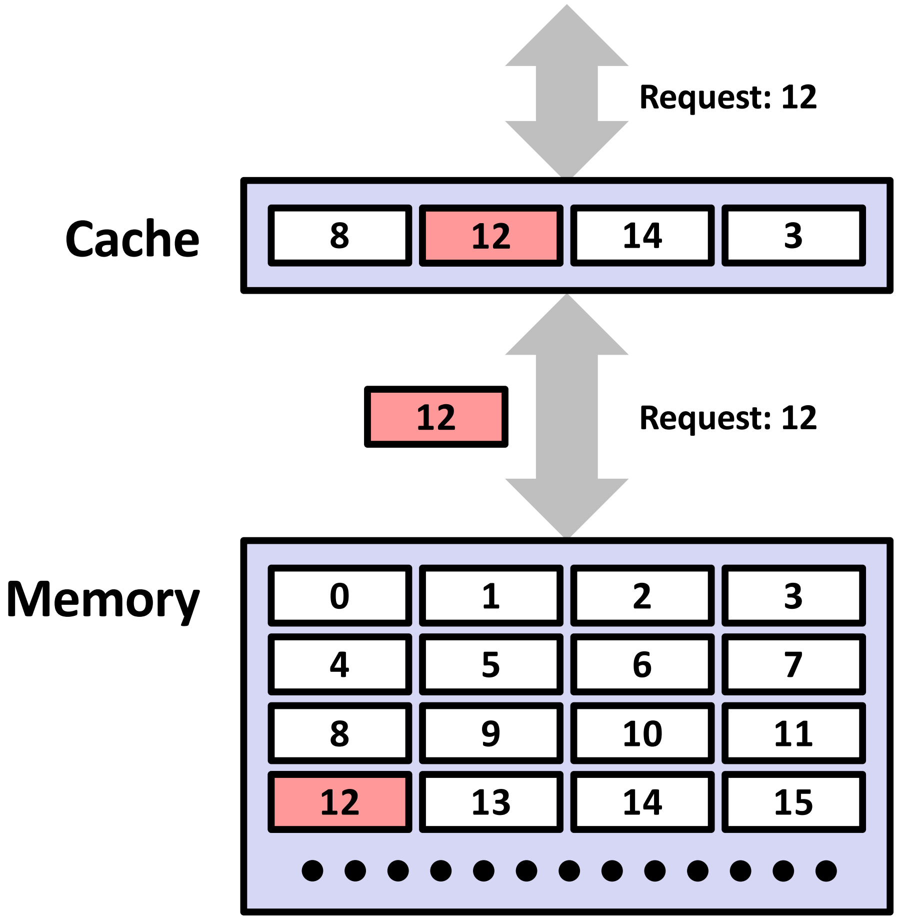
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

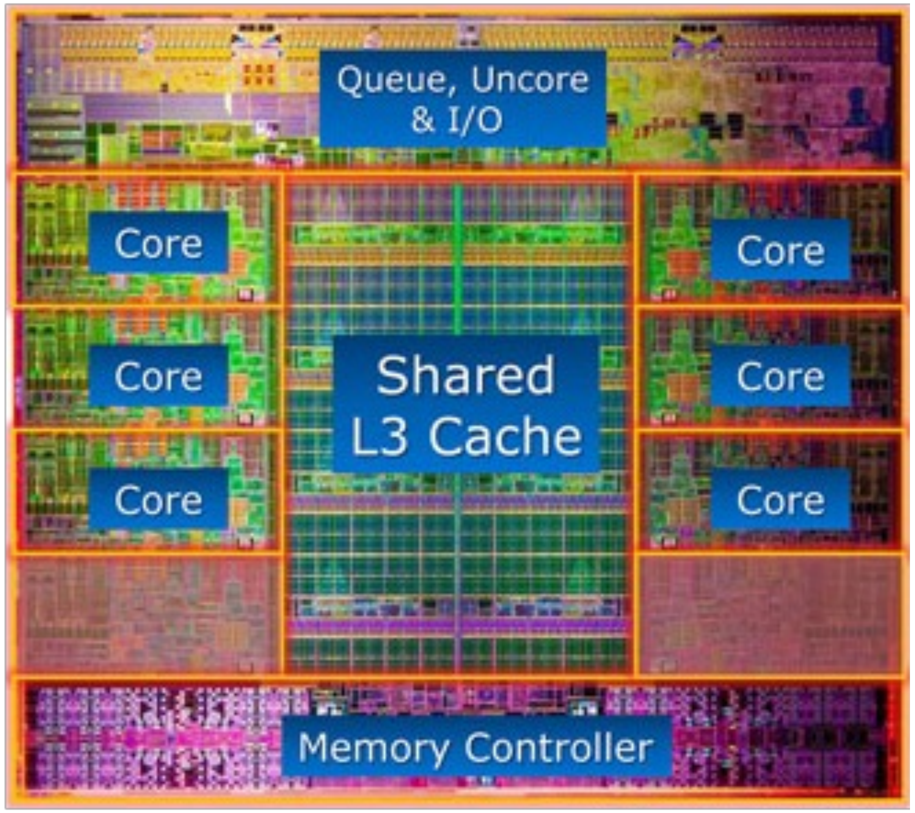
Block b is not in cache:
Miss!

Block b is fetched from memory

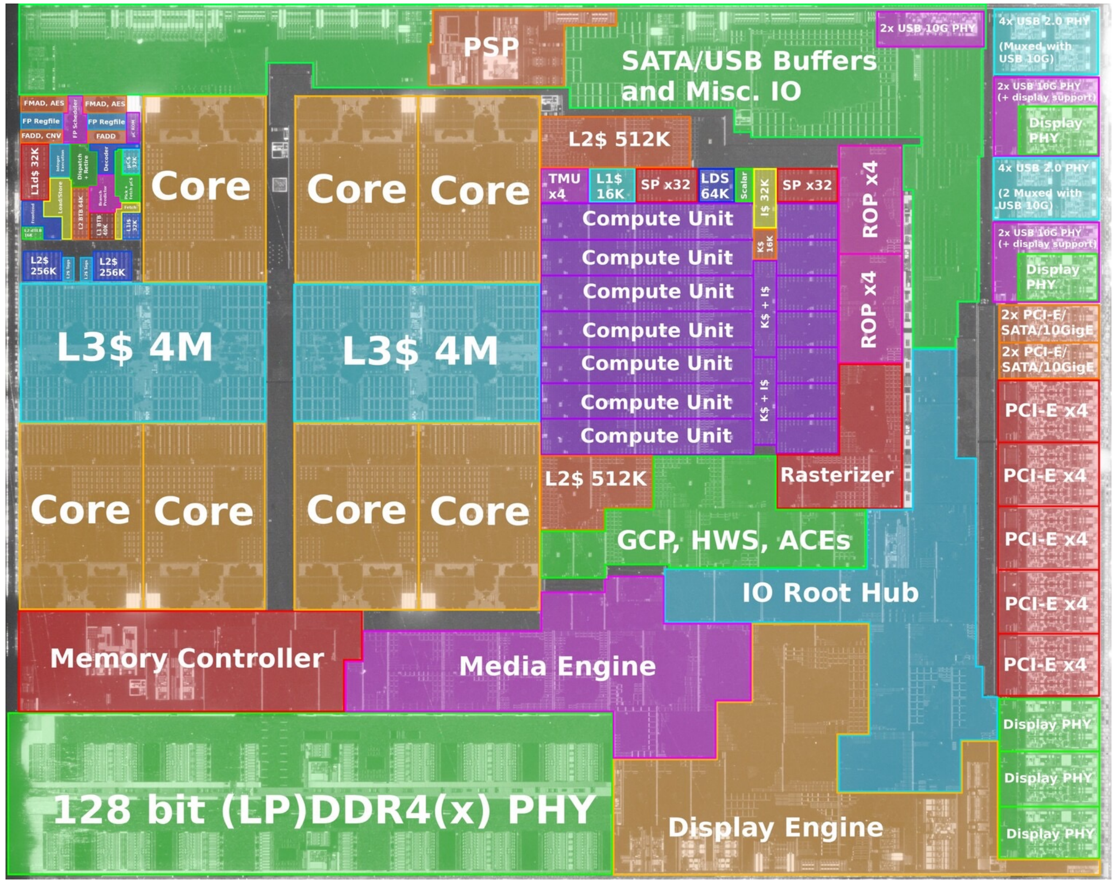
Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Cache Hierarchy in Real-Life CPU



Cache Hierarchy in Real-Life CPU



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 stalls for 100 cycles when memory is busy
- Next steps
  - Aggressive in-order CPU that keeps doing useful work in the presence of cache miss (until a RAW hazard is unavoidable)
  - Out-of-order CPU that continues doing useful work in the presence of cache miss and RAW hazard