COMP2300-COMP6300-ENGN2219
Computer Organization &
Program Execution

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian
National
University

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

Recall: Temporal Parallelism

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

Recall: Automotive Pipeline

Recall: Pipelining

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

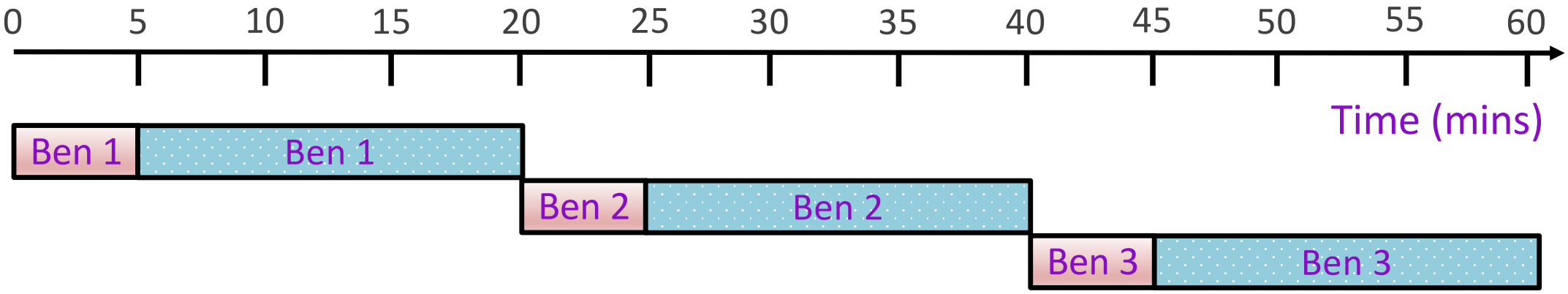
- The challenge of pipelining is to find stages of equal length

- Let's go back to baking cookies

Recall: Cookie Parallelism

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

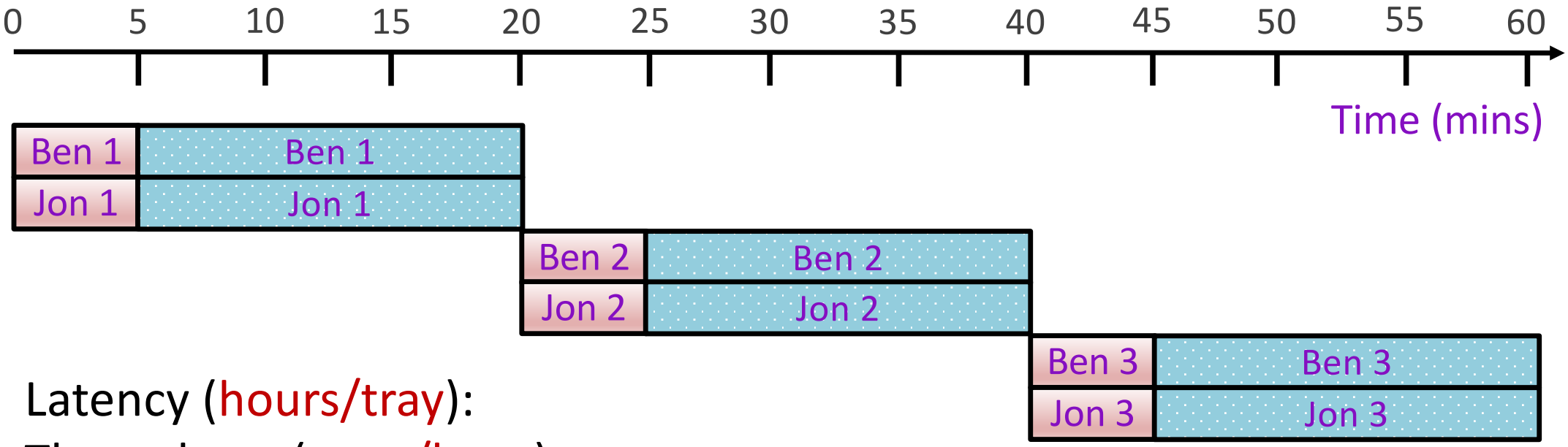
No Parallelism (Ben Only)



Latency (hours/tray):

Throughput (trays/hour):

Spatial Parallelism (Ben & Jon)

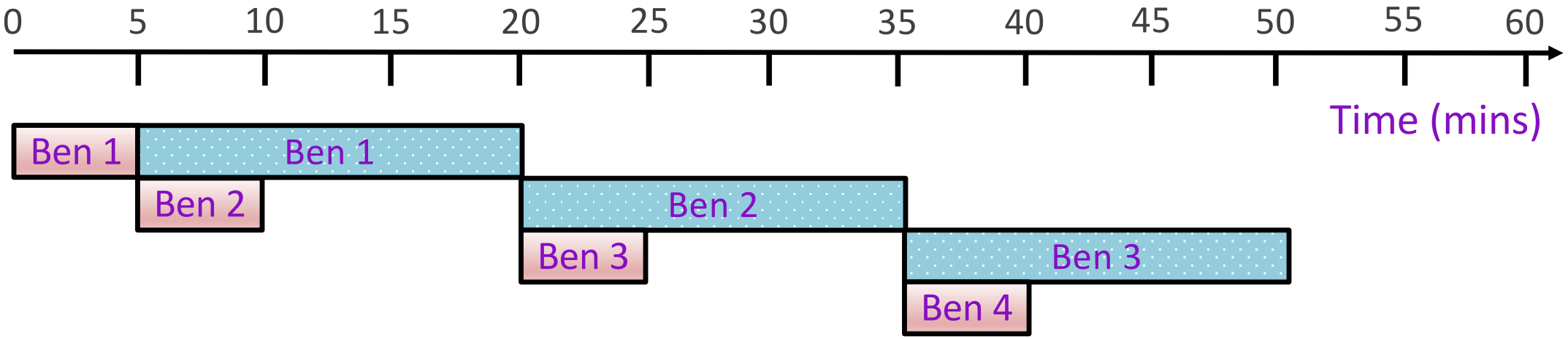


Latency (hours/tray):

Throughput (trays/hour):

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

Pipelining (Ben Only)

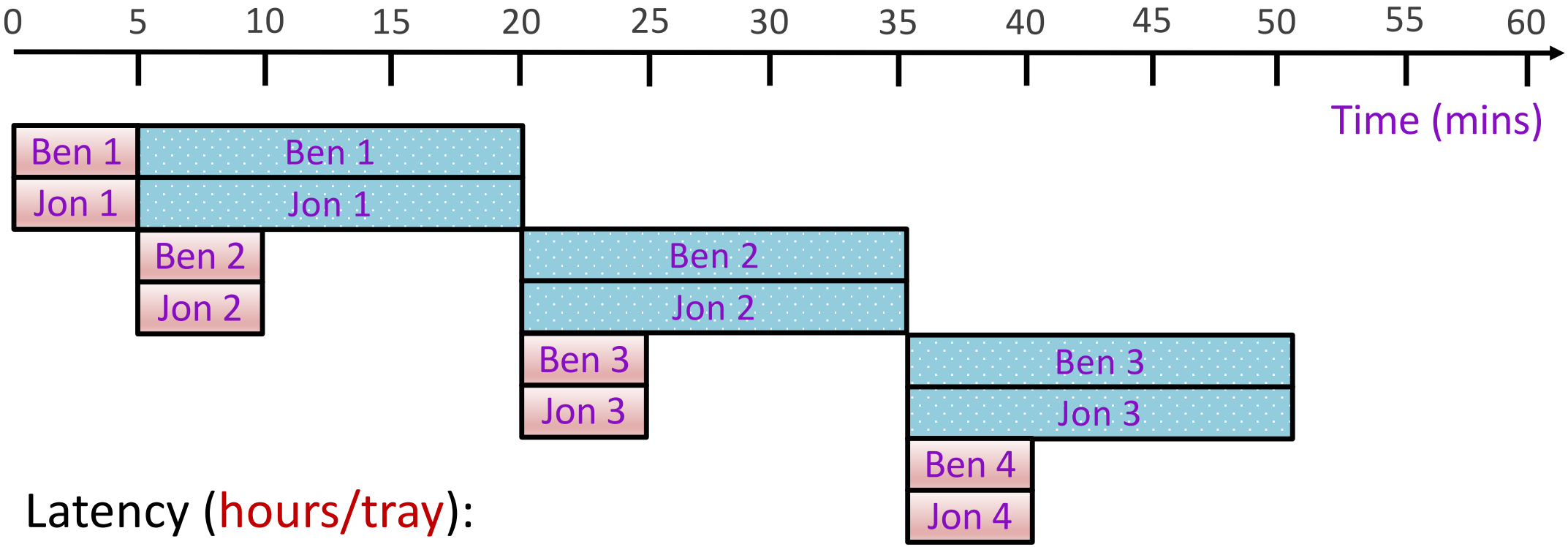


Latency (hours/tray):

Throughput (trays/hour):

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

Spatial + Temporal Parallelism



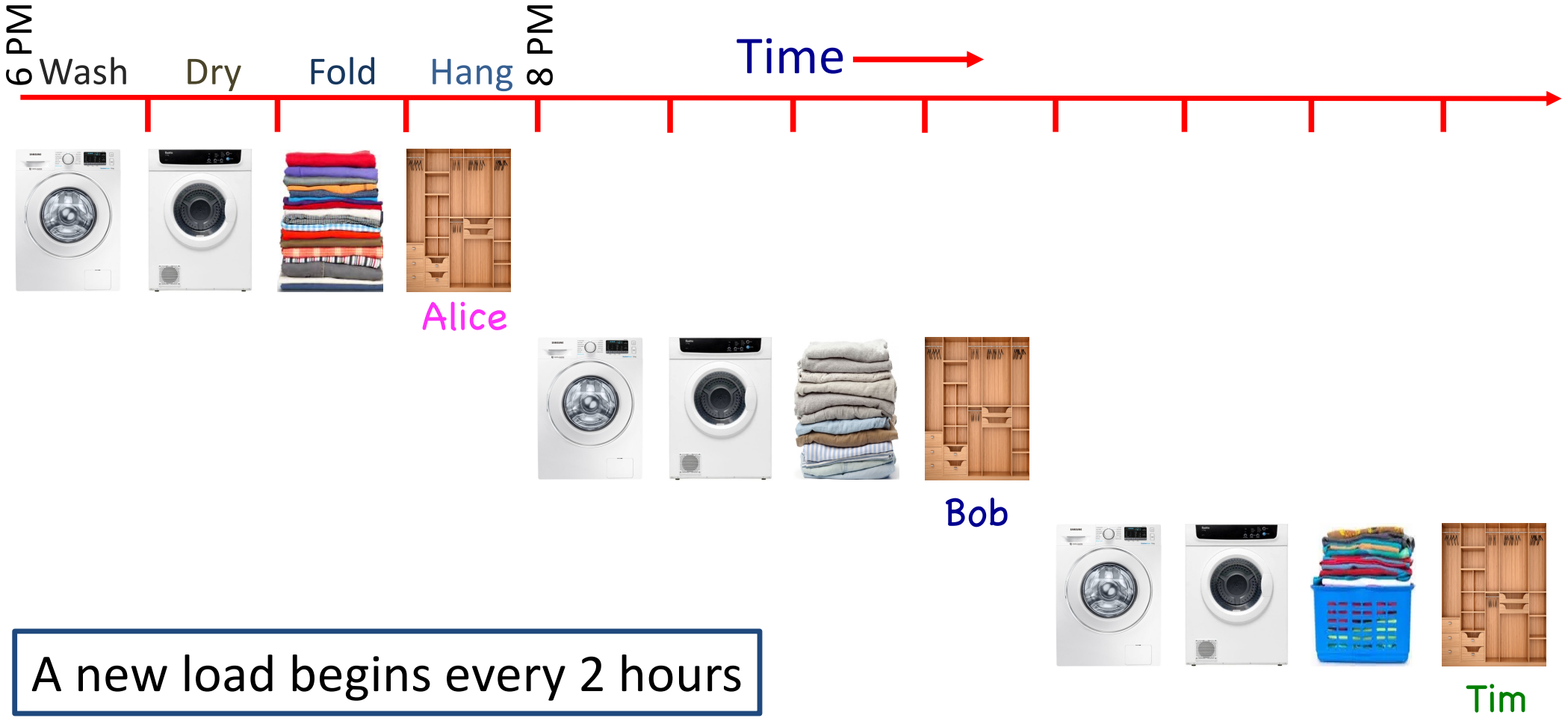
Latency (hours/tray):

Throughput (trays/hour):

Answers Explained

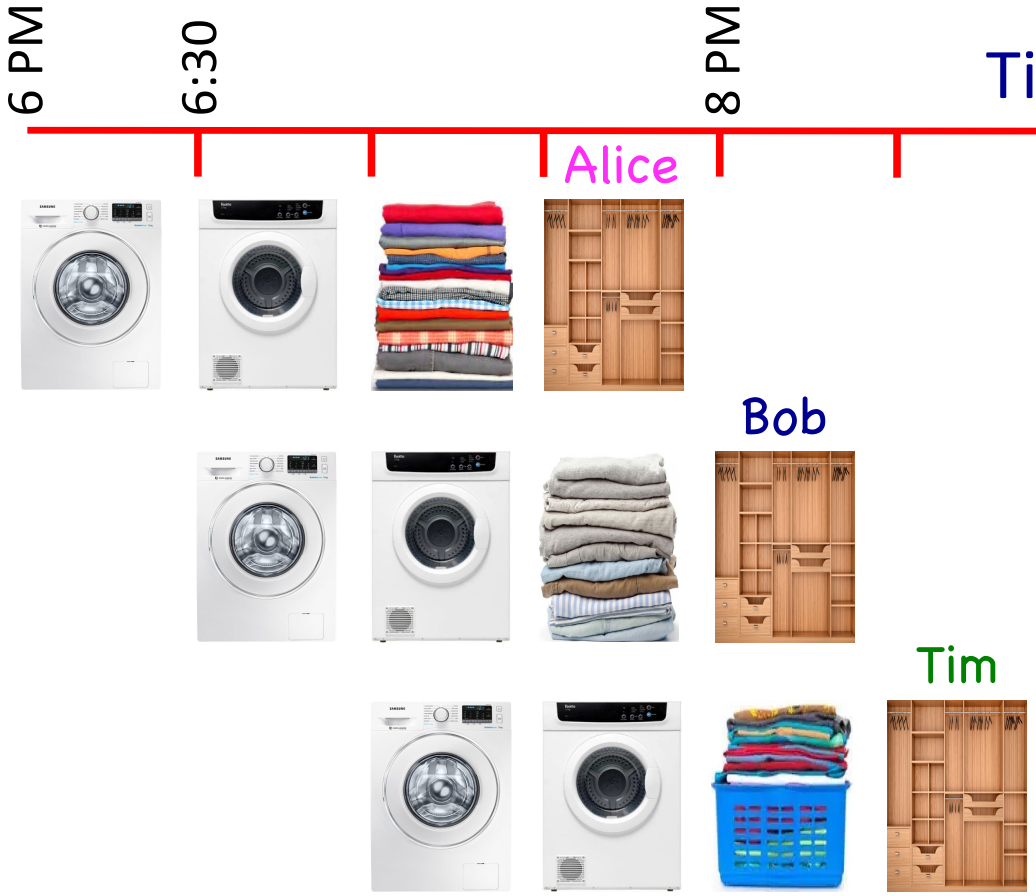
- **No parallelism**
  - Latency is clearly 20 minutes (1/3 hours/tray)
  - Throughput is 3 trays per hour
- **Spatial parallelism**
  - Latency remains unchanged as it still takes 20 mins to finish a tray
  - Throughput is doubled via duplication: 6 trays per hour
- **Pipelining**
  - Latency for a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

Sequential Laundry



A new load begins every 2 hours

Pipelined Laundry



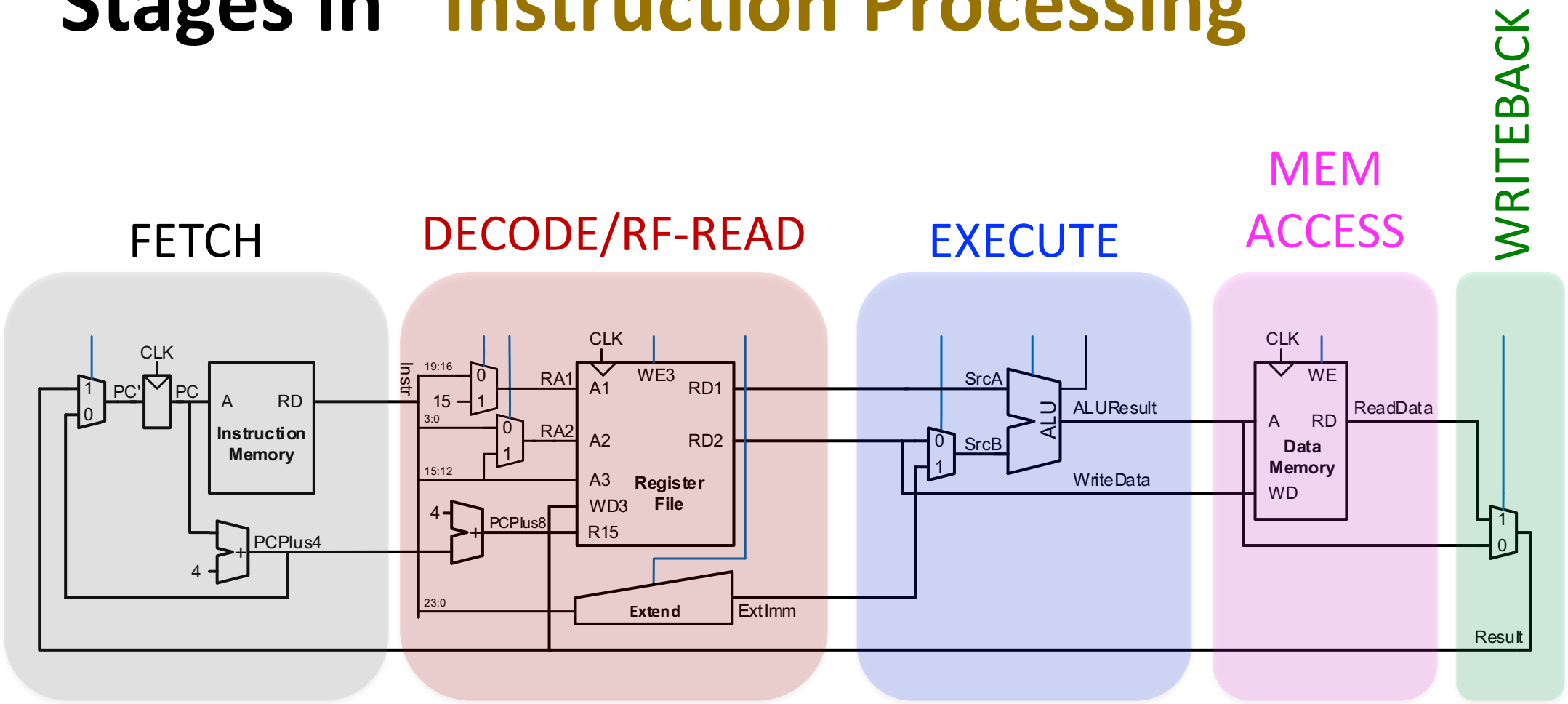
- A new load begins every 30 mins
  - 120 mins divided by 4
  - Speed-up of 4!

Recall: Pipelining Circuits

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

Pipelined Microarchitecture

Stages in "Instruction Processing"



Pipelined Microarchitecture: Key Idea

- Multiple instructions (up to 5) can be in the pipeline in any cycle

- Each instruction can be in a different stage
  - Idea is for **"maximizing utilization"** of hardware resources

- Stages must be isolated from one another using pipelined register (non-arch. registers). Referred to as "PPR"

- The work of a stage should be preserved in a PPR each cycle

Key Idea (Continued)

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

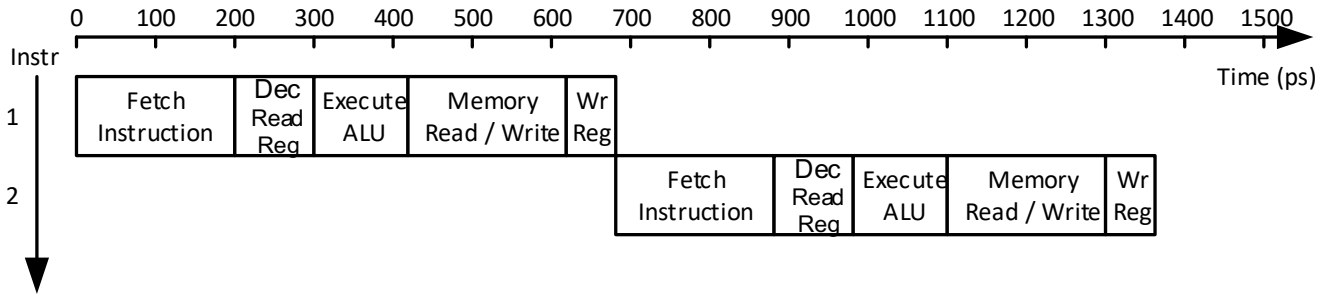
Timing Diagrams

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

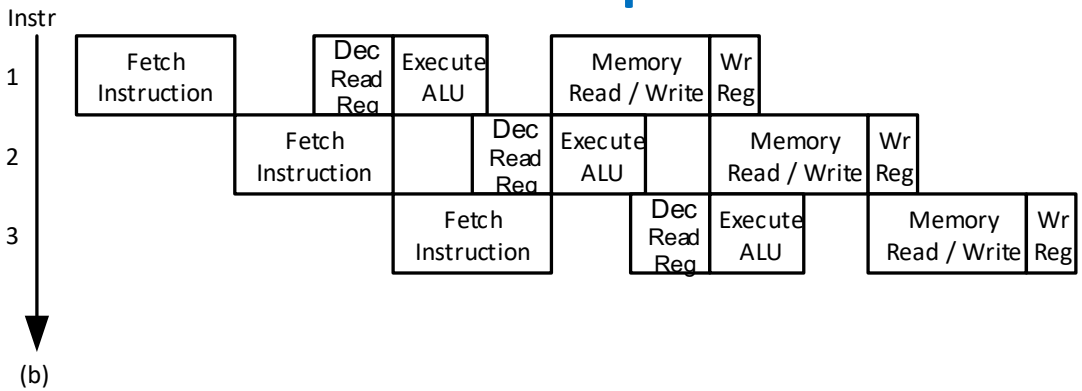
Timing Diagrams

Assumption of logic element delays from Table 7.5 of textbook

Single-Cycle



Pipelined



Performance Analysis

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

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

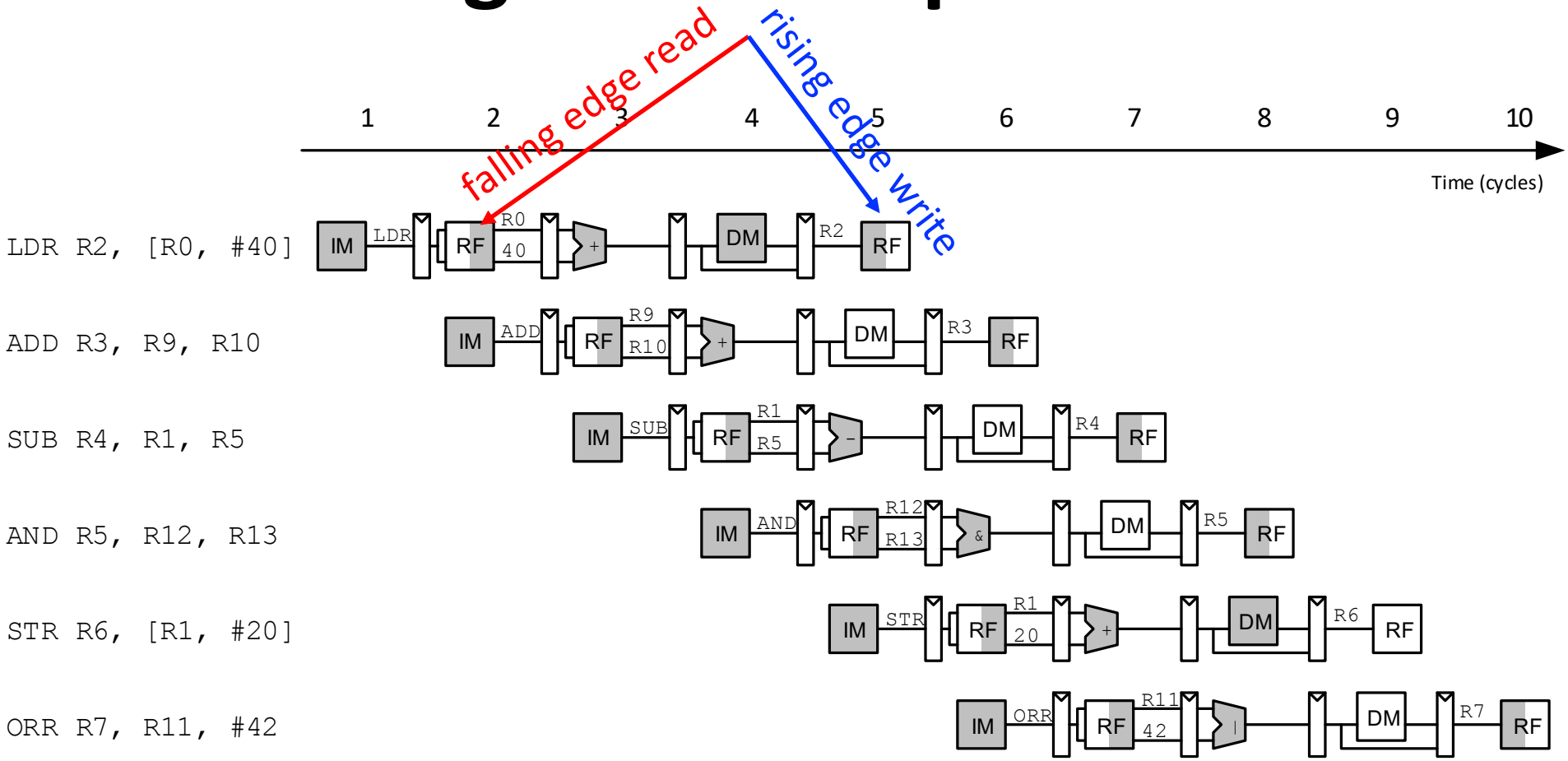
Instruction Latency with Pipelining

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

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

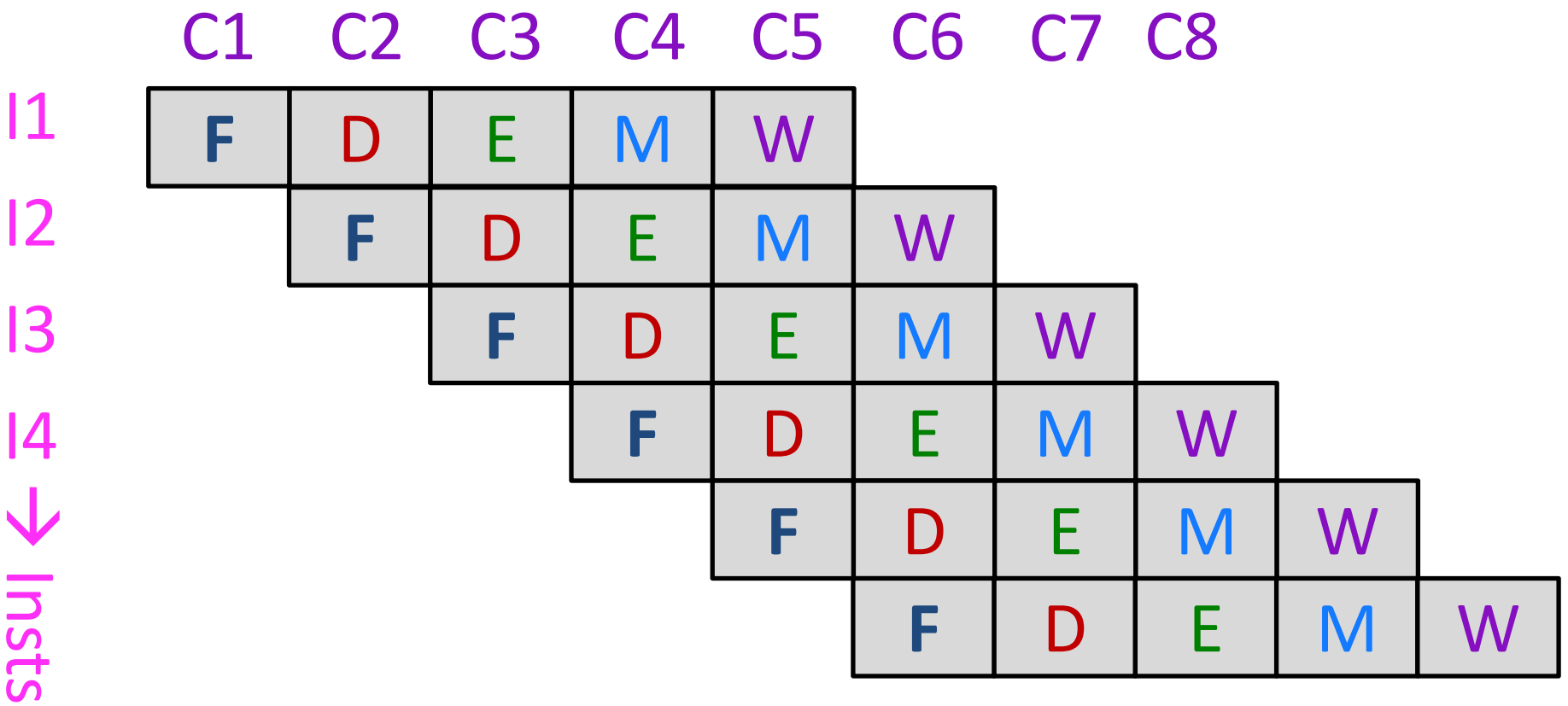
- Pipelining helps increase the throughput of an entire workload
  - Workload = Number of instructions
  - Workload must be **"sufficiently"** large

Abstract Diagrams of Pipelined uArch



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

Simplified View of Pipelining

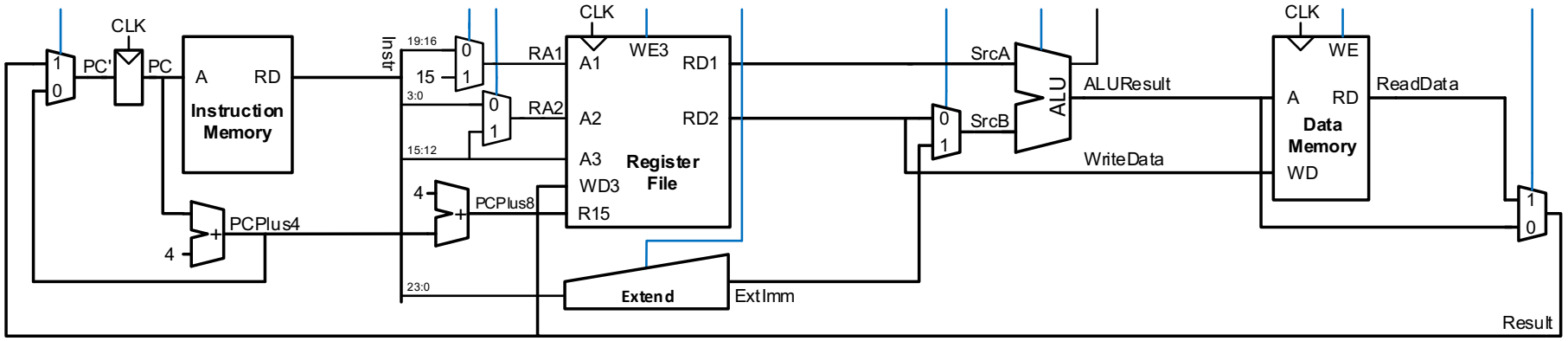


Let’s complete the picture

- Start with the single-cycle microarchitecture

- And insert pipeline registers

Single-Cycle

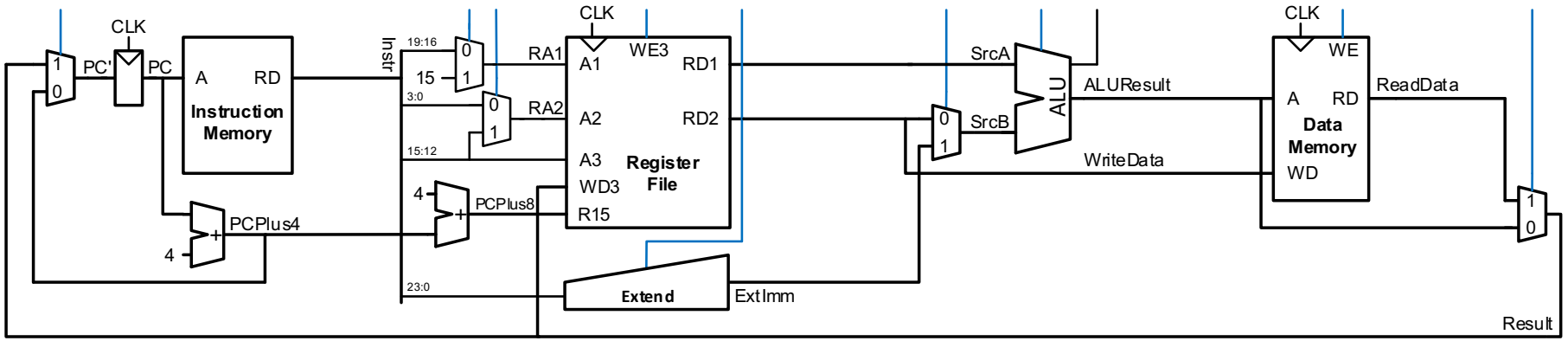


- Once we insert pipeline registers, we would need to pass the results of one stage to the next stage via the pipeline registers

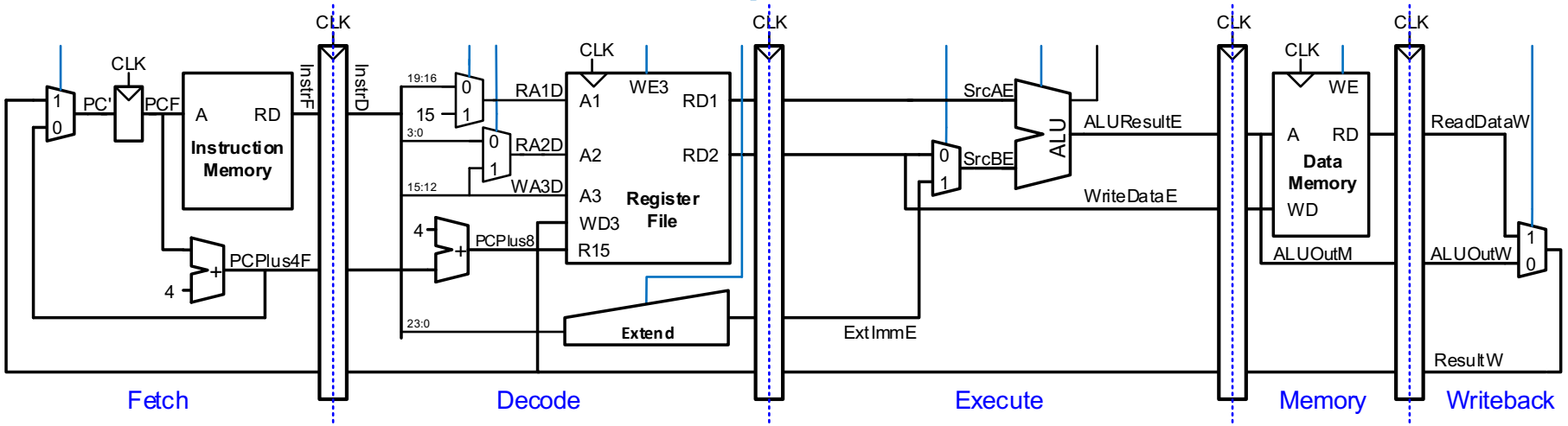
- What is the outcome of the FETCH stage?

Pipeline Microarchitecture

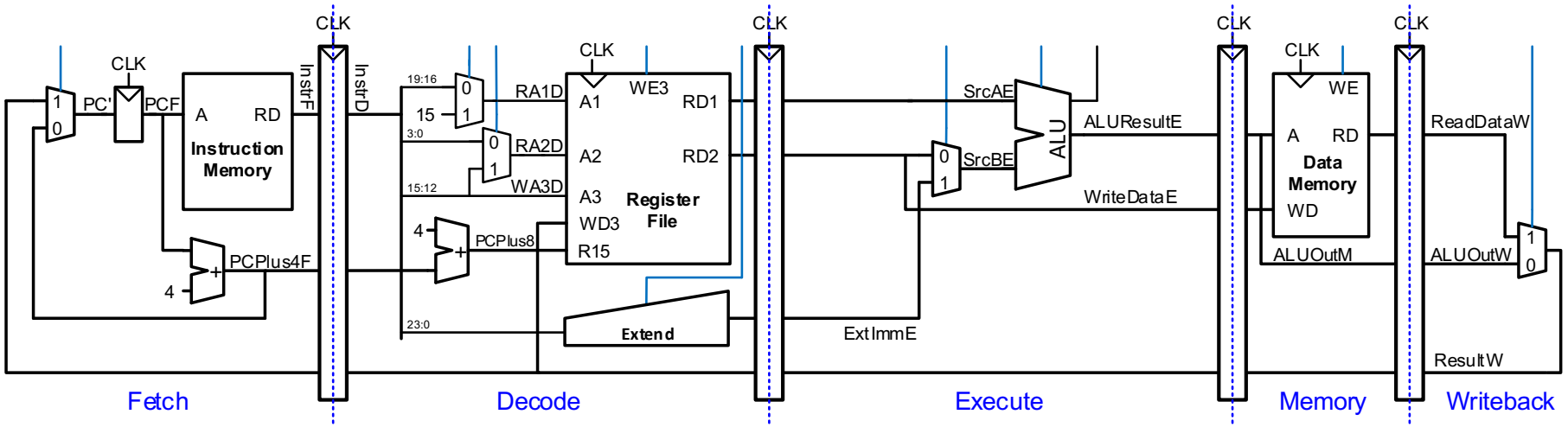
Single-Cycle



Pipelined



Pipeline Microarchitecture



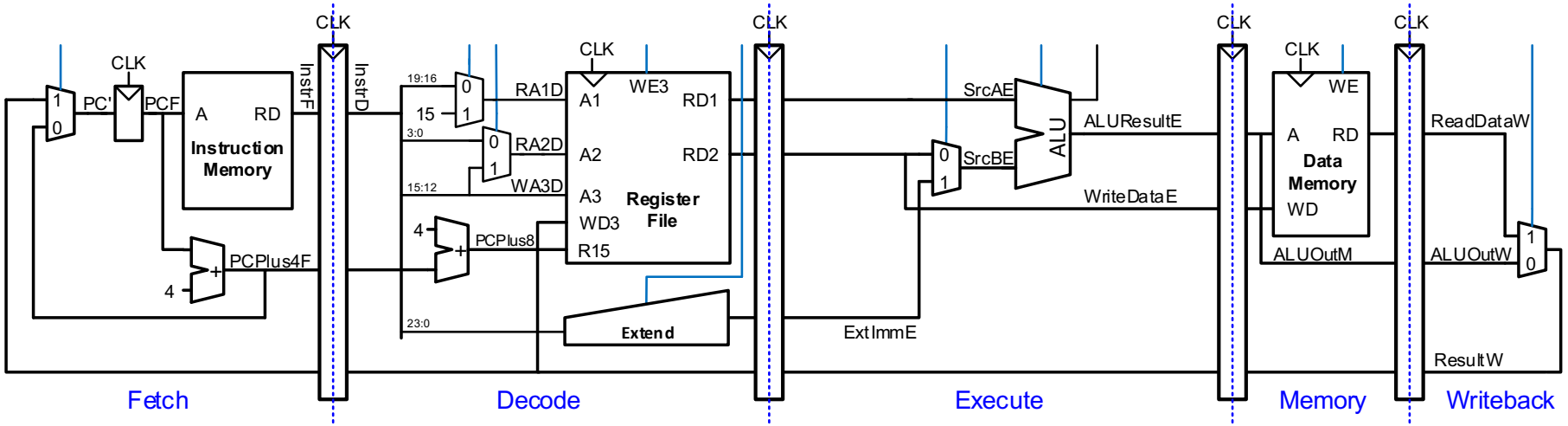
- Stages and their boundaries are indicated in blue
- Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside

Pipeline Operation

- Consider the example instruction sequence

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

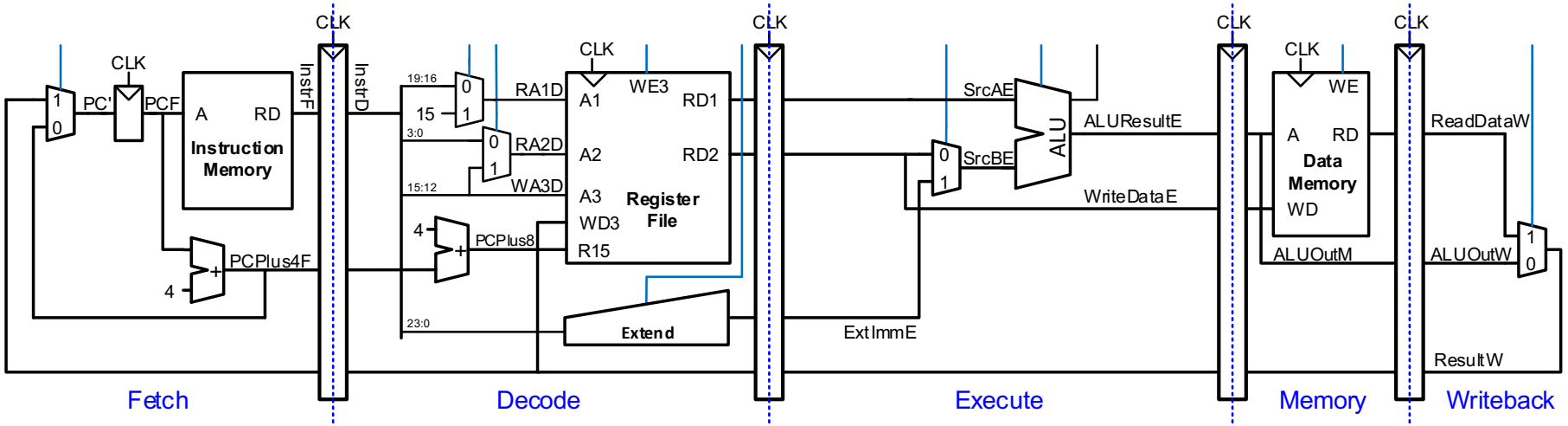
Pipeline Operation: Cycle 1



I1

- Is the pipeline fully utilized? NO

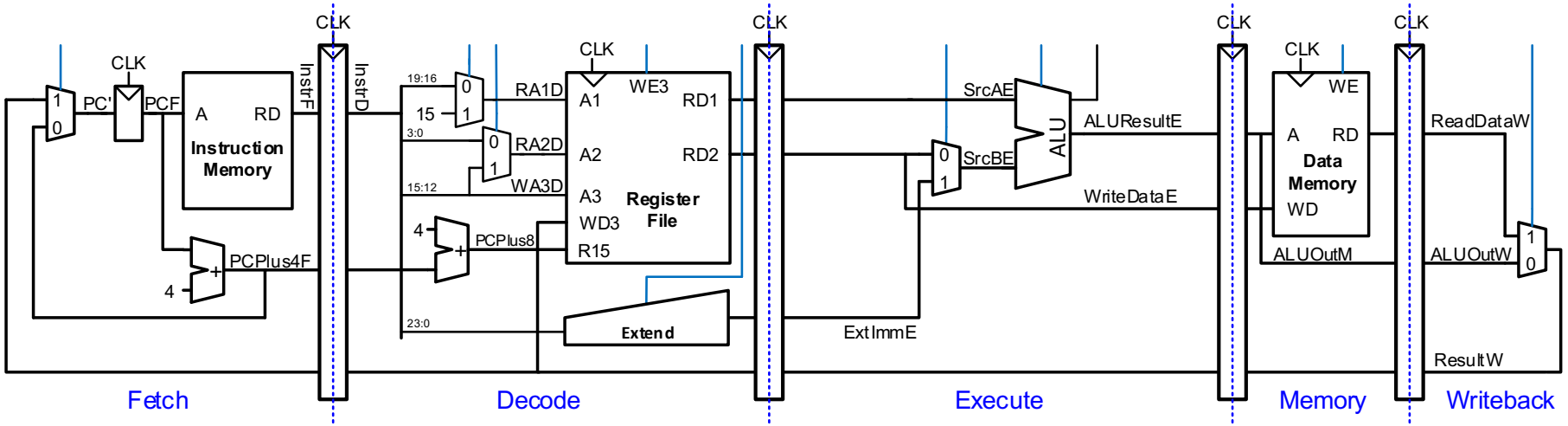
Pipeline Operation: Cycle 2



I2 I1

❑ Is the pipeline fully utilized? NO

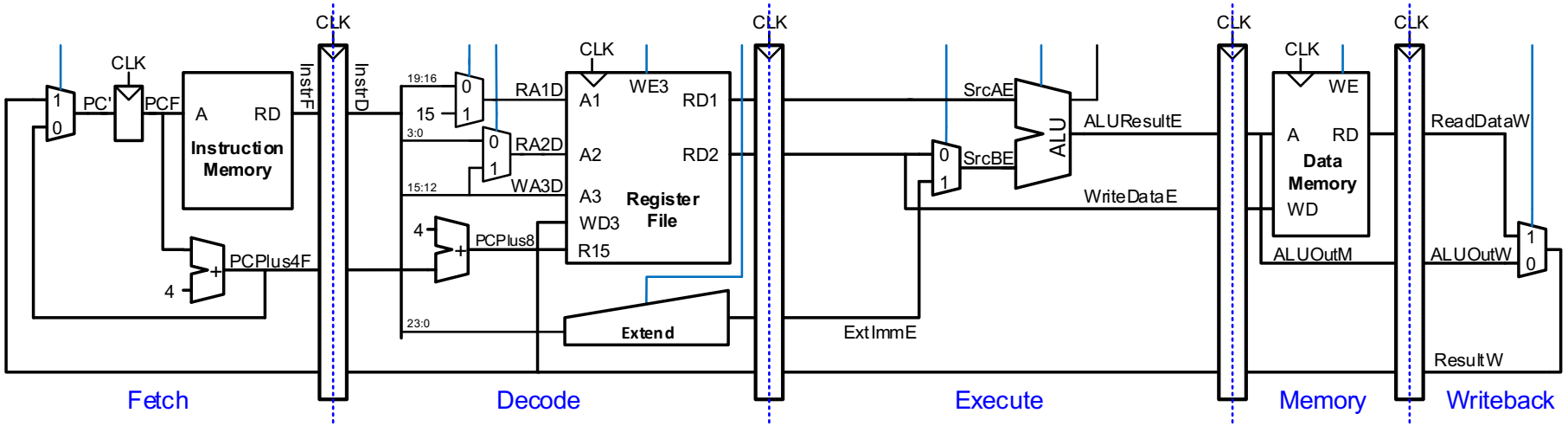
Pipeline Operation: Cycle 3



I3 I2 I1

- Is the pipeline fully utilized? NO

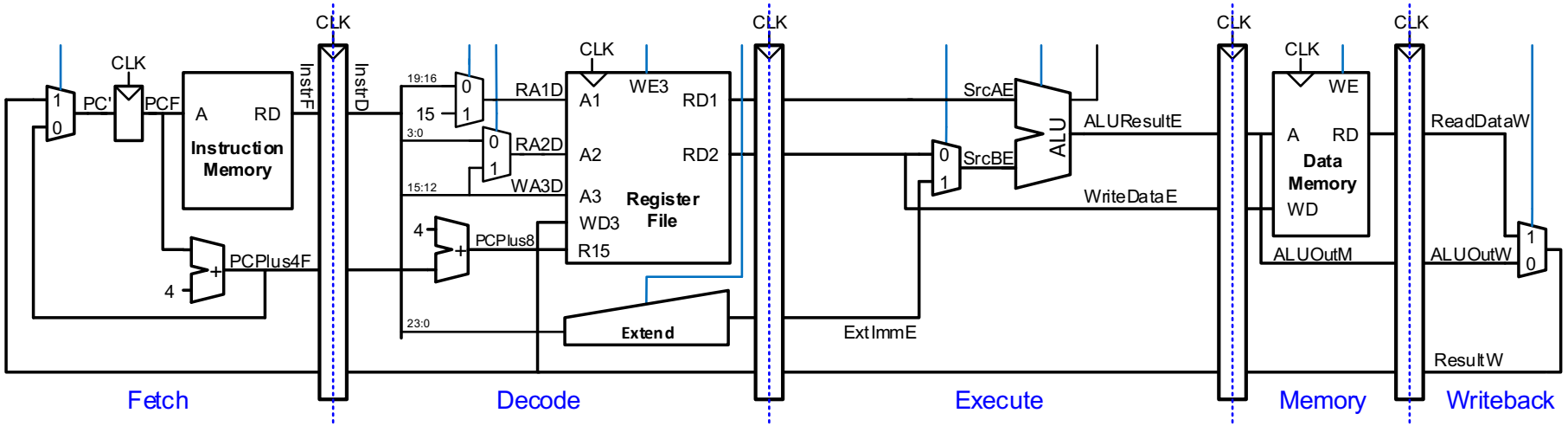
Pipeline Operation: Cycle 4



I4 I3 I2 I1

- Is the pipeline fully utilized? NO

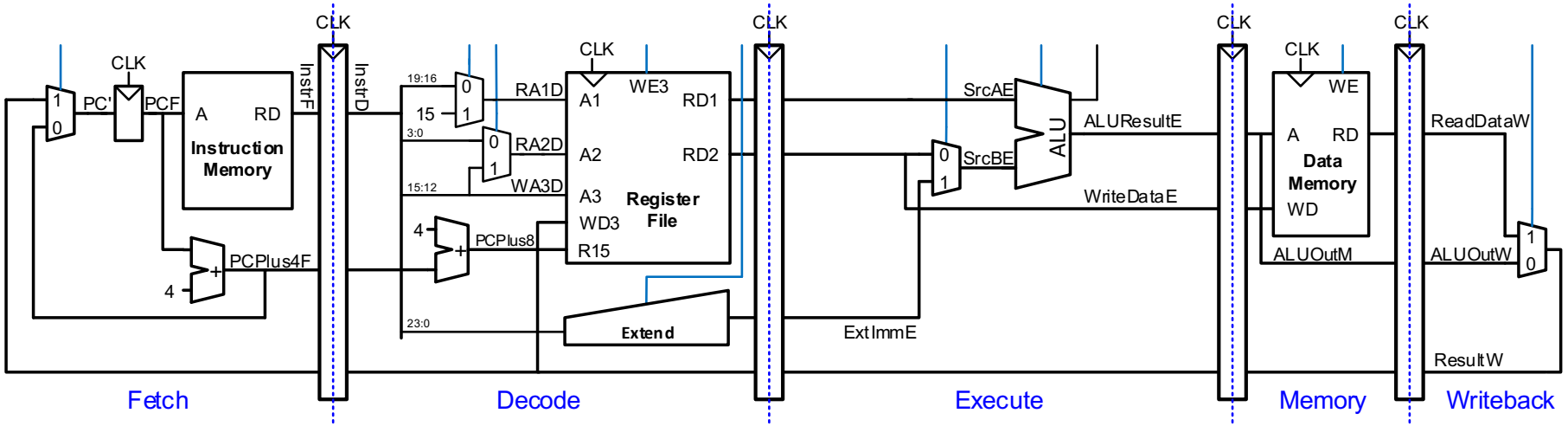
Pipeline Operation: Cycle 5



I5 I4 I3 I2 I1

- Is the pipeline fully utilized? YES

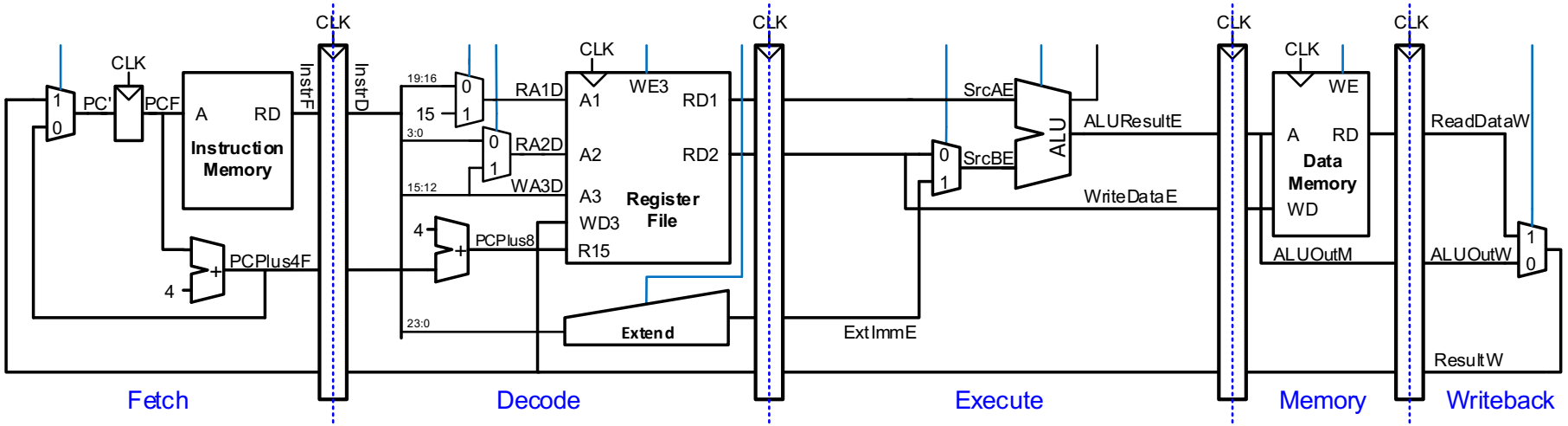
Pipeline Operation: Cycle 6



I6 I5 I4 I3 I2

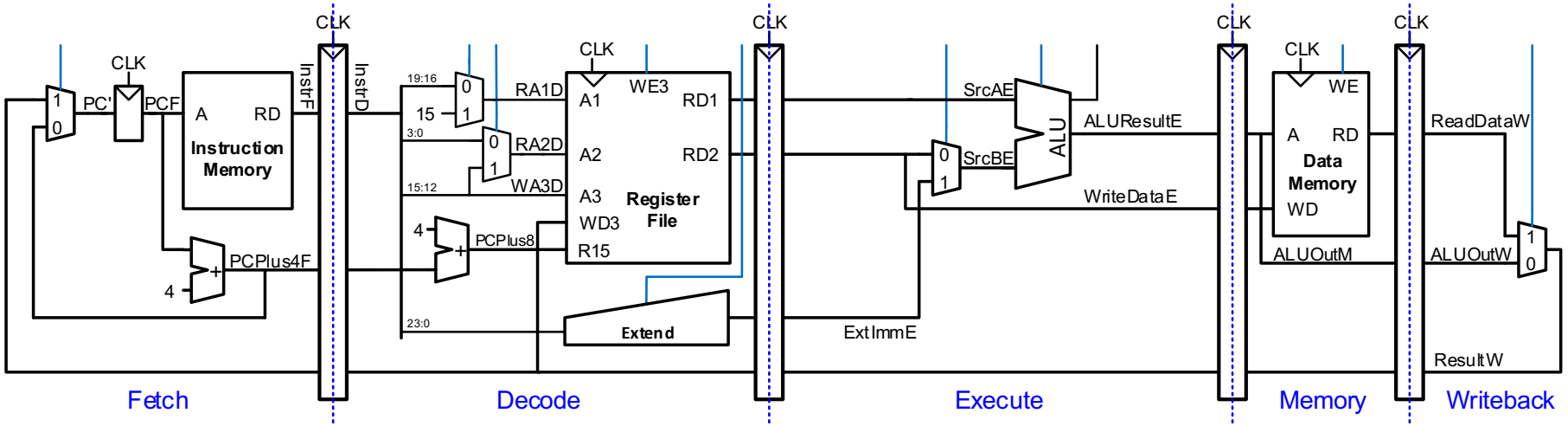
- Is the pipeline fully utilized? YES

Pipeline Operation: Cycle 7



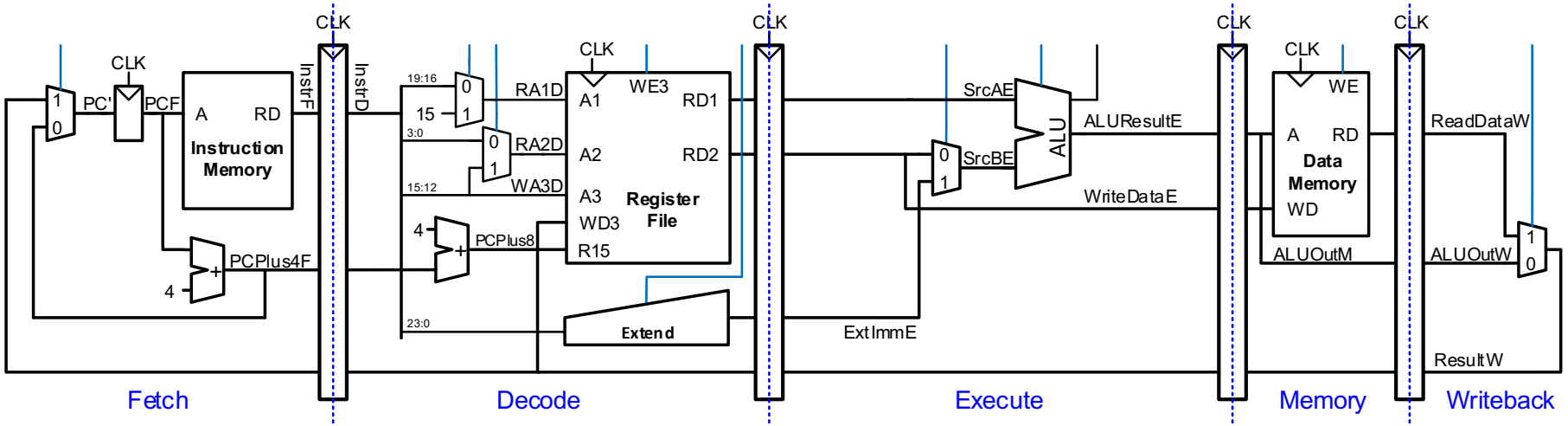
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 8



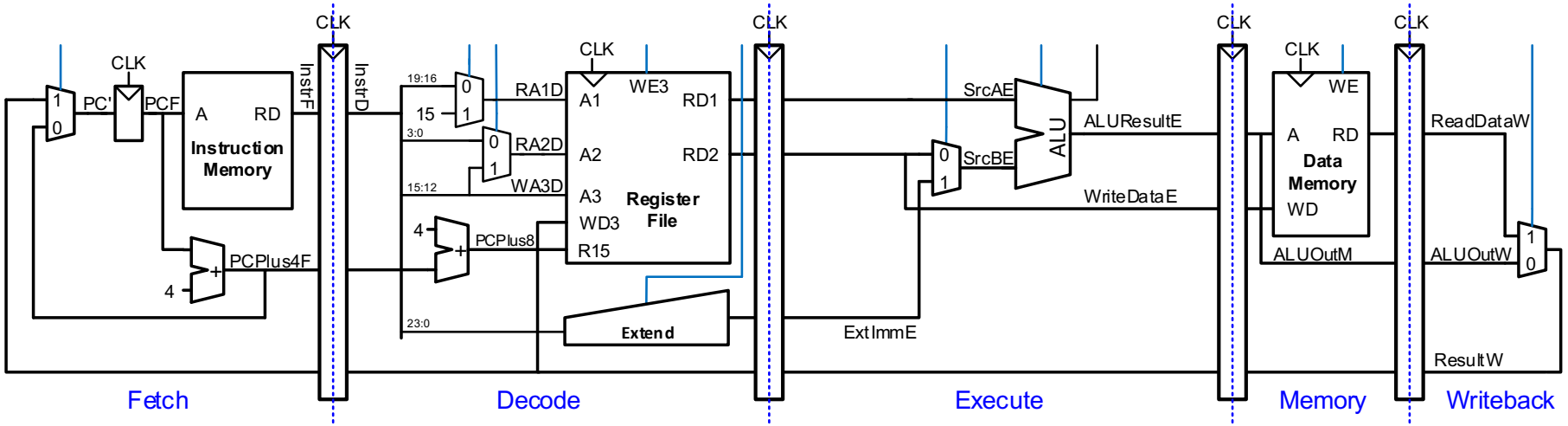
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 9



❑ Is the pipeline fully utilized? NO

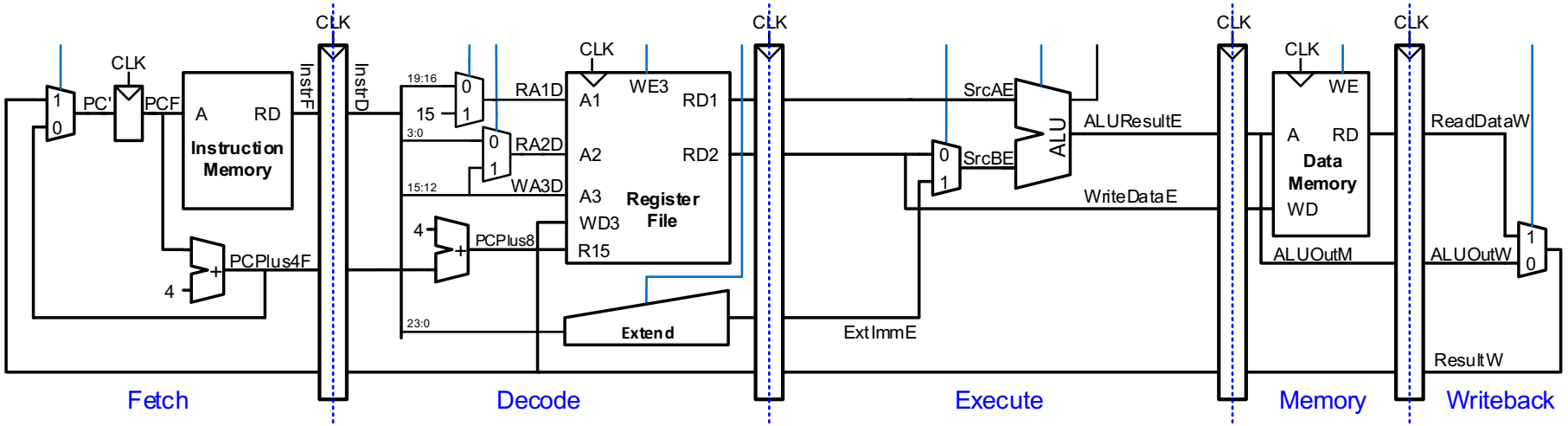
Pipeline Operation: Cycle 10



I6

- Is the pipeline fully utilized? NO

Pipeline Operation



- No more instructions to execute

Performance Analysis

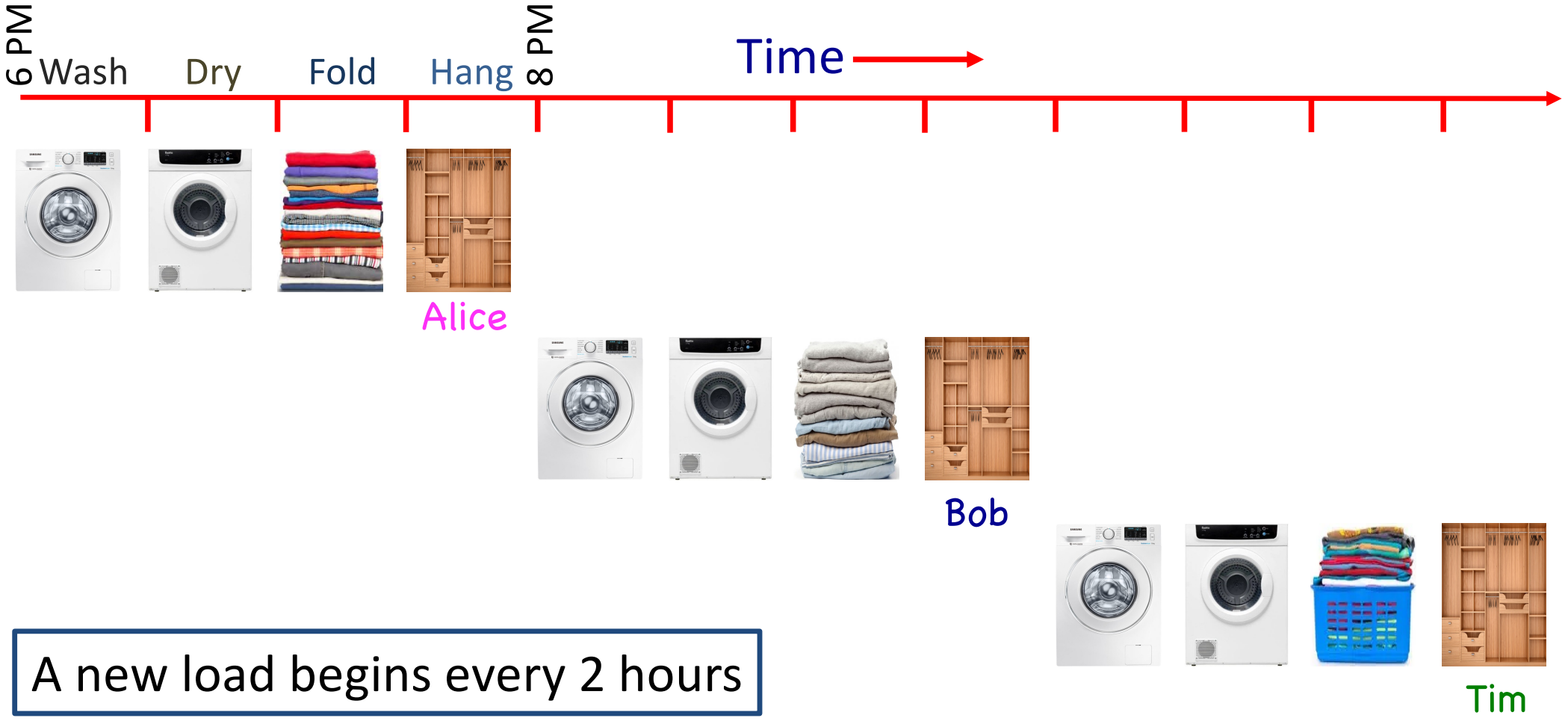
- The **6 instructions** took **10 cycles** to finish execution

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

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

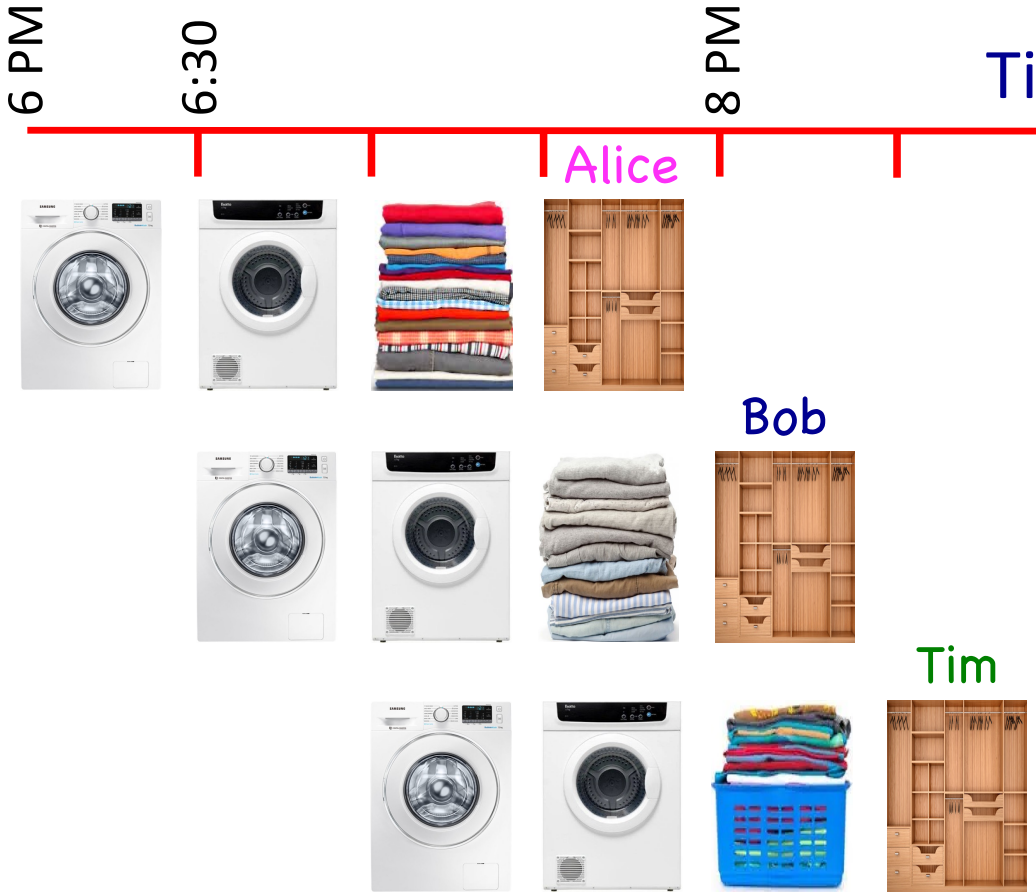
- Why is IPC less than 1?
  - It takes some time to fill and some time to drain the pipeline
    - During this time pipeline is operating below its potential

Sequential Laundry



A new load begins every 2 hours

Recall: Pipelined Laundry



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

Pipeline Idealism vs. Reality

- **Pipeline fill time:** The time it takes to fill the pipeline and make it operate at maximum efficiency

- **Pipeline drain time:** The time that is wasted when there is no more work to do in the pipeline

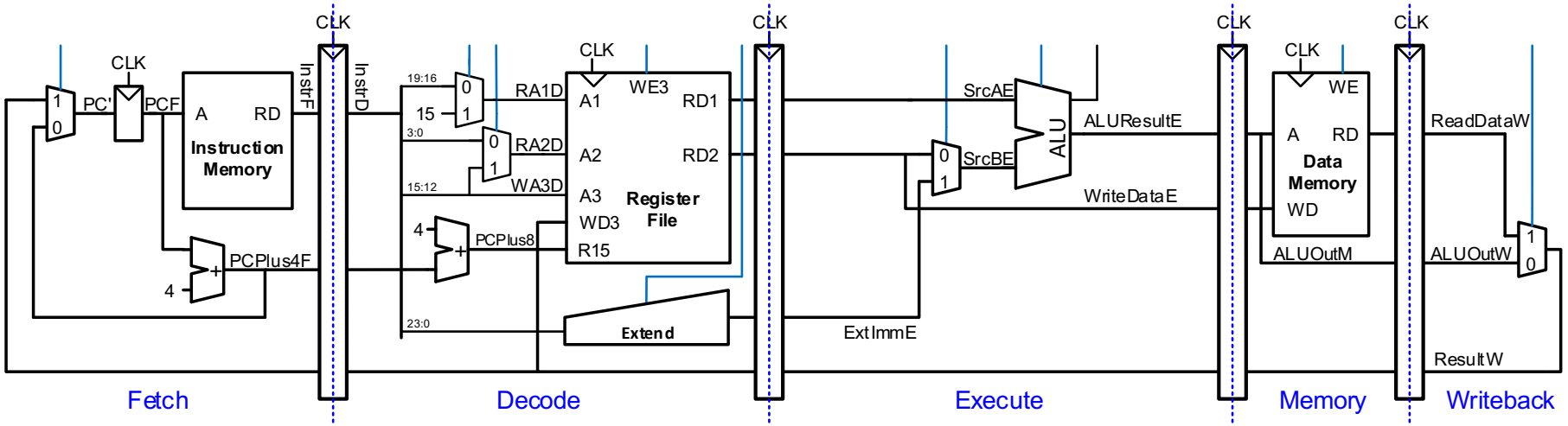
- The two factors limit the pipeline from delivering ideal speed-up
  - In the case when the amount of work is small relative to the number of stages in the pipeline

- Let's revisit the previous example

Performance Analysis

- The **6 instructions** took **10 cycles** to finish execution
- Cycles per Instruction (CPI) is : 10/6 = 1.66
  - Conversely, instruction per cycle (IPC) is: 0.6
- What if we have 1 billion instructions instead of 6?
  - CPI = (4 + 1000000000)/1000000000 = ~1
- Computer programs execute billions of instructions, so the overhead of filling/draining is amortized

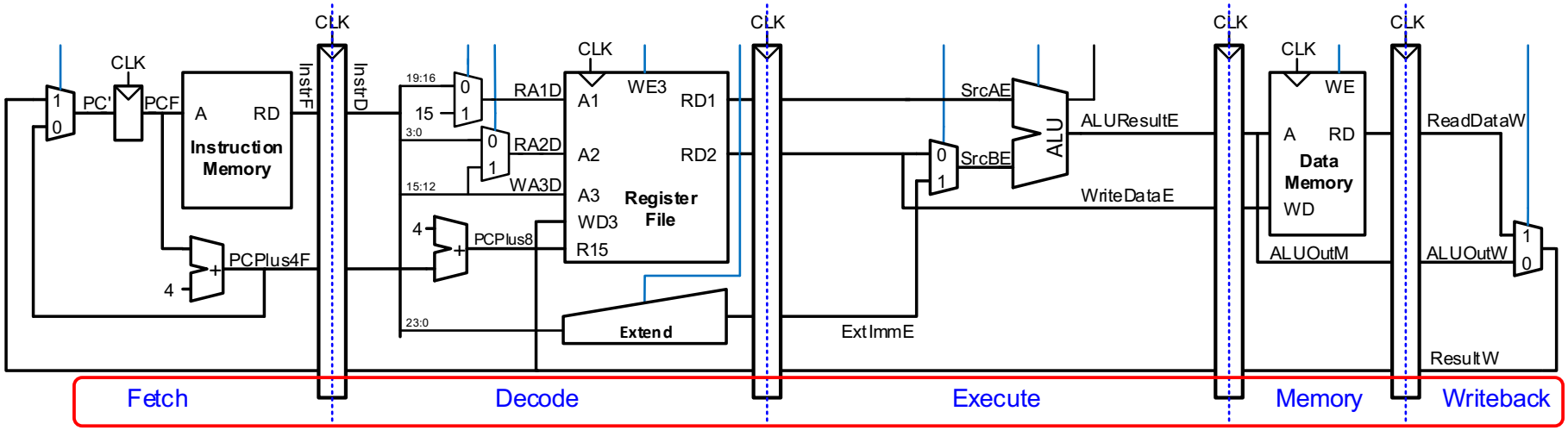
Bug in Pipelined Hardware!



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

We were Here.

Recap: Pipelined Data



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

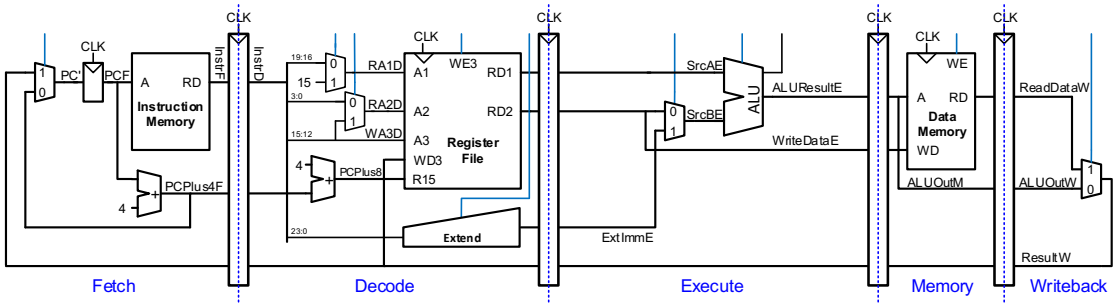
- **Think:** What is the width of each pipeline register?

Recap: Stages

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

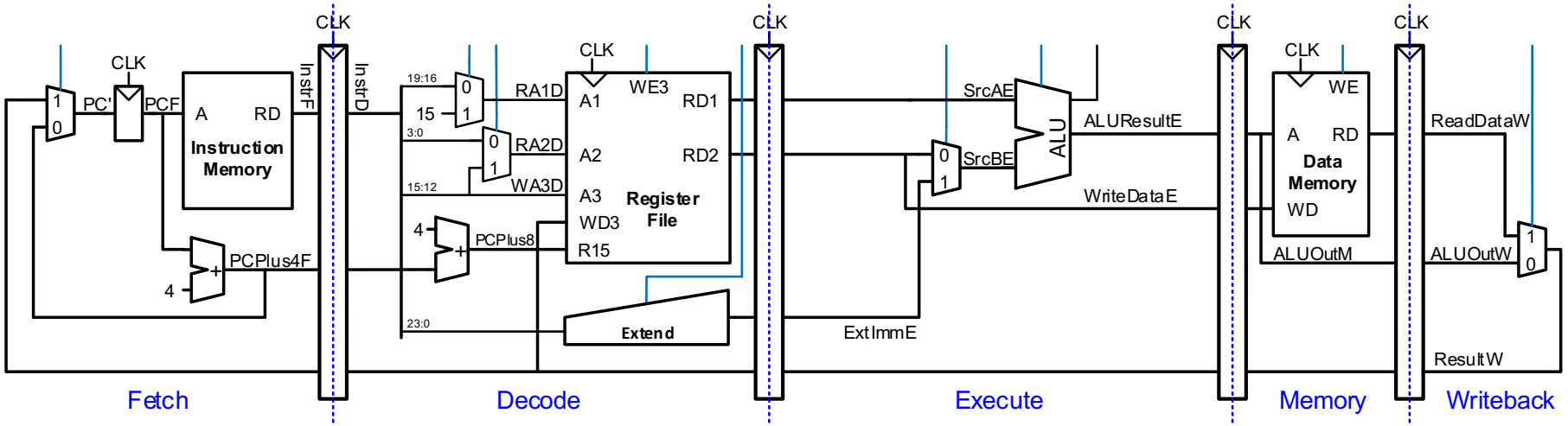
Recap: Pipeline Register Names

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



Bug in Pipelined Hardware!

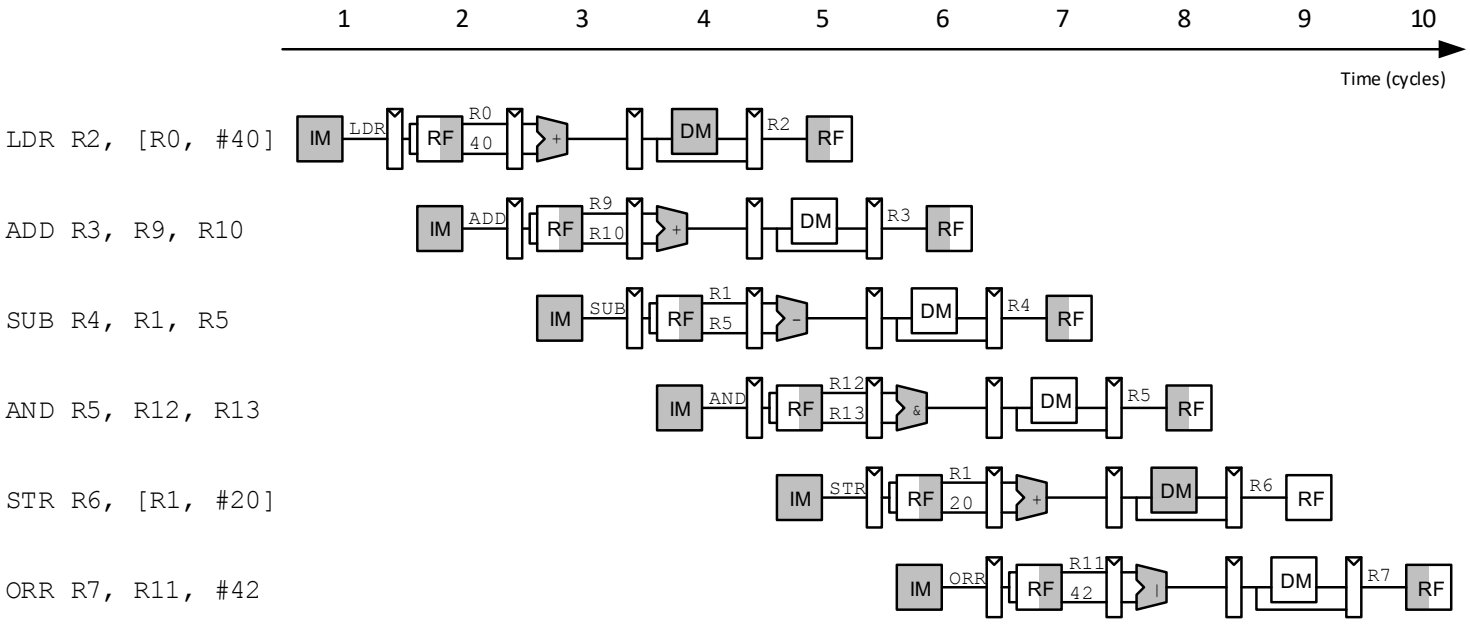
- The error is in the register file write logic that operates in the writeback stage



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

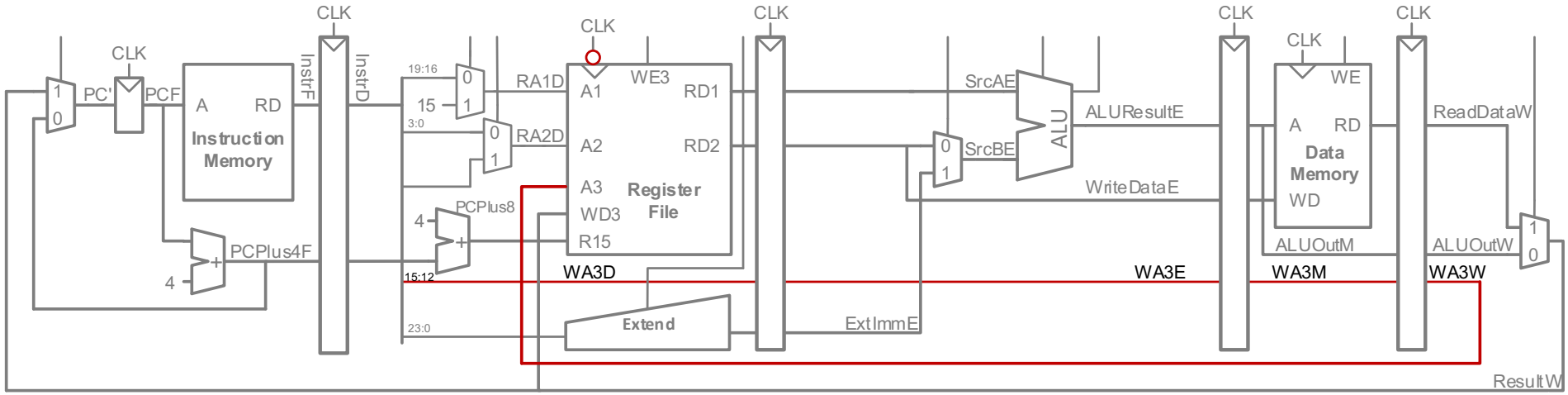
Bug in Pipelined Hardware!

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



Corrected Pipelined Datapath

- Here is the corrected pipelined datapath

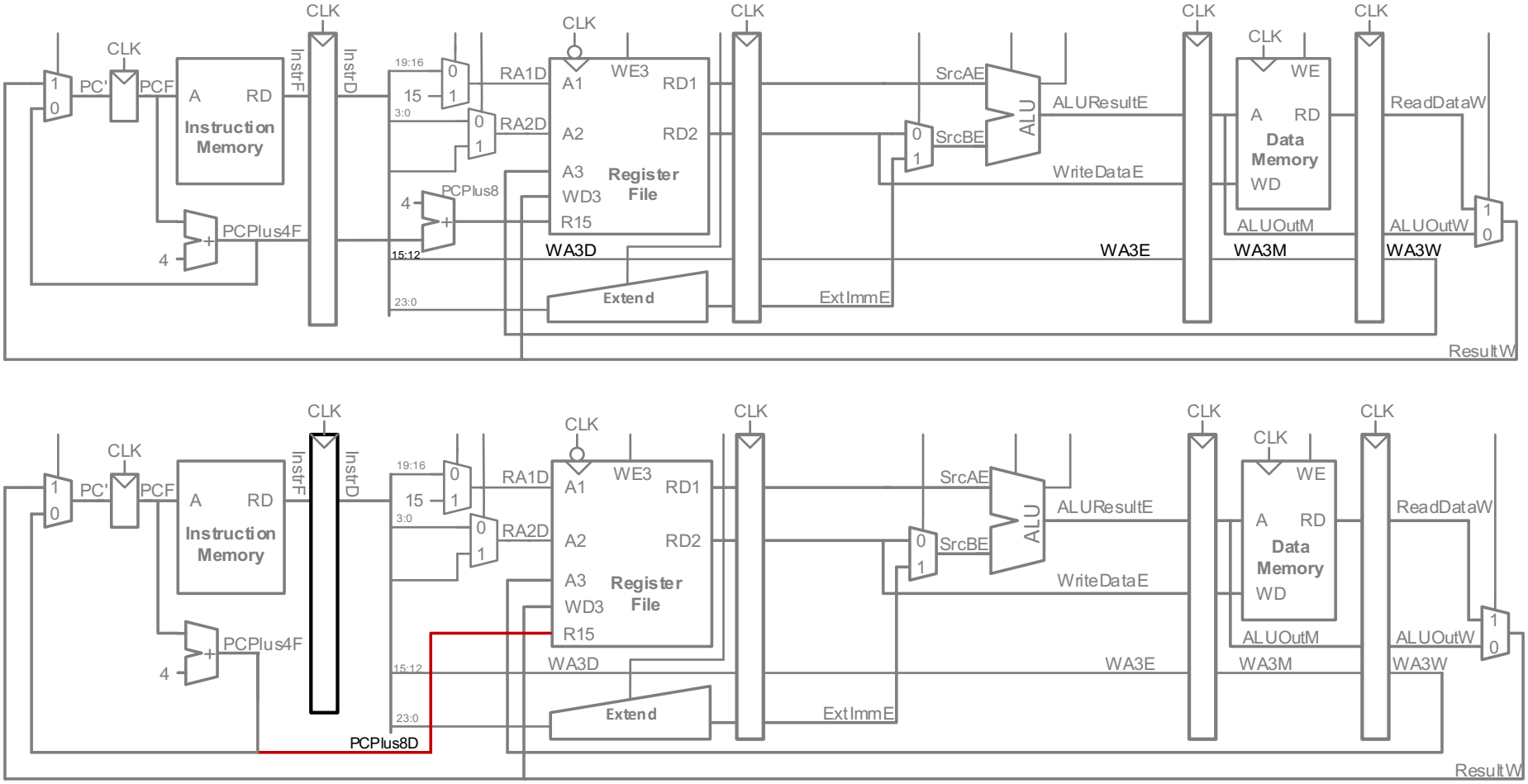


- The `WA3` signal is now pipelined along through the Execution, Memory, and Writeback stages so it remains sync with the rest of the instruction

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

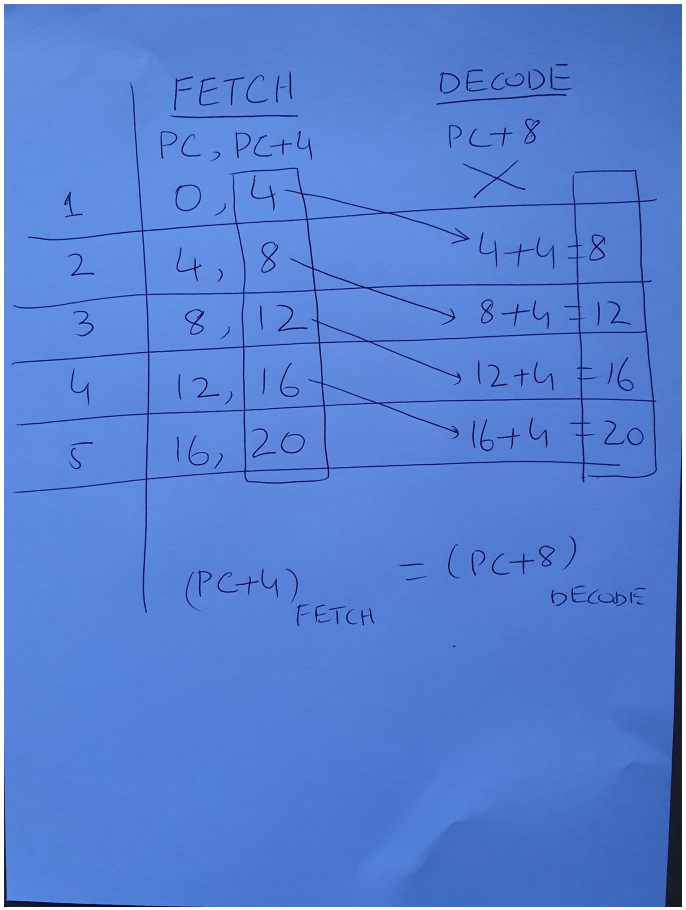
Optimized Pipelined Datapath

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



Optimized Pipelined Datapath

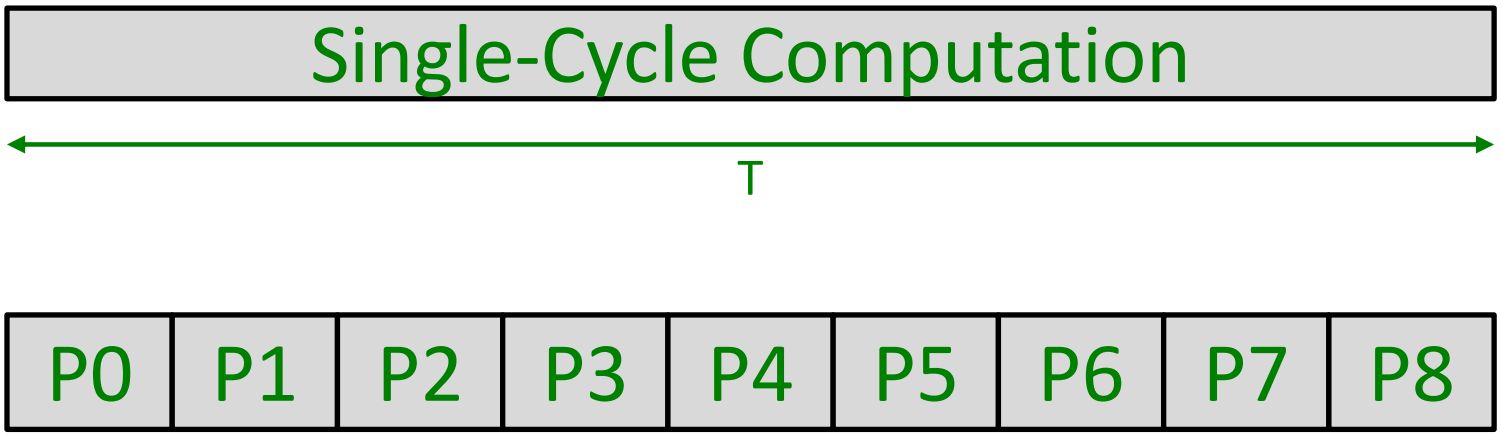
- Why the optimization works?



Balanced Pipeline (1)

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

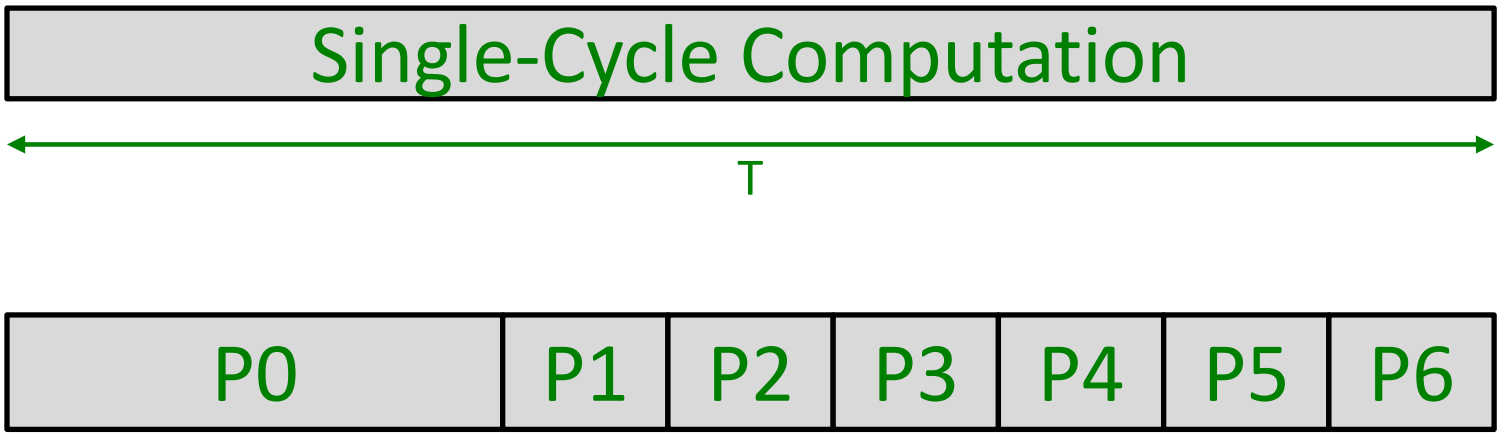
Balanced Pipeline (Example 1)



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

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

Balanced Pipeline (Example 2)



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

Control Unit for Pipelined uArch

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

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

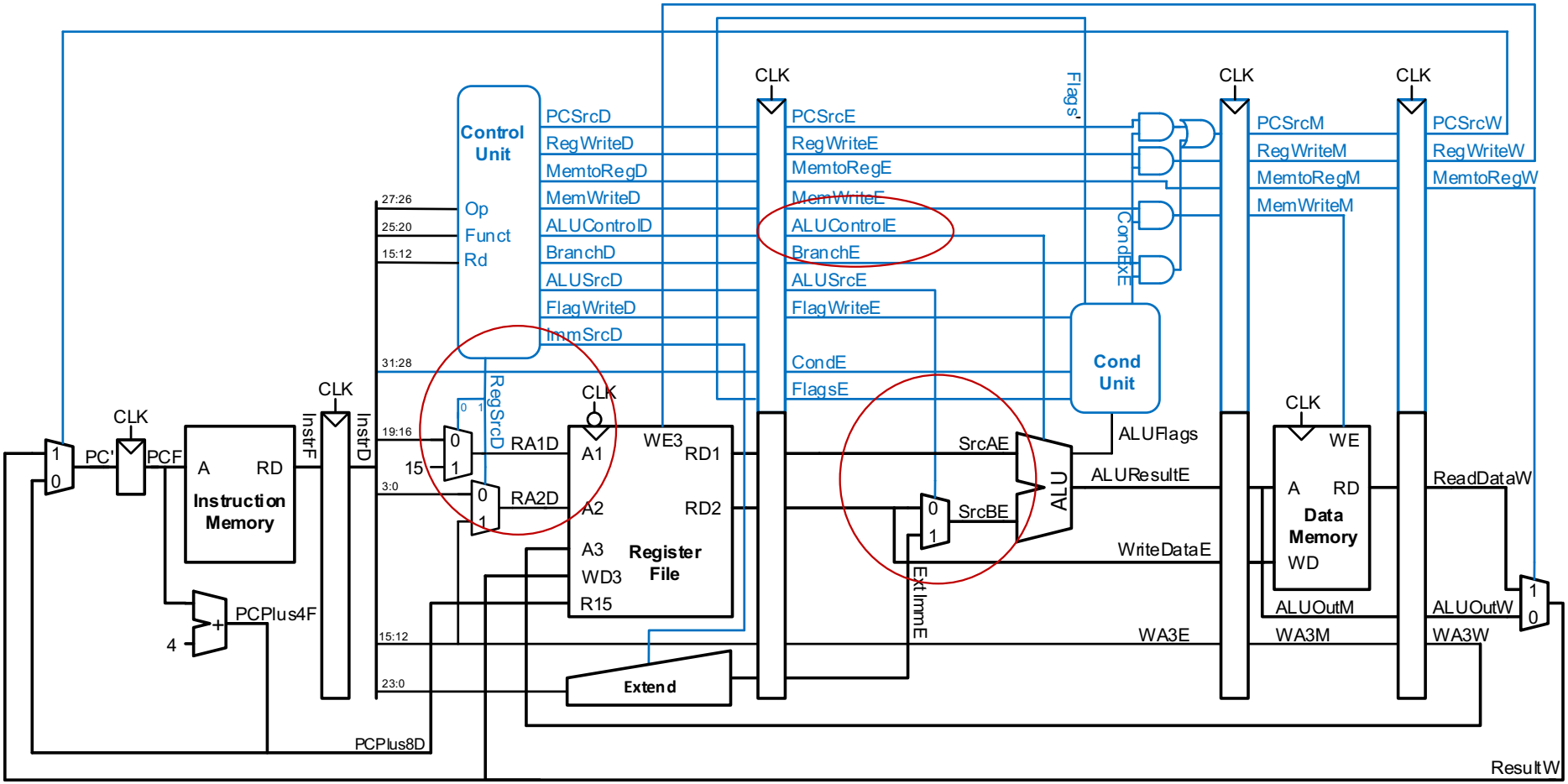
- These control signals must be pipelined along with the data

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

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

Pipelined Processor Control

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



Pipeline Hazards

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

- Hazards are a part of real life

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

Pipeline Hazards (Three Types)

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

Hazard Mitigation

- Hardware for concurrent instruction execution must deal with hazards

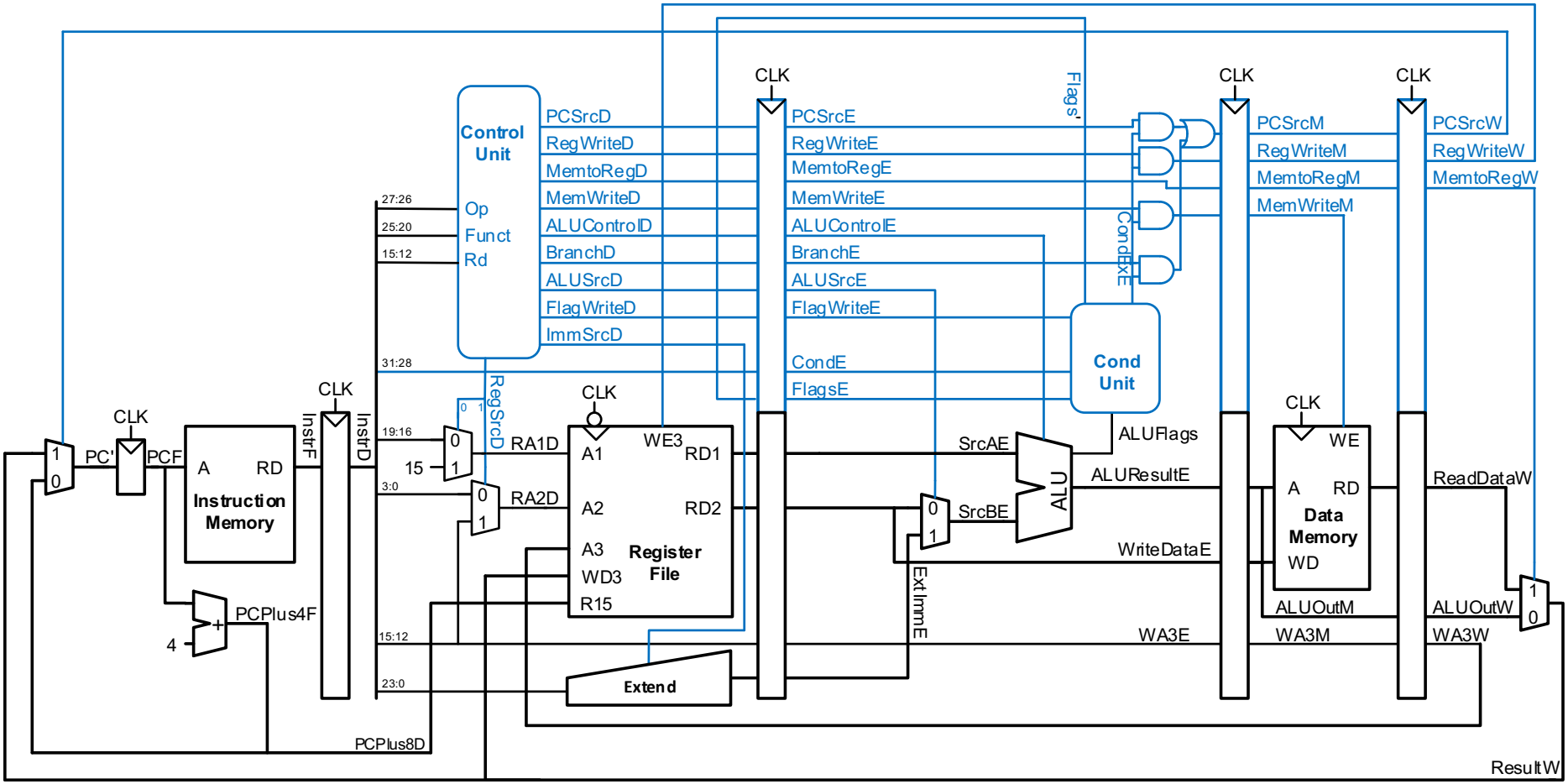
- From the processor's perspective:
  - Different solutions with different tradeoffs
    - Architectural state requires "serious" recovery
    - Architectural state is **untouched**, and hazard avoided
    - Lots of logic dedicated to hazard avoidance
    - Be very defensive (**stall the world** until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

Pipeline Hazards (Another View)

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

Pipeline Hazards (Another View)

- Identify backward flows (control and data)



Recall: Data Dependences (Week 7)

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

Read-After-Write Hazards

- True dependences lead to read-after-write hazards

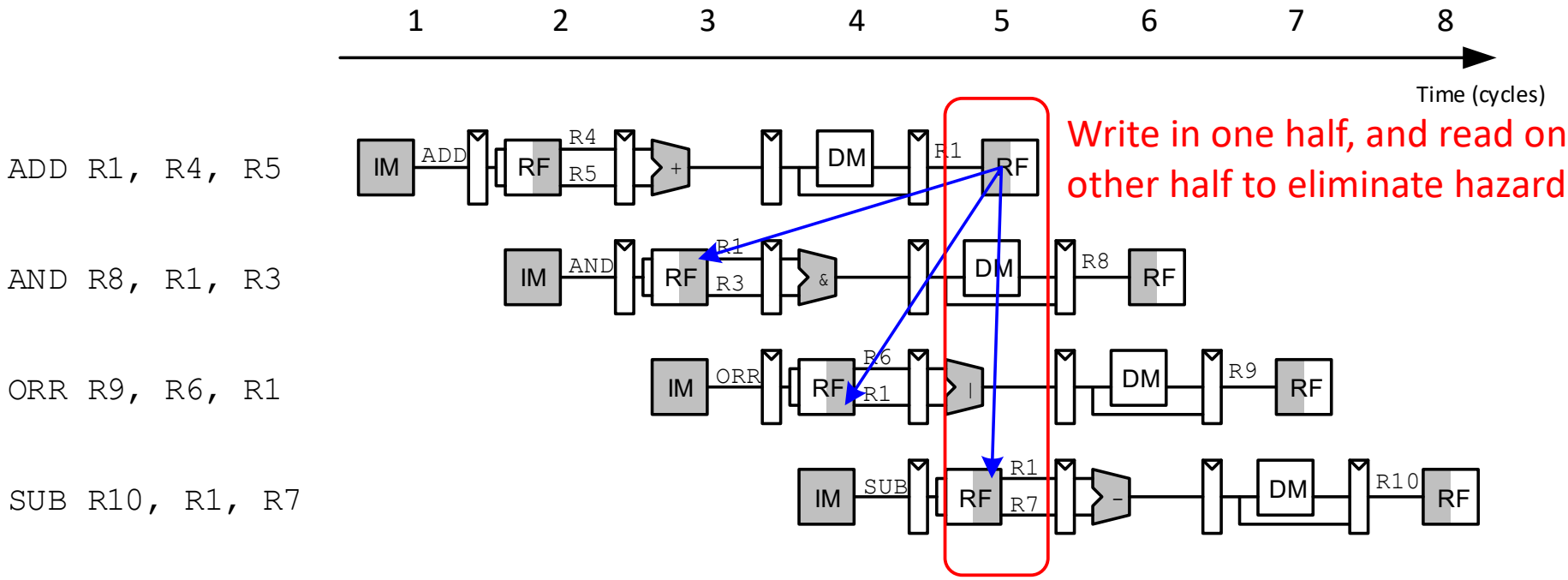
- **Think:** These hazards are not possible in a single-cycle microarchitecture

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

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

Pipeline Hazards (Example)

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



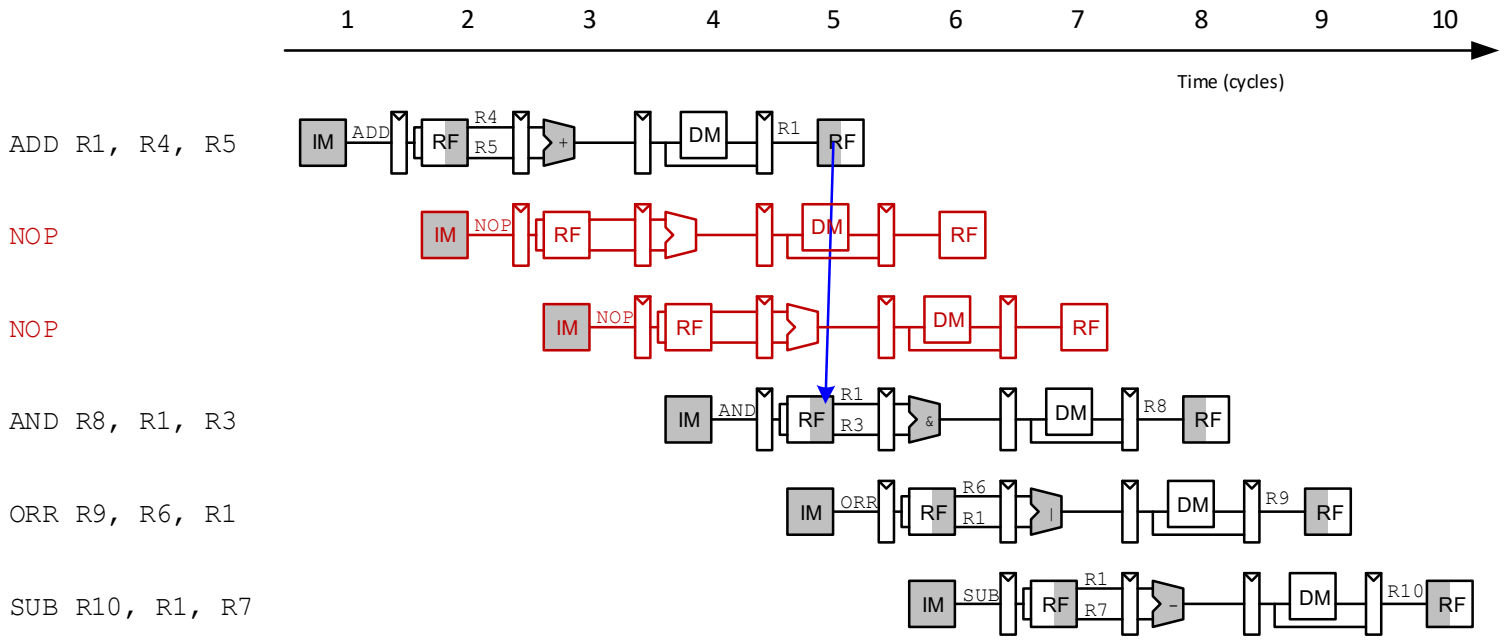
- Use a clever register read/write policy to eliminate one hazard
  - What can we do about the remaining two hazards?

Solution # 1: Software Interlocking

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

Example: Software Interlocking

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



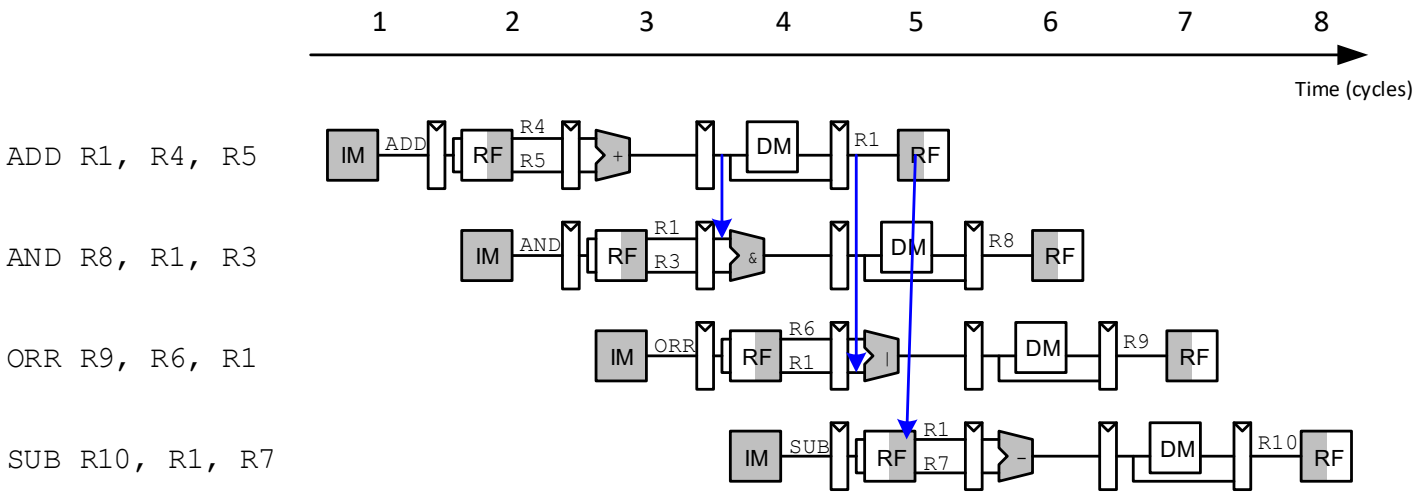
Solution # 2: Forwarding or Bypassing

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

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

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

Forwarding Example

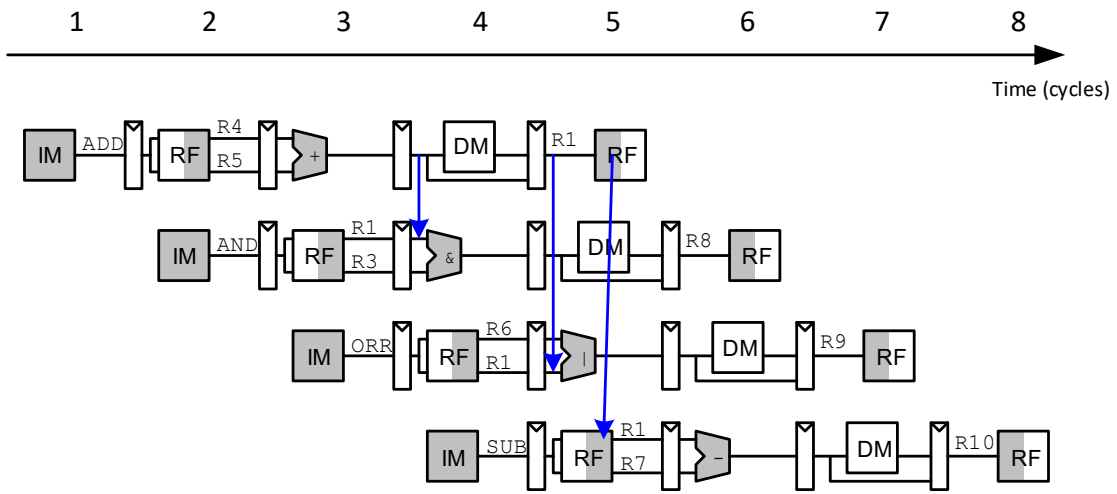


- Sum from the ADD instruction is computed by ALU in **cycle 3** and is not strictly needed by the AND instruction until the ALU uses it in **cycle 4**
- No need to wait for the results to appear in register file

Forwarding Example

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

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



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

Necessary Conditions for Forwarding

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

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

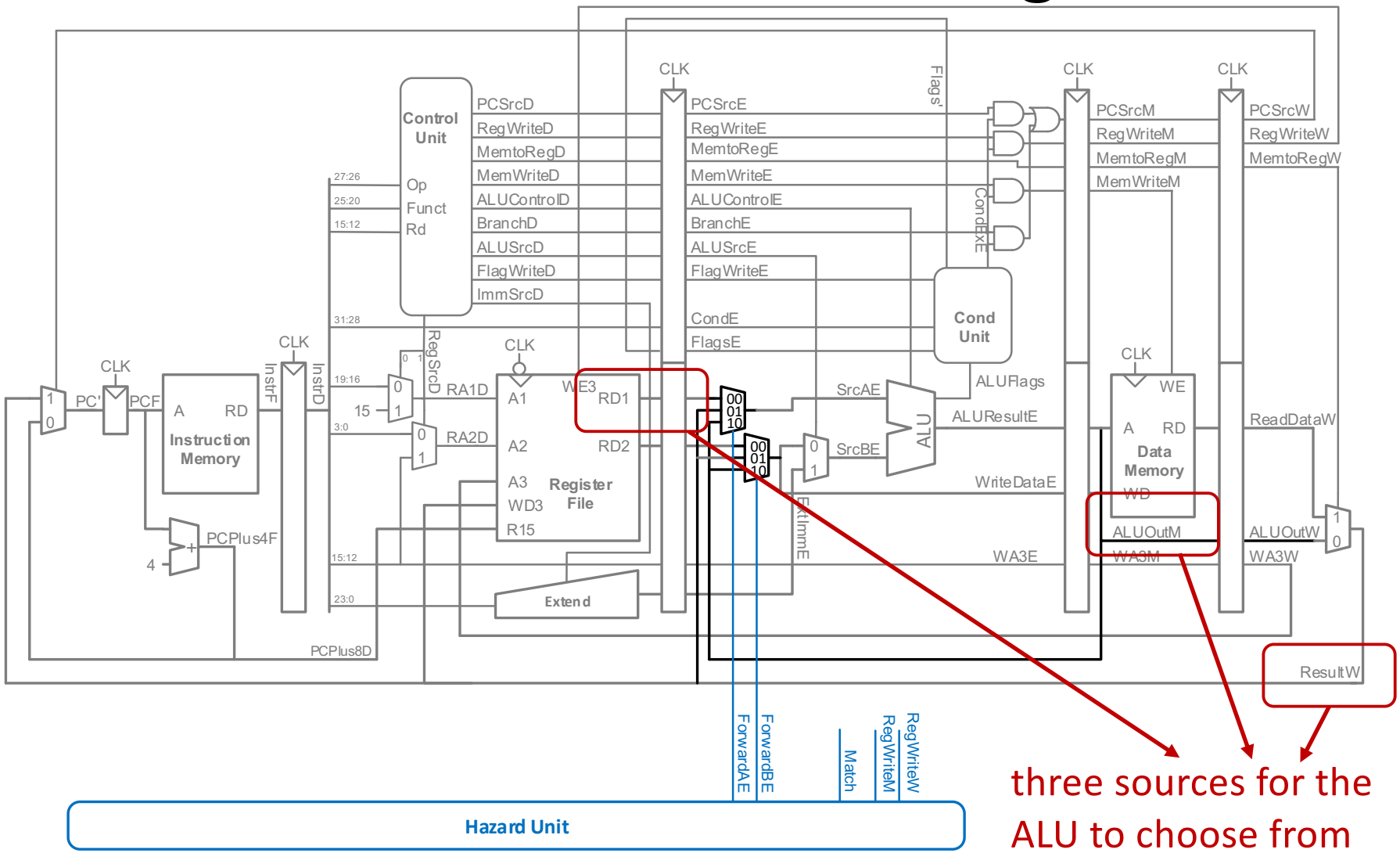
Necessary Conditions for Forwarding

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

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

ForwardBE same but with Match2E

Pipelined Processor with Forwarding



three sources for the ALU to choose from

Coming Attractions

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

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

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

Load-Use Hazard

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

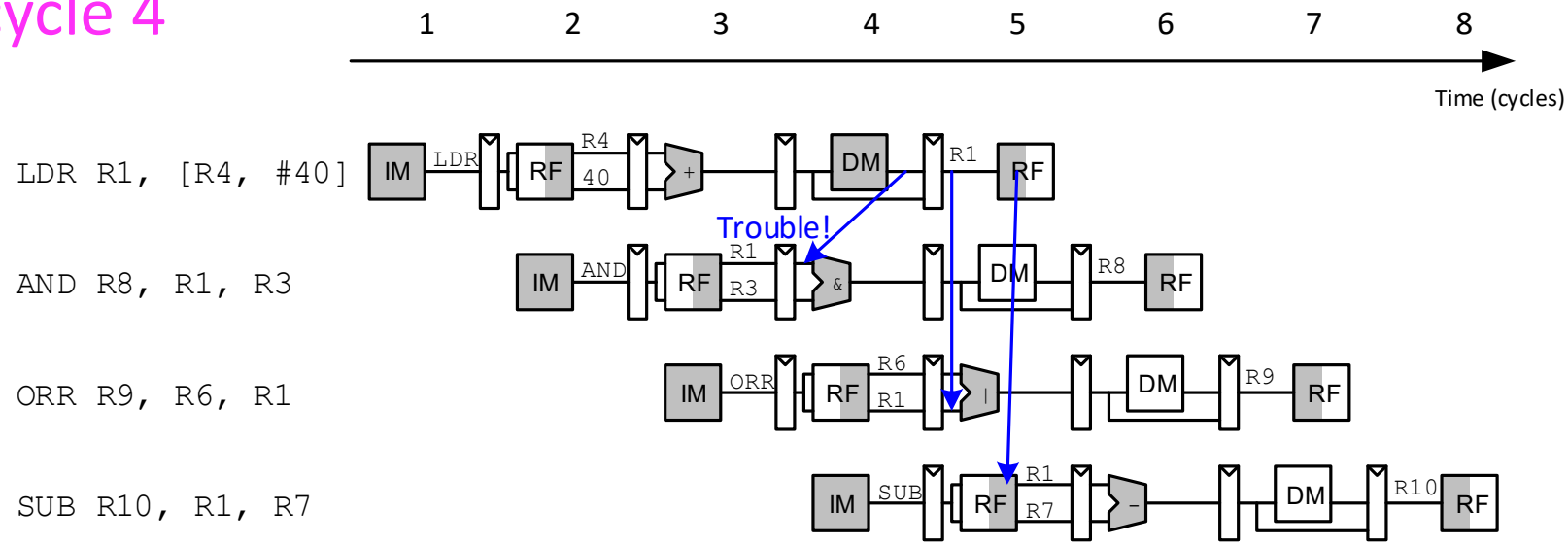
- `LDR` does not finish reading data until the end of the **MEM stage**
  - The result cannot be forwarded to the **EX-stage** of the next instruction
  - We call Load followed by its use a Load-Use hazard

- Load-Use hazard cannot be solved with forwarding

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

Load-Use Hazard

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



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

Stalls to Resolve Load-Use Hazards

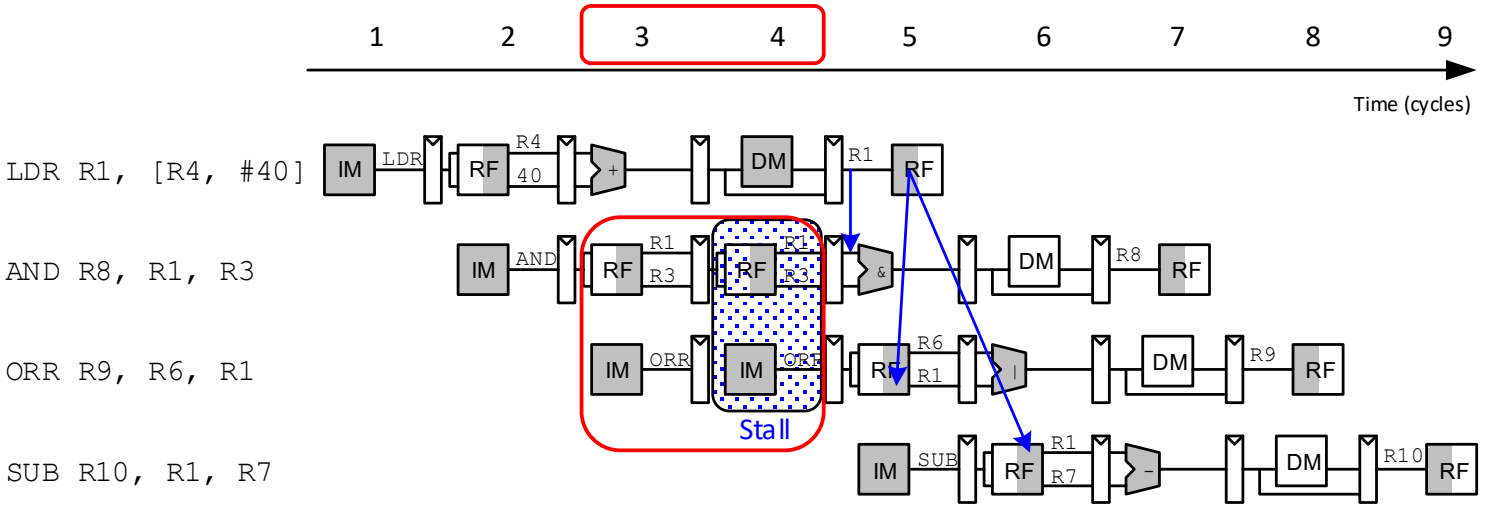
- The dependent instruction can be detected as the "user" of `LDR` in the **Decode stage**

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

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

Stalls to Resolve Load-Use Hazards

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



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

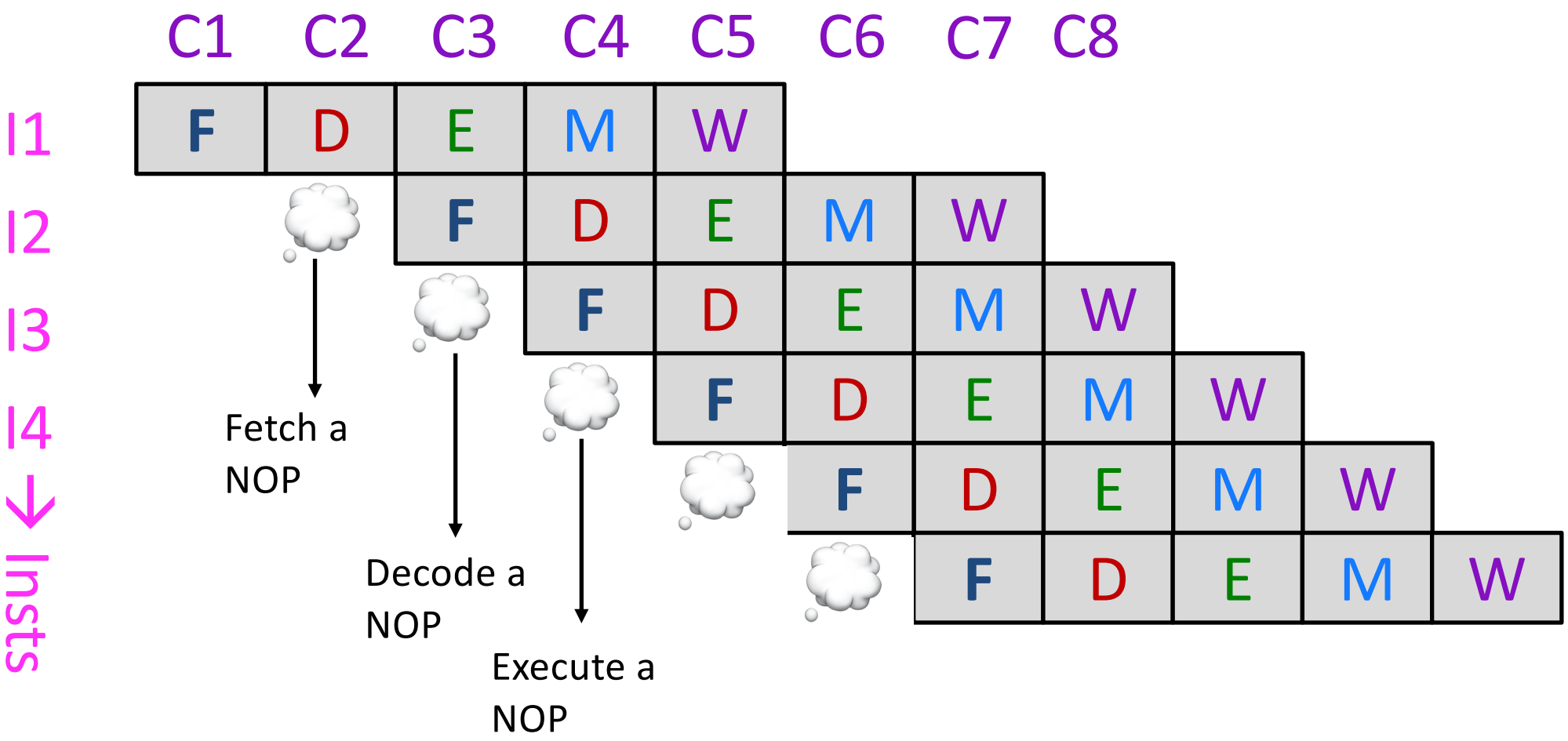
Pipeline Bubbles

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

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

- It behaves like a NOP instruction

Visualizing Bubbles



Implementing Stalls

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

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

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

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

Logic to Compute Stalls and Flushes

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

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

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

ldrstall = *Match_12D_E* • *MemtoRegE*

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

Pipelined CPU with Stalls to Solve Load-Use Hazard

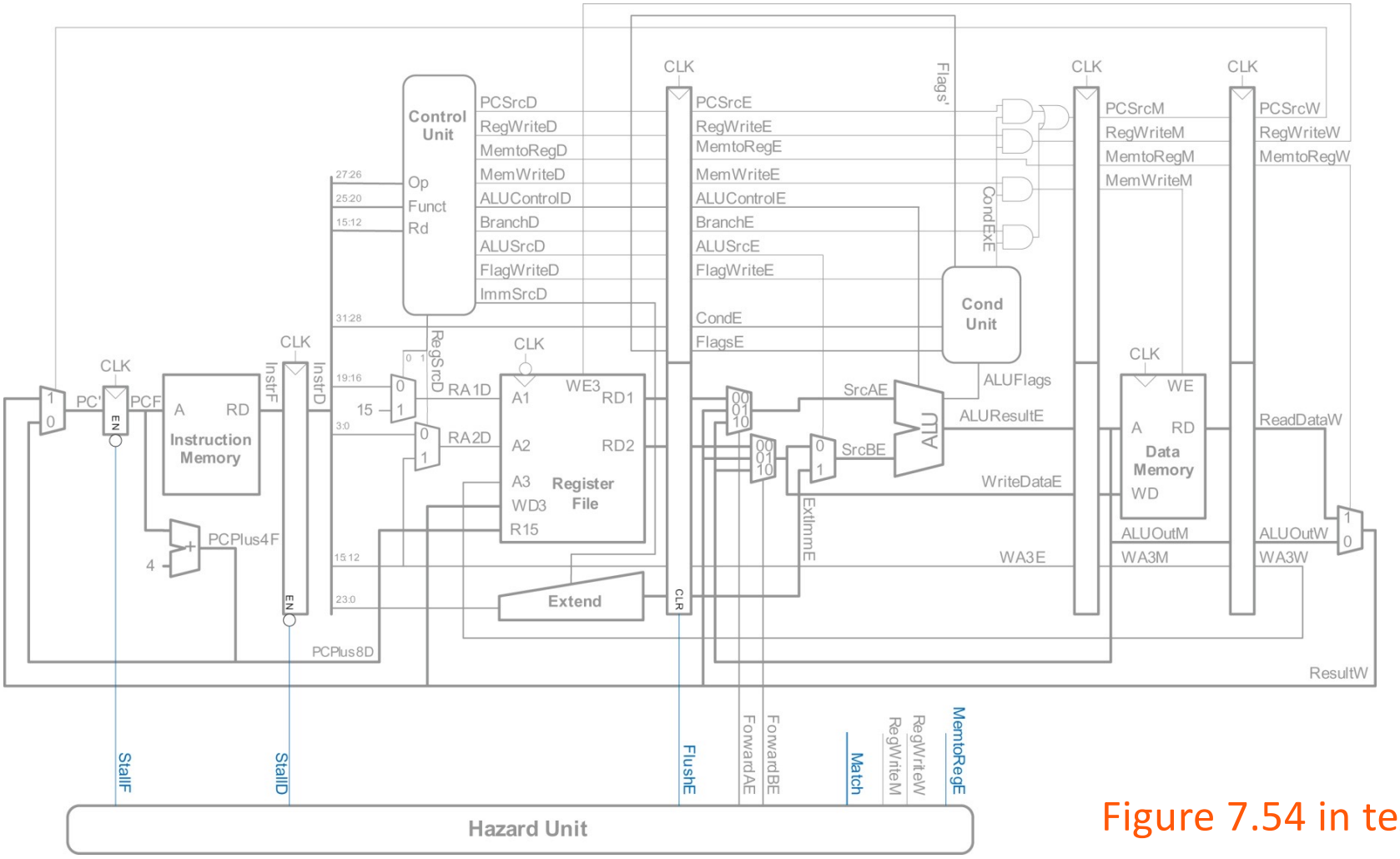


Figure 7.54 in textbook

Control Hazards

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

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

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

Solving Control Hazards

- There are two solutions

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

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

Simplest Branch Predictor

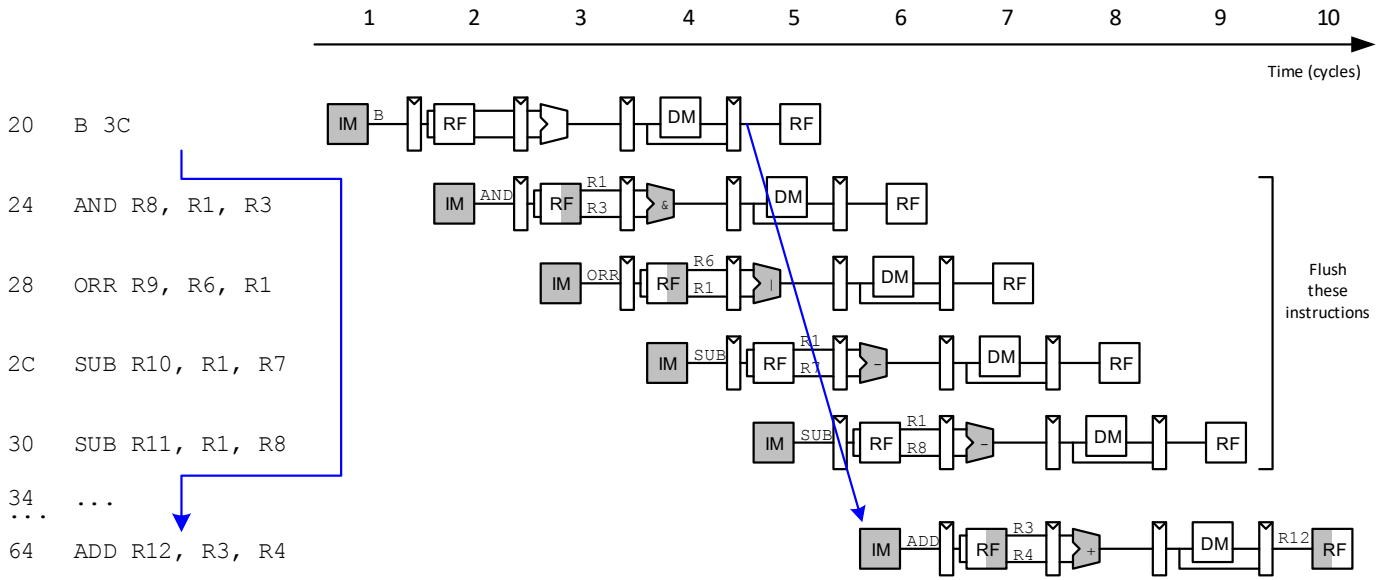
- Predict-always-untaken (eager)

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

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

Flushing when Branch is Taken

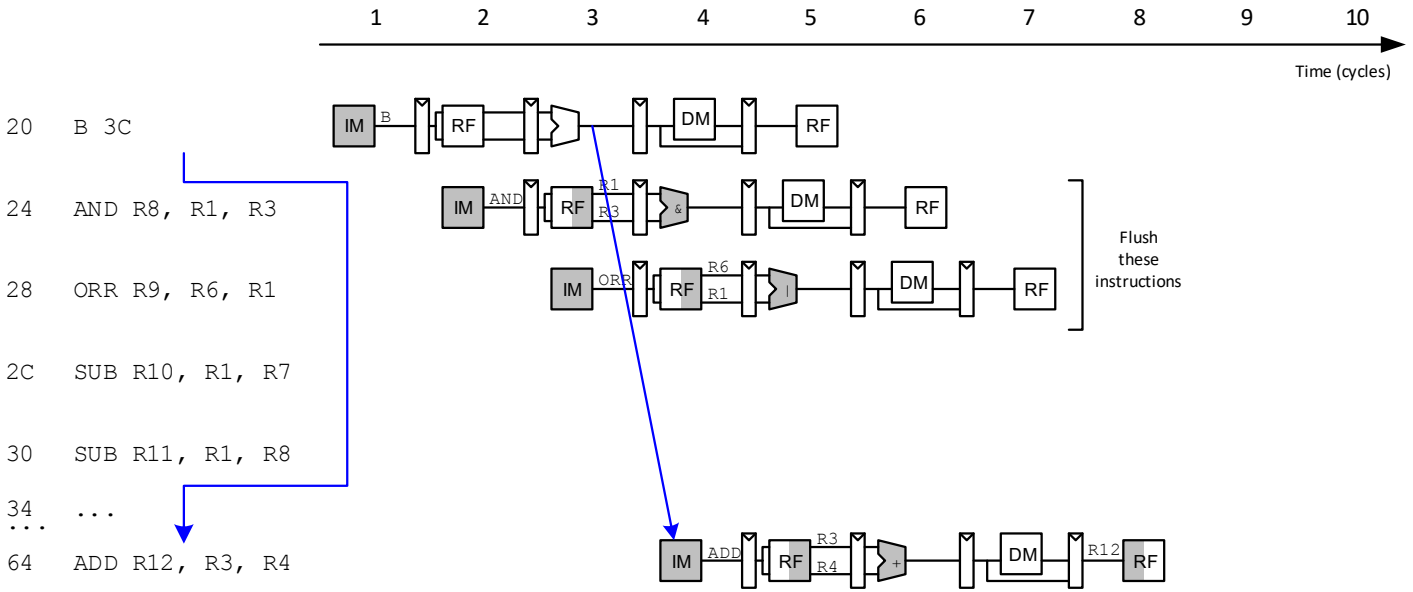
- Fetching the next instruction is an example of predict-always-untaken



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

Early Branch Resolution

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



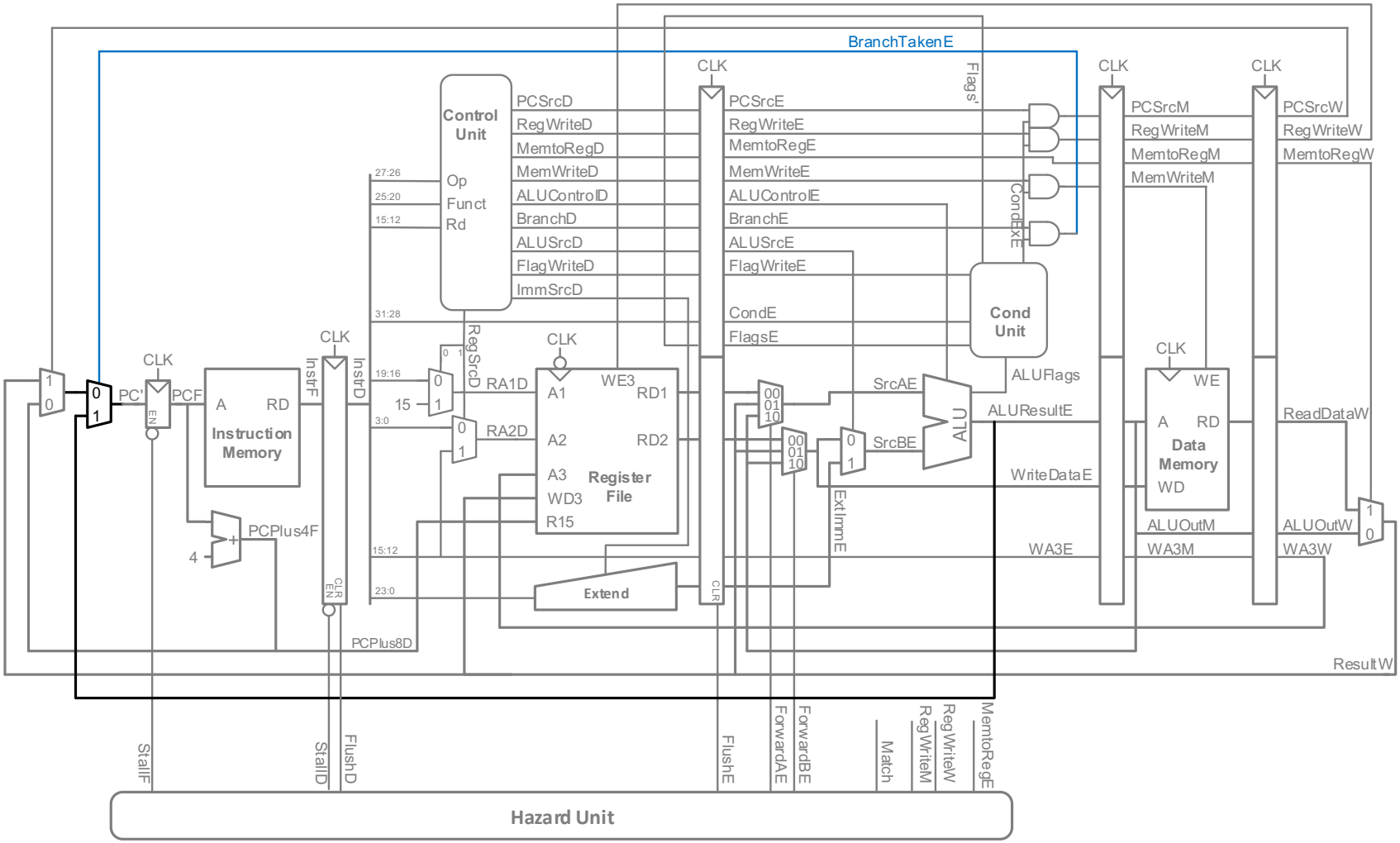
- Flush the two instructions in the **F** and **D** stages

Hardware Changes for Early Resolution

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

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

Pipelined Processor with Early BTA



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

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

  PCWrPendingF = PCSrcD + PCSrcE + PCSrcM

  PC write is in progress in D, E, M

- **Stall Fetch** if *PCWrPendingF*

  StallF = ldrStallD + PCWrPendingF

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

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

  FlushD = PCWrPendingF + PCSrcW + BranchTakenE

- **Flush Execute** if branch is taken

  FlushE = ldrStallD + BranchTakenE

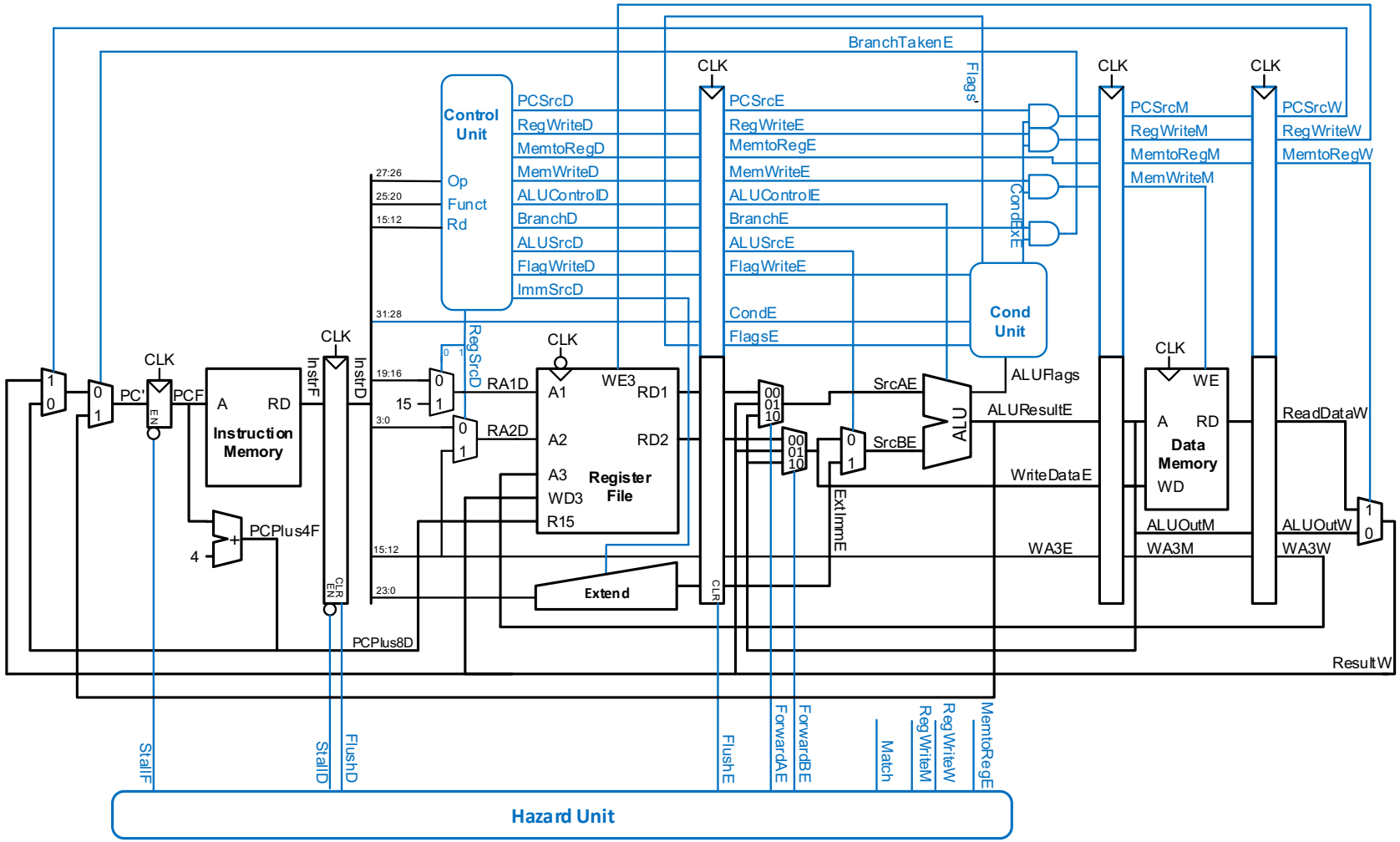
  Flush D if PC write in progress in D, E, M, or W, or branch taken in E

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

  StallD = ldrStallD

  Stall Decode if LDR-Use hazard

ARM Processor with Full Hazard Handling

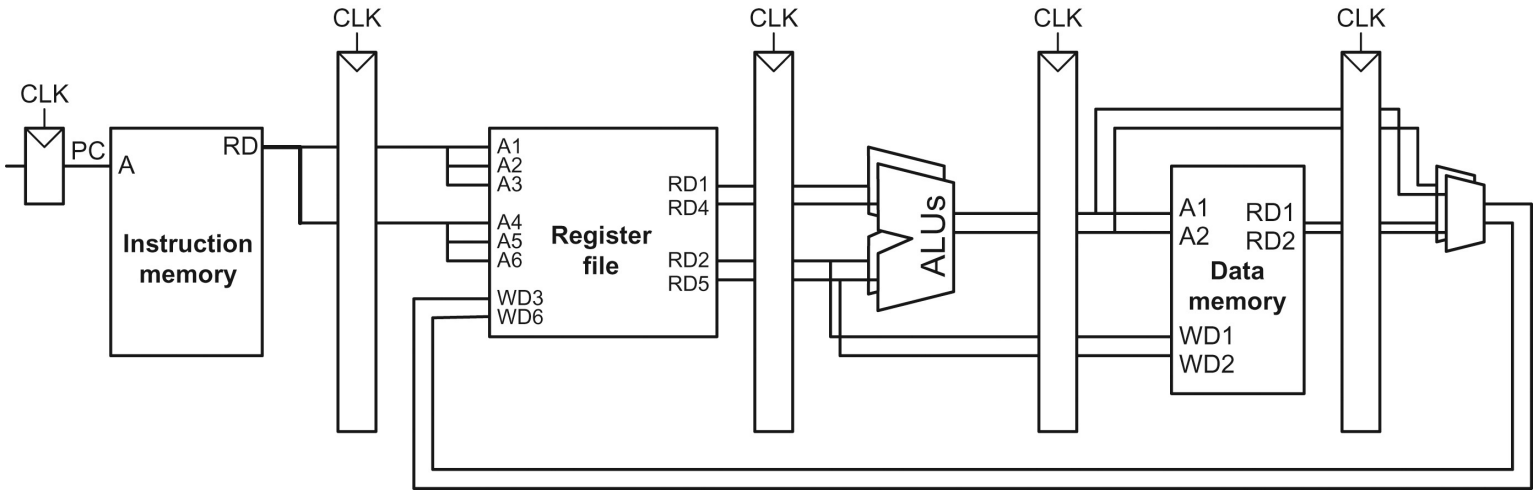


Superscalar Processor

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

Superscalar: Idea and Datapath

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

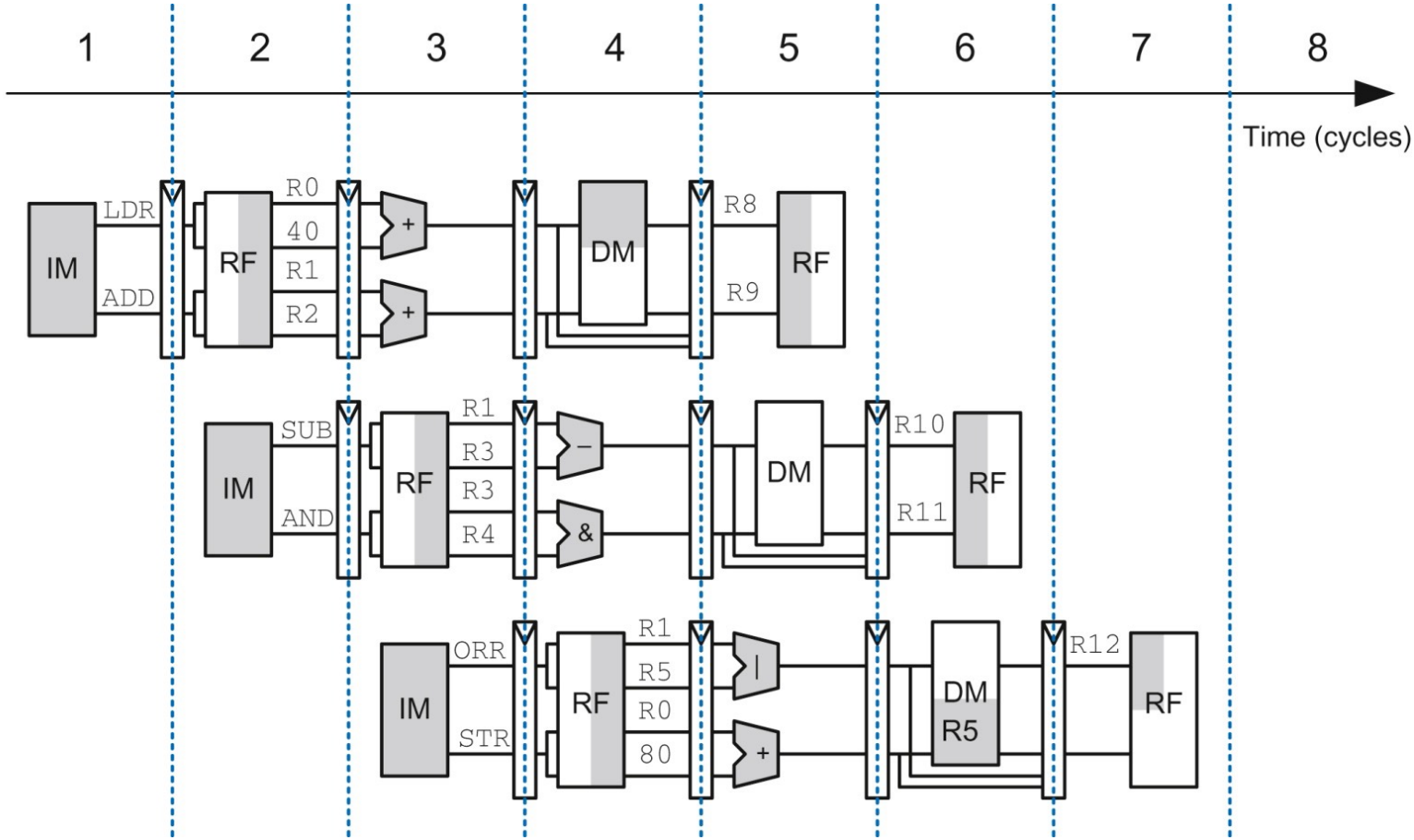


- Requires 6-ported register file (4 reads, 2 writes), 2 ALUs, 2-ported data memory
- Ideal CPI = 0.5 and IPC = 2
- Dependencies and hazards inhibit ideal IPC
- Above figure does not show forwarding and hazard detection logic

Superscalar: Pipeline Operation

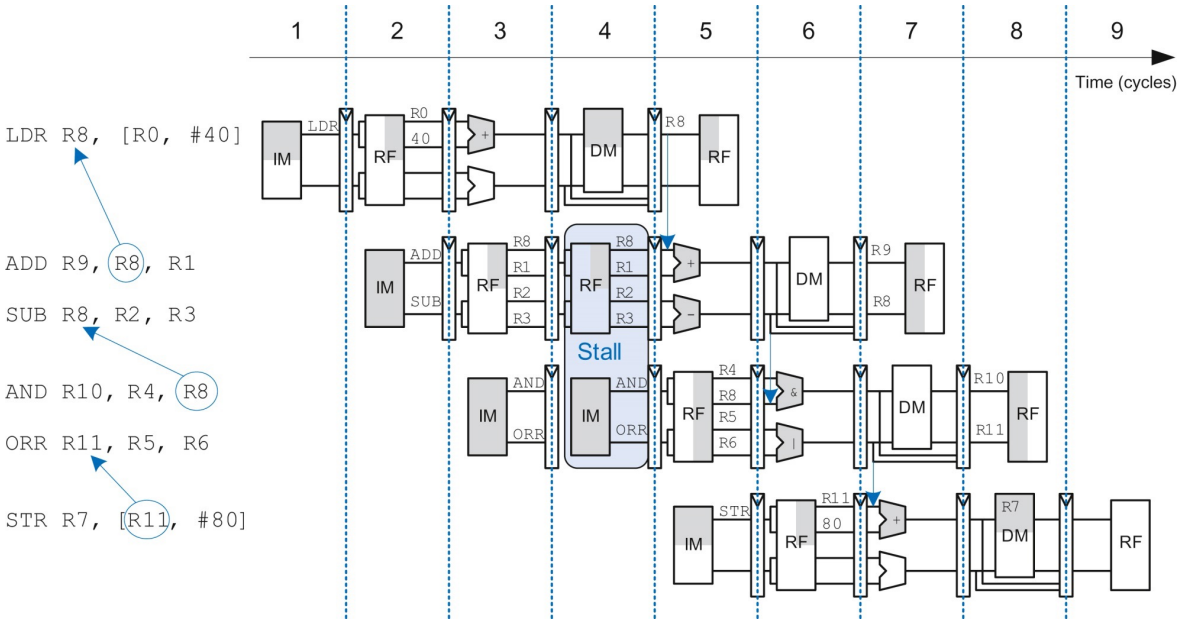
- Example program where IPC = 2 is possible

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

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

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



Superscalar: Impact of Dependencies

- Example of program with data dependences



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

In-Order Superscalar: Tradeoffs

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

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

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

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

We were Here.

Recap: 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 see some examples

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

Stall in the Fetch stage

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

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 Fetch stage until writeback stage has the new PC
  - 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 EX, flush Fetch and Decode PPR

Recap: How to 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

Recap: Final Conditions for Hazard Detection

- ***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

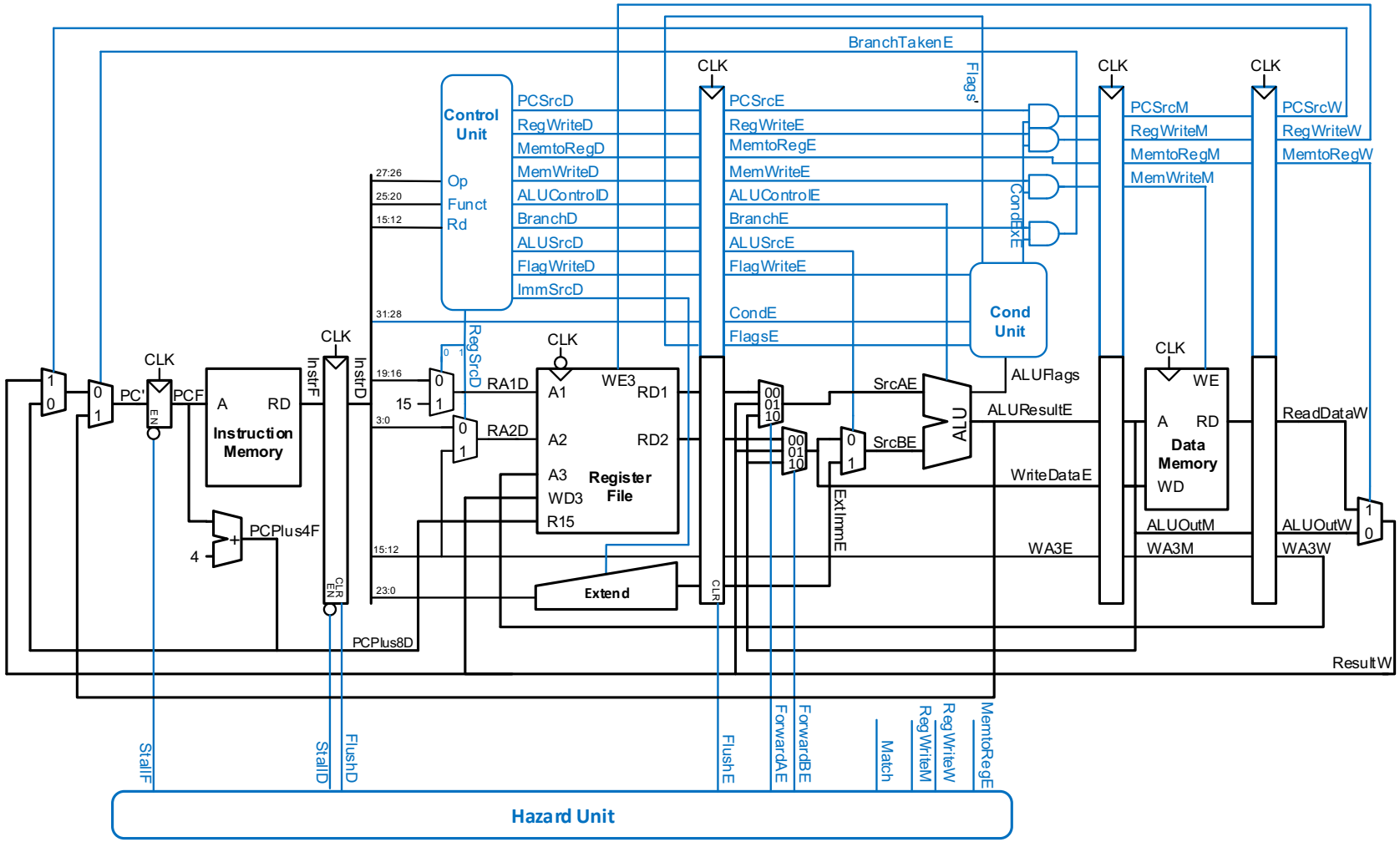
  Flush D if PC write in progress in D, E, M, or W, or branch taken in E

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

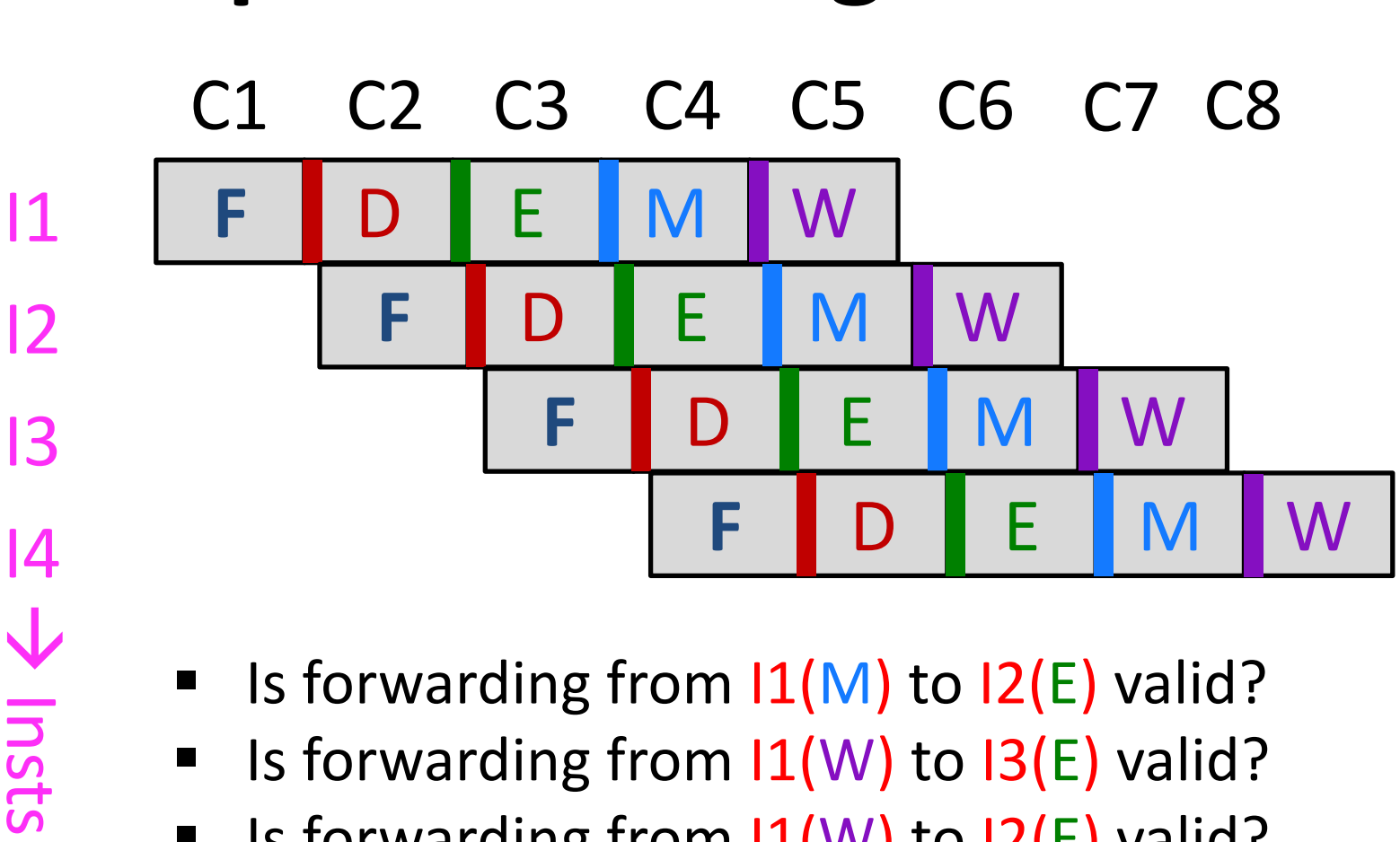
  StallD = ldrStallD

  Stall Decode if LDR-Use hazard

Recall: ARM Processor with Full Hazard Handling



Recap: Forwarding



PPR Code

D

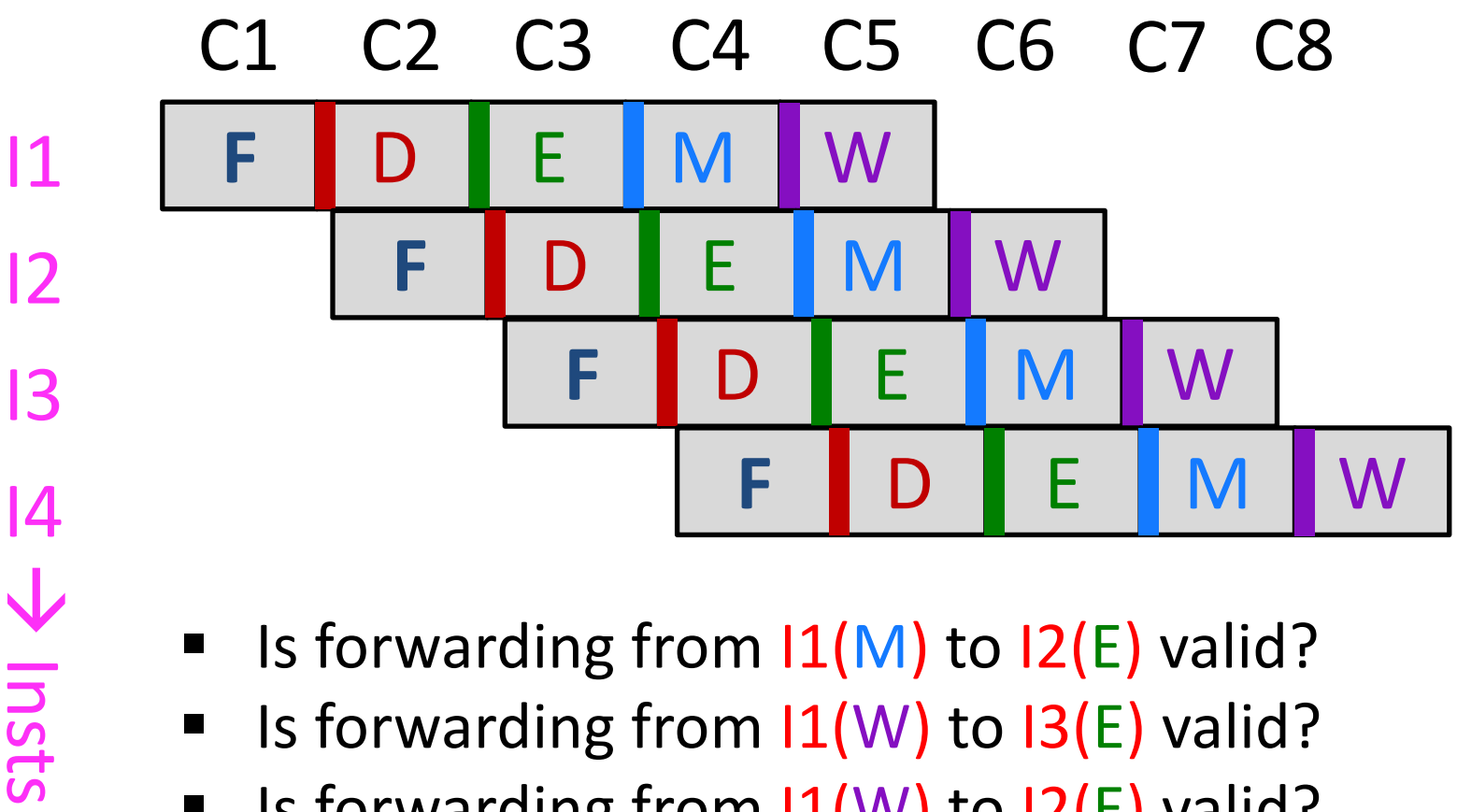
E

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 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?

Branch Prediction

- The cost of a branch misprediction (branch misprediction penalty) increases for superscalars
- Lots of wasted effort flushed
- Need more accurate branch predictors

Dynamic Branch Prediction





- Fork in the road in all cases
  - But, context is different
- **What will you do to go at full-speed?**
  - **Static prediction:** always-left, always-right, return if wrong
  - **Dynamic prediction:** memorize <context-direction> pair in head: <cherry tree, right>, <windmills, left>

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)

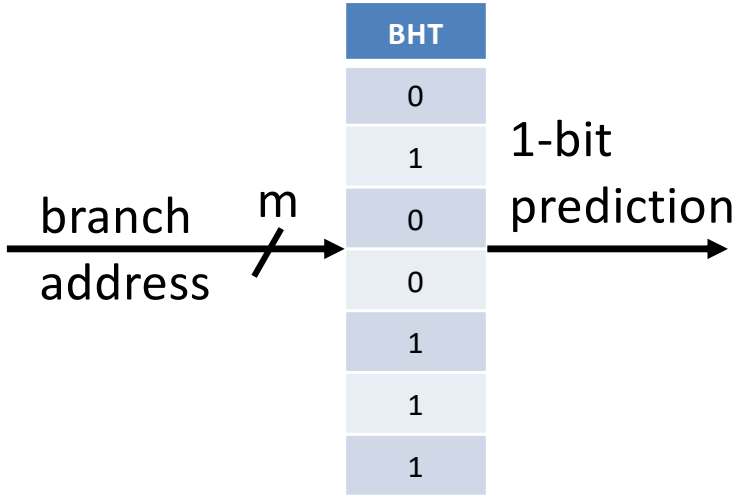
Branch Identification & Behavior

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

- **Branch behavior**
  - Outcome of the **condition test** from ALU
  - Can also 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: Predict/compute target address and fetch target

- Outcome matches prediction: **Noting to do**
- Outcome does not match prediction:
  - Flip the entry in the BHT
  - Flush the pipeline (EX, ID, IF), update PC

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

Accuracy/Perf of 1-bit Predictor

Consider the following loop:

```
for (i = 1; i < n; i++)
{
        do something;
}
```

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

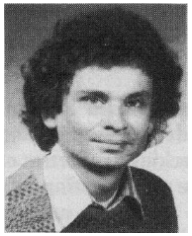
- What is the prediction accuracy of a 1-bit branch predictor if n is equal to 10?
  - Accuracy is 80% for a branch that is taken 90% of the time

Anomalous Decision

- When branches that are strongly biased toward one direction suddenly takes a different path/direction
  - Branches at the end of a loop are usually taken (**backward**) except for the case of the loop exit (**forward**)

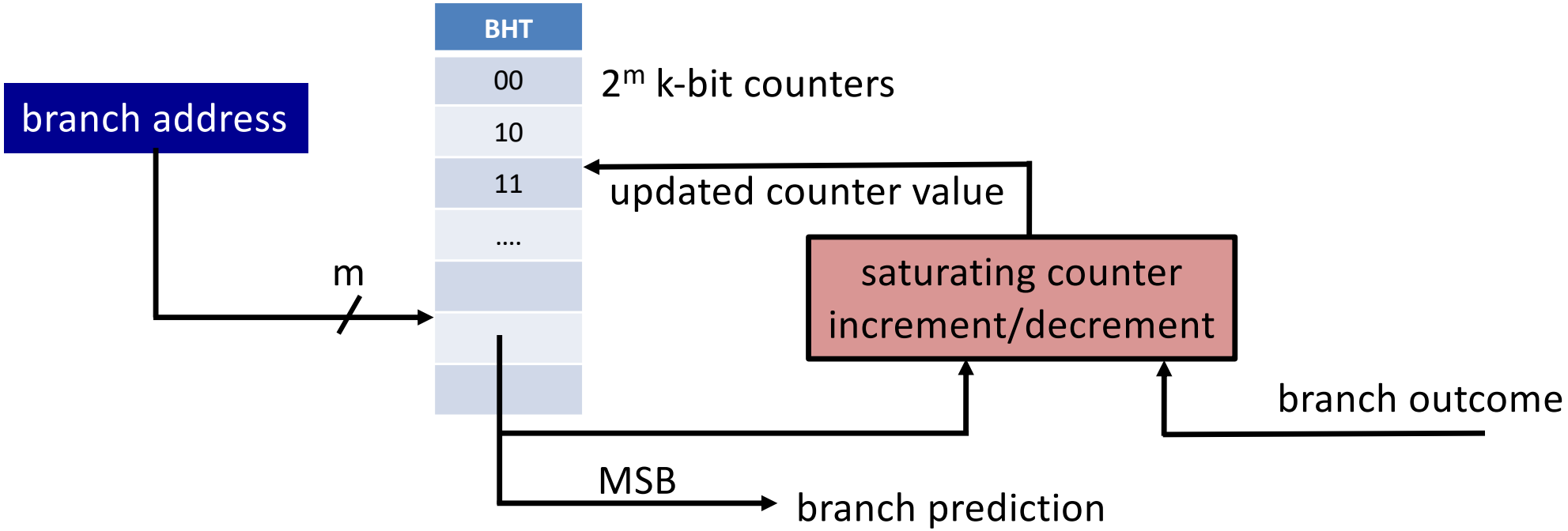
- A 1-bit predictor is "thrown off" by a single anomolous decision

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



k = 2

- A saturating counter maps the outcomes of several recent branches on to a **counter** with different states

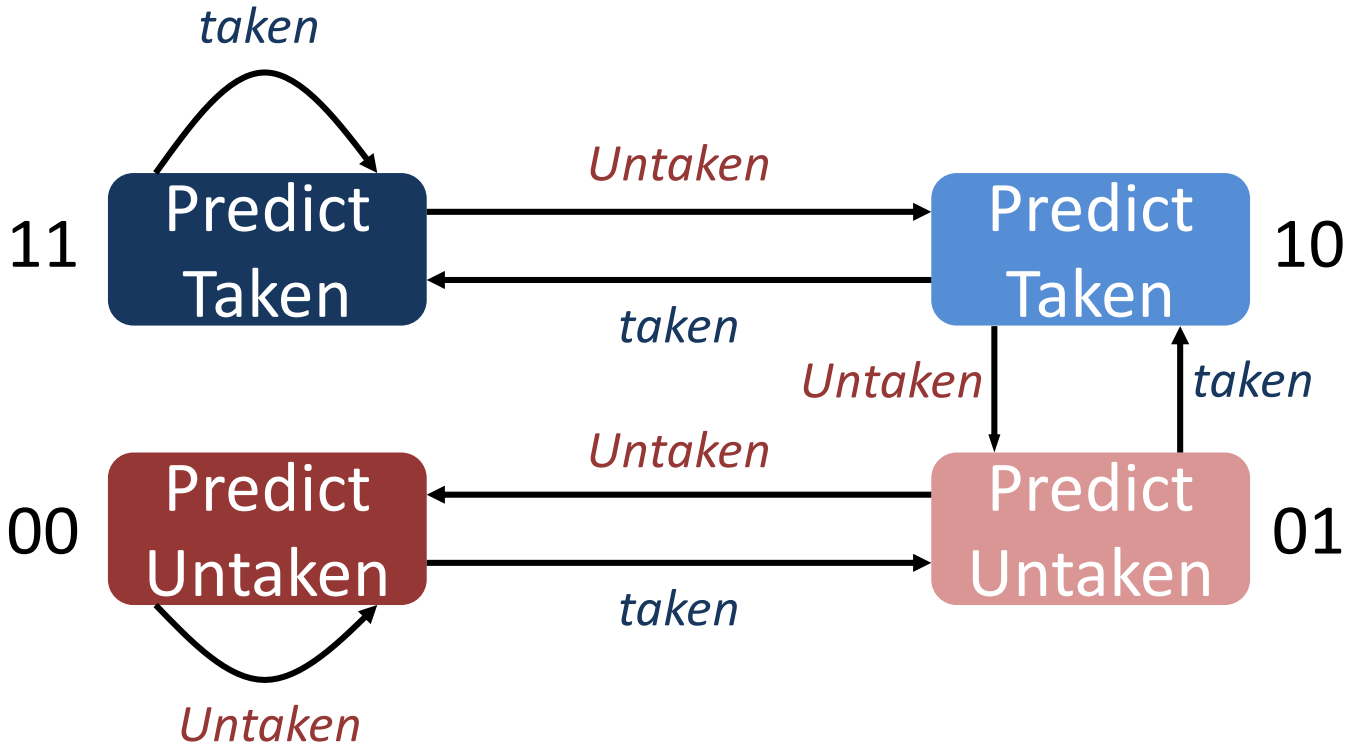
Four states:

1. *Strongly not-taken (SN or **0**0)*
2. *Weakly not-taken (WN or **0**1)*
3. *Weakly taken (WT or **1**0)*
4. *Strongly taken (ST or **1**1)*

The same outcome must occur multiple times to reach the strong states

State Diagram with k = 2

The state transitions show the *bimodal* behavior of Smith predictor



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	Branch Direction	$Smith_1$ (k = 1)		$Smith_2$ (k = 2)	
		State	Prediction	State	Prediction
A	1	1	1	11	1
B	1	1	1	11	1
C	0 *anomaly*	1	1 (misprediction)	11	1 (misprediction)
D	1 *anomaly*	0	0 (misprediction)	10	1
E	1	1	1	11	1
F	1	1	1	11	1

Branch Correlation

- In real programs, behavior of one branch is correlated with that of another
- Two types of correlation
  - Global branch correlation
  - Local branch correlation

Global Branch Correlation

- Can we predict the behavior of one branch based on the direction of another branch?

```
if (aa == 2)          B1
      aa = 0;
if (bb == 2)          B2
      bb = 0
if (aa != bb)         B3
      {…}
```

(B1, B2, B3)
(T, T, ?)
(T, F, ?)

```
if (counter > 15)                 B1
{
      reset = 1;
}
…
…
if (reset == 1)                   B2
      {…}
if (counter < 2)
      {…}                         B3
```

(B1, B2, B3)
(T, ?, ?)
(F, ?, ?)

Local Branch Correlation

```
for (i = 1; i < n; i++)
{
        do something;
}
```

n = 3 : 11011011011011011011

n = 4 : 11101110111011101110

- Capturing **four recent branch outcomes** in a window reveal distinct patterns

n = 3 : 11011011011011011011
n = 4 : 11101110111011101110

Local Branch Correlation

- Insight 1: Global Branch Correlation: Behavior of one branch is often **correlated** with the behavior of other branches

- Insight 2: Local Branch Correlation: Behavior of a branch is often correlated with the past outcomes of the same branch **other than** the outcome of the branch the last time it was executed

Correlating Branch Predictors

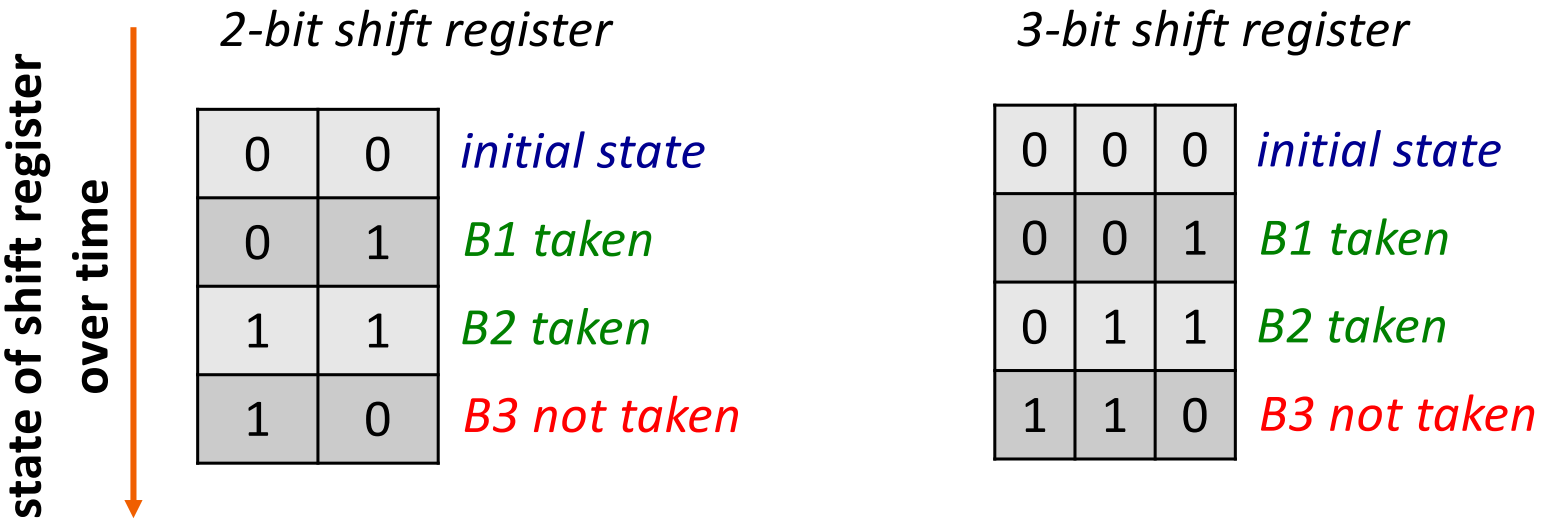
- Branch predictors that use the behavior (**outcomes**) of other branches to make a prediction

- **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

- What we are going to see next is called two-level adaptive branch prediction
  - Two separate levels of branch history to make the prediction

Recording Branch History

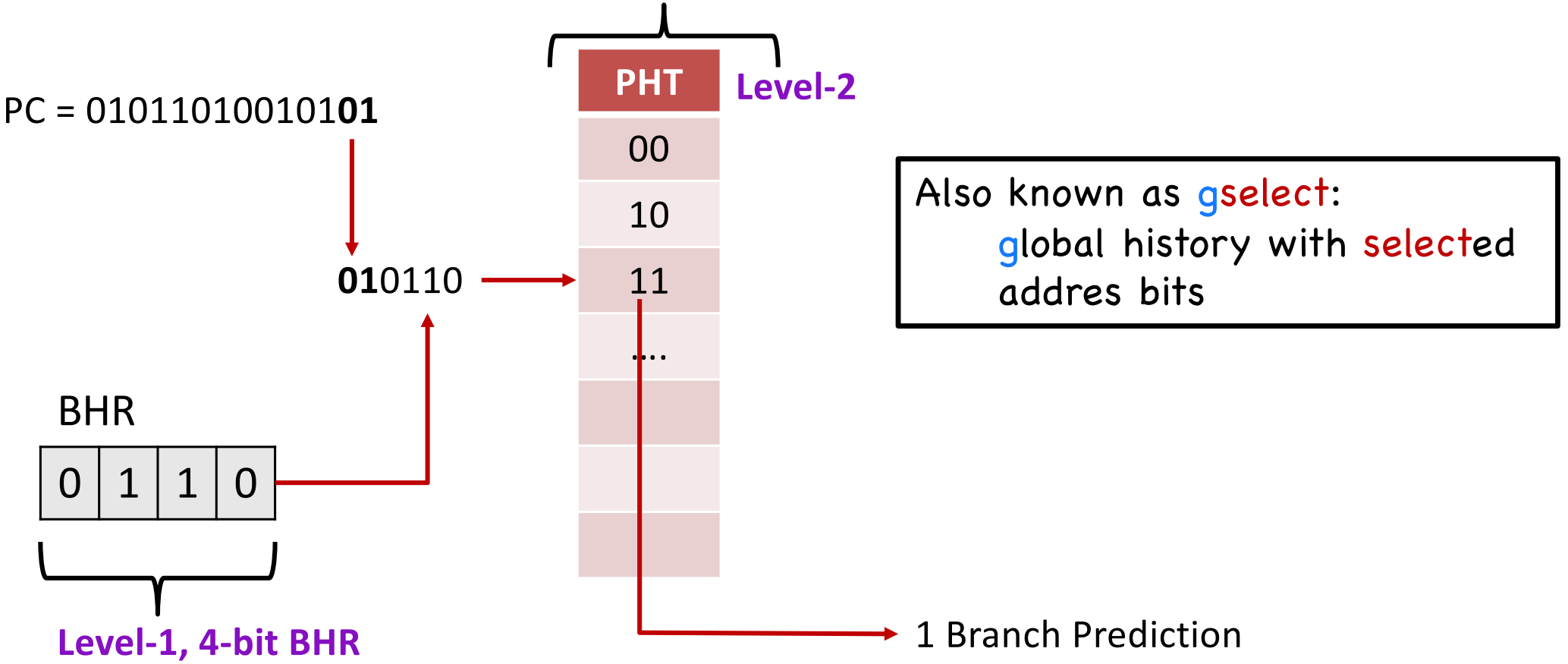
- The global history of the most recent m branches can be recorded in an *m-bit* shift register (*branch history register or BHR*)



- Shift the *actual* outcome at one end and *discard* the oldest outcome

Global-History Two-Level Predictor

Pattern History Table (PHT): *A table of saturating 2-bit counters. Indexed with a concatenation of BHR and a selection of branch address bits*



Also known as gselect:
global history with selected addres bits

Operation of gselect

- The counters in the indexed PHT provides prediction (and are updated) in a manner similar to Smith's algorithm
- The contents of BHR are shifted with the correct outcome of the branch
- PHT is indexed with a concatenation of address and history bits

Remembering gselect: Global branch history + Per-address PHT indexing

Intuition for gselect

- Behavior of one branch is correlated with that of an earlier branch
  - The branches test conditions involving the same variable
  - One instruction gaurds a variable that another one tests

Irrelevant Branches & Training Time

```
x = 0
if (something) /* branch A */
        x = 3;
if (someothercondition) /* branch B */
        y += 19;
if (x <= 0) /* branch C */
        dosomething();
```

C	B	A
1	0	0
0	0	1
1	1	0
0	1	1

- Tracking A could help achieve perfect prediction for C
- The “irrelevant” branch B increases the training time of gselect (predictor must **learn** to ignore these irrelevant history bits)

Size of PHT

- For a harware budget of 4 KB, how many entries can reside in the PHT at once? (Assume 2-bit saturating counters)

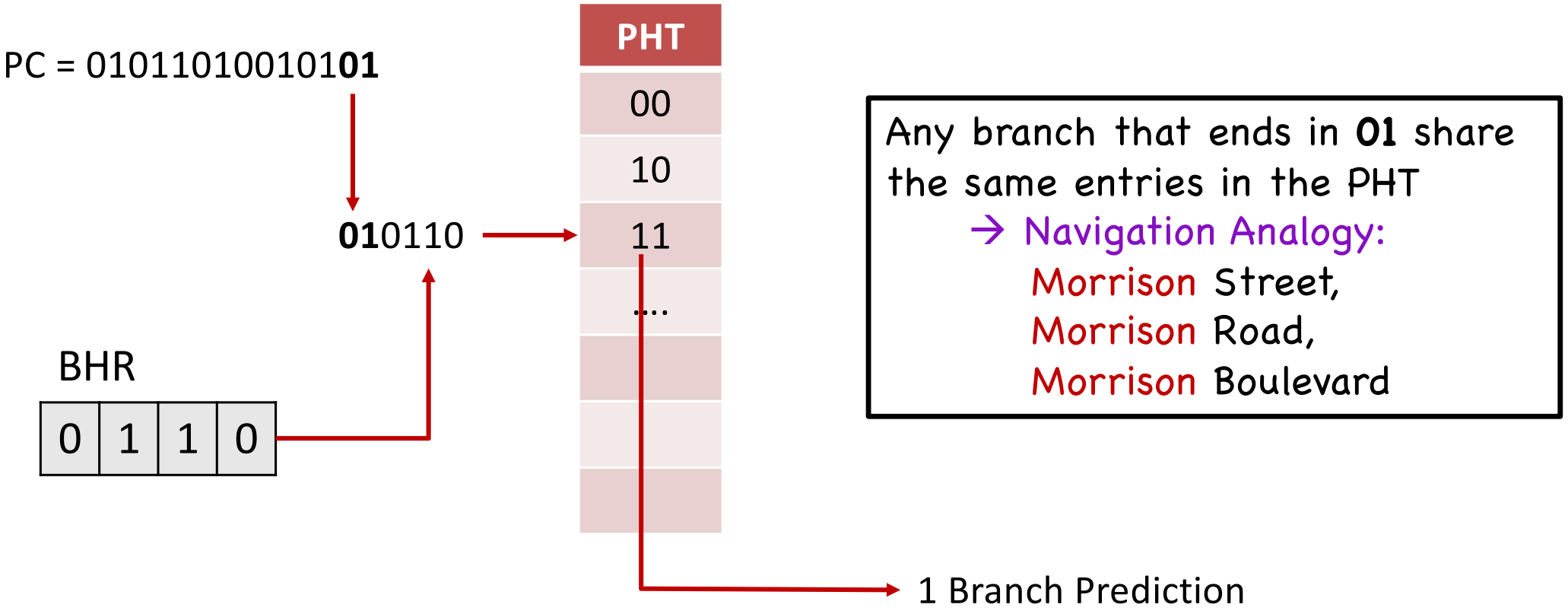
- **General formula:** For an **X KB** PHT, the PHT contains **4*X** 2-bit counters
  - Each byte can hold four 2-bit saturating counters

Indexing the PHT

- **History bits:** h
- **Address bits:** m
- Number of entries in the PHT: 2^{h+m}

- For a 4 KB PHT with 16,384 entries, we have 14 bits (total)
  - Distributing 14 bits across history and address exposes interesting tradeoffs

PHT Indexing: # Address Bits Matter



Any branch that ends in **01** share the same entries in the PHT

→ Navigation Analogy:

Morrison Street,

Morrison Road,

Morrison Boulevard

PHT Indexing: # History Bits Matter

- Using more history bits allows the predictor to correlate against more complex branch history patterns

- Optimally balancing the address vs. history bits depends on:
  - Compiler's arrangement of the code
  - Program
  - ISA
  - Program inputs
  - Anything else?

A Lot More to Say on Branch Prediction!

- Important component of a modern processor
  - Especially superscalar and out-of-order processors
- We can have local history two-level predictors as well
  - A table of BHRs (one per branch) instead of a single BHR
- Prediction accuracy above 90%
- **State of art:** Deep neural networks, machine learning approaches
- **Random branches** are increasingly common (ML, NLP, GPT)

Branch Target Buffer (BTB)

- Buffer = A small memory for storing “some” information
  - Similar to an SRAM cache

- 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

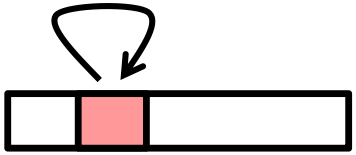
- Branch is predicted to be taken
  - Get target address from BTB
- Branch is predicted untaken
  - `PC = PC + 4`
- If the prediction is correct
  - Continue normal execution
- If the prediction is incorrect
  - Initiate pipeline flush (details are not in scope)

Locality and its Exploitation

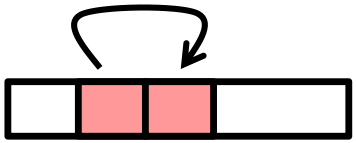
Locality

- **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**

Qualitative Estimates of Locality

- **Claim:** Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

- **Question:** Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Example

- **Question:** Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

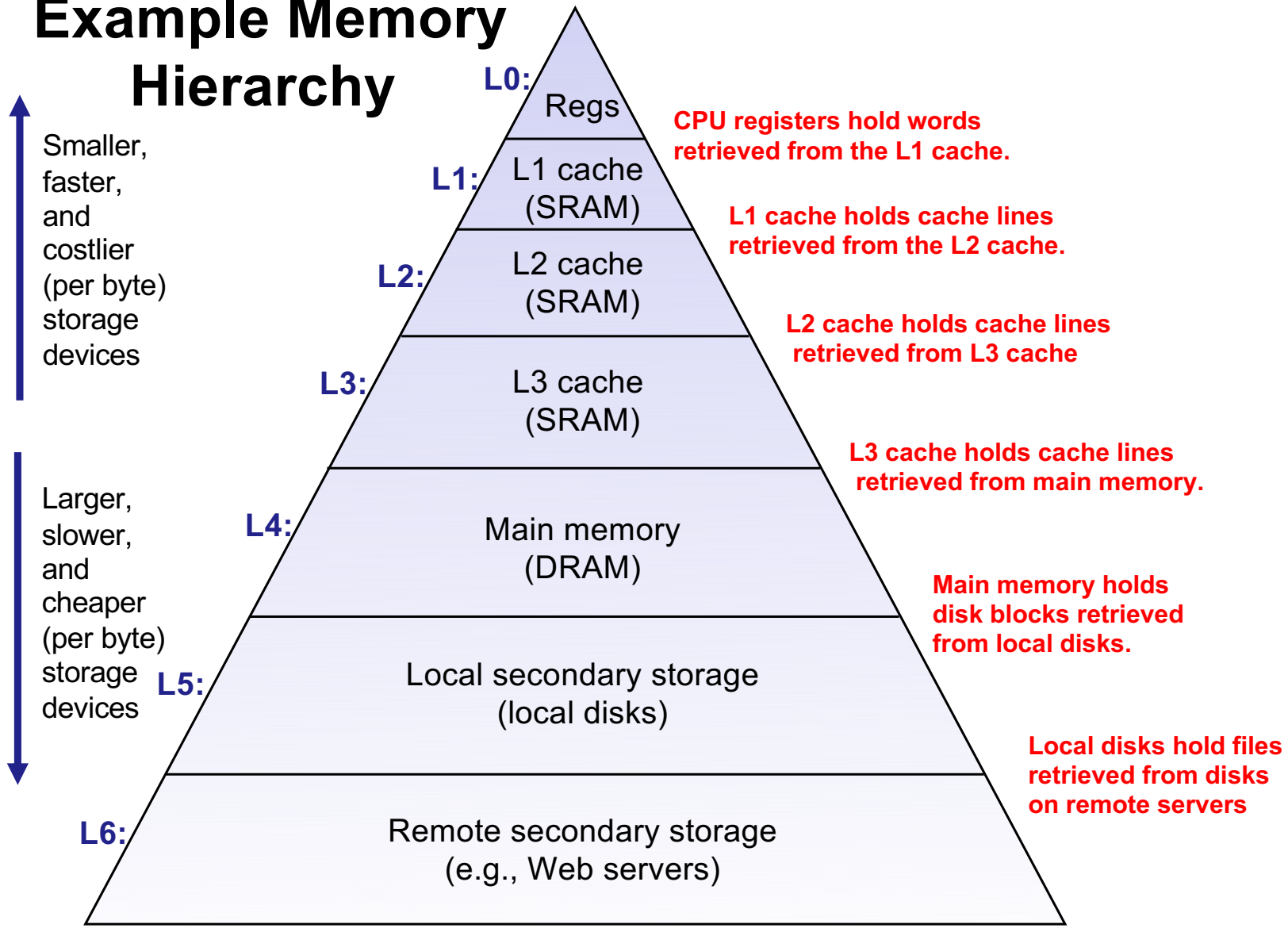
Memory Hierarchies

- **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.**

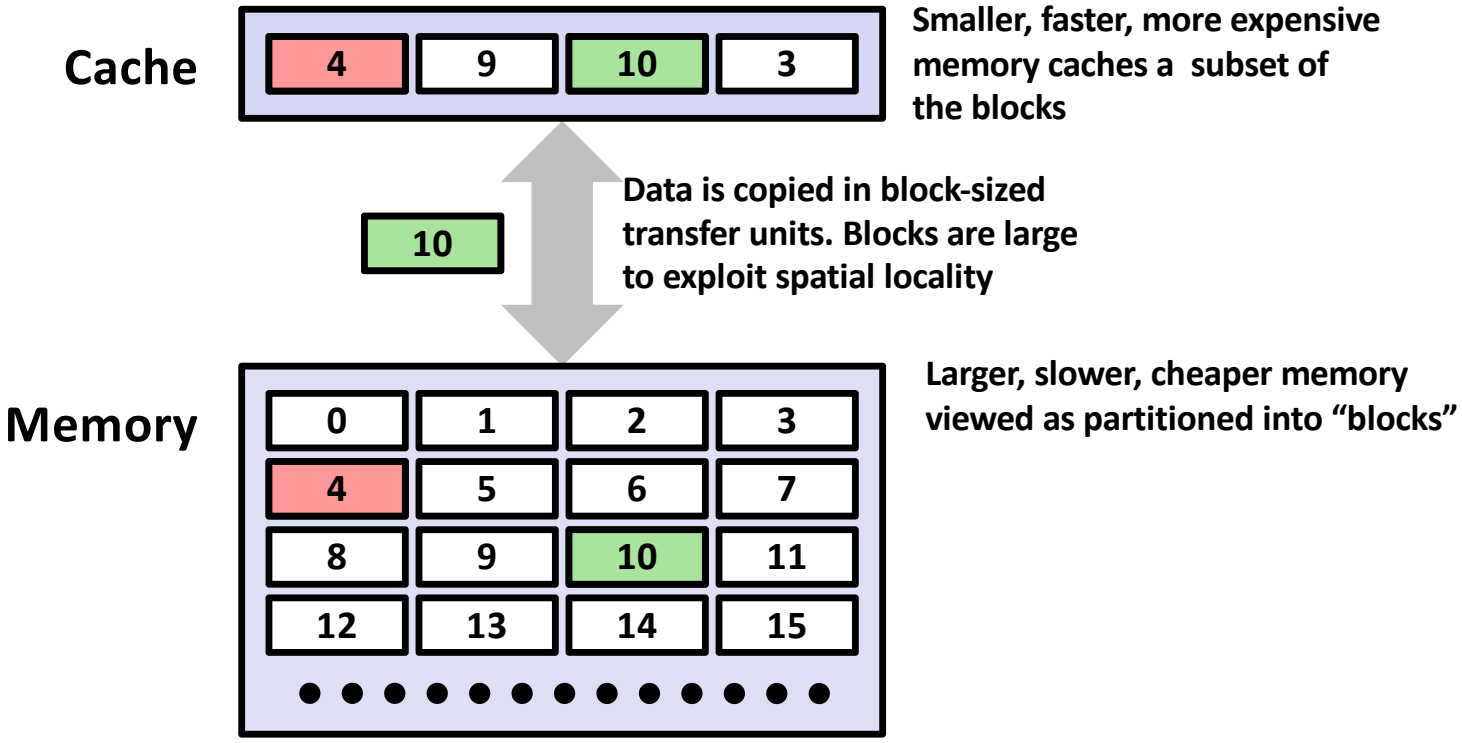
Example Memory Hierarchy



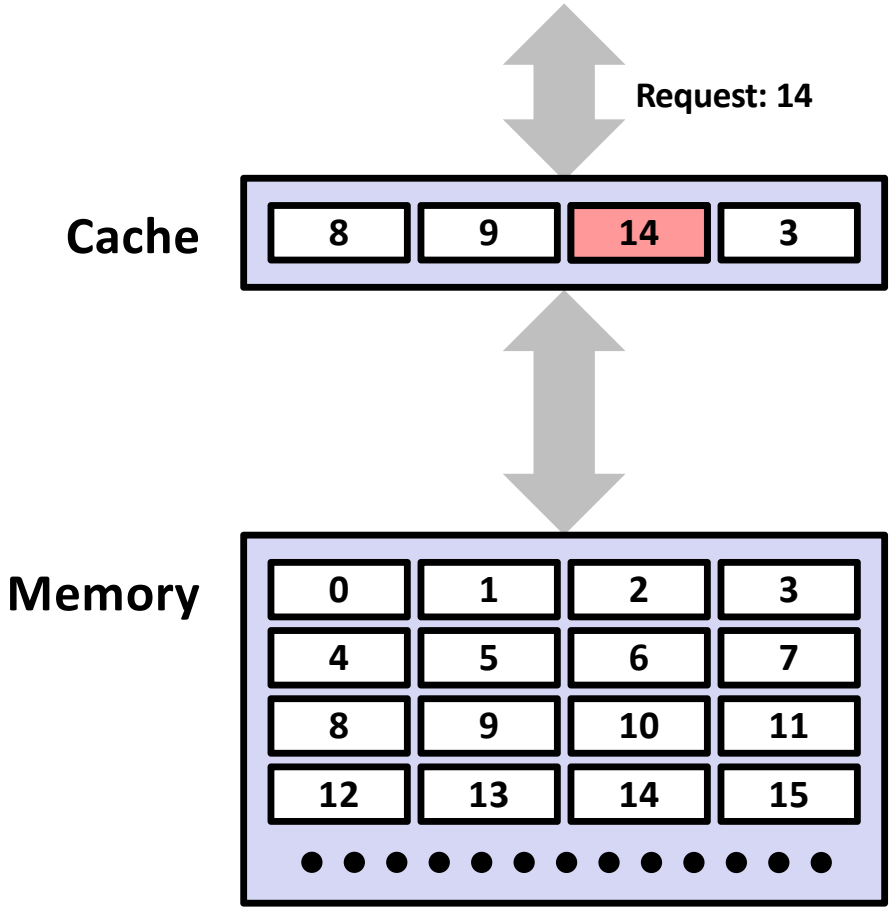
Caches

- ***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



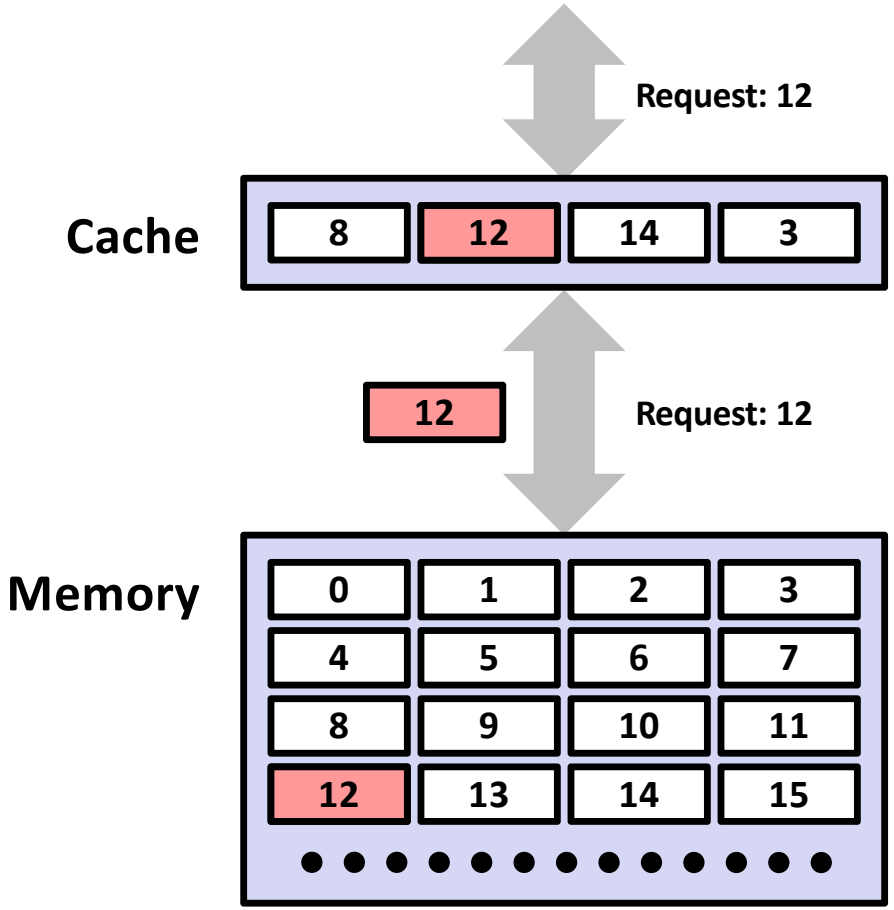
General Cache Concepts: Hit



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)

General Caching Concepts: Types of Cache Misses

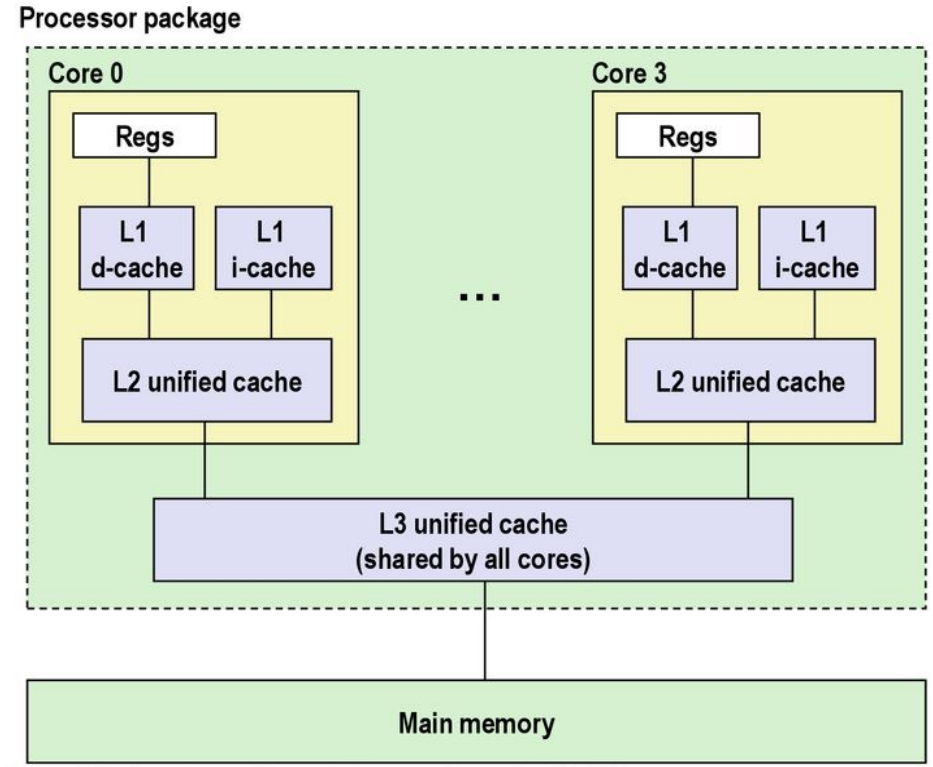
- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.
- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Summary

- **The speed gap between CPU, memory and mass storage continues to widen.**
- **Well-written programs exhibit a property called *locality*.**
- **Memory hierarchies based on *caching* close the gap by exploiting locality.**
- **A lot more on this in COMP2310**
- **For subsequent discussions, remember:**
  - Load miss = cache miss
  - Cache hit is very fast (few cycles, between 1 – 10)
  - Cache miss take 100+ cycles to resolve

Example Memory Hierarchy

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

35

Instruction-Level Parallelism (ILP)

Instruction-Level Parallelism (ILP)

- Since 1985, all processors have used pipelining to overlap the execution of instructions to improve performance
  - This overlap is termed as instruction-level parallelism (ILP)

- The simple **in-order** (*scalar* and *superscalar*) pipelines we have seen exploit limited forms of ILP

- In-order pipeline stalls in the face of data and control dependences both of which are abundant in real programs

- In-order pipeline stalls when a long-latency operation is in the pipeline
  - Long-latency memory accesses (cache misses)
  - Floating point operations

Problem with ILP Exploitation via an In-Order Pipeline

- Consider the following scenario

```
ADD   R0,  R0,  #4
LDR   R1,  [R0, #0]
ADD   R2,  R2,  #1
ADD   R3,  R3,  #2
ADD   R4,  R3,  #1
SUB   R5,  R1,  #1
```

- Load misses in the cache (100 cycles penalty to access memory)
- `LDR` must stall. (The dependent instruction on `LDR` is `SUB`)
- All subsequent instructions that do not depend on `LDR` also stall

Problem with In-Order Pipeline

- Consider the following scenario

```
ADD   R0,  R0,  #4
LDR   R1,  [R0, #0]
ADD   R2,  R2,  #1
ADD   R3,  R3,  #2
ADD   R4,  R3,  #1
SUB   R5,  R1,  #1
```

- **Two options to exploit more ILP:**
  - Stall until dependent instruction `SUB` is encountered (aggressive but still in order)
  - Keep executing independent instructions (ones that come after `SUB`) as long as their operands are ready (out of order)

Dependences and Hazards

- To transition from in-order to out-of-order, we need to understand new forms of data hazards

- **Recall:** Dependence is a program's property

- **Recall:** Hazard is a microarchitecture property

- In addition to read-after-write and control hazards, there are two new hazards to consider

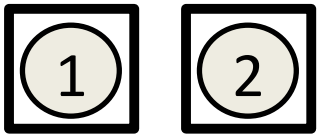
Dependences and Hazards

- Next, we will investigate the different data dependency types
- And their potential for hazards
- And how do they limit exploiting ILP for a processor that can execute instructions whenever operands are ready
- We don’t know how it can be done yet ....

Example Sequence 1

1. LDR R2, [R6, #0]
2. ADD R3, R4, R5

Two independent instructions



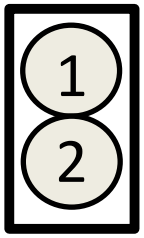
Example Sequence 2

1. LDR R2, [R6, #0]
2. ADD R3, R2, R5

Data or true dependence

i2 needs the result of **i1**

A single (dependent) instruction chain



Example Sequence 3

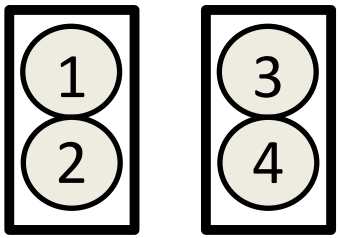
```
1.  LDR   R2,   [R6,   #0]
2.  ADD   R3,   R2,    R5
3.  LDR   R4,   [R6,   #0]
4.  ADD   R7,   R4,    R9
```

Data or true dependence

i2 needs the result of **i1**

i4 needs the result of **i3**

Two independent instruction chains



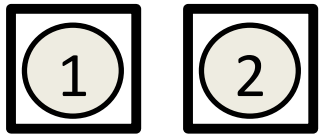
Example Sequence 4

1. LDR R2, [R6, #0]
2. ADD R6, R3, R5

False (anti) dependence

i2 needs to write what **i1** needs to read

Two independent instructions



- Limitation of number of registers in the ISA, but ILP exists
- We can rename register R6 in **the second instruction** and eliminate the (false) dependency

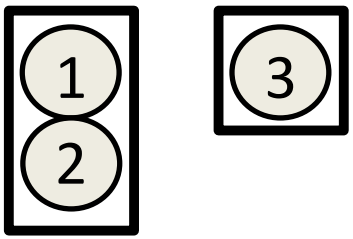
Example Sequence 5

1. LDR R2, [R6, #0]
2. ADD R3, R2, R5
3. ADD R2, R7, R8

true dependence

Output dependence
i1 and **i3** wants to write to the same register

Instruction chain (**i1→i2**) + independent instruction (**i3**)



- Limitation of # registers, but ILP exists
- We can rename register R2 in **(3)** and execute (3) earlier than 1 – 2 chain

Dependences and Hazards

- True dependence results in:
  - Read-after-write hazard (RAW)
- Anti-dependence results in:
  - Write-after-read hazard (WAR)
- Output dependence results in:
  - Write-after-write hazard (WAW)
- **Single-cycle CPU:** Each instruction takes one cycle; one instruction at any time
  - None of the dependences result in a hazard
- **In-order pipeline:** Multiple instructions in different stages (possibility of RAW)
- **Out-of-order:** ALL BETS ARE OFF!

What we have learnt so far

- Dependences exist in real programs due to
  - Programmer's intention
  - Limited number of registers
  - Branches

- Dependences lead to hazards (microarchitecture dependent)

- True dependences create a chain of instructions that cannot be reordered

- False dependences can be eliminated to uncover ILP

Instruction Scheduling

- A straight-line instruction sequence can be executed in different orderings
  - Each ordering is an instruction schedule
  - Must respect true dependences

```
ADD   R0, R0, #4
LDR   R1, [R0, #0]
ADD   R2, R2, #1
ADD   R3, R3, #2
ADD   R4, R3, #1
SUB   R5, R1, #1
```

- Original sequence

```
ADD   R0, R0, #4
LDR   R1, [R0, #0]
SUB   R5, R1, #1
ADD   R2, R2, #1
ADD   R3, R3, #2
ADD   R4, R3, #1
```

- Valid reordering

```
LDR   R1, [R0, #0]
ADD   R0, R0, #4
ADD   R2, R2, #1
ADD   R3, R3, #2
ADD   R4, R3, #1
SUB   R5, R1, #1
```

- Invalid reordering

Instruction Scheduling

- Some reorderings or schedules exhibit high ILP than others depending on the processor
- Consider the following two instruction schedules

```
LDR   R0,  [R7, #4]
ADD   R1,  R0,  #0
LDR   R2,  [R7,  #8]
ADD   R3,  R2,  #2
```

```
LDR   R0,  [R7, #4]
LDR   R2,  [R7,  #8]
ADD   R1,  R0,  #0
ADD   R3,  R2,  #2
```

- **On a processor that**
  - stalls due to a load-use hazard,
  - each memory access takes 100 cycles,
  - and multiple memory accesses can be resolved concurrently
  - Which schedule is better?
  - What about a single-cycle CPU?

A Historical Debate

- **Historical (ongoing) debate: How best to exploit ILP?**
  - Dynamically in hardware (dynamic = during execution)
    - No need to recompile code, portable, transparent, hardware has more knowledge of program behavior: loop counters, inputs, branch behavior
    - Power, area, energy, security issues (end of Moore's law, transition to multicore)
  - Statically in software (find parallelism at compile time)
    - Compiler can do whole-program optimizations, inspired innovations in compiler technology, commercial failure ....

ILP Exploitation: A Taxonomy

- **Statically scheduled superscalar processor**
  - Compiler schedules instructions during program creation
  - Hardware does no reordering of instructions (sequential unless branch changes PC)
  - Compiler can create "interesting schedules" by doing deep program analysis
  - Schedule is **static** as it does not change dynamically other than "basic" dependency checks

- **VLIW (Very Long Instruction Word) processor**
  - Static scheduling by compiler. Instruction words are very large. Up to 28 insts. in a bundle
  - Compiler does **"smart"** analysis to construct "interesting" schedules (interesting = high ILP)
  - Conceptually same as above. "Some differences" in philosophy (smart compiler, dumb hw.)
  - True VLIW: No dependency checking in hardware

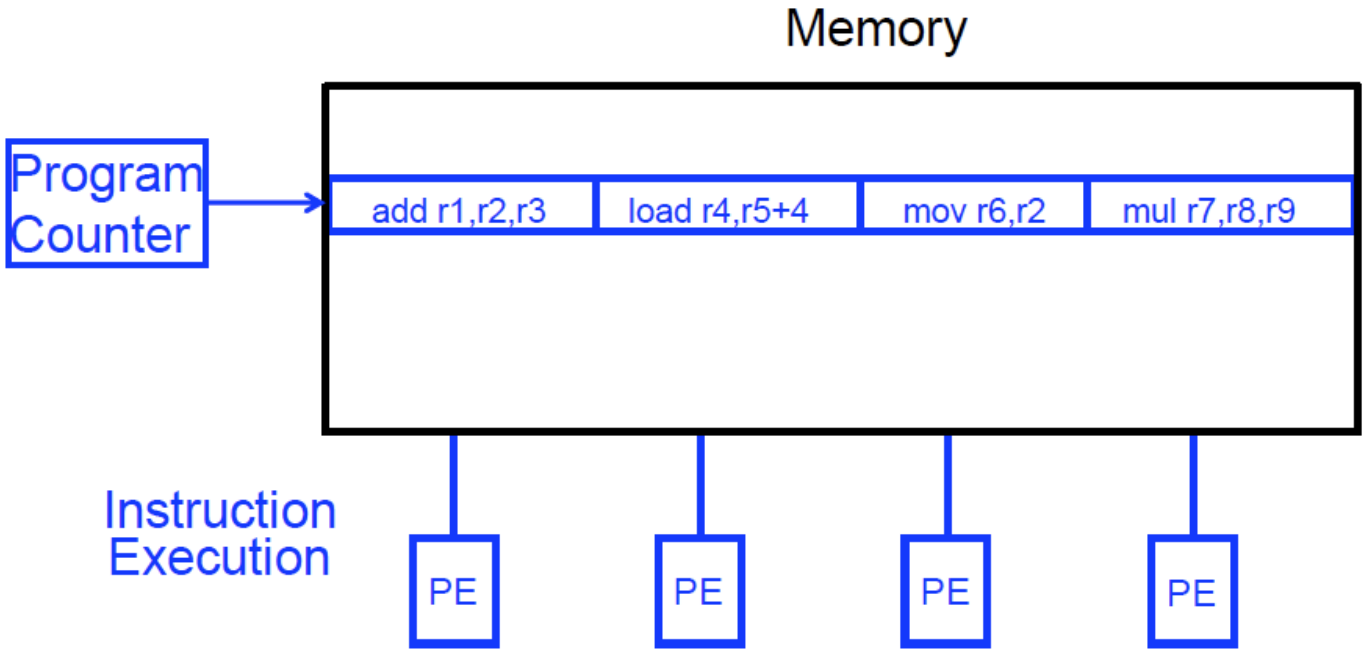
- **Dynamically scheduled superscalar processor**
  - Hardware does scheduling during program execution
  - Can reorder instructions to extract maximum ILP
  - Hardware can construct different "instruction schedules" based on different executions of the same set of basic blocks (different branch outcomes)

VLIW Architectures
(Very Long Instruction Word)

VLIW Concept

- Superscalar
  - Hardware fetches multiple instructions and checks dependencies between them
- VLIW (Very Long Instruction Word)
  - Software (compiler) packs independent instructions in a larger "instruction bundle" to be fetched and executed concurrently
  - Hardware fetches and executes the instructions in the bundle concurrently
- No need for hardware dependency checking between concurrently-fetched instructions in the VLIW model
  - **Simple hardware, complex compiler**

VLIW Concept



- Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.
  - ELI: Enormously longword instructions (512 bits)

VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD/vector processors)

- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

- Traditional VLIW Characteristics

  1. Multiple instruction fetch/execute, multiple functional units

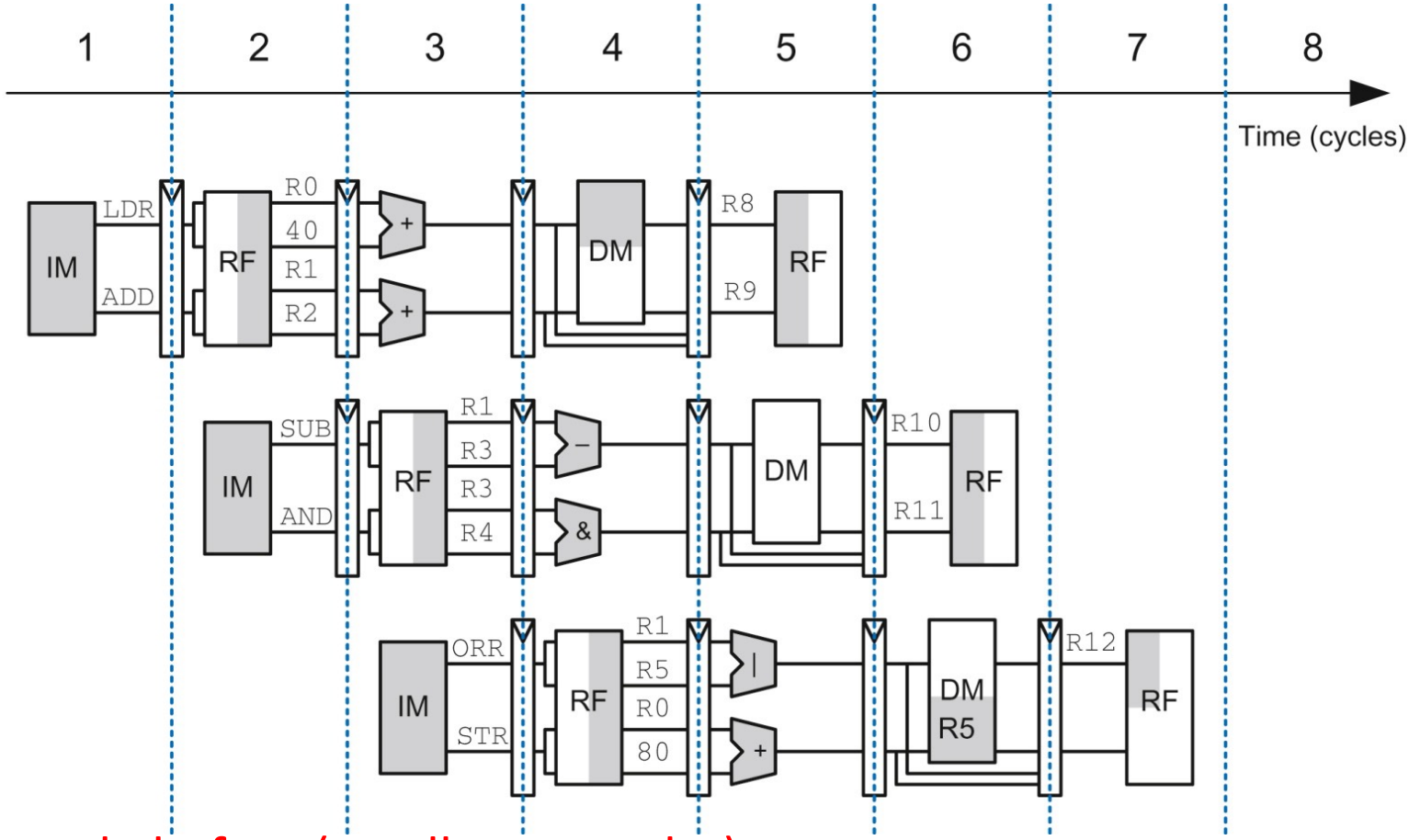
  2. All instructions in a bundle are executed in lock step

  3. Instructions in a bundle statically aligned to be directly supplied into the functional units

VLIW Example: 2-Wide Bundles

- 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]
```



- We saw this example before (recall superscalar)

VLIW Lock-Step Execution

- Lock-step (all or none) execution
  - If any operation in a VLIW instruction stalls, all concurrent operations stall

- In a truly VLIW machine:
  - the compiler handles all dependency-related stalls
  - hardware does **not** perform dependency checking
  - What about variable latency operations? Memory stalls?

VLIW Philosophy & Principles

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

**Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau**

**Department of Computer Science, Yale University
New Haven, CT 06520**

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

Fisher et al., "Parallel Processing: A Smart Compiler and a Dumb Machine," CC 1984.

VLIW Philosophy & Principles

- Philosophy similar to RISC (simple instructions and hardware)
  - Except "multiple instructions in parallel: in VLIW

- RISC (John Cocke+, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Josh Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design, low power

Our Focus

- **Dynamically scheduled superscalar processor**
  - Hardware does scheduling during program execution
  - Can reorder instructions to extract maximum ILP
  - Hardware can construct different “instruction schedules” based on different executions of the same set of basic blocks (different branch outcomes)