

# Digital Design & Computer Arch.

## Lecture 13: Pipelining

Prof. Onur Mutlu

ETH Zürich

Spring 2021

16 April 2021

# Required Readings

---

## ■ This week

- Pipelining
  - H&H, Chapter 7.5
- Pipelining Issues
  - H&H, Chapter 7.8.1-7.8.3

## ■ Next week

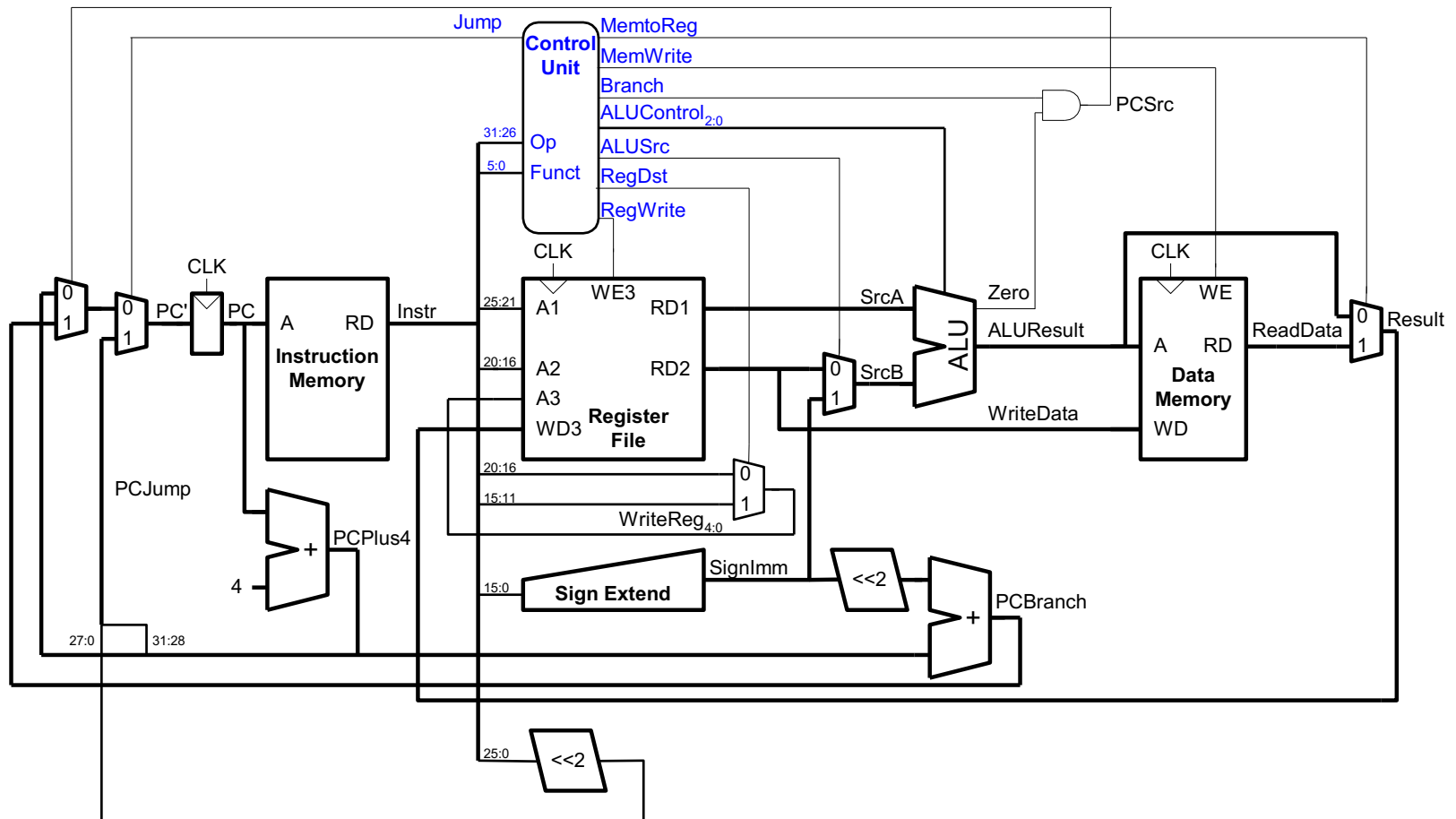
- Out-of-order execution
  - H&H, Chapter 7.8-7.9
- Smith and Sohi, “**The Microarchitecture of Superscalar Processors**,” Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

# Agenda for Today & Next Few Lectures

---

- Last week & yesterday
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures
- Today & next week
  - Pipelining
  - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Next week & the week after
  - Out-of-Order Execution
  - Issues in OoO Execution: Load-Store Handling, ...

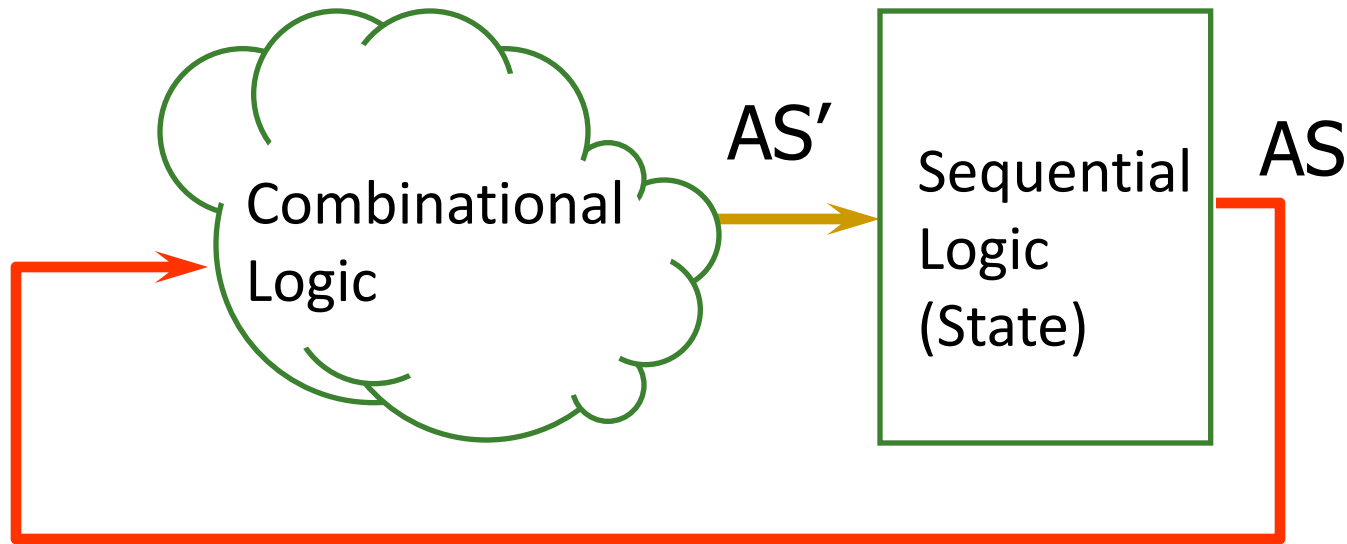
# Review: Single-Cycle MIPS Processor



# Review: Single-Cycle MIPS FSM

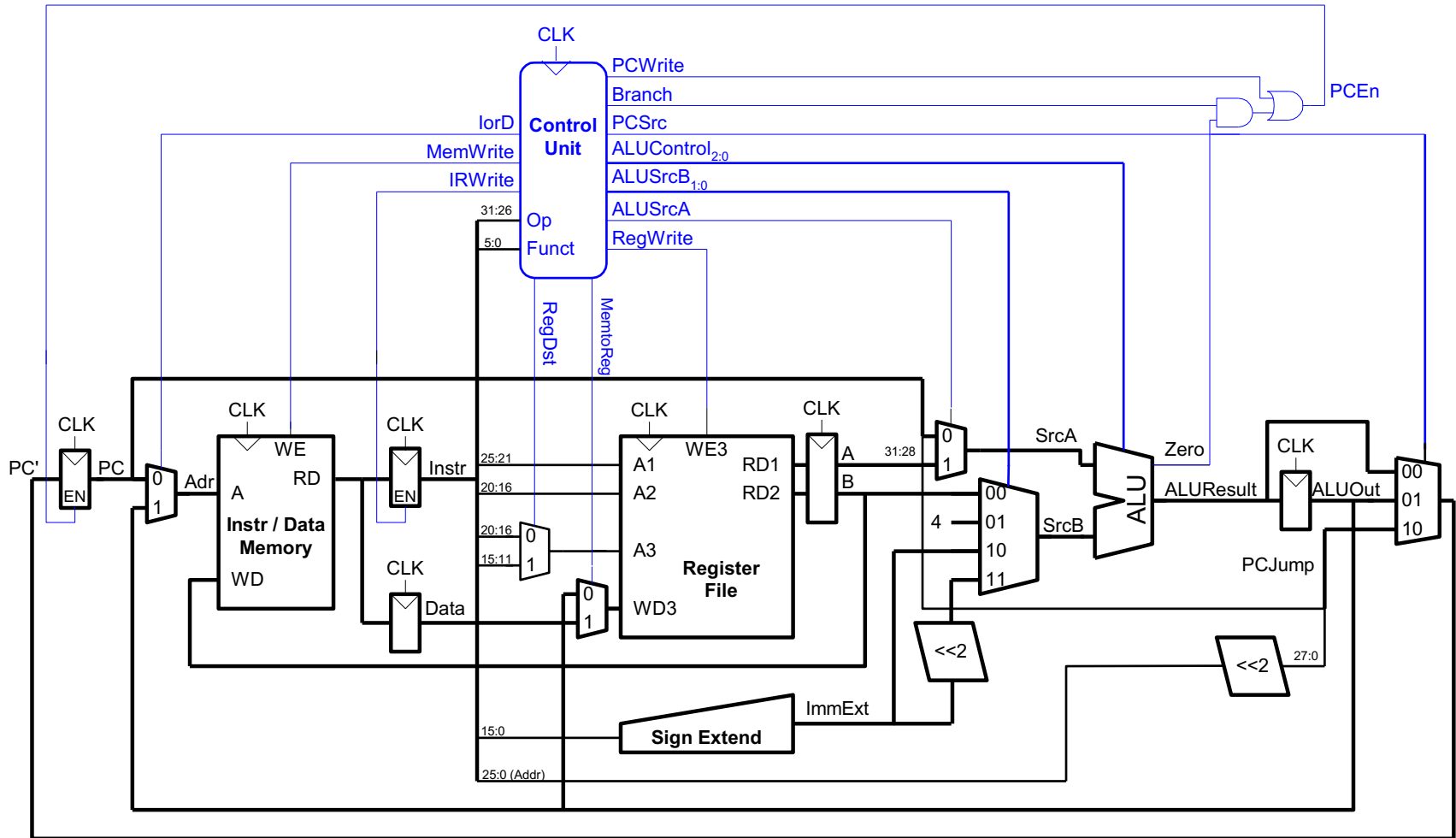
---

- Single-cycle machine

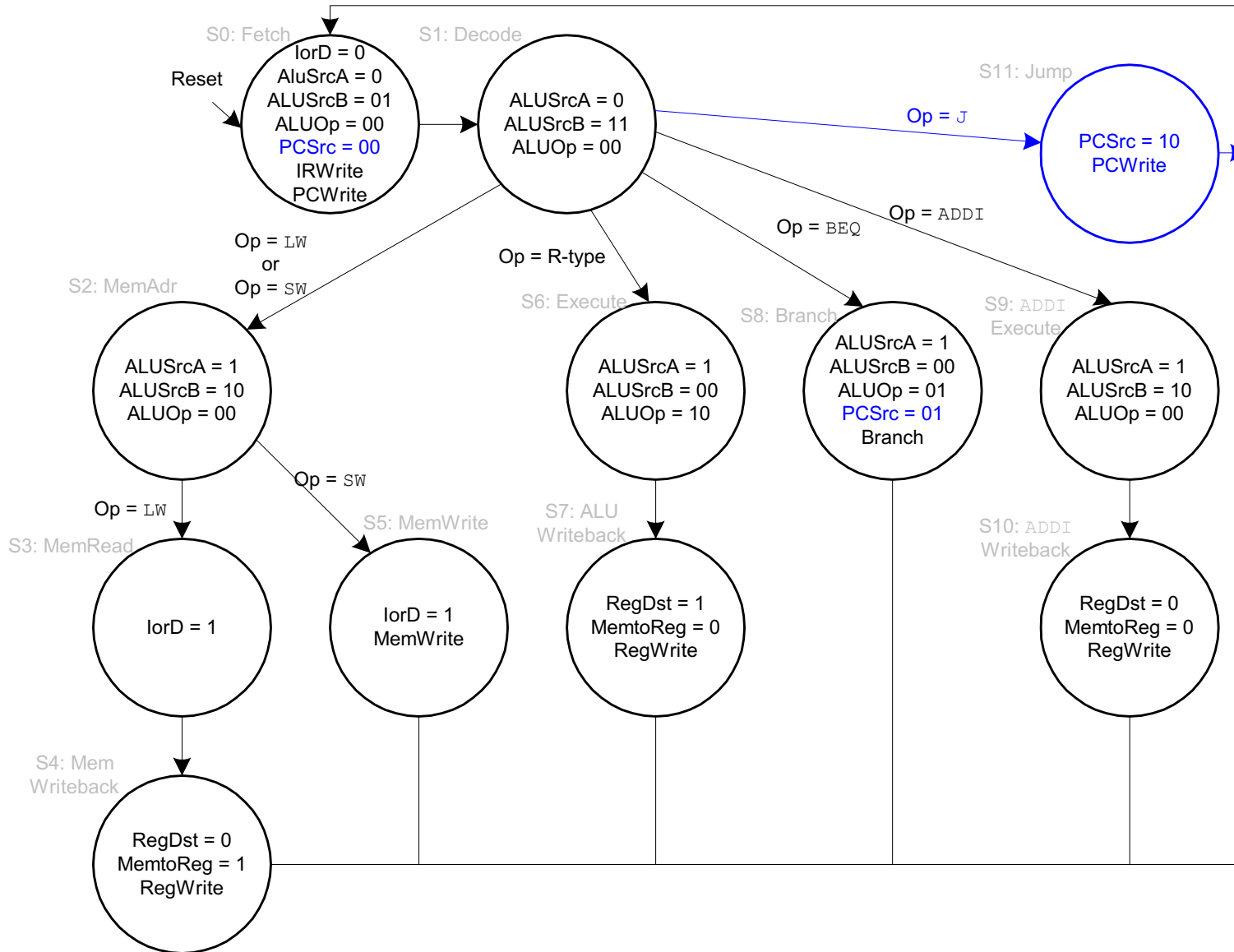


# Can We Do Better?

# Review: Multi-Cycle MIPS Processor



# Review: Multi-Cycle MIPS FSM



**What is the  
shortcoming of  
this design?**

**What does  
this design  
assume  
about memory?**



# Can We Do Better?

# Can We Do Better?

---

- What limitations do you see with the multi-cycle design?
- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening

# Can We Use the Idle Hardware to Improve Concurrency?

---

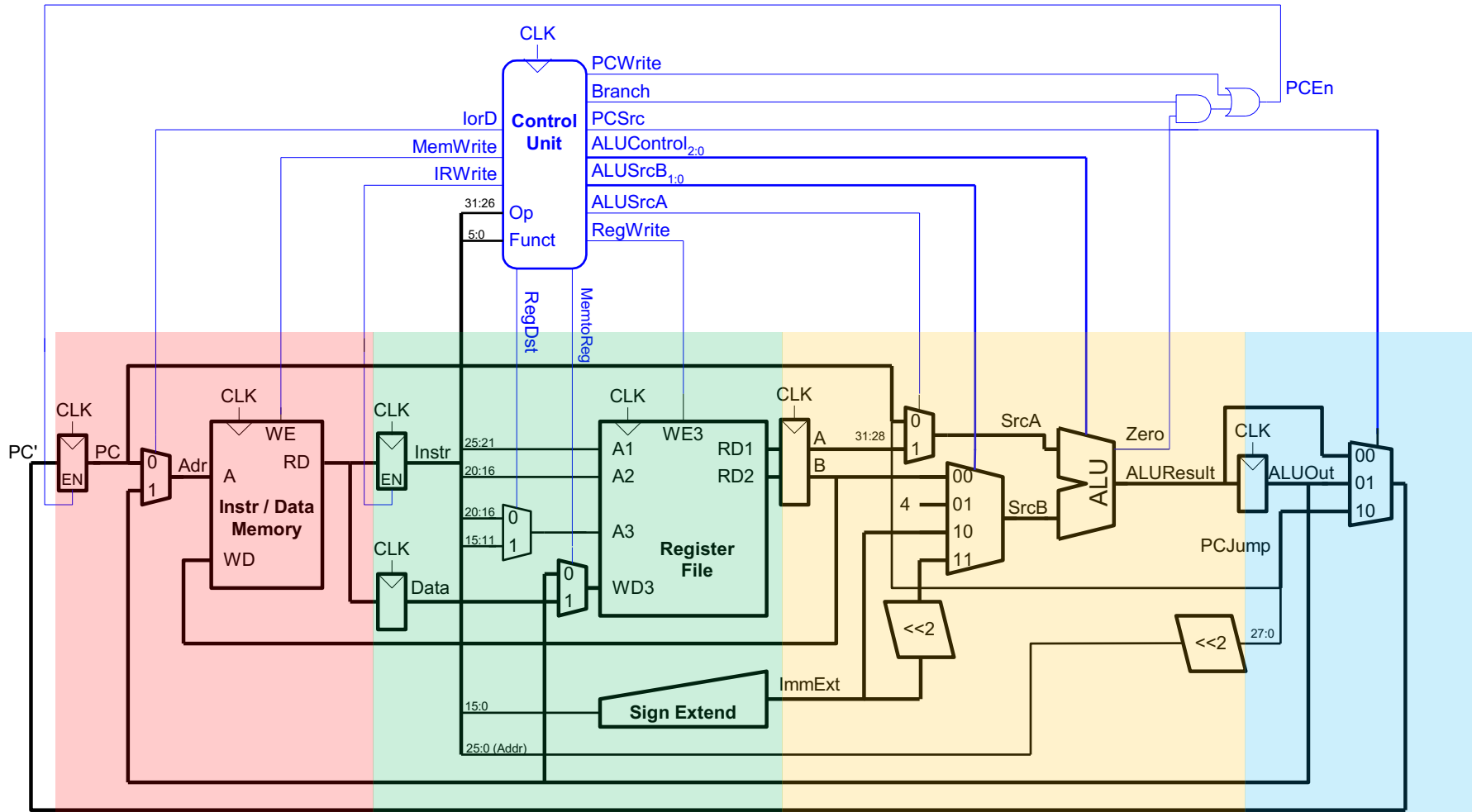
- Goal: **More concurrency → Higher instruction throughput** (i.e., more “work” completed in one cycle)
- Idea: When an instruction is using some resources in its processing phase, **process other instructions on idle resources** not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

# Can Have Different Instructions in Different Stages

---

- ❑ Fetch
  - ❑ Decode
  - ❑ Evaluate Address
  - ❑ Fetch Operands
  - ❑ Execute
  - ❑ Store Result
1. Instruction fetch (IF)
  2. Instruction decode and register operand fetch (ID/RF)
  3. Execute/Evaluate memory address (EX/AG)
  4. Memory operand fetch (MEM)
  5. Store/writeback result (WB)

# Can Have Different Instructions in Different Stages



Of course, we need to be more careful than this!

# Pipelining

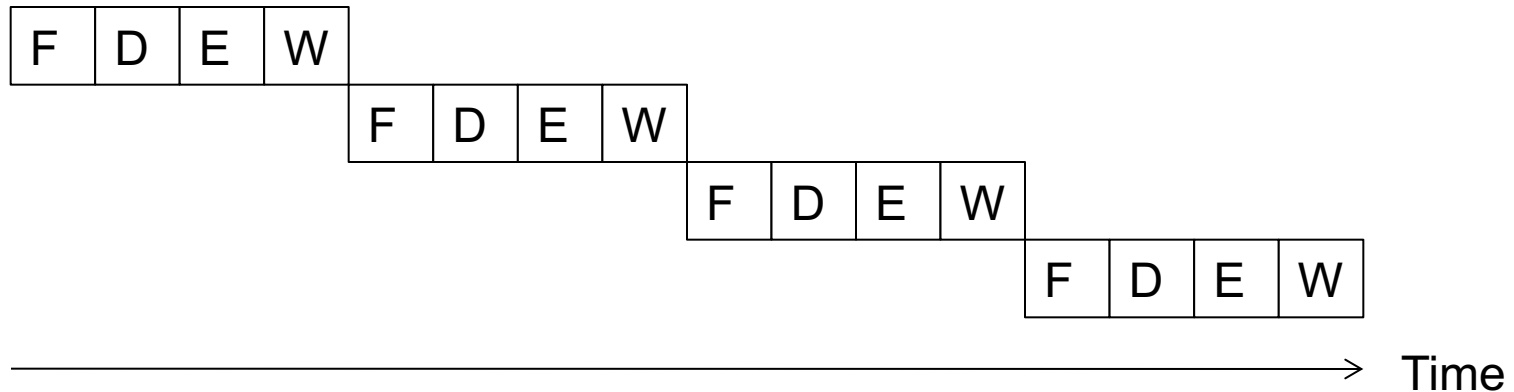
# Pipelining: Basic Idea

---

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: **Increases instruction processing throughput (1/CPI)**
- Downside: Start thinking about this...

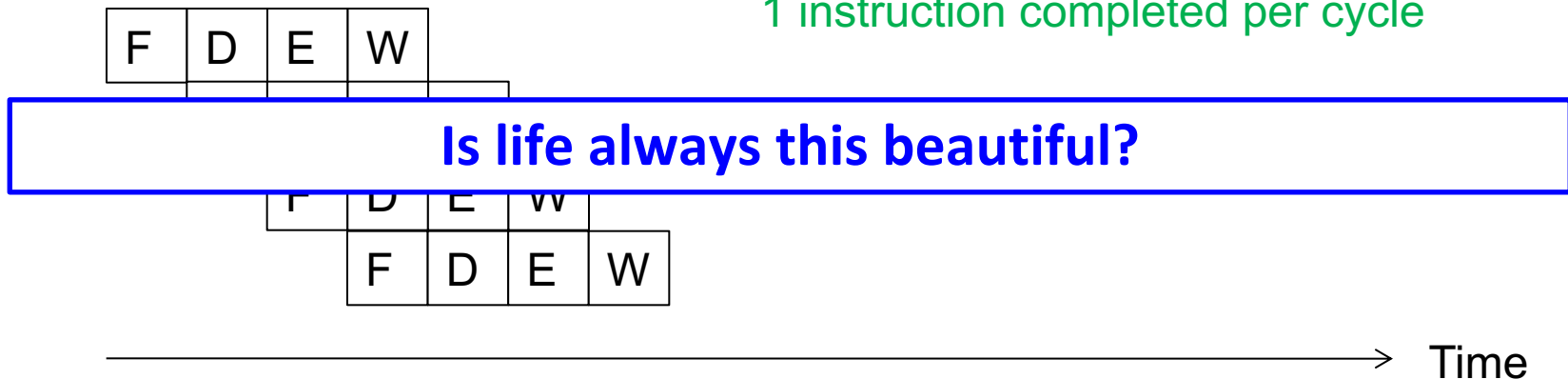
# Example: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction



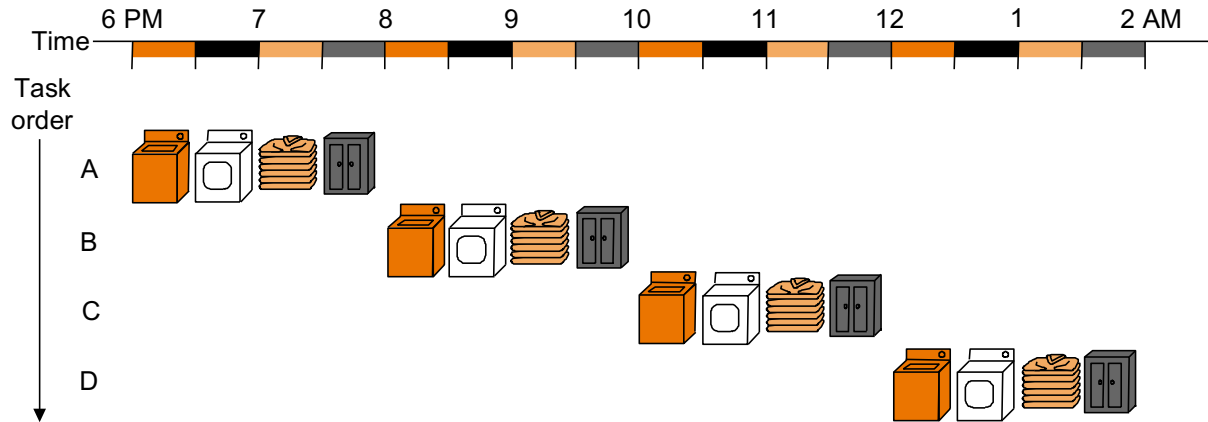
- Pipelined: 4 cycles per 4 instructions (steady state)

1 instruction completed per cycle



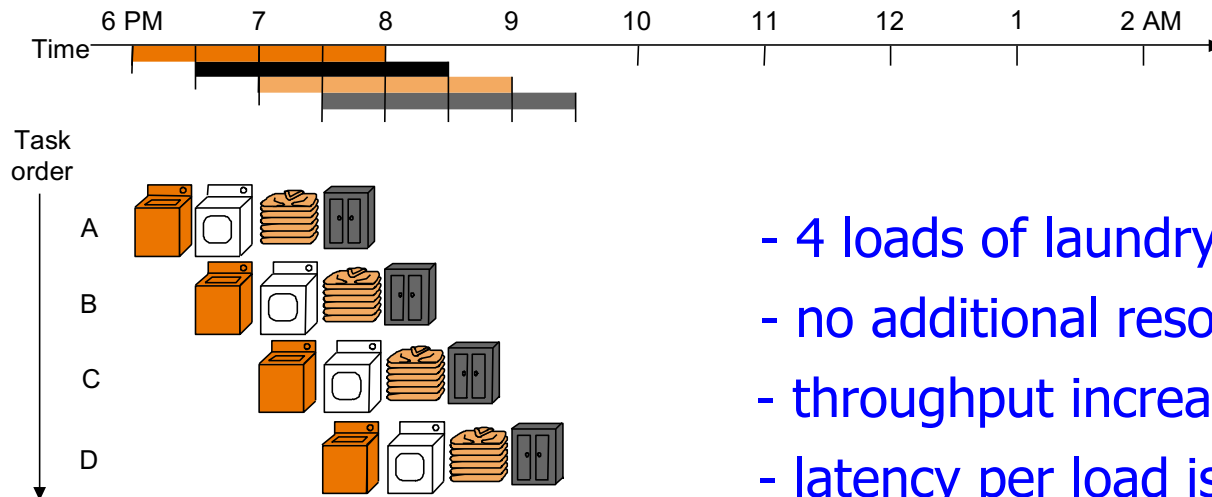
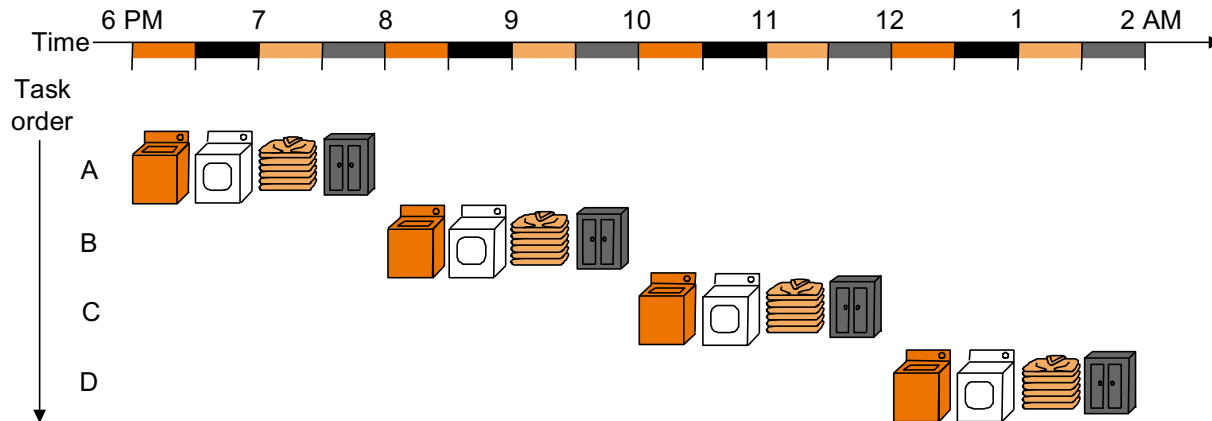


# The Laundry Analogy



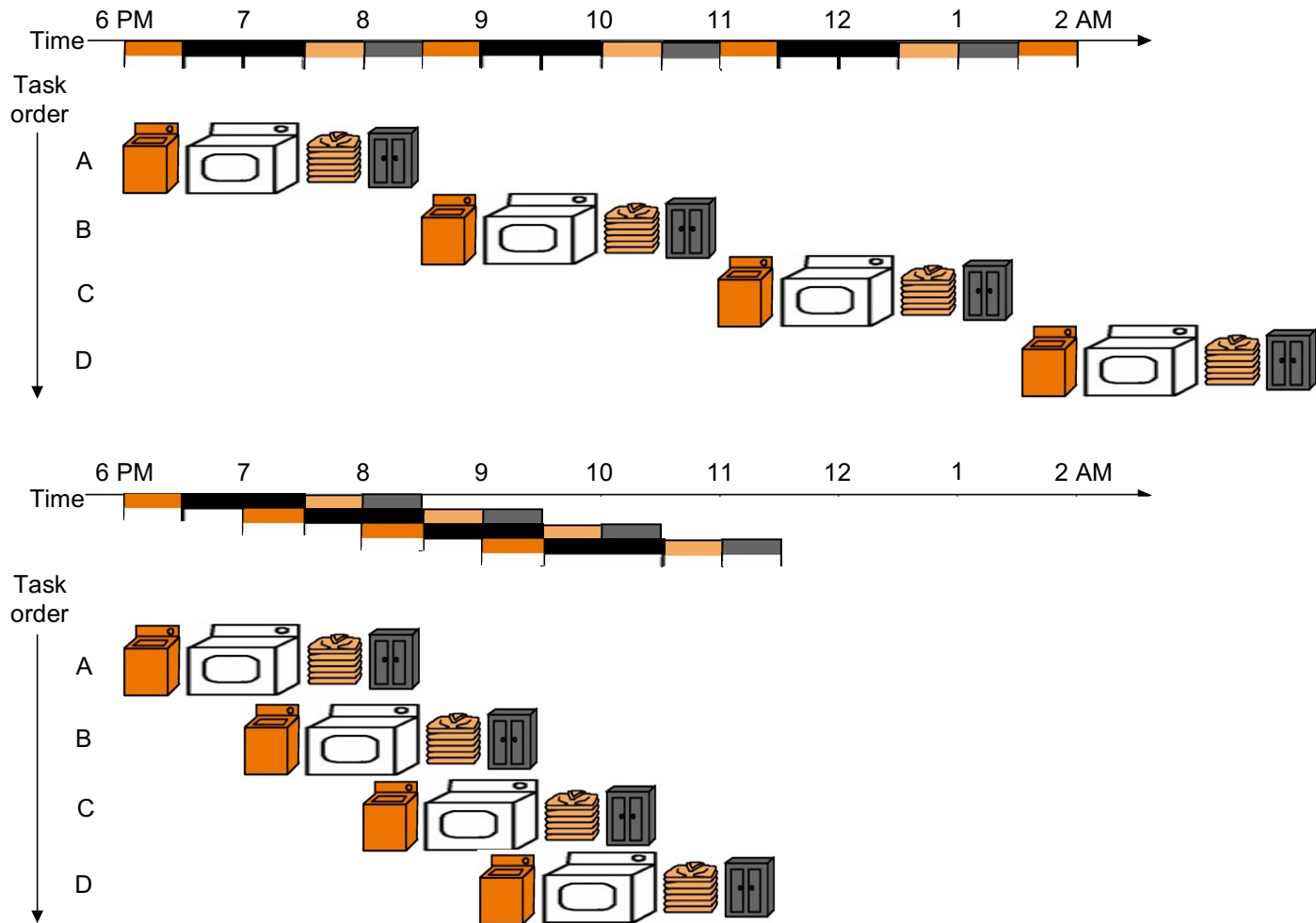
- “place one dirty load of clothes in the washer”
  - “when the washer is finished, place the wet load in the dryer”
  - “when the dryer is finished, take out the dry load and fold”
  - “when folding is finished, ask your roommate (??) to put the clothes away”
- steps to do a load are sequentially dependent
  - no dependence between different loads
  - different steps do not share resources

# Pipelining Multiple Loads of Laundry



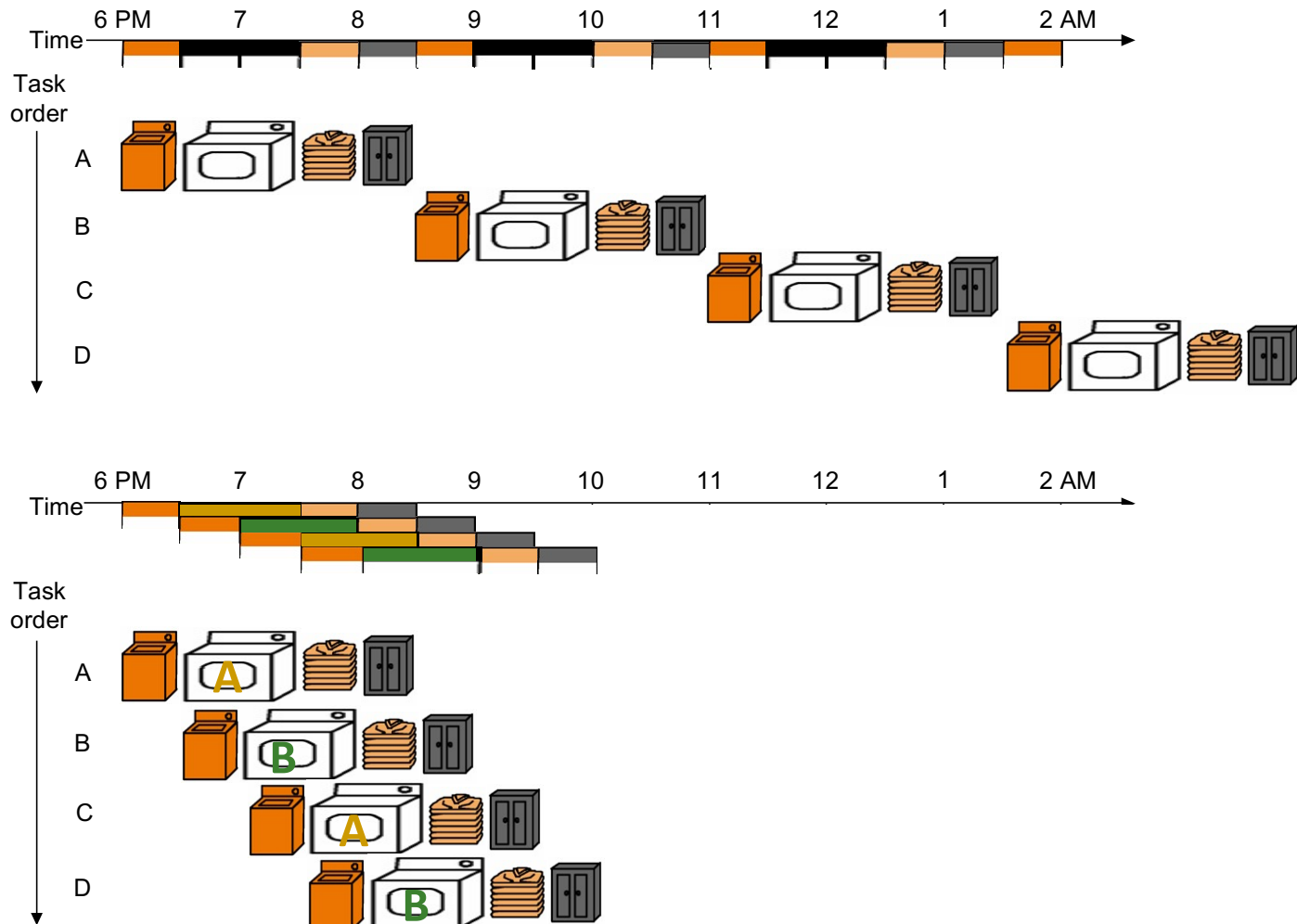
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

# Pipelining Multiple Loads of Laundry: In Practice



the slowest step (the dryer) decides throughput

# Pipelining Multiple Loads of Laundry: In Practice



throughput restored (2 loads per hour) using 2 dryers

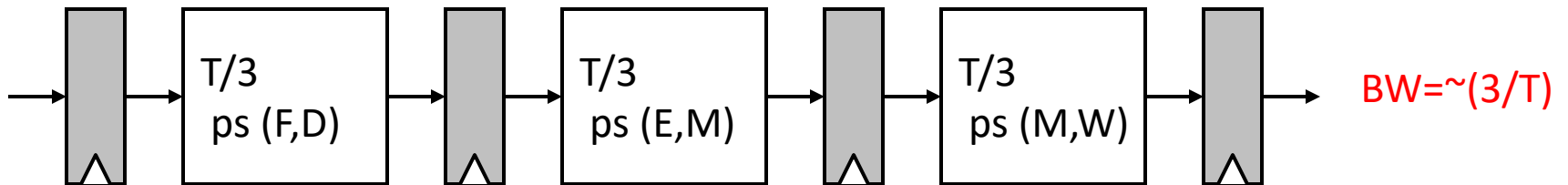
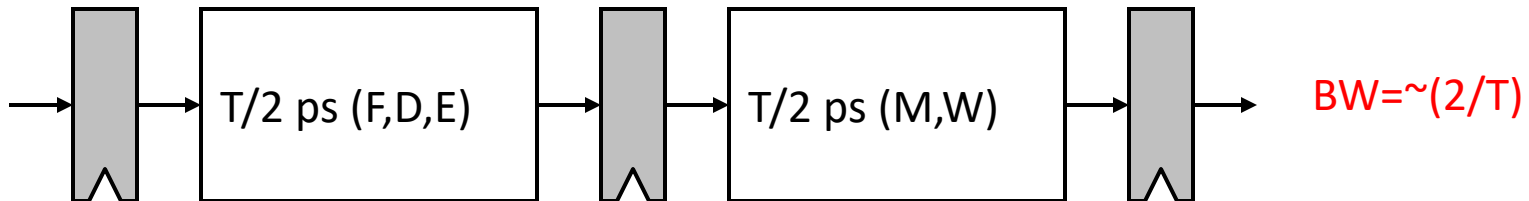
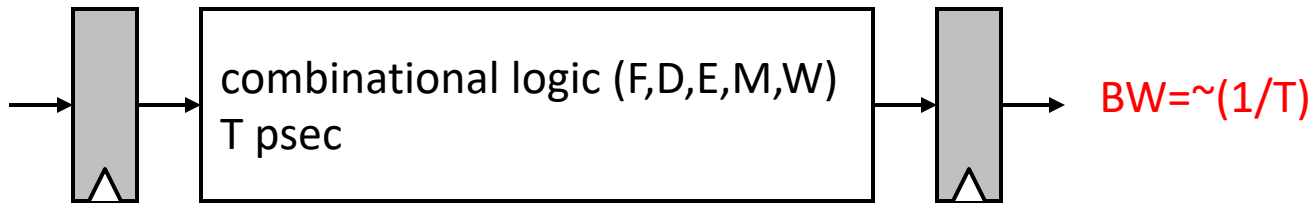
# An Ideal Pipeline

---

- Goal: **Increase throughput with little increase in cost** (hardware cost, in case of instruction processing)
- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of **independent operations**
  - No dependences between repeated operations
- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?

# Ideal Pipelining

BW means Bandwidth  
Same as Throughput (in this context)



# More Realistic Pipeline: Throughput

- Nonpipelined version with delay  $T$

$$BW = 1/(T+S) \text{ where } S = \text{register delay}$$

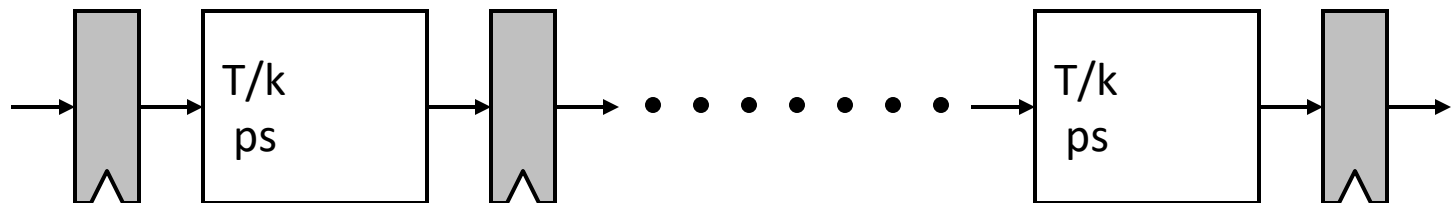


- k-stage pipelined version

$$BW_{k\text{-stage}} = 1 / (T/k + S)$$

$$BW_{\max} = 1 / (1 \text{ gate delay} + S)$$

**Register delay reduces throughput  
(sequencing overhead b/w stages)**



# More Realistic Pipeline: Cost

---

- Nonpipelined version with combinational cost  $G$

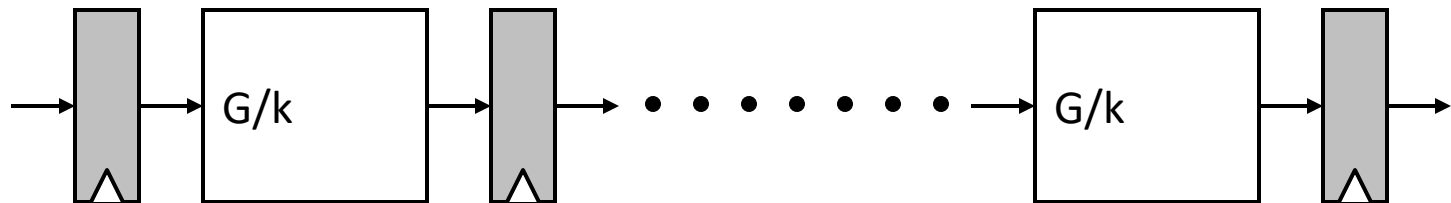
$\text{Cost} = G + R$  where  $R$  = register cost



- $k$ -stage pipelined version

$\text{Cost}_{k\text{-stage}} = G + Rk$

**Registers increase hardware cost**

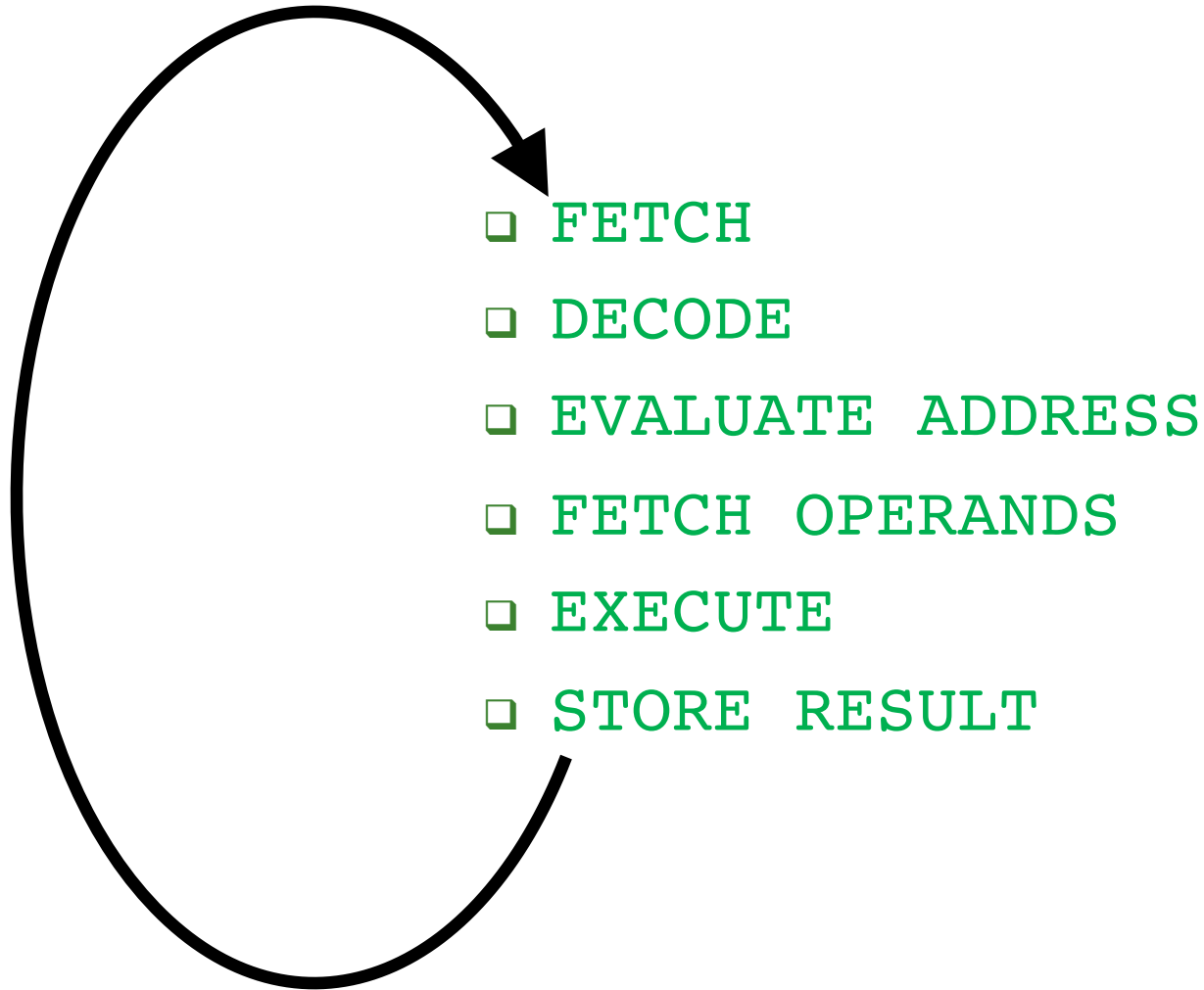




# Pipelining Instruction Processing

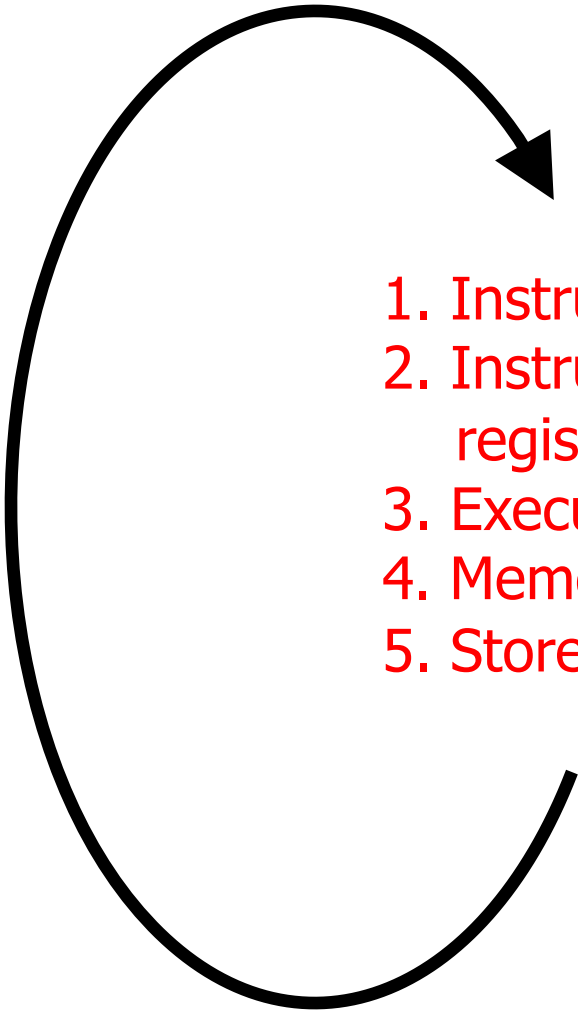
# Remember: The Instruction Processing Cycle

---

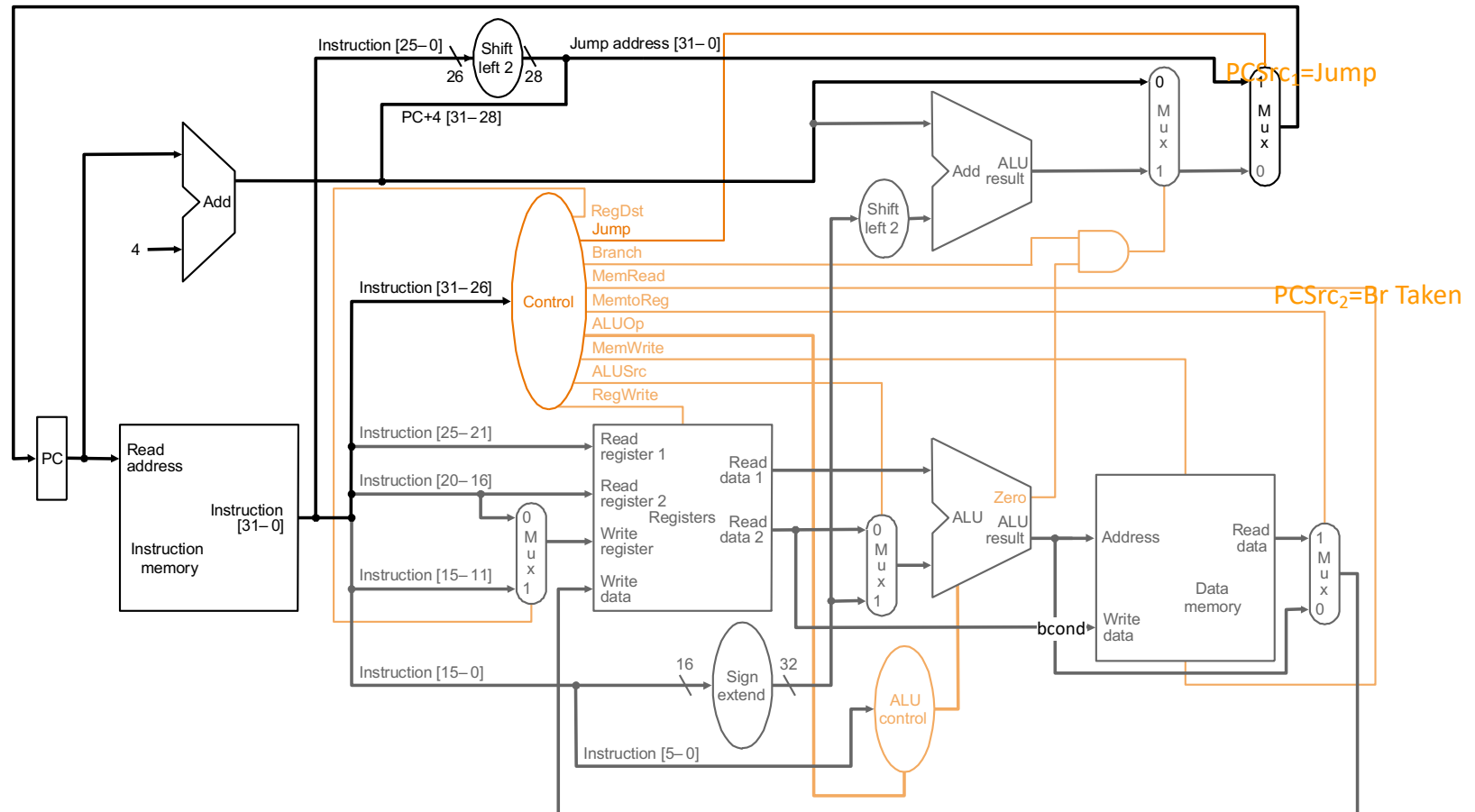


# Remember: The Instruction Processing Cycle

---

- 
1. Instruction fetch (IF)
  2. Instruction decode and register operand fetch (ID/RF)
  3. Execute/Evaluate memory address (EX/AG)
  4. Memory operand fetch (MEM)
  5. Store/writeback result (WB)

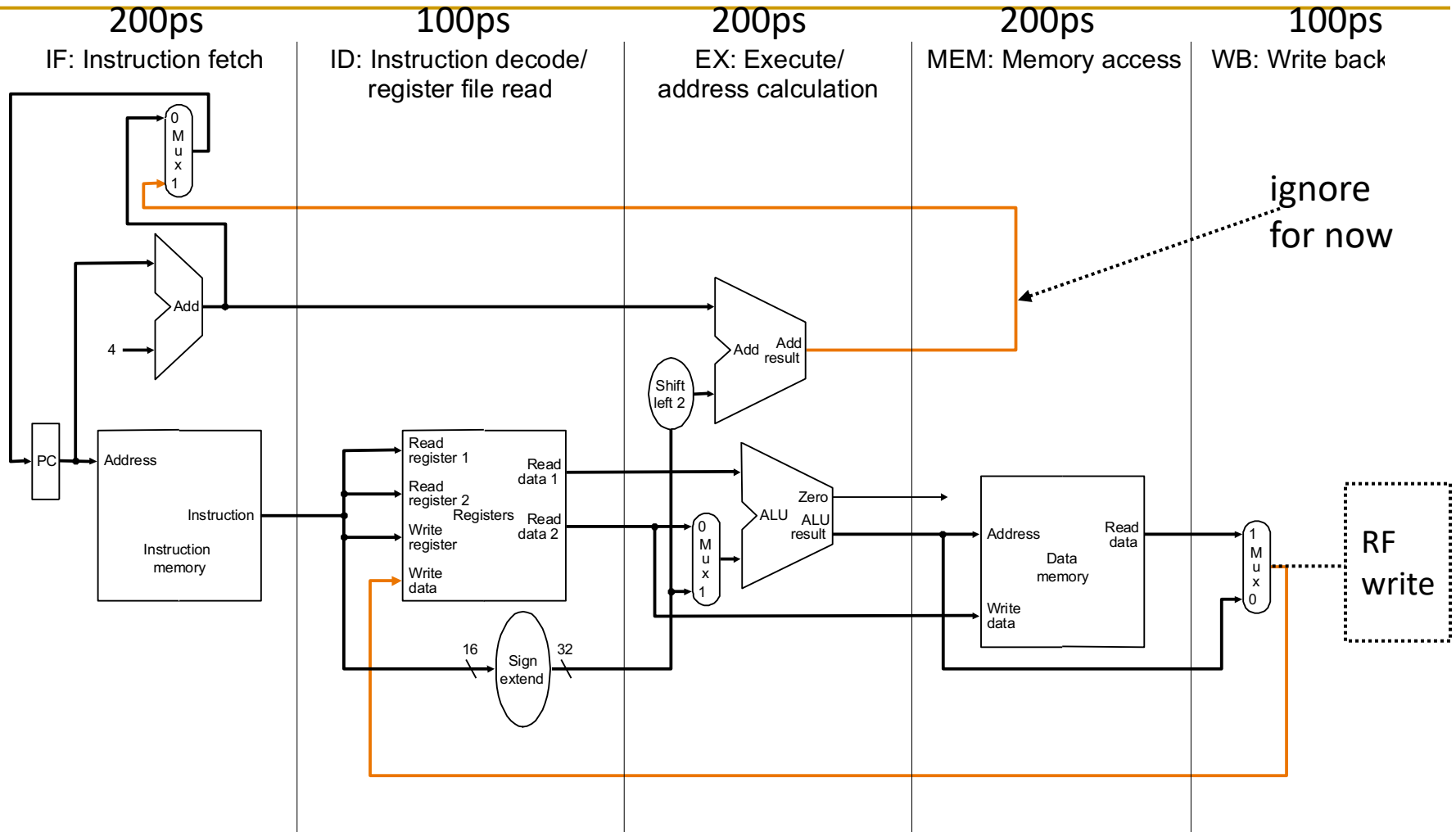
# Remember the Single-Cycle Uarch



ALU operation



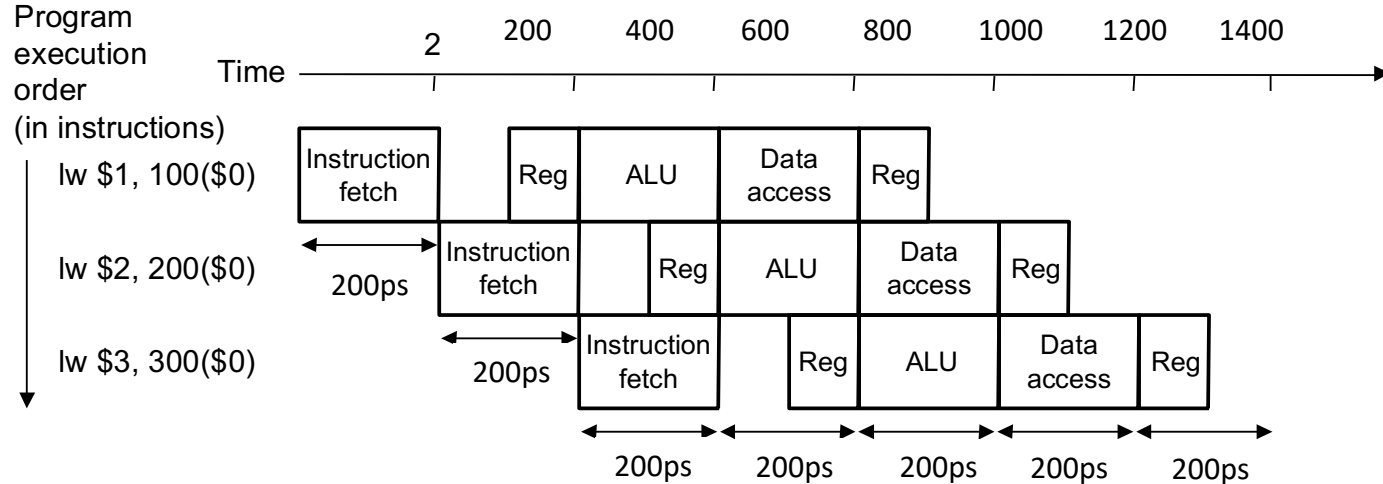
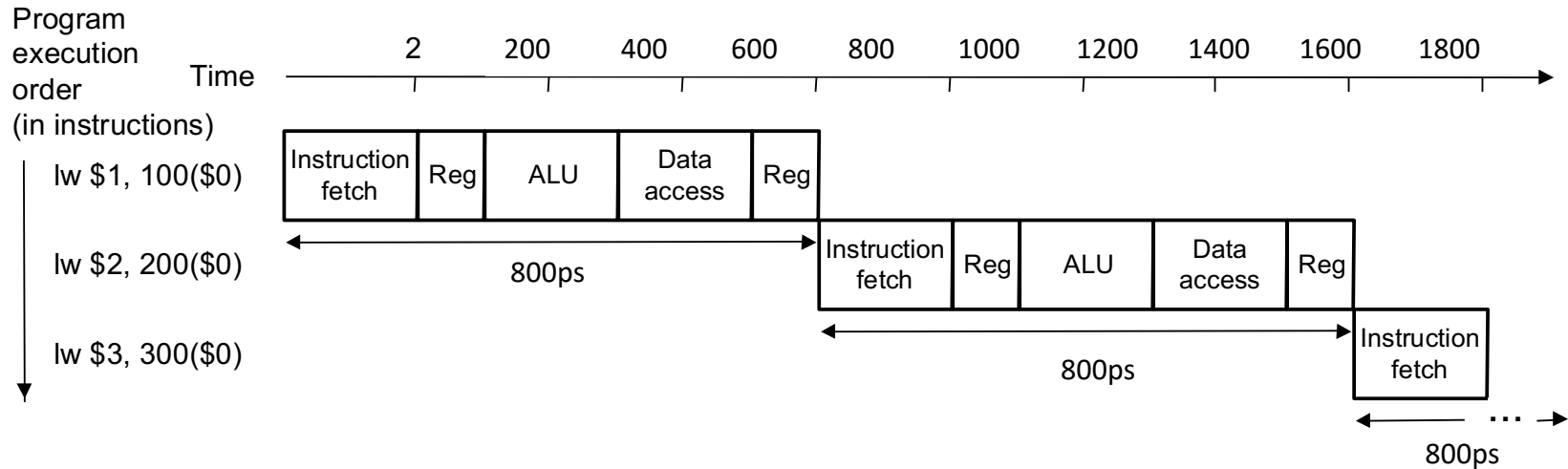
# Dividing Into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

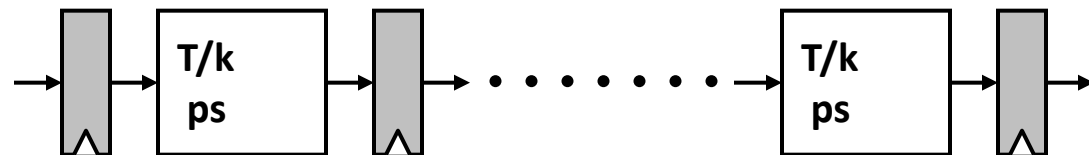
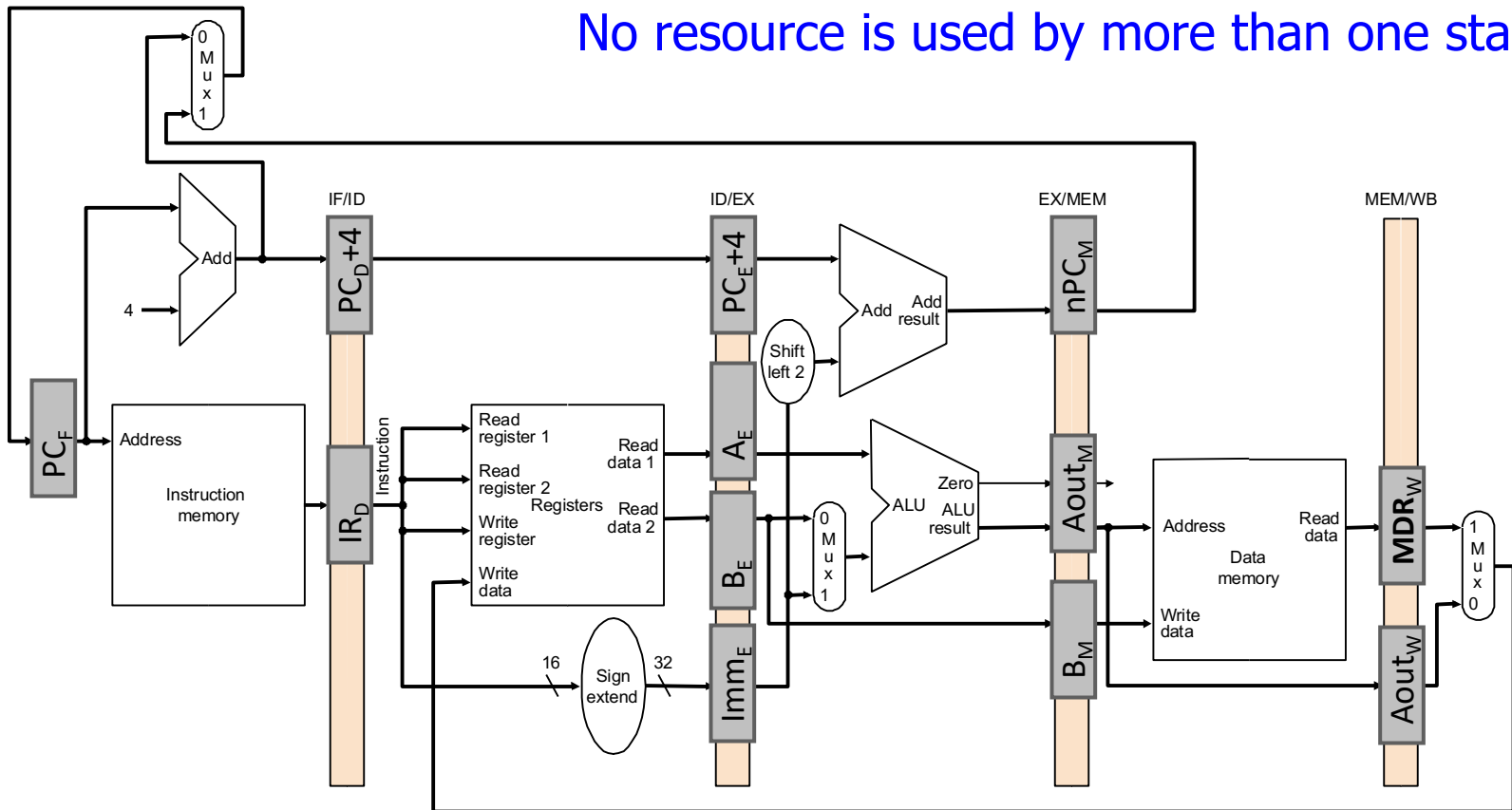
# Instruction Pipeline Throughput



5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

# Enabling Pipelined Processing: Pipeline Registers

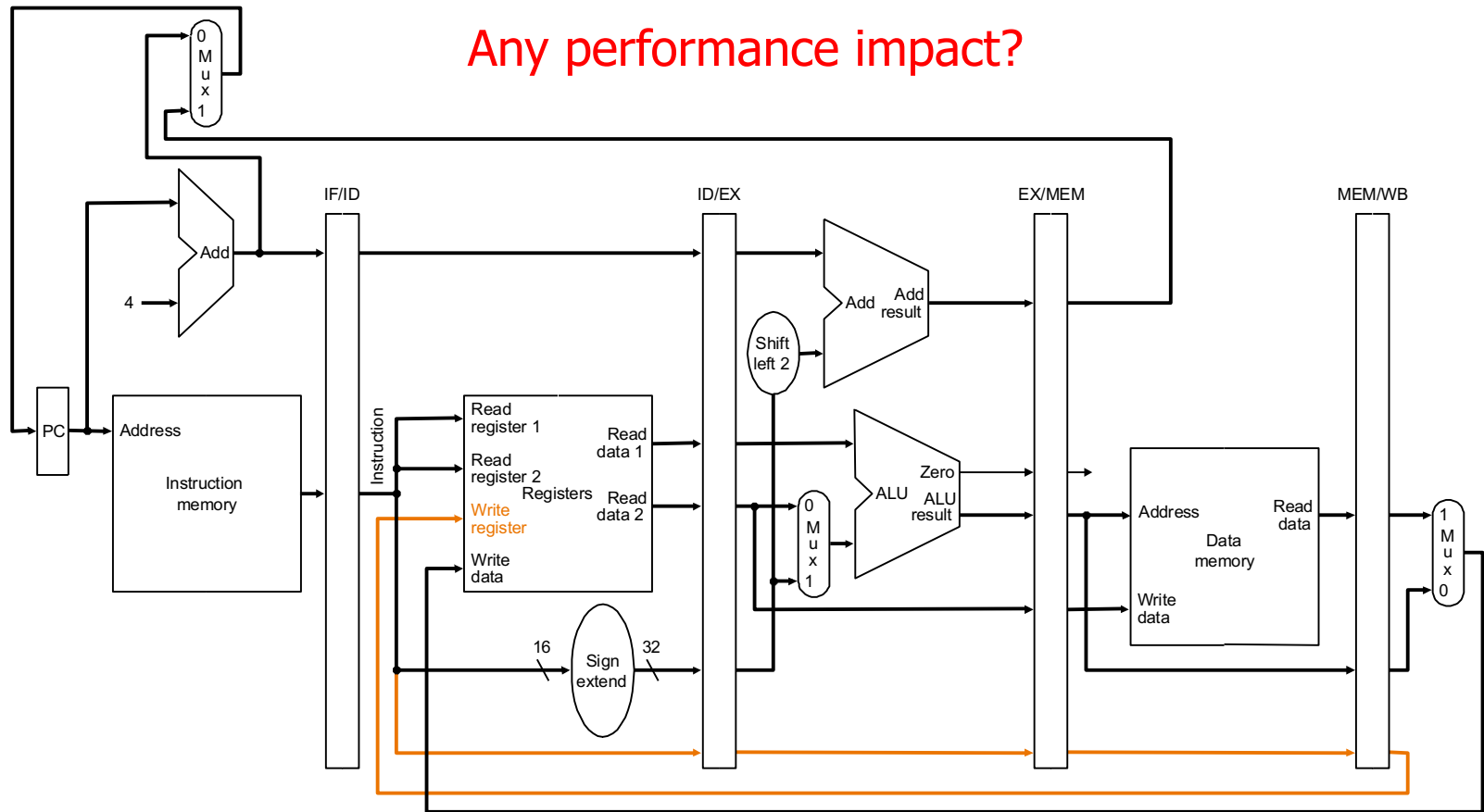
No resource is used by more than one stage



# Pipelined Operation Example

All instruction classes must follow the same path and timing through the pipeline stages.

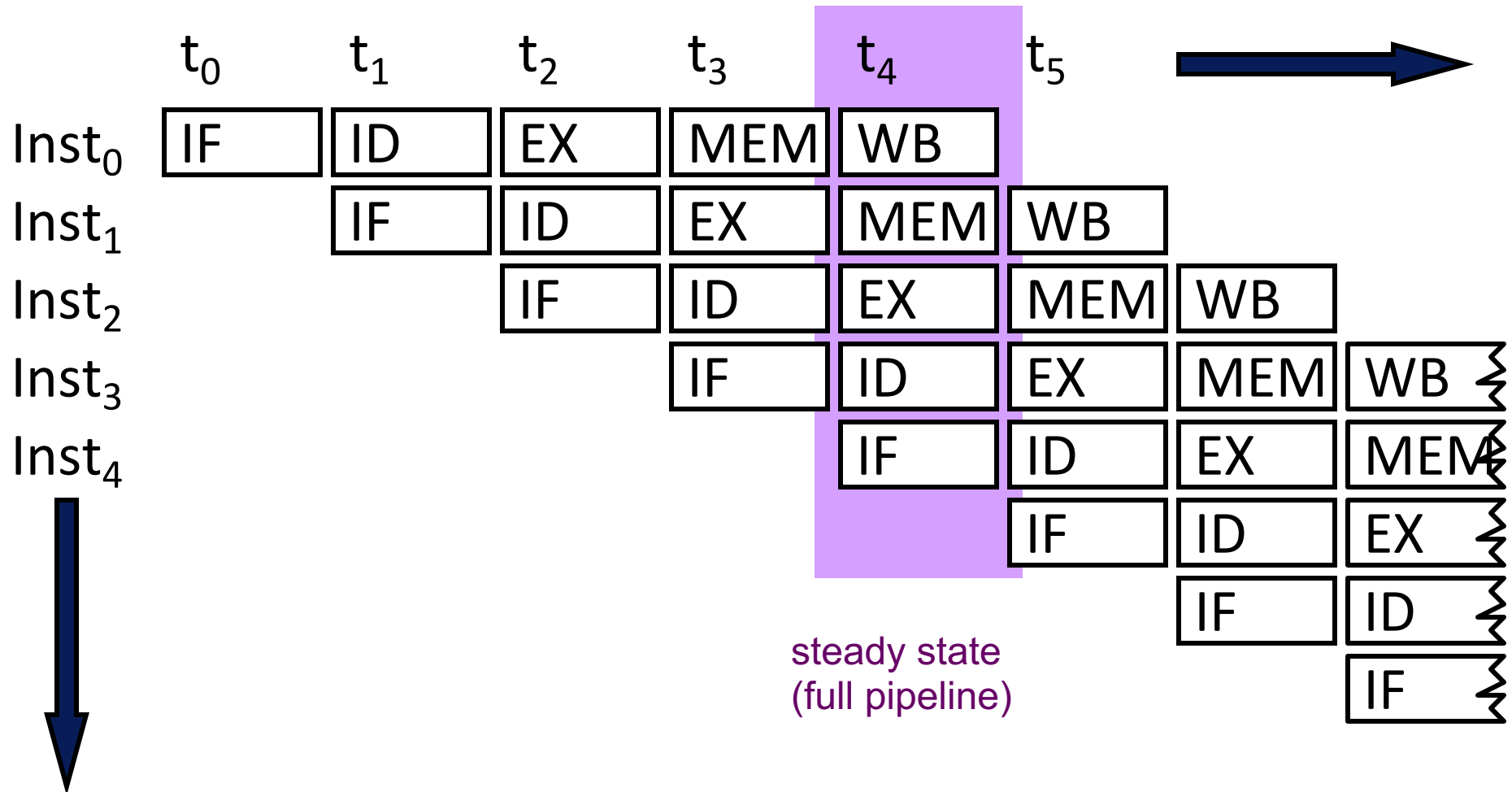
Any performance impact?







# Illustrating Pipeline Operation: Operation View

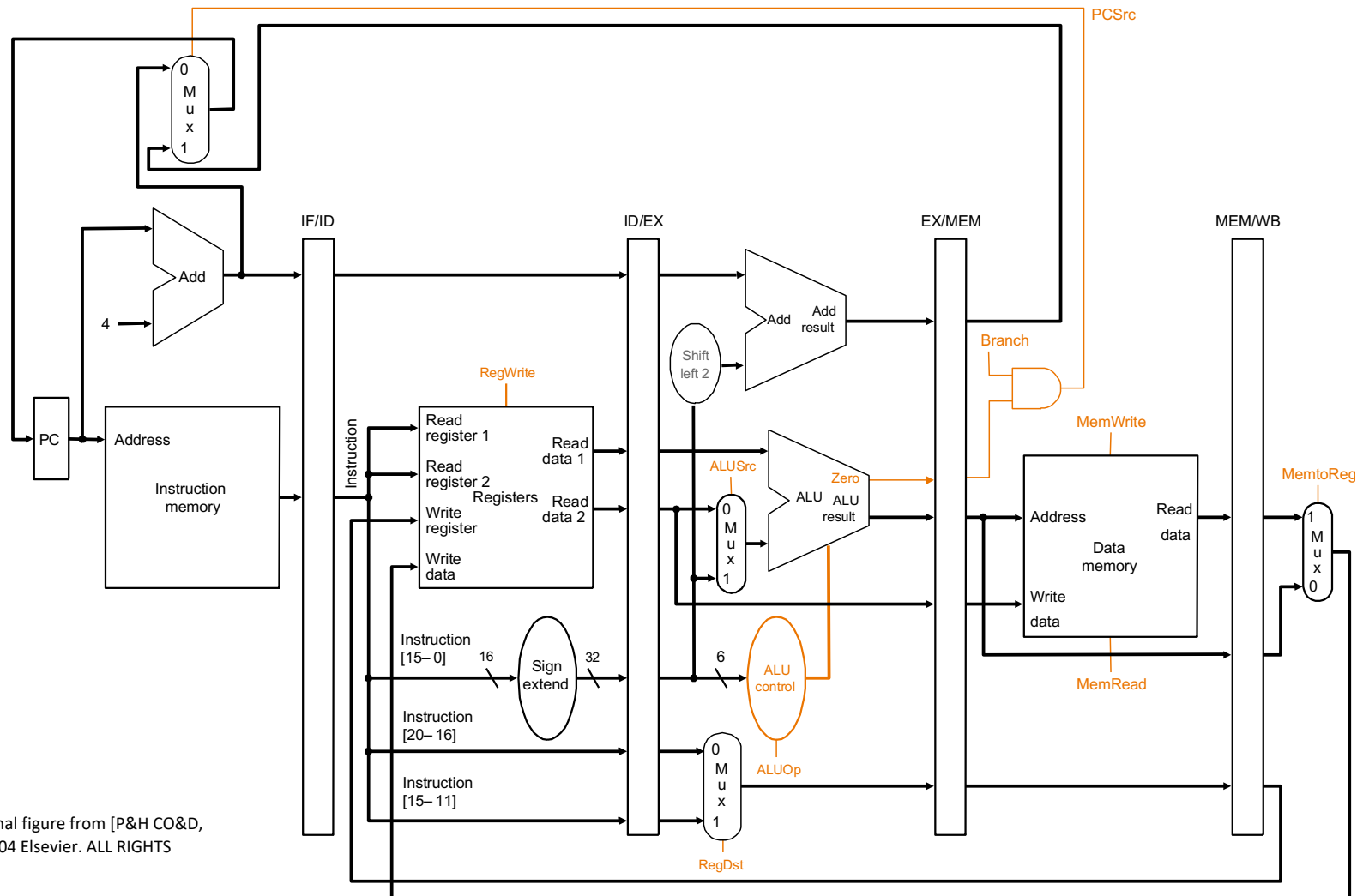


# Illustrating Pipeline Operation: Resource View

---

	t <sub>0</sub>	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>	t <sub>7</sub>	t <sub>8</sub>	t <sub>9</sub>	t <sub>10</sub>
IF	l <sub>0</sub>	l <sub>1</sub>	l <sub>2</sub>	l <sub>3</sub>	l <sub>4</sub>	l <sub>5</sub>	l <sub>6</sub>	l <sub>7</sub>	l <sub>8</sub>	l <sub>9</sub>	l <sub>10</sub>
ID		l <sub>0</sub>	l <sub>1</sub>	l <sub>2</sub>	l <sub>3</sub>	l <sub>4</sub>	l <sub>5</sub>	l <sub>6</sub>	l <sub>7</sub>	l <sub>8</sub>	l <sub>9</sub>
EX			l <sub>0</sub>	l <sub>1</sub>	l <sub>2</sub>	l <sub>3</sub>	l <sub>4</sub>	l <sub>5</sub>	l <sub>6</sub>	l <sub>7</sub>	l <sub>8</sub>
MEM				l <sub>0</sub>	l <sub>1</sub>	l <sub>2</sub>	l <sub>3</sub>	l <sub>4</sub>	l <sub>5</sub>	l <sub>6</sub>	l <sub>7</sub>
WB					l <sub>0</sub>	l <sub>1</sub>	l <sub>2</sub>	l <sub>3</sub>	l <sub>4</sub>	l <sub>5</sub>	l <sub>6</sub>

# Control Points in a Pipeline



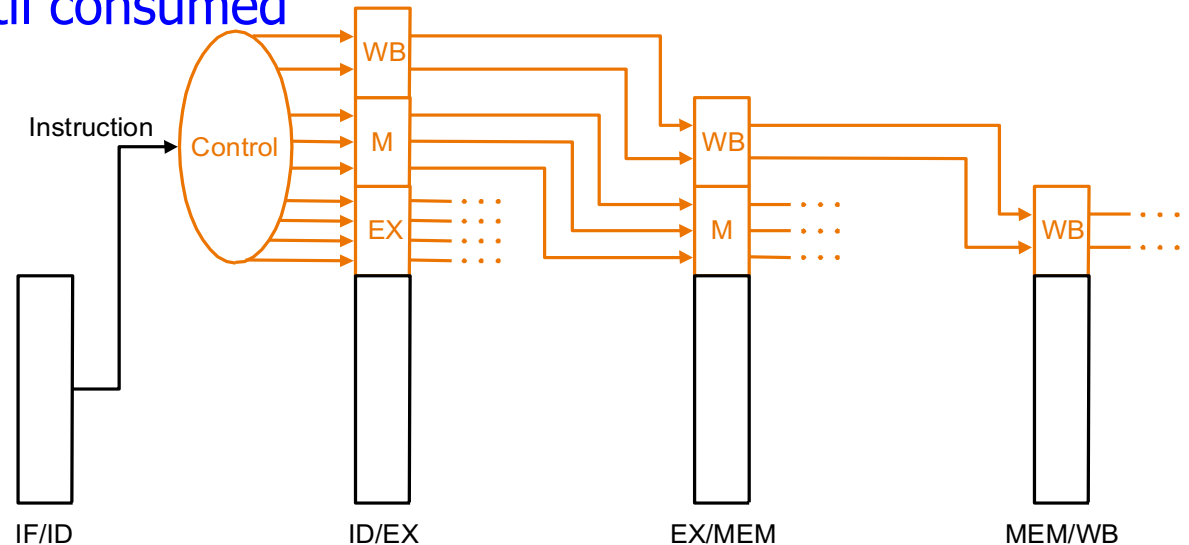
Based on original figure from [P&H CO&D,  
COPYRIGHT 2004 Elsevier. ALL RIGHTS  
RESERVED.]

Identical set of control points as the single-cycle datapath

# Control Signals in a Pipeline

## ■ For a given instruction

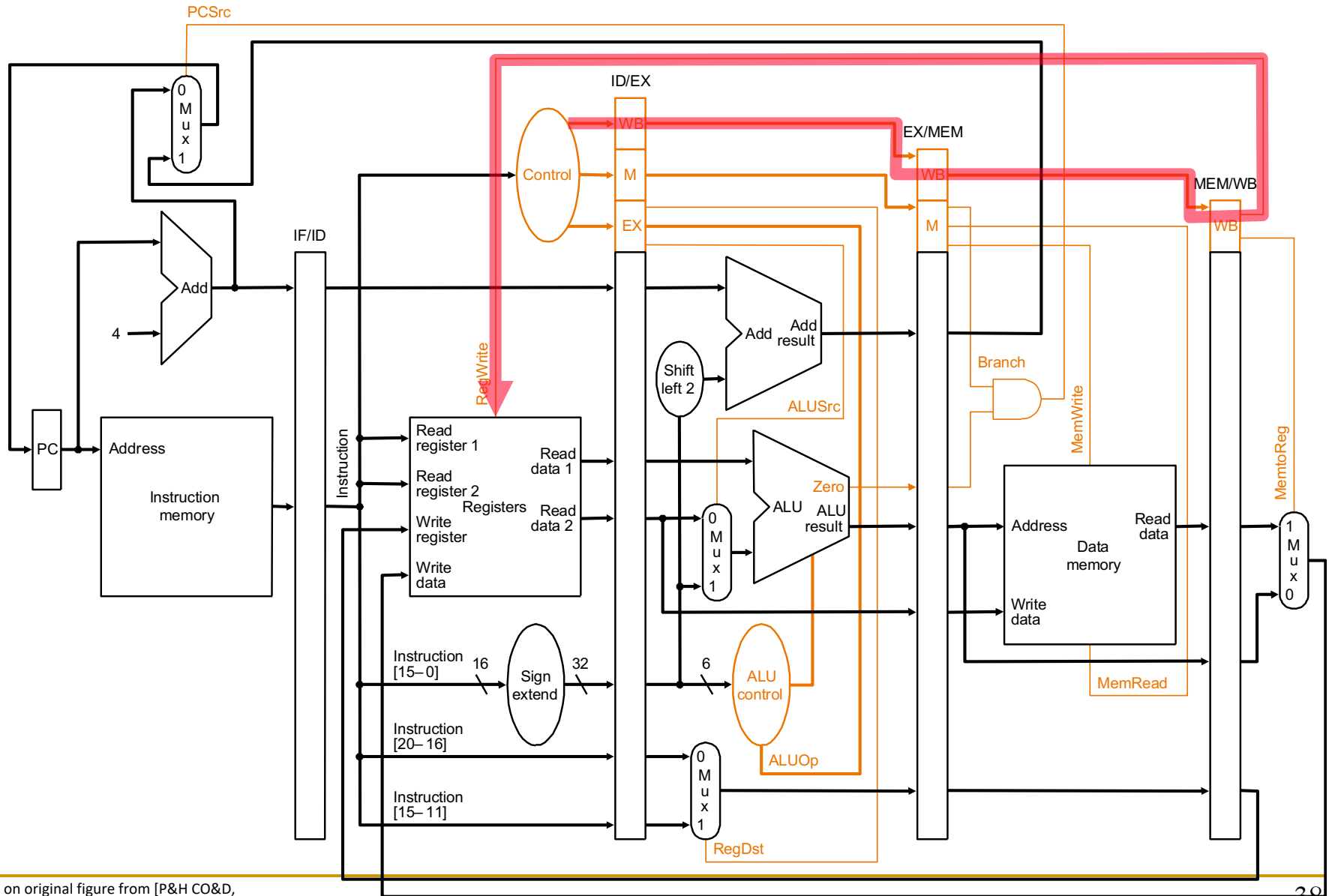
- same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
- ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



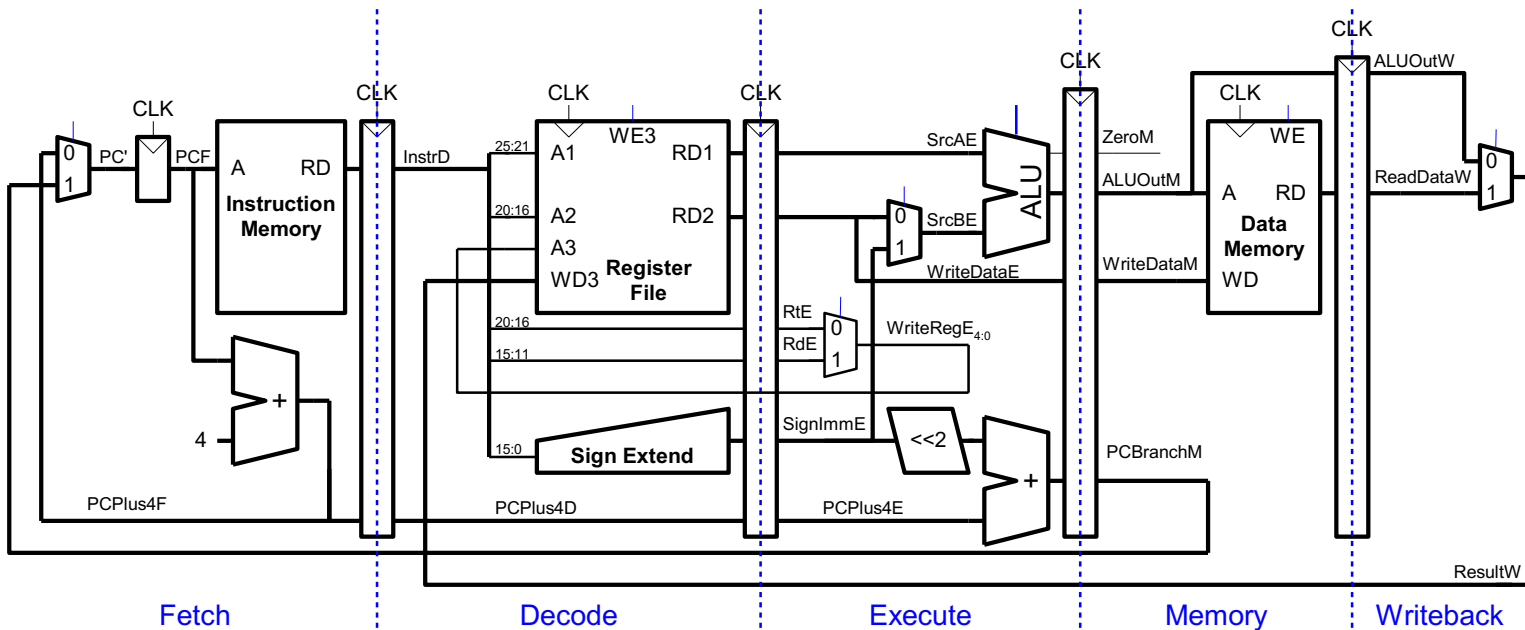
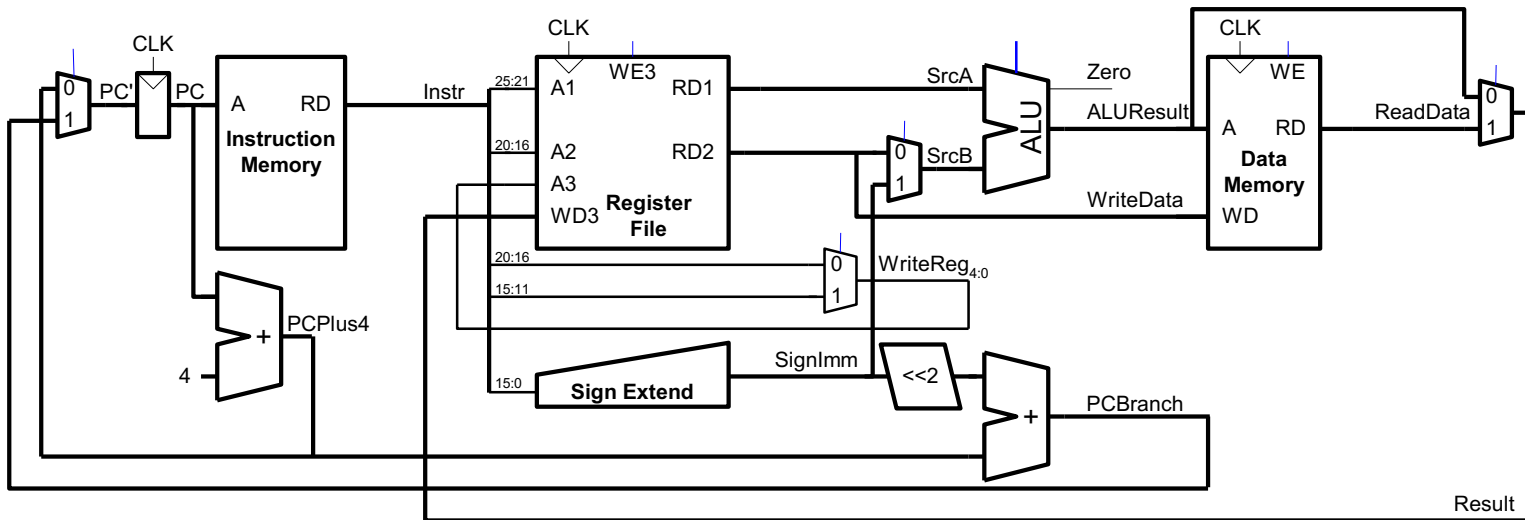
- ⇒ Option 2: carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

Which one is better?

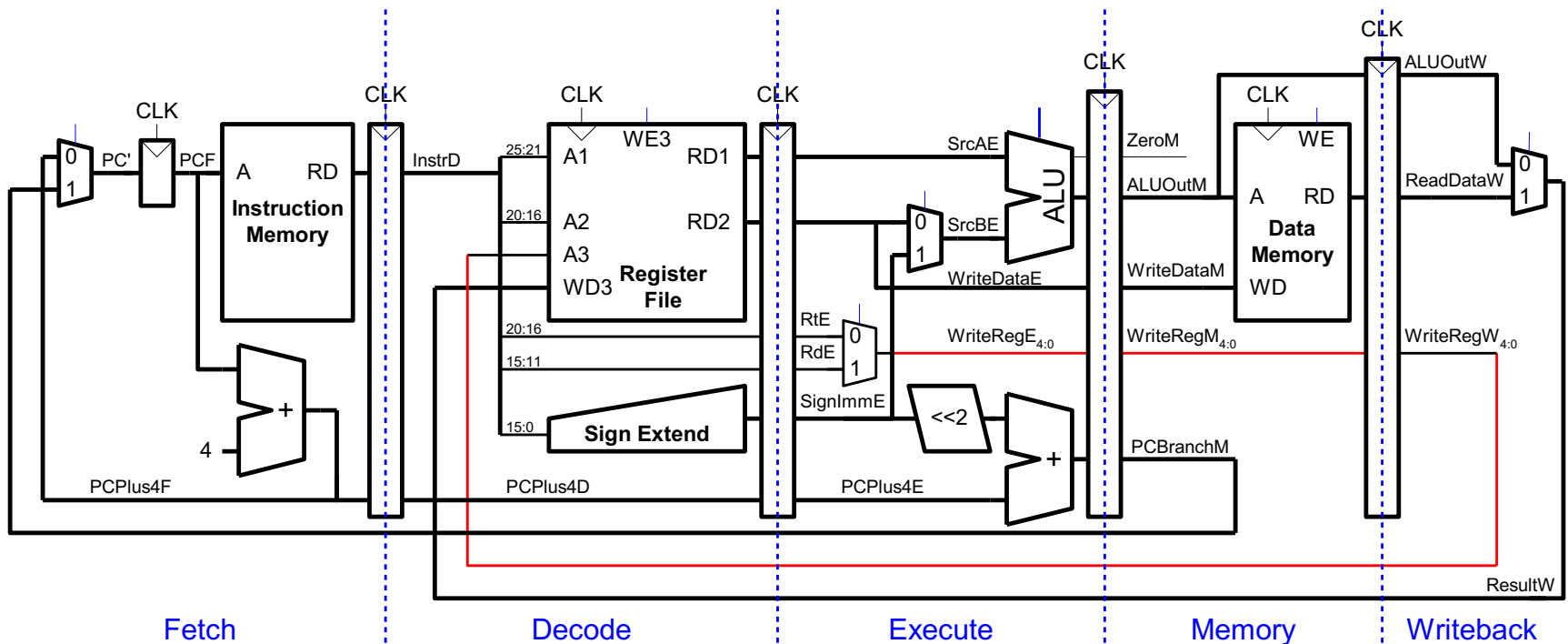
# Pipelined Control Signals



# Another Example: Single-Cycle and Pipelined



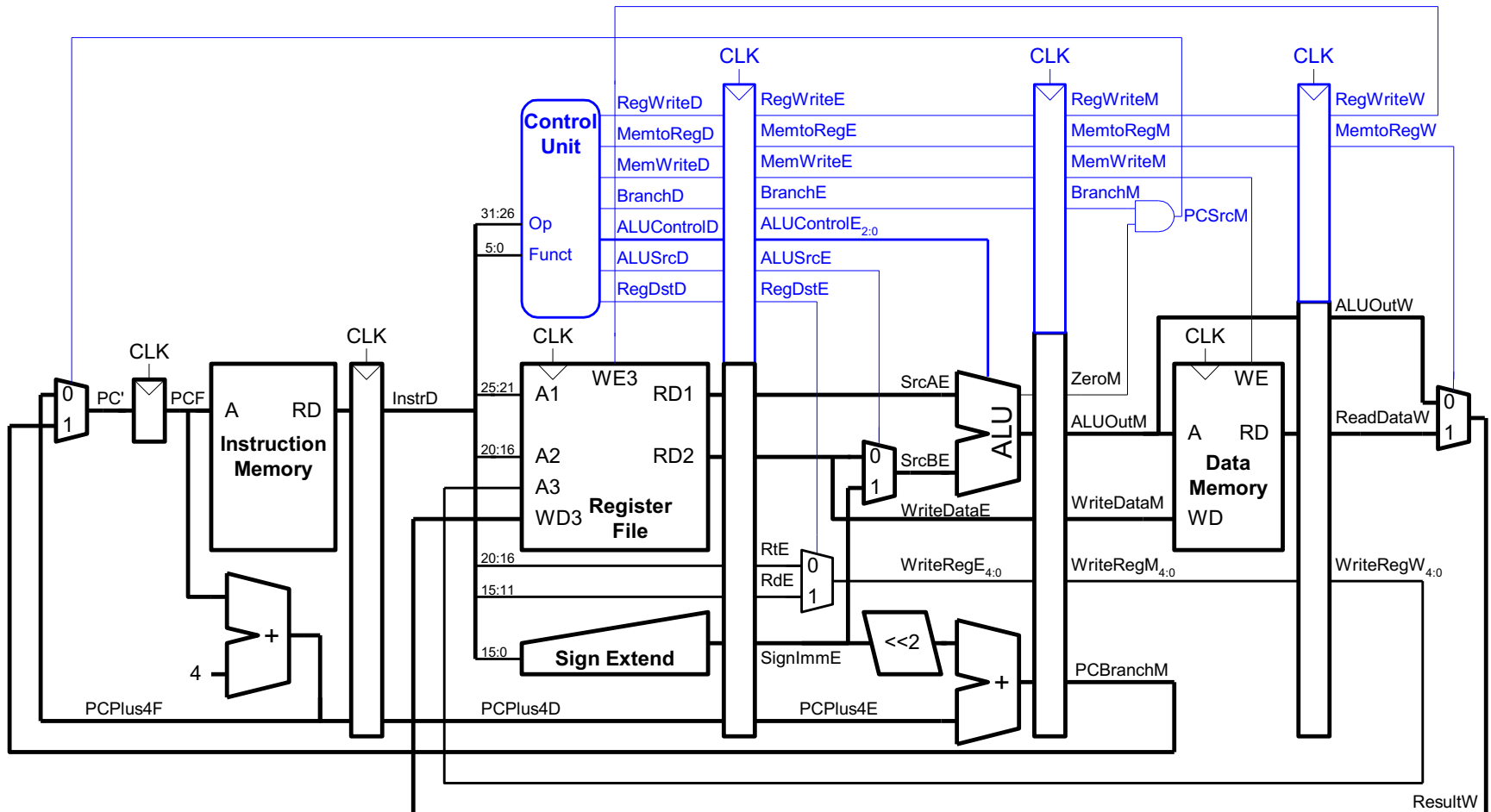
# Another Example: **Correct** Pipelined Datapath



- **WriteReg control signal must arrive at the same time as Result**



# Another Example: Pipelined Control



- Same control unit as single-cycle processor  
Control delayed to proper pipeline stage

# Remember: An Ideal Pipeline

---

- Goal: **Increase throughput with little increase in cost** (hardware cost, in case of instruction processing)
- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of **independent operations**
  - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - **What about the instruction processing “cycle”?**

# Instruction Pipeline: Not An Ideal Pipeline

---

- Identical operations ... NOT!

- ⇒ different instructions → not all need the same stages

- Forcing different instructions to go through the same pipe stages

- external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

- ⇒ different pipeline stages → not the same latency

- Need to force each stage to be controlled by the same clock

- internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)

- Independent operations ... NOT!

- ⇒ instructions are not independent of each other

- Need to detect and resolve inter-instruction dependences to ensure the pipeline provides correct results

- pipeline stalls (pipeline is not always moving)

# Issues in Pipeline Design

---

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing *stalls*

# Causes of Pipeline *Stalls*

---

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

# Dependences and Their Types

---

- Also called “dependency” or *less desirably* “hazard”
- Dependences dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately

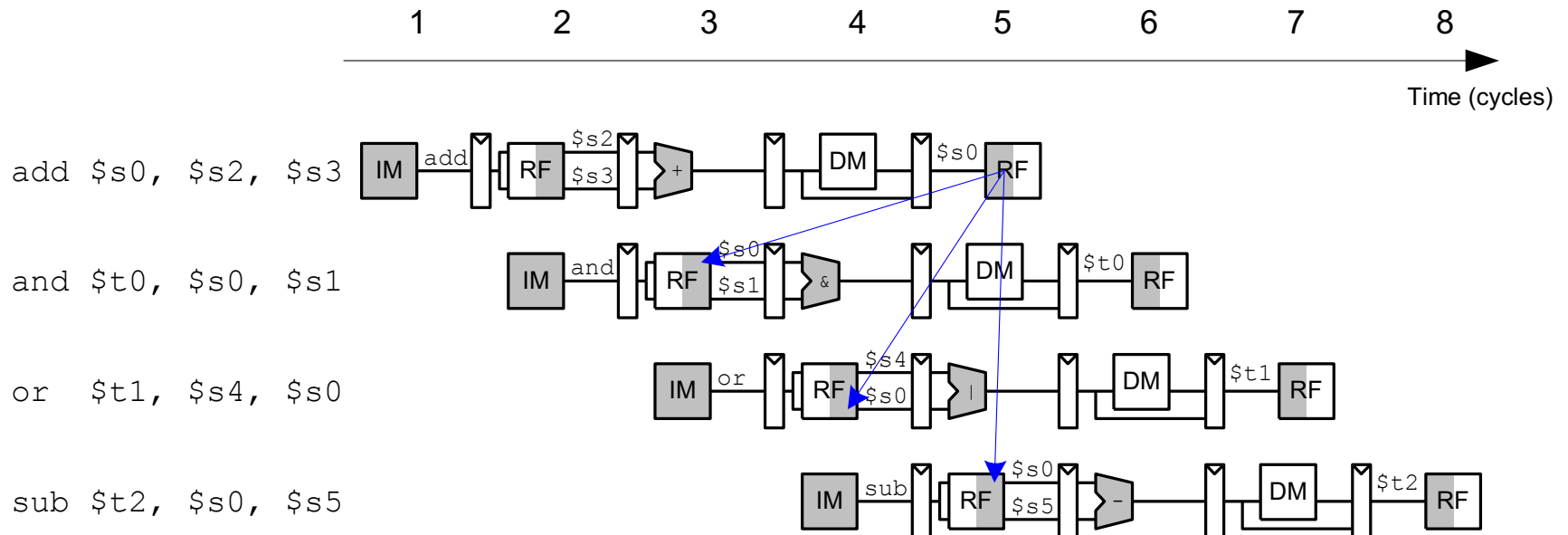
# Handling Resource Contention

---

- Happens when instructions in two pipeline stages need the same resource
- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures
- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?

# Example Resource Dependence: RegFile

- The register file can be read and written in the same cycle:
  - write takes place during the 1st half of the cycle
  - read takes place during the 2nd half of the cycle => no problem!!!
  - However, operations that involve register file have only *half a clock cycle* to complete the operation...





# Data Dependences

---


- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)
- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value
    - We will later see what we can do about them

# Data Dependence Types

---

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR)

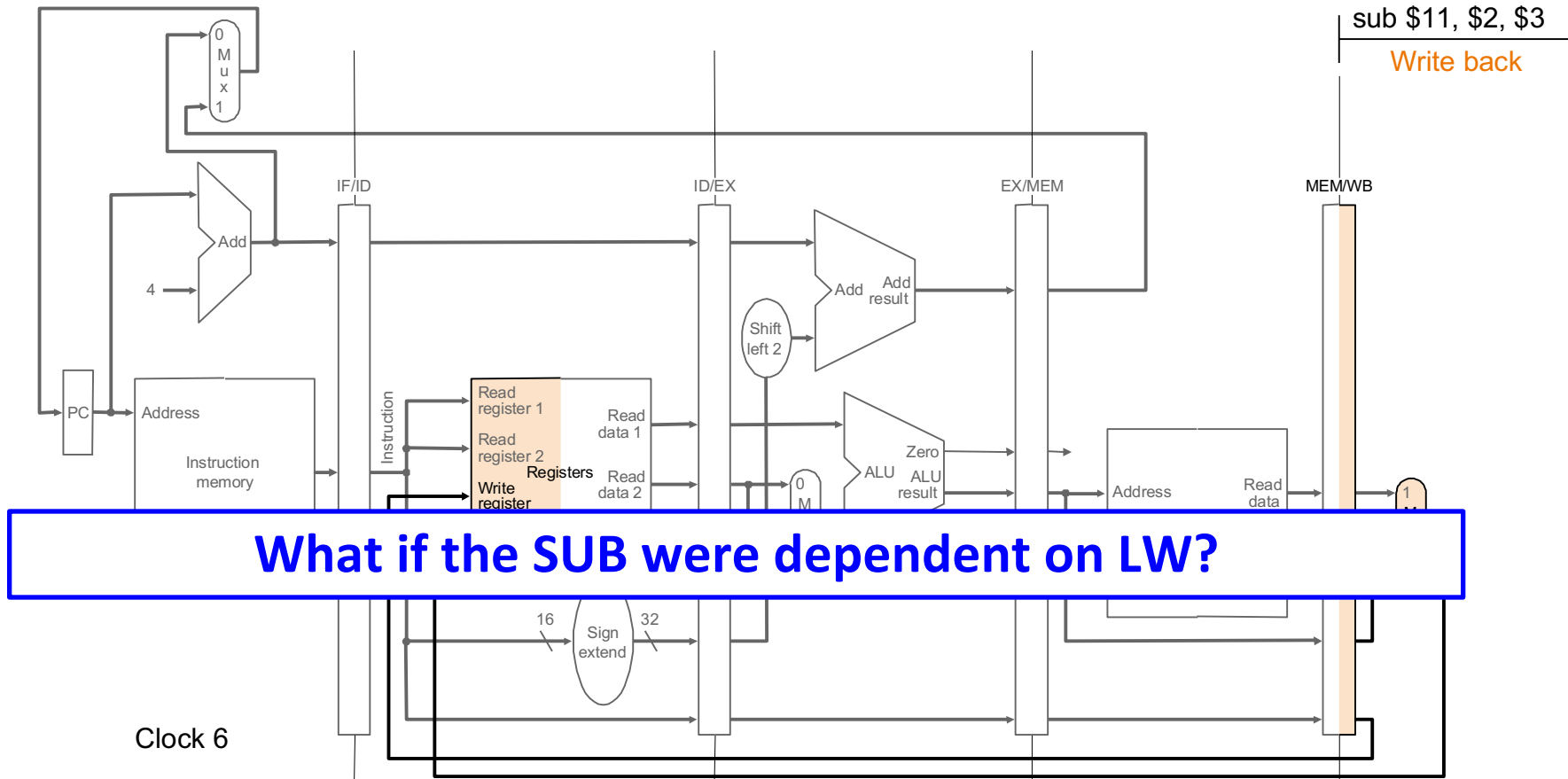
Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write  
(WAW)

# Pipelined Operation Example



Clock 6

sub \$11, \$2, \$3

Write back

MEM/WB

EX/MEM

ID/EX

IF/ID

4

PC

Address

Instruction memory

Instruction

Read register 1

Read register 2

Write register

Read data 1

Read data 2

Registers

Shift left 2

Add

Add result

Zero

ALU result

0

M

Address

Read data

1

16

Sign extend

32

# Data Dependence Handling

# Reading for Next Few Lectures

---

- H&H, Chapter 7.5-7.9
- Smith and Sohi, “[The Microarchitecture of Superscalar Processors](#),” Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

# How to Handle Data Dependences

---


- Anti and output dependences are easier to handle
  - write to the destination only in last stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

# Remember: Data Dependence Types

---

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR)

Output-dependence

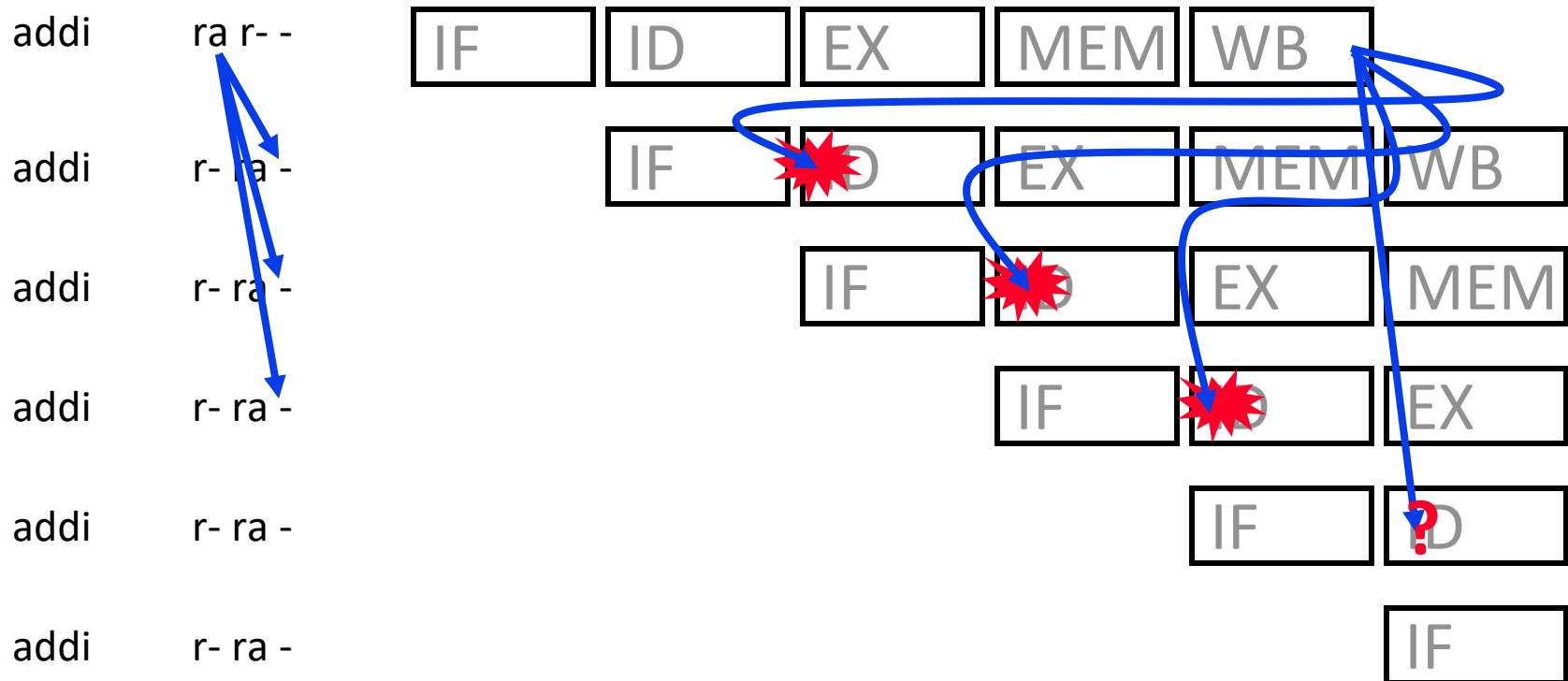
$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write  
(WAW)

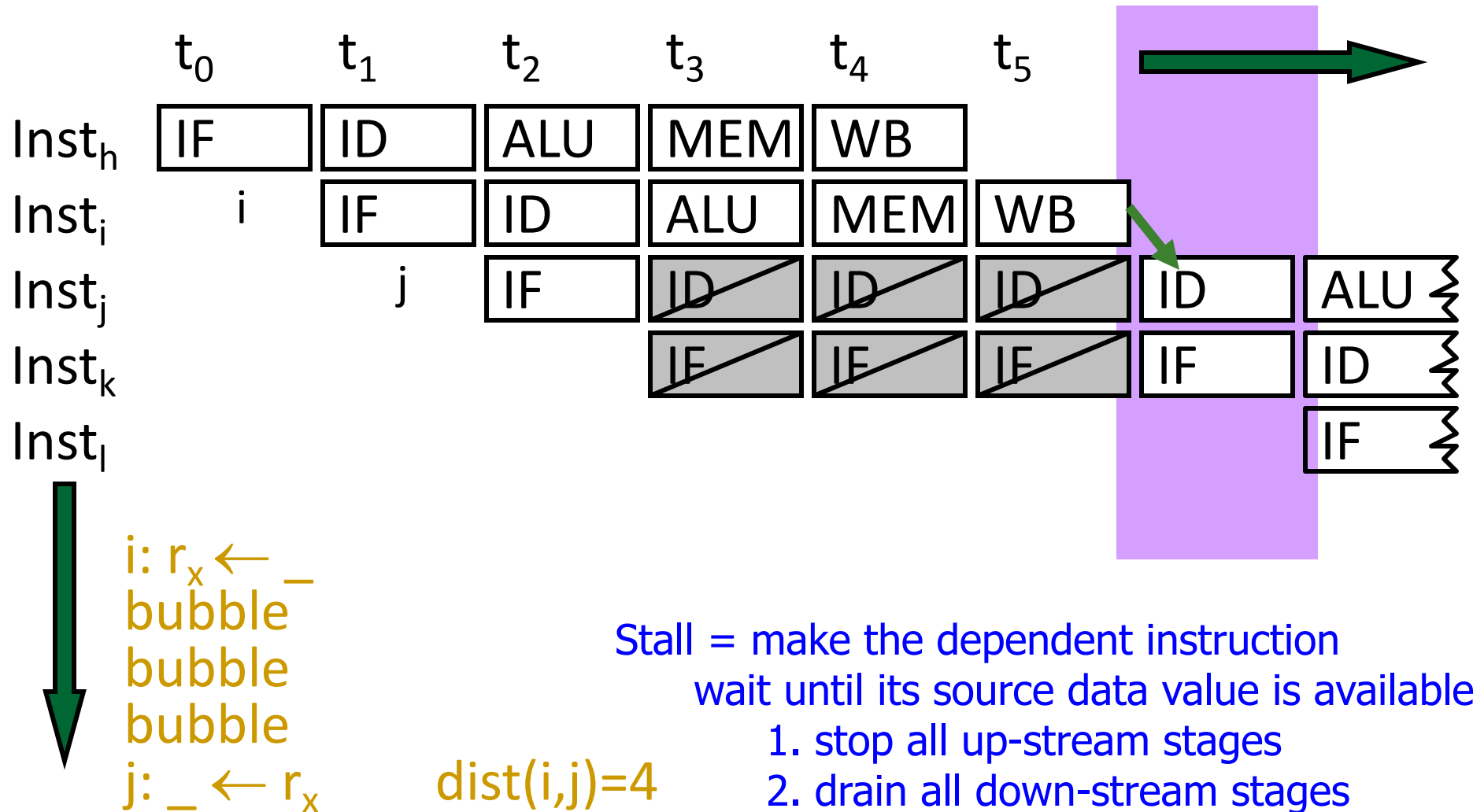
# RAW Dependence Handling

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?





# Pipeline Stall: Resolving Data Dependence



# Interlocking

---

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- Software based interlocking  
vs.
- Hardware based interlocking
- MIPS acronym?

# Approaches to Dependence Detection (I)

---

## ■ Scoreboarding

- ❑ Each register in register file has a Valid bit associated with it
- ❑ An instruction that is writing to the register resets the Valid bit
- ❑ An instruction in Decode stage checks if all its source and destination registers are Valid
  - Yes: No need to stall... No dependence
  - No: Stall the instruction

## ■ Advantage:

- ❑ Simple. 1 bit per register

## ■ Disadvantage:

- ❑ Need to stall for all types of dependences, not only flow dep.

# Approaches to Dependence Detection (II)

---

## ■ Combinational dependence check logic

- ❑ Special logic checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
- ❑ Yes: stall the instruction/pipeline
- ❑ No: no need to stall... no flow dependence

## ■ Advantage:

- ❑ No need to stall on anti and output dependences

## ■ Disadvantage:

- ❑ Logic is more complex than a scoreboard
- ❑ Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

# Once You Detect the Dependence in Hardware

---

- What do you do afterwards?
- Observation: Dependence between two instructions is detected before the communicated data value becomes available
- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: ...

# Data Forwarding/Bypassing

---

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

# Aside: A Special Case of Data Dependence

---

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter

# Aside: Control Dependence

---

- Question: **What should the fetch PC be in the next cycle?**
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?



# Digital Design & Computer Arch.

## Lecture 13: Pipelining

Prof. Onur Mutlu

ETH Zürich

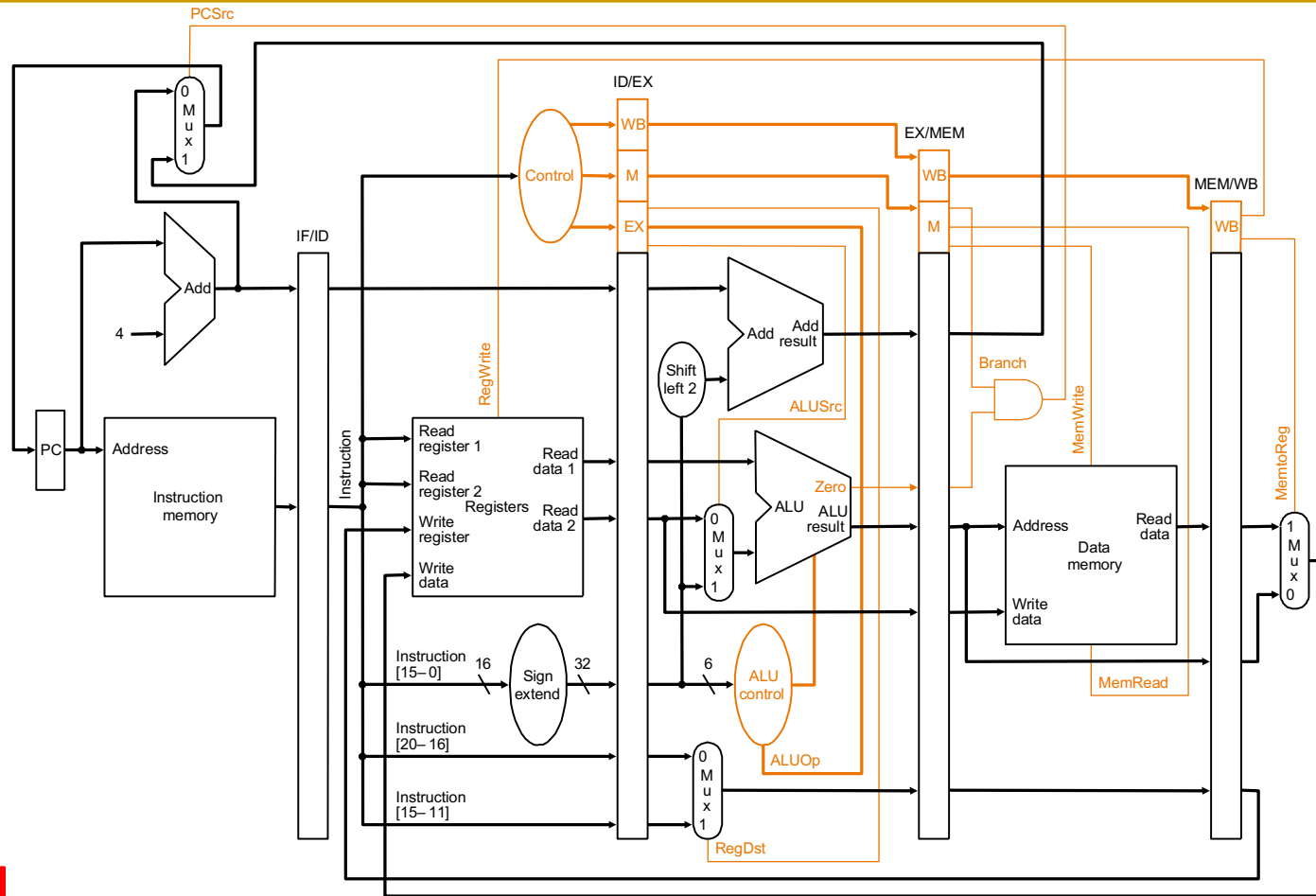
Spring 2021

16 April 2021

We did not cover the following slides.  
They are for your benefit.  
We will cover them in future lectures.

# Data Dependence Handling: Concepts and Implementation

# How to Implement Stalling

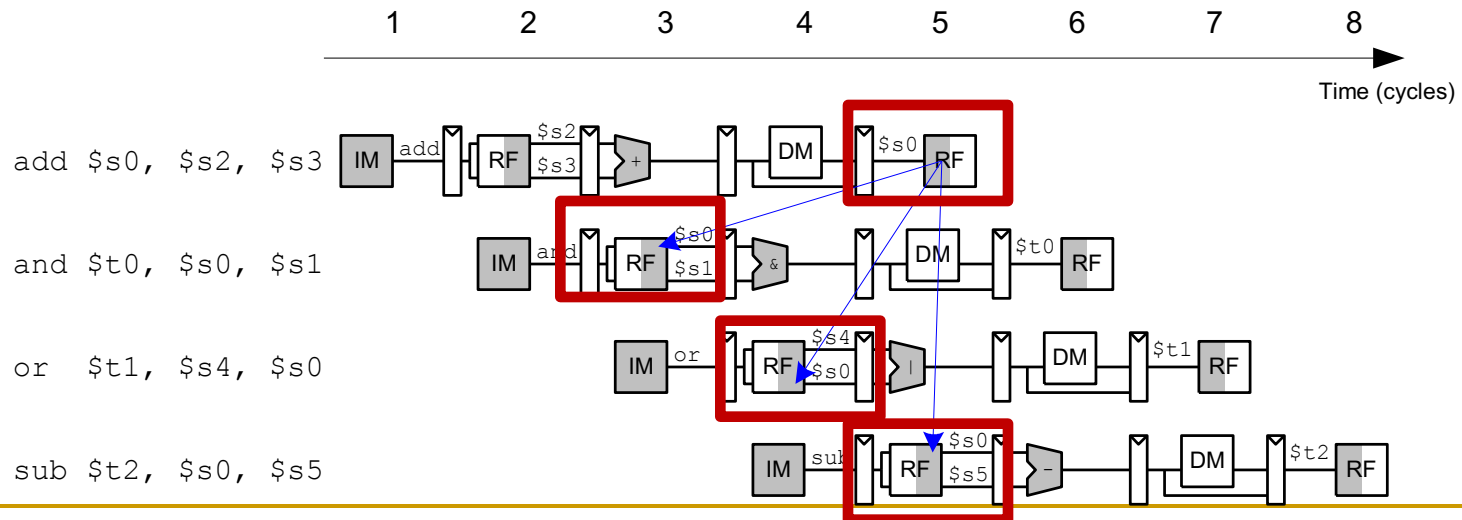


## ■ Stall

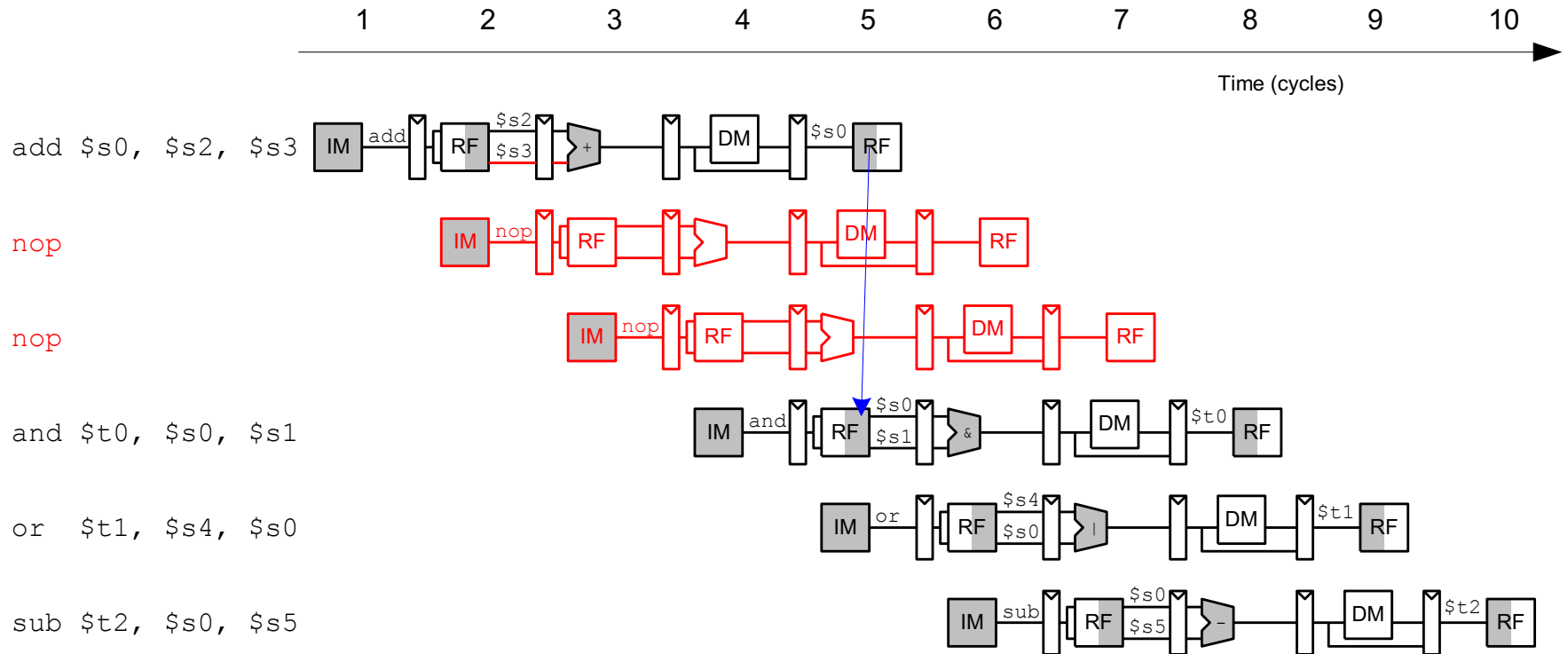
- ❑ disable **PC** and **IF/ID** latching; ensure stalled instruction stays in its stage
- ❑ Insert **"invalid"** instructions/nops into the stage following the stalled one (called **"bubbles"**)

# RAW Data Dependence Example

- One instruction writes a register (\$s0) and next instructions read this register => read after write (RAW) dependence.
- **add** writes into \$s0 in the first half of cycle 5
- **sub** reads \$s0 in 2nd half of cycle 5, getting the correct value
- subsequent instructions read the correct value of \$s0



# Compile-Time Detection and Elimination



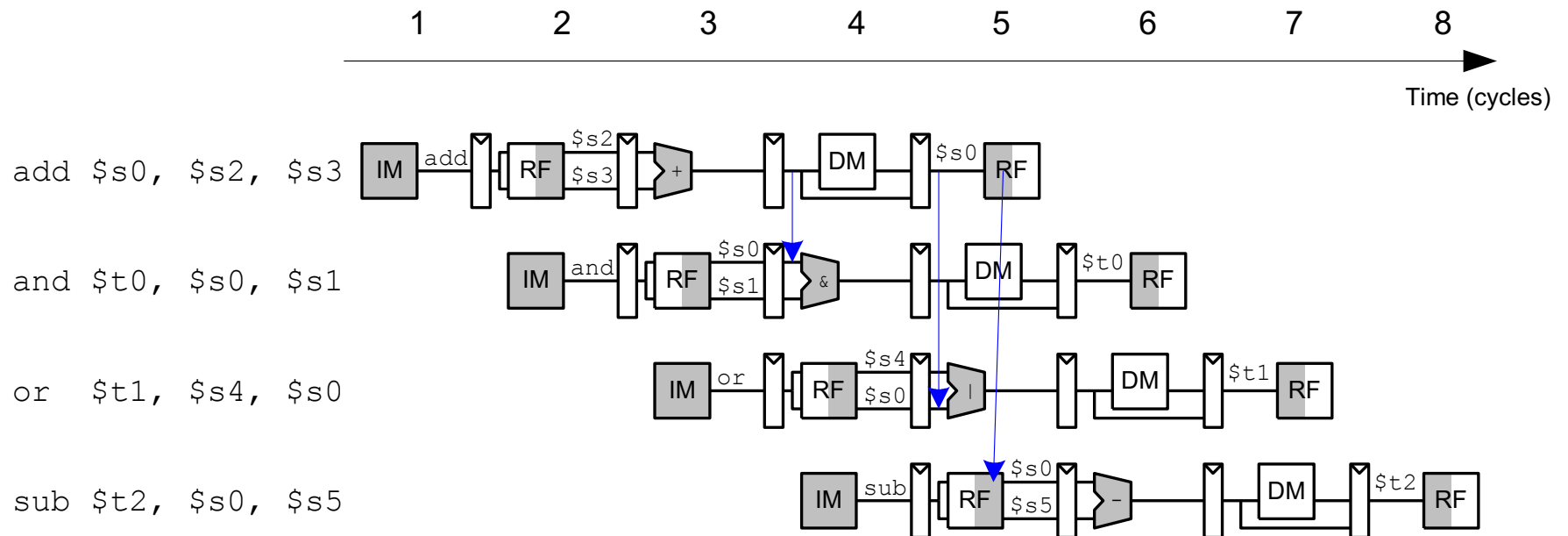
- Insert enough NOPs for the required result to be ready
- Or (if you can) move independent useful instructions up

# Data Forwarding

---

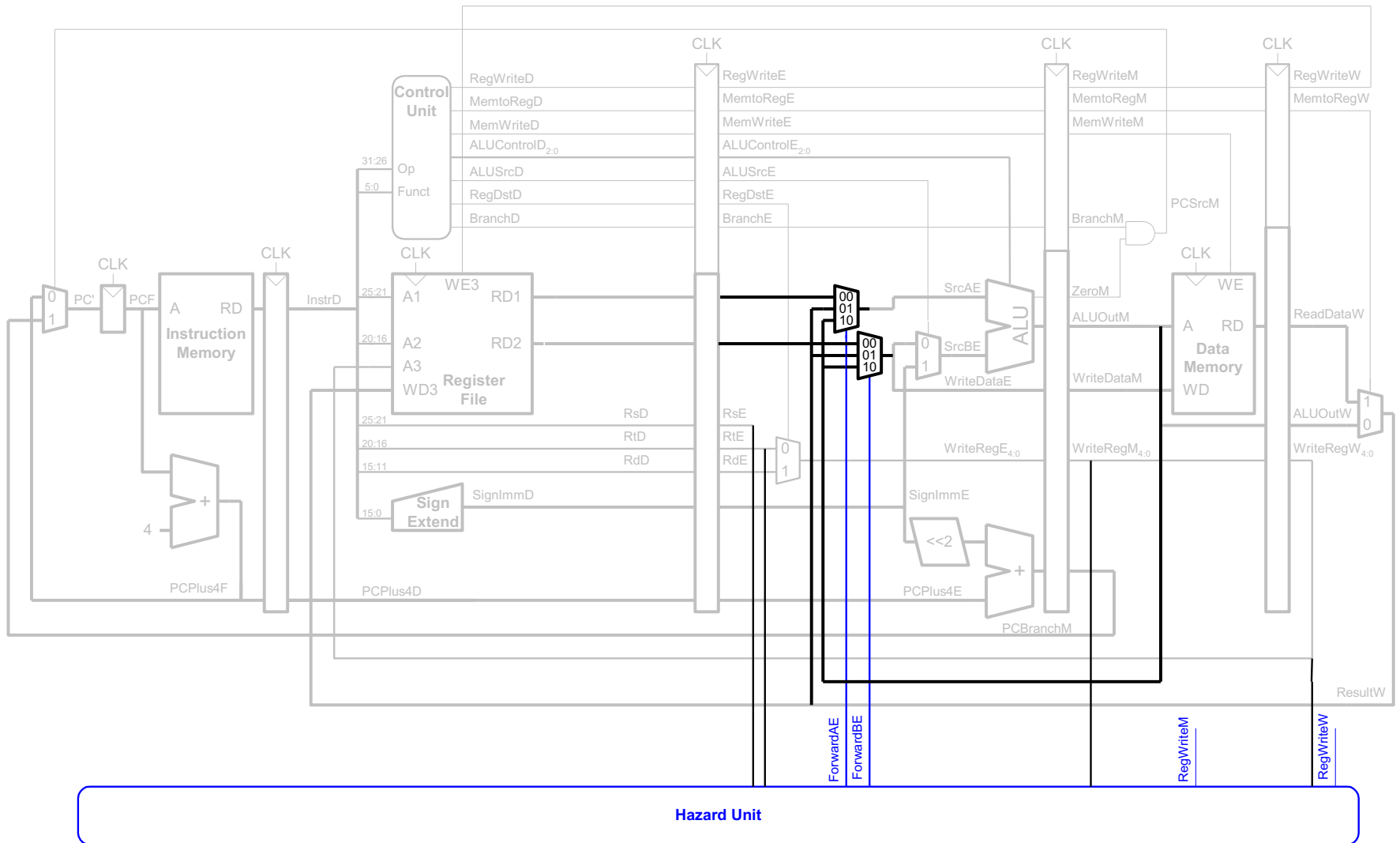
- Also called Data Bypassing
  - We have already seen the basic idea before
    - Forward the result value to the dependent instruction as soon as the value is available
  - Remember dataflow?
    - Data value supplied to dependent instruction as soon as it is available
    - Instruction executes when all its operands are available
  - Data forwarding brings a pipeline closer to data flow execution principles
-

# Data Forwarding





# Data Forwarding



# Data Forwarding

---

- Forward to Execute stage from either:
    - Memory stage or
    - Writeback stage
  - When should we forward from either Memory or Writeback stage?
    - If that stage will write to a destination register and the destination register matches the source register.
    - If both the Memory and Writeback stages contain matching destination registers, the Memory stage should have priority, because it contains the *more recently executed* instruction.
-

# Data Forwarding (in Pseudocode)

---

- Forward to Execute stage from either:

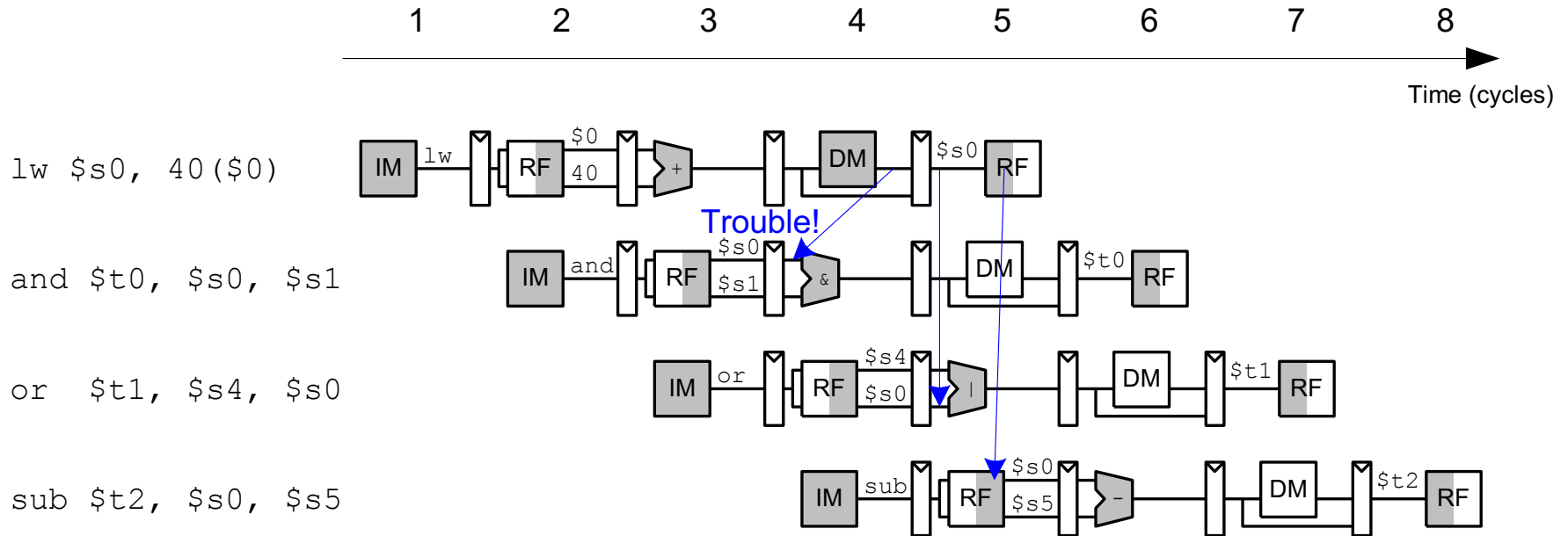
- Memory stage or
- Writeback stage

- Forwarding logic for *ForwardAE* (*pseudo code*):

```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
    ForwardAE = 10 # forward from Memory stage
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
    ForwardAE = 01 # forward from Writeback stage
else
    ForwardAE = 00 # no forwarding
```

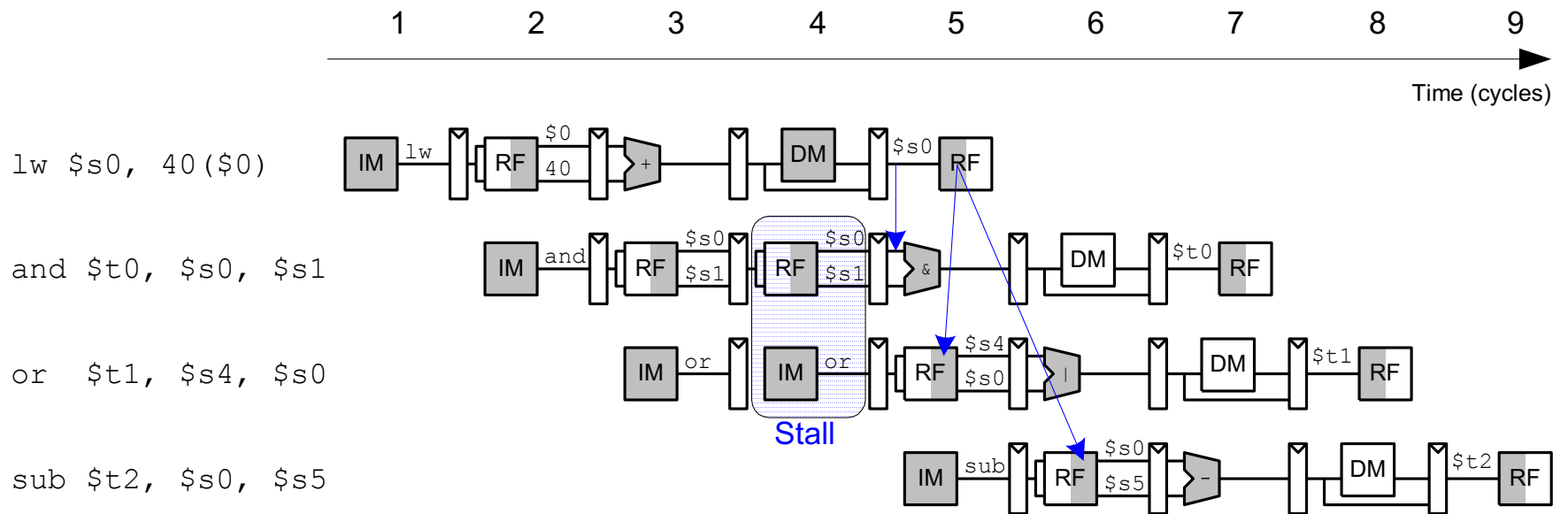
- Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE*
-

# Stalling



- Forwarding is sufficient to resolve RAW data dependences
- *Unfortunately, there are cases when forwarding is **not possible***
  - ❑ *due to pipeline design and instruction latencies*
  - ❑ The lw instruction *does not finish* reading data until the end of the Memory stage
    - its result *cannot be forwarded* to the Execute stage of the next instruction

# Stalling

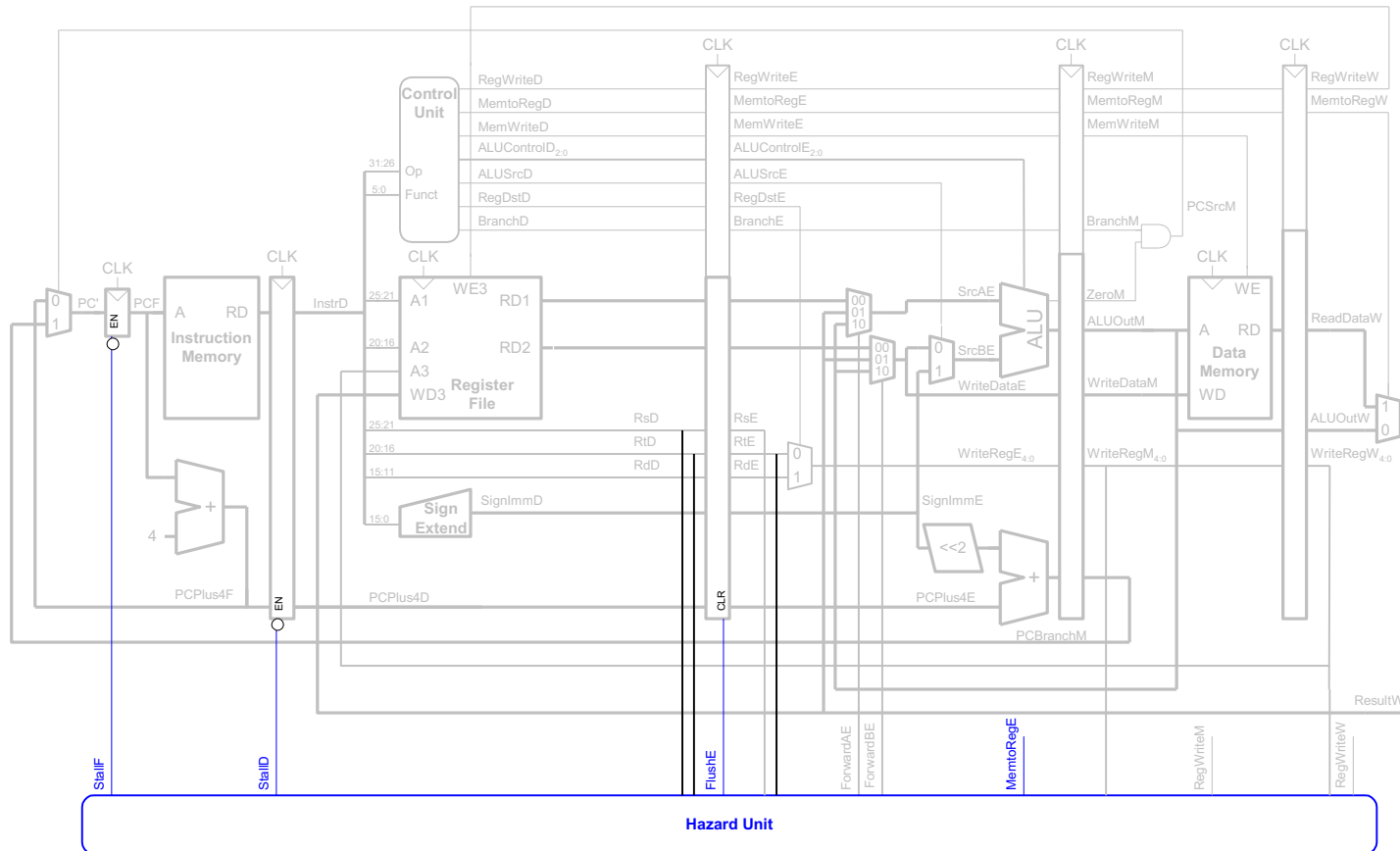


# Hardware Needed for Stalling

---

- Stalls are supported by
    - adding enable inputs (EN) to the Fetch and Decode pipeline registers
    - and a synchronous reset/clear (CLR) input to the Execute pipeline register
      - or an INV bit associated with each pipeline register, indicating that contents are INValid
  - When a lw stall occurs
    - StallD and StallF are asserted to force the Decode and Fetch stage pipeline registers to hold their old values.
    - FlushE is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble
-

# Stalling and Dependence Detection Hardware



# Recall: How to Handle Data Dependences

---

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

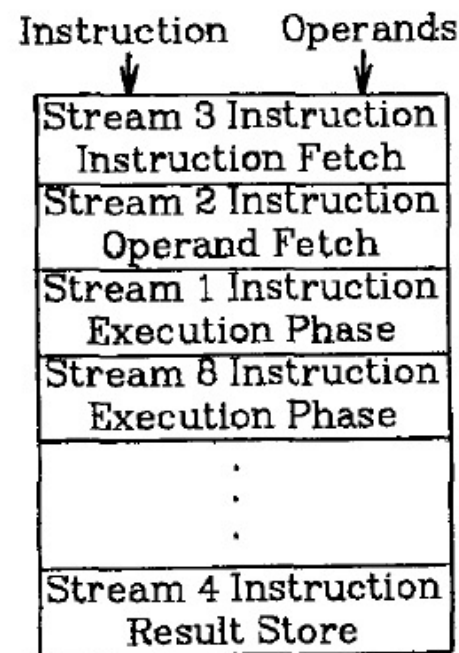


# Fine-Grained Multithreading

# Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline



# Fine-Grained Multithreading (II)

---

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

# Fine-Grained Multithreading: History

---

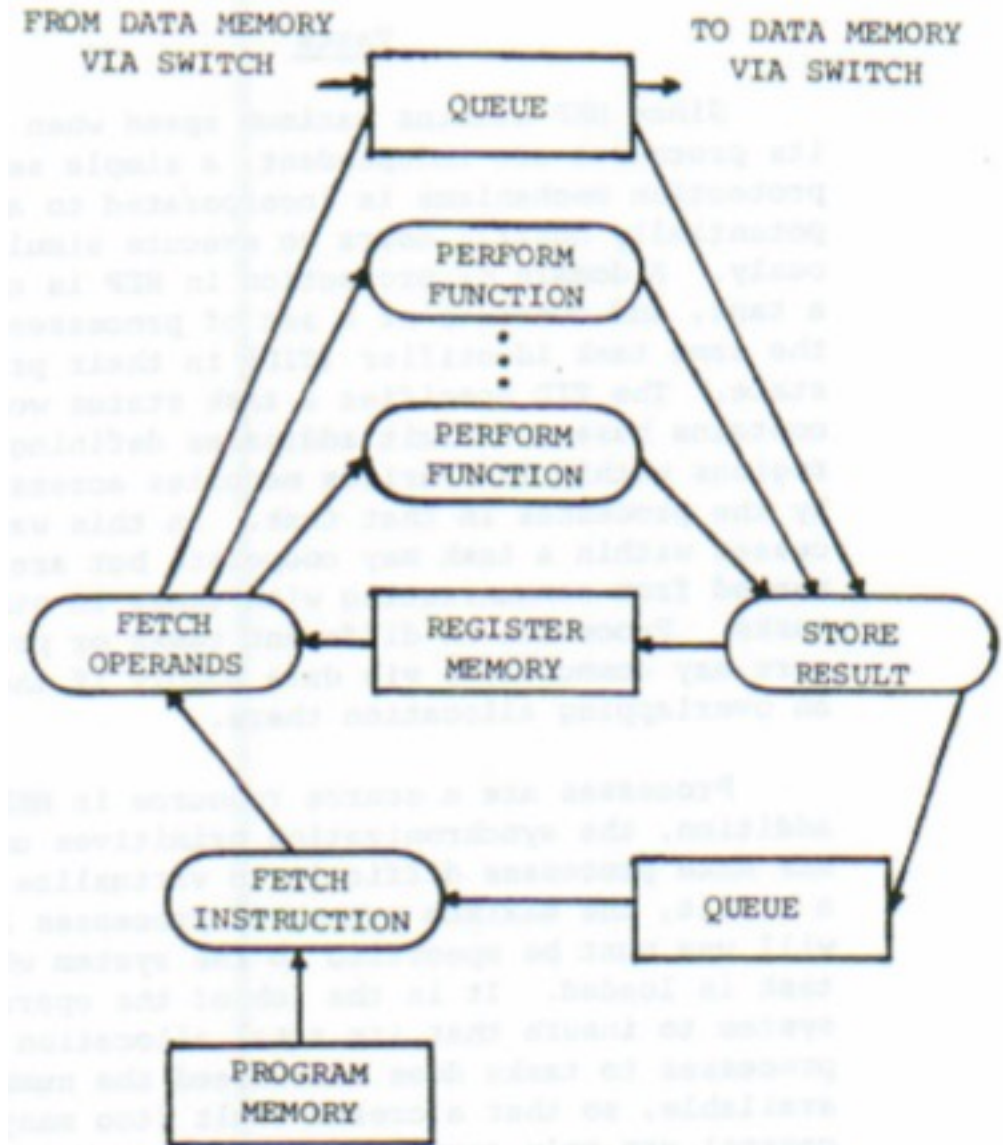
- CDC 6600's peripheral processing unit is fine-grained multithreaded
  - Thornton, “[Parallel Operation in the Control Data 6600](#),” AFIPS 1964.
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles
  
- Denelcor HEP (Heterogeneous Element Processor)
  - Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can have only 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance tradeoff

# Fine-Grained Multithreading in HEP

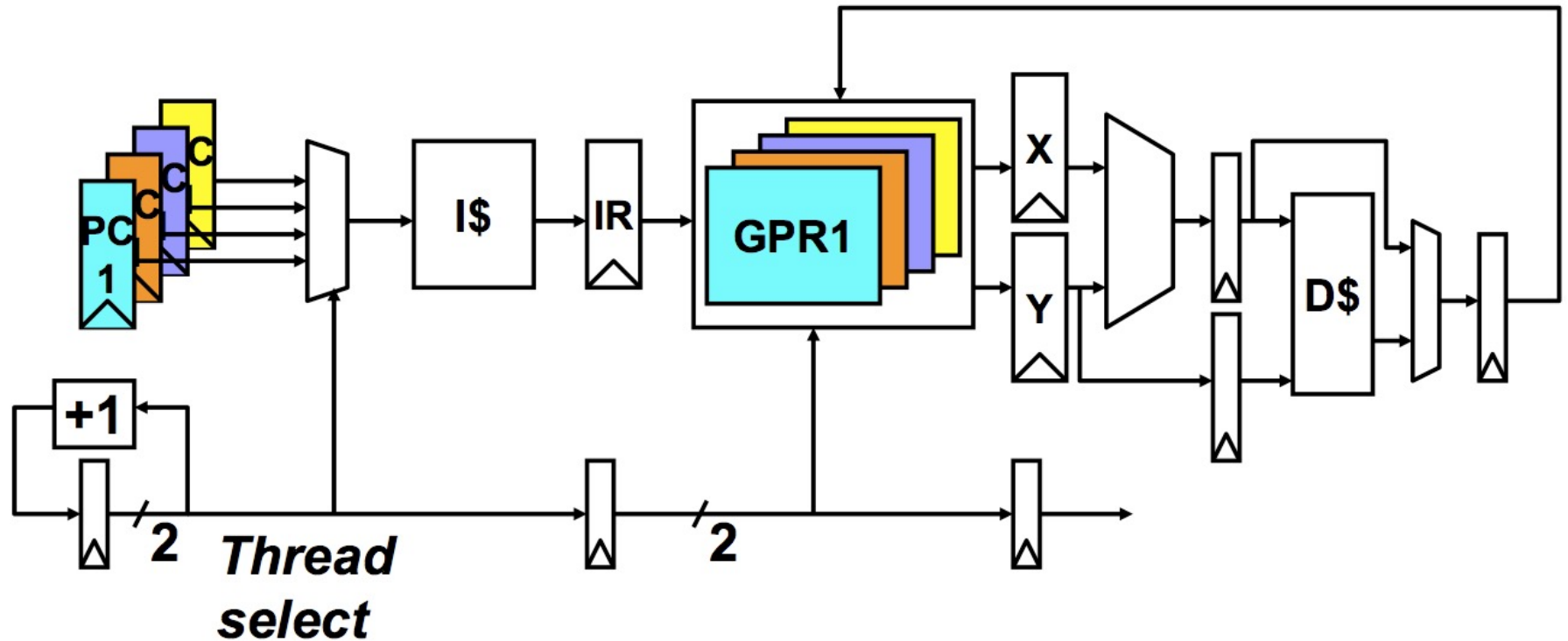
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - ❑ assuming no memory access
- No control and data dependency checking



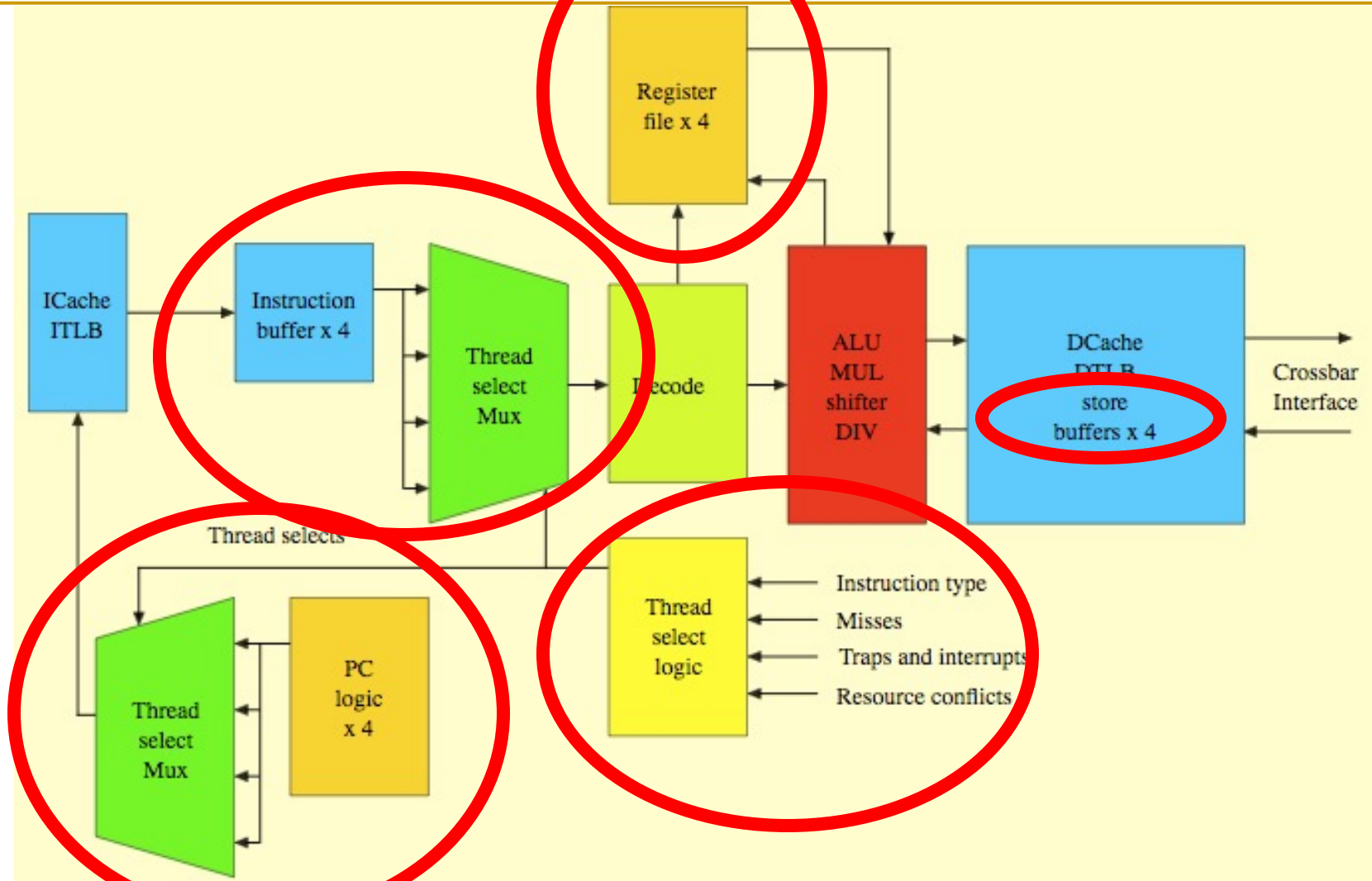
Burton Smith  
(1941-2018)



# Multithreaded Pipeline Example



# Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

# Fine-Grained Multithreading

---

## ■ Advantages

- + No need for dependency checking between instructions  
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, utilization

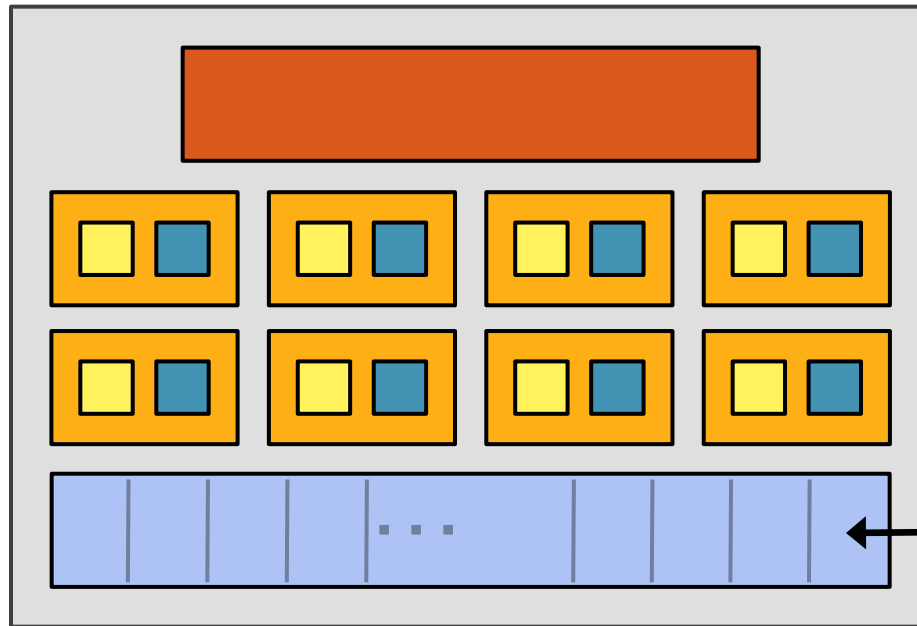
## ■ Disadvantages

- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles from the same thread)
- Resource contention between threads in caches and memory
- Some dependency checking logic *between* threads remains (load/store)

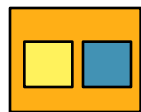


# Modern GPUs are FGMT Machines

# NVIDIA GeForce GTX 285 “core”



64 KB of storage  
for thread contexts  
(registers)



= data-parallel (SIMD) func. unit,  
control shared across 8 units



= multiply-add



= multiply



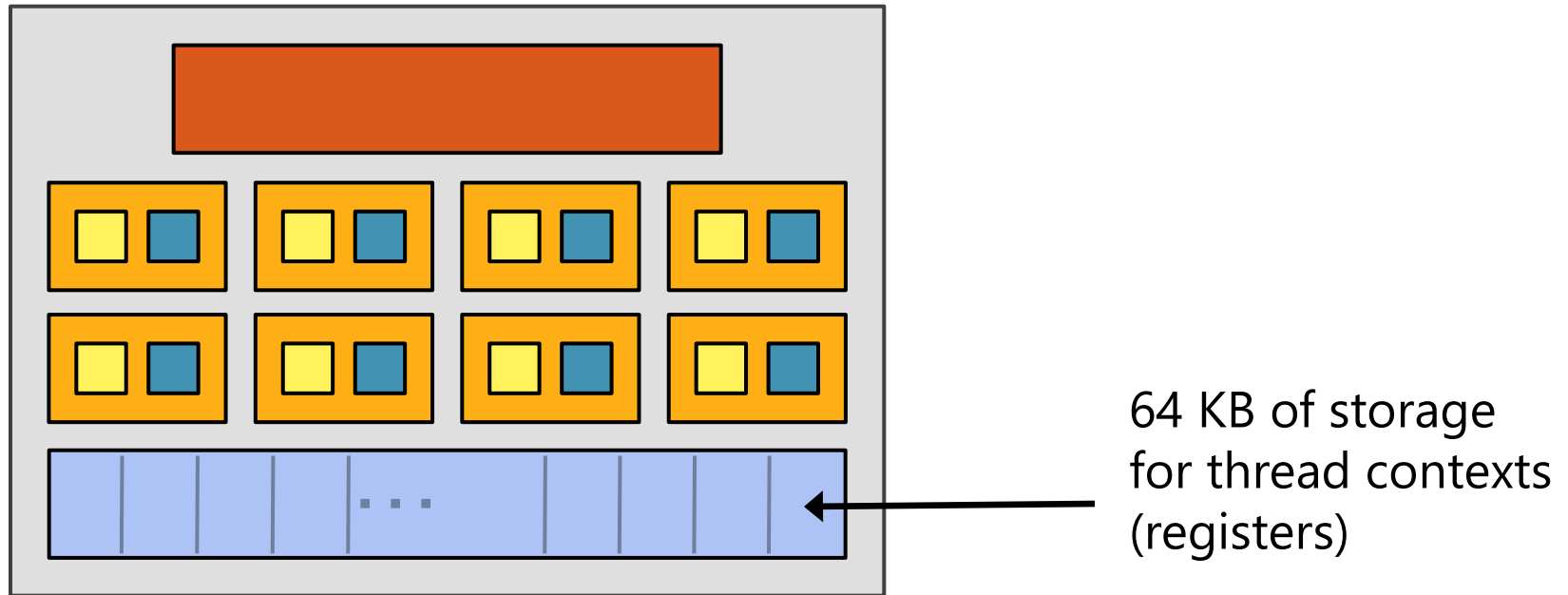
= instruction stream decode



= execution context storage

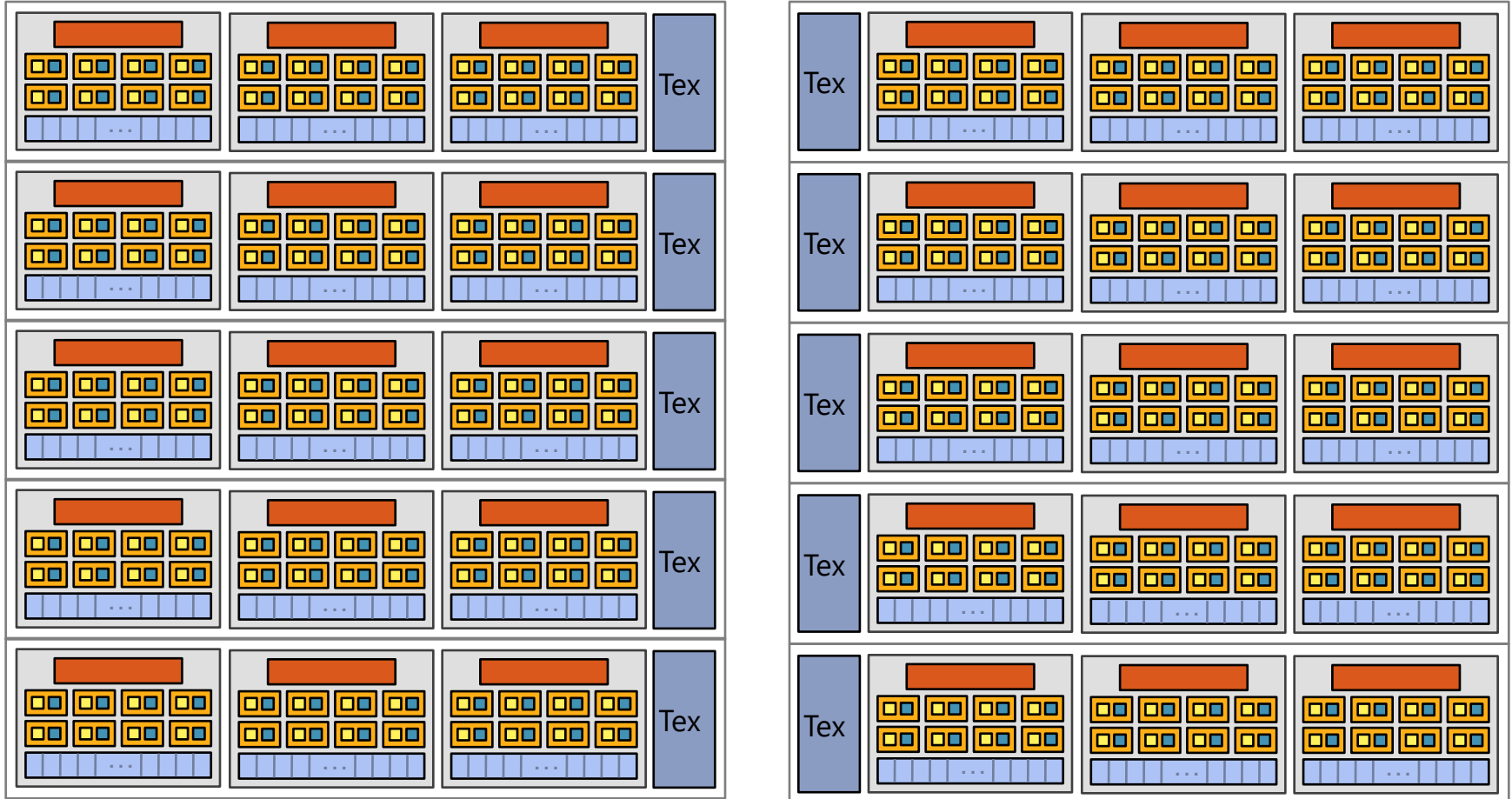
# NVIDIA GeForce GTX 285 “core”

---



- Groups of 32 **threads** share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

# NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

# Further Reading for the Interested (I)

---

A PIPELINED, SHARED RESOURCE MIMD COMPUTER

Burton J. Smith  
Denelcor, Inc.  
Denver, Colorado 80205



Burton Smith  
(1941-2018)

**Architecture and applications of the HEP multiprocessor computer system**

**Burton J. Smith**  
Denelcor, Inc., 14221 E. 4th Avenue, Aurora, Colorado 80011

# Further Reading for the Interested (II)

---

## The Tera Computer System\*

Robert Alverson

David Callahan  
Allan Porterfield

Daniel Cummings  
Burton Smith

Brian Koblenz

Tera Computer Company  
Seattle, Washington USA

### 4 Processors

Each processor in a Tera computer can execute multiple instruction streams simultaneously. In the current implementation, as few as one or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different stream may be issued in each tick without interfering with its predecessors. When an instruction finishes, the stream to which it belongs thereby becomes ready to execute the next instruction. As long as there are enough instruction streams in the processor so that the average instruction latency is filled with instructions from other streams, the processor is being fully utilized. Thus, it is only necessary to have enough streams to hide the expected latency (perhaps 70 ticks on average); once latency is hidden the processor is running at peak performance and additional streams do not speed the result.

# Recall: How to Handle Data Dependences

---

- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Predict the needed value(s), execute “speculatively”, and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

# A Special Case of Data Dependence

---

- Control dependence
  - Data dependence on the Instruction Pointer / Program Counter



# Control Dependence

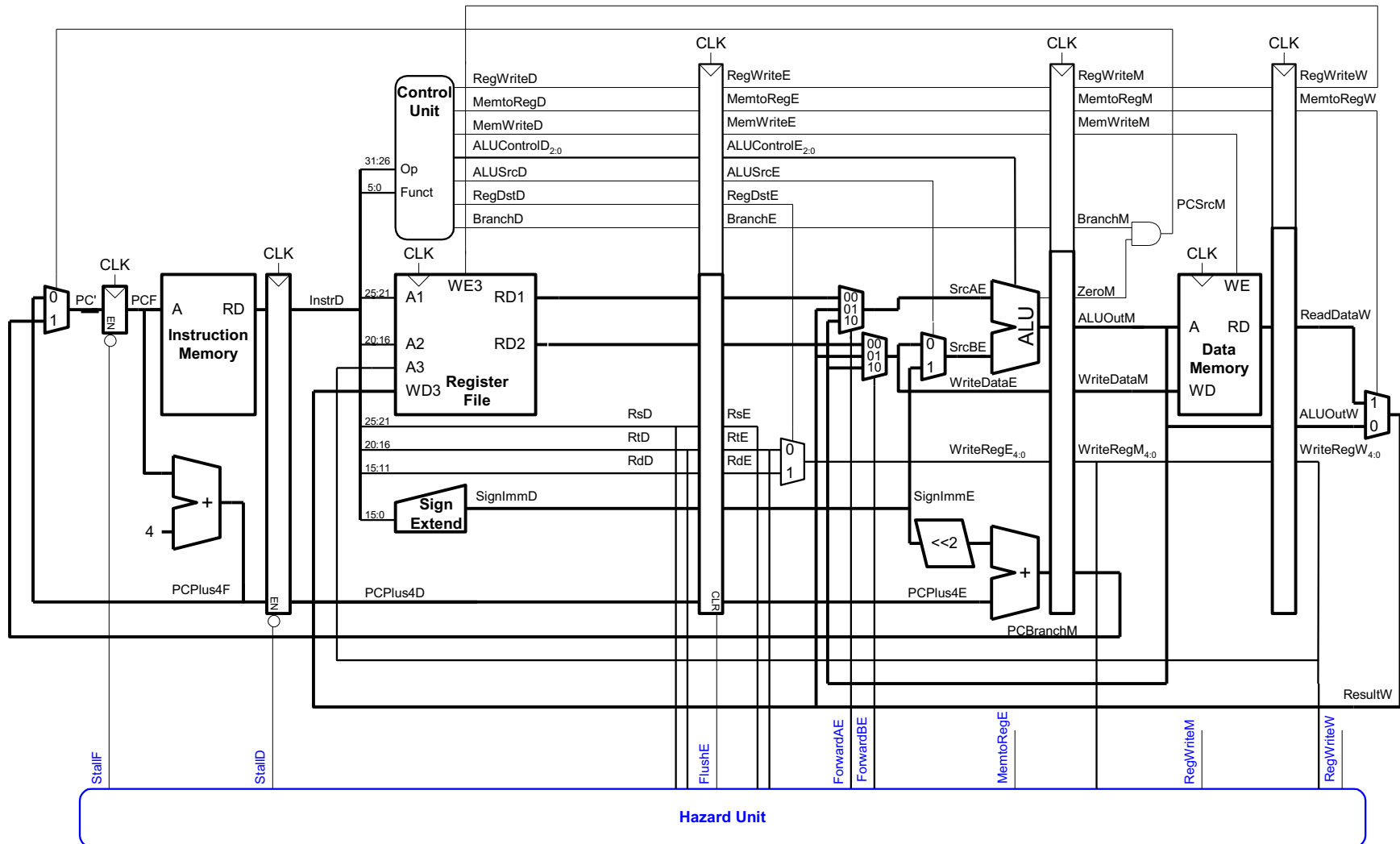
---

- Question: **What should the fetch PC be in the next cycle?**
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?

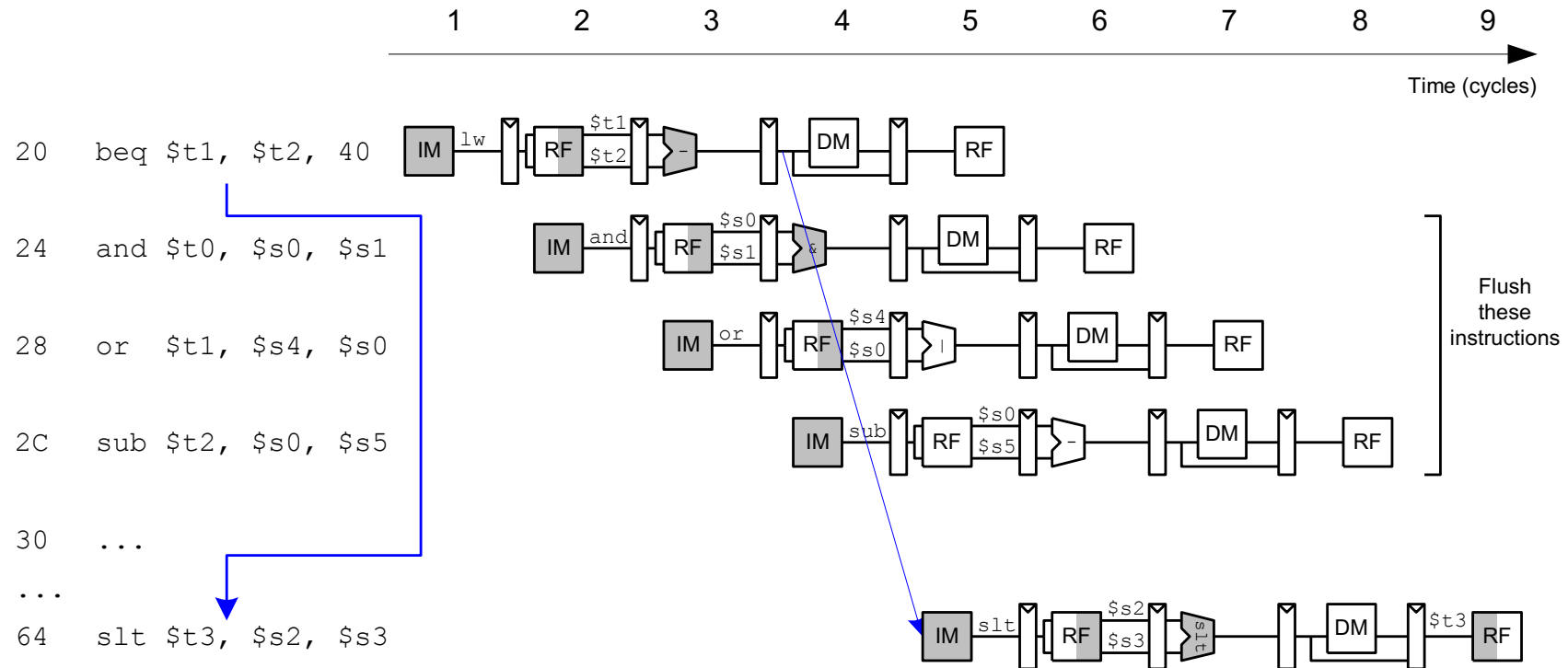
# Control Dependences

- **Special case of data dependence: dependence on PC**
- **beq:**
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch is resolved
    - Always predict that the next sequential instruction is fetched
    - Called “Always not taken” prediction
  - These instructions must be flushed if the branch is taken
- **Branch misprediction penalty**
  - number of instructions flushed when branch is taken
  - May be reduced by determining branch earlier

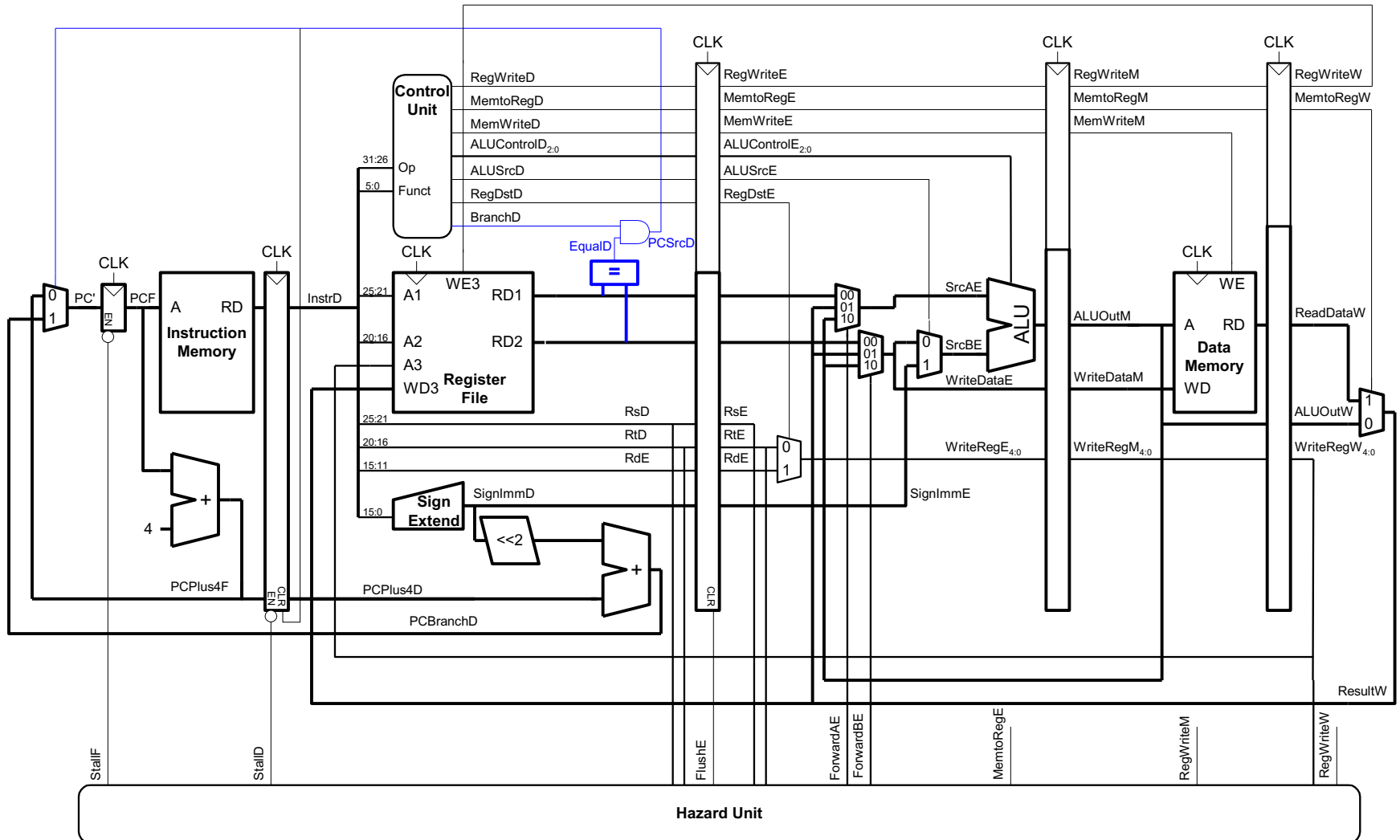
# Control Dependence: Original Pipeline



# Control Dependence

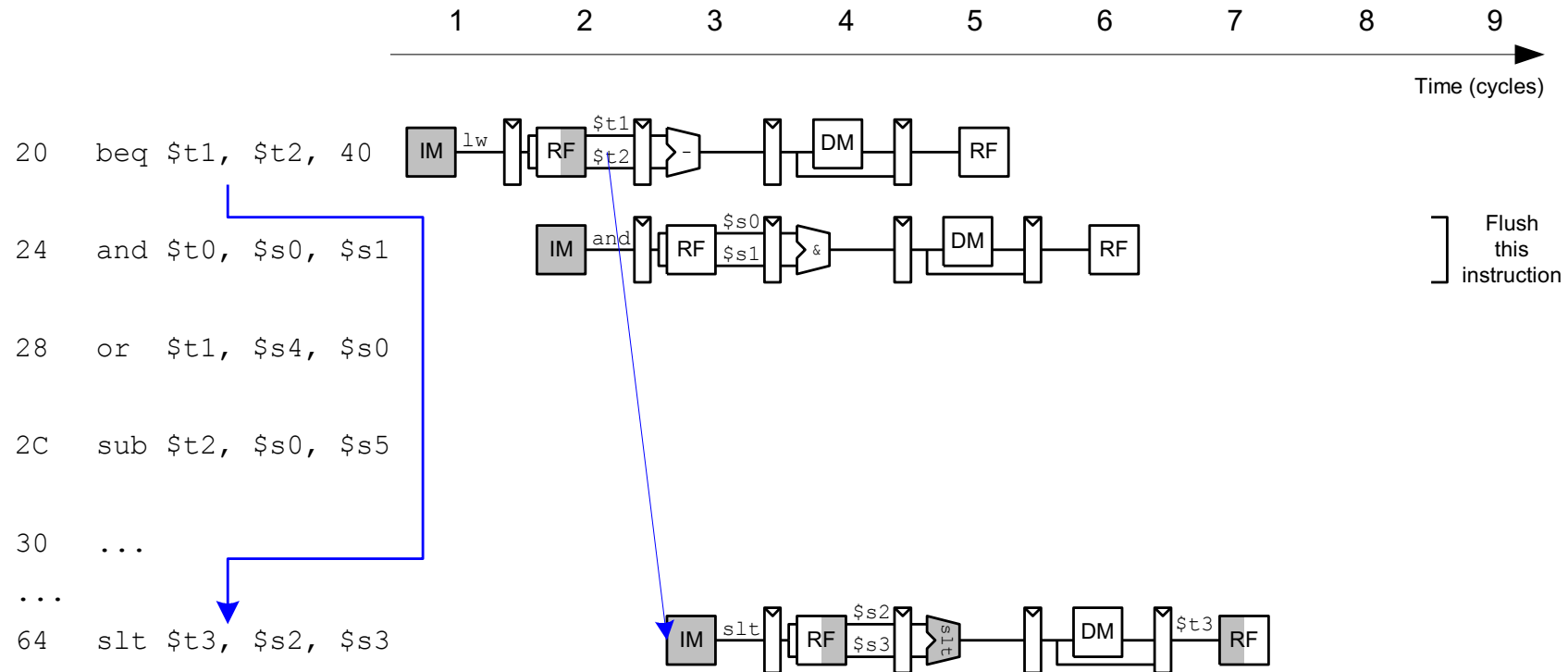


# Early Branch Resolution



Introduces another data dependency in Decode stage..

# Early Branch Resolution



# Early Branch Resolution: Good Idea?

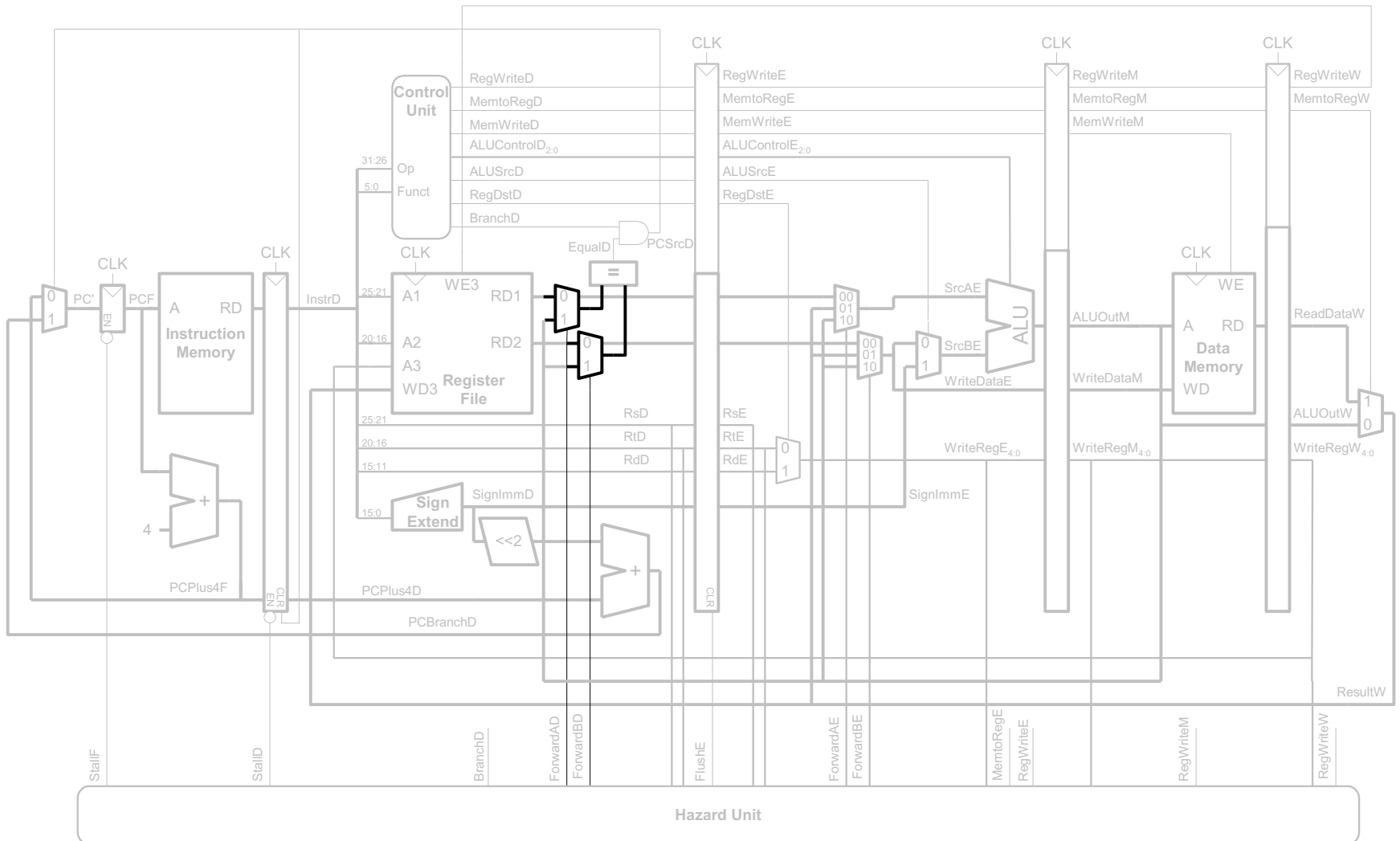
## ■ Advantages

- Reduced branch misprediction penalty
  - Reduced CPI (cycles per instruction)

## ■ Disadvantages

- Potential increase in clock cycle time?
  - Higher  $T_{\text{clock}}$ ?
- Additional hardware cost
  - Specialized and likely not used by other instructions

# Data Forwarding for Early Branch Resolution



Data forwarding for early branch resolution.



# Forwarding and Stalling Hardware Control

```
// Forwarding logic:
assign ForwardAD = (rsD != 0) & (rsD == WriteRegM) & RegWriteM;
assign ForwardBD = (rtD != 0) & (rtD == WriteRegM) & RegWriteM;

//Stalling logic:
assign lwstall = ((rsD == rtE) | (rtD == rtE)) & MemtoRegE;

assign branchstall = (BranchD & RegWriteE &
                     (WriteRegE == rsD | WriteRegE == rtD))
                     |
                     (BranchD & MemtoRegM &
                     (WriteRegM == rsD | WriteRegM == rtD));

// Stall signals;
assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FlushE = lwstall | branchstall;
```

# Doing Better: Smarter Branch Prediction

- **Guess whether branch will be taken**
  - Backward branches are usually taken (loops)
  - Consider history of whether branch was previously taken to improve the guess
- **Good prediction reduces the fraction of branches requiring a flush**

# Questions to Ponder

---

- What is the role of the hardware vs. the software in data dependence handling?
  - Software based interlocking
  - Hardware based interlocking
  - Who inserts/manages the pipeline bubbles?
  - Who finds the independent instructions to fill “empty” pipeline slots?
  - What are the advantages/disadvantages of each?
    - Think of the performance equation as well

# Questions to Ponder

---

- What is the role of the hardware vs. the software in the order in which instructions are executed in the pipeline?
  - Software based instruction scheduling → static scheduling
  - Hardware based instruction scheduling → dynamic scheduling
- How does each impact different metrics?
  - Performance (and parts of the performance equation)
  - Complexity
  - Power consumption
  - Reliability
  - ...

# More on Software vs. Hardware

---

- Software based scheduling of instructions → static scheduling
  - ❑ Compiler orders the instructions, hardware executes them in that order
  - ❑ Contrast this with **dynamic scheduling** (in which hardware can execute instructions out of the compiler-specified order)
  - ❑ How does the compiler know the latency of each instruction?
- What information does the compiler not know that makes static scheduling difficult?
  - ❑ Answer: Anything that is determined at run time
    - Variable-length operation latency, memory addr, branch direction
- How can the compiler alleviate this (i.e., estimate the unknown)?
  - ❑ Answer: Profiling

# More on Static Instruction Scheduling

Trace Scheduling Idea

(a) (b) (c) (d)

TRACE SCHEDULING LOOP-FREE CODE

43

Lecture 16. Static Instruction Scheduling - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

7,136 views • Feb 26, 2015

46 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture  
23K subscribers

SUBSCRIBED



Lecture 16: Static Instruction Scheduling  
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)  
Date: Feb 23rd, 2015

Lecture 16 slides (pdf): <http://www.ece.cmu.edu/~ece447/s15/li...>

<https://www.youtube.com/onurmutlulectures>

# Lectures on Static Instruction Scheduling

---

- Computer Architecture, Spring 2015, Lecture 16

- Static Instruction Scheduling (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=isBEVkJjgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18>

- Computer Architecture, Spring 2013, Lecture 21

- Static Instruction Scheduling (CMU, Spring 2013)
- <https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21>

# Pipelined Performance Example

- **SPECINT2006 benchmark:**
  - 25% loads
  - 10% stores
  - 11% branches
  - 2% jumps
  - 52% R-type
- **Suppose:**
  - 40% of loads used by next instruction
  - 25% of branches mispredicted
- **All jumps flush next instruction**
- **What is the average CPI?**



# Pipelined Performance Example Solution

- Load/Branch CPI = 1 when no stall/flush, 2 when stall/flush.

Thus:

- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$

*Average CPI for load*

- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

*Average CPI for branch*

- And

- *Average CPI* =

# Pipelined Performance Example Solution

- Load/Branch CPI = 1 when no stall/flush, 2 when stall/flush.

Thus:

- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$

*Average CPI for load*

- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$

*Average CPI for branch*

- And

- *Average CPI* =  $(0.25)(1.4) +$   
 $(0.1)(1) +$   
 $(0.11)(1.25) +$   
 $(0.02)(2) +$   
 $(0.52)(1)$

*load*  
*store*  
*beq*  
*jump*  
*r-type*

= **1.15**

# Pipelined Performance

- There are 5 stages, and 5 different timing paths:

$$T_c = \max \{$$

$$t_{pcq} + t_{mem} + t_{setup}$$

$$2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup})$$

$$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}$$

$$t_{pcq} + t_{memwrite} + t_{setup}$$

$$2(t_{pcq} + t_{mux} + t_{RFwrite})$$

}

*fetch*

*decode*

*execute*

*memory*

*writeback*

- The operation speed *depends* on the *slowest operation*
- Decode and Writeback use register file and have only half a clock cycle to complete, that is why there is a 2 in front of them

# Pipelined Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	$t_{pcq\_PC}$	30
Register setup	$t_{setup}$	20
Multiplexer	$t_{mux}$	25
ALU	$t_{ALU}$	200
Memory read	$t_{mem}$	250
Register file read	$t_{RFread}$	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	$t_{eq}$	40
AND gate	$t_{AND}$	15
Memory write	$T_{memwrite}$	220
Register file write	$t_{RFwrite}$	100

$$\begin{aligned}T_c &= 2(t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \\&= 2[150 + 25 + 40 + 15 + 25 + 20] \text{ ps} \\&= 550 \text{ ps}\end{aligned}$$

# Pipelined Performance Example

- For a program with 100 billion instructions executing on a pipelined MIPS processor:
  - $CPI = 1.15$
  - $T_c = 550 \text{ ps}$
- Execution Time       $= (\# \text{ instructions}) \times CPI \times T_c$   
                              $= (100 \times 10^9)(1.15)(550 \times 10^{-12})$   
                              $= 63 \text{ seconds}$

# Performance Summary for MIPS arch.

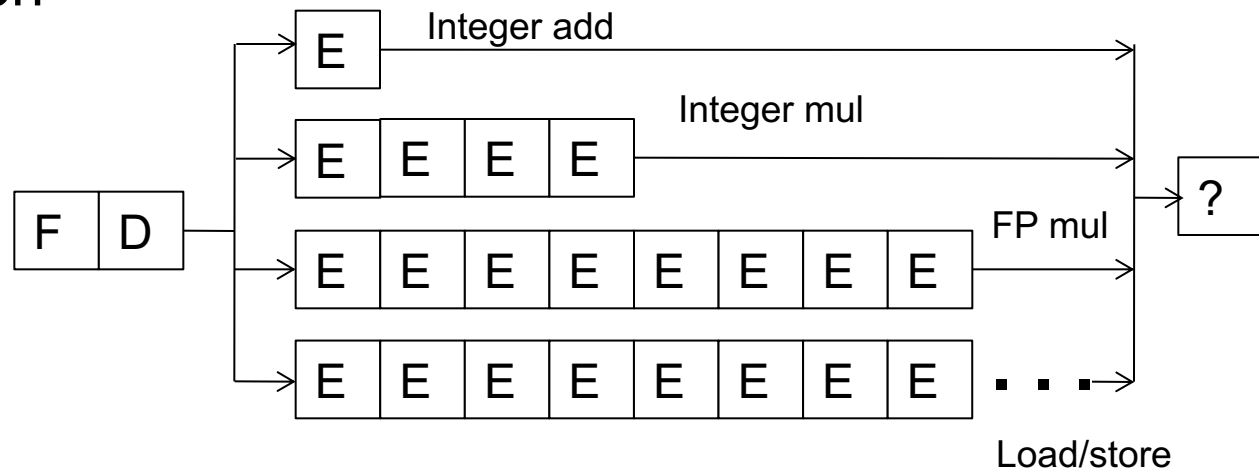
Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Multicycle	133	0.71
Pipelined	63	1.51

- Fastest of the three MIPS architectures is *Pipelined*.
- However, even though we have 5 fold pipelining, it is not 5 times faster than single cycle.

# Pipelining and Precise Exceptions: Preserving Sequential Semantics

# Multi-Cycle Execution

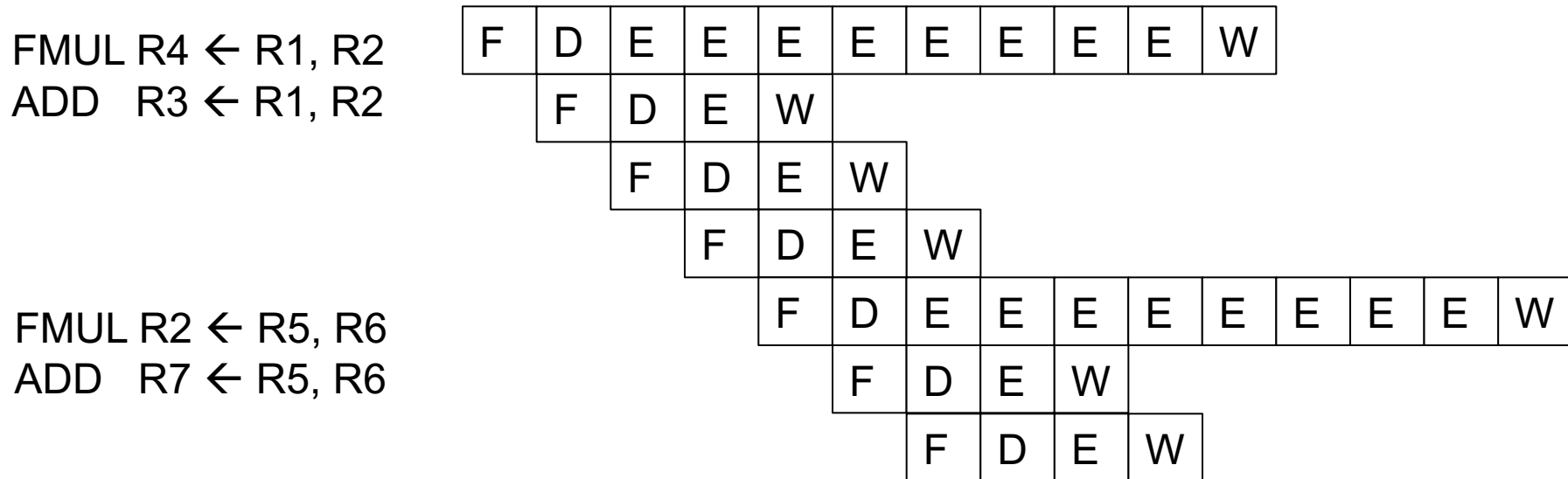
- Not all instructions take the same amount of time for “execution”
- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution





# Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULTiply



- What is wrong with this picture in a Von Neumann architecture?
  - Sequential semantics of the ISA NOT preserved!
  - What if FMUL incurs an exception?

# Exceptions vs. Interrupts

---

## ■ Cause

- ❑ Exceptions: internal to the running thread
- ❑ Interrupts: external to the running thread

## ■ When to Handle

- ❑ Exceptions: when detected (and known to be non-speculative)
- ❑ Interrupts: when convenient
  - Except for very high priority ones
    - ❑ Power failure
    - ❑ Machine check (error)

## ■ Priority: process (exception), depends (interrupt)

## ■ Handling Context: process (exception), system (interrupt)

---

# Precise Exceptions/Interrupts

---

- The architectural state should be consistent (precise) when the exception/interrupt is ready to be handled

1. All previous instructions should be completely retired.

2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state

# Checking for and Handling Exceptions in Pipelining

---

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic
  - Ensures architectural state is precise (register file, PC, memory)
  - Flushes all younger instructions in the pipeline
  - Saves PC and registers (as specified by the ISA)
  - Redirects the fetch engine to the appropriate exception handling routine

# Why Do We Want Precise Exceptions?

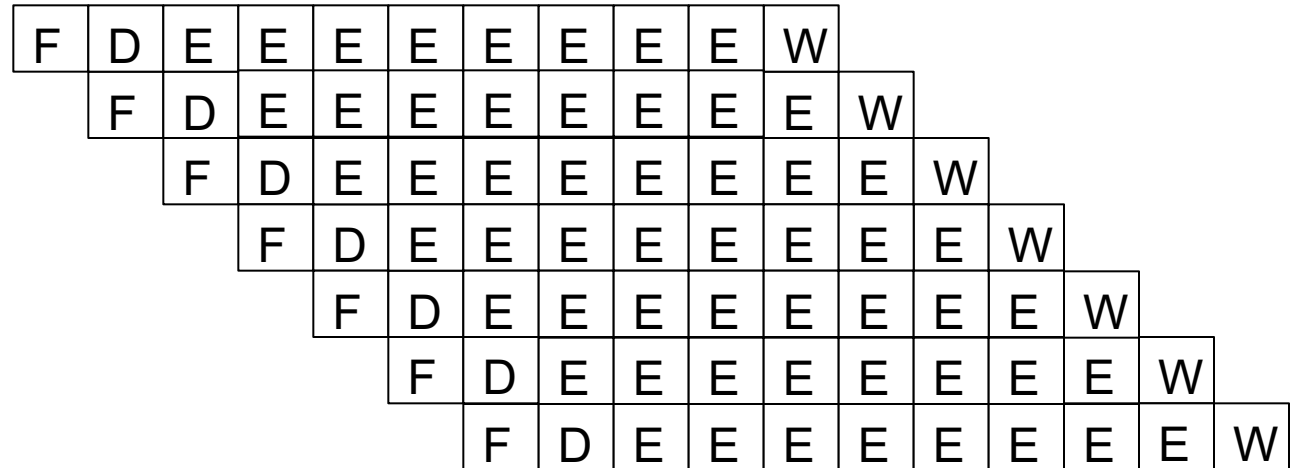
---

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. Dataflow
- Aids software debugging
- Enables (easy) recovery from exceptions
- Enables (easily) restartable processes
- Enables traps into software (e.g., software implemented opcodes)

# Ensuring Precise Exceptions in Pipelining

- Idea: Make each operation take the same amount of time

FMUL R3  $\leftarrow$  R1, R2  
ADD R4  $\leftarrow$  R1, R2



- Downside
  - ❑ Worst-case instruction latency determines all instructions' latency
  - ❑ What about memory operations?
  - ❑ Each functional unit takes worst-case number of cycles?

# Solutions

---

- Reorder buffer

- History buffer
- Future register file
- Checkpointing

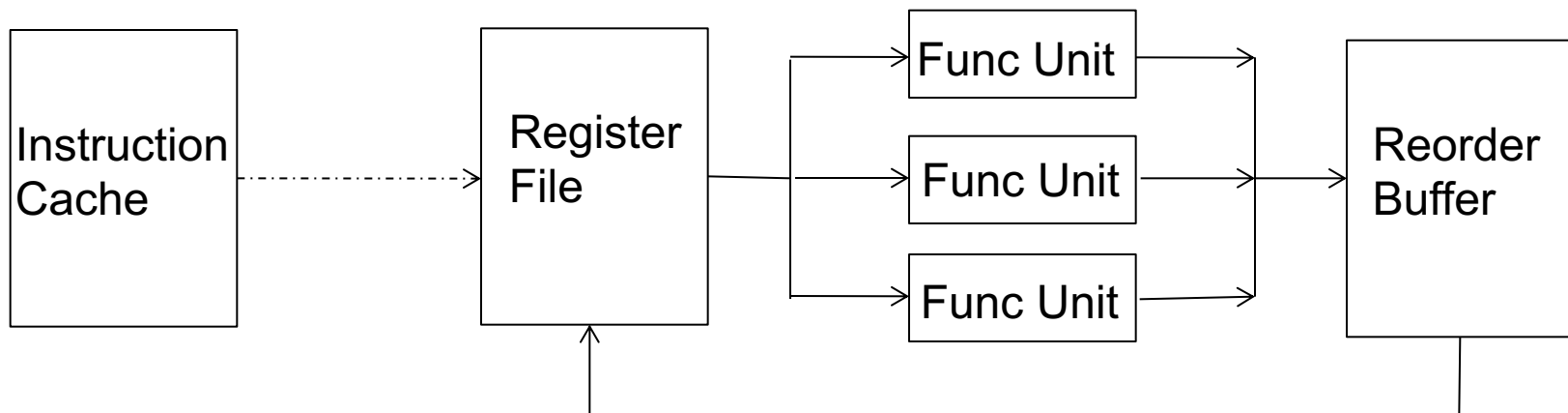
We will not cover these

- Suggested reading
  - Smith and Plezskun, “[Implementing Precise Interrupts in Pipelined Processors](#),” IEEE Trans on Computers 1988 and ISCA 1985.

# Recall: Solution I: Reorder Buffer (ROB)

---

- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves the next-sequential entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed without exceptions, its result moved to reg. file or memory





# Reorder Buffer

---

- Buffers information about all instructions that are decoded but not yet retired/committed

# What's in a ROB Entry?

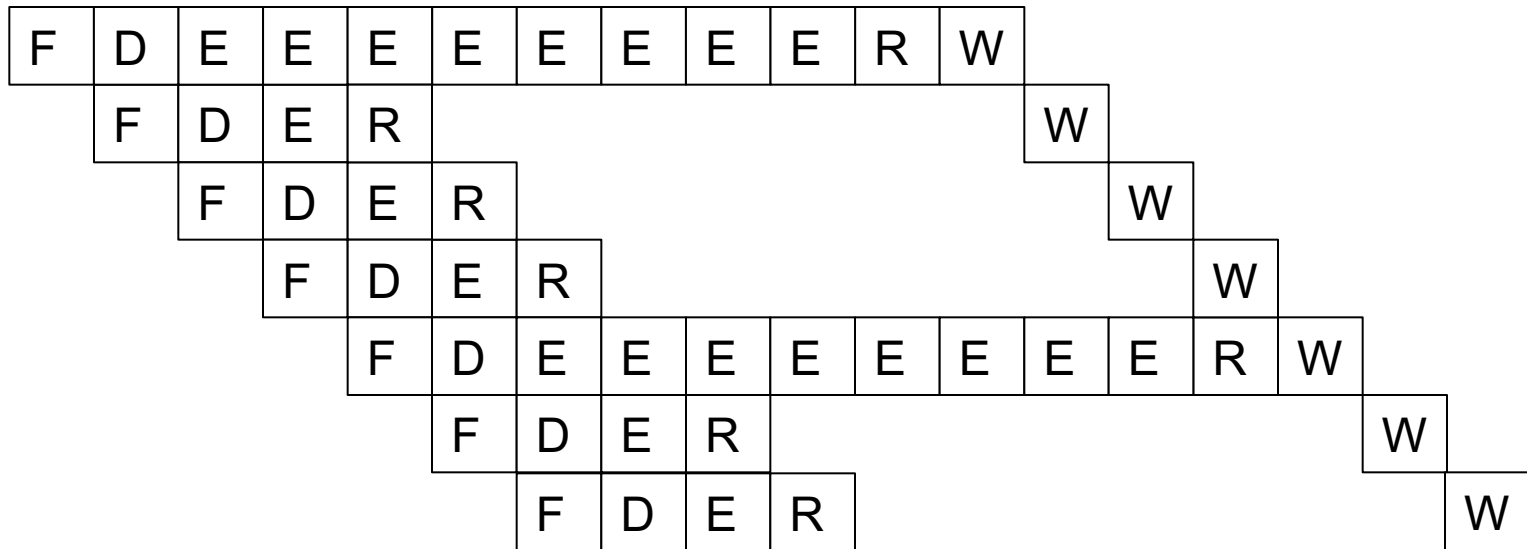
---

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exception?
---	-----------	------------	-----------	-----------	----	---	------------

- Everything required to:
  - ❑ correctly reorder instructions back into the program order
  - ❑ update the architectural state with the instruction's result(s), if instruction can retire without any issues
  - ❑ handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction
- Need valid bits to keep track of readiness of the result(s) and find out if the instruction has completed execution

# Reorder Buffer: Independent Operations

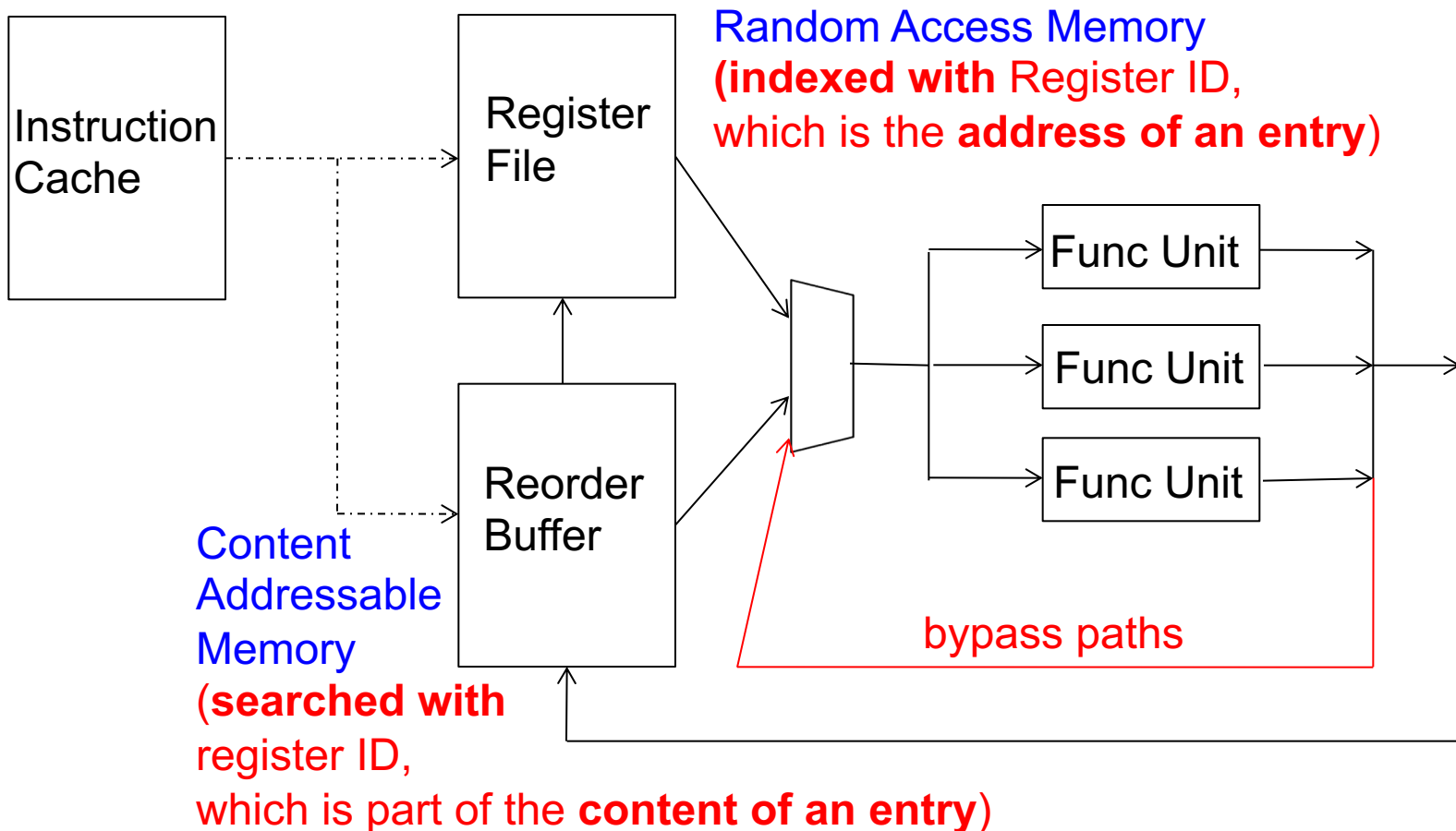
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- What if a later instruction needs a value in the reorder buffer?
  - ❑ One option: stall the operation → stall the pipeline
  - ❑ Better: Read the value from the reorder buffer. **How?**

# Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)

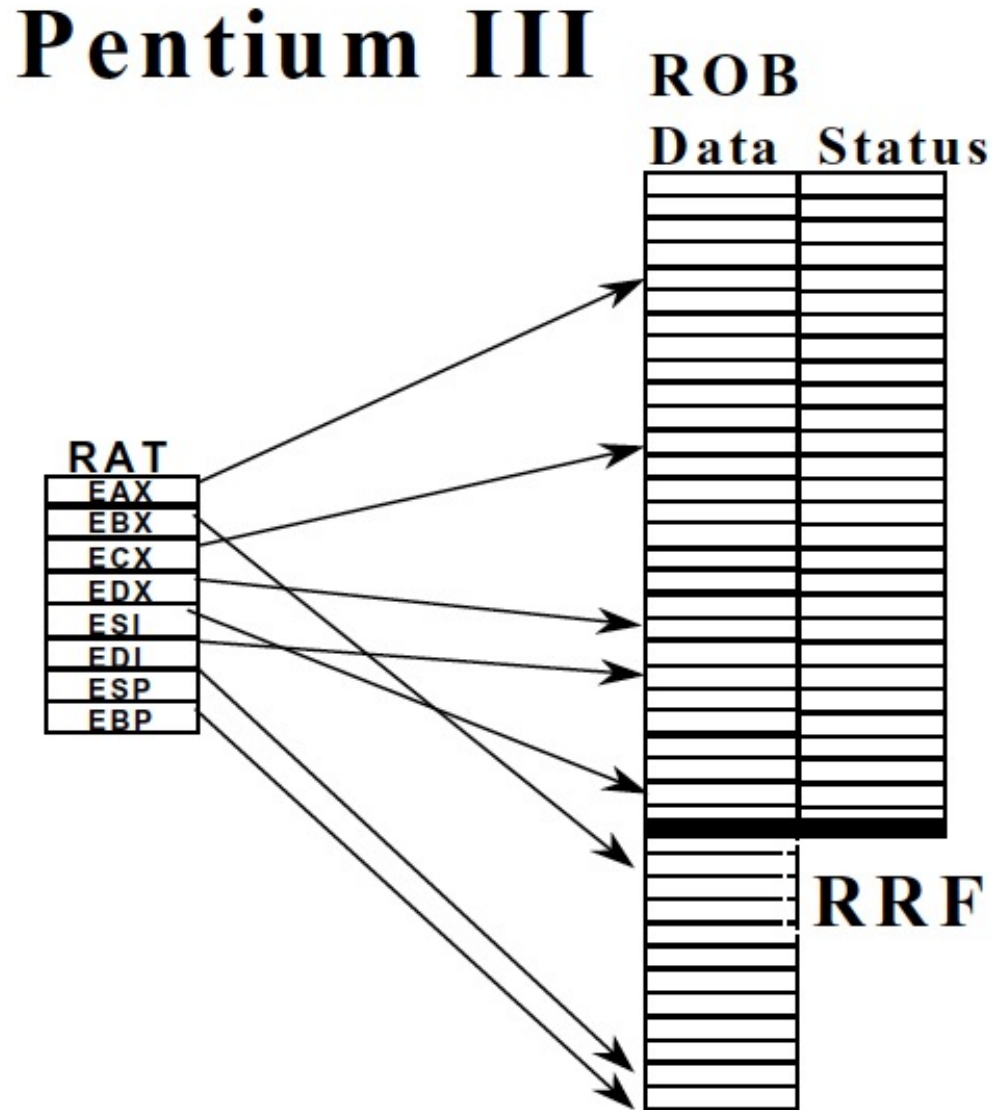


# Simplifying Reorder Buffer Access

---

- Idea: Use indirection
- Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register
- Access reorder buffer next
- Now, reorder buffer does not need to be content addressable

# Reorder Buffer in Intel Pentium III



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

# Important: Register Renaming with a Reorder Buffer

---


- Output and anti dependencies are **not true dependencies**
  - ❑ WHY? The same register refers to values that have nothing to do with each other
  - ❑ **They exist due to lack of register ID's (i.e. names) in the ISA**
- The register ID is **renamed** to the reorder buffer entry that will hold the register's value
  - ❑ Register ID → ROB entry ID
  - ❑ Architectural register ID → Physical register ID
  - ❑ After renaming, ROB entry ID used to refer to the register
- This eliminates anti and output dependencies
  - ❑ Gives the illusion that there are a large number of registers

# Recall: Data Dependence Types

---

True (flow) dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW) -- **True**

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR) -- **Anti**

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$

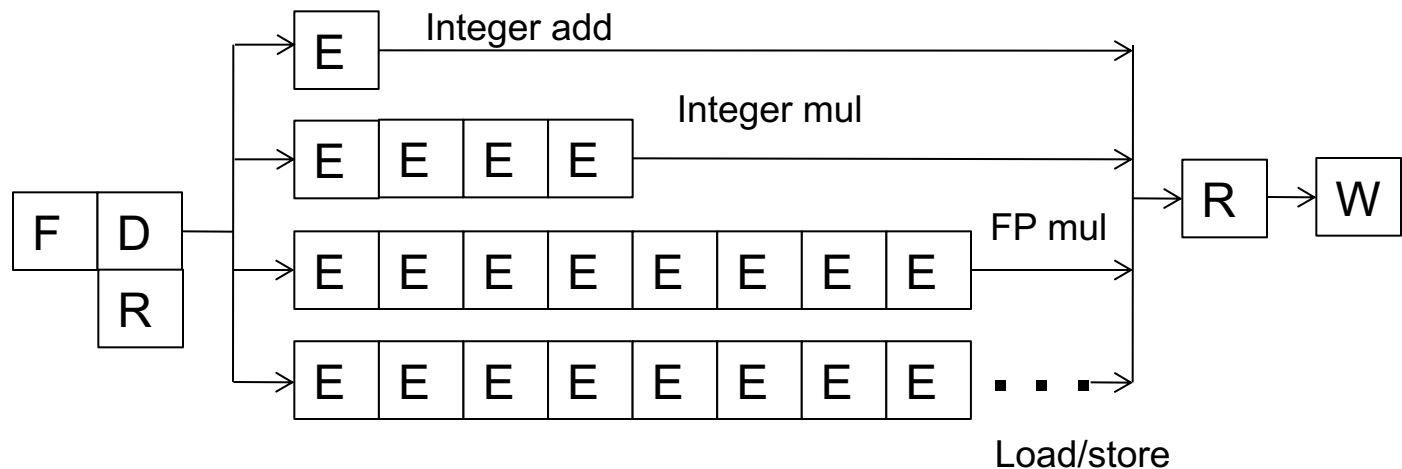


Write-after-Write  
(WAW) -- **Output**



# In-Order Pipeline with Reorder Buffer

- **Decode (D)**: Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- **Execute (E)**: Instructions can complete out-of-order
- **Completion (R)**: Write result to **reorder buffer**
- **Retirement/Commit (W)**: Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**



# Reorder Buffer Tradeoffs

---

## ■ Advantages

- ❑ Conceptually simple for supporting precise exceptions
- ❑ Can eliminate false dependences

## ■ Disadvantages

- ❑ Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
  - CAM or indirection → increased latency and complexity

## ■ Other solutions aim to eliminate the disadvantages

- ❑ History buffer
  - ❑ Future file
  - ❑ Checkpointing
- We will not cover these