

# Digital Design & Computer Arch.

## Lecture 14: Pipelining Issues

Prof. Onur Mutlu

ETH Zürich

Spring 2020

3 April 2020

# 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

---

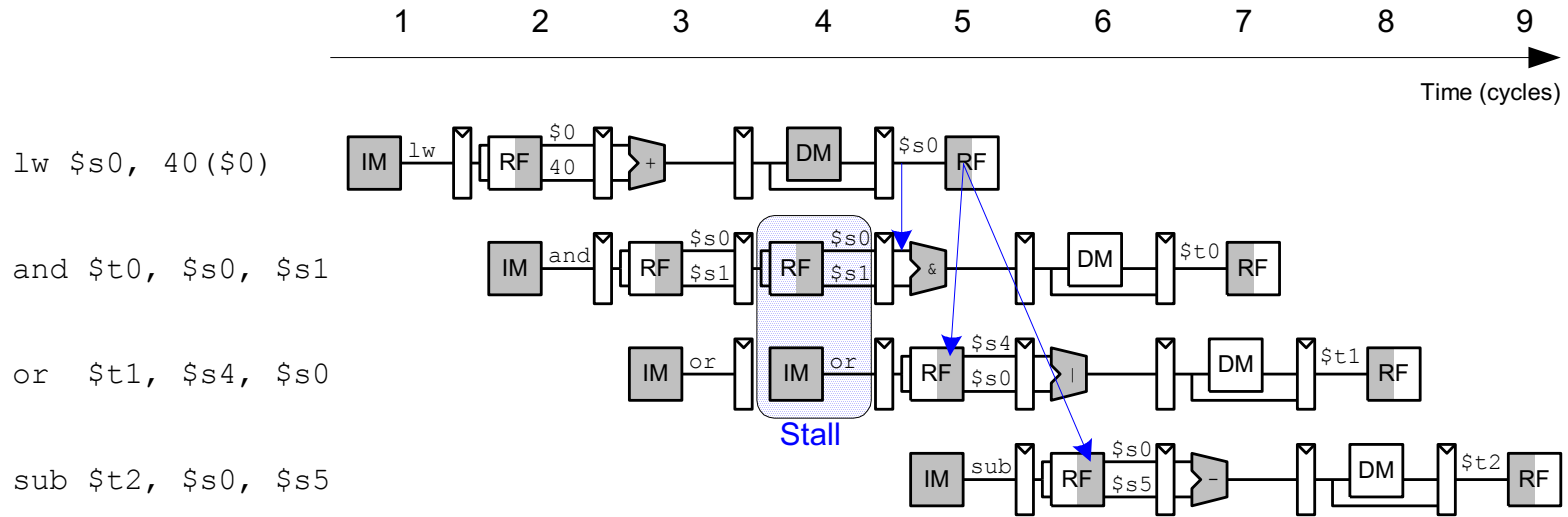
- Last week
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures
- This week
  - Pipelining
  - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Next week
  - Out-of-Order Execution
  - Issues in OoO Execution: Load-Store Handling, ...

# Review: 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
- Six 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
  - **Detect and move it out of the way** for independent instructions
  - **Predict** the needed value(s), execute “speculatively”, **and verify**
  - **Do something else** (fine-grained multithreading)
    - No need to detect

# Stalling



# Stalling Hardware

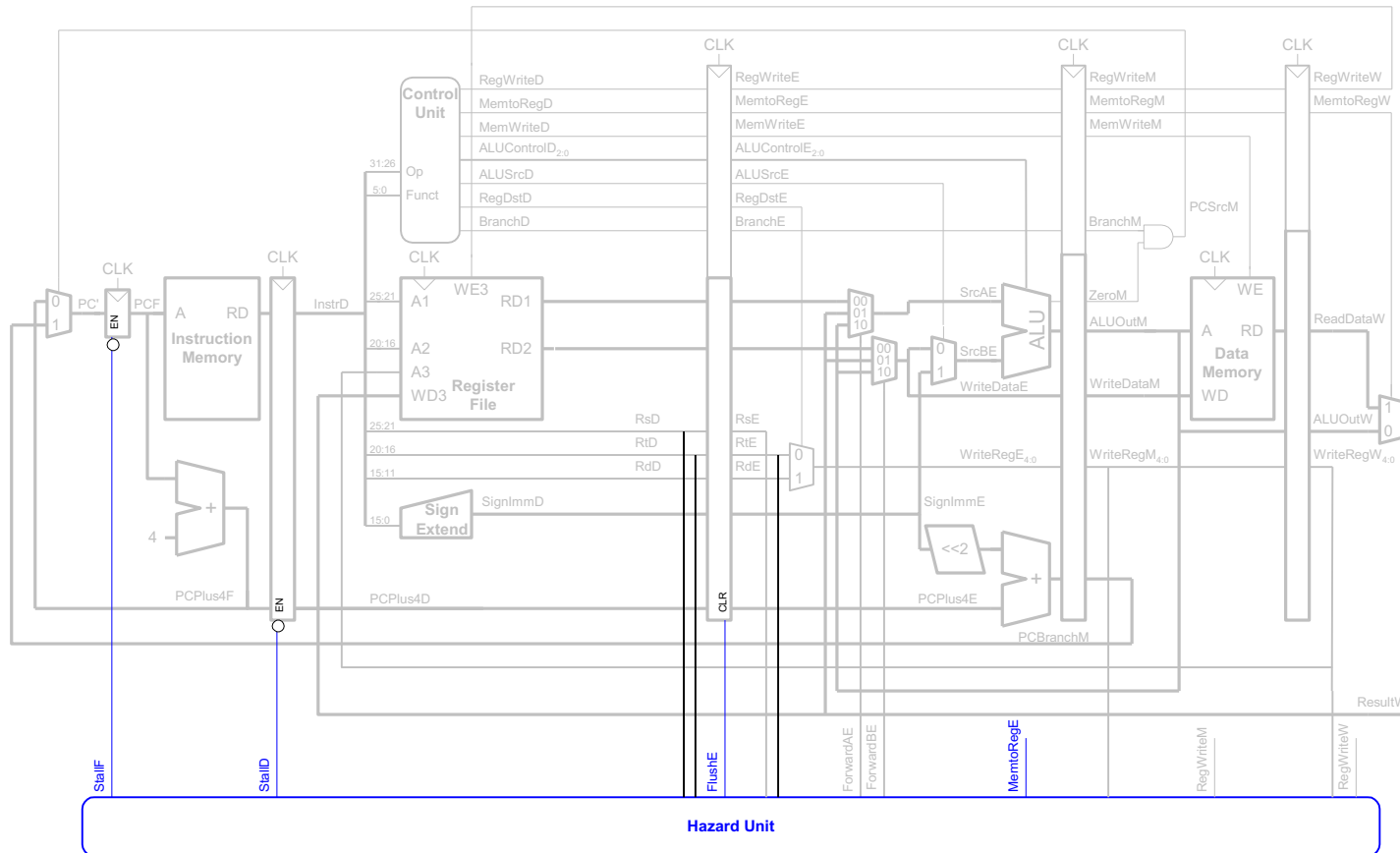
## ■ Stalls are supported by:

- 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

## ■ 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 Hardware

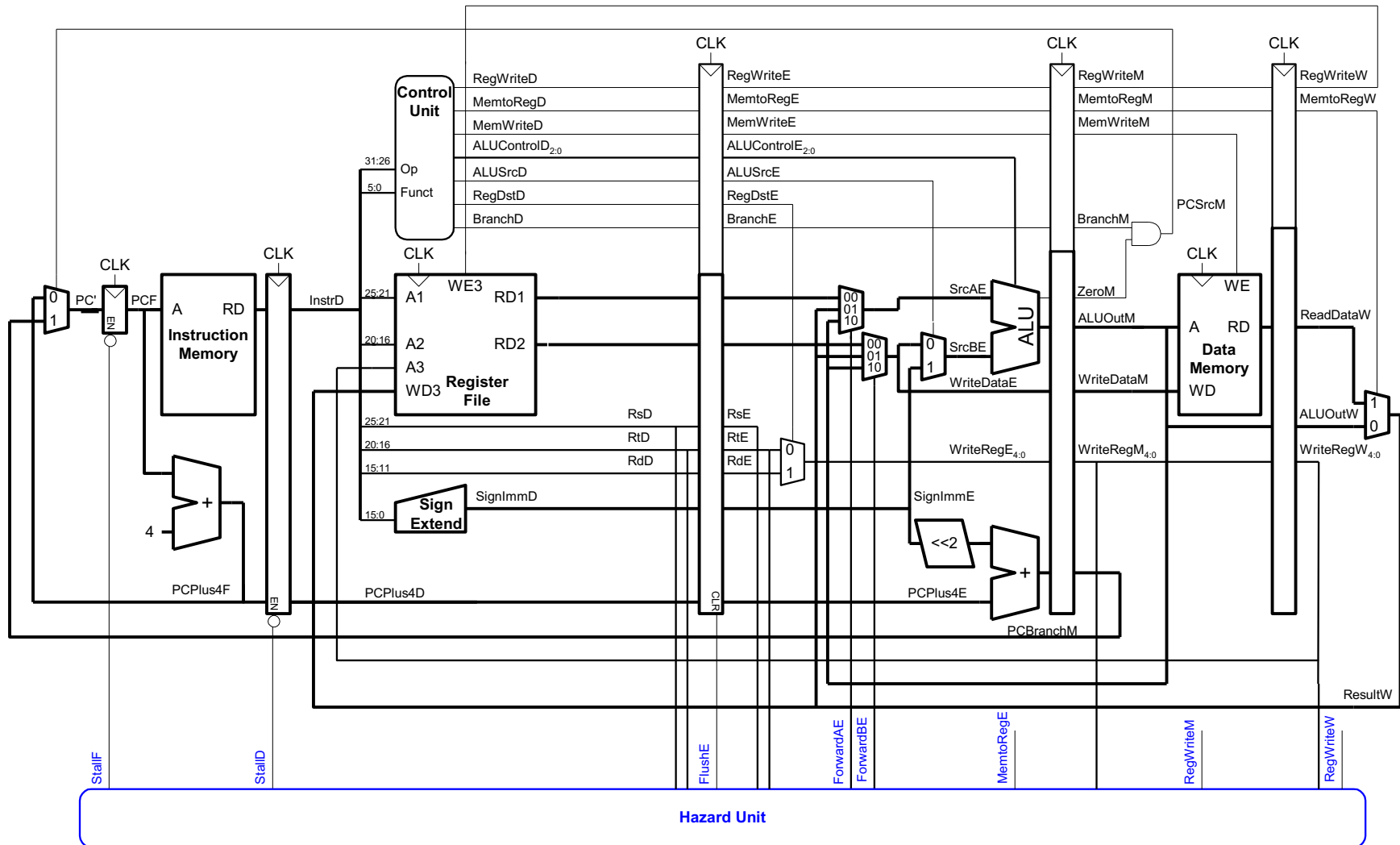


# Control Dependences

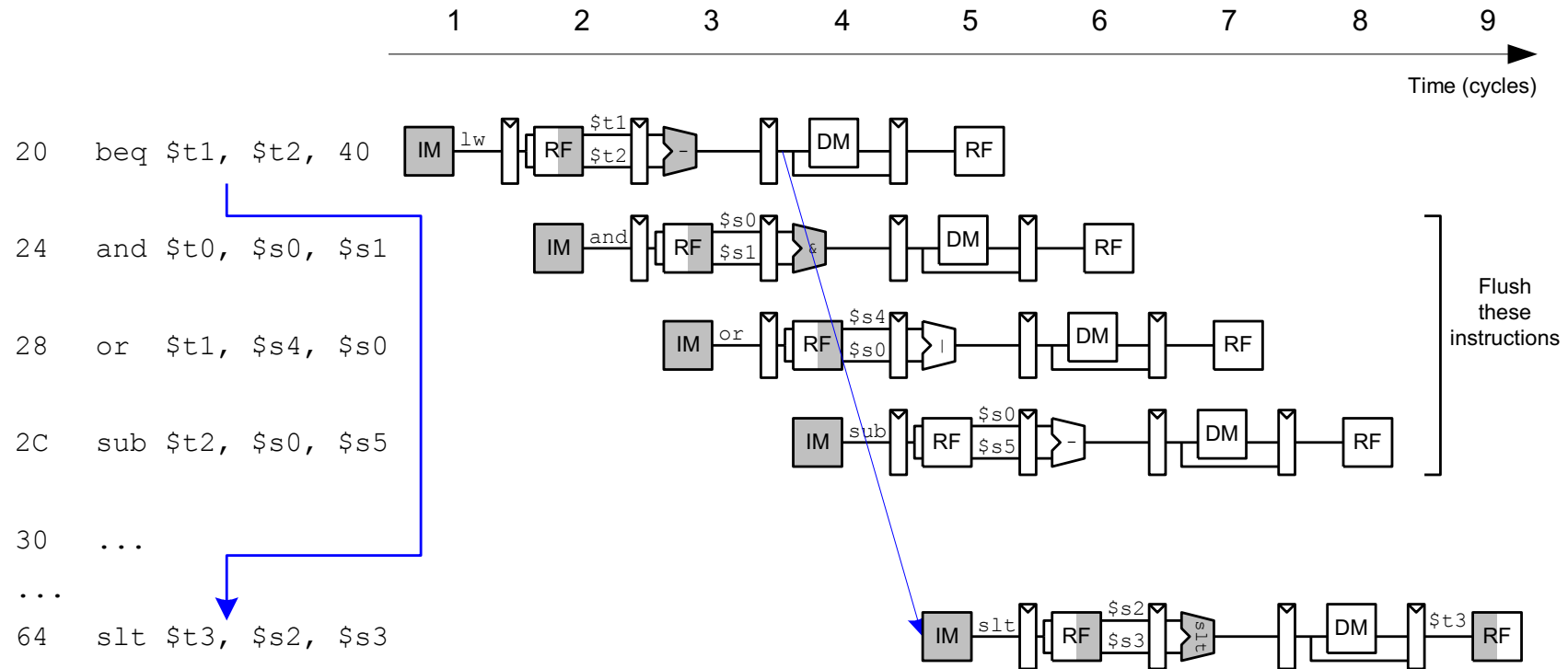
- **Special case of data dependence: dependence on PC**
- **beq:**
  - branch is not determined until the fourth stage of the pipeline
  - Instructions after the branch are fetched before branch is resolved
    - Always predict that the next sequential instruction is fetched
    - Called “Always not 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



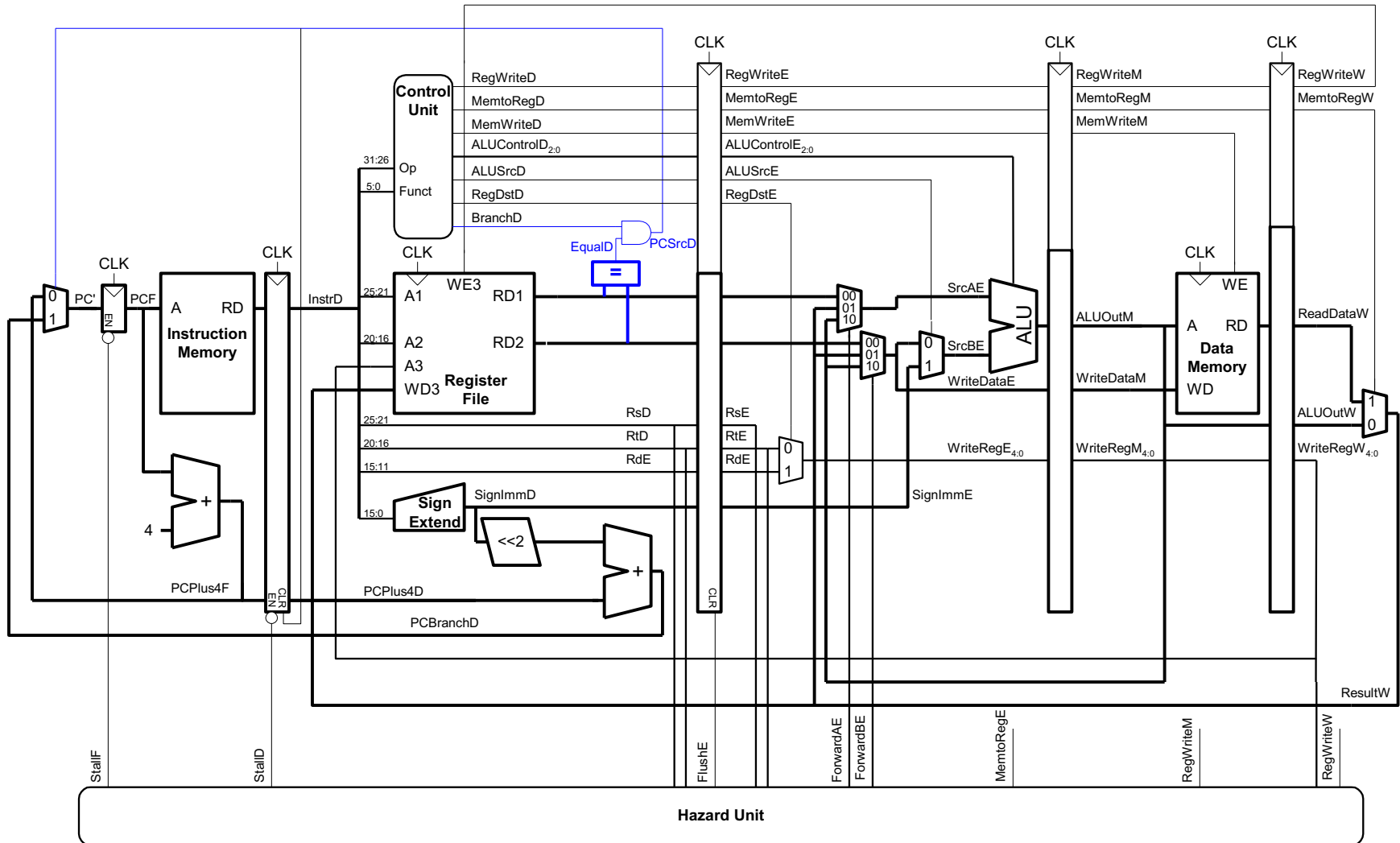
# Control Dependence: Original Pipeline



# Control Dependence

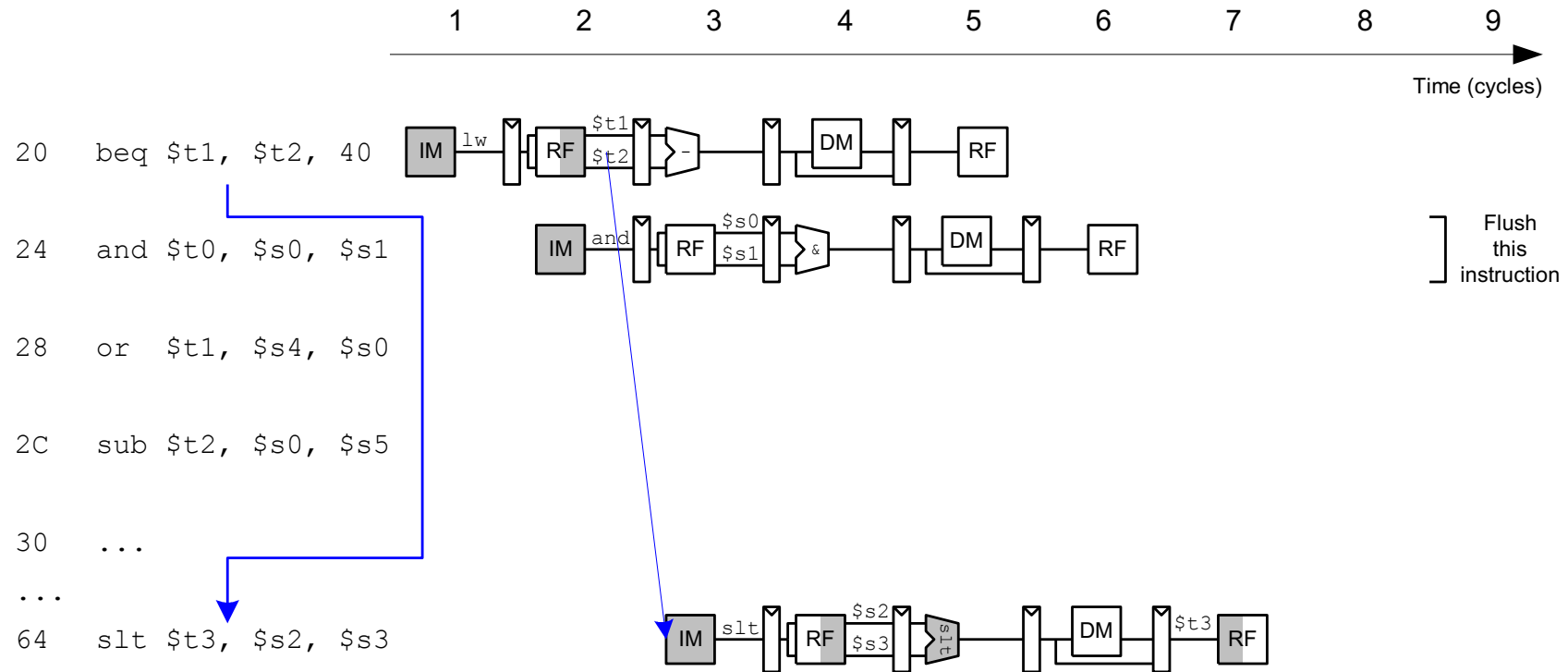


# Early Branch Resolution



Introduces another data dependency in Decode stage..

# Early Branch Resolution



# Early Branch Resolution: Good Idea?

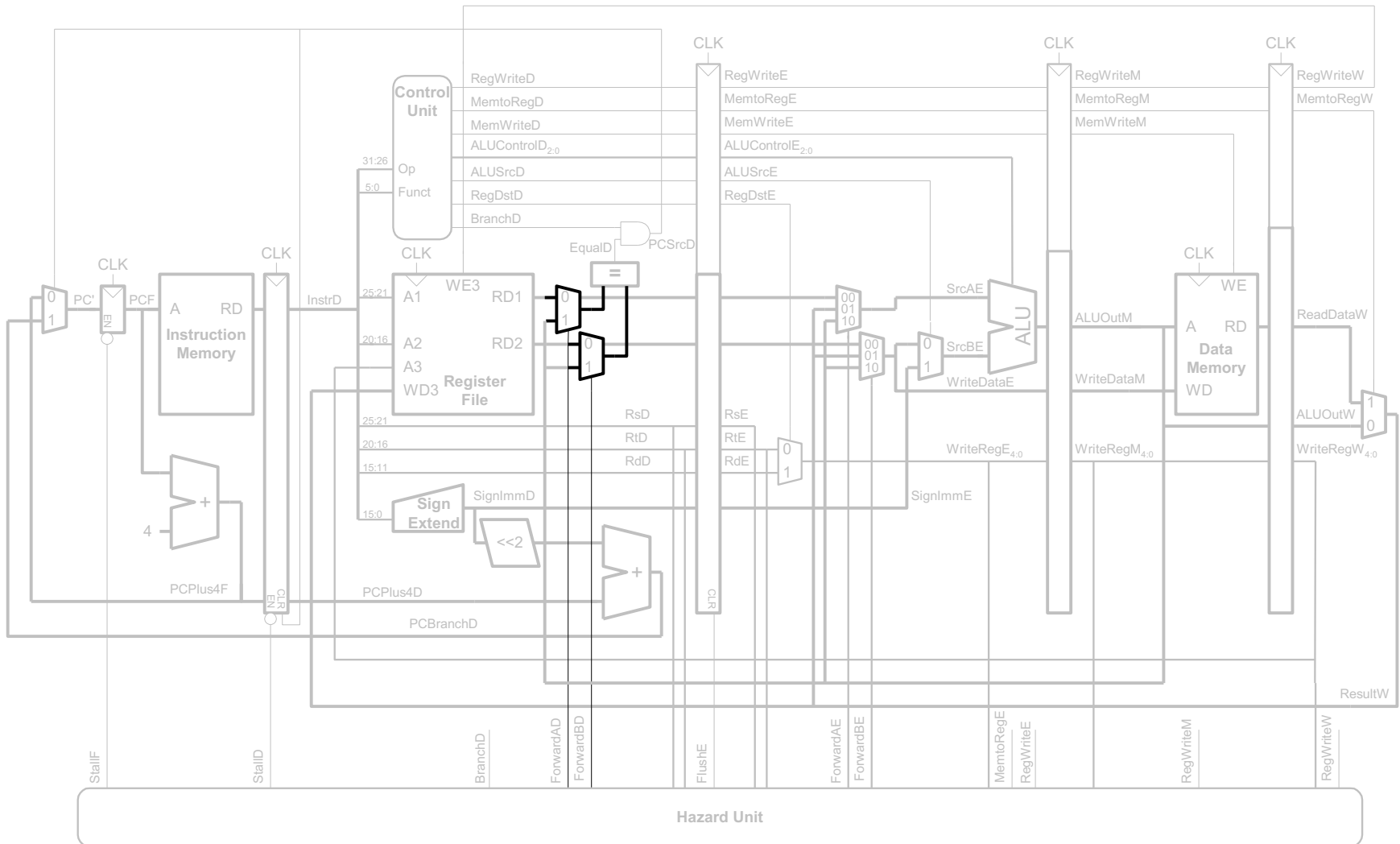
## ■ Advantages

- 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.

# Control Forwarding and Stalling Hardware

```
// 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**



# 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 = (# instructions)  $\times$  CPI  $\times$   $T_c$   
=  $(100 \times 10^9)(1.15)(550 \times 10^{-12})$   
= 63 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.

# 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

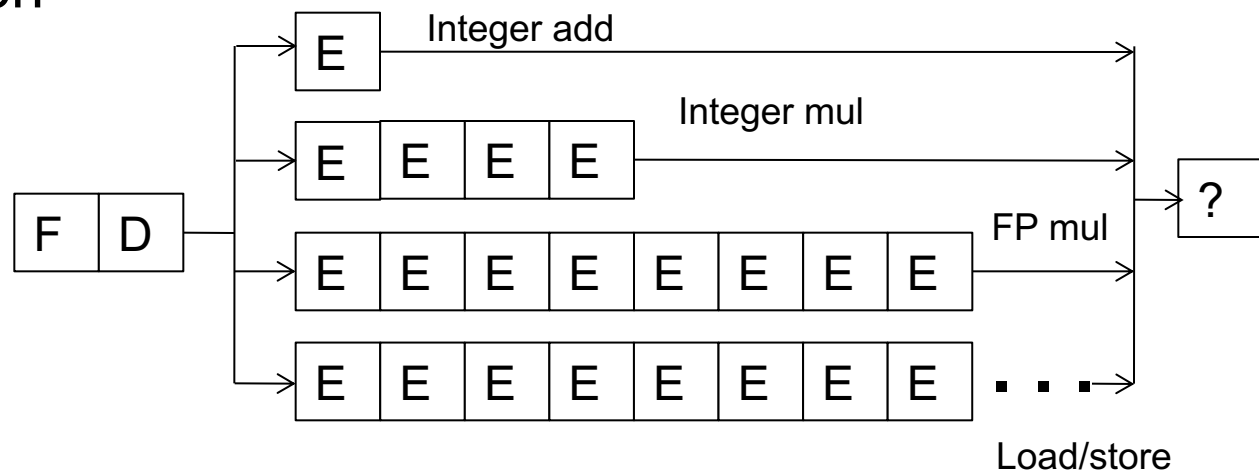
---

- 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

# Pipelining and Precise Exceptions: Preserving Sequential Semantics

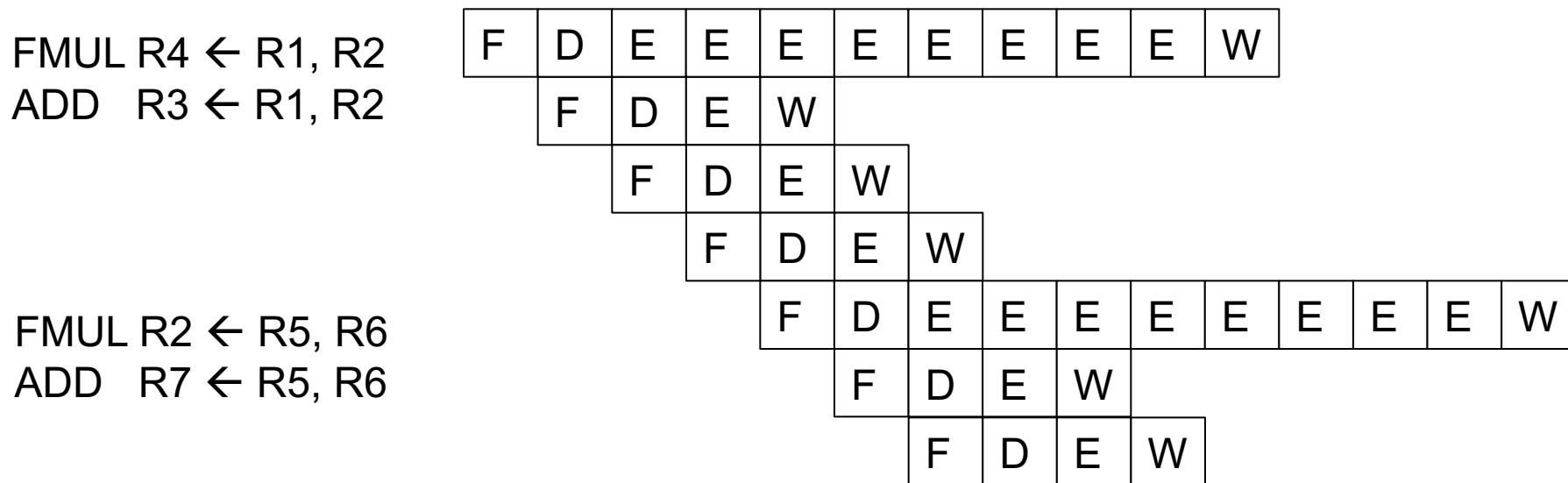
# Multi-Cycle Execution

- 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?

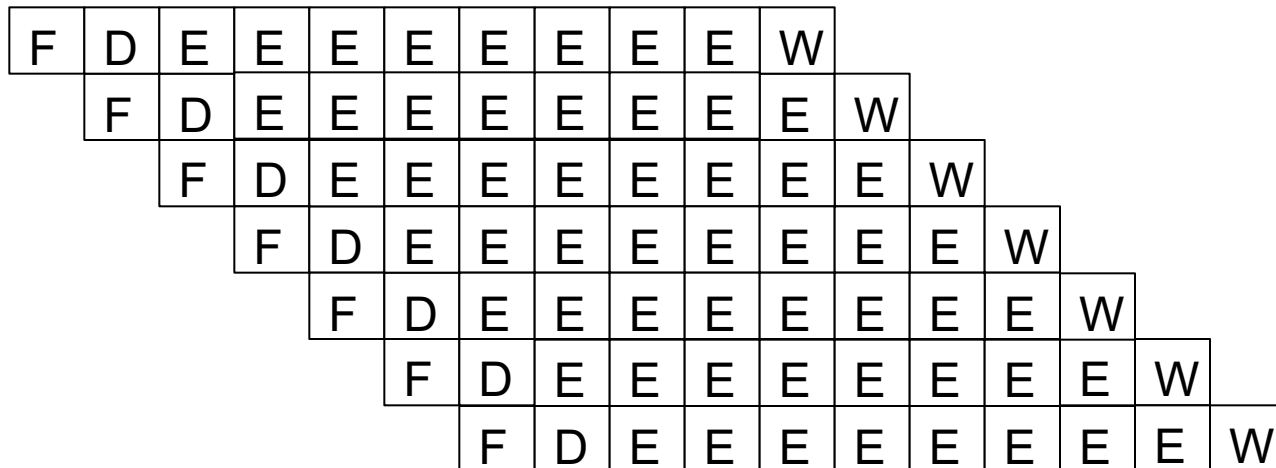
---

- 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 ← R1, R2  
ADD R4 ← 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

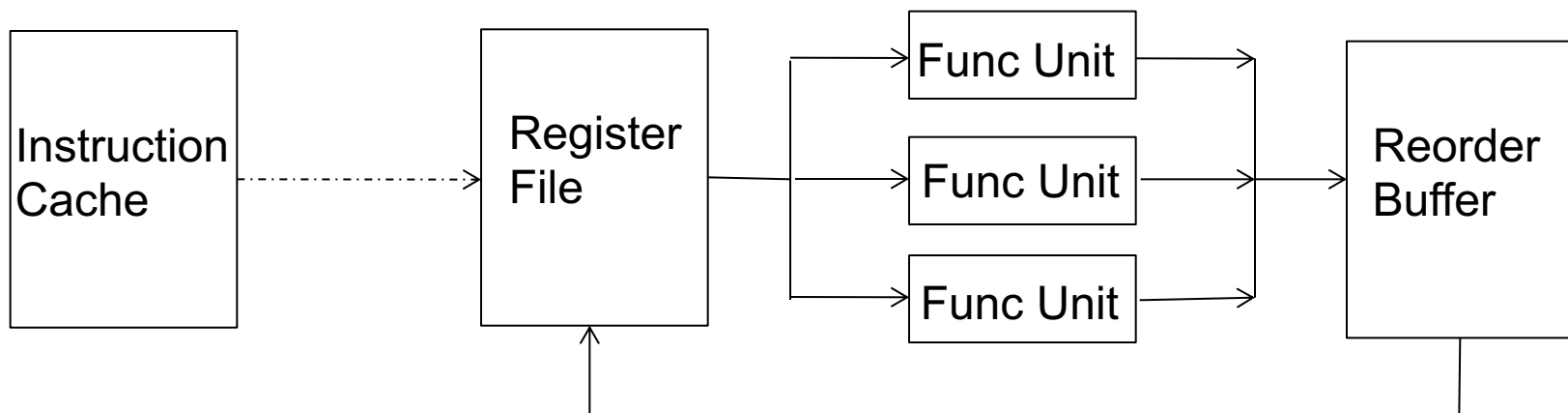
We will not cover these

- 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?

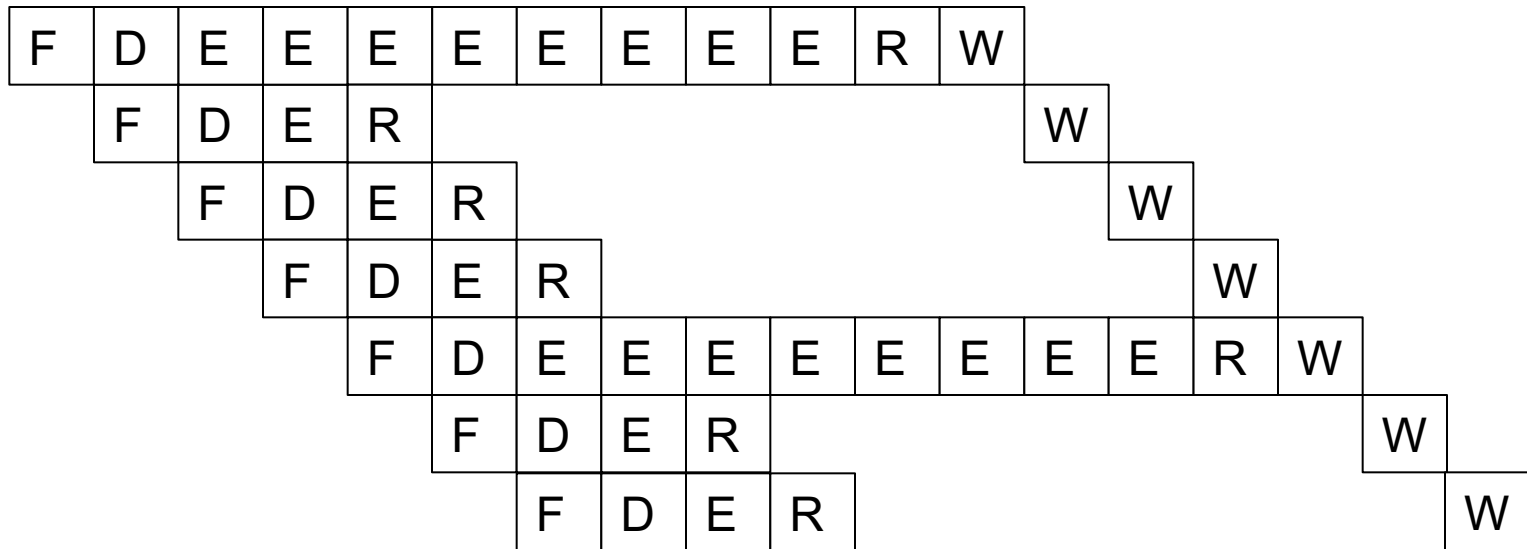
---

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

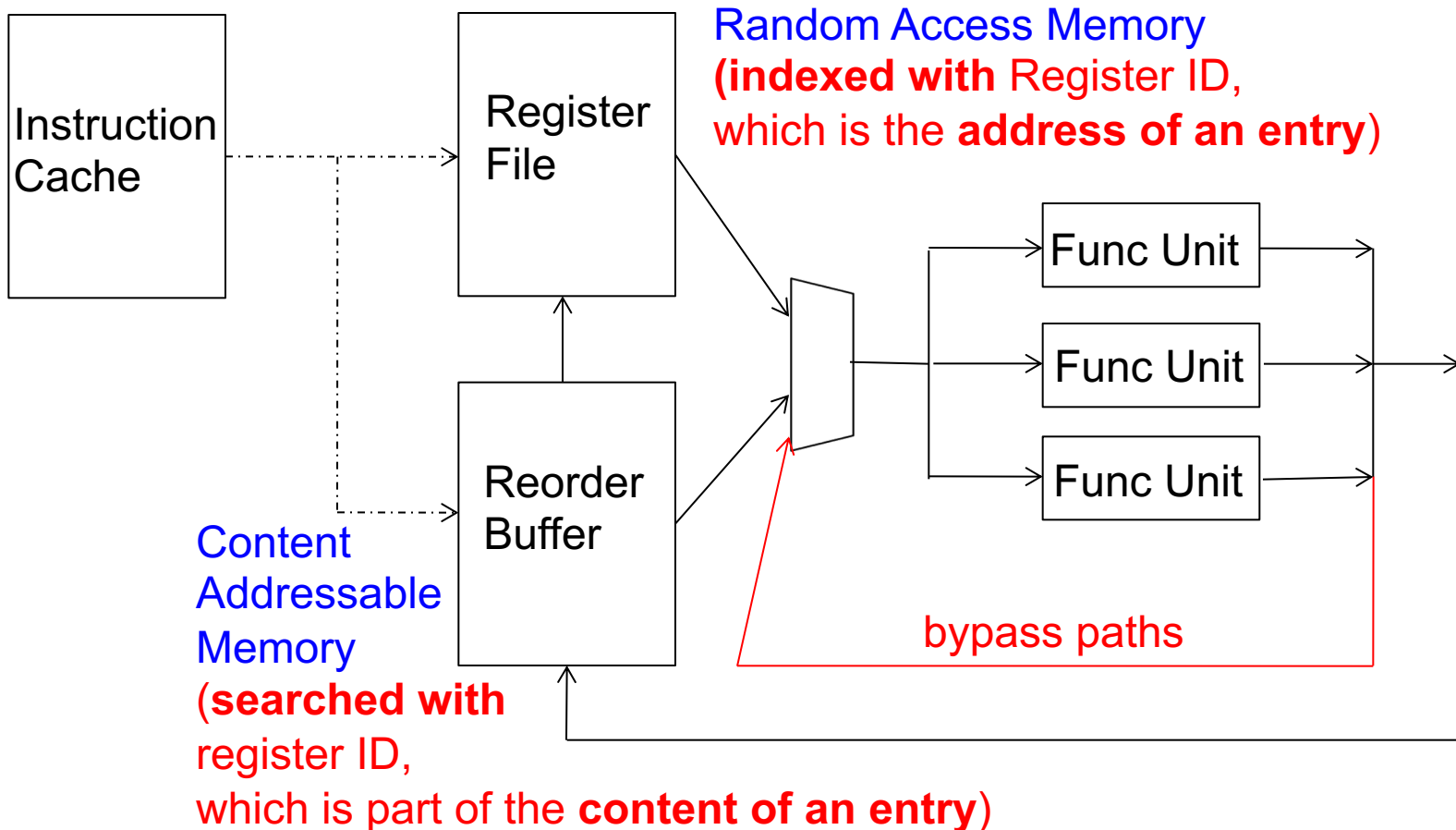
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- 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)





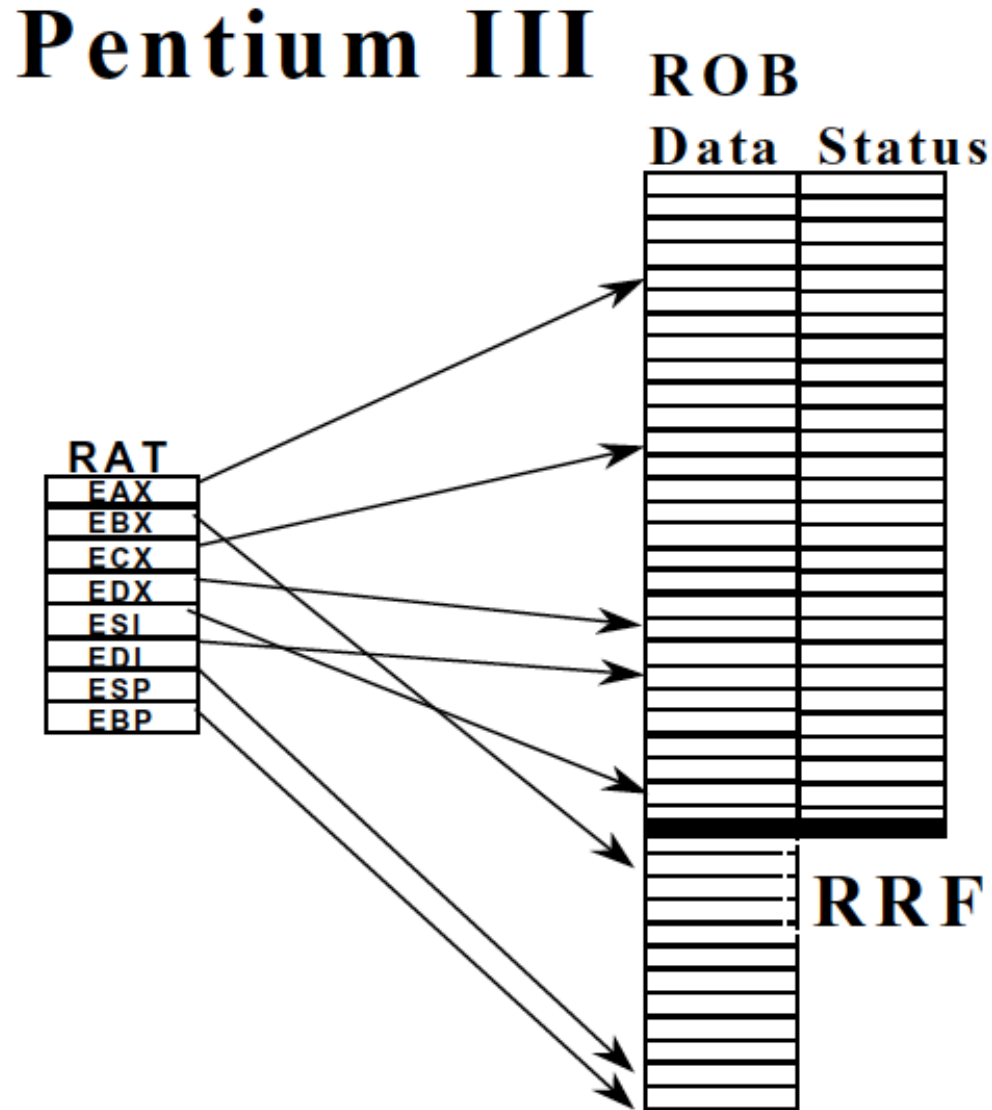
# Simplifying Reorder Buffer Access

---

- 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

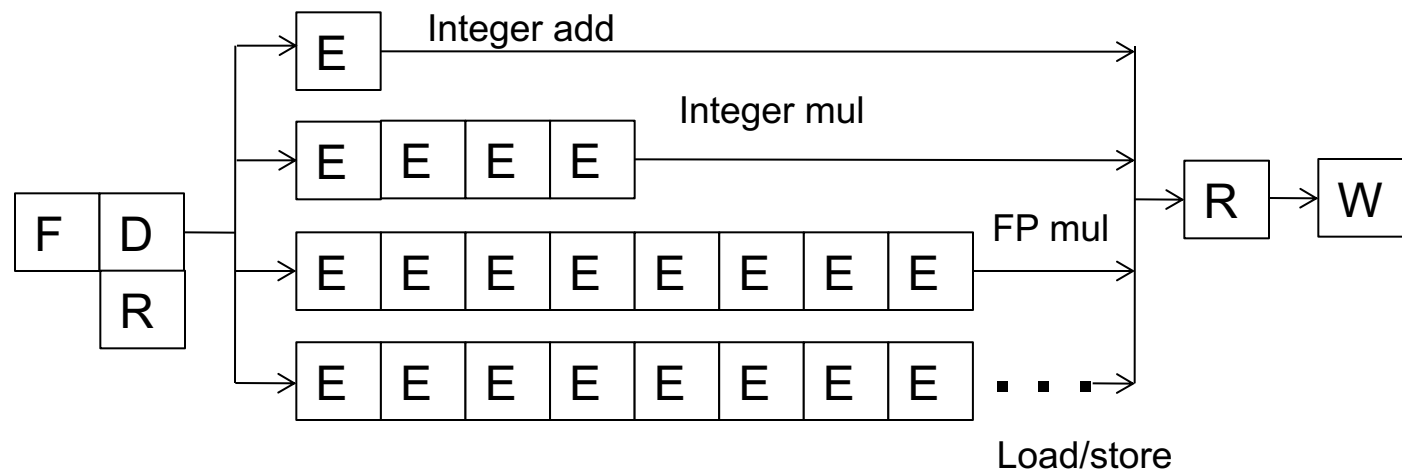
$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



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 We will not cover these
- Checkpointing

# Digital Design & Computer Arch.

## Lecture 14: Pipelining Issues

Prof. Onur Mutlu

ETH Zürich

Spring 2020

3 April 2020