

Digital Design & Computer Arch.

Lecture 15a: Precise Exceptions

Prof. Onur Mutlu

ETH Zürich

Spring 2021

23 April 2021

Required Readings

■ Last week & This week

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

■ This week & 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

■ Earlier

- Single-cycle Microarchitectures
- Multi-cycle Microarchitectures

■ This week

- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...

■ Today & Next week

- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...

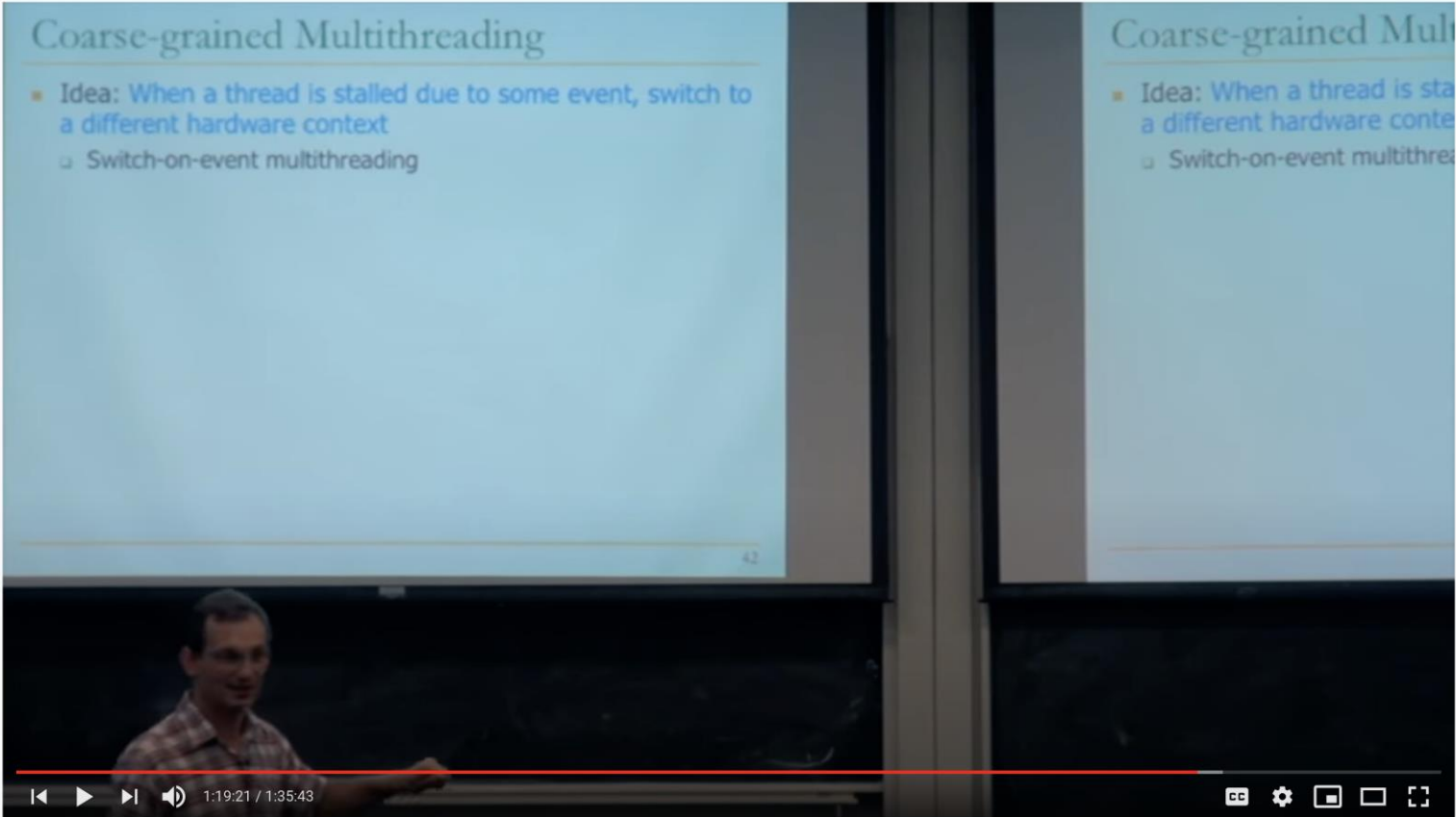
Recall: How to Handle Data Dependences

- 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
- 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

Fine-Grained Multithreading

Modern GPUs are FGMT Machines

More on Multithreading (I)



The video player shows a lecture slide titled "Coarse-grained Multithreading". The slide content is as follows:

- Idea: When a thread is stalled due to some event, switch to a different hardware context
 - Switch-on-event multithreading

The video player interface includes a progress bar at 1:19:21 / 1:35:43, a like count of 10, and a comment count of 0. The video is from the channel "Carnegie Mellon Computer Architecture" (23K subscribers).

Carnegie Mellon Computer Architecture
23K subscribers

Lecture 9: Multithreading
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: September 26, 2013.

ANALYTICS EDIT VIDEO

More on Multithreading (II)

Intel Pentium 4 Hyperthreading

- Long latency load handling
 - Multi-level scheduling window
- More partitioned structures
 - I-TLB
 - Instruction Queues
 - Store buffer
 - Reorder buffer
- 5% area overhead due to SMT

Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal 2002.

replay loop
4-inst
soft line fetch
MICRO HT6

Carnegie Mellon -Parallel Computer Architecture 2012 - Onur Mutlu - Lecture 10 - Multithreading II

1,594 views • Sep 21, 2013

11 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 10: Multithreading II

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 28, 2012.

SUBSCRIBED



More on Multithreading (III)

g (Tandem, Compaq Himalaya)

Lockstepping (Tandem, Compaq Himalaya)

the processor, compare the results of two
re committing an instruction

Idea: Replicate the processor, compare the results of two
processors before committing an instruction

Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 13-Multi-threading II

1,132 views • Sep 21, 2013

8 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 13: Multi-threading III

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: October 5, 2012.

SUBSCRIBED



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

More on Multithreading (IV)



Carnegie Mellon - Parallel Computer Architecture 2013 - Onur Mutlu - Lec 15 - Speculation 1

915 views • Sep 21, 2013

9 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture
1.81K subscribers

Lecture 15: Speculation I

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: October 10, 2012.

SUBSCRIBED



Lectures on Multithreading

■ Parallel Computer Architecture, Fall 2012, Lecture 9

- ❑ Multithreading I (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=iqi9wFqFiNU&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=51

■ Parallel Computer Architecture, Fall 2012, Lecture 10

- ❑ Multithreading II (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=e8lfl6MbILg&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=52

■ Parallel Computer Architecture, Fall 2012, Lecture 13

- ❑ Multithreading III (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=7vkDpZ1-hHM&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=53

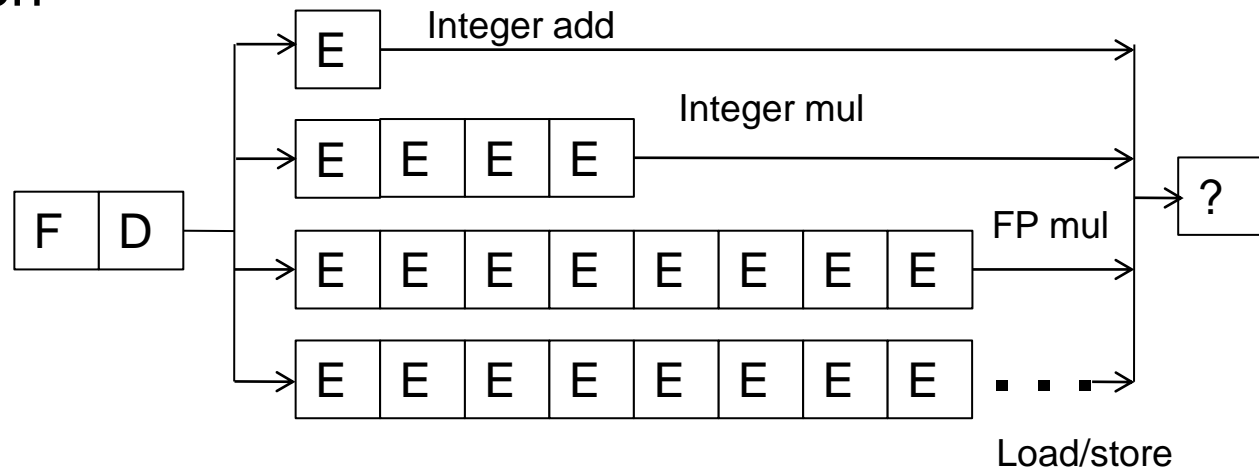
■ Parallel Computer Architecture, Fall 2012, Lecture 15

- ❑ Speculation I (CMU, Fall 2012)
- ❑ https://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=54

Pipelining and Precise Exceptions: Preserving Sequential Semantics

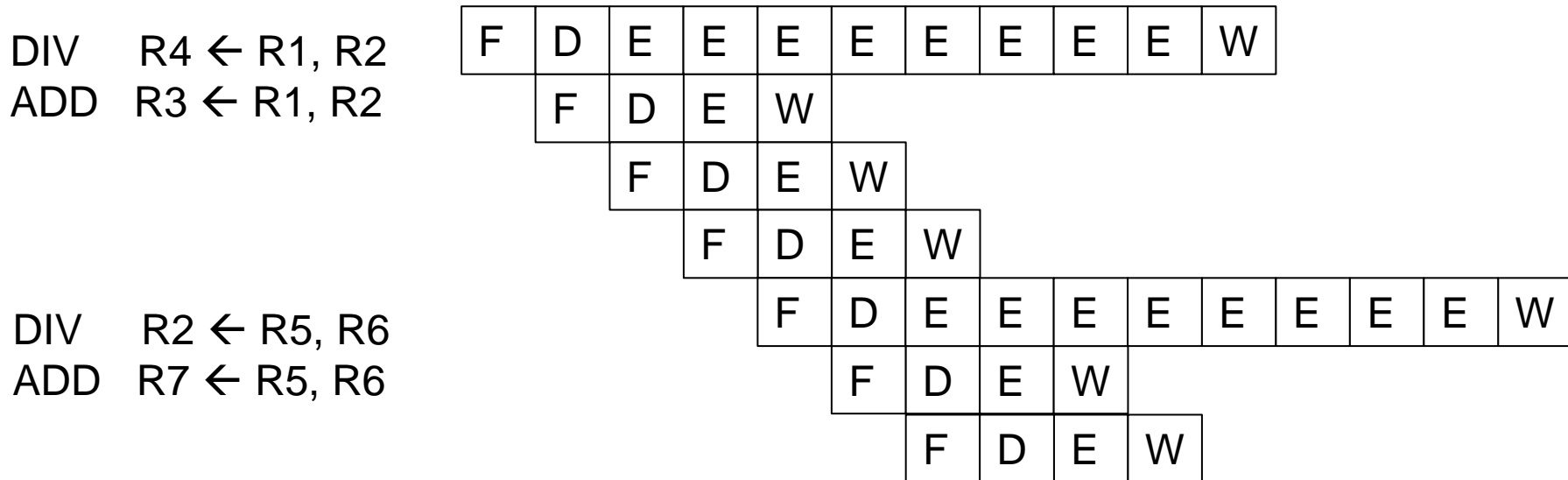
Multi-Cycle Execution

- 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 Integer DIVide



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

Exceptions and Interrupts

- “Unplanned” changes or interruptions in program execution
- Due to internal problems in execution of the program
→ Exceptions
- Due to external events that need to be handled by the processor
→ Interrupts
- Both exceptions and interrupts require
 - ❑ stopping of the current program
 - ❑ saving the architectural state
 - ❑ handling the exception/interrupt → switch to handler
 - ❑ return back to program execution (if possible and makes sense)

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

- Easy to do in single-cycle and multi-cycle machines
- Single-cycle
 - Instruction boundaries == Cycle boundaries
- Multi-cycle
 - Add special states in the control FSM that lead to the exception or interrupt handlers
 - Switch to the handler only at a precise state → before fetching the next instruction

Precise Exceptions in Multi-Cycle FSM

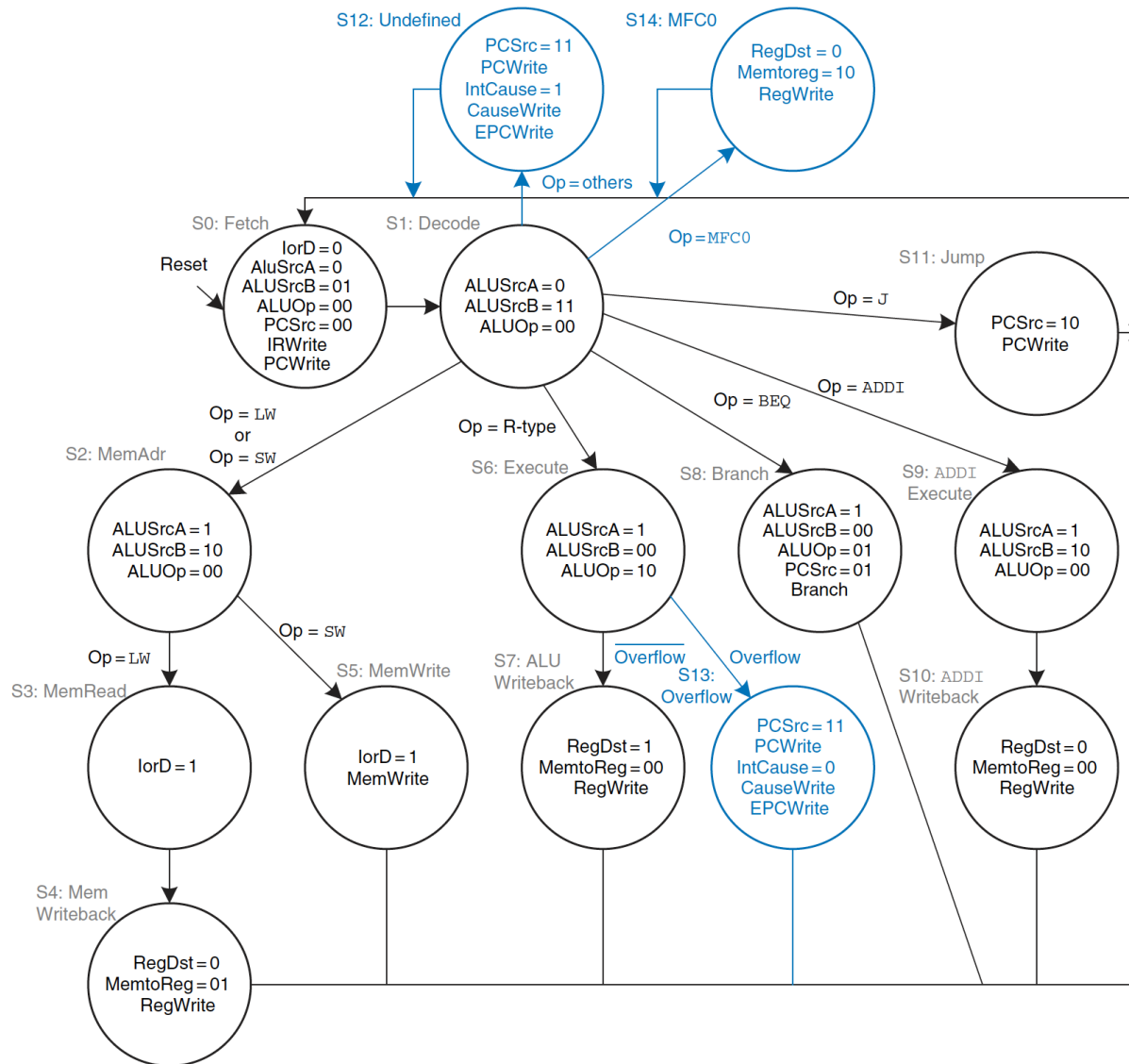


Figure 7.64 Controller supporting exceptions and mfc0

Precise Exceptions in Multi-Cycle Datapath

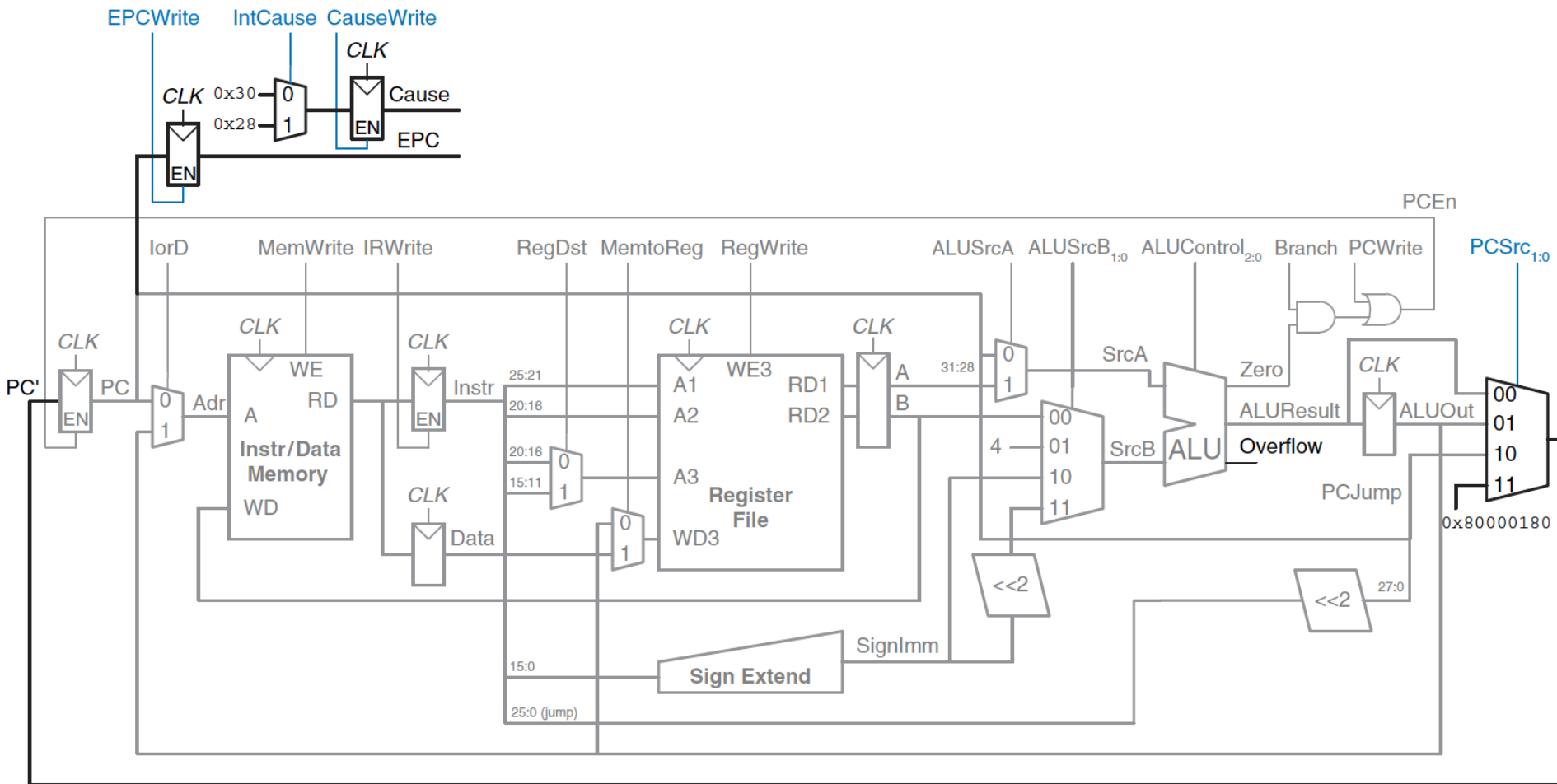
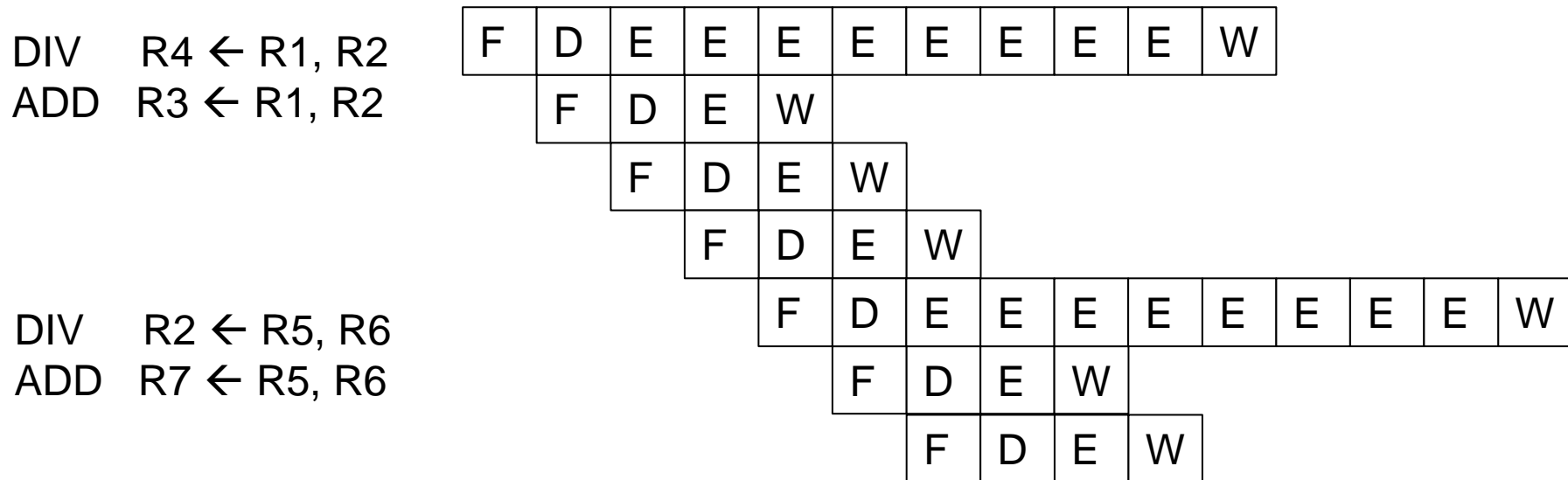


Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

Multi-Cycle Execute: More Complications

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



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

Ensuring Precise Exceptions in Pipelining

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

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

F	D	E	E	E	E	E	E	E	E	W									
	F	D	E	E	E	E	E	E	E	E	W								
		F	D	E	E	E	E	E	E	E	E	W							
			F	D	E	E	E	E	E	E	E	E	W						
				F	D	E	E	E	E	E	E	E	E	W					
					F	D	E	E	E	E	E	E	E	E	W				
						F	D	E	E	E	E	E	E	E	E	W			
							F	D	E	E	E	E	E	E	E	E	W		
								F	D	E	E	E	E	E	E	E	E	W	

- 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

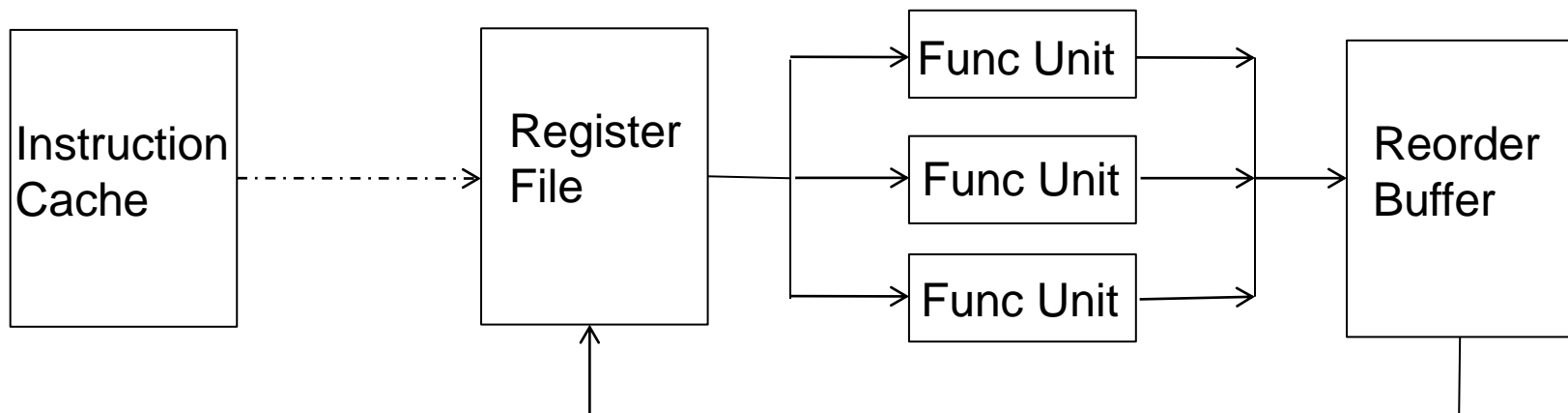
- History buffer
- Future register file
- Checkpointing

We will not cover these
See suggested lecture videos from Spring 2015

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

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

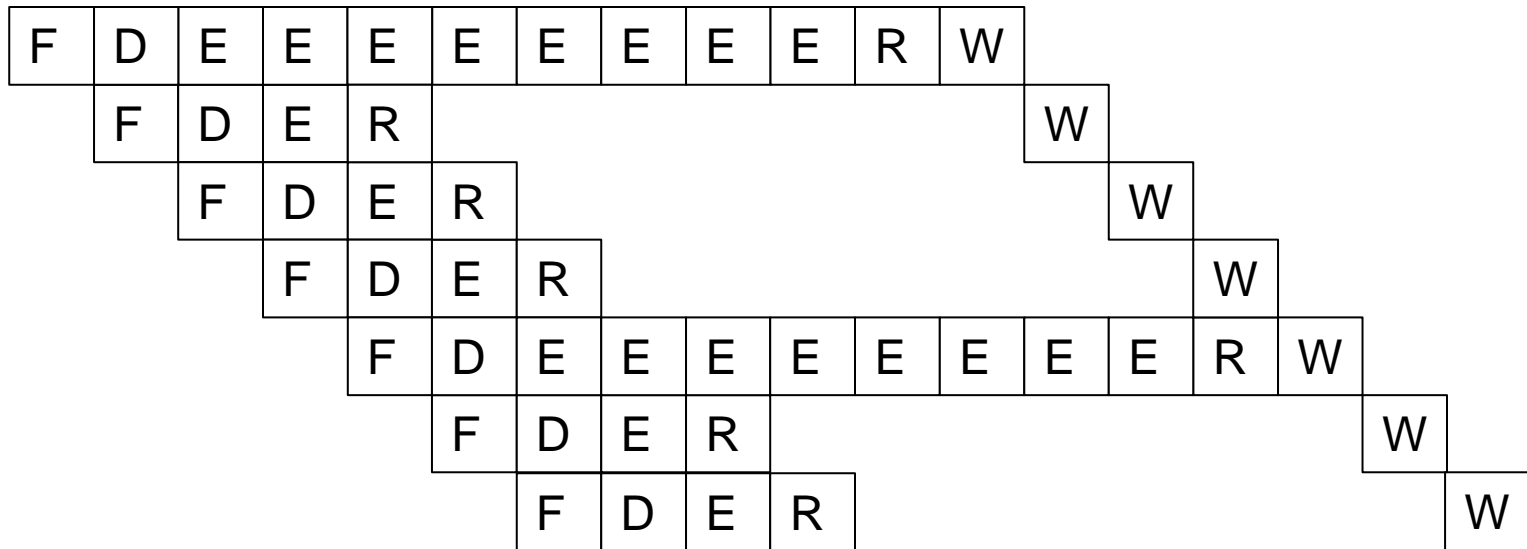
What's in a ROB Entry?

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

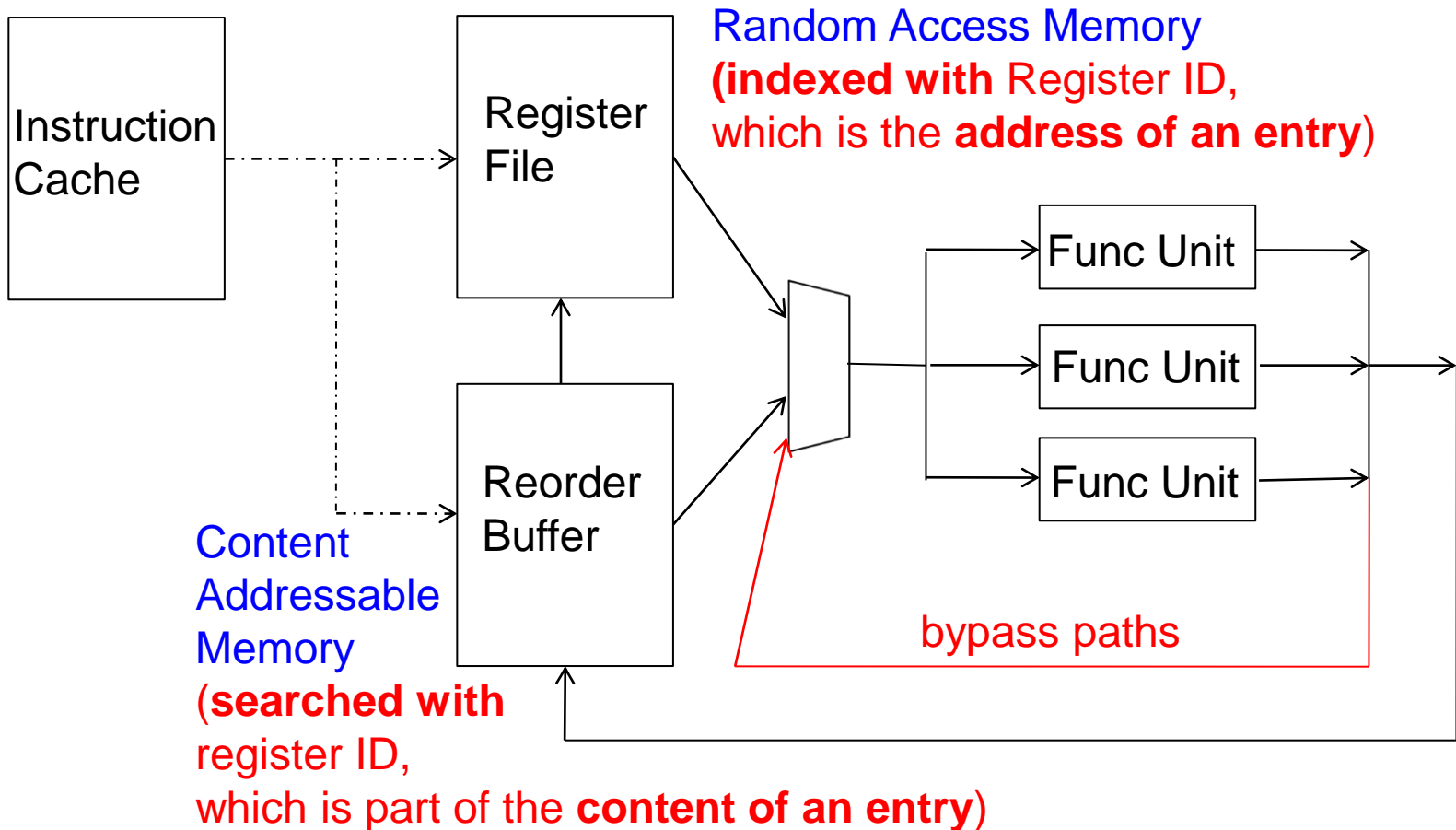
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- 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?

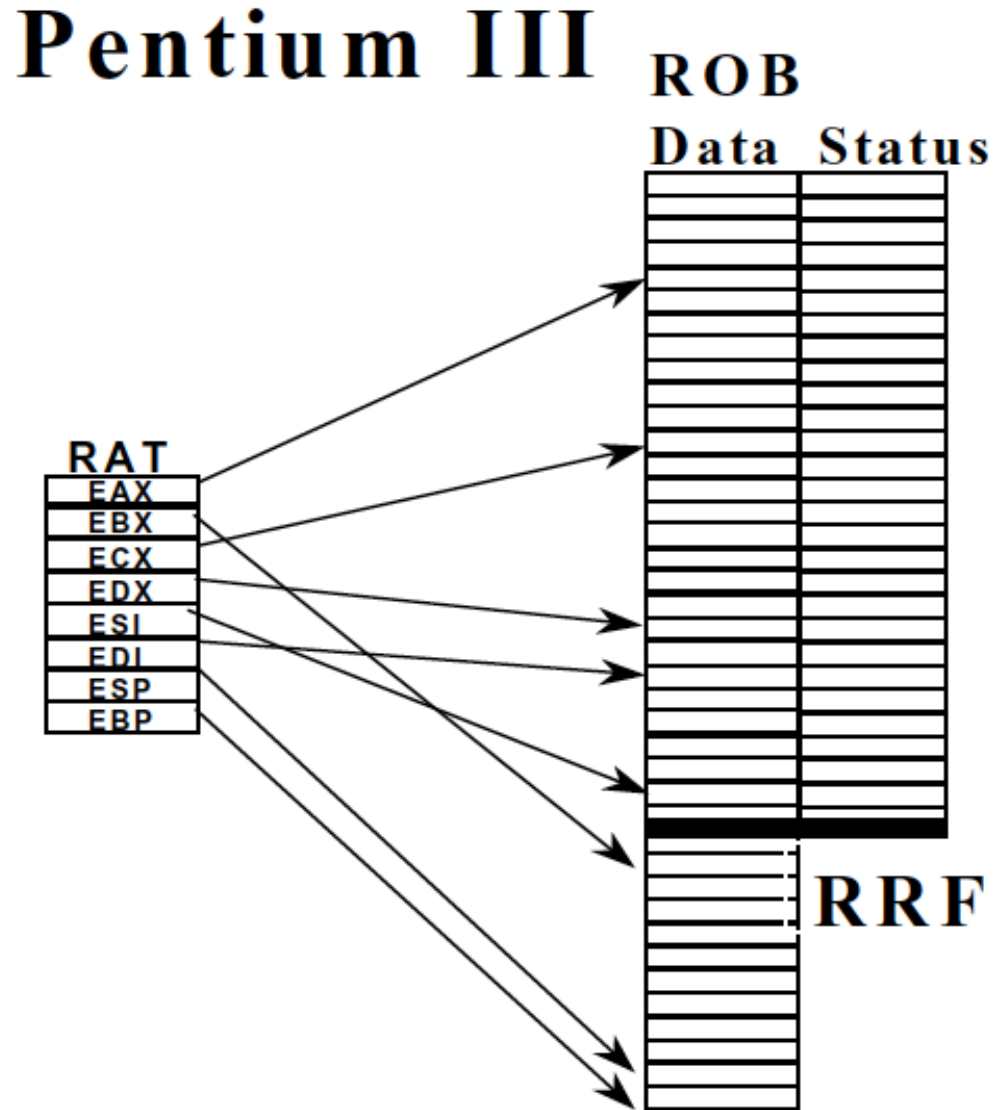
- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



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 dependences are **not true dependences**
 - ❑ 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 dependences
 - ❑ 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

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



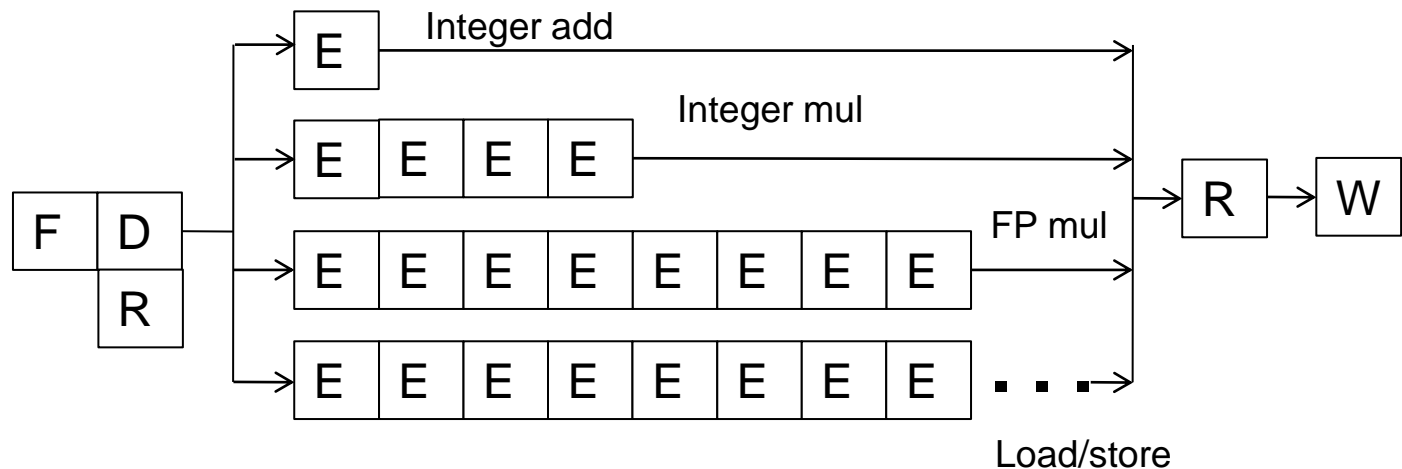
Write-after-Write
(WAW) -- **Output**

Renaming Example

- Assume
 - Register file has a pointer to the reorder buffer entry that contains or will contain the value, if the register is not valid
 - Reorder buffer works as described before
- Where is the latest definition of R3 for each instruction below in sequential order?
 - LD R0(0) → R3
 - LD R3, R1 → R10
 - MUL R1, R2 → R3
 - MUL R3, R4 → R11
 - ADD R5, R6 → R3
 - ADD R7, R8 → R12

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
See suggested lecture videos from Spring 2015

More on State Maintenance & Precise Exceptions

History Buffer

```
graph LR; IC[Instruction Cache] -.-> RF[Register File]; RF --> FU1[Func Unit]; RF --> FU2[Func Unit]; RF --> FU3[Func Unit]; FU1 --> HB[History Buffer]; FU2 --> HB; FU3 --> HB; HB -.->|Used only on exceptions| RF;
```

- Advantage:
 - Register file contains up-to-date values for incoming instructions → History buffer access not on critical path
- Disadvantage:
 - Need to read the old value of the destination register
 - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

29

Lecture 11. Precise Exceptions, State Maintenance/Recovery - CMU - Comp. Arch. 2015 - Onur Mutlu

11,990 views • Feb 12, 2015

77 0 SHARE SAVE ...



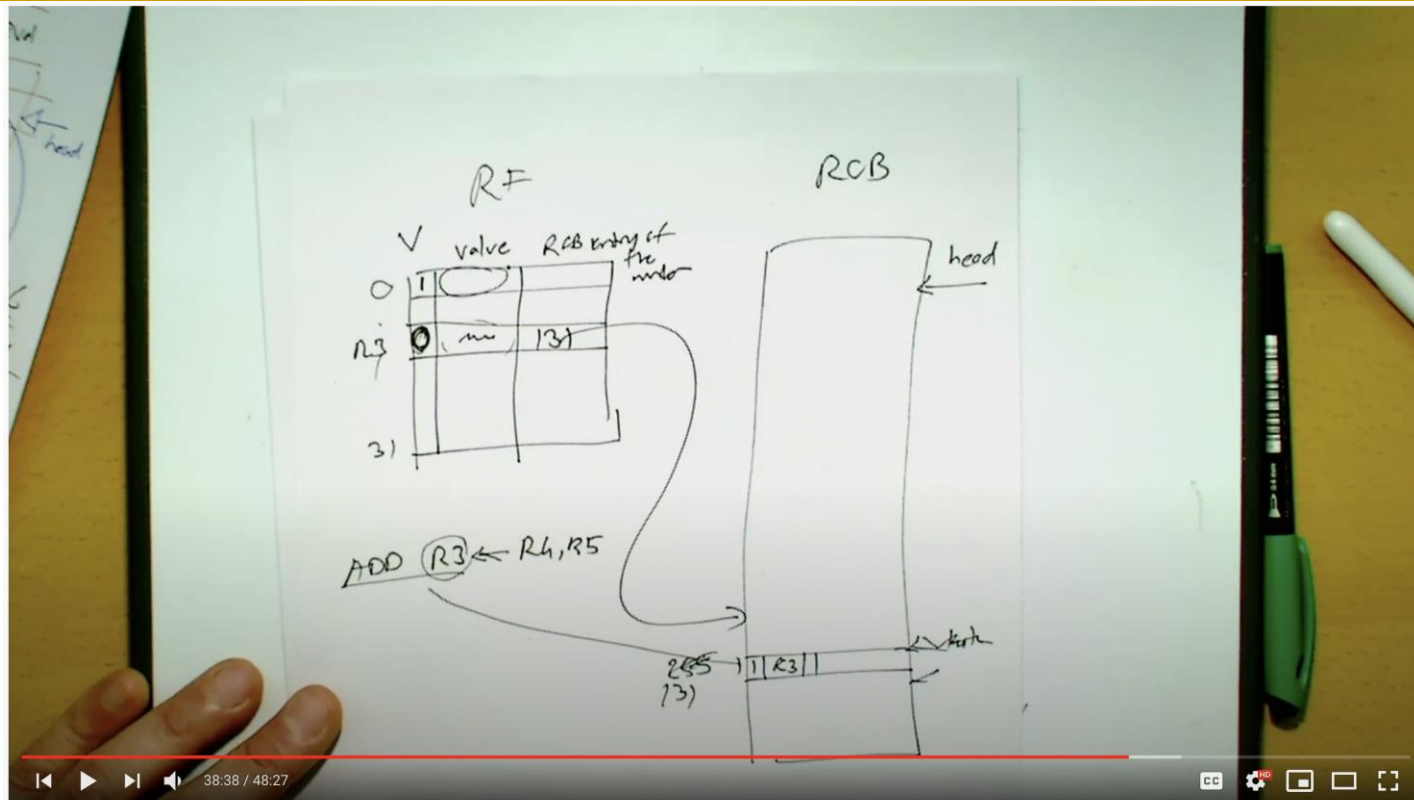
Carnegie Mellon Computer Architecture
23K subscribers

Lecture 11. Precise Exceptions, State Maintenance, State Recovery
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)
Date: Feb 11th, 2015

ANALYTICS

EDIT VIDEO

More on State Maintenance & Precise Exceptions



ETH ZÜRICH HAUPTGEBÄUDE

Design of Digital Circuits - Lecture 15a: Reorder Buffer (ETH Zürich, Spring 2019)

2,462 views • Apr 15, 2019

35 0 SHARE SAVE ...



Onur Mutlu Lectures
15.6K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2019 (<https://safari.ethz.ch/digitaltechnik...>)
Professor Onur Mutlu (<http://people.inf.ethz.ch/omutlu>)

Lecture 15a: Reorder Buffer
Lecturer: Onur Mutlu
Date: April 11, 2019

ANALYTICS

EDIT VIDEO

Lectures on State Maintenance & Recovery

- **Computer Architecture, Spring 2015, Lecture 11**
 - Precise Exceptions, State Maintenance/Recovery (CMU, Spring 2015)
 - <https://www.youtube.com/watch?v=nMfbtzWizDA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=13>
- **Digital Design & Computer Architecture, Spring 2019, Lecture 15a**
 - Reorder Buffer (ETH Zurich, Spring 2019)
 - <https://www.youtube.com/watch?v=9yo3yhUijQs&list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&index=17>

Suggested Readings for the Interested

- Smith and Plezskun, “Implementing Precise Interrupts in Pipelined Processors,” IEEE Trans on Computers 1988 and ISCA 1985.
- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, 1995
- Hwu and Patt, “Checkpoint Repair for Out-of-order Execution Machines,” ISCA 1987.
- Backup Slides

Digital Design & Computer Arch.

Lecture 15a: Precise Exceptions

Prof. Onur Mutlu

ETH Zürich

Spring 2021

23 April 2021

Backup Slides on Precise Exceptions

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

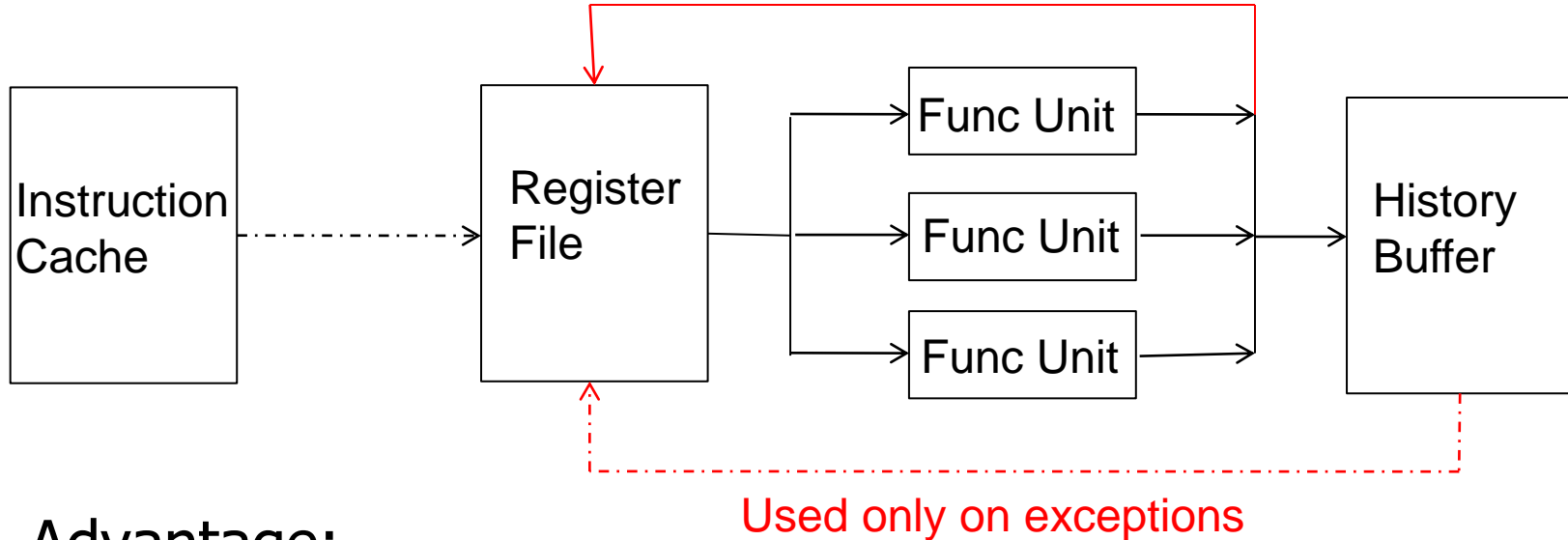
■ Other solutions aim to eliminate the disadvantages

- ❑ History buffer
- ❑ Future file
- ❑ Checkpointing

Solution II: History Buffer (HB)

- Idea: Update the register file when instruction completes, but UNDO UPDATES when an exception occurs
- When instruction is decoded, it reserves an HB entry
- When the instruction completes, it stores the old value of its destination in the HB
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head

History Buffer



■ Advantage:

- ❑ Register file contains up-to-date values for incoming instructions
→ History buffer access not on critical path

■ Disadvantage:

- ❑ Need to read the old value of the destination register
- ❑ Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

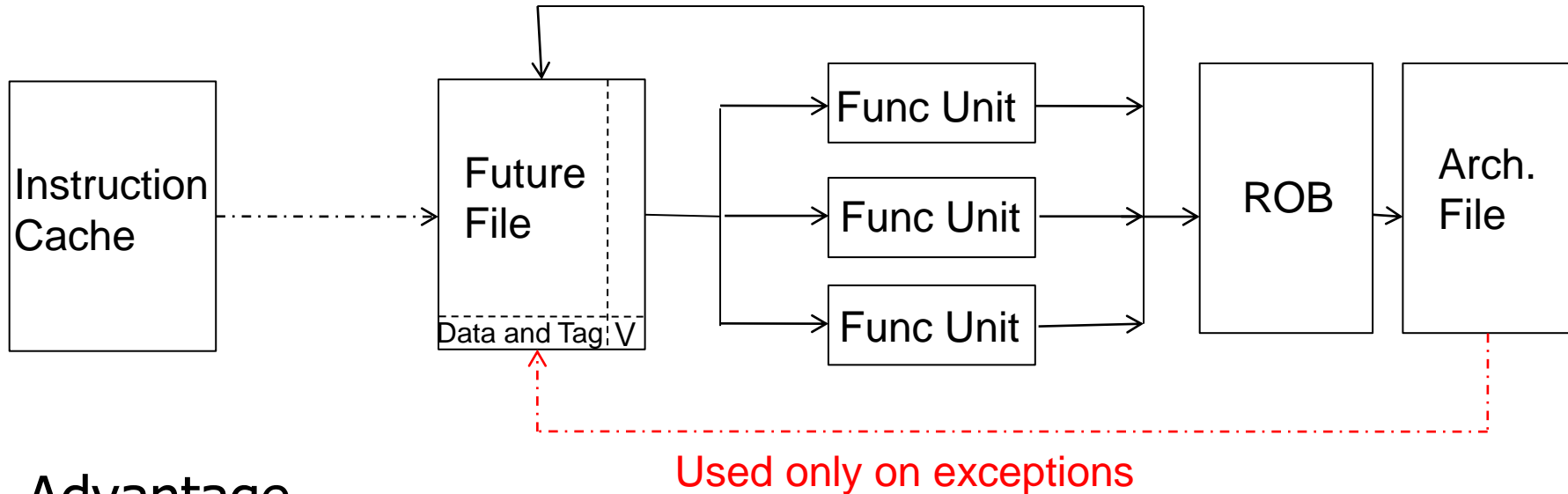
Comparison of Two Approaches

- Reorder buffer
 - ❑ Pessimistic register file update
 - ❑ Update only with non-speculative values (in program order)
 - ❑ Leads to complexity/delay in accessing the new values
- History buffer
 - ❑ Optimistic register file update
 - ❑ Update immediately, but log the old value for recovery
 - ❑ Leads to complexity/delay in logging old values
- Can we get the best of both worlds?
 - ❑ Principle: Heterogeneity
 - ❑ Idea: Have both types of register files

Solution III: Future File (FF) + ROB

- Idea: Keep two register files (speculative and architectural)
 - Arch reg file: Updated in program order for precise exceptions
 - Use a reorder buffer to ensure in-order updates
 - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)
- Future file is used for fast access to latest register values (speculative state)
 - Frontend register file
- Architectural file is used for state recovery on exceptions (architectural state)
 - Backend register file

Future File



■ Advantage

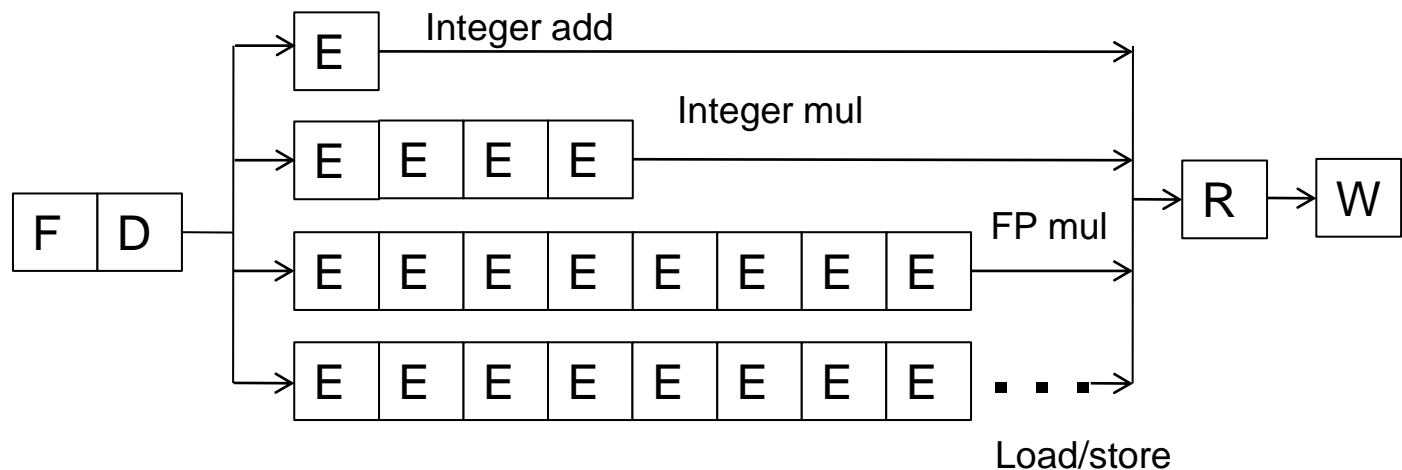
- ❑ No need to read the new values from the ROB (no CAM or indirection) or the old value of destination register

■ Disadvantage

- ❑ Multiple register files
- ❑ Need to copy arch. reg. file to future file on an exception

In-Order Pipeline with Future File and Reorder Buffer

- **Decode (D)**: Access future file, 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 **and future file**
- **Retirement/Commit (W)**: Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline, **copy architectural file to future file**, and start from exception handler
- **In-order dispatch/execution, out-of-order completion, in-order retirement**

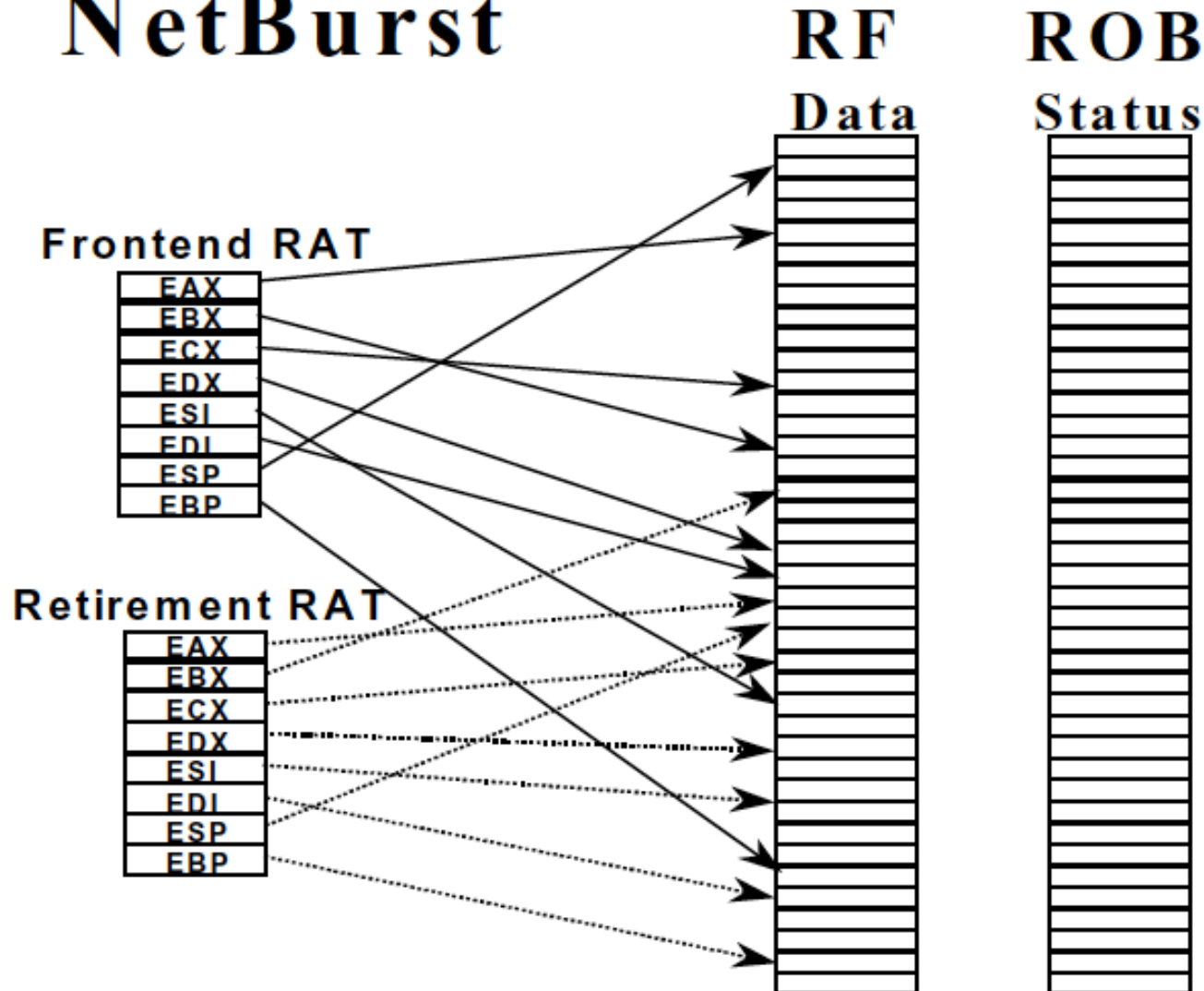


Can We Reduce the Overhead of Two Register Files?

- Idea: Use indirection, i.e., pointers to data in frontend and retirement
 - Have a single storage that stores register data values
 - Keep two register maps (speculative and architectural); also called register alias tables (RATs)
- Future map used for fast access to latest register values (speculative state)
 - Frontend register map
- Architectural map is used for state recovery on exceptions (architectural state)
 - Backend register map

Future Map in Intel Pentium 4

NetBurst



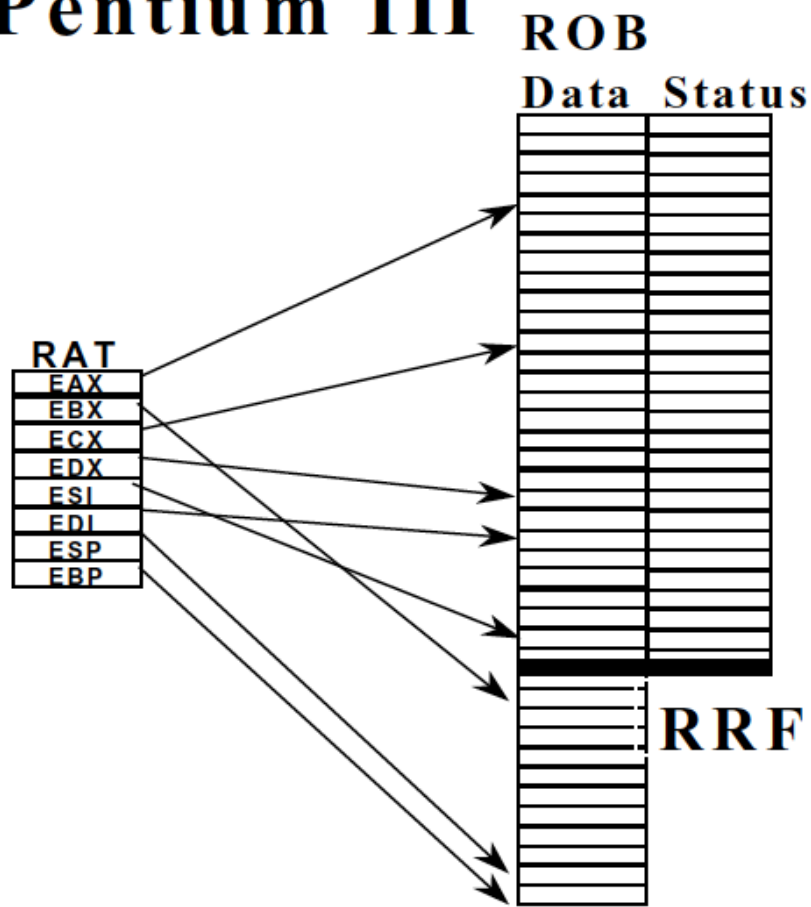
Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Many modern processors are similar:

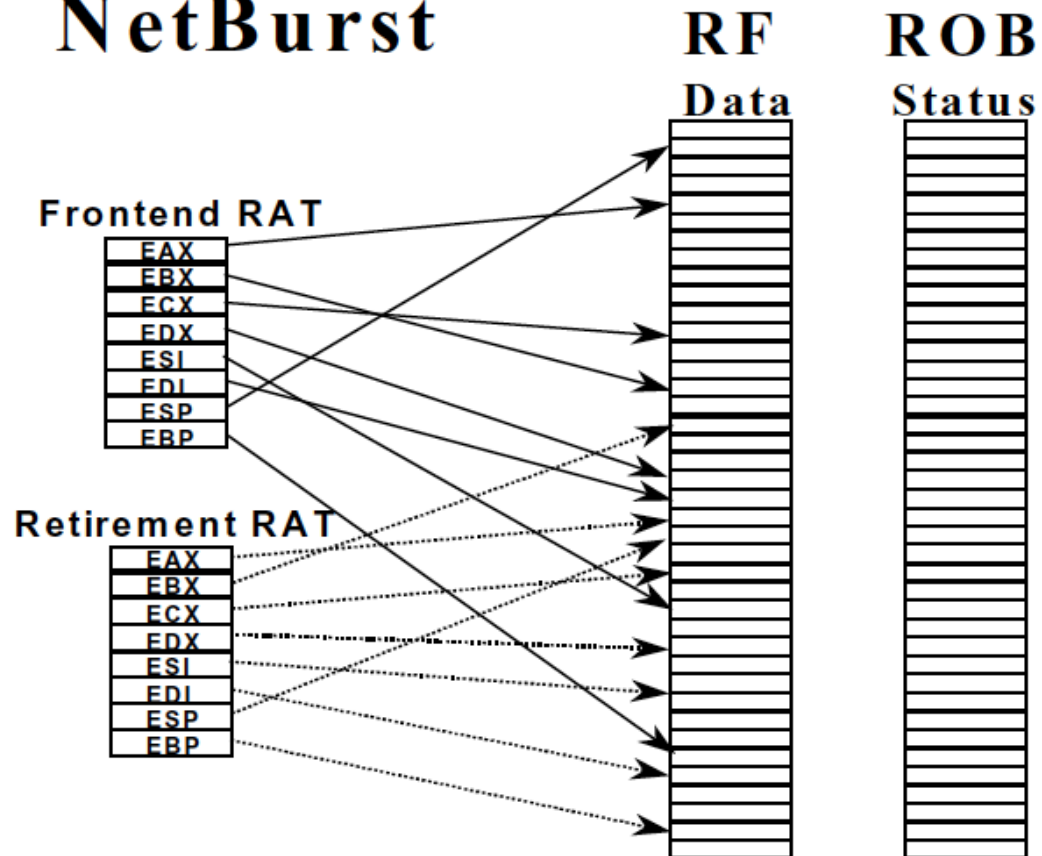
- MIPS R10K
- Alpha 21264

Reorder Buffer vs. Future Map Comparison

Pentium III



NetBurst



Before We Get to Checkpointing ...

- Let's cover what happens on exceptions
- And branch mispredictions

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
 - ❑ Recovers architectural state (register file, IP, and memory)
 - ❑ Flushes all younger instructions in the pipeline
 - ❑ Saves IP and registers (as specified by the ISA)
 - ❑ Redirects the fetch engine to the exception handling routine
 - Vectored exceptions

Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an “exception”
 - Except it is not visible to software (i.e., it is microarchitectural)
- What about branch misprediction recovery?
 - Similar to exception handling except can be initiated before the branch is the oldest instruction (not architectural)
 - All three state recovery methods can be used
- Difference between exceptions and branch mispredictions?
 - Branch mispredictions are much more common
 - need fast state recovery to minimize performance impact of mispredictions

How Fast Is State Recovery?

- Latency of state recovery affects
 - ❑ Exception service latency
 - ❑ Interrupt service latency
 - ❑ Latency to supply the correct data to instructions fetched after a branch misprediction
- Which ones above need to be fast?
- How do the three state maintenance methods fare in terms of recovery latency?
 - ❑ Reorder buffer
 - ❑ History buffer
 - ❑ Future file

Branch State Recovery Actions and Latency

- Reorder Buffer
 - ❑ Flush instructions in pipeline younger than the branch
 - ❑ Finish all instructions in the reorder buffer

- History buffer
 - ❑ Flush instructions in pipeline younger than the branch
 - ❑ Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values one by one into the register file

- Future file
 - ❑ Wait until branch is the oldest instruction in the machine
 - ❑ Copy arch. reg. file to future file
 - ❑ Flush entire pipeline

Can We Do Better?

- Goal: Restore the frontend state (future file) such that the correct next instruction after the branch can execute right away after the branch misprediction is resolved
- Idea: Checkpoint the frontend register state/map at the time a branch is decoded and keep the checkpointed state updated with results of instructions older than the branch
 - Upon branch misprediction, restore the checkpoint associated with the branch
- Hwu and Patt, “Checkpoint Repair for Out-of-order Execution Machines,” ISCA 1987.

Checkpointing

- When a branch is decoded
 - Make a copy of the future file/map and associate it with the branch
- When an instruction produces a register value
 - All future file/map checkpoints that are younger than the instruction are updated with the value
- When a branch misprediction is detected
 - Restore the checkpointed future file/map for the mispredicted branch when the branch misprediction is resolved
 - Flush instructions in pipeline younger than the branch
 - Deallocate checkpoints younger than the branch

Checkpointing

■ Advantages

- Correct frontend register state available right after checkpoint restoration → Low state recovery latency
- ...

■ Disadvantages

- Storage overhead
- Complexity in managing checkpoints
- ...

Many Modern Processors Use Checkpointing

- MIPS R10000
- Alpha 21264
- Pentium 4

- Yeager, “The MIPS R10000 Superscalar Microprocessor,” IEEE Micro, April 1996

- Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro, March-April 1999.

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

Summary: Maintaining Precise State

- Reorder buffer
- History buffer
- Future register file
- Checkpointing
- Readings
 - Smith and Plezskun, “[Implementing Precise Interrupts in Pipelined Processors](#),” IEEE Trans on Computers 1988 and ISCA 1985.
 - Hwu and Patt, “[Checkpoint Repair for Out-of-order Execution Machines](#),” ISCA 1987.

Registers versus Memory

- So far, we considered mainly registers as part of state
- What about memory?
- What are the fundamental differences between registers and memory?
 - Register dependences known statically – memory dependences determined dynamically
 - Register state is small – memory state is large
 - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)

Maintaining Speculative Memory State: Stores

- Handling out-of-order completion of memory operations
 - UNDOing a memory write more difficult than UNDOing a register write. Why?
 - One idea: Keep store address/data in reorder buffer
 - How does a load instruction find its data?
 - Store/write buffer: Similar to reorder buffer, but used only for store instructions
 - Program-order list of un-committed store operations
 - When store is decoded: Allocate a store buffer entry
 - When store address and data become available: Record in store buffer entry
 - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data
- We will get back to this!