Digital Design & Computer Arch.

Lecture 15a: Precise Exceptions

Prof. Onur Mutlu

ETH Zürich

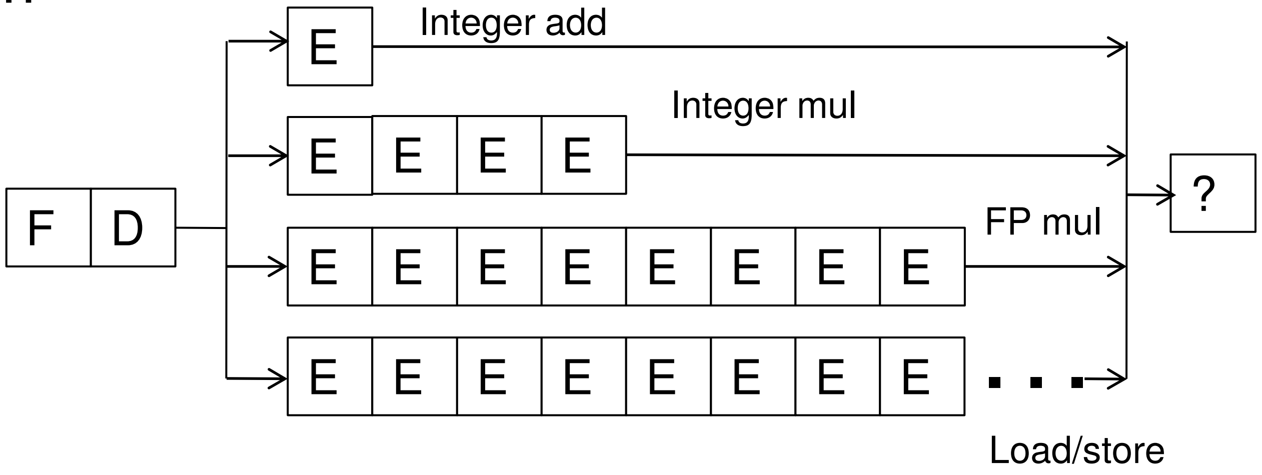
Spring 2021

23 April 2021

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

DIV R4 ← R1, R2	F	D	E	E	E	E	E	E	E	E	W				
ADD R3 ← R1, R2		F	D	E	W										
			F	D	E	W									
				F	D	E	W								
DIV R2 ← R5, R6					F	D	E	E	E	E	E	E	E	E	W
ADD R7 ← R5, R6						F	D	E	W						
							F	D	E	W					

  - 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

See H&H Section 7.7 for a treatment of exceptions in multi-cycle uarch

Multi-Cycle Execute: More Complications

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

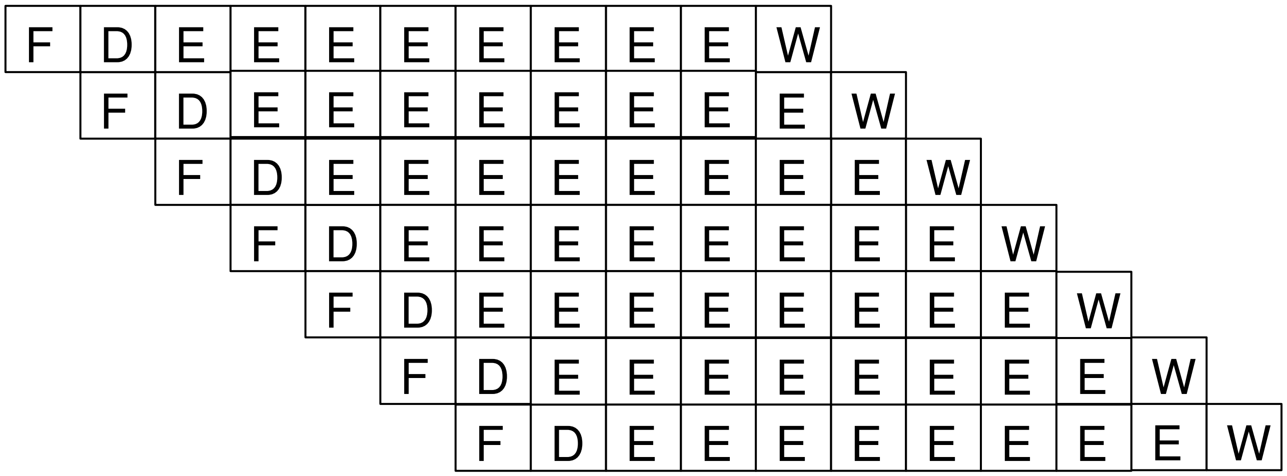
DIV R4 ← R1, R2	F	D	E	E	E	E	E	E	E	E	W				
ADD R3 ← R1, R2		F	D	E	W										
			F	D	E	W									
				F	D	E	W								
DIV R2 ← R5, R6					F	D	E	E	E	E	E	E	E	E	W
ADD R7 ← R5, R6						F	D	E	W						
							F	D	E	W					

  - 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 ← 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

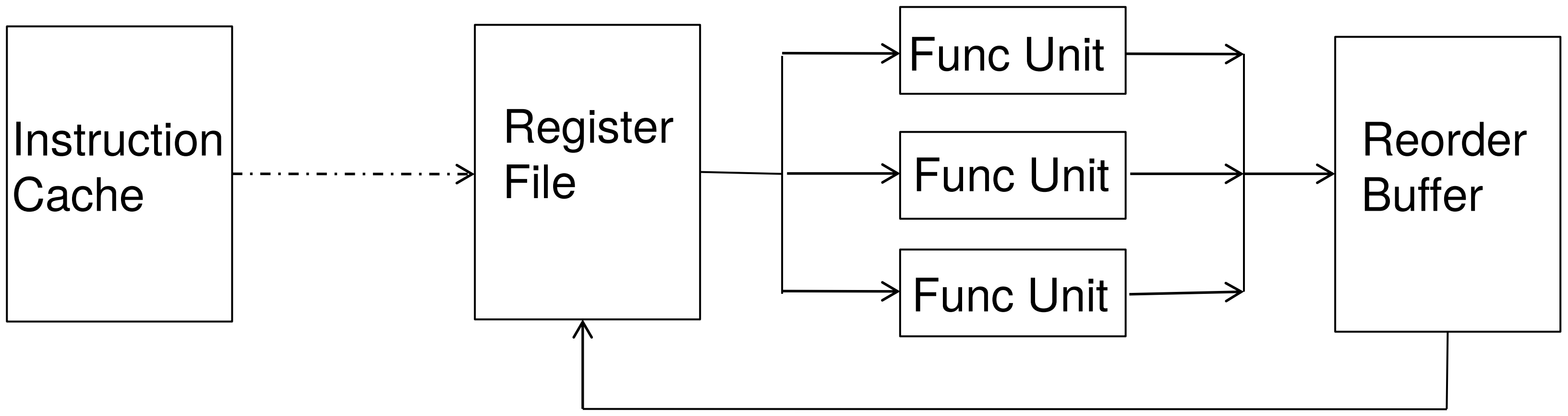
- Checkpointing

We will not cover these
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

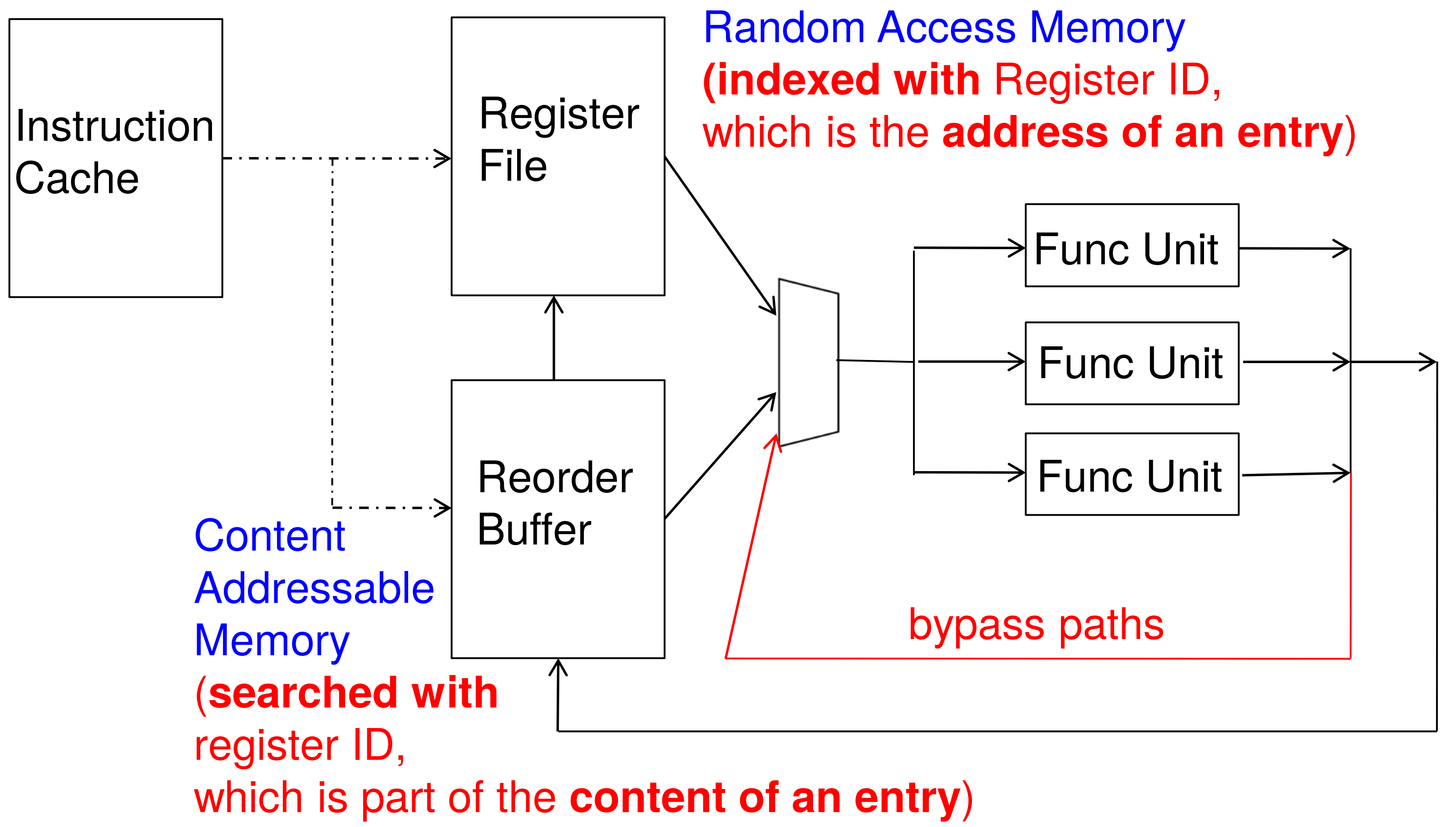
- Result first written to ROB on instruction completion
- Result written to register file at commit time

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

- 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)


Instruction Cache
Register File
Random Access Memory
(**indexed with** Register ID,
which is the **address of an entry**)
Func Unit
Func Unit
Func Unit
Reorder Buffer
Content Addressable Memory
(**searched with** register ID,
which is part of the **content of an entry**)
bypass paths

Simplifying Reorder Buffer Access

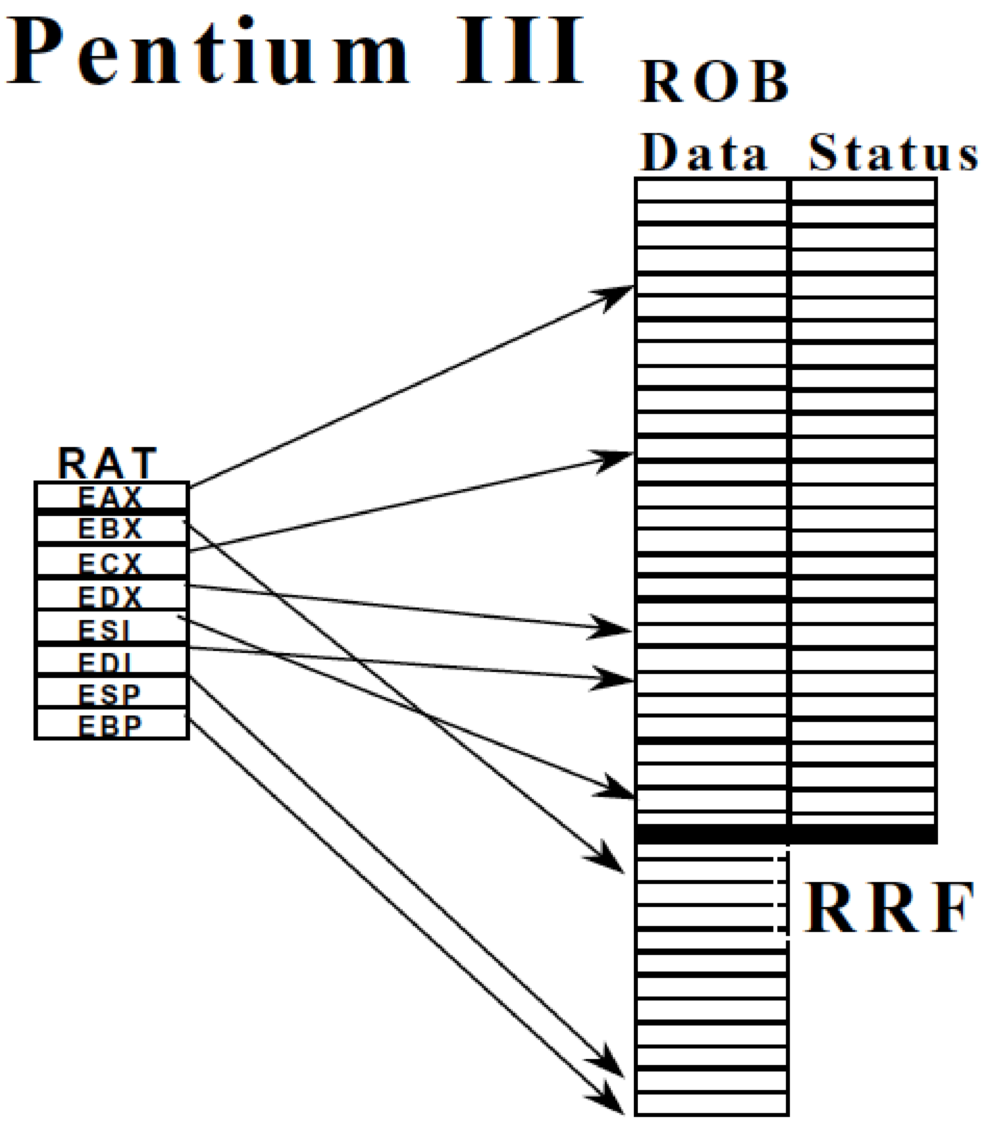
- Idea: Use indirection

- 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

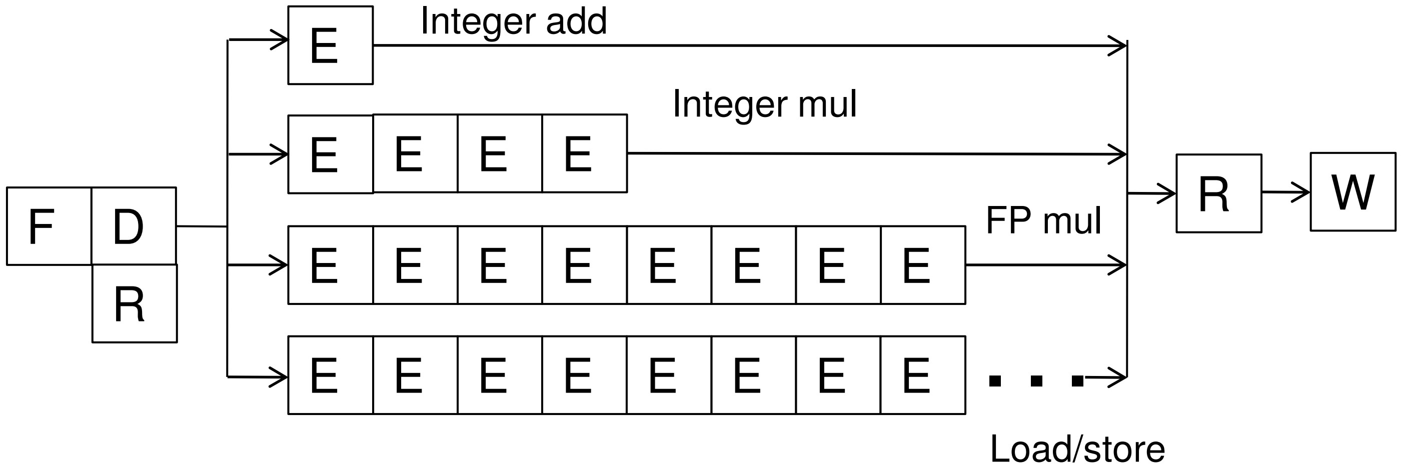
  MUL R3, R4 → R11

  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

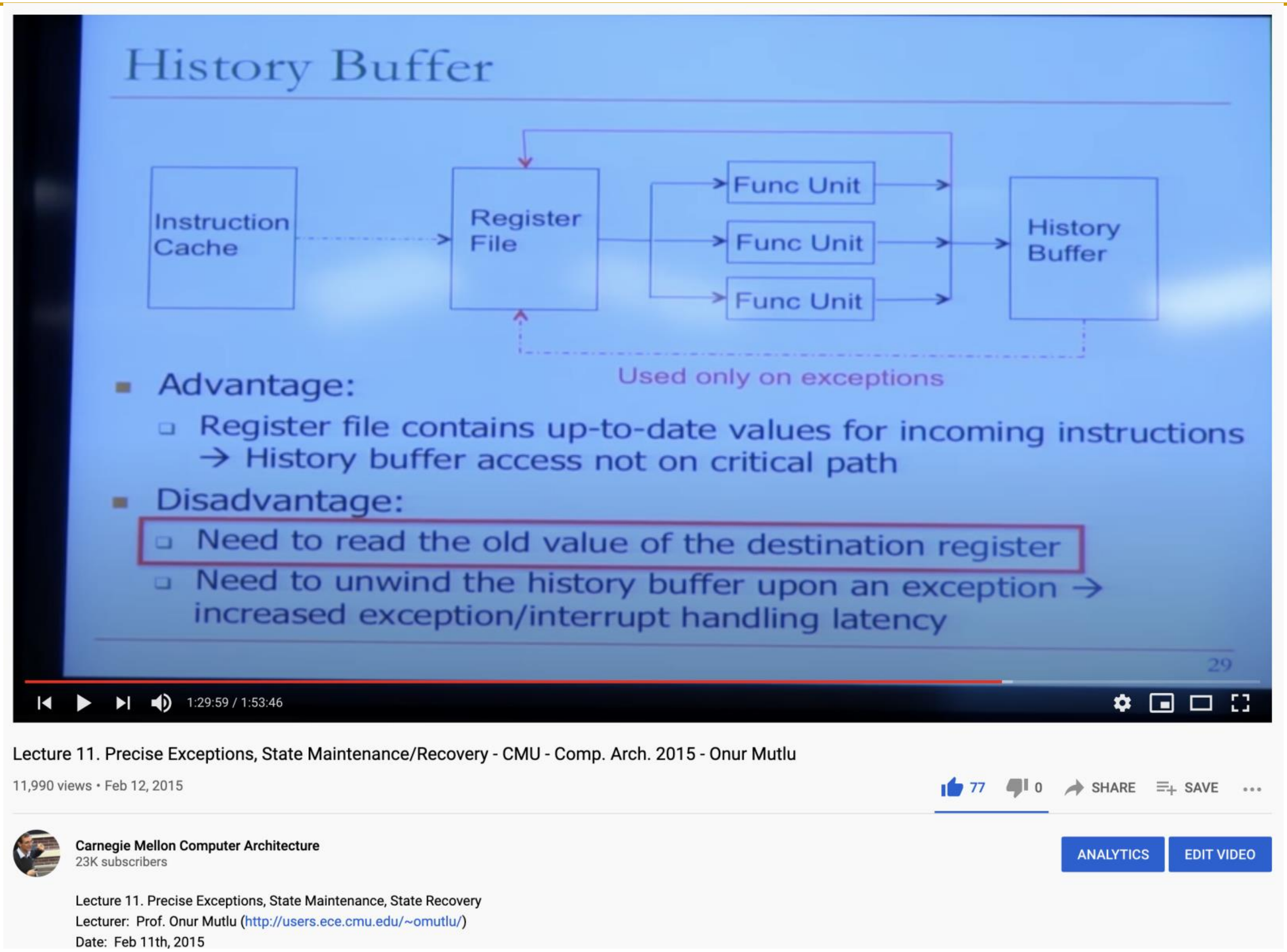
- Advantages
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependences

- 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

- 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

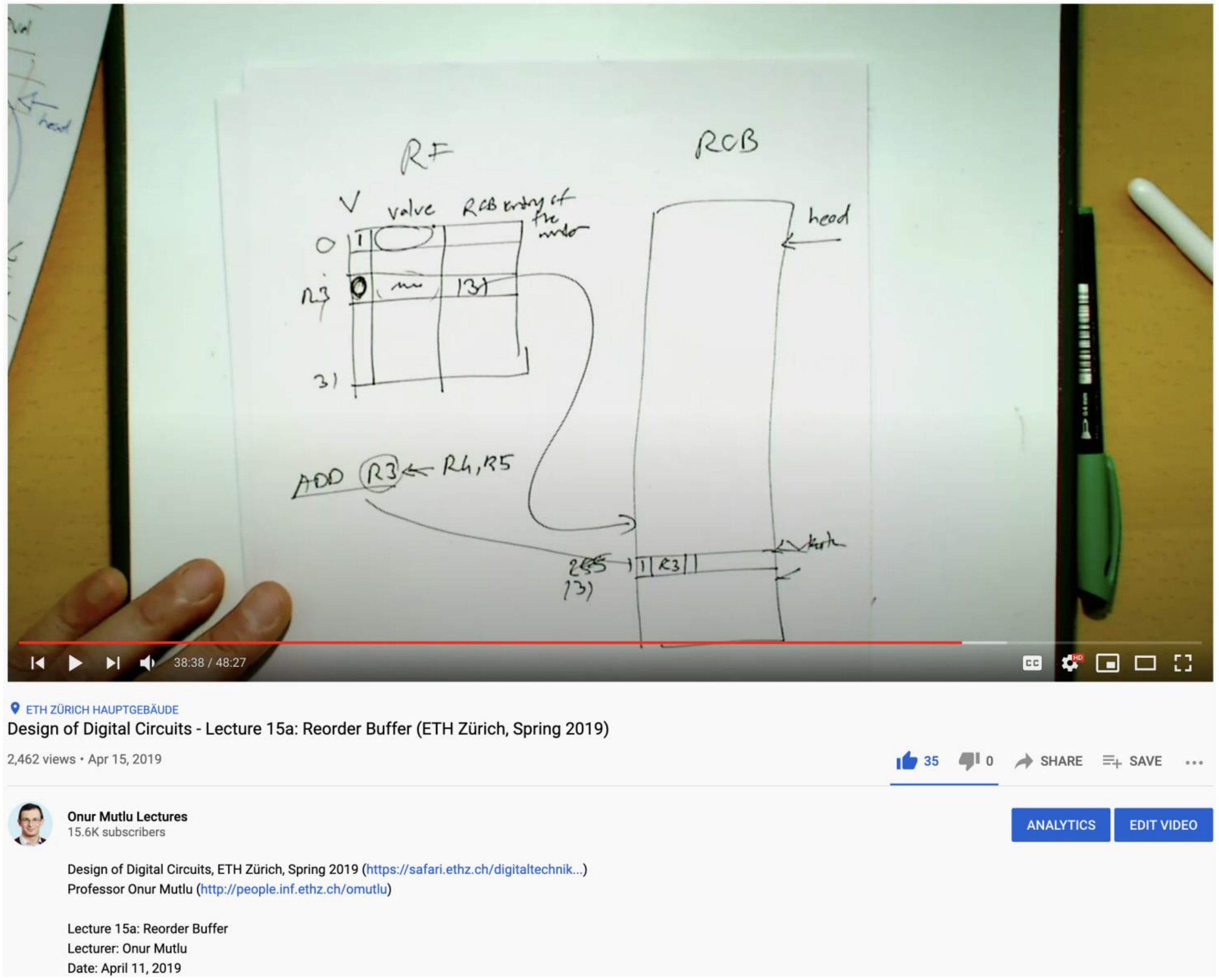
11,990 views • Feb 12, 2015

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

https://www.youtube.com/onurmutlulectures

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

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

https://www.youtube.com/onurmutlulectures

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