

# Design of Digital Circuits

## Lecture 15a: Reorder Buffer

Prof. Onur Mutlu

ETH Zurich

Spring 2019

11 April 2019

# Required Readings

---

## ■ This 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

## ■ Optional

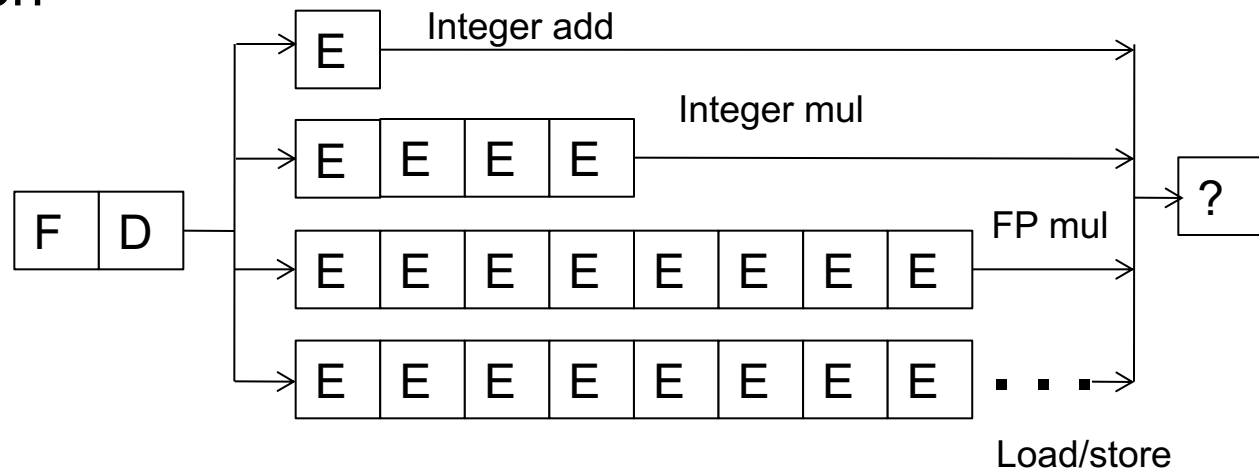
- Kessler, “**The Alpha 21264 Microprocessor**,” IEEE Micro 1999.

## ■ Next Week

- McFarling, “**Combining Branch Predictors**,” DEC WRL Technical Report, 1993.

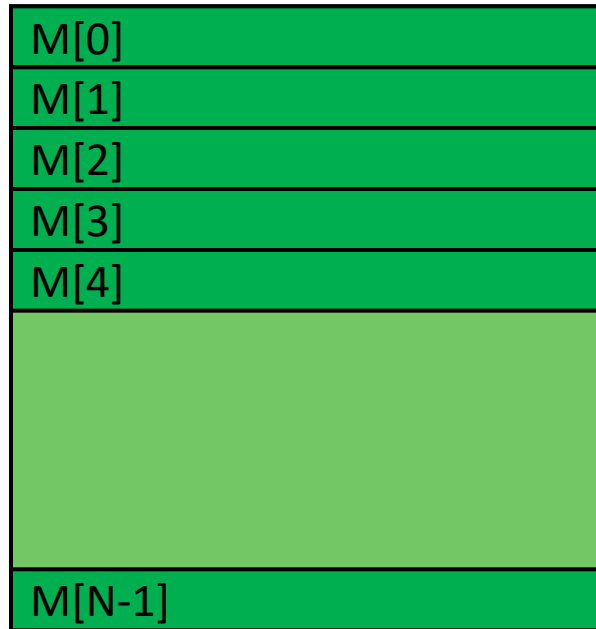
# Recall: Multi-Cycle Execution

- 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



# Recall: Programmer Visible (Architectural) State

---



Memory  
array of storage locations  
indexed by an address



Registers

- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address  
of the current instruction

Instructions (and programs) specify how to transform  
the values of programmer visible state

# Recall: Precise Exceptions/Interrupts

---

- The architectural state should be consistent (precise) when an 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

# Recall: 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)

# Recall: 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

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	
									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

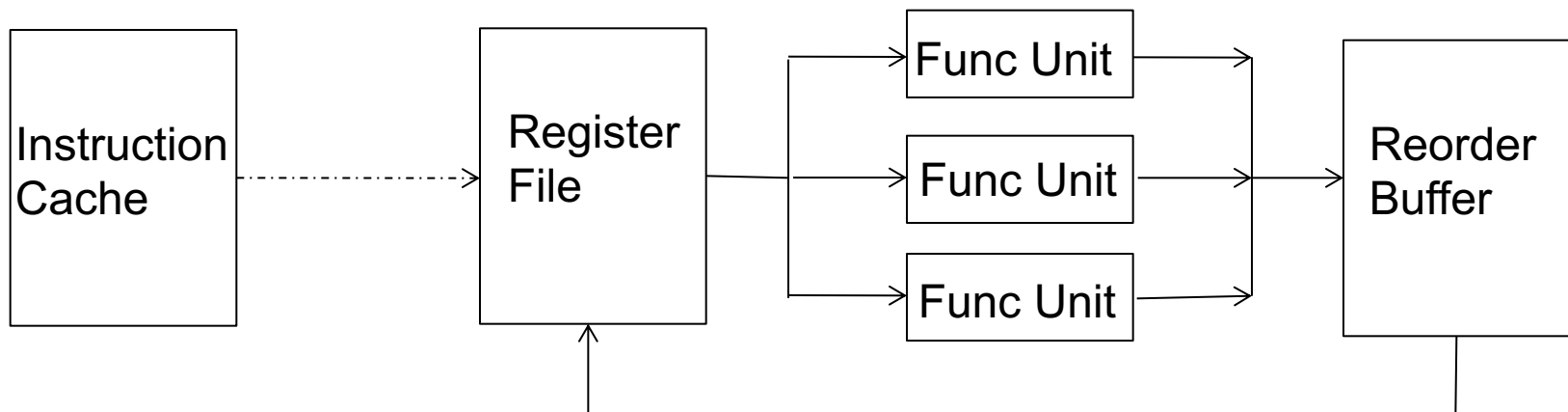
- Suggested reading
  - Smith and Plezskun, “[Implementing Precise Interrupts in Pipelined Processors](#),” IEEE Trans on Computers 1988 and ISCA 1985.



# 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?

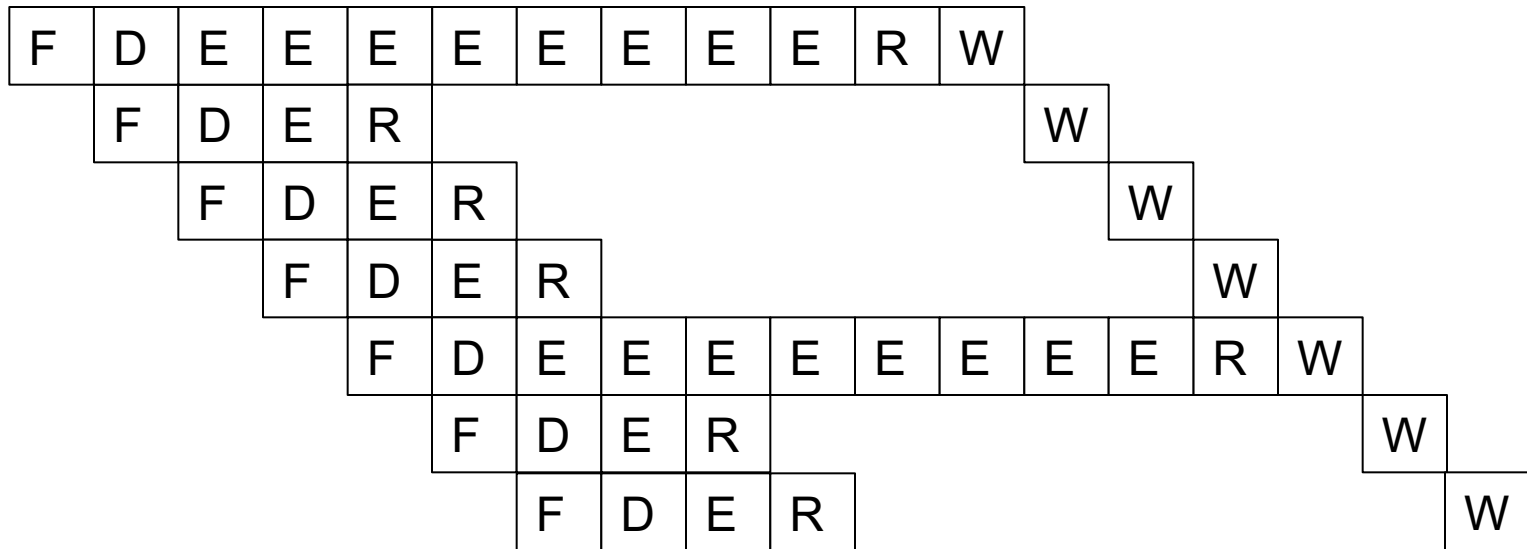
---

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

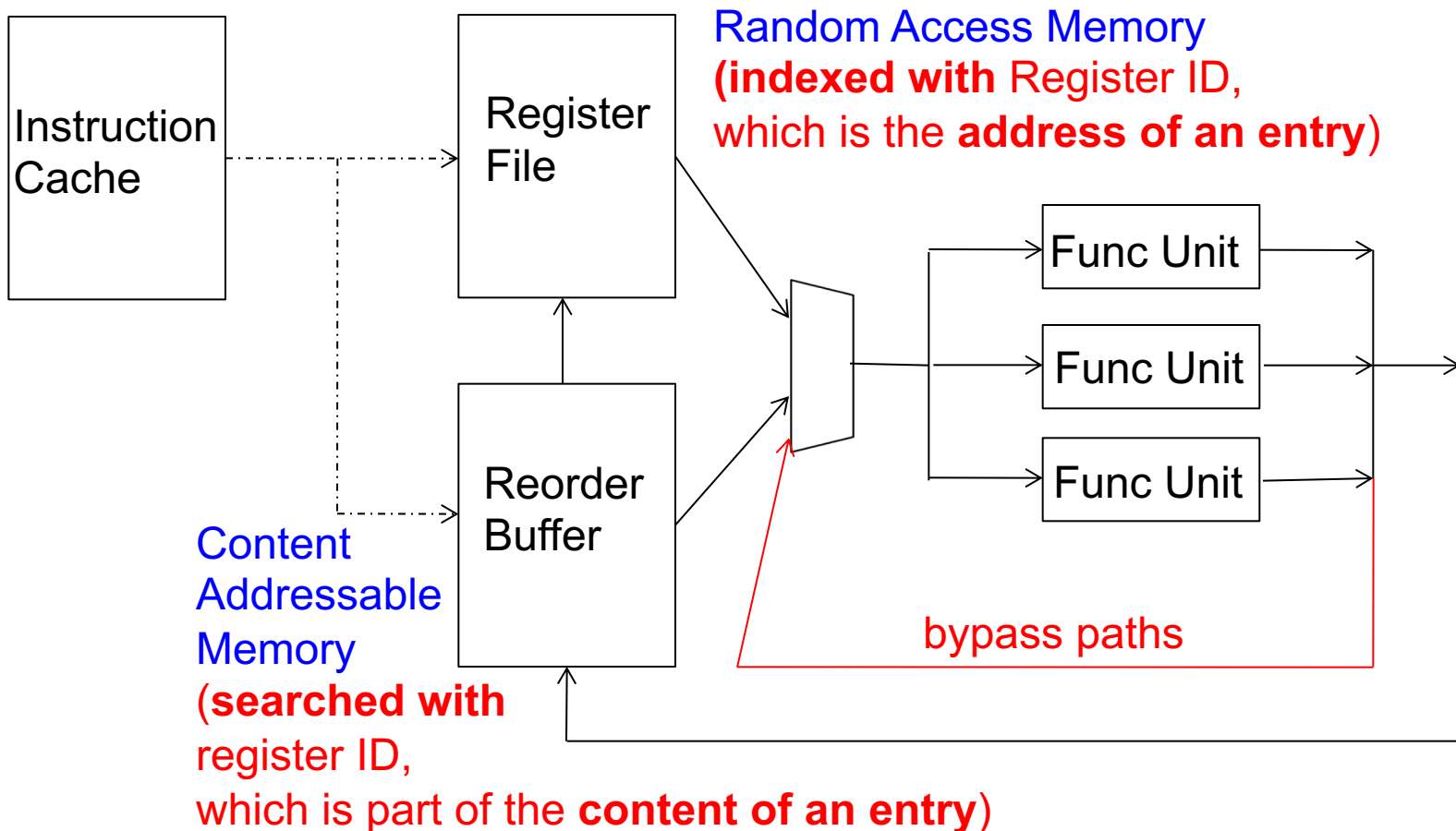
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- What if a later operation needs a value in the reorder buffer?
  - Read reorder buffer in parallel with the register file. **How?**

# Reorder Buffer: How to Access?

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



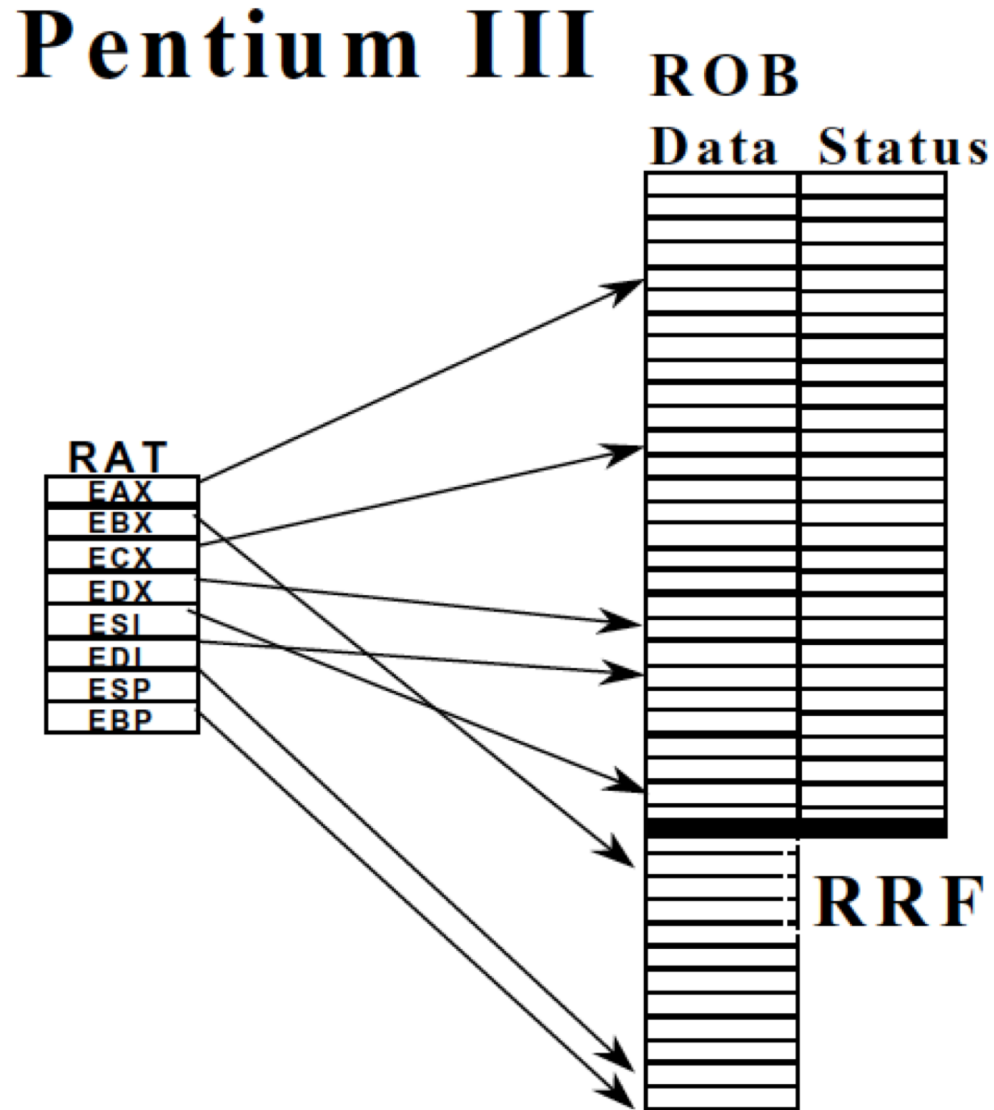
# Simplifying Reorder Buffer Access

---

- 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

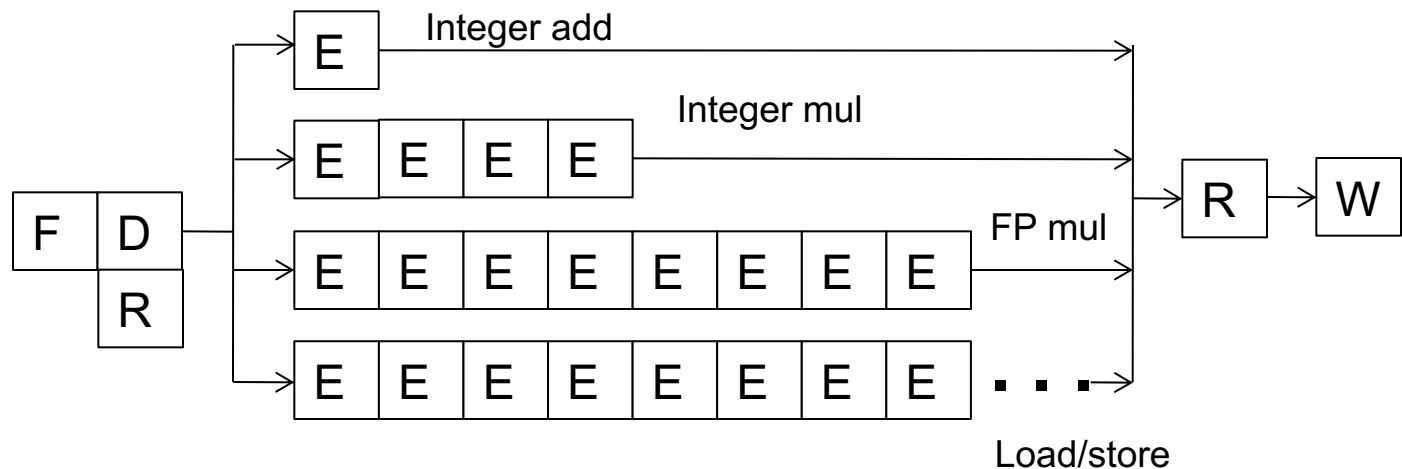
$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



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

# Design of Digital Circuits

## Lecture 15a: Reorder Buffer

Prof. Onur Mutlu

ETH Zurich

Spring 2019

11 April 2019

# Recall: 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