15-740/18-740 Computer Architecture Lecture 8: Precise Exceptions

Prof. Onur Mutlu
Carnegie Mellon University

Fall 2011, 9/28/2011

Reminder: Papers to Read and Review

- Review Set 5 – due October 3 Monday
  - Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Transactions on Computers 1988 (earlier version: ISCA 1985).
  - Sprangle and Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," ISCA 2002.

- Review Set 6 – due October 7 Friday
  - Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of R&D, Jan. 1967.
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proc. IEEE, Dec. 1995.

Review of Last Lecture

- More performance metrics
- Microcoded machines
- Pipelining
- Issues in pipelining

Today

- More issues in pipelining
- Precise exceptions

Causes of Pipeline Stalls

- Data dependencies
- Control dependencies
- Resource contention

Review: Issues in Pipelining: Increased CPI

- Data dependency stall: what if the next ADD is dependent

ADD R3 ← R1, R2
ADD R4 ← R3, R7

F	D	E	W			
	F	D	D	D	E	W

  - Solution: data forwarding. Can this always work?
    - How about memory operations? Cache misses?
    - If data is not available by the time it is needed: STALL
  - What if the pipeline was like this?

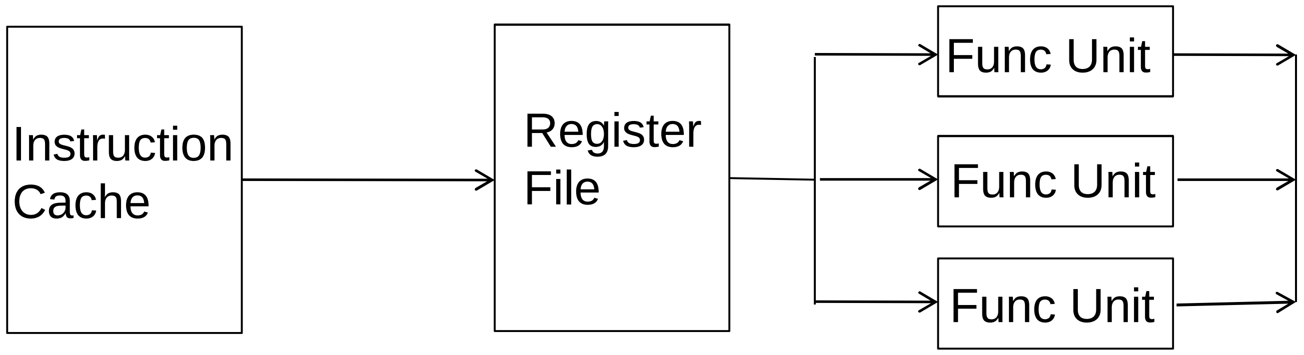
LD R3 ← R2(0)
ADD R4 ← R3, R7

F	D	E	M	W		
	F	D	E	E	M	W

    - R3 cannot be forwarded until read from memory
    - Is there a way to make ADD not stall?

Review: Implementing Stalling

- Hardware based interlocking
  - Common way: scoreboard
  - i.e. valid bit associated with each register in the register file
  - Valid bits also associated with each forwarding/bypass path



Issues in Pipelining: Increased CPI

- Control dependency stall: what to fetch next

BEQ R1, R2, TARGET	F	D	E	W			
		F	F	F	D	E	W

  - Solution: predict which instruction comes next
    - What if prediction is wrong?

  - Another solution: hardware-based fine-grained multithreading
    - Can **tolerate** both data and control dependencies
    - **Read:** James Thornton, “Parallel operation in the Control Data 6600,” AFIPS 1964.
    - **Read:** Burton Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

Issues in Pipelining: Increased CPI

- Resource Contention Stall
  - What if two concurrent operations need the same resource?

```
LD    R1 ← R2(4)
ADD  R2 ← R1, R5
ADD  R6 ← R3, R4
```

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

  - Examples:
    - *Instruction fetch* and *data fetch* both need memory. Solution?
    - *Register read* and *register write* both need the register file
    - A *store instruction* and a *load instruction* both need to access memory. Solution?

Issues in Pipelining: Multi-Cycle Execute

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

FMUL 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								
FMUL R2 ← R5, R6					F	D	E	E	E	E	E	E	E	E	W
ADD R4 ← R5, R6						F	D	E	W						
							F	D	E	W					

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

Handling Exceptions in Pipelining

- Exceptions versus 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
- Priority: process (exception), depends (interrupt)
- Handling Context: process (exception), system (interrupt)

Precise Exceptions/Interrupts

- The architectural state should be consistent 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

Why Do We Want Precise Exceptions?

- Aid software debugging
- Enable (easy) recovery from exceptions, e.g. page faults
- Enable (easily) restartable processes
- Enable 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	F	D	E	E	E	E	E	E	E	E	W						
ADD R4 ← 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

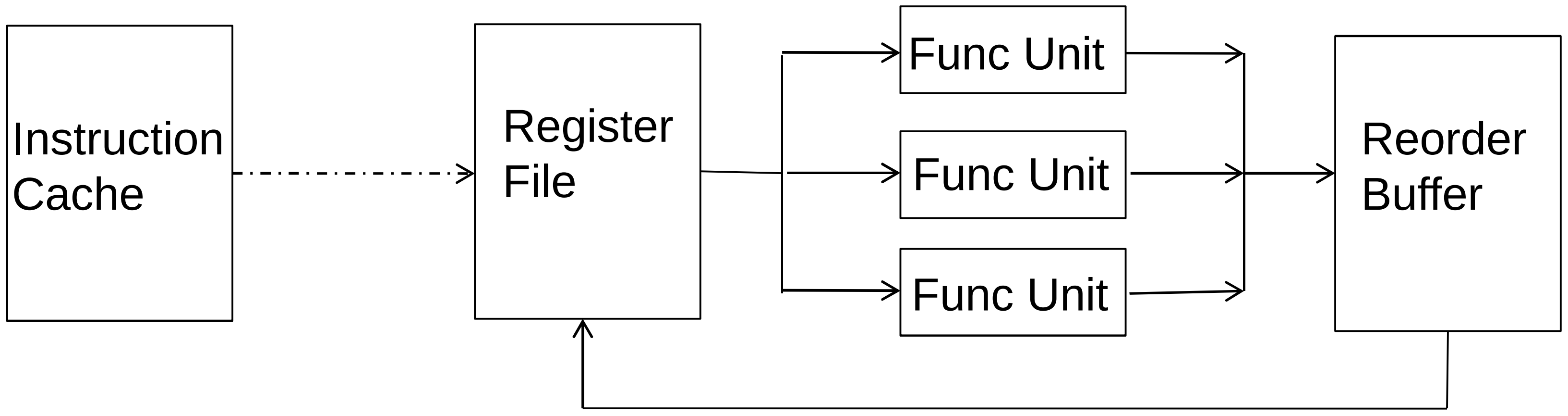
- Downside
  - What about memory operations?
  - Each functional unit takes 500 cycles?

Solutions

- Reorder buffer
- History buffer
- Future register file
- Checkpointing
- Reading
  - 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.

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 an entry in the ROB
- When instruction completes, it writes result into ROB entry
- When instruction oldest in ROB and it has completed, its result moved to reg. file or memory



Reorder Buffer: Independent Operations

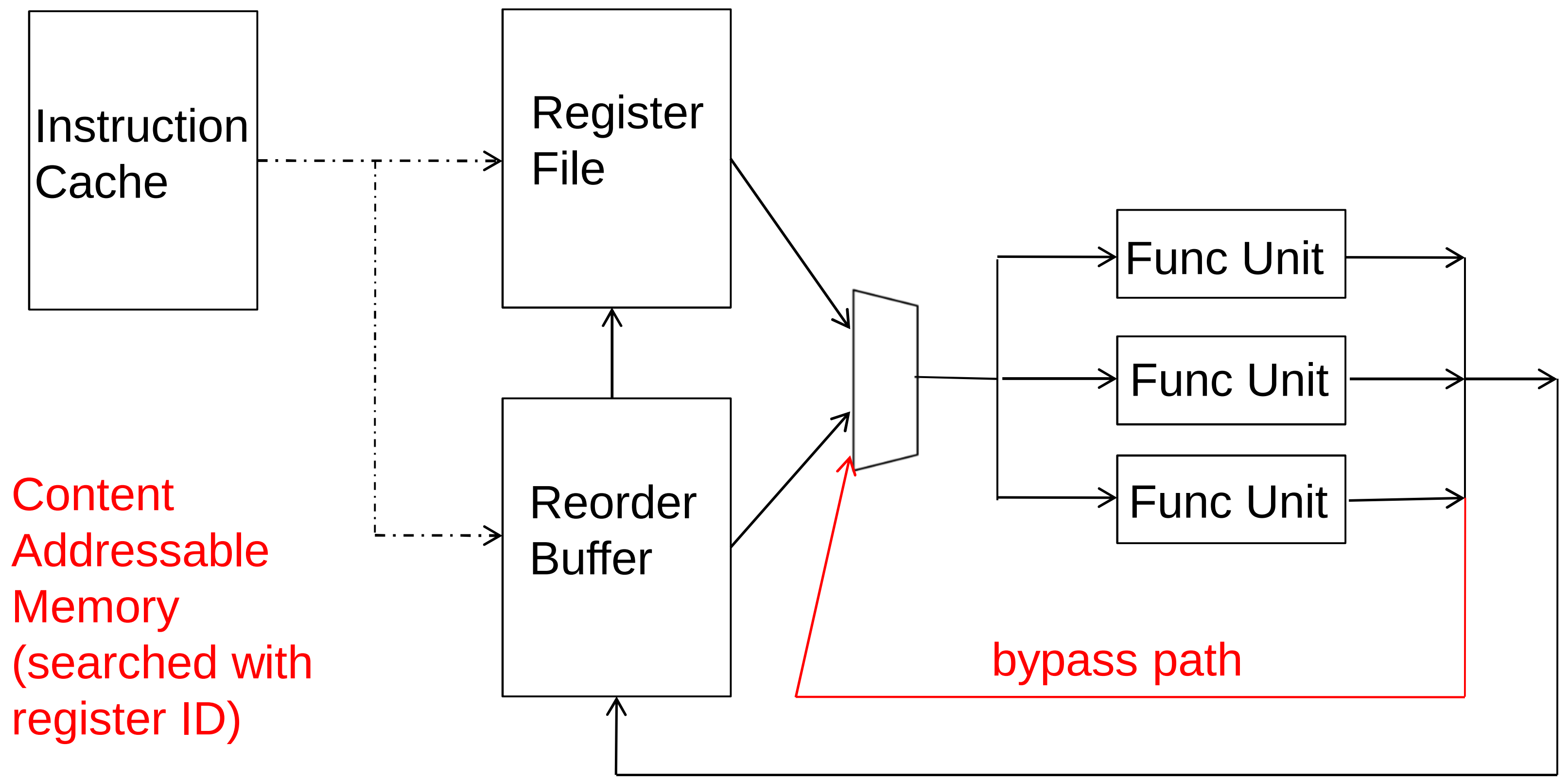
- Results first written to ROB, then to register file at commit time

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 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 paths)



Simplifying Reorder Buffer Access

- Idea: Use indirection
- Access register file first
  - 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
- Access reorder buffer next

- What is in a reorder buffer entry?

V	DestRegID	DestRegVal	StoreAddr	StoreData	BranchTarget	PC/IP	Control/valid bits

  - Can it be simplified further?

What is Wrong with This Picture?

FMUL 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								
FMUL R2 ← R5, R6					F	D	E	E	E	E	E	E	E	E	W
ADD R4 ← R5, R6						F	D	E	W						
							F	D	E	W					

- What is R4's value at the end?
  - The first FMUL's result
  - Output dependency not respected

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