15-740/18-740
Computer Architecture
Lecture 9: More on Precise Exceptions

Prof. Onur Mutlu
Carnegie Mellon University

Fall 2011, 9/30/2011

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

- Causes of stalls in pipelining
- Fine-grained multithreading
- Multi-cycle execute
- Exceptions vs. interrupts
- Precise exceptions
  - Why?
- Reorder buffer
- Intro to register renaming

Today

- More solutions to enable precise exceptions

Solutions to Enable Precise Exceptions

- Reorder buffer

- History buffer

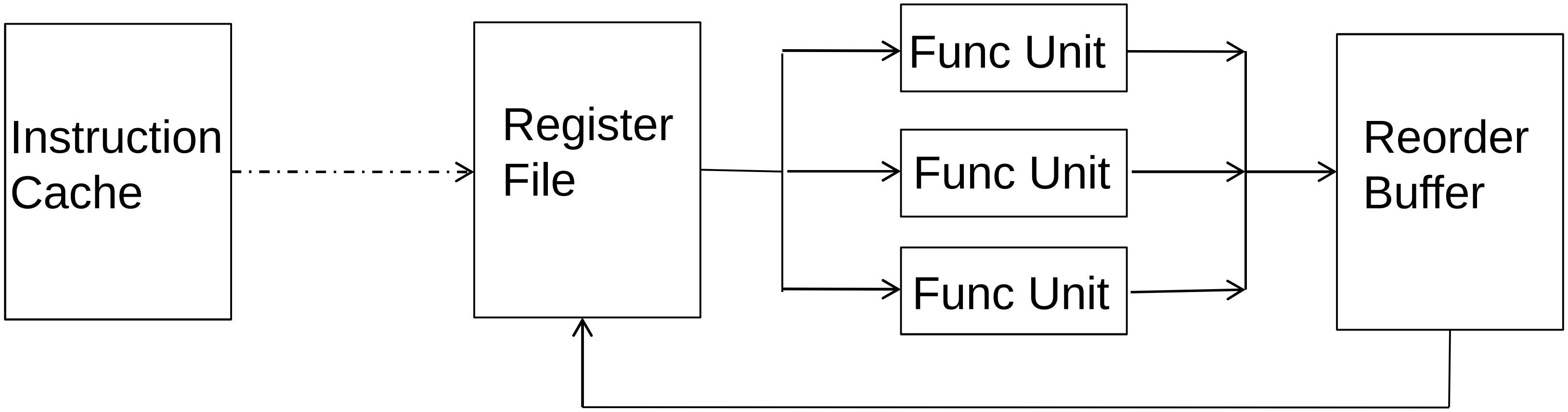
- Future register file

- 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.

Review: 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



Review: Reorder Buffer: Independent Operations

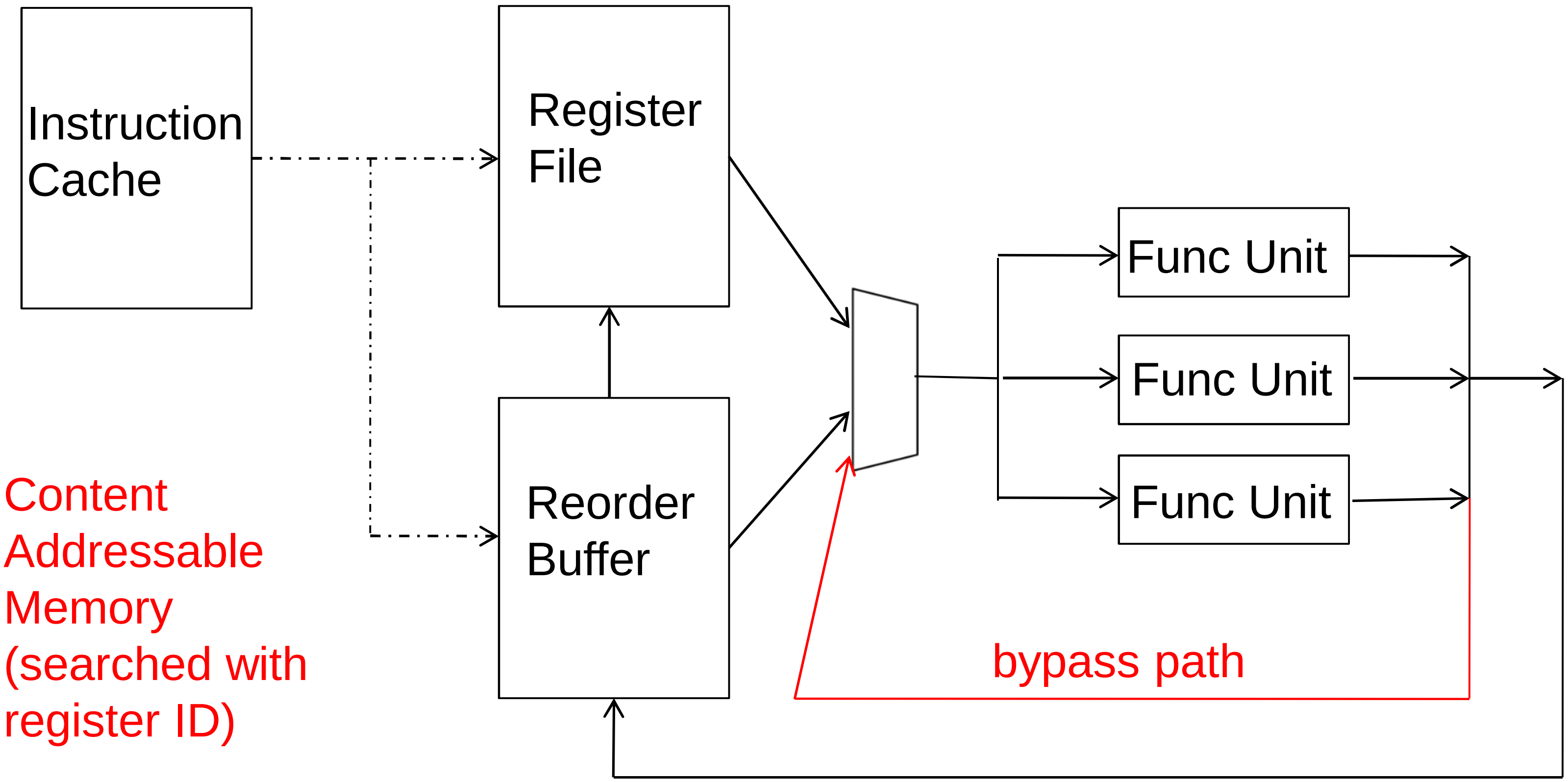
- Results first written to ROB, then to register file at commit time

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?

Review: Reorder Buffer: How to Access?

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



Instruction Cache

Register File

Reorder Buffer

Func Unit

Func Unit

Func Unit

bypass path

Content Addressable Memory (searched with register ID)

Review: 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?

Review: 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

Reorder Buffer Pros and Cons

- Pro
  - Conceptually simple for supporting precise exceptions

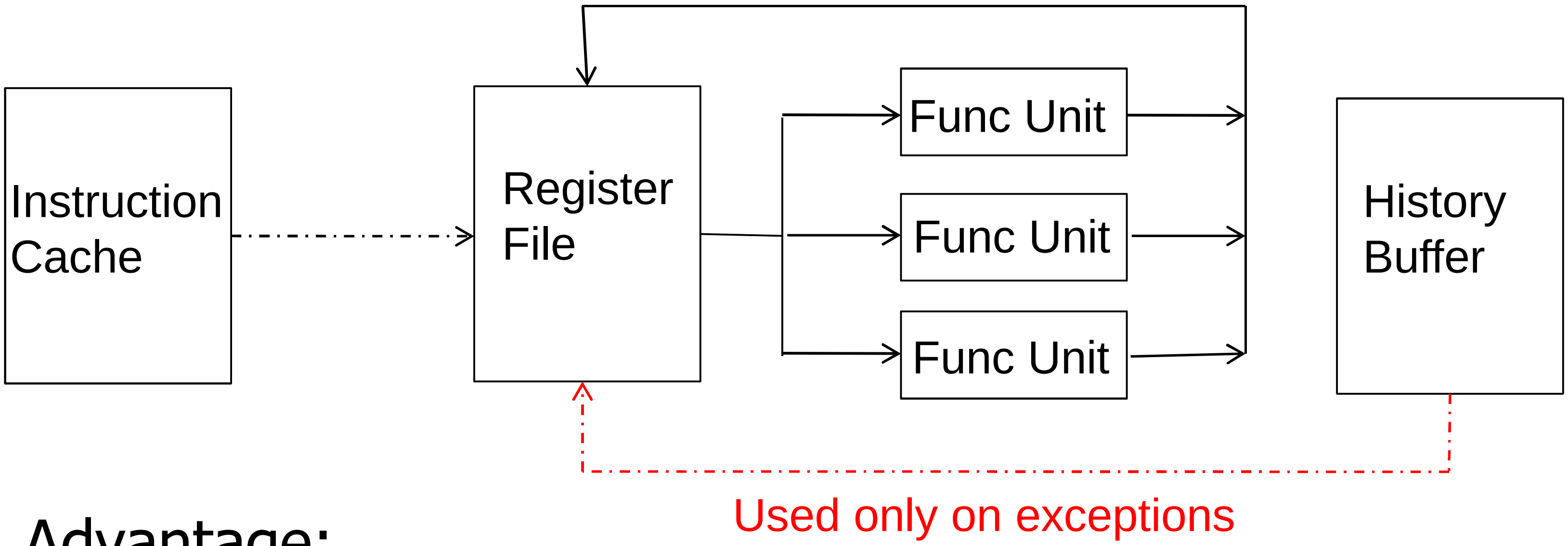
- Con
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection → increased latency and complexity

Solution II: History Buffer (HB)

- Idea: Update architectural state 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. History buffer access not on critical path
- Disadvantage:
  - Need to read the old value of the destination
  - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

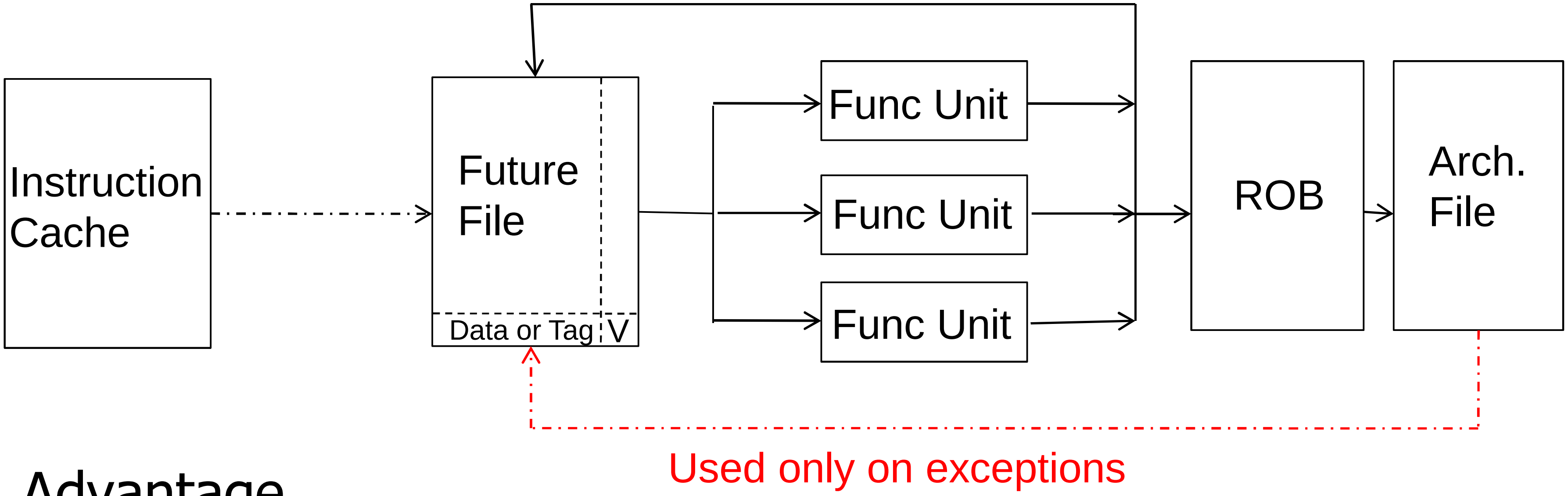
Solution III: Future File (FF)

- Idea: Keep two register files:
  - Arch reg file: Updated in program order for precise exceptions
  - 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

- Architectural file is used for recovery on exceptions

Future File



- Advantage
  - No sequential scanning of history buffer: Upon exception, simply copy arch file to future file
  - No need for extra read of destination value
- Disadvantage
  - Multiple register files + reorder buffer

Checkpointing

- Idea: Periodically checkpoint the register file state. When exception/interrupt occurs, go back to the most recent checkpoint and re-execute instructions one by one to re-generate exception.

- State guaranteed to be precise only at checkpoints.

- Advantage:
  - Per-instruction reorder buffer is not needed
  - Allows for aggressive execution between checkpoints
- Disadvantage:
  - Interrupt latency depends on distance from checkpoint
  - Number of checkpoints?

  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

Summary: Precise 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

- What about branch misprediction recovery?
  - Similar to exception handling except can be initiated before the branch is the oldest instruction
  - All three state recovery methods can be used

- Difference between exceptions and branch mispredictions?
  - Branch mispredictions more common: need fast recovery

Pipelining Issues: Stores

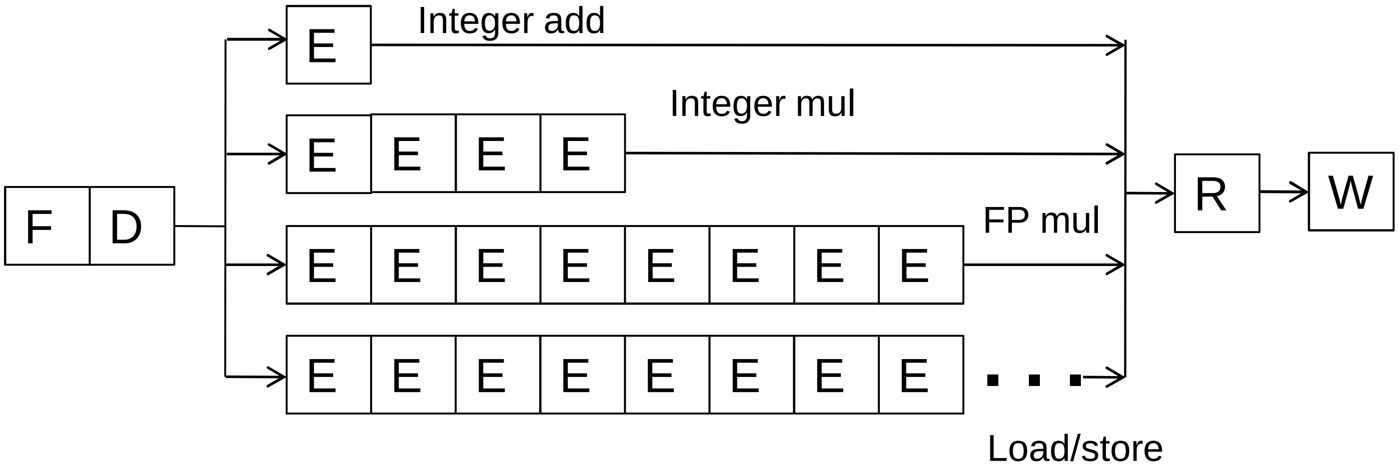
- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. Why?
  - One idea: Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - Store/write buffer: Similar to reorder buffer, but used only 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

Summary: Precise 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

Putting It Together: In-Order Pipeline with Future File

- Decode (D): Access future file, allocate entry in reorder buffer, store buffer, check if instruction can execute, if so **dispatch** instruction
- Execute (E): Instructions can complete out-of-order, store-load dependencies determined
- Completion (R): Write result to reorder/store buffer
- Retirement/Commit (W): Write result to architectural register file or memory
- In-order dispatch/execution, out-of-order completion, in-order retirement



F
D
E
Integer add
E E E E
Integer mul
E E E E E E E E
FP mul
E E E E E E E E
. . .
Load/store
R
W