

18-447

Computer Architecture

Lecture 11: Precise Exceptions, State Maintenance, State Recovery

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 2/11/2015

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...

Reminder: Announcements

- Homework 2 due today (Feb 11)
- Lab 3 online & due next Friday (Feb 20)
 - Pipelined MIPS
 - Competition for high performance
 - You can optimize both cycle time and CPI
 - Document and clearly describe what you do during check-off

Reminder: Readings for Next Few Lectures (I)

- P&H Chapter 4.9-4.11
- 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
- McFarling, “Combining Branch Predictors,” DEC WRL Technical Report, 1993. *HW3 summary paper*
- Kessler, “The Alpha 21264 Microprocessor,” IEEE Micro 1999.

Reminder: Readings for Next Few Lectures (II)

- Smith and Plezskun, “**Implementing Precise Interrupts in Pipelined Processors,**” IEEE Trans on Computers 1988 (earlier version in ISCA 1985). ***HW3 summary paper***

Readings Specifically for Today

- Smith and Plezskun, “Implementing Precise Interrupts in Pipelined Processors,” IEEE Trans on Computers 1988 (earlier version in ISCA 1985). ***HW3 summary paper***
- 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

Review: How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
 - Stall the pipeline until we know the next fetch address
 - Guess the next fetch address (branch prediction)
 - Employ delayed branching (branch delay slot)
 - Do something else (fine-grained multithreading)
 - Eliminate control-flow instructions (predicated execution)
 - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Review of Last Few Lectures

- Control dependence handling in pipelined machines
 - ❑ Delayed branching
 - ❑ Fine-grained multithreading
 - ❑ Branch prediction
 - Compile time (static)
 - ❑ Always NT, Always T, Backward T Forward NT, Profile based
 - Run time (dynamic)
 - ❑ Last time predictor
 - ❑ Hysteresis: 2BC predictor
 - ❑ Global branch correlation → Two-level global predictor
 - ❑ Local branch correlation → Two-level local predictor
 - ❑ Hybrid branch predictors
 - ❑ Predicated execution
 - ❑ Multipath execution
 - ❑ Return address stack & Indirect branch prediction

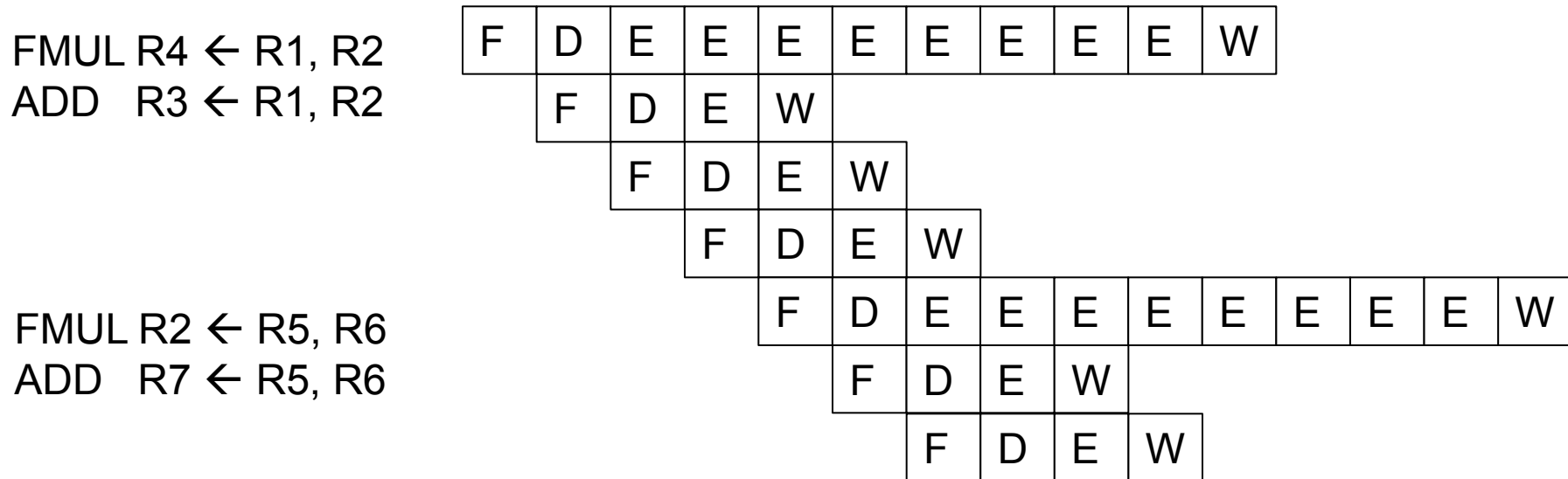
Pipelining and Precise Exceptions: Preserving Sequential Semantics

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 FP MULTiply



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

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 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?

- Semantics of the von Neumann model ISA specifies it
 - Remember von Neumann vs. Dataflow
- Aids software debugging
- Enables (easy) recovery from exceptions, e.g. page faults
- Enables (easily) restartable processes
- Enables 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 \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	

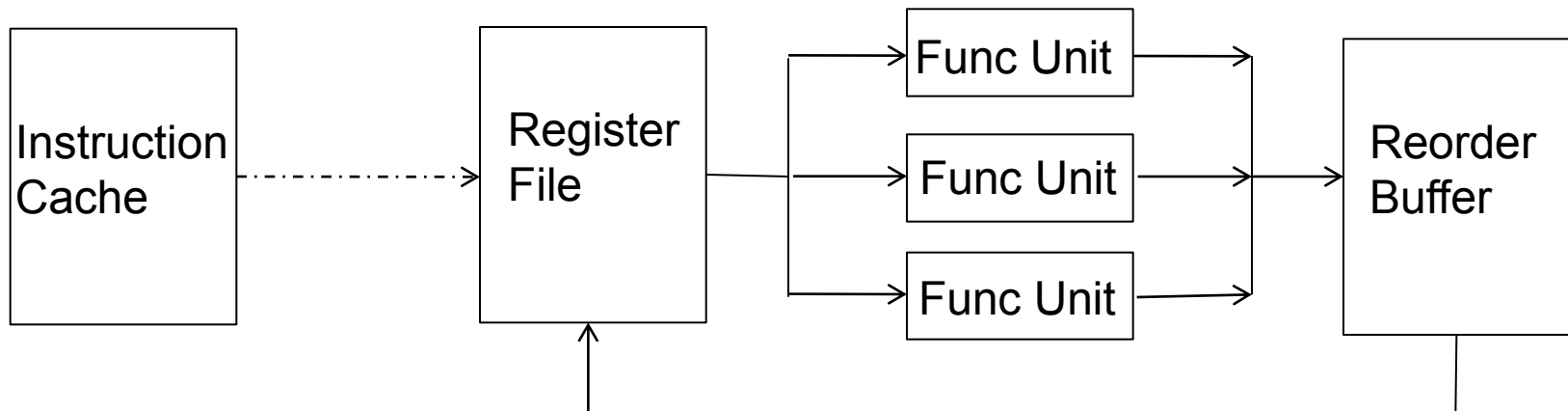
- Downside
 - ❑ Worst-case instruction latency determines all instructions' latency
 - ❑ What about memory operations?
 - ❑ Each functional unit takes worst-case number of cycles?

Solutions

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

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 without exceptions, its result moved to reg. file or memory



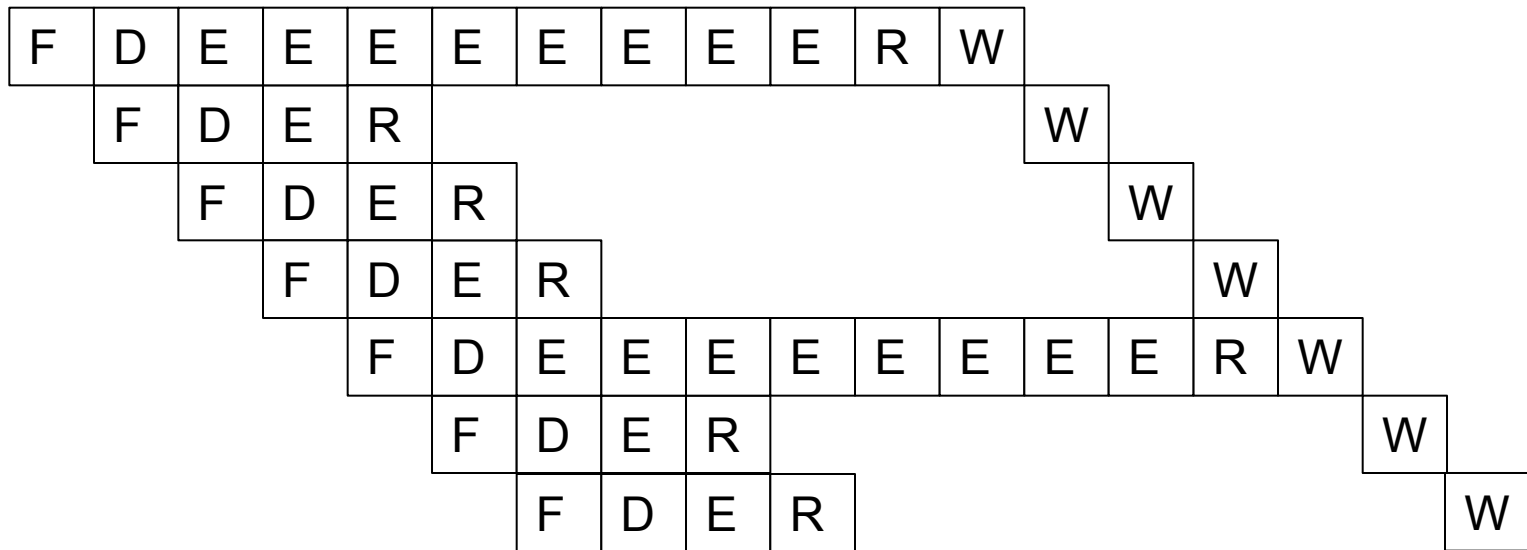
What's in a ROB Entry?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exc?
---	-----------	------------	-----------	-----------	----	---	------

- Need valid bits to keep track of readiness of the result(s)

Reorder Buffer: Independent Operations

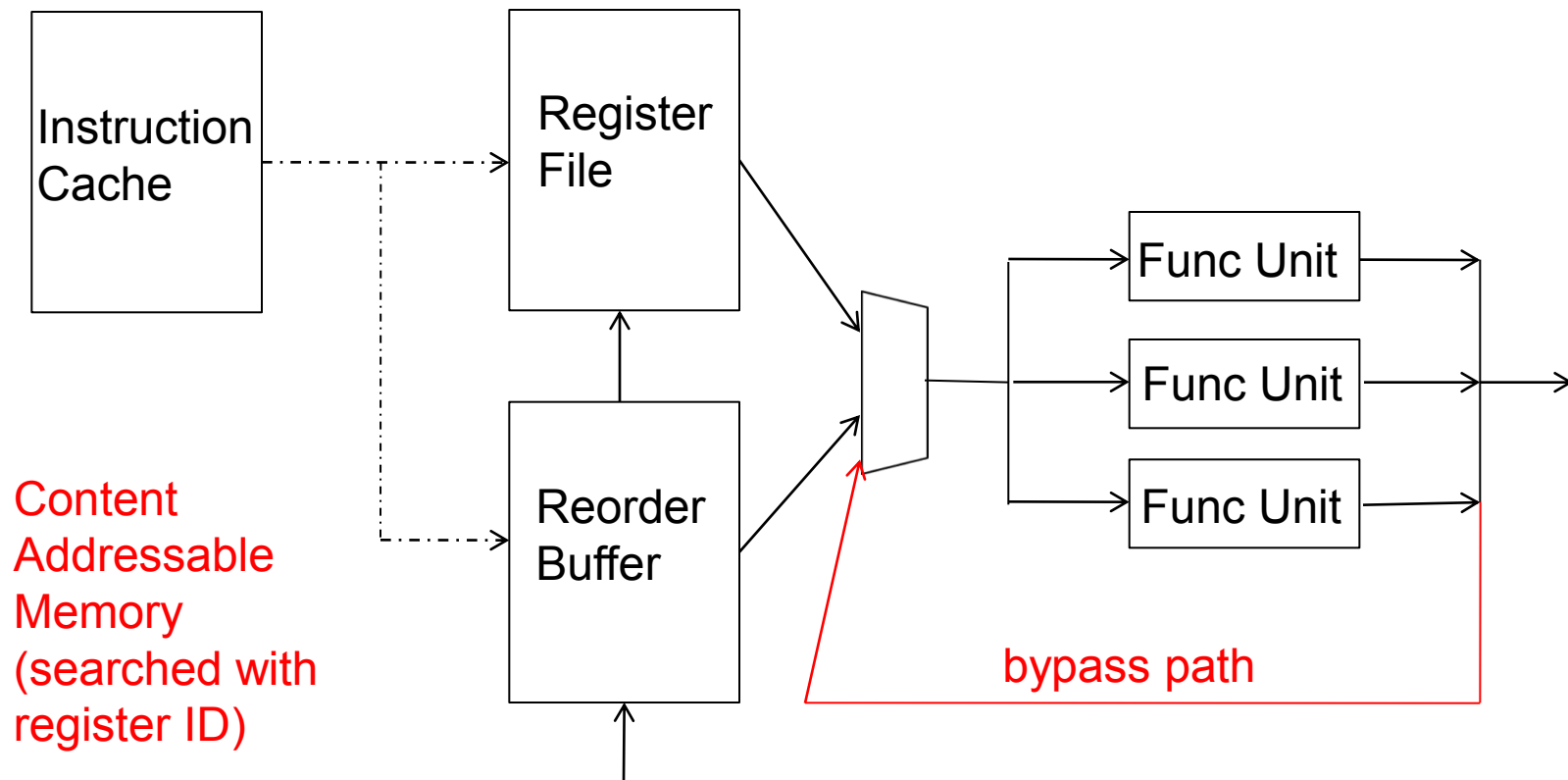
- Results first written to ROB, then to register file at commit time



- 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?

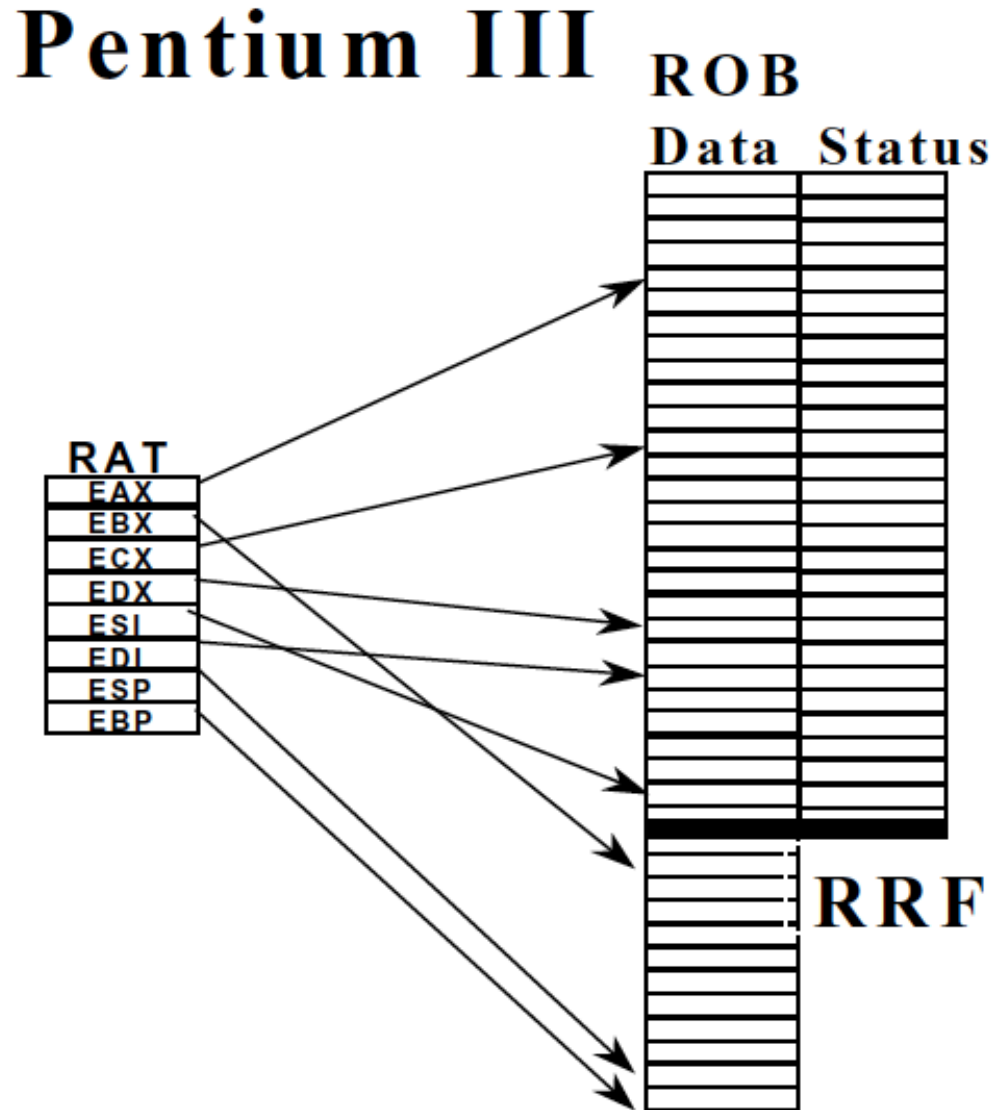
- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



Simplifying Reorder Buffer Access

- Idea: Use indirection
- Access register file first
 - 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

Renaming Example

- Assume
 - Register file has pointers to reorder buffer 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

Reorder Buffer Storage Cost

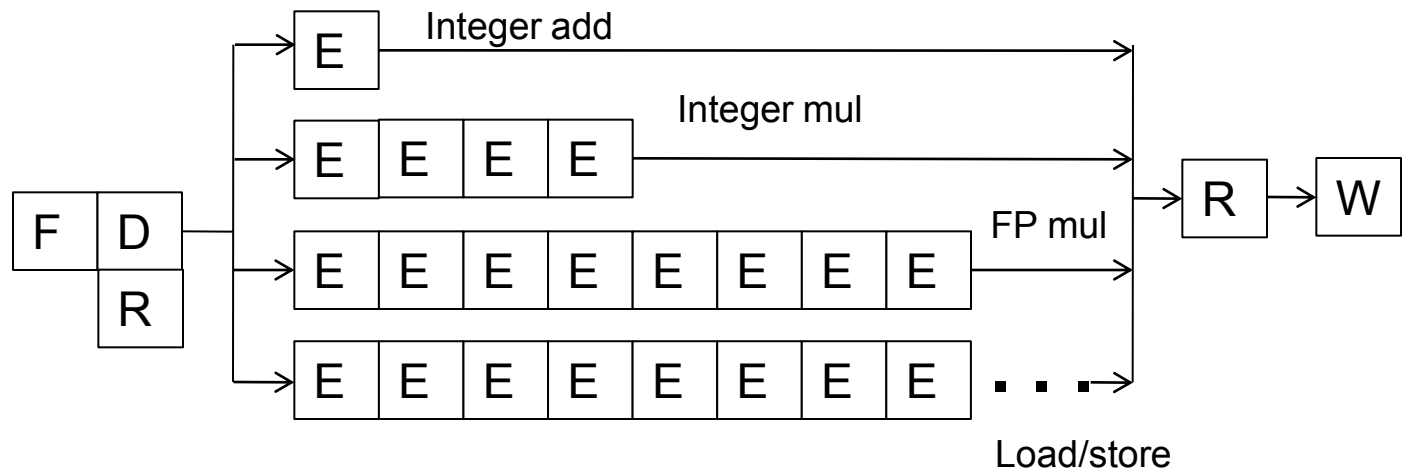
- Idea: Reduce reorder buffer entry storage by specializing for instruction types

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC/IP	Control/val id bits	Exc?
---	-----------	------------	-----------	-----------	-------	------------------------	------

- Do all instructions need all fields?
- Can you reuse some fields between instructions?
- Can you implement separate buffers per instruction type?
 - LD, ST, BR, ALU

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 dependencies

■ 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

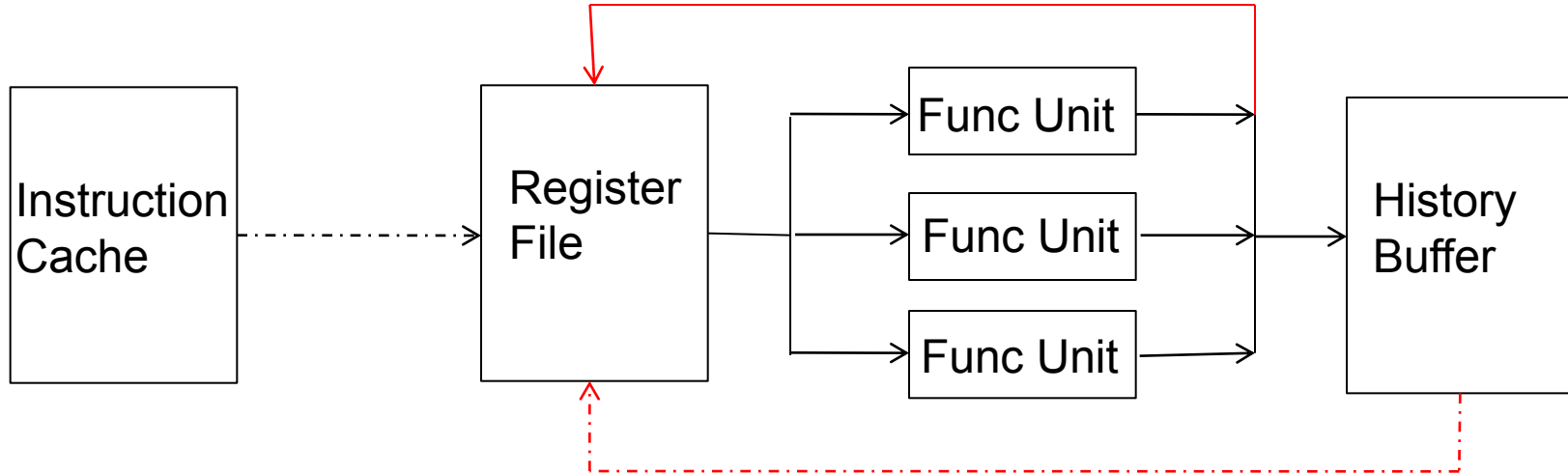
■ Other solutions aim to eliminate the disadvantages

- ❑ 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



Used only on exceptions

■ 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

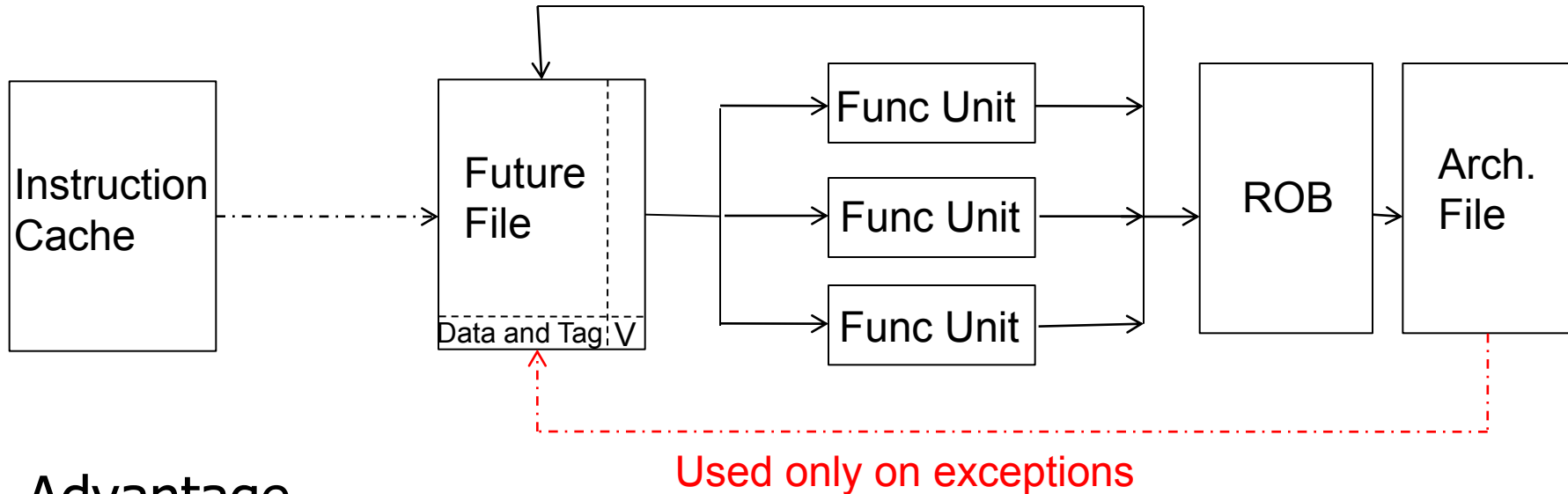
Comparison of Two Approaches

- 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

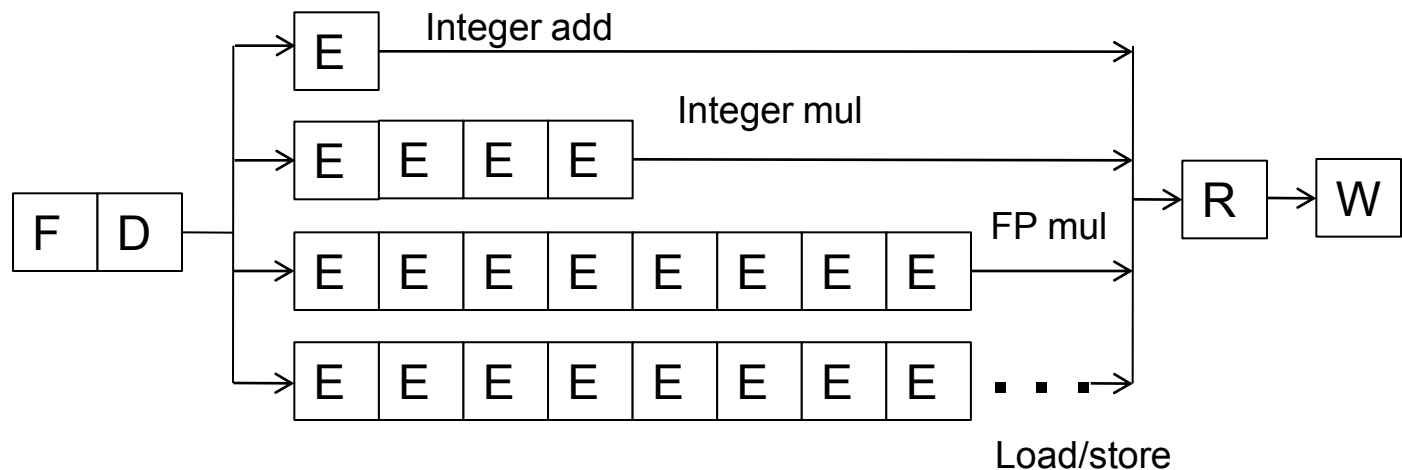
- ❑ No need to read the new values from the ROB (no CAM or indirection) or the old value of destination register

■ 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**

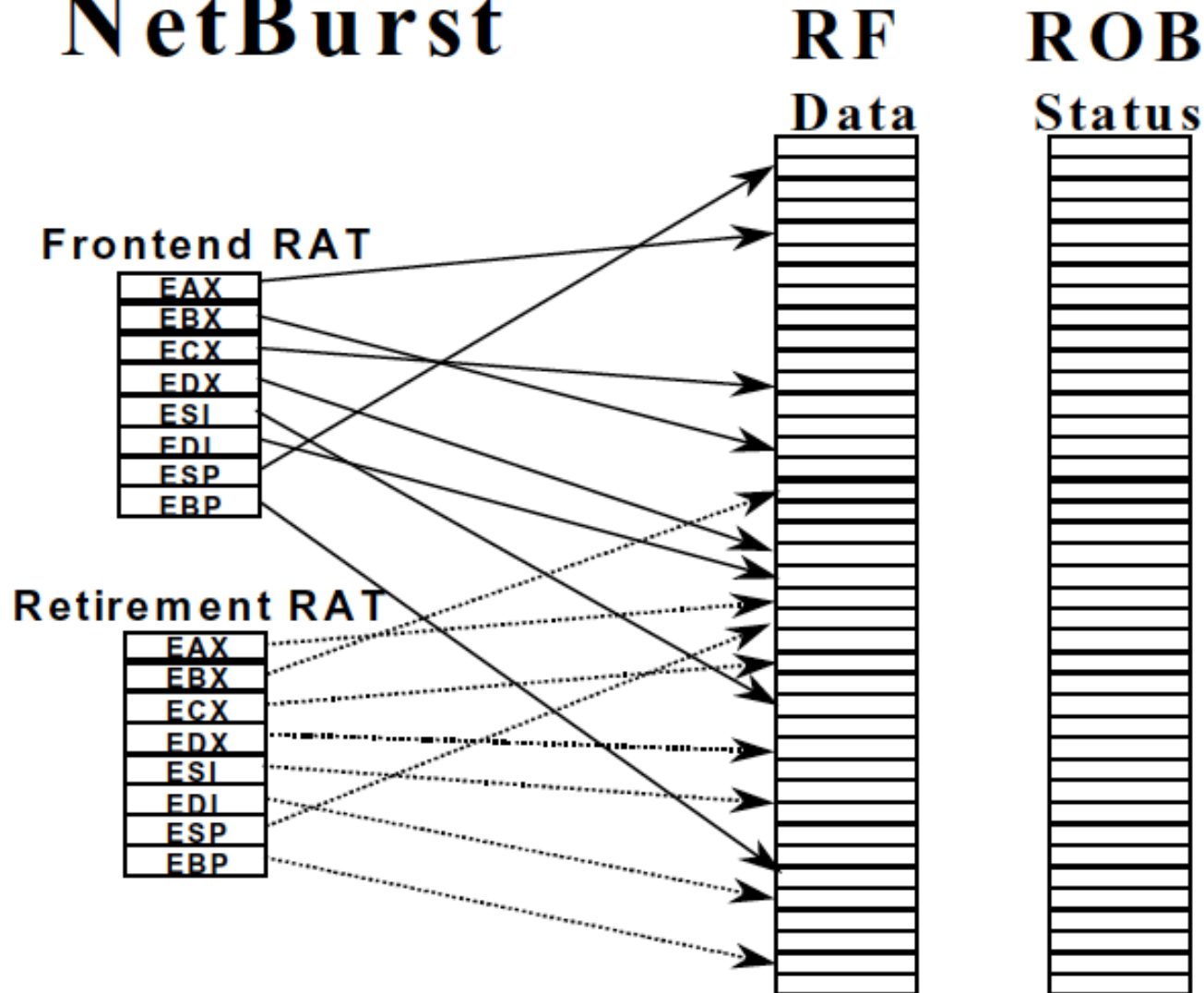


Can We Reduce the Overhead of Two Register Files?

- 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

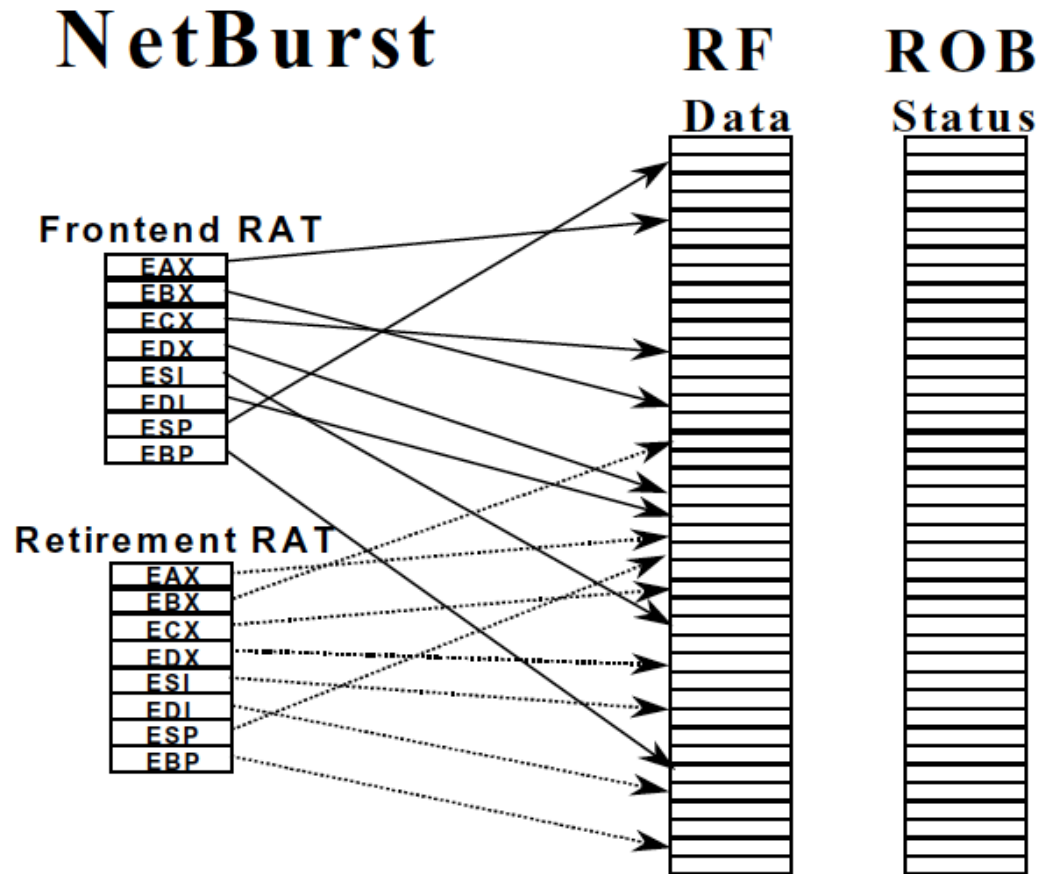
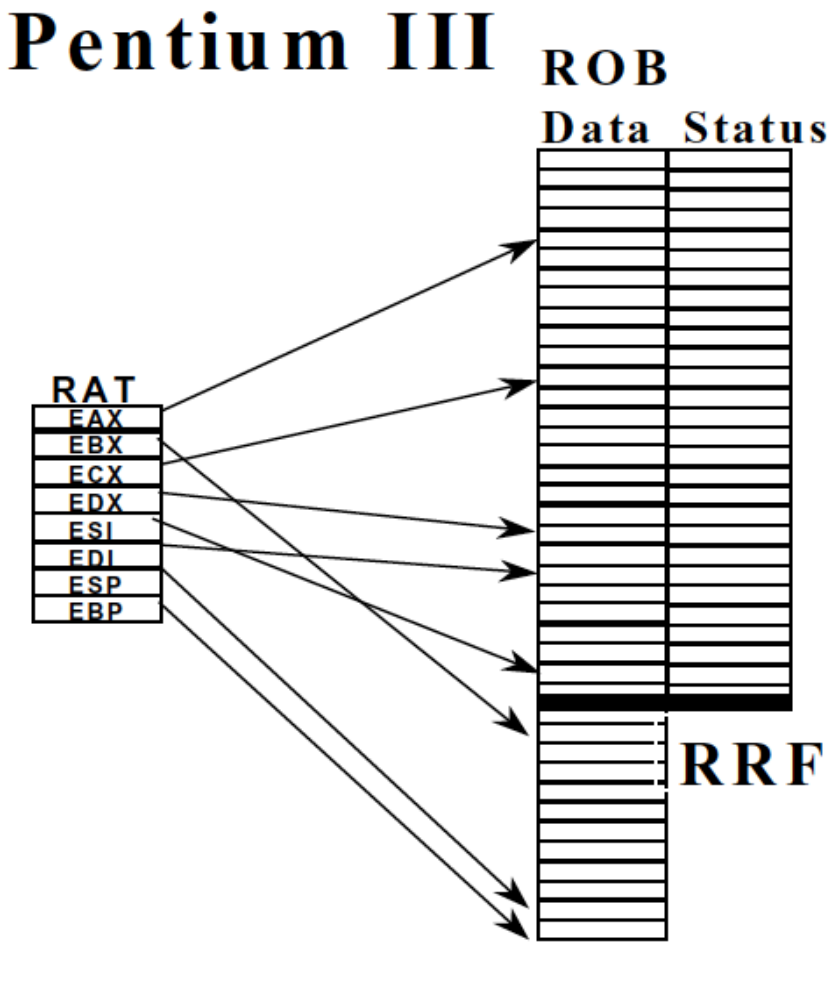


Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Many modern processors are similar:

- MIPS R10K
- Alpha 21264

Reorder Buffer vs. Future Map Comparison



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)

We did not cover the following slides in lecture.
These are for your preparation for the next lecture.

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!