Agenda for Today & Next Few Lectures

- Prior to last week: Microarchitecture Fundamentals
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures

- Last week & today: Pipelining
  - Pipelining
  - Pipelined Processor Design
    - Control & Data Dependence Handling
    - Precise Exceptions: State Maintenance & Recovery

- Today & soon: Out-of-Order Execution
  - Out-of-Order Execution
  - Issues in OoO Execution: Load-Store Handling, ...
Readings

- **This week**
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.7, 7.8.1-7.8.3

- **Today & Next week**
  - Out-of-order execution
    - H&H, Chapter 7.8-7.9
    - More advanced pipelining
    - Interrupt and exception handling
    - Out-of-order and superscalar execution concepts
Pipelining and Precise Exceptions: Preserving Sequential Semantics
Multi-Cycle Execution

- Not all instructions take the same amount of time for in the “execute stage” of the pipeline.

- 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Registers</th>
<th>Instruction</th>
<th>Operation</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV</td>
<td>R4 ← R1, R2</td>
<td></td>
<td>ADD</td>
<td>R3 ← R1, R2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>R2 ← R5, R6</td>
<td></td>
<td>ADD</td>
<td>R7 ← R5, R6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What is wrong with this picture in a Von Neumann architecture?
  - Sequential semantics of the ISA NOT preserved!
  - What if DIV incurs an exception? (e.g., DIV by zero)
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 and Interrupts: Examples

- **Exception** examples
  - Divide by zero
  - Overflow
  - Undefined opcode
  - General protection (or access protection)
  - Page fault
  - ...

- **Interrupt** examples
  - I/O device needing service (e.g., keyboard input, video input)
  - (Periodic) system timer expiration
  - Power failure
  - Machine check
  - ...
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
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
6.1 INTERRUPT AND EXCEPTION OVERVIEW

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT n instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-check architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine-check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor’s interrupt and exception-handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

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
    \(\rightarrow\) before fetching the next instruction

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture
Precise Exceptions in Multi-Cycle Datapath

EPC register: Holds the exception causing PC
Cause register: Holds the cause of the exception
Exception Handler starts at address 0x80000180

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture
Precise Exceptions in Multi-Cycle FSM

- Supports
  - Overflow
  - Undefined instruction

- mfc0 instruction is used to copy the exception cause into a general-purpose register

See H&H Section 7.7 for a treatment of exceptions in multi-cycle microarchitecture
Precise Exceptions in Multi-Cycle Datapath

In summary, an exception causes the processor to jump to the exception handler. The exception handler saves registers on the stack, then uses mfc0 to look at the cause and respond accordingly. When the handler is finished, it restores the registers from the stack, copies the return address from EPC to $k0 using mfc0, and returns using jr $k0.

Figure 7.63 Datapath supporting mfc0
Multi-Cycle Execute: More Complications

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

```
| DIV     | R4 ← R1, R2 |
| ADD     | R3 ← R1, R2 |
```

```
| DIV     | R2 ← R5, R6 |
| ADD     | R7 ← R5, R6 |
```

- What is wrong with this picture in a Von Neumann architecture?
  - Sequential semantics of the ISA NOT preserved!
  - What if DIV incurs an exception? (e.g., DIV by zero)
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 video from Spring 2015
Also see backup slides

Suggested reading
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.

ROB is implemented as a circular queue in hardware.
A hardware structure that keeps information about all instructions that are **decoded** but **not yet retired**/committed.

ROB is implemented as a circular queue in hardware.
What’s in a ROB Entry?

- 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

<table>
<thead>
<tr>
<th>V</th>
<th>DestRegID</th>
<th>DestRegVal</th>
<th>StoreAddr</th>
<th>StoreData</th>
<th>PC</th>
<th>Valid bits for reg/data + control bits</th>
<th>Exception?</th>
</tr>
</thead>
</table>

V: Valid bit
Reorder Buffer: Independent Operations

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

What if a later instruction needs a value in the reorder buffer?
  - One option: stall the operation \(\rightarrow\) 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 → Reorder Buffer

Random Access Memory (indexed with Register ID, which is the address of an entry)

Content Addressable Memory (searched with register ID, which is part of the content of an entry)

bypass paths
**Reorder Buffer Example**

**Register File (RF)**

<table>
<thead>
<tr>
<th>Value Valid?</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
</tr>
</tbody>
</table>

Initially: all registers are valid in RF & ROB is empty

**Simulate:**
- MUL R1, R2 → R3
- MUL R3, R4 → R11
- ADD R5, R6 → R3
- ADD R3, R8 → R12

**Reorder Buffer (ROB)**

<table>
<thead>
<tr>
<th>Entry Valid?</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Oldest instruction

Youngest instruction
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 Example

Register File (RF)

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Value Valid?
Value (pointer to ROB entry)

Initially: all registers are valid in RF & ROB is empty

Simulate:
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer (ROB)

Entry 0
Entry 1
Entry 2
Entry 8
Entry 13
Entry 14
Entry 15

Entry Valid?
Dest reg ID
Dest reg value
Dest reg written?

Oldest instruction
Youngest instruction

Initially:
all registers are valid in RF & ROB is empty

Simulate:
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12
A Register Alias Table (RAT) points to where each register’s current value is (or will be)
Intel Pentium Pro (1997)
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 $\rightarrow$ ROB entry ID
  - Architectural register ID $\rightarrow$ 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

**Flow dependence**

\[ r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{Read-after-Write (RAW)} \]

\[ r_5 \leftarrow r_3 \text{ op } r_4 \]

**Anti dependence**

\[ r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{Write-after-Read (WAR)} \]

\[ r_1 \leftarrow r_4 \text{ op } r_5 \]

**Output dependence**

\[ r_3 \leftarrow r_1 \text{ op } r_2 \quad \text{Write-after-Write (WAW)} \]

\[ r_5 \leftarrow r_3 \text{ op } r_4 \]

\[ r_3 \leftarrow r_6 \text{ op } r_7 \]
Register Renaming Example (On Your Own)

- 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 R3, R8 → R12
Reorder Buffer Example

Register File (RF)

<table>
<thead>
<tr>
<th>Value Valid?</th>
<th>Value</th>
<th>Tag (pointer to ROB entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initially: all registers are valid in RF & ROB is empty

Simulate:
- LD R0(0) → R3
- LD R3, R1 → R10
- MUL R1, R2 → R3
- MUL R3, R4 → R11
- ADD R5, R6 → R3
- ADD R3, R8 → R12

Reorder Buffer (ROB)

<table>
<thead>
<tr>
<th>Entry 0</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry 1</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry 2</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry 8</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry 13</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry 14</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry 15</th>
<th>Dest reg ID</th>
<th>Dest reg value</th>
<th>Dest reg written?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Oldest instruction

Youngest instruction
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**

ROB is implemented as a circular queue in hardware
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 $\rightarrow$ increased latency and complexity

- **Other solutions aim to eliminate the disadvantages**
  - **History buffer**
  - **Future file**
  - **Checkpointing**

*We will not cover these*

*See suggested lecture video from Spring 2015*

*Also see backup slides*
More on State Maintenance & Precise Exceptions

Lecture 11. Precise Exceptions, State Maintenance/Recovery - CMU - Comp. Arch. 2015 - Onur Mutlu

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

https://www.youtube.com/watch?v=nMfbtzWizDA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=13
More on State Maintenance & Precise Exceptions

https://www.youtube.com/watch?v=upJPVXEuqIQ&list=PL5Q2soXY2Zi-iBn_sw_B63HtdbTNmphLc&index=18
More on State Maintenance & Precise Exceptions

https://www.youtube.com/watch?v=9yo3yhUijQs&list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&index=17
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](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=PL5Q2soXY2Zi8J58xLKBNFQFHR03GrXxA9&index=17](https://www.youtube.com/watch?v=9yo3yhUijQs&list=PL5Q2soXY2Zi8J58xLKBNFQFHR03GrXxA9&index=17)

- **Digital Design & Computer Architecture, Spring 2021, Lecture 15a**
  - Precise Exceptions (ETH Zurich, Spring 2021)
  - [https://www.youtube.com/watch?v=upJPVXEuqIQ&list=PL5Q2soXY2Zi-iBn_sw_B63HtdbTNmphLc&index=18](https://www.youtube.com/watch?v=upJPVXEuqIQ&list=PL5Q2soXY2Zi-iBn_sw_B63HtdbTNmphLc&index=18)

[https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures)
Suggested Readings for the Interested


- Backup Slides
Backup Slides on Precise Exceptions
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 $\rightarrow$ 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

- **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

Used only on exceptions
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


Many modern processors are similar:
- MIPS R10K
- Alpha 21264
Reorder Buffer vs. Future Map Comparison

Pentium III vs. NetBurst

https://courses.cs.washington.edu/courses/cse378/10au/lectures/Pentium4Arch.pdf
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.

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
- ...


Summary: Maintaining Precise State

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

Readings
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)
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 when we discuss store-load handling
Reorder Buffer Example

Register File (RF)

Initially: all registers are valid in RF & ROB is empty

Simulate:
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Value Valid?
Value or Tag (i.e., pointer to ROB entry)

Reorder Buffer (ROB)

Oldest instruction
Youngest instruction

Entry Valid?
Dest reg ID
Dest reg value
Dest reg written?