CS 152: Discussion Section 6
Out-of-Order Execution

Albert Ou, Yue Dai
03/06/2020
Administrivia

- Lab 2 due 10:30am on Mon, March 9
- Problem Set 3 due 10:30am on Mon, March 16
- Midterm 1 scores will be available on Gradescope on Wed, March 11
  - One week to submit regrade requests
  - Note: Regrades invite further scrutiny (score might increase or decrease)
Post-Midterm Poll

- What topics should we cover in future discussions?
- Better scheduling of office hours?
Agenda

- Precise exceptions review
  - Underappreciated (yet vital) component of HW/SW contract
  - Recurring concept in OoO context
- Register renaming
  - Cannot understand OoO without understanding this
- Tomasulo’s algorithm
Precise Exception Model

- Q: Should instruction A be killed in pipeline flush if instruction B has committed?
- Q: What should EPC point to?

insn A
insn B
insn C
insn D
interrupt
Why are Precise Exceptions Useful?

Restartable

- Not all traps terminate a program
  - Page faults, syscalls, etc.
- Well-defined architectural state simplifies returning from exception
  - Resume execution by jumping back to EPC (or EPC+4)
  - No visible side effects from partial execution → no need to save/restore microarchitectural state
Why are Precise Exceptions Useful?

Deterministic

- Valuable for reproducibility and debugging
- Easy to identify the exact instruction that faulted
- Program state (registers, coredump, commit trace) matches mental model that programmers have about sequential execution
Why are Precise Exceptions Problematic?

Microarchitectural complexity

- Must preserve enough information for hardware to recover architectural state and repair internal state
  - Checkpointing rename tables
- In-order commit requirement can limit performance
  - Head-of-line blocking in ROB
- Difficult to avoid partial side effects for more complex instructions
  - Vector memory operations
Why is Out-of-Order Execution Useful?

• Exploit instruction-level parallelism (ILP) to keep processor busy
  □ Make suboptimal code run fast
• Dynamically schedule around long-latency instructions
  
  ```
  ld x2, 0(x1)  # cache miss: 200 cycles
  add x5, x3, x4
  ld x7, 4(x6)
  ```
• Initiate long-latency instructions earlier
What Limits OoO Performance?

A: fmul f1, f0, f2
B: fadd f0, f3, f1
C: fmul f3, f2, f3
D: fadd f3, f3, f1

- Want to issue instruction C right after A, but cannot reorder it earlier due to WAR hazard on B (f3)
- Suppose only four F registers exist, and it is not feasible for compiler to choose f2 as the destination of C since f2 is read by a later instruction
What Limits OoO Performance?

- WAW/WAR hazards
  - Caused by reuse of limited set of architectural (named) registers
  - Would not exist if an infinite number of registers were available
  - Not a “true” data dependency
- How can x86 (8 “GPRs”) and x86-64 (16 GPRs) implementations achieve high performance?
- How can we use more registers than what the ISA specifies?
Register Renaming

- Main idea: Decouple architectural registers (used for expressing dataflow) from physical registers (used for storage)
  - For each in-flight instruction, rename the destination register with a unique tag that refers to a separate buffer to hold result
  - Somehow maintain relationship between tags and ISA registers
- “All problems in computer science can be solved by another level of indirection” - David Wheeler, inventor of the subroutine call
Register Renaming

A: \( \text{fmul } f1, f0, f2 \)  \( \rightarrow \)  \( \text{fmul } P4, P0, P2 \)
B: \( \text{fadd } f0, f3, f1 \)  \( \rightarrow \)  \( \text{fadd } P5, P3, P4 \)
C: \( \text{fmul } f3, f2, f3 \)  \( \rightarrow \)  \( \text{fmul } P6, P2, P3 \)
D: \( \text{fadd } f3, f3, f1 \)  \( \rightarrow \)  \( \text{fadd } P7, P6, P4 \)

- Resembles single static assignment (SSA) form

<table>
<thead>
<tr>
<th>Rename Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>f0</td>
</tr>
<tr>
<td>f1</td>
</tr>
<tr>
<td>f2</td>
</tr>
<tr>
<td>f3</td>
</tr>
</tbody>
</table>
Tomasulo’s Algorithm (Q1)

- **On instruction dispatch** (in program order):
  1. Allocate reservation station (RS) entry
  2. If source register has “present” (P) bit set in register file (RF) entry, copy value into tag/data field in RS and set P bit for operand
  3. Otherwise, copy tag from RF into RS and clear P bit for operand
  4. Replace RF entry for destination register with tag assigned to RS entry \( (\text{tag}_{\text{dest}}) \)

- **Prior to execution**:
  1. For missing operands, monitor result bus for tag match; replace tag with value; set P
  2. When all operands are present, issue to functional unit

- **On completion**:
  1. Broadcast \(<\text{tag}_{\text{dest}}, \text{result}>\) on result bus for RF and other RS entries to consume
  2. Deallocate RS entry
Tomasulo’s Algorithm

Q: Why can’t the reservation station entry for an instruction be deallocated immediately on issue?

A: \texttt{fmul} f4, f0, f1  \quad \# \text{Dispatched and issued immediately; RS is freed}

B: \texttt{fmul} f5, f2, f3  \quad \# \text{Allocated same RS as A before A has written back}

f4 and f5 now assigned the same tag in regfile, causing instruction B to incorrectly clobber f4 on writeback
Q: Why are exceptions *imprecise* in this implementation?

- Register file is irrevocably modified on dispatch
- No mechanism to recover original value of destination register if instruction causes an exception
How to Regain Precise Exceptions?

Reorder Buffer (ROB) separates commit from completion:

- **Completion**: Result available (out-of-order)
- **Commit**: Architectural state updated (in-order)
Data-in-ROB

Both tags and data held in ROB, with separate architectural register file
Unified Physical Register File

Physical register file holds both committed and temporary values; Only tags held in ROB
Renaming with Unified PRF (Q2)

- **On dispatch:**
  1. Allocate new physical register for destination from free list
  2. Update decode-stage mapping

- **On commit:**
  1. Update architectural mapping
  2. Deallocate *previous* physical register for destination; re-add to free list

- **On exception:**
  1. Repair decode-stage rename table by un-renaming in reverse order; walk through ROB entries from newest to oldest (MIPS R10k approach)