15-740/18-740
Computer Architecture
Lecture 7: Out-of-Order Execution

Prof. Onur Mutlu
Carnegie Mellon University
Readings

General introduction and basic concepts
- Hennessy and Patterson, Sections 2.1-2.10 (inclusive).

Modern designs

Seminal papers
Reviews Due

- Due September 30
Last Time …

- Exceptions vs. Interrupts
- Precise Exceptions
  - What
  - How
    - Reorder buffer
    - History buffer
    - Future file
    - Checkpointing
- Register renaming: architectural vs. physical registers
- Handling out-of-order completion of stores
  - Store buffer
Today and the Next Related Lectures

- Exploiting Instruction Level Parallelism (ILP)

- Out-of-order execution
  - Tomasulo’s algorithm
  - Precise exceptions

- Superscalar processing
  - Instruction dependency checking/detection

- Better instruction supply
  - Control flow handling: branch prediction, predication, etc
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
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
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
Review: In-order pipeline

- **Problem**: A true data dependency stalls dispatch of younger instructions into functional (execution) units

- **Dispatch**: Act of sending an instruction to a functional unit
Can We Do Better?

What do the following two pieces of code have in common (with respect to execution in the previous design)?

<table>
<thead>
<tr>
<th>Code</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMUL R3 ← R1, R2</td>
<td>LD R3 ← R1 (0)</td>
</tr>
<tr>
<td>ADD R3 ← R3, R1</td>
<td>ADD R3 ← R3, R1</td>
</tr>
<tr>
<td>ADD R1 ← R6, R7</td>
<td>ADD R1 ← R6, R7</td>
</tr>
<tr>
<td>IMUL R3 ← R6, R8</td>
<td>IMUL R3 ← R6, R8</td>
</tr>
<tr>
<td>ADD R7 ← R3, R9</td>
<td>ADD R7 ← R3, R9</td>
</tr>
</tbody>
</table>

Answer: First ADD stalls the whole pipeline!

- ADD cannot dispatch because its source registers unavailable
- Later independent instructions cannot get executed

How are the above code portions different?

- Answer: Load latency is variable (unknown until runtime)
- What does this affect? Think compiler vs. microarchitecture
Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen THREE:
  - 1.
  - 2.
  - 3.
- What are the disadvantages of the above three?

- Any other way to prevent dispatch stalls?
  - Actually, you have briefly seen the basic idea before
    - Dataflow: fetch and “fire” an instruction when its inputs are ready
  - Problem: in-order dispatch (issue, execution)
  - Solution: out-of-order dispatch (issue, execution)
Terminology

- Issue vs. dispatch
- Scheduling
- Execution, completion, retirement/commit
- Graduation
- Out-of-order execution versus superscalar processing
Out-of-order Execution (Dynamic Scheduling)

- Idea: Move the dependent instructions out of the way of independent ones
- Rest areas for dependent instructions: Reservation stations

- Monitor the source “values” of each instruction in the resting area
- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order

- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation
In-order vs. Out-of-order Dispatch

- **In order dispatch:**

```
F  D  E  E  E  E  E  R  W
F  D        STALL       E  R  W
 F          STALL       D  E  R  W
F  D  E  E  E  E  E  E  R  W
 F          STALL       E  R  W
```

- **Tomasulo + precise exceptions:**

```
F  D  E  E  E  E  E  E  R  W
F  D        WAIT       E  R  W
 F          D  E  R  W
F  D  E  E  E  E  E  E  R  W
 F          D  E  R  W
```

- **16 vs. 12 cycles**
Enabling OoO Execution

1. Need to link the consumer of a value to the producer
   - Register renaming: Associate a “tag” with each data value
2. Need to buffer instructions until they are ready
   - Insert instruction into reservation stations after renaming
3. Instructions need to keep track of readiness of source values
   - Broadcast the “tag” when the value is produced
   - Instructions compare their “source tags” to the broadcast tag
     → if match, source value becomes ready
4. When all source values of an instruction are ready, dispatch the instruction to functional unit (FU)
   - What if more instructions become ready than available FUs?
Tomasulo’s Algorithm

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units

- Variants of it used in most high-performance processors
  - Most notably Pentium Pro, Pentium M, Intel Core(2)
  - Alpha 21264, MIPS R10000, IBM POWER5

- What is the major difference today?
  - **Precise exceptions**: IBM 360/91 did NOT have this
Two Humps in a Modern Pipeline

- **Hump 1**: Reservation stations (scheduling window)
- **Hump 2**: Reordering (reorder buffer, aka instruction window or active window)
General Organization of an OOO Processor

Register Renaming

- 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 because not enough register ID’s (i.e. names) in the ISA
- The register ID is renamed to the reservation station entry that will hold the register’s value
  - Register ID $\rightarrow$ RS entry ID
  - Architectural register ID $\rightarrow$ Physical register ID
  - After renaming, RS entry ID used to refer to the register

- This eliminates anti- and output- dependencies
  - Approximates the performance effect of a large number of registers even though ISA has a small number
Tomasulo’s Algorithm: Renaming

- Register rename table (register alias table)

<table>
<thead>
<tr>
<th></th>
<th>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Tomasulo’s Algorithm

- If reservation station available before renaming
  - Instruction + renamed operands (source value/tag) inserted into the reservation station
  - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
  - Watches common data bus (CDB) for tag of its sources
  - When tag seen, grab value for the source and keep it in the reservation station
  - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
  - Arbitrate for CDB
  - Put tagged value onto CDB (tag broadcast)
  - Register file is connected to the CDB
    - Register contains a tag indicating the latest writer to the register
    - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
  - Reclaim rename tag
    - no valid copy of tag in system!
An Exercise

Assume ADD (4 cycle execute), MUL (6 cycle execute)

Assume one adder and one multiplier

How many cycles

- in a non-pipelined machine
- in an in-order-dispatch pipelined machine with future file and reorder buffer
- in an out-of-order dispatch pipelined machine with future file and reorder buffer