18-447
Computer Architecture
Lecture 14: Out-of-Order Execution

Prof. Onur Mutlu
Carnegie Mellon University
Spring 2013, 2/18/2013
Reminder: Homework 3

- Homework 3
  - Due Feb 25
  - REP MOVS in Microprogrammed LC-3b, Pipelining, Delay Slots, Interlocking, Branch Prediction
Lab Assignment 3 Due March 1

- Lab Assignment 3
  - Due Friday, March 1
  - Pipelined MIPS implementation in Verilog
  - All labs are individual assignments
  - No collaboration; please respect the honor code

- Extra credit: Optimize for execution time!
  - Top assignments with lowest execution times will get extra credit.
  - And, it will be fun to optimize...
Reminder: A Note on Testing Your Code

- Testing is critical in developing any system

- You are responsible for creating your own test programs and ensuring your designs work for all possible cases

- That is how real life works also...
  - No one gives you all possible test cases, workloads, users, etc. beforehand
Course Feedback Sheet

- Was due Feb 15, in class
- But, please still turn it in
- We would like your honest feedback on the course
Reading for Today

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts
Readings for Next Lecture

- SIMD Processing
- Basic GPU Architecture


- Stay tuned for more readings...
Readings for Next Week

- Virtual Memory
- Section 5.4 in Patterson & Hennessy
- Section 8.8 in Hamacher et al.
State maintenance and recovery mechanisms
- Reorder buffer
- History buffer
- Future file
- Checkpointing

Interrupts/exceptions vs. branch mispredictions

Handling register vs. memory state
Today

- Out-of-order execution
Out-of-Order Execution
(Dynamic Instruction Scheduling)
An 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)?

  IMUL R3 ← R1, R2
  ADD R3 ← R3, R1
  ADD R1 ← R6, R7
  IMUL R5 ← R6, R8
  ADD R7 ← R3, R5

  LD R3 ← R1 (0)
  ADD R3 ← R3, R1
  ADD R1 ← R6, R7
  IMUL R5 ← R6, R8
  ADD R7 ← R3, R5

- 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. Fine-grained multithreading
  - 2. Value prediction
  - 3. Compile-time instruction scheduling/reordering

- 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 (scheduling, or execution)
  - Solution: out-of-order dispatch (scheduling, or execution)
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 + precise exceptions:

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

```
IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5
```

- Out-of-order dispatch + precise exceptions:

```
F D 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 WAIT 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 to execute
   - 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, need to dispatch the instruction to its functional unit (FU)
   - Instruction wakes up if all sources are ready
   - If multiple instructions are awake, need to select one per FU
Tomasulo’s Algorithm

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

- What is the major difference today?
  - **Precise exceptions:** IBM 360/91 did NOT have this

- Variants used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15
Two Humps in a Modern Pipeline

- **Hump 1**: Reservation stations (scheduling window)
- **Hump 2**: Reordering (reorder buffer, aka instruction window or active window)
Tomasulo’s Machine: IBM 360/91

- From memory: Load buffers to FP FU
- From instruction unit: FP registers to FP FU
- Operation bus: FP FU to reservation stations
- FP registers
- Common data bus
- Store buffers to memory
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 → RS entry ID
  - Architectural register ID → 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>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R9</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 imprecise exceptions (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine imprecise exceptions (full forwarding)
Exercise Continued

Pipeline structure

```
MUL R1, R2, → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5
```

MUL takes 6 cycles
ADD takes 4 cycles

How many cycles total with data forwarding?

```
F D E W
↓
Can take multiple cycles
```

26
Exercise Continued

FD123456W
FD-----D1234W
F-------D1234W
FD1234W
FD-----D123456W
F-------D1234W

Execution timeline w/ scoreboard:

FD123456W
FD  E1234W
F    D1234W
FD01234W
FD    1234W
F    D1234W

31 cycles

25 cycles
Exercise Continued

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

Tomasulo's algorithm + full forwarding

20 cycles
How It Works
Cycle 0

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

- Initial contents of the register okes table

- Recording status are all invalid
Cycle 2:

\[ \text{MUL } R1, R2 \rightarrow R3 \]
reads its sources from the RAT
writes to its destination in the RAT
renames its destination
allocates a reservation station entry
allocates a tag for its destination register

places its sources in the reservation station entry that is allocated.

---

End of cycle 2:

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>R3</td>
<td>O</td>
<td>X</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>R5</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

---

\[ \text{MUL at } X \text{ becomes ready to execute.} \]
(What if multiple instructions become ready at the same time?)

both of its sources are valid in the reservation station X
Cycle 3

- Cycle 3:
  - MUL at X starts execution
  - ADD R3, R4, RS gets renamed and placed into the ADDER registers stations

End of cycle 3:

<table>
<thead>
<tr>
<th>R1</th>
<th>1 ~ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>1 ~ 2</td>
</tr>
<tr>
<td>R3</td>
<td>0 X 3</td>
</tr>
<tr>
<td>R4</td>
<td>1 ~ 4</td>
</tr>
<tr>
<td>R5</td>
<td>0 ~ 4</td>
</tr>
<tr>
<td>R6</td>
<td>1 ~ 6</td>
</tr>
<tr>
<td>R7</td>
<td>1 ~ 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U tag value</th>
<th>V tag value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>X 1 ~ 1</td>
<td>Y 1 ~ 11</td>
</tr>
</tbody>
</table>

- ADD at a cannot be ready to execute because one of its sources is not ready
- It is waiting for the value with the tag X to be broadcast (by the MUL in X1)

Aside: Does the tag need to be associated with the RS entry of the producer?
Answer: No; Tag is a tag for the value that is communicated.

RS is a place to hold the results like value communicated: these two are completely orthogonal.
Cycle 4

cycle 4:  ADD R2, R6 → R7 gets renamed and placed into RS

end of cycle 4:

- ADD at b becomes ready to execute (both sources are ready!)

- At cycle 5, it is sent to the adder out-of-program order
  → It is executed before the add in A
Cycle 7

end of cycle 7:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>2</td>
</tr>
<tr>
<td>R3</td>
<td>X</td>
</tr>
<tr>
<td>R4</td>
<td>4</td>
</tr>
<tr>
<td>R5</td>
<td>d</td>
</tr>
<tr>
<td>R6</td>
<td>6</td>
</tr>
<tr>
<td>R7</td>
<td>b</td>
</tr>
<tr>
<td>R8</td>
<td>8</td>
</tr>
<tr>
<td>R9</td>
<td>c</td>
</tr>
<tr>
<td>R10</td>
<td>Y</td>
</tr>
<tr>
<td>R11</td>
<td></td>
</tr>
</tbody>
</table>

X
1 ~ 1
1 ~ 2
1 ~ 4
1 ~ 6
1 ~ 8
1 ~ 9
0 ~ a
0 ~ b
0 ~ c
0 ~ y

* All 6 instructions removed.
- Note what happened to R5
Cycle 8:

- MUL at X and ADD at b
  broadcast their tags and values

- RS entries waiting for these tags capture the values
  and set the Valid bs accordingly

  → (What is needed in HW to accomplish this?)

  CAM on tags that are broadcast for all RS
  entries' sources

- RAT entries waiting for these tags also capture the
  values and set the Valid bS accordingly
An Exercise, with Precise Exceptions

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

- 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 reorder buffer (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine with reorder buffer (full forwarding)
Out-of-Order Execution with Precise Exceptions

- **Idea:** Use a reorder buffer to reorder instructions before committing them to architectural state

- An instruction updates the register alias table (essentially a future file) when it completes execution

- An instruction updates the architectural register file when it is the oldest in the machine and has completed execution
Out-of-Order Execution with Precise Exceptions

- Hump 1: Reservation stations (scheduling window)
- Hump 2: Reordering (reorder buffer, aka instruction window or active window)
Enabling OoO Execution, Revisited

1. Link the consumer of a value to the producer
   - Register renaming: Associate a “tag” with each data value

2. Buffer instructions until they are ready
   - Insert instruction into reservation stations after renaming

3. Keep track of readiness of source values of an instruction
   - 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)
   - Wakeup and select/schedule the instruction
Summary of OOO Execution Concepts

- Register renaming eliminates false dependencies, enables linking of producer to consumers
- Buffering enables the pipeline to move for independent ops
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch
An out-of-order engine dynamically builds the dataflow graph of a piece of the program
which piece?

The dataflow graph is limited to the instruction window
Instruction window: all decoded but not yet retired instructions

Can we do it for the whole program?
Why would we like to?
In other words, how can we have a large instruction window?
Can we do it efficiently with Tomasulo’s algorithm?
Dataflow Graph for Our Example

MUL   R3 ← R1, R2
ADD   R5 ← R3, R4
ADD   R7 ← R2, R6
ADD   R10 ← R8, R9
MUL   R11 ← R7, R10
ADD   R5 ← R5, R11
State of RAT and RS in Cycle 7

end of cycle 7:

- All 6 instructions renamed.
  - Note what happened to R5
Dataflow Graph

**Nodes:** operations performed by the instruction

**Arrows:** tags in Tomasulo's algorithm

\[
\begin{align*}
\text{MUL} & \quad R1, R2 \to R3 \quad (x) \\
\text{ADD} & \quad R3, R4 \to R5 \quad (a) \\
\text{ADD} & \quad R2, R6 \to R7 \quad (b) \\
\text{ADD} & \quad R8, R9 \to R10 \quad (c) \\
\text{MUL} & \quad R7, R10 \to R11 \quad (y) \\
\text{ADD} & \quad R5, R11 \to R5 \quad (d)
\end{align*}
\]
Restricted Data Flow

- An out-of-order machine is a “restricted data flow” machine
  - Dataflow-based execution is restricted to the microarchitecture level
  - ISA is still based on von Neumann model (sequential execution)

- Remember the data flow model (at the ISA level):
  - Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
Questions to Ponder

- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - **Latency tolerance**: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - **What limits the latency tolerance scalability of Tomasulo’s algorithm?**
    - **Active/instruction window size**: determined by register file, scheduling window, reorder buffer
So far, we considered register based value communication between instructions

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)
Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance

- Observation and Problem: Memory address is not known until a load/store executes

- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores need to be handled after their execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine
Memory Dependence Handling (II)

- When do you schedule a load instruction in an OOO engine?
  - Problem: A younger load can have its address ready before an older store’s address is known
  - Known as the memory disambiguation problem or the unknown address problem

- Approaches
  - **Conservative:** Stall the load until all previous stores have computed their addresses (or even retired from the machine)
  - **Aggressive:** Assume load is independent of unknown-address stores and schedule the load right away
  - **Intelligent:** Predict (with a more sophisticated predictor) if the load is dependent on the/any unknown address store