Announcements

- Lab due Friday (Feb 21)
- Homework 3 due next Wednesday (Feb 26)
- Exam coming up (before Spring Break)
Reminder: Lab Late Day Policy Adjustment

- Your total late days have increased to 7

- Each late day beyond all exhausted late days costs you 15% of the full credit of the lab
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
Lab 2 Grade Distribution

# Students

0 10 20 30 40 50 60 70 80 90 100

Lab 2 Grade Distribution
Lab 2 Statistics

- MAX       99.62
- MIN       62.74
- MEDIAN    92.59
- MEAN      89.26
- STD       10.21
HW 2 Grade Distribution

HW 2 Grade Distribution

Number of Students

0 10 20 30 40 50 60 70 80 90 100

0 5 10 15 20 25
HW 2 Statistics

- MAX 100
- MIN 0
- MEDIAN 92.98
- MEAN 81.98
- STD 24.82
Readings for Past Few Lectures (I)

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts


Readings for Past Few Lectures (II)

Readings Specifically 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
- Other execution models: VLIW, Dataflow


- Stay tuned for more readings...
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
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 ← R9, R9
```

```
LD R3 ← R1 (0)
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R9, R9
```

- **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:

  - 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:

- 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)
General Organization of an OOO Processor

Tomasulo’s Machine: IBM 360/91

- Load buffers from memory
- FP FU from instruction unit
- FP registers
- Operation bus
- Reservation stations
- 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

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

Pipeline structure

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

```
MUL takes 6 cycles
ADD takes 4 cycles

How many cycles total with data forwarding?
```

```
Exercise Continued

FD 1 2 3 4 5 6 W
FD - - - - - D 1 2 3 4 W
F - - - - - D 1 2 3 4 W
FD 1 2 3 4 W
FD - - - - D 1 2 3 4 5 6 W
F - - - - D 1 2 3 4 W

Execution timeline w/ scoreboard:
31 cycles

FD 1 2 3 4 5 6 W
FD D 1 2 3 4 W
F D 1 2 3 4 W
FD 1 2 3 4 W
FD 1 2 3 4 W
FD 1 2 3 4 5 6 W
F D 1 2 3 4 W

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>R1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></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>R11</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

- Initial contents of the register index table
- Reservation stations 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.

\[ \text{End of cycle 2:} \]

\[ \begin{array}{c|c|c}
V & \text{tag} & \text{value} \\
\hline
R1 & 1 & 1 \\
R2 & 1 & 2 \\
R3 & 0 & X \\
R4 & 1 & 4 \\
R11 & 1 & 11 \\
\end{array} \]

- MUL at X becomes ready to execute.  
(Wait 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 gates renamed and placed into the ADDER recovers 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 ~ 4</td>
</tr>
<tr>
<td>R4</td>
<td>1 ~ 4</td>
</tr>
<tr>
<td>RS</td>
<td>0 a ~ 4</td>
</tr>
<tr>
<td>RB</td>
<td>1 ~ 6</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 X)

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 register values while they become ready. These two are completely orthogonal.
Cycle 4

cycle 4:  \[ \text{ADD} R2, R6 \rightarrow R7 \] 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>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>0</td>
<td>x</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>R5</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>R8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>R11</td>
<td>0</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>o</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>c</td>
</tr>
</tbody>
</table>

* All 6 resistors 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 bit accordingly

  \( \rightarrow \) (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 bits accordingly
An Exercise, with Precise Exceptions

- 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