6.823 SPring15 Quiz 2 Review (L09-L14)

TA: Po-An Tsai, Hsin-Jung Yang
Lecture 9-14 are all about ILP

• Instruction-level parallelism (ILP)
  – Execute as many instructions as possible at the same time to maximize throughput and hide long latency by memory/ALU

• Why this is not easy?
  – Dependencies between instructions
Dependency

• Data dependency
  – RAW WAR WAW

• Control dependency
  – Branch jump

• Structural dependency
  – Only one ALU
Ways to Solve Dependency

- For false dependency
  - Indirection
- For true dependency
  - Stall
  - Bypass
  - Speculate => best if you speculate mostly correctly
Ways to Solve Dependency

• Data dependency
  – RAW WAR WAW
  => Scoreboard (L9) ROB, Register renaming (L10), Store queue (L13)

• Control dependency
  – Branch jump
  => Branch prediction (L11)

• Structural dependency
  – Only one ALU
  => Superscalar (L9)
Speculation

• We speculate a lot!
  – Branch
  – No exceptions
  – Addresses for load/store are not the same

• How to manage old and new values?
  – Greedy/lazy (L12)
Mis-speculation

• Recovery according to your policy (L12)
  – Snapshot
  – Rollback
Hide Long Latency

• A cache miss takes 100 cycles
• A divide takes 20 cycles
  – Execute following instructions, but hold them until those long-latency instruction finish.

• Use Little’s law to calculate how many instruction in flight
Hide Long Latency

• A cache miss takes 100 cycles
• A divide takes 20 cycles
  – Switch to another independent thread until they finish (L14)

• Use Little’s law to calculate how many threads needed
Example OoO Pipeline

Figure 1: Out-of-order Pipeline
Branch Prediction

Figure 1: Out-of-order Pipeline
Branch Prediction

• Speculate what the next instruction is
  – Static
  – 1 bit predictor
  – 2 bit predictor
  – Global history (a history register and lots of predictor)
  – Local history (many history registers and lots of predictor)
  – Combined local and global history
# Branch Prediction

- **Data management**

## Branch Predictor Recovery

<table>
<thead>
<tr>
<th></th>
<th>1-Bit Counter Recovery</th>
<th>2-Bit Counter Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC</strong></td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td><strong>Lazy</strong></td>
<td></td>
<td><strong>Lazy</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Global History Recovery</th>
<th>Local History Recovery</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PC</strong></td>
<td>10101010</td>
<td>10101010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01010101</td>
</tr>
<tr>
<td><strong>Greedy</strong></td>
<td></td>
<td><strong>Greedy!!</strong></td>
</tr>
</tbody>
</table>
Branch Target Buffer

• Store the next PC of branch/jmp instructions seen last time

• Get address earlier than predictor
Combine BTB and Predictor(BHT)

BHT in later pipeline stage corrects when BTB misses a predicted taken branch

BTB/BHT only updated after branch resolves in E stage
Out-of-Order Execution

When can we execute an instruction out-of-order?

Need to consider data dependency
(register dependency, memory dependency)
## Data Dependency

- **Register Dependency**

<table>
<thead>
<tr>
<th>Data-dependence</th>
<th>Anti-dependence</th>
<th>Output-dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ r_3 \leftarrow (r_1) \text{ op } (r_2) ]</td>
<td>[ r_3 \leftarrow (r_1) \text{ op } (r_2) ]</td>
<td>[ r_3 \leftarrow (r_1) \text{ op } (r_2) ]</td>
</tr>
<tr>
<td>[ r_5 \leftarrow (r_3) \text{ op } (r_4) ]</td>
<td>[ r_1 \leftarrow (r_4) \text{ op } (r_5) ]</td>
<td>[ r_3 \leftarrow (r_6) \text{ op } (r_7) ]</td>
</tr>
<tr>
<td>Read-after-Write (RAW) hazard</td>
<td>Write-after-Read (WAR) hazard</td>
<td>Write-after-Write (WAW) hazard</td>
</tr>
</tbody>
</table>
## Data Dependency

### Register Dependency

**Data-dependence**
\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \\
    r_5 & \leftarrow (r_3) \text{ op } (r_4)
\end{align*}
\]
Read-after-Write (RAW) hazard

**Anti-dependence**
\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \\
    r_1 & \leftarrow (r_4) \text{ op } (r_5)
\end{align*}
\]
Write-after-Read (WAR) hazard

**Output-dependence**
\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \\
    r_3 & \leftarrow (r_6) \text{ op } (r_7)
\end{align*}
\]
Write-after-Write (WAW) hazard

*Handled by register naming*
Data Dependency

- Memory Dependency

```c
st r1, 4(r2)
ld r3, 8(r4)
```

When is the load dependent on the store?

*When* \((r2 + 4) == (r4 + 8)\)

Do we know this issue when the instruction is decoded?  **No**
Data Dependency

• Memory Dependency

\[
\begin{align*}
\text{st} & \ r1, \ 4(r2) \\
\text{ld} & \ r3, \ 8(r4)
\end{align*}
\]

Solution:

(1) Stall: can execute load before store only if the addresses are known to be different

(2) Address speculation: guess \(r2+4 \neq r4+8\) and execute load before store

Speculative Load Buffer
Data Dependency

- Speculative Load Buffer

Speculation check:
Detect if a load has executed before an earlier store to the same address – missed RAW hazard

On load execute: mark entry valid, instruction number and tag
On load commit: clear valid bit
On load abort: clear valid bit
On store execute: if tag matches and the instruction is younger than the store -> Abort!
Speculative Data Management

• When do we do speculation?
  – Branch prediction
  – Assume no exceptions/interrupts
  – Assume no memory dependency
  – ...

• How do we manage speculative data?
  – Greedy (or Eager) Update
    • Update the value in place
    • Maintain a log of old values to use for recovery
  – Lazy Update
    • Buffer the new value and leave the old value in place
    • Replace the old value only at ‘commit’ time
Speculative Data Management

- **Type of speculative data**
  - **Branch prediction**
    - history registers, prediction counters (see P.13)
  - **Register values**
    - Lazy update: store new values in the ROB and update registers during commit
    - Hybrid: store both new and old values in the unified physical register file
  - **Store values to memory**
    - Laze update: store new values in the speculative store buffer and write to non-speculative store buffer/cache/memory during commit
Speculative Data Management

- Register Value Management
  - Approach 1: store new values in the ROB and update registers during commit

Space Inefficient
Speculative Data Management

- Register Value Management
  - Approach 2: keep all data values in a physical register file

```
<table>
<thead>
<tr>
<th>Rename Table</th>
<th>Physical Regs</th>
<th>Free List</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>P0</td>
<td>P0</td>
</tr>
<tr>
<td>R1</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>R2</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>R3</td>
<td>P3</td>
<td>P2</td>
</tr>
<tr>
<td>R4</td>
<td>P4</td>
<td>P4</td>
</tr>
<tr>
<td>R5</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>P6</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Reorder buffer contains only tags
Out-of-Order Execution

Pre-Issue Check:

- The ROB is checked for available slots
- The free list is checked for free rename registers (if necessary)
- For store instructions, the non-speculative store buffer is checked for available slots
Out-of-Order Execution

ROB Insert:

– The instruction is inserted into the ROB only if all the checks in the previous cycle (Pre-Issue check) pass
– The destination register is renamed
Out-of-Order Execution

**Rename Table**

<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>P6</td>
<td>P0</td>
<td>P7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Physical Regs**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;R6&gt;</td>
<td>p</td>
<td></td>
<td>&lt;R3&gt;</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;R7&gt;</td>
<td>p</td>
<td></td>
<td></td>
<td>p</td>
</tr>
</tbody>
</table>

**Free List**

- ld r1, 0(r3)
- add r3, r1, #4
- sub r6, r7, r6
- add r3, r3, r6
- ld r6, 0(r1)

**ROB**

<table>
<thead>
<tr>
<th>use</th>
<th>ex</th>
<th>op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>LPRd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>ld</td>
<td>p</td>
<td>P7</td>
<td></td>
<td></td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
</tr>
</tbody>
</table>
Out-of-Order Execution

**Rename Table**

<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>P6</td>
</tr>
</tbody>
</table>

**Physical Regs**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;R6&gt;</td>
<td></td>
<td></td>
<td>&lt;R1&gt;</td>
</tr>
</tbody>
</table>

**Free List**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
</table>

**ROB**

<table>
<thead>
<tr>
<th>use</th>
<th>ex</th>
<th>op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>LPRd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>ld</td>
<td>p</td>
<td>P7</td>
<td></td>
<td></td>
<td></td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
</tr>
<tr>
<td>✗</td>
<td>add</td>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r3</td>
<td>P7</td>
<td>P1</td>
</tr>
<tr>
<td>✗</td>
<td>sub</td>
<td>P6</td>
<td>p</td>
<td>P5</td>
<td></td>
<td></td>
<td>r6</td>
<td>P5</td>
<td>P3</td>
</tr>
<tr>
<td>✗</td>
<td>add</td>
<td>P1</td>
<td></td>
<td>P3</td>
<td>r3</td>
<td></td>
<td></td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>✗</td>
<td>ld</td>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r6</td>
<td>P3</td>
<td>P4</td>
</tr>
</tbody>
</table>

Ready to execute
Out-of-Order Execution

Reg File Read:
- Operand values are read from the unified physical register file

Execute:
- Integer and floating point operations are sent to the appropriate functional units
- Stores enter the speculative store buffer
- Loads read from the store buffer or cache/memory
Out-of-Order Execution

Reg File Write:

- The output from the functional units/memory is written into the unified register file and the ROB is notified.
Out-of-Order Execution

- Rename Table
- Physical Regs
- Free List
- ROB

Example:
- LD r1, 0(r3)
- ADD r3, r1, #4
- SUB r6, r7, r6
- ADD r3, r3, r6
- LD r6, 0(r1)
Commit:

- Instructions are committed in-order
- Free the previously mapped physical register
- Data is written to cache/memory/non-speculative store buffer when a store is committed
- The ROB entry is freed after commit
Out-of-Order Execution

Renamed Table

| R0 | P0 |
| R1 | P0 |
| R2 | P1 |
| R3 | P1 |
| R4 | P2 |
| R5 | P2 |
| R6 | P4 |
| R7 | P6 |

Physical Regs

| P0 | <R1> | p |
| P1 |     |   |
| P2 |     |   |
| P3 |     |   |
| P4 |     |   |
| P5 | <R6> | p |
| P6 | <R7> | p |
| P7 | <R3> | p |
| P8 | <R1> | p |
| Pn |     |   |

Free List

| P0 |
| P1 |
| P2 |
| P3 |
| P4 |
| P5 |
| P6 |
| P7 |
| P8 |

ROB

<table>
<thead>
<tr>
<th>use</th>
<th>ex</th>
<th>op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>LPRd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>ld</td>
<td>p</td>
<td>P7</td>
<td>P2</td>
<td>PR2</td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>add</td>
<td>p</td>
<td>P0</td>
<td></td>
<td></td>
<td>r3</td>
<td>P7</td>
<td>P1</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>sub</td>
<td>p</td>
<td>P6</td>
<td>p</td>
<td>P5</td>
<td>r6</td>
<td>P5</td>
<td>P3</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>add</td>
<td></td>
<td>P1</td>
<td>P3</td>
<td>r3</td>
<td>P1</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>ld</td>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td>r6</td>
<td>P3</td>
<td>P4</td>
</tr>
</tbody>
</table>

Commit

ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6
ld r6, 0(r1)
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

Take instructions from different programs

*Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe*

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)
Scheduling Policy

• Fine-grained multithreading
  – Context switch among threads every cycle

• Coarse-grained multithreading
  – Context switch among threads every few cycles, e.g., on:
    • Function unit data hazard,
    • L1 miss,
    • L2 miss...
Simultaneous Multithreading (SMT)

- Share OOO structures between threads
The end

Good luck!! 😊