Lecture 6: Dynamic Instruction Scheduling

• Last Time
  - Finish basic microarchitecture
  - Pipelining

• Announcements
  - CS logins
  - HW1 solution set

• Today
  - Brief hw1 discussion
  - More pipelining discussion, continue from hazards
  - Multi-issue and dynamic instruction scheduling
  - VAX paper
  - Pentium4 performance counters

Slides courtesy of Stephen W. Keckler, UT-Austin

Types of Data Hazards

• RAW (read after write)
  - only hazard for ‘fixed’ pipelines
  - later instruction must read after earlier instruction writes

• WAW (write after write)
  - variable-length pipeline
  - later instruction must write after earlier instruction writes

• WAR (write after read)
  - pipelines with late read
  - later instruction must write after earlier instruction reads
How Do We Speed up the Pipeline?

- Instruction Level Parallelism (ILP)
  - Multi-issue (in-order execution to multiple pipes)
  - Dynamic scheduling ("Superscalar")
  - Compiler schedules ILP (VLIW/EPIC)
- Undetermined dependencies at compile time => dynamic scheduling
  - Object code compatibility
  - Simplify compiler
- WAR/WAW hazards => register renaming
  \[
  \begin{align*}
  \text{ADD} & \ R1, R2, R3 \ \Rightarrow \ \text{ADD} & \ R1, R2, R3 \\
  \text{SUB} & \ R1, R4, R5 \ \Rightarrow \ \text{SUB} & \ R1', R4, R5
  \end{align*}
  \]
- Too many branches => better branch prediction
  - Or use predication to eliminate branches
- Unknown dependencies (control/data) => speculate
- One more solution!

Multiple Issue – No instruction reordering
Multiple Issue (Details)

- Dependencies and structural hazards checked at run-time
- Can run existing binaries
  - Recompile for performance, not correctness
  - Example - Pentium

- More complex issue logic
  - Swizzle next N instructions into position
  - Check dependencies and resource needs
  - Issue M ≤ N instructions that can execute in parallel

Example Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op
Latency: LD:2, int:1, F*:1, F+:1

<table>
<thead>
<tr>
<th>cycle</th>
<th>instruction</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LD F0, 0(R1)</td>
<td>// a[i]</td>
</tr>
<tr>
<td>2</td>
<td>LD F2, 0(R2)</td>
<td>// b[i]</td>
</tr>
<tr>
<td>4</td>
<td>MULTD F8, F0, F2</td>
<td>// a[i] * b[i] (stall)</td>
</tr>
<tr>
<td>5</td>
<td>ADDD F12, F8, F16</td>
<td>// + c</td>
</tr>
<tr>
<td>6</td>
<td>SD F12, 0(R3)</td>
<td>// d[i]</td>
</tr>
<tr>
<td>7</td>
<td>ADDI R1, R1, 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ADDI R2, R2, 4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADDI R3, R3, 4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ADDI R4, R4, 1</td>
<td>// increment I</td>
</tr>
<tr>
<td>9</td>
<td>SLT R5, R4, R6</td>
<td>// i&lt;n-1</td>
</tr>
<tr>
<td>10</td>
<td>BNEQ R5, R0, LOOP</td>
<td></td>
</tr>
</tbody>
</table>

Old CPI = 12/11 = 1.09
New CPI = 10/11 = 0.91
Rescheduled for Multiple Issue

Issue rules: at most 1 LD/ST, at most 1 floating op
Latency: LD:2, int:1, F*:1, F+:1

Latency: LD:2, int:1, F*:1, F+:1

```
loop:
  ld f0, 0(r1) // a[i]
  addi r1, r1, 4
  ld f2, 0(r2) // b[i]
  addi r2, r2, 4
  multd f8, f0, f2 // a[i] * b[i]
  addi r4, r4, 1 // increment i
  addd f12, f8, f16 // + c
  slt r5, r4, r6 // i<n-1
  sd f12, 0(r3) // d[i]
  addi r3, r3, 4
  bneq r5, r0, loop
```

Old CPI = 0.91
New CPI = 7/11 = 0.64

The Problem with Static Scheduling

- In-order execution
  - an unexpected long latency blocks ready instructions from executing
  - binaries need to be rescheduled for each new implementation
  - small number of named registers becomes a bottleneck

```
lw r1, c //miss 50 cycles
lw r2, d
mul r3, r1, r2
sw r3, c
lw r4, b //ready
add r5, r4, r9
sw r5, a
lw r6, f
lw r7, g
add r8, r6, r7
sw r8, e
```
Dynamic Scheduling

- Determine execution order of instructions at run time
- Schedule with knowledge of run-time variable latency
  - cache misses
- Compatibility advantages
  - avoid need to recompile old binaries
  - avoid bottleneck of small named register sets
    - but still need to deal with spills
- Significant hardware complexity

Example

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|       | LD R1,A | I M X X X X X X X X X X C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | LD R2,B | I C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | MUL R3,R1,R2 | I X X C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | LD R4,C | I C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | LD R5,D | I C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | MUL R6,R5,R4 | I X X C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | ADD R7,R3,R8 | I X X C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|       | SD R7,E | I C |       |       |       |       |       |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- 10 cycle data memory (cache) miss
- 3 cycle MUL latency
- 2 cycle add latency