Lecture 14: Instruction Level Parallelism

• Last time
  - Pipelining in the real world
    - Control hazards
    - Other pipelines

• Today
  - Take QUIZ 10 over P&H 4.10-15, before 11:59pm today
  - Homework 5 due Thursday March 11, 2010
  - Instruction level parallelism
    - Multi-issue (Superscalar) and out-of-order execution
Where Are We?

Pipelined in-order processor
Simple branch prediction
Instruction/data caches (on-chip)

Out-of-order instruction execution
“Superscalar”
Sophisticated branch prediction

DEC Alpha 21064
Introduced in 1992

DEC Alpha 21264
Introduced 1998
Dynamic Multiple Issue (Superscalar)
No instruction reordering, Choose 0, 1 ... N
MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF</td>
</tr>
</tbody>
</table>
Hazards in Dual-Issue MIPS

• More instructions executing in parallel
• EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    • add $t0, $s0, $s1
    • load $s2, 0($t0)
    • Split into two packets, effectively a stall

• Load-use hazard
  - Still one cycle use latency, but now two instructions

• More aggressive scheduling required
What Hardware Do We Need?
What Hardware Do We Need?

• Wider fetch i-cache bandwidth
• Multiported register file
• More ALUs
• Restrictions on issue of load/stores because N ports to the data cache slows it down too much
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 $M \leq N$ instructions that can execute in parallel
Example Multiple Issue

Issue rules: at most 1 load/store, at most 1 floating op

Latency: load=1, int=1, float-mult = 2, float-add = 1

```assembly
LOOP:
  LD   F0, 0(R1)  // a[i]  1
  LD   F2, 0(R2)  // b[i]  2
  MULTD F8, F0, F2  // a[i] * b[i]  4 (stall)
  ADDD F12, F8, F16  // + c  5
  SD   F12, 0(R3)  // d[i]  6

  ADDI R1, R1, 4  7
  ADDI R2, R2, 4  
  ADDI R3, R3, 4  
  ADDI R4, R4, 1  // increment I  8
  SLT  R5, R4, R6  // i<n-1  9
  BNEQ R5, R0, LOOP 10
```

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 - 1, int-1, F*-2, F+1

```
LOOP:
    LD    F0, 0(R1)     // a[i]                      1
    ADDI  R1, R1, 4
    LD    F2, 0(R2)     // b[i]                      2
    ADDI  R2, R2, 4
    MULTD F8, F0, F2    // a[i] * b[i]                4
    ADDI  R4, R4, 1     // increment I
    ADDD  F12, F8, F16  // + c                        5
    SLT   R5, R4, R6    // i<n-1                      6
    SD    F12, 0(R3)    // d[i]                       6
    ADDI  R3, R3, 4
    BNEQ  R5, R0, LOOP
```

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

Given a two way issue processor, what’s the best possible CPI? IPC?
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

Top = without dynamic scheduling, Bottom = with dynamic scheduling

- 10 cycle data memory (cache) miss
- 3 cycle MUL latency
- 2 cycle add latency
Dynamic Scheduling
Basic Concept

Sequential Instruction Stream

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>R1,A</td>
</tr>
<tr>
<td>LW</td>
<td>R2,B</td>
</tr>
<tr>
<td>ADD</td>
<td>R3,R1,R2</td>
</tr>
<tr>
<td>SW</td>
<td>R3,C</td>
</tr>
<tr>
<td>LW</td>
<td>R4,8(A)</td>
</tr>
<tr>
<td>LW</td>
<td>R5,8(B)</td>
</tr>
<tr>
<td>ADD</td>
<td>R6,R4,R5</td>
</tr>
<tr>
<td>SW</td>
<td>R6,8(C)</td>
</tr>
<tr>
<td>LW</td>
<td>R7,16(A)</td>
</tr>
<tr>
<td>LW</td>
<td>R8,16(B)</td>
</tr>
<tr>
<td>ADD</td>
<td>R9,R7,R8</td>
</tr>
<tr>
<td>SW</td>
<td>R9,16(C)</td>
</tr>
<tr>
<td>LW</td>
<td>R10,24(A)</td>
</tr>
<tr>
<td>LW</td>
<td>R11,24(B)</td>
</tr>
</tbody>
</table>

Window of Waiting Instructions on operands & resources

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R3,R1,R2</td>
</tr>
<tr>
<td>SW</td>
<td>R3,C</td>
</tr>
<tr>
<td>ADD</td>
<td>R6,R4,R5</td>
</tr>
<tr>
<td>SW</td>
<td>R6,8(C)</td>
</tr>
<tr>
<td>LW</td>
<td>R7,16(A)</td>
</tr>
<tr>
<td>LW</td>
<td>R8,16(B)</td>
</tr>
<tr>
<td>ADD</td>
<td>R9,R7,R8</td>
</tr>
<tr>
<td>SW</td>
<td>R9,16(C)</td>
</tr>
</tbody>
</table>

Execution Resources

Register File

Instructions waiting to commit

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>R4,8(A)</td>
</tr>
<tr>
<td>LW</td>
<td>R5,8(B)</td>
</tr>
</tbody>
</table>

Issue Logic
Implementation I - Register Scoreboard

- Tracks register writes
  - busy = pending write
- Detect hazards for scheduler

**Register File**

<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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**What about:**

- ADD R3, R1, R2
  - Wait until R1 is valid
  - Mark R3 valid when complete
- SUB R4, R0, R3
  - Wait for R3

**LD**
- R3, (0) R7
- ADD R4, R3, R5
- LD R3, (4) R7
Implementing A Simple Instruction Window

Often called reservation stations
reg = name, value

Result sequence: R4, R7, R5, R1, R6, R3
Instruction Window Policies

• Add an instruction to the window
  - only when dest register is not busy
  - mark destination register busy
  - check status of source registers and set ready bits

• When each result is generated
  - compare dest register field to all waiting instruction source register fields
  - update ready bits
  - mark dest register not busy

• Issue an instruction when
  - execution resource is available
  - all source operands are ready

• Result
  - issues instructions out of order as soon as source registers are available
  - allows only one operation in the window per destination register
Register Renaming (1)

What about this sequence?

\[
\begin{align*}
\text{LW} & \quad R1, \ 0(R4) \\
\text{ADD} & \quad R2, \ R1, \ R3 \\
\text{LW} & \quad R1, \ 4(R4) \\
\text{ADD} & \quad R5, \ R1, \ R3
\end{align*}
\]

Can’t add 3 to the window since R1 is already busy
Need 2 R1s!
Register Renaming (2)

Add a tag field to each register
- translates from virtual to physical register name

In window
LW R1, 0(R4)
ADD R2, R1, R3

Next instruction
LW R1, 4(R4)
Register Renaming (3)

Add instruction to window even if dest register is busy

When adding instruction to window
- read data of non-busy source registers and retain
- read tags of busy source registers and retain
- write tag of destination register with slot number

When result is generated:
- compare tag of result to not-ready source fields
- grab data if match

LW R1, 0 (R4)
ADD R2, R1, R3
LW R1, 4 (R4)
ADD R5, R1, R3
Power Efficiency

• **Complexity of dynamic scheduling and speculations requires power**
• **Multiple simpler cores may be better**

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>
Summary

• Summary
  - Pipelining is simple, but a correct high performance implementation is complex
  - Dynamic multiple issue
  - Static multiple issue (VLIW)
  - Out-of-order execution - dependencies, renaming, etc.

• Next Time
  - Caches (new topic!)
  - Homework 5 due Thursday March 11, 2010
  - Read: P&H 5.1-5