Computer Systems Organization

CSE 521/560M
Lecture 12
Prof. Patrick Crowley

Plan for Today

• HW1 & HW2
• Thoughts on commentary
• Questions
• Today’s discussion
Compiler Techniques for Exposing ILP

- It is best to fill the pipeline with independent instructions.
- In general, separate dependent instructions from their sources by $n$ instructions, where $n$ is the latency of the source instructions.

Sample Code

For (i=1000; i>0; i=i-1)

\[
x[i] = x[i] + s
\]

Loop:

- L.D F0,0(R1)
- ADD.D F4,F0,F2
- S.D F4,0(R1)
- DADDUI R1,R1,#-8
- BNE R1,R2,Loop
Machine Characteristics

- 5 stage integer pipeline
  - Branches have 1 cycle delay (branch delay slot)
- Floating point latencies
  - F-p Op → f-p Op: 3
  - F-p Op → S.D: 2
  - L.D → f-p Op: 1
  - L.D → S.D: 0

Original Loop Schedule

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Cycle issued</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
</tr>
<tr>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>ADD,D</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1,R1,#-8</td>
</tr>
<tr>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, Loop</td>
</tr>
<tr>
<td>Stall</td>
<td></td>
</tr>
</tbody>
</table>
Better Loop Schedule

Loop:  
L.D  
DADDUI  
ADD.D  
stall  
BNE  
S.D  
F0, 0 (R1)  
R1, R1, #–8  
F4, F0, F2  

- Only 1 stall remains
- Iteration time reduced from 10 to 6 cycles
- Needed to
  - Change store address

Critical Instructions

Loop:  
L.D  
DADDUI  
ADD.D  
stall  
BNE  
S.D  
F0, 0 (R1)  
R1, R1, #–8  
F4, F0, F2  
F4, 8 (R1)  

- This dependence chain determines loop latency
- Other three constitute loop overhead
Loop Unrolling

Loop Unrolling & Scheduling

Loop: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
L.D F6, -8(R1)
ADD.D F8, F6, F2
S.D F8, -8(R1)
DADDUI R1, R1, #16
BNE R1, R2, Loop

- Amortize loop overhead
- Update load/store addresses
- Update array bound check
- What is the loop duration?

Loop: L.D F0, 0(R1)
L.D F6, -8(R1)
ADD.D F4, F0, F2
ADD.D F8, F6, F2
DADDUI R1, R1, #16
S.D F4, 16(R1)
BNE R1, R2, Loop
S.D F8, 8(R1)

- What is the loop duration?
Limits and Tradeoffs

- Diminishing returns from loop unrolling
- Code size increases with loop unrolling
  - Embedded systems
  - Instruction cache performance
- Register allocation
  - Each unrolled loop iteration contributes live values, creates *register pressure*

Summarized Compiler/Programmer Activities

1. Determine that loop iterations are independent
2. Use different registers for each iteration
3. Remove extra loop termination instructions
4. Schedule the code according to instruction latencies while preserving dependences
5. Re-order loads and stores from different iterations (memory alias analysis)
6. Update index calculations and load and store offsets
Static Multiple Issue

<table>
<thead>
<tr>
<th>Integer</th>
<th>F-P</th>
<th>Clock Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>L.D</td>
<td>F6,~8(R1)</td>
<td>2</td>
</tr>
<tr>
<td>L.D</td>
<td>F10,~16(R1)</td>
<td>3</td>
</tr>
<tr>
<td>L.D</td>
<td>F14,~24(R1)</td>
<td>4</td>
</tr>
<tr>
<td>L.D</td>
<td>F18,~32(R1)</td>
<td>5</td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
<td>5</td>
</tr>
<tr>
<td>S.D</td>
<td>F8,~8(R1)</td>
<td>6</td>
</tr>
<tr>
<td>S.D</td>
<td>F12,~16(R1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADDU</td>
<td>R1,R1,#-40</td>
<td>9</td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,Loop</td>
<td>10</td>
</tr>
<tr>
<td>S.D</td>
<td>F12,~16(R1)</td>
<td>11</td>
</tr>
</tbody>
</table>

- Technique can be used on a statically-scheduled superscalar as well
- Unrolled 5 times
- Loop duration is 12 cycles

Using Static Branch Prediction

LD  R1,0(R2)
stall
DSUBU  R1,R1,R3
BEQZ  R1,L
OR  R4,R5,R6
DADDU  R10,R4,R3

L:  DADDU  R7,R8,R8

- Fill stall with an instruction
  - Fall-through?
  - Branch target?
- Slot can be utilized if branch outcome biased in one direction
Profile-driven Static Prediction

![Bar chart showing misprediction rate for different benchmarks]

VLIW Approach to ILP

- Organize program into groups of independent instructions
  - All instructions in a group can issue at the same time
  - No need for dynamic issue/scheduling decisions
- Emphasis on
  - Simple hardware
  - Sophisticated compiler
  - Wide-issue machines (where hw overhead is greatest)
Sample VLIW Machine

- Each ‘wide’ instruction has 5 operations
  - 1 integer (or branch)
  - 2 f-p
  - 2 memory references
- Instruction size
  - Example: 3*32 + 2*24 = 144 bits

Sources of Parallelism

- Instructions come from a traditional code sequences
  - programmers still use traditional programming languages
- Use loop unrolling and code scheduling
  - Local scheduling used between branches
  - Global scheduling used across branches (we’ll look at trace scheduling later this week)
## Instruction Scheduling Example

<table>
<thead>
<tr>
<th>MEM 1</th>
<th>MEM 2</th>
<th>FP 1</th>
<th>FP 2</th>
<th>Int/branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,8(R1)</td>
<td>L.D F10,16(R1)</td>
<td>L.D F14,24(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D F18,32(R1)</td>
<td>L.D F22,40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
</tr>
<tr>
<td>L.D F26,48(R1)</td>
<td></td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td>S.D F8,8(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
</tr>
<tr>
<td>S.D 12,16(R1)</td>
<td>S.D F16,24(R1)</td>
<td></td>
<td></td>
<td>DADDUI R1,R1,#-56</td>
</tr>
<tr>
<td>S.D F20,24(R1)</td>
<td>S.D F24,32(R1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F28,8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>BNE R1,R2,Loop</td>
</tr>
</tbody>
</table>

- Unrolled 7 times to eliminate all stalls
- Loop duration is 9 cycles
- Is this good resource utilization?

## Assignment

- **Readings**
  - For Wednesday
    - H&P: 4.1-4.4
  - For Monday
    - H&P: 4.5-4.6
    - Ash: Skim Ch. 8, Read Ch. 9