Lecture 8: Eliminating Stalls Using Compiler Support

Instruction Level Parallelism

- gcc 17% control transfer
  - 5 instructions + 1 branch
  - Reordering among 5 instructions may not uncover enough instruction level parallelism to eliminate all stalls
  - To eliminate remaining stalls we must look beyond single block and find more instruction level parallelism
- Loop level parallelism one opportunity
- Illustrate the above using DLX with Floating Point as an example
FP Loop: Where are the Hazards?

Instruction | Instruction | Latency in producing result | using result | clock cycles
---|---|---|---|---
FP ALU op | Another FP ALU op | 3
FP ALU op | Store double | 2
Load double | FP ALU op | 1
Load double | Store double | 0
Integer op | Integer op | 0

FP Loop Hazards

Instruction | Instruction | Latency in producing result | using result | clock cycles
---|---|---|---|---
FP ALU op | Another FP ALU op | 3
FP ALU op | Store double | 2
Load double | FP ALU op | 1
Load double | Store double | 0
Integer op | Integer op | 0

- Where are the stalls?
**FP Loop Showing Stalls**

1. Loop:  
   
   ```
   LD F0,0(R1) ; F0 = vector element
   ```

2. Stall

3. ADDD F4,F0,F2 ; add scalar in F2

4. Stall

5. Stall

6. SD 0(R1),F4 ; store result

7. SUBI R1,R1,8 ; decrement pointer 8B (DW)

8. BNEZ R1,Loop ; branch R1!=zero

9. Stall ; delayed branch slot

- **Rewrite code to minimize stalls?**

---

**Revised FP Loop Minimizing Stalls**

1. Loop:  
   
   ```
   LD F0,0(R1)
   ```

2. Stall

3. ADDD F4,F0,F2

4. SUBI R1,R1,8

5. BNEZ R1,Loop ; delayed branch

6. SD 8(R1),F4 ; altered when move past SUBI

- **Unroll loop 4 times code to make faster?**
### Unroll Loop Four Times

1. Loop: 
   - LD \( F_0, 0(R_1) \)
   - ADDD \( F_4, F_0, F_2 \)
   - SD \( 0(R_1), F_4 \) : drop SUBI & BNEZ
   - LD \( F_6, -8(R_1) \)
   - ADDD \( F_8, F_6, F_2 \)
   - SD \( -8(R_1), F_8 \) : drop SUBI & BNEZ
   - LD \( F_{10}, -16(R_1) \)
   - ADDD \( F_{12}, F_{10}, F_2 \)
   - SD \( -16(R_1), F_{12} \) : drop SUBI & BNEZ
   - LD \( F_{14}, -24(R_1) \)
   - ADDD \( F_{16}, F_{14}, F_2 \)
   - SD \( -24(R_1), F_{16} \)
   - SUBI \( R_1, R_1, #32 \) : alter to \( 4*8 \)
   - BNEZ \( R_1, LOOP \)
   - NOP

\[15 + 4 \times (1+2) = 27 \text{ clock cycles, or } 6.8 \text{ per iteration}\]

Assumes \( R_1 \) is multiple of 4

---

### Unrolled Loop That Minimizes Stalls

1. Loop: 
   - LD \( F_0, 0(R_1) \)
   - LD \( F_6, -8(R_1) \)
   - LD \( F_{10}, -16(R_1) \)
   - LD \( F_{14}, -24(R_1) \)
   - ADDD \( F_4, F_0, F_2 \)
   - ADDD \( F_8, F_6, F_2 \)
   - ADDD \( F_{12}, F_{10}, F_2 \)
   - ADDD \( F_{16}, F_{14}, F_2 \)
   - SD \( 0(R_1), F_4 \)
   - SD \( -8(R_1), F_8 \)
   - SD \( -16(R_1), F_{12} \)
   - SUBI \( R_1, R_1, #32 \) : alter to \( 4*8 \)
   - BNEZ \( R_1, LOOP \)
   - SD \( 8(R_1), F_{16} \) ; \( 8-32 = -24 \)

14 clock cycles, or 3.5 per iteration
Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP op. 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F6,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td>4</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F6</td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td>SUBI R1,R1,#48</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>SD -0(R1),F28</td>
<td>BNEZ R1,LOOP</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

- Unrolled 7 times to avoid delays
- 7 results in 9 clocks, or 1.3 clocks per iteration
- Need more registers in VLIW

Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (- Tomasulo in SW)
SW Pipelining Example

Symbolic Loop Unrolling

- Less code space
- Overhead paid only once vs. each iteration in loop unrolling

100 iterations = 25 loops with 4 unrolled iterations each
Compile-time Analysis

- Compiler analysis is performed to detect data dependences.
- Further analysis is performed to identify stalls (must have knowledge of the HW).
- Unroll loop and reorder code to eliminate stalls.

Compiler Perspective on Data Dependences

- **Flow dependence** (RAW hazard for HW)
  - Instruction \( j \) writes a register or memory location that instruction \( i \) reads from and instruction \( j \) is execution first.
- **Anti-dependence** (WAR hazard for HW)
  - Instruction \( j \) writes a register or memory location that instruction \( i \) reads from and instruction \( i \) is executed first.
- **Output dependence** (WAW hazard for HW)
  - Instruction \( i \) and instruction \( j \) write the same register or memory location; ordering between instructions must be preserved.
Dependency Analysis

• Easy to determine for registers
  – By looking at fixed register names dependences can be easily found

• For memory in some cases it is easy but in general it can be hard
  – From same iteration
    0(R1) != -8(R1) != -16(R1) != -24(R1)
  – From different loop iterations
    20(R6) != 20(R6) if R6 has changed
  – Is 100(R4) = 20(R6)? If references are to two different arrays there is no dependence. But in general this is hard to determine.

• Unroll loop if instructions from different iterations are not dependent upon each other.

Dependence Analysis

• Final kind of dependence called control dependence

• Example
  
  if p1 {S1;}
  if p2 {S2;}

  S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

• Strict enforcement of control dependences limits parallelism
  – unrolling eliminated conditional branches to overcome this limitation.
Summary

- Instruction Level Parallelism can be uncovered by the compiler.
- Loops are an important source of instruction level parallelism.
- Dependency analysis is key to uncovering instruction level parallelism.