FP Loop Example

♦ Add a scalar to a vector:
  for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;

Where are the Hazards?

• First translate into MIPS code:
  - To simplify, assume 8 is lowest address

Loop:
  L.D F0,0(R1); F0=vector element
  ADD.D F4,F0,F2; add scalar from F2
  S.D 0(R1),F4; store result
  DSUBUI R1,R1,8; decrement pointer 8B (DW)
  BNEZ R1,Loop; branch R1!=zero
  NOP; delayed branch slot

Where are the stalls?
FP Loop Showing Stalls

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
</tbody>
</table>

1. Loop: L.D F0, 0(R1); F0 = vector element
2. stall
3. ADD.D F4, F0, F2; add scalar in F2
4. stall
5. stall
6. S.D 0(R1), F4; store result
7. DSUBUI R1, R1, 8; decrement pointer 8B (DW)
8. BNEZ R1, Loop; branch R1! = zero
9. stall; delayed branch slot

♦ 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

1. Loop: L.D F0, 0(R1)
2. stall
3. ADD.D F4, F0, F2
4. DSUBUI R1, R1, 8
5. BNEZ R1, Loop; delayed branch
6. S.D 8(R1), F4; altered when move past DSUBUI

Move S.D after BNEZ by changing address of S.D

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
</tbody>
</table>

6 clocks, but just 3 for execution, 3 for loop overhead; How can we make it faster?
Unroll Loop Four Times (straightforward way)

1 Loop: L.D F0, 0(R1)  
2 ADD.D F4, F0, F2  
3 S.D 0(R1), F4 ; drop DSUBUI & BNEZ  
4 L.D F6, -8(R1)  
5 ADD.D F8, F6, F2  
6 S.D -8(R1), F8 ; drop DSUBUI & BNEZ  
7 L.D F10, -16(R1)  
8 ADD.D F12, F10, F2  
9 S.D -16(R1), F12 ; drop DSUBUI & BNEZ  
10 L.D F14, -24(R1)  
11 ADD.D F16, F14, F2  
12 S.D -24(R1), F16  
13 DSUBUI R1, R1, #32 ; alter to 4*8  
14 BNEZ R1, LOOP  
15 NOP

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration  
Assumes R1 is multiple of 4

Unrolled Loop Details

♦ Do not usually know upper bound of loop (at compile time)
♦ Suppose it is n, and we would like to unroll the loop to make k copies of the body
♦ Instead of a single unrolled loop, we generate a pair of consecutive loops:
  • 1st executes (n mod k) times and has a body that is the original loop
  • 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
  • For large values of n, most of the execution time will be spent in the unrolled loop
Unrolled Loop That Minimizes Stalls

1. Loop: L.D F0,0(R1)
2. L.D F6,-8(R1)
3. L.D F10,-16(R1)
4. L.D F14,-24(R1)
5. ADD.D F4,F0,F2
6. ADD.D F8,F6,F2
7. ADD.D F12,F10,F2
8. ADD.D F16,F14,F2
9. S.D 0(R1),F4
10. S.D -8(R1),F8
11. S.D -16(R1),F12
12. DSUBUI R1,R1,#32
13. BNEZ R1,LOOP
14. S.D 8(R1),F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration

 Compiler Perspectives on Code Movement

♦ Compiler concerned about potential dependencies in program
  ♦ Whether or not a HW hazard depends on pipeline
  ♦ (True) Data dependencies (RAW if a hazard for HW)
    ♦ Instruction i produces a result used by instruction j, or
    ♦ Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i
  ♦ If dependent, can not execute in parallel
  ♦ Easy to determine for registers (fixed names)
  ♦ Hard for memory ("memory disambiguation" problem):
    ♦ Does 100(R4) = 20(R6)?
    ♦ From different loop iterations, does 20(R6) = 20(R6)?
  ♦ Our example required compiler to know that if R1 doesn’t change then:
    0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)
Steps Compiler Performed to Unroll

- Check if OK to move the S.D after DSUBUI and BNEZ, and calculate amount to adjust S.D offset
- Determine unrolling the loop would be OK by finding that the loop iterations are independent (GCD test)
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Schedule the code, preserving any true dependencies

When Safe to Unroll Loop?

- Example: A,B,C distinct & non-overlapping
  ```c
  for (i=0; i<100; i=i+1) {
    A[i+1] = A[i] + C[i];    /* S1 */
    B[i+1] = B[i] + A[i+1];  /* S2 */
  }
  ```
  Where are data dependencies?
  1. S2 uses the value, A[i+1], computed by S1 in same iteration
  2. S1 uses a value computed by S1 in an earlier iteration.
  The same is true of S2 for B[i]
  This is a “loop-carried dependence”: between iterations

- Greatest Common Divisor (GCD) test
  ```c
  a j + b = c k + d;  a j - c k = d - b;  denote x=gcd(a,c) then
  a= xy and c= xz;  y(\text{j} - zk) = d - b  \Rightarrow  yz = (d - b) / x
  (yz) is an integer only if x=gcd(a,c) divides (d-b)
  ```
  ```c
  for (i=0; i<100; i=i+1) {
  }
  ```
Another possibility: Software Pipelining

♦ Observation: if iterations from loops are independent, we can get more 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)

Software Pipelining Example

Before: Unrolled 3 times
1 L.D 0(R1)
2 ADD.D F0,F0,F2
3 S.D 0(R1),F4
4 L.D F6,-8(R1)
5 ADD.D F8,F6,F2
6 S.D -8(R1),F8
7 L.D F10,-16(R1)
8 ADD.D F12,F10,F2
9 S.D -16(R1),F12
10 DSUBUI R1,R1,#24
11 BNEZ R1,LOOP

After: Software Pipelined
1 S.D 0(R1),F4; Stores M[i]
2 ADD.D F4,F0,F2; Adds to M[i-1]
3 L.D F0,-16(R1); Loads M[i-2]
4 DSUBUI R1,R1,#8
5 BNEZ R1,LOOP

• "Symbolic" Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Avoid structural hazards

5 cycles per iteration
Another alternative: Change Instruction Set

- Superscalar processors decide on the fly how many instructions to issue (up to n)
  - HW complexity $O(n^2)$ - must limit n
- Why not allow compiler to schedule instruction level parallelism explicitly?
- Format the instructions in a potential issue packet so that HW need not check for dependencies

VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
- Tradeoff instruction space for simple decoding/issuing
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => can execute in parallel
  - E.g., 1 integer operations, 2 FP ops, 2 Memory refs
    - 24 bits per field => 5*24=120 bits wide
  - Need compiling technique that schedules across several branches
Reduce number of branches by unrolling loops

1 Loop:  
1. L.D  F0,0(R1)  
2. L.D  F6,-8(R1)  
3. L.D  F10,-16(R1)  
4. L.D  F14,-24(R1)  
5. ADD.D  F4,F0,F2  
6. ADD.D  F8,F6,F2  
7. ADD.D  F12,F10,F2  
8. ADD.D  F16,F14,F2  
9. S.D  0(R1),F4  
10. S.D  -8(R1),F8  
11. S.D  -16(R1),F12  
12. DSUBUI  R1,R1,#32  
13. BNEZ  R1,LOOP  
14. S.D  8(R1),F16  ;  8-32 = -24

L.D to ADD.D: 1 Cycle  
ADD.D to S.D: 2 Cycles

Loop Unrolling in VLIW (5 ops per packet)

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D  F0,0(R1)</td>
<td>L.D  F6,-8(R1)</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D  F10,-16(R1)</td>
<td>L.D  F14,-24(R1)</td>
<td></td>
<td>2</td>
<td></td>
<td>2</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>3</td>
<td>ADD.D  F8,F6,F2</td>
<td>3</td>
</tr>
<tr>
<td>L.D  F26,-48(R1)</td>
<td>L.D  F28,-56(R1)</td>
<td>ADD.D  F12,F10,F2</td>
<td>4</td>
<td>ADD.D  F16,F14,F2</td>
<td>4</td>
</tr>
<tr>
<td>S.D  0(R1),F4</td>
<td>S.D  -8(R1),F8</td>
<td>ADD.D  F20,F18,F2</td>
<td>5</td>
<td>ADD.D  F24,F22,F2</td>
<td>5</td>
</tr>
<tr>
<td>S.D  -16(R1),F12</td>
<td>S.D  -24(R1),F16</td>
<td></td>
<td>6</td>
<td>DSUBUI  R1,R1,#32</td>
<td>6</td>
</tr>
<tr>
<td>S.D  -32(R1),F20</td>
<td>S.D  -40(R1),F24</td>
<td></td>
<td>7</td>
<td>BNEZ  R1,LOOP</td>
<td>7</td>
</tr>
<tr>
<td>S.D  -0(R1),F28</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid stalls
7 results in 9 clocks, or 1.3 clocks per iteration (2.3X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SuperScalar)
Software Pipelining with Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,-48(R1)</td>
<td>ST 0(R1),F4</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F6,-56(R1)</td>
<td>ST -8(R1),F8</td>
<td>ADD.D F20,F18,F2</td>
<td>DSUBUI R1,R1,#24</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>L.D F10,-40(R1)</td>
<td>ST 8(R1),F12</td>
<td>ADD.D F24,F22,F2</td>
<td>BNEZ R1,LOOP</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Software pipelined across 9 iterations of original loop

- In each “iteration” (3 cycles) of above loop, we:
  - Store to \( m,m-8,m-16 \) (iterations I-3,I-2,I-1)
  - Compute for \( m-24,m-32,m-40 \) (iterations I,I+1,I+2)
  - Load from \( m-48,m-56,m-64 \) (iterations I+3,I+4,I+5)

- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

Note: Need fewer registers for software pipelining (only using 12 registers here, was using 15)

---

HW (Superscaler w/Tomasulo) vs. SW (VLIW)

- **HW advantages:**
  - HW better at memory disambiguation since knows actual addresses
  - HW better at branch prediction with low overhead
  - HW maintains precise exception model
  - Same software works across multiple implementations
  - Smaller code size (not as many nops filling blank instructions)

- **SW advantages:**
  - Window of instructions that is examined for parallelism much bigger
  - Speculation can be based on large-scale program behavior, not just local information
  - Much less hardware involved in VLIW (for issuing instructions)
  - More involved types of speculation can be done
VLIW Main Drawbacks

- **Increase in code size**
  - Generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - Whenever VLIW instructions are not full, unused functional units and wasted bits in instruction
- **Binary code incompatibility**
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **IA-64**: instruction set architecture
- **Itanium™** is name of first implementation (2001)
- **IA-64** instructions encoded in 128-bit wide bundles
  - Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length
- 128 64-bit integer reg + 128 82-bit floating point registers
- Hardware checks dependencies
- Predicated execution
- Integer registers configured to accelerate procedure calls using a register stack
  - mechanism similar to that used in SPARC architecture
  - Registers 0-31 are always accessible and addressed as 0-31
  - Registers 32-127 are used as a register stack and each procedure is allocated a set of registers
SPARC Register Window Mechanism

SPARC Register Window Mechanism

Itanium™ EPIC: 10 Stage Pipeline