Software Techniques - Example

- This code, add a scalar to a vector:
  
  for (i=1000; i>0; i=i–1)
  x[i] = x[i] + s;
- Assume following latencies for all examples
  - Ignore delayed branch in these examples

### Instruction Latency

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in cycles</th>
<th>stalls between using result in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

FP Loop: Where are the Hazards?

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

for (i=1000; i>0; i=i–1)
  x[i] = x[i] + s;

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

9 clock cycles: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

1 Loop: L.D F0,D(R1)
2 DADDUI R1,R1,-8
3 ADD D F4,F0,F2
4 stall
5 stall
6 S.D 8(R1),F4; altered offset when move DADDUI
7 BNEZ R1,Loop

Swap DADDUI and S.D by changing address of S.D

7 clock cycles, but just 3 for execution (L.D, ADD,D,S.D), 4 for loop overhead; How make 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
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. L.D F14,-24(R1)
11. ADD.D F16,F14,F2
12. S.D -24(R1),F16
13. DADDUI R1,R1,#-32
14. BNEZ R1,LOOP

27 clock cycles, or 6.75 per iteration (Assumes R1 is a multiple of 4)

Rewrite loop to minimize stalls?

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. DADDUI R1,R1,#-32
14. DADDUI R1,R1,#-32
15. S.D 0(R1),F4
16. S.D -8(R1),F8
17. S.D -16(R1),F12
18. ADD.D F4,F0,F2
19. ADD.D F8,F6,F2
20. ADD.D F12,F10,F2
21. ADD.D F16,F14,F2
22. DADDUI R1,R1,#-32
23. BNEZ R1,LOOP

14 clock cycles, or 3.5 per iteration

Unrolled Loop Detail

- Do not usually know upper bound of loop
- 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

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
  1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
  2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
  3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
  4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
  5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

- Decrease in amount of overhead amortized with each extra unrolling
  - Amdahl's Law
- Growth in code size
  - For larger loops, concern it increases the instruction cache miss rate
- Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
  - If not be possible to allocate all live values to registers, may lose some or all of its advantage
- Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

5 Loop Unrolling Decisions (cont.)