Scheduling example for dual-issue MIPS

**Schedule** this code for dual-issue MIPS

Loop:

- `lw $t0, 0($s1)`  # $t0=array element
- `add $t0, $t0, $s2`  # add scalar in $s2
- `sw $t0, 0($s1)`  # store result
- `addi $s1, $s1,-4`  # decrement pointer
- `bne $s1, $zero, Loop`  # branch $s1!=0

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td><code>lw $t0, 0($s1)</code></td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td><code>nop</code></td>
<td>2</td>
</tr>
<tr>
<td>add $t0, $t0, $s2</td>
<td><code>nop</code></td>
<td>3</td>
</tr>
<tr>
<td>addi $s1, $s1,-4</td>
<td><code>sw $t0, 0($s1)</code></td>
<td>4</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td><code>nop</code></td>
<td>5</td>
</tr>
</tbody>
</table>

- IPC = 5/5 = 1 (c.f. peak dual-issue IPC = 2 and single-issue IPC = 5/6 = 0.83 for single-issue pipeline)
Limits to ILP: data dependencies

• Data dependencies determine:
  – Order which results should be computed
  – Possibility of hazards
  – Degree of freedom in scheduling instructions

=> limit to ILP

• Data/Name dependency hazards:
  – Read After Write (RAW)
  – Write After Read (WAR)
  – Write After Write (WAW)
Data dependency: RAW

- A true data dependency because values are transmitted between the instructions
- Dependency is clear when registers are involved – less obvious when memory is involved. Alias analysis is required for memory
Name dependency (antidependence): WAR

- lw $s0, 0($t0)
- ...
- add $t0, $s1, $s2
- ...
- ...
- add $s4, $s2, $s0
- ..... 
- sub $s2, $s1, $s3
- lw $t2, 0($s2)
- ....
- lw $s2, 4($t0)

- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)
Name dependency (output dependency): WAW

- Just a name dependency – no values being transmitted
- Dependency can be removed by renaming registers (either by compiler or HW)
Impact of branches on data flow

- **Data flow:** actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data

- **Example:**
  
  ```
  add $s2,$s1,$s0
  beq $s2, $t2, L
  sub $s2,$s3,$s4
  L: ...
  or $s6,$s2,$s5
  ```

- or depends on `add` or `sub`? Must preserve data flow on execution.
- Willing to execute instructions that should not have been executed, thereby violating the control dependences, **if** can do so without affecting correctness of the program
Re-schedule example for dual-issue MIPS

- **Re-Schedule** this code for dual-issue MIPS

  Loop:  lw $t0, 0($s1)       # $t0=array element
         add $t0, $t0, $s2     # add scalar in $s2
         sw $t0, 0($s1)        # store result
         addi $s1, $s1, -4     # decrement pointer
         bne $s1, $zero, Loop  # branch $s1!=0

<table>
<thead>
<tr>
<th></th>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>addi $s1, $s1, -4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>add $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

- **IPC = 5/4 = 1.25** (c.f. peak IPC = 2)
Exposing ILP using loop unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
    - Reuse of a register name
Loop unrolling example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: addi $s1, $s1, −16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>add $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>add $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>add $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>add $t3, $t3, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

- IPC = 14/8 = 1.75
- Closer to 2, but at cost of registers and code size
Limits to loop unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
2. Growth in code size (might not fit in instruction memory cache)
3. Register pressure: loop unrolling increase demands on registers and they are few of them to begin with.
Summary of VLIW architectures

• **Advantages:**
  – Simplified HW for management of hazards and scheduling (important for power and cost)
  – Works well in data-intensive applications with little control

• **Disadvantages:**
  – Some hazards can’t be resolved during compile time
  – Poor portability and backward compatibility

• Found a niche in embedded market (e.g., DSPs)