Static, multiple-issue (superscaler) pipelines

Start more than one instruction in the same cycle

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Pipe stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU or branch instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Load or store instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Load or store instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>ALU or branch instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Load or store instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>ALU or branch instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Load or store instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

A static two-issue datapath
Forwarding paths in a superscaler

Superscaler execution (two pipelines)

- A “super-instruction” is actually two MIPS instructions one to execute on the ALU/branch pipe and the other on the load/store pipe

- The compiler packs super instructions – may use a no-op in a super-instruction if cannot find two suitable instructions.

- Should make sure that there is no data hazards (by inserting no-ops)
  - Fewer no-ops inserted if the architecture supports forwarding
  - More no-ops inserted if hardware does not support forwarding

- In each cycle a super-instruction is fetched and its two instructions are pushed through the two pipelines.

- Effectiveness depends on the ability of the compiler to pack two instructions into every super-instruction.
Loop scheduling on a super-scalar pipeline

**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 if $s1 != 0
```

**An equivalent code:**

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

**A schedule on two pipelines (with hardware support for forwarding):**

<table>
<thead>
<tr>
<th>ALU or bne instructions</th>
<th>lw/sw instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $s1, $s1, -4</td>
<td>lw  $t0, 0($s1)</td>
</tr>
<tr>
<td>add $t0, $t0, $s2</td>
<td>cycle 2 cycle 3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw  $t0, 4($s1)</td>
</tr>
</tbody>
</table>

- Takes 4 cycles to execute one iteration (assuming perfect branch prediction)

Loop unrolling

**Loop:**

```
lw  $t0, 0($s1)
add $s1, $s1, -4
add $t0, $t0, $s2
sw  $t2, 4($s1)
```

**Loop (unrolled):**

```
lw  $t0, 0($s1)
add $t0, $t0, $s2
sw  $t2, 4($s1)
```

- 5 instructions per iteration
- 8 instructions per two iterations

- Duplicate the body of the loop (lw, add, sw) using $t1, a register different than $t0
- Update the loop index only once (subtract 8 rather than 4 from $s1)
- Change the constants (offsets) to reflect the new values of the loop index.

- Advantages: fewer total executed instructions (less overhead for loop control)
- Disadvantages: use more registers
- Problem: what if the number of iterations is not even?
Scheduling the unrolled loop

Loop:  
\[
\begin{align*}
\text{lw} & \quad \text{t0, 0(s1)} \\
\text{lw} & \quad \text{t1, -4(s1)} \\
\text{add} & \quad \text{s1, s1, -8} \\
\text{add} & \quad \text{t0, t0, s2} \\
\text{add} & \quad \text{t1, t1, s2} \\
\text{sw} & \quad \text{t0, 8(s1)} \\
\text{sw} & \quad \text{t1, 4(s1)} \\
\text{bne} & \quad \text{s1, zero, Loop}
\end{align*}
\]

ALU or bne instructions | lw/sw instructions |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: addi s1, s1, -8</td>
<td>lw t0, 0(s1)</td>
</tr>
<tr>
<td>add t0, t0, s2</td>
<td>lw t1, -4(s1)</td>
</tr>
<tr>
<td>add t1, t1, s2</td>
<td>sw t0, 8(s1)</td>
</tr>
<tr>
<td>bne s1, zero, Loop</td>
<td>sw t1, 4(s1)</td>
</tr>
</tbody>
</table>

5 cycles per two iterations (ignoring control hazards)

Unrolling 4 times

Loop:  
\[
\begin{align*}
\text{addi} & \quad \text{s1, s1, -16} \\
\text{lw} & \quad \text{t0, 0(s1)} \\
\text{lw} & \quad \text{t1, 12(s1)} \\
\text{add} & \quad \text{t0, t0, s2} \\
\text{add} & \quad \text{t1, t1, s2} \\
\text{add} & \quad \text{t2, t2, s2} \\
\text{add} & \quad \text{t3, t3, s2} \\
\text{bne} & \quad \text{s1, zero, Loop}
\end{align*}
\]

ALU or bne instructions | lw/sw instructions |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: addi s1, s1, -16</td>
<td>lw t0, 0(s1)</td>
</tr>
<tr>
<td>add t0, t0, s2</td>
<td>lw t1, 12(s1)</td>
</tr>
<tr>
<td>add t1, t1, s2</td>
<td>lw t2, 8(s1)</td>
</tr>
<tr>
<td>add t2, t2, s2</td>
<td>lw t3, 4(s1)</td>
</tr>
<tr>
<td>add t3, t3, s2</td>
<td>sw t0, 16(s1)</td>
</tr>
<tr>
<td>bne s1, zero, Loop</td>
<td>sw t1, 12(s1)</td>
</tr>
</tbody>
</table>

8 cycles per four iterations (ignoring control hazards)

- Is there a limitation on the number of times we can unroll?
- How will the schedule change if the hardware does not support forwarding and stalling?
Hazards in the Dual-Issue MIPS

- More instructions executing in parallel cause more hazards
- **Data hazard**
  - Even with forwarding paths between the two pipelines
    - Can’t use ALU result in load/store in same packet
      ```
      add $t0, $s0, $s1
      lw $s2, 0($t0)
      ```
    - Load-use hazard
      ```
      lw $s2, 0($t0)
      add $t0, $s2, $s1
      ```
  - Schedule in two packets separated by at least one cycle
- **Control Hazard**
  - Penalty for a wrong branch is proportional to issue width.
  - Example:
    - if branch is resolved in EX stage, then 4 instructions have to be squashed in case of a branch misprediction.
    - 5 instruction have to be squashed if the instruction after the branch is issued at the same cycle as the branch.

Dynamic Scheduling

- **Static scheduling**
  - Schedule instructions at compile time to get the best execution time
- **Dynamic scheduling**
  - Hardware changes instruction execution sequence in order to minimize execution time (out of order execution)
  - Dependences must be honored
    (read after write, write after read and write after write).
  - For the best result, control dependences must be tackled
- **Components of dynamic scheduling**
  - Check for dependences ⇒ “do we have ready instructions?”
  - Select ready instructions and map them to multiple function units
- **Instruction window**
  - When we look for parallel instructions, we want to consider many instructions (in “instruction window”) for the best result
  - Branches hinder forming a large, accurate window
The Opteron X4 Microarchitecture

Instruction Fetch
Decode
Dispatch
Execute
Mem
Write Back

The ARM Cortex-A8 architecture

FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The Address Generation Unit (AGU) uses a Branch Target Buffer (BTB), Global History Buffer (GHB), and a Return Stack (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.