Static, multiple-issue (superscalar) 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>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 superscalar

Superscalar execution (case of two pipelines)

**Static scheduling:**
- The compiler forms "super-instructions", each being two MIPS instructions (that do not have data dependence) one to execute on the ALU/branch pipe and the other on the load/store pipe.
- In each cycle a super-instruction is fetched and its two instructions are pushed through the two pipelines.
- To pack a super instructions, the compiler should
  - use no-ops in super-instructions if cannot find suitable MIPS instructions.
  - make sure that there is no data hazards (by inserting no-ops) – note that more no-ops will be inserted if hardware does not support forwarding.

**Dynamic scheduling:**
- Up to two MIPS instructions are fetched and buffered every cycle
- Up to two MIPS instructions are decoded and buffered every cycle
- Up to two MIPS instructions are moved to the execution pipelines every cycle depending on the data hazards that are dynamically detected.
- Note that more data hazards will be dynamically detected if hardware does not support forwarding
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
Loop: lw $t0, 0($s1)       // $t0 = array element
     addi $s1, $s1, -4   // decrement pointer
     add $t0, $t0, $s2   // add scalar in $s2
     sw $t0, 4($s1)      // store result
     bne $s1, $zero, Loop // branch if $s1 != 0
```

An equivalent code that separates dependent instructions ("lw" and "add"):

```
Loop: lw $t0, 0($s1)       // $t0 = array element
     addi $s1, $s1, -4   // decrement pointer
     add $t0, $t0, $s2   // add scalar in $s2
     sw $t0, 4($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></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)
     addi $s1, $s1, -4
     add $t0, $t0, $s2
     sw $t0, 4($s1)
     bne $s1, $zero, Loop
```

```
Loop: lw $t1, -4($s1)
     addi $s1, $s1, -8
     add $t0, $t0, $s2
     add $t1, $t1, $s2
     sw $t0, 8($s1)
     sw $t1, 4($s1)
     bne $s1, $zero, Loop
```

- 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:  lw $t0, 0($s1)  \\
 lw $t1, -4($s1)  \\
 addi $s1, $s1, -8  \\
 add $t0, $t0, $s2  \\
 add $t1, $t1, $s2  \\
 sw $t0, 8($s1)  \\
 sw $t1, 4($s1)  \\
bne $s1, $zero, Loop

<table>
<thead>
<tr>
<th>ALU or bne instructions</th>
<th>lw/sw instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>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

<table>
<thead>
<tr>
<th>ALU or bne instructions</th>
<th>lw/sw instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
<tr>
<td></td>
<td>sw $t2, 8($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 schedule two instructions in the same cycle if the second depends on the first
    \[
    \begin{align*}
    \text{add} & \quad t0, s0, s1 \\
    \text{l w} & \quad s2, 0(t0)
    \end{align*}
    \]
  – Load-use hazard
    \[
    \begin{align*}
    \text{l w} & \quad s2, 0(t0) \\
    \text{add} & \quad t0, s2, s1
    \end{align*}
    \]
  • Should be separated by at least one cycle

• Control Hazard
  – Penalty for a mis-predicted branch is proportional to issue width.
  – Example: if branch is resolved in EX stage, then 4 bubbles have to be introduced if the branch is mis-predicted (assuming that the instruction following the branch is never scheduled in the same cycle as the branch)

The Opteron X4 Microarchitecture

Instruction Fetch

Decode

Register read and Dispatch

Execute

Mem

Write Back
Dynamic Scheduling

- Static scheduling
  - Schedule instructions at compile time to get the best execution time

- Dynamic scheduling
  - Dependences must be honored while instructions are dispatched
    (read after write, write after read and write after write).
  - Instructions may execute out of order to minimize execution time (as long as the scheduler is sure that dependences are not violated)
  - For the best result, control dependences must be tackled

Components of dynamic scheduling
- Check for dependences \( \Rightarrow \) “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 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.
The Intel Core i7 architecture

Figure 4.77: The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

What is instruction level parallelism (ILP)?

- Execute independent instructions in parallel
  - Provide more hardware function units (e.g., adders, cache ports)
  - Detect instructions that can be executed in parallel (in hardware or software)
  - Schedule instructions to multiple function units (in hardware or software)
- Goal is to improve instruction throughput
- Pipelining (a single pipeline) is a form of ILP which ideally gives CPI = 1
- With multiple pipelines, we can achieve CPI < 1 (IPC > 1)
- ILP is different from thread-level parallelism and task-level parallelism (discussed in section 6).