Handling exceptions (Section 4.9)

- Exceptions (traps or interrupts) are an important part of a processor
- Modern OS and debuggers depend on the hardware support for exceptions
- Interrupt mechanism allows efficient use of CPU cycles by allowing the CPU to do useful work while waiting for an event.

- Precise exception
  - Execute all instructions before the faulting (trapped/interrupted) instruction
  - Do not execute the faulting instruction and the following ones
  - Resume from the faulting instruction after the exception is taken care of

- How does MIPS handle exceptions?
  - Record the PC of the next instruction in a special place (the \textit{EPC} register)
  - Record the cause of the exception (the \textit{Cause} register)
  - Squash the faulting instruction and the instructions following it
  - Jump to a pre-determined handling routine (PC = 8000 00180)
  - When done, the handling routine will execute a "return from interrupt" to resume from the instruction following the faulting instruction (EPC \rightarrow PC), or the faulting instruction (EPC-4 \rightarrow PC) – depending on implementation.

The Handler

- Typically, the exception handler will determine the action(s) to be taken if consequences are not fatal
  - take corrective action
  - use EPC to return to program

- Otherwise
  - Report error using EPC, cause, …
  - Terminate program

- Handler jumps to an address which is determined by the cause of the exception
  - Addresses are stored in a vector which is indexed by the cause
  - Example:

<table>
<thead>
<tr>
<th>Undefined opcode:</th>
<th>C000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overflow:</td>
<td>C000 0020</td>
</tr>
<tr>
<td>…</td>
<td>C000 0040</td>
</tr>
</tbody>
</table>

For external interrupts: the handler address is stored in a vector which is indexed by the interrupt type. The PC+4 is saved in EPC and the handler’s address replaces the current PC.
Exceptions in a Pipeline

- Can be seen as another form of control hazard

- Consider overflow on an `add` in EX stage
  
  `add $3, $2, $1`
  
  - Prevent $3 from being clobbered
  - Complete the instructions that entered the pipeline before the "add"
  - Flush the "add" and subsequent instructions
  - Set the `Cause` register
  - Save PC+4 in `EPC` register
  - Transfer control to handler

- To flush instructions, use the same techniques as for flushing the instructions following a miss-predicted branch
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>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

---

86

87
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

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

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>Loop: addi $s1, $s1, -4</td>
<td>lw $t0, 0($s1)</td>
</tr>
<tr>
<td>add $t0, $t0, $s2</td>
<td>sw $t0, 4($s1)</td>
</tr>
<tr>
<td>addi $s1, $s1, -4</td>
<td></td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>cycle 1</td>
</tr>
<tr>
<td></td>
<td>cycle 2</td>
</tr>
<tr>
<td></td>
<td>cycle 3</td>
</tr>
<tr>
<td></td>
<td>cycle 4</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 $t2, 4($s1)
     bne $s1, $zero, Loop

unrolling

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

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 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>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 0($s1)</td>
<td>lw $t1, -4($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1, -8</td>
<td>sw $t0, 8($s1)</td>
<td>2</td>
</tr>
<tr>
<td>add $t0, $t0, $s2</td>
<td>sw $t1, 4($s1)</td>
<td>3</td>
</tr>
<tr>
<td>add $t1, $t1, $s2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

- Note that we ignored control hazards

Unrolling 4 times

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

- 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.
  - Should be careful when scheduling a branch with a lw/sw

- Should be very careful if instruction will be executed out of order.

---

Dynamic Scheduling

- **Static scheduling**
  - Schedule instructions at compiler 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
Dynamic multiple-issue processors

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.

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).