When do MIPS exceptions occur?

- **IF**
  - page fault on instruction fetch
  - misaligned memory access
  - memory protection violation
- **ID**
  - undefined or illegal opcode
- **EX**
  - arithmetic exception
- **MEM**
  - page fault on data fetch/store
  - misaligned memory access
  - memory protection violation
- **WB**: None!

Examples of exception handling

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- Handle the MEM fault first, then restart

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

How is this done?

- Answer: Don’t handle exceptions until the WB stage
  - each instruction has an associated status vector that keeps track of faults
  - any bit set in the status vector turns off register writes and memory writes
  - in WB stage, the status vector is checked and any fault is handled
  - So, since instructions reach WB in proper order, faults for earlier instructions are handled before faults for later instructions
    » Unfortunately, will need to violate this later (for instructions that don’t reach WB in proper order)

Commitment

- When an instruction is guaranteed to complete, it is committed
- Life is easier if no instruction changes the permanent machine state before it is committed
- In MIPS, commitment occurs at the end of the MEM stage - that’s why register update occurs in the stage after that
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started

Complications with long instructions

- So far, all MIPS instructions take 5 cycles
- But haven’t talked yet about the floating point instructions
- Take it on faith that floating point instructions are inherently slower than integer arithmetic instructions
  » doubters may consult Appendix H in H&P online
How slow is slow?

- Some typical times:
  - **latency** is the number of cycles between an instruction that produces a result and one that uses it.
  - **initiation interval** is the number of cycles between two instructions of the same kind (for example, two ADD.Fs).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Initiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU uses</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Load/store</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ADD.F, SUB.F</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DIV.F</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

Examples

- If have a string of instructions:
  - ADD
  - SUB
  - AND
  - OR
  - SLLI

  - then there are no delays in the pipeline, because
  - **initiation=1** means can start one of these instructions every cycle.
  - **latency=0** means that results from one instruction will be available when the next instruction needs them.

Examples (cont.)

- Suppose have a string of instructions
  - ADD.F
  - SUB.F

  - Then **initiation=1** means that can start SUB.F one cycle behind ADD.F.
  - But **latency=3** means that this will work right **only if** SUB.F doesn’t need ADD.F’s results.
  - If it does need the results, then need **two instructions** in between ADD.F and SUB.F to prevent bubbles in the pipeline.

Examples (cont.) - Fig. A.32

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Italics** shows where data is needed
**blue** where a result is available.

Functional units - Fig. A.31

Hazards caused by long instructions

- The floating point adder and multiplier are pipelined, but the divider is not - that is why the initiation interval for divide is 25.
  - A program will run very slowly if it does too many of these!
- It will also run slowly if the results of the divide are needed too soon.
FP stalls from RAW hazards – Fig. A.33

<table>
<thead>
<tr>
<th>Inst.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F10, F2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL.D F2, F1, F6</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td></td>
</tr>
<tr>
<td>ADD.D F2, F1, F6</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D F2, F1, F6</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Long instructions (cont.)

- It is possible that two instructions enter the WB stage at the same time

<table>
<thead>
<tr>
<th>Inst.</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D F6, M7</td>
<td>IF</td>
<td>ID</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>stall</td>
<td>stall</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>S.D</td>
<td>stall</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>MEM</td>
</tr>
</tbody>
</table>

- A structural hazard

Long instructions (cont.)

- Instructions can finish in the wrong order
- This can cause WAW hazards
- This violation of WB ordering defeats the previous strategy for precise exception handling
  - problem is out-of-order completion

<table>
<thead>
<tr>
<th>Inst.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.D F10, F6, F14</td>
<td>IF</td>
<td>ID</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD.D F10, F12, F14</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F10, F12</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Possible fixes (cont.)

- Let the exception handler finish the instructions in the pipeline and then restart the pipe at the next instruction
- Have the floating point units diagnose exceptions in their first or second stages, so can handle them by methods that work well for handling integer exceptions
How to detect hazards in ID

- Early detection would prevent trouble
- Check for structural hazards:
  - will the divide unit clear in time?
  - will WB be possible when we need it?
- Check for RAW data hazards:
  - will all source registers be available when needed?
- Check for WAW data hazards:
  - is the destination register for any ADD.D, multiply or divide instruction the same register as the destination for this instruction?
- If anything dangerous could happen, delay the execute cycle so no conflict occurs

MIPS R4000 pipeline stages

- IF – 1st half instruction fetch
  - PC selection and start instruction cache access
- IS – 2nd half instruction fetch
  - complete instruction cache access
- RF – instruction decode, register fetch, hazard checking, instruction cache hit detection
- EX – execution
  - includes effective address computation, ALU operation, branch target computation and condition evaluation

MIPS R4000 pipeline (cont.)

- DF – 1st half data fetch
  - 1st half of data cache access
- DS – 2nd half data fetch
  - complete data cache access
- TC – tag check
  - determine whether data cache access hit
- WB – write back for loads and ALU operations

MIPS R4000 pipeline (cont.)

A 2 cycle load delay

A 3 cycle branch delay – 1 delay slot + 2 cycle stall for taken branch (untaken just delay slot)
Forwarding

- Deeper pipeline increases number of levels of forwarding for ALU operations
  - 4 possible sources for an ALU bypass
    - EX/DF
    - DF/DS
    - DS/TC
    - TC/WB

Floating point pipeline

- 3 functional units
  - divider, multiplier, adder
- Double precision FP ops take
  - from 2 (negate) up to 112 cycles (square root)
- Effectively 8 stages, combined in different orders for various FP operations
  - one copy of each stage, and some instructions use a stage zero or more times, and in different orders
- Overall, rather complicated ...
  - see H&P for more details

R4000 pipeline performance

- 4 major causes of pipeline stalls
  - load stalls – from using load result 1 or 2 cycles after load
  - branch stalls – 2 cycles on every taken branch, or empty branch delay slot
  - FP result stalls – RAW hazards for an FP operand
  - FP structural stalls – from conflicts for functional units in FP pipeline

SPEC92 benchmarks

Assuming a perfect cache – 5 integer and five FP programs

![SPEC92 benchmarks graph]

Dynamically scheduled pipelines

- We’ll cover this, and the scoreboard technique, in a bit
  - need some general background first

Pitfalls

- Unexpected hazards do occur ...
  - for example, when a branch is taken before a previous instruction finishes
- Extensive pipelining can slow a machine down, or lead to worse cost-performance
  - more complex hardware can cause a longer clock cycle, killing the benefits of more pipelining
Pitfalls (cont.)

• A poor compiler can make a good machine look bad
  – compiler writers need to understand the architecture in order to
    » optimize efficiently and
    » avoid hazards
  – better to eliminate useless instructions, than make them run faster