Precise Interrupts

[H&P §A.4] Precise interrupts are required in order for a program to have \textit{sequential semantics}—the same behavior as a program whose instructions are executed in order.

In order to achieve sequential semantics in a pipelined processor, we—

- Complete the instructions before the offending one.
- Quash (the effects of) instructions following it.
- Save the PC of the faulting instruction.
- Force a trap instruction into the pipeline on the next IF.

\textbf{What makes precise interrupts hard?}

Delayed branches cause a problem. What is it?

After the exception has been handled, a special instruction (\textbf{RFE}) is used to reload the PC(s) and restart the instruction stream.

Several problems can arise in doing this.

- The faulting instruction may write its result before the exception can be handled.
- Then the hardware must retrieve the source operands, even if one of them has been overwritten by the instruction.
- Or, the source operand(s) may have been overwritten by other instructions, especially in the case of an FP operation.

Several processors, including the MIPS R8000, have a \textit{precise} and an \textit{imprecise} floating-point interrupt mode, with the precise mode being much slower.

Interrupts can occur in almost any pipeline stage.
• IF – memory problems
• ID – illegal
• EX –
• MEM – memory problems (see IF above)
• WB – none

Which interrupt should be handled first?

*Example: A data-page fault.*

<table>
<thead>
<tr>
<th>Clock #</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>Instr. $i$</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>Instr. $i+1$</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>Instr. $i+2$</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>Instr. $i+3$</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>Instr. $i+4$</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>Instr. $i+5$</td>
<td>trap</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>
</tr>
<tr>
<td>Instr. $i+6$</td>
<td>trap handler</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>
</tr>
</tbody>
</table>

Events:

• Preceding instruction already complete
• Quash succeeding instructions—prevent them from modifying state.
• Inject a “trap” instruction, which jumps to trap handler. Trap handler saves precise state

*Example: Out-of-order interrupts*

Which page fault should we take?

<table>
<thead>
<tr>
<th>Clock #</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>Instr. $i$</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>Instr. $i+1$</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>

*Answer:* Even though instruction $i+1$ faults first, we must service the $i$ fault first.
Solution: The hardware posts all interrupts caused by a given instruction in a status vector associated with the instruction.

This vector must be carried along as the instruction makes its way through the pipeline.

The vector is checked upon entering WB. If there are any instructions, ___________ is handled.

This solves the problem for our simple pipeline, since instructions finish in order.

OOO execution and interrupts

The preceding example assumed an in-order pipeline. This makes precise interrupts easy

Consider OOO execution

<table>
<thead>
<tr>
<th>Clock #</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>MUL.D F0,F2,F4</td>
<td>IF</td>
<td>ID</td>
<td>IS</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>ADD.D F6,F8,F8</td>
<td>IF</td>
<td>ID</td>
<td>IS</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problem: ADD.D has already completed; state is imprecise!

Solutions:

- Imprecise interrupts—ignore the problem.
  This makes it impossible to guarantee that we handle page faults correctly. How does the following example show that?

<table>
<thead>
<tr>
<th>Clock #</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>Instr. i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>page fault</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr. i+1</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>Instr. i+2</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>
IEEE Standard strongly suggests precise interrupts

• In-order completion: Stall pipeline when necessary

• Software cleanup—Save pipeline-state information for trap handlers.

• Hardware cleanup—Restore consistent state

• Re-order instructions—Complete them out of order, retire them in order.

Complications from instruction sets

If an instruction modifies state early, then causes an interrupt, state change must be undone.

Example: Autoincrement, autodecrement mode

First operand of VAX instruction uses autodecrement addressing mode, which writes a register. Then a page fault occurs when the second operand is accessed.

Since instruction execution cannot be completed, we must restore the register written by autodecrement to its original value.

We can’t avoid the problem by waiting till the instruction is done to update the value in the autodecremented register. Why?

Example: Long-running instructions

It is not enough to be able to restore state; we must make progress from interrupt to interrupt.

Case 1: MVC on IBM 360 copies 256 bytes

Most 360 models had no virtual memory, so interrupts are not allowed to stop MVC.
Case 2: Example: MVC on IBM 370 copies 256 bytes.

Has virtual memory, so first we access all pages involved; after that, no interrupts allowed.

Case 3: MVCL on IBM 370 copies up to $2^{24}$ bytes.

- Has VM; two addresses and length are in registers.
- Registers saved and restored on interrupts (making progress).

Example: Condition codes

Many processors set the condition code implicitly as part of all, or almost all instructions. What is one advantage of this?

But the processor must then determine when the condition code has been set for the last time before the branch. What’s the problem here?

Reorder buffer (ROB)

[H&P §3.7] A reorder buffer (ROB) is a FIFO with head and tail pointers.

On instruction dispatch, we—

- Reserve ROB entry at tail and advance tail pointer.
- Record ROB entry (ROB tag) with instruction.

On completion of an instruction, we—

- Write value into the ROB entry specified by the instruction’s tag.
- Do not write result into the register file.
- If instruction caused an exception, set the exception bit in the ROB for offending instruction.
Institute a new phase: Instruction retire (a.k.a. commit, graduate).

- Check instruction at the head of the ROB
- If completed, check exception bit.
- If no exception, commit state (write value into register file) and advance head pointer.
- If exception, quash subsequent instructions in ROB (back up the tail ptr to the head ptr).

Our pipeline now looks like this:

```
IF  ID  IS  EX  WB  RE
  in order  out of order  in order
```

The reorder buffer operates like this:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Dest</th>
<th>Result</th>
<th>Exception</th>
<th>Completed</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F6</td>
<td>no</td>
<td></td>
<td>✓</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>F2</td>
<td>no</td>
<td></td>
<td>✓</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>F0</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>F8</td>
<td>no</td>
<td></td>
<td>✓</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>F10</td>
<td></td>
<td></td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>F6</td>
<td>no</td>
<td></td>
<td>✓</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Shown: The two loads (A and B) have completed and retired.

The MUL.D has not completed.
The ADD.D and SUB.D have completed out-of-order, but the ROB prevents them from retiring (updating state).

Thus, if the MUL.D posts an exception, the register-file state is precise.

**Precise Tomasulo Pipeline with ROB**

*Update state out of order: imprecise!*

*Retire: Update state in order: precise!*

A ROB requires a bypass.

**Scenario:**

- Producer instruction completed – value in ROB.
- Producer instruction not retired – value *not* in register file.
- Then, dependent instruction is dispatched into window.

**Explain why a bypass is needed.**