Exceptions - Basics

Exception = unprogrammed control transfer

- system takes action to handle the exception
  - must record the address of the offending instruction
  - record any other information necessary to return afterwards
- returns control to user
- must save & restore user state
Two Types of Exceptions

♦ Interrupts
  • caused by external events:
    » Network, Keyboard, Disk I/O, Timer
  • asynchronous to program execution
    » Most interrupts can be disabled for brief periods of time
    » may be handled between instructions
    » simply suspend and resume user program

♦ Traps
  • caused by internal events
    » exceptional conditions (overflow)
    » errors (parity)
    » page faults (non-resident page)
  • synchronous to program execution
  • condition must be remedied by the handler
  • instruction may be retried and program continued or program may be aborted

Exceptions - Examples

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synchronous vs. asynchronous</th>
<th>User request vs. counted</th>
<th>User maskable vs. nonmaskable</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO device request</td>
<td>Asynchronous</td>
<td>Counted</td>
<td>Nonmaskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Hardware fault</td>
<td>Synchronous</td>
<td>Counted</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Floating point arithmetic overflow or underflow</td>
<td>Synchronous</td>
<td>Counted</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>Synchronous</td>
<td>Counted</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Missaligned memory access</td>
<td>Synchronous</td>
<td>Counted</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Memory protection violations</td>
<td>Synchronous</td>
<td>Counted</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Using undefined instructions</td>
<td>Synchronous</td>
<td>Counted</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Asynchronous</td>
<td>Counted</td>
<td>Nonmaskable</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>
Exceptions in MIPS pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>Possible exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation; memory error</td>
</tr>
</tbody>
</table>

- How do we stop the pipeline? How do we restart it?
- Do we interrupt immediately or wait?
- 5 instructions, executing in 5 different pipeline stages!
  - Who caused the interrupt?

Multiple exceptions

Time (clock cycles)

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5
Load Ifetch Reg Reg Reg Reg Reg
Add Ifetch Reg Reg Reg Reg

Data page fault
Arithmetic exception

Time (clock cycles)

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5
Load Ifetch Reg Reg Reg Reg Reg
Add Ifetch Reg Reg Reg Reg

Data page fault
Instruction page fault
Precise Interrupts/Exceptions

- Exceptions should be **Precise** or clean, i.e., the outcome should be exactly the same as in a non-pipelined machine
- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions **completed**
  - Offending instruction and all following instructions act as if they have not even started
  - Same code will work on different processor implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Relationship between precise interrupts and speculation

- Speculation: guess and check
- Important for branch prediction:
  - Need to “take our best shot” at predicting branch direction
- If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly:
  - This is exactly the same as precise exceptions!
- Technique for both precise interrupts/exceptions and speculation: **in-order completion or commit**
HW support for precise interrupts

- Need HW buffer for results of uncommitted instructions: reorder buffer
  - An instruction **commits** when it completes its execution and all its predecessors have already committed
  - Once instruction commits, result is put into register
  - Therefore, easy to undo speculated instructions on mispredicted branches or exceptions
  - Supplies operands between execution complete & commit

---

Reorder Buffer

Result Shift Register

<table>
<thead>
<tr>
<th>Stage</th>
<th>Functional unit source</th>
<th>Valid Instruction</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Integer ADD</td>
<td>1 5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Flt. Pt. ADD</td>
<td>1 4</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry #</th>
<th>Dest. Reg.</th>
<th>Result</th>
<th>Exceptions</th>
<th>Valid Rslt</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>10</td>
<td></td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>9</td>
<td></td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reorder (circular) Buffer

Future (speculative) File

Source Data to functional units

Control

Copy upon exceptions

Architectural File

REORDER BUFFER

Common Result Bus

Future File

Result Shift Register

Entry # Dest. Reg. Result Exceptions Valid Rslt PC
3 0 6
4 10 0
5 9 0
6 0

ECE568/Koren Part.8 11
Copyright 2011 Koren UMass
4 Steps of Speculative Tomasulo Algorithm

1. **Issue** — get instruction from FP Op Queue
   
   If reservation station, reorder buffer slot, and result shift register slot free, issue instr & send operands & reorder buffer no. for destination. (this stage sometimes called “dispatch”)

2. **Execution** — operate on operands (EX)
   
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; this takes care of RAW. (sometimes called “issue”)

3. **Write result** — finish execution (WB)
   
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available

4. **Commit** — update register with result from reorder buffer
   
   When instr. at head of reorder buffer & result valid (present), update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes entries in reorder buffer. (sometimes called “graduation”)

---

**Tomasulo With Reorder buffer -1**

---

**FP Op Queue**

**Reorder Buffer**

**RAT**

**Registers**

**FP adders**

**FP multipliers**

---

**Done?**

ROB7

ROB6

ROB5

ROB4

ROB3

ROB2

ROB1

---

**To Memory**

from Memory

---

**FP adders**

**FP multipliers**

---

**Destination**

**From Memory**

**To Memory**

---

**Destination**

**From Memory**

---

**Destination**

**From Memory**

---

**Destination**

**From Memory**
**Code Example**

1. LD \( F_0, 10(R2) \)
2. ADDD \( F_{10}, F_4, F_0 \)
3. DIVD \( F_2, F_{10}, F_6 \)
4. BNE \( F_2, <...> \)
5. LD \( F_4, 0(R3) \)
6. ADDD \( F_0, F_4, F_6 \)
7. ADDD \( F_0, F_4, F_6 \)

Finishes 2\(^{nd}\) (9 cycles)

Finishes 1\(^{st}\) (3 cycles)
Tomasulo With Reorder buffer - 2

FP Op Queue

Reorder Buffer

RAT

Registers

Dest

2 ADDD R(F4), ROB1

FP adders

Reservation Stations

FP multipliers

Dest Value Instruction Done?

ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1

2 ADDD R(F4), ROB1

To Memory

F0 ADDD F10, F4, F0 N

F0 LD F0, 10(R2) N

F0 F2 F4 F10

FP adders

1 10+R2

Oldest

Newest

Dest

from Memory

to Memory

FP adders

FP adders

FP multipliers

FP multipliers

Tomasulo With Reorder buffer - 3

FP Op Queue

Reorder Buffer

RAT

Registers

Dest

2 ADDD R(F4), ROB1

FP adders

Reservation Stations

FP multipliers

Dest Value Instruction Done?

ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1

2 ADDD R(F4), ROB1

To Memory

F2 DIVD F2, F10, F6 N

F10 ADDD F10, F4, F0 N

F0 LD F0, 10(R2) N

F0 F2 F4 F10

FP adders

1 10+R2

Oldest

Newest

Dest

from Memory

to Memory

FP adders

FP adders

FP multipliers

FP multipliers

ECE568: Koren Part 8.17

Adapted from UCB and other sources

Copyright UCB & Morgan Kaufmann
Tomasulo With Reorder buffer - 4

FP Op Queue

Reorder Buffer

FP adders

FP multipliers

RAT

Registers

To Memory

From Memory

Dest Value Instruction Done?

F0
F4
---
F2
F10
F0

ADDD F0,F4,F6
LD F4,0(R3)
BNE F2,<…>
DIVD F2,F10,F6
ADDD F10,F4,F0
LD F0,10(R2)

N
N
N
N
N

ROB7
ROB6
ROB5
ROB3
ROB2
ROB1

FP adders

Reservation Stations

FP multipliers

Tomasulo With Reorder buffer - 5

FP Op Queue

Reorder Buffer

FP adders

FP multipliers

RAT

Registers

To Memory

From Memory

Dest Value Instruction Done?

F0
F4
---
F2
F10
F0

ADDD F0,F4,F6
LD F4,0(R3)
BNE F2,<…>
DIVD F2,F10,F6
ADDD F10,F4,F0
LD F0,10(R2)

N
N
N
N
N

ROB7
ROB6
ROB5
ROB3
ROB2
ROB1
### Tomasulo With Reorder buffer - 13

<table>
<thead>
<tr>
<th>FP Op Queue</th>
<th>Dest</th>
<th>Value</th>
<th>Instruction</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F0</td>
<td>&lt;val3&gt;</td>
<td>ADDD F0,F4,F6</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>F0</td>
<td>&lt;val2&gt;</td>
<td>ADDD F0,F4,F6</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td>M[10]</td>
<td>LD F4,0(R3)</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>&lt;val5&gt;</td>
<td>DIVD F2,F10,F6</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>F10</td>
<td>&lt;val4&gt;</td>
<td>ADDD F10,F4,F0</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>F0</td>
<td>M[20]</td>
<td>LD F0,10(R2)</td>
<td>E</td>
</tr>
</tbody>
</table>

### Tomasulo With Reorder buffer - 14

<table>
<thead>
<tr>
<th>FP Op Queue</th>
<th>Dest</th>
<th>Value</th>
<th>Instruction</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F0</td>
<td>&lt;val3&gt;</td>
<td>ADDD F0,F4,F6</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>F0</td>
<td>&lt;val2&gt;</td>
<td>ADDD F0,F4,F6</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td>M[10]</td>
<td>LD F4,0(R3)</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>&lt;val5&gt;</td>
<td>DIVD F2,F10,F6</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>F10</td>
<td>&lt;val4&gt;</td>
<td>ADDD F10,F4,F0</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>F0</td>
<td>M[20]</td>
<td>LD F0,10(R2)</td>
<td>E</td>
</tr>
</tbody>
</table>
Tomasulo With Reorder buffer - 15

- **FP adders**
  - Dest: F0, F4, F6
  - Instruction: ADDD, LD
  - Done? Y, C, E

- **FP multipliers**
  - Dest: F0, F10
  - Instruction: DIVD, ADDD
  - Done? E

- **RAT**
  - F0 = <val3>
  - F2 = <val15>
  - F4 = M[10]
  - F10 = <val4>

- **Registers**
  - Dest: F0, F2, F4, F10
  - Value: M[10], M[20]
  - Instruction: ADDD, LD, DIVD

- **Reorder Buffer**
  - ROB: ROB7, ROB6, ROB5, ROB4, ROB3, ROB2, ROB1

- **To Memory**
  - Dest: ROB7, ROB6

- **From Memory**
  - Dest: ROB1, ROB2

Tomasulo With Reorder buffer - 16

- **FP adders**
  - Dest: F0, F4, F6
  - Instruction: ADDD, LD
  - Done? C, E

- **FP multipliers**
  - Dest: F0, F10
  - Instruction: DIVD, ADDD
  - Done? E

- **RAT**
  - F0 = <val3>
  - F2 = <val15>
  - F4 = M[10]
  - F10 = <val4>

- **Registers**
  - Dest: F0, F2, F4, F10
  - Value: M[10], M[20]
  - Instruction: ADDD, LD, DIVD

- **Reorder Buffer**
  - ROB: ROB7, ROB6, ROB5, ROB4, ROB3, ROB2, ROB1

- **To Memory**
  - Dest: ROB7, ROB6

- **From Memory**
  - Dest: ROB1, ROB2