MIPS Exceptions

- Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - How to restart the pipeline?
  - Who caused the interrupt?

<table>
<thead>
<tr>
<th>Stage</th>
<th>Problem interrupts occurring</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 interrupt</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>
Pipelining Complications

- Simultaneous exceptions in more than one pipeline stage, e.g.,
  - Load with data page fault in MEM stage
  - Add with instruction page fault in IF stage
  - Add fault will happen BEFORE load fault

- Solution 1
  - Interrupt ASAP
  - Restart everything that is incomplete

- Solution 2
  - Precise interrupts - exceptions handled in order they would occur on unpipelined machine
    - At first interrupt, turn off write control (MEM stage interrupt prevents store)
    - Post all exceptions to interrupt status vector (one per instr in pipeline)
    - At WB stage, service any interrupts in order
    - Guarantees precise interrupts for simple integer pipeline
ISA Complications

• Interrupt Complexity in MIPS
  - Instruction is “committed” only at end of MEM stage
  - Committed instructions change the “state” of the program
  - Precise exceptions easier to implement

• Interrupt Complexity in IA-32
  - State changes occur before last stage (autoincrement addr mode)
  - Autoincrement causes register change in middle of instruction execution
  - Aborted instruction leaves “state” change intact
  - Adds WAR and WAW hazards since writes no longer last stage
  - Options:
    • Restart instr stream at point half way through instruction (difficult)
    • Disallow state change prior to commit (costly & difficult)
    • “Back-out” of instruction to previous state for changes prior to commit
The influence of complexity...

• Multicycle operations complicate Pipelines
• Pipelines provide superior throughput
• Some instructions may take 100s of cycles to complete
• Fixed number of clock cycles per instruction inefficient
• Pipelining such instructions too complex
• Solution: pipeline “micro instructions”
  – Micro instructions are simple building blocks of complex instruction set (CISC in IA-32 or VAX)
    • Processors of the 1980s
  – Or just start with a reduced instruction set (RISC)
    • Processors of the 1990s
• Of course multicycle operations like fp are unavoidable and must be handled as efficiently as possible...
Conceptual view of FP pipeline

- Suppose FP has same pipeline as INT
- Two changes
  - EX may be repeated
  - Multiple EX FP units
- Stalls
  - structural or data hazards
  - EX completion of multi-cycle op
- Four non-piped functional units
  - int load/store, ALU ops, branches
  - FP and int multiply
  - FP add, sub, conversion
  - FP int, divide

Can’t we pipeline the EX stage further?
The Execution Pipeline
(e.g. Pipelined Functional Units)

- Latency
  - number of intervening cycles between an instruction that produces a result and an instruction that uses the result
  - Ex: ALU reg1 op reg2 stored in EX (0); load reg1 loaded in MEM (1)

- Initiation interval
  - number cycles that must elapse between issuing two ops of a given type

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data memory (integer and FP loads)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply (also integer multiply)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP divide (also integer divide)</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

- Higher clock rate ➔ fewer logic levels per pipe stage
- Fewer logic levels per stage ➔ larger number of stages for complex instr
- So, faster clock rate ➔ longer latency for operations
Multiple Outstanding Operations

- Pipelined EX unit allows
  - 4 outstanding FP adds
  - 7 outstanding FP/int multiplies
  - 1 outstanding FP divide

Requires:
Additional temp regs
### Multi-cycle Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.D</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>L.D</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>S.D</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>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F4,0(R2)</td>
</tr>
<tr>
<td>MUL.D</td>
<td>F0,04,05</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F2,00,03</td>
</tr>
<tr>
<td>S.D</td>
<td>F2,0(R2)</td>
</tr>
</tbody>
</table>

RAW hazards
Hazard Detection and Forwarding

- Structural hazards
  - For divides
  - Multiple reg writes in single cycle can occur (single write port)

- Data hazards
  - Out of order (OOO) completion possible
    - WAW hazards since WB occur OOO (L.D one cycle earlier + dest F2)
    - Problems with exceptions
  - RAW hazards more frequent due to long latency ops

<table>
<thead>
<tr>
<th>Instruction</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 F0,F4,F6</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 F2,F4,F6</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>...</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 F2,0(R2)</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>
Implementing Hazard Detection and Forwarding

- Multiple reg writes in single cycle can occur (single write port)
  - Multiple write ports, not efficiently used
  - Track use of write port and stall before issue (ID stage)
- WAW hazards since WB occur OOO
  - Delay issue of load
  - Stamp out result of add.d
- Before instruction issue:
  - Check for structural hazards
  - Check for RAW hazards
  - Check for WAW hazards
- Conceptually the same to implement, yet more cumbersome
Precise Exceptions in Multicycle Ops

DIV F0, F2, F4
ADD F10, F10, F8
SUB F12, F12, F14

- Assume Add completes prior to divide and SUB causes exception
  - Result: imprecise exception

- Approaches:
  - Ignore the problem (use imprecise exceptions)
  - Buffer issued ops until earlier ops complete.
    - Expense with large diffs in instruction latency (complex)
  - Allow somewhat imprecise exceptions (utilize software)
  - Allow issue only when you know previous issues will complete before this instruction causes an exception
Can you hear me now?

- Finishing up appendix A, then on to Chapter 3
- Discussed basic control from other text
- Discussed basic pipeline control from our text
  - Hazard avoidance and detection (RAW, WAR, WAW)
    - Data forwarding in some cases, stalls in others
  - Branch
    - Stalls and branch prediction (more on this later)
- Discussed difficulties in pipelining
  - Exceptions
    - Precise (difficult to implement, slow, but required for virtual mem)
    - Handling earlier is better
  - Multicycle operations
    - Results in more complicated hazard detection
    - Precise exceptions difficult with OOO completion (int pipeline easier)
- Up Next:
  - An example: MIPS R4000
  - How to implement OOO execution? Scoreboarding