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

Examples

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

- Then, no delays in the pipeline, because
  - Instructions can be issued to EX stage every cycle
  - Results from one instruction will be available when the next instruction needs them without stall
Latency and Initiation Interval

- **Latency**
  - the (min.) number of cycles between an instruction that produces a result and the one that uses it

- **Initiation interval**
  - the (min.) 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>MUL.F</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>DIV.F</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

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 three instructions in between ADD.F and SUB.F to prevent bubbles in the pipeline

Examples (cont.) - Fig. C.36

<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>MUL.D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></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>
<td></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>
<td></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

Hazards caused by long instructions

- The floating point adder and multiplier are pipelined, but the divider is not (initiation interval: 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 (RAW hazards)
FP stalls from RAW hazards – Fig. C.37

<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 F4, 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>
</tr>
<tr>
<td>MUL.D F0, F4, 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>M6</td>
</tr>
<tr>
<td>ADD.D F2, F0, F8</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, 0(R2)</td>
<td>stall</td>
<td>IF</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Structural Hazard from Long instructions

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

<table>
<thead>
<tr>
<th>ADD.D</th>
<th>IF</th>
<th>ID</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADD</td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DADD</td>
<td>IF</td>
<td>ID</td>
<td>ALU</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Long instructions - WAW structural Hazard

- Instructions can finish in the wrong order
- can cause WAW hazards

<table>
<thead>
<tr>
<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>
</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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td>L.D F4, 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>
</tr>
</tbody>
</table>

Long instructions (cont.)

- Instructions can finish in the wrong order
- can cause WAW hazards
- Solutions?
  1) stop fetching
  2) turn off writes
  3) let pipeline drain
  4) handle
- Violation of WB ordering defeats the previous strategy for precise exception handling
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

Maintaining Precise Exceptions

- Out-of-order completion is the problem!!!
  - What happens if sub faults?
    - DIV.D F0, F2, F4
    - ADD.D F10, F10, F8
    - SUB.D F10, F12, F14
    - What about F10?

Approaches?
1. Give up and just do imprecise exception handling
   - tempting, but annoying to users
2. Buffer results until all previous instructions complete
   - since so many instructions can be active, this is expensive - requires a lot of supporting hardware
3. Write, to memory, a history file of register and memory changes so can undo instructions if necessary
   - or keep a future file of computed results that are waiting for MEM or WB

A case study: MIPS R4000

- MIPS64 architecture, with deeper 8 stage pipeline
  - to get higher clock rates
  - extra stages come from memory accesses
  - techniques called superpipelining

IF IS RF EX DF DS TC WB

© 2003 Elsevier Science (USA). All rights reserved.
MIPS R4000 pipeline stages

- **IF** – 1st half instruction fetch
  - PC selection
  - Initiation of instruction cache access
- **IS** – 2nd half instruction fetch
  - complete instruction cache access
- **RF**
  - instruction decode
  - register fetch
  - hazard checking
  - instruction cache hit detection (tag check)

MIPS R4000 pipeline stages (cont.)

- **EX** – execution
  - includes effective address computation, ALU operation, branch target computation and condition evaluation
- **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 – Handling Load Delay

- **A 2-cycle load delay**

Handling Load Delay (cont.)

- 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

MIPS R4000 pipeline – Handling Load Delay (cont.)

- **LW R1 IF IS RF EX DF DS TC WB**
- **ADD R2, R1 IF IS RF Stall Stall EX DF DS TC WB**
- **SUB R3, R1 IF IS Stall Stall RF EX DF DS TC**
- **OR R4, R1 IF Stall Stall IS RF EX DF DS**

Might need to restart ADD’s ALU
MIPS R4000 pipeline – Handling Branch Delay

- A 3-cycle branch delay – 1 delay slot + 2 cycle stalls for taken branch (if untaken, just delay slot)
  - For 2 stalls, uses “predicted-not-taken” strategy

Floating point pipeline

- 3 floating point 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

MIPS R4000 FP Unit

- FP adder
- Mantissa ADD stage
- FP divider
- Divide pipeline stage
- FP multiplier
- Exception test stage
- FP multiplier
- First stage of multiplier
- FP multiplier
- Second stage of multiplier
- FP adder
- Rounding stage
- FP adder
- Operand shift stage
- Unpack FP numbers

MIPS 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

<table>
<thead>
<tr>
<th>FP 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>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Subtract</td>
<td>S+A</td>
<td>A+R</td>
<td>R-S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>U</td>
<td>E=M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>A</td>
<td>R</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td>D^2</td>
<td>D</td>
<td>A</td>
<td>D-A</td>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>Square root</td>
<td>U</td>
<td>E</td>
<td>(A+R)^n</td>
<td>...</td>
<td>A</td>
<td>R</td>
<td>112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td>U</td>
<td>S</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute value</td>
<td>U</td>
<td>S</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP compare</td>
<td>U</td>
<td>A</td>
<td>R</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SPEC92 benchmarks

- Assuming a perfect cache –
  - 5 integer and five FP programs

For integer

What we already know about pipelining

- Avoid structural hazards, data hazards and control hazards in order to get better pipeline performance
- Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

Accomplish this by techniques such as
- instruction reordering
- hardware modifications to detect branches earlier
- compiler approaches to reduce branch delays

Each technique reduces one or more of the stall components of the Pipeline CPI