Revisiting Branch Hazard Solutions

- Stall
- Predict Not Taken
- Predict Taken
- Branch Delay Slot

Predict Not Taken

<table>
<thead>
<tr>
<th>Branch</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Filling the delay slot (e.g., in the compiler)

Can be done when?

Improves performance when?

```
lw R1, 10000(R7)
add R5, R6, R1
beqz R5, label:
sub R8, R1, R3
add R4, R8, R9
and R2, R4, R8
```

label: add R2, R5, R8
Problems filling delay slot

1. need to predict direction of branch to be most effective
2. limited by correctness restriction
   - correctness restriction can be removed by a canceling branch
     branch likely or branch not likely
e.g.,
   - beqz likely
delay slot instruction fall-through instruction squashed/nullified/canceled if branch not taken

Branch Likely

Branch Performance

CPI = BCPI + pipeline stalls from branches per instruction
   = 1.0 + branch frequency * branch penalty
assume 20% branches, 67% taken:

<table>
<thead>
<tr>
<th>branch</th>
<th>taken</th>
<th>not taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>penalty</td>
<td>penalty</td>
</tr>
<tr>
<td>stall</td>
<td>predict taken</td>
<td>predict not taken</td>
</tr>
<tr>
<td>predict</td>
<td>delayed branch</td>
<td></td>
</tr>
</tbody>
</table>

Static Branch Prediction

- Static branch prediction takes place at compile time, dynamic branch prediction during program execution
- static bp done by software, dynamic bp done in hardware
- Static branch prediction enables
  - more effective code scheduling around hazards (how?)
  - more effective use of delay slots
MIPS Integer Pipeline Performance

- Only stalls for load hazards and branch hazards, both of which can be reduced (but not eliminated) by software

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Load stalls</th>
<th>Branch stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>5%</td>
<td>9%</td>
</tr>
<tr>
<td>eqntott</td>
<td>3%</td>
<td>6%</td>
</tr>
<tr>
<td>espre</td>
<td>4%</td>
<td>7%</td>
</tr>
<tr>
<td>gcc</td>
<td>4%</td>
<td>8%</td>
</tr>
<tr>
<td>li</td>
<td>9%</td>
<td>9%</td>
</tr>
</tbody>
</table>

But now, the real world interrupts...

- Pipelining is not as easy as we have made it seem so far...
  - interrupts and exceptions
  - long-latency instructions

Exceptions and Interrupts

- Transfer of control flow (to an exception handler) without an explicit branch or jump
- are often unpredictable
- examples
  - I/O device request
  - OS system call
  - arithmetic overflow/underflow
  - FP error
  - page fault
  - memory-protection violation
  - hardware error
  - undefined instruction

Classes of Exceptions

- synchronous vs. asynchronous
- user-initiated vs. coerced
- user maskable vs. nonmaskable
- within instruction vs. between instructions
- resume vs. terminate

when the pipeline can be stopped just before the faulting instruction, and can be restarted from there (if necessary), the pipeline supports *precise exceptions*
Basic Exception Methodology

- turn off writes for faulting instruction and following
- force a trap into the pipeline at the next IF
- save the PC of the faulting instruction (not quite enough for delayed branches)

Exceptions Can Occur In Several Places in the pipeline

- IF -- page fault on memory access, misaligned memory access, memory-protection violation
- ID -- illegal opcode
- EX -- arithmetic exception
- MEM -- page fault, misaligned access, memory-protection violation
- WB -- none
(and, of course, asynchronous can happen anytime)

Simplifying Exceptions in the ISA

- Each instruction changes machine state only once
  - autoincrement
  - string operations
  - condition codes
- Each instruction changes machine state at the end of the pipeline (when you know it will not cause an exception)

Handling Multicycle Operations

- Unrealistic to expect that all operations take the same amount of time to execute
- FP, some memory operations will take longer
- This violates some of the assumptions of our simple pipeline
Multiple Execution Pipelines

<table>
<thead>
<tr>
<th>FU</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP divide</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

New problems

• structural hazards
  – divide unit
  – WB stage
• WAW hazards are possible
• out-of-order completion
• WAR hazards still not possible

Structural hazards and WAW hazards

• structural hazards
  – divide unit
  – WB stage

<table>
<thead>
<tr>
<th>Instruction</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>ADDD</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></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>
</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>
</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>
</tr>
<tr>
<td>LD</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>LD</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td>MEM</td>
<td></td>
<td>WB</td>
</tr>
</tbody>
</table>

WAW hazards

<table>
<thead>
<tr>
<th>Instruction</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>ADDD F8, ...</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>
</tr>
<tr>
<td>LD F8, ...</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td>MEM</td>
<td></td>
<td></td>
<td>WB</td>
</tr>
</tbody>
</table>
**MIPS R4000 Pipeline**

- scalar, superpipelined
  - IF—first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS—second half of access to instruction cache.
  - RF—instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX—execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF—data fetch, first half of access to data cache.
  - DS—second half of access to data cache.
  - TC—tag check, determine whether the data cache access hit.
  - WB—write back for loads and register-register operations.

**R4000 Data Load Hazard**

- Is there an integer arithmetic data hazard?

**R4000 Branch Hazard**

- predict not taken, branch delay slot
- not taken -> no penalty (unless branch likely or no delay slot instruction)
- taken -> 2 stall cycles if delay slot instruction used
Key Points

- Data Hazards can be significantly reduced by forwarding
- Branch hazards can be reduced by early computation of condition and target, branch delay slots, branch prediction
- Data hazard and branch hazard reduction require complex compiler support
- Exceptions are hard, precise exceptions are really hard
- Variable-length instructions introduce structural hazards, WAW hazards, more RAW hazards