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

Branch

IF (bubble) (bubble) (bubble) (bubble)

Branch Target

IF | ID | EX | MEM | WB

Delayed Branch

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

Branch

IF | ID | EX | MEM | WB |

I+1 (delay slot)

IF | ID | EX | MEM | WB |

Branch Target

IF | ID | EX | MEM | WB |

T+1

IF | ID | EX | MEM | WB |

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 __________ of branch to be most effective
2. limited by ______________ restriction

• __________ restriction can be removed by a canceling branch
  branch likely or branch not likely
e.g.,
  * beqz likely
  * delay slot instruction
  * fall-through instruction

Branch Likely

Delay Slot Utilization

• 18% of delay slots left empty
• 11% of delay slots (1) use canceling branches and (2) end up getting canceled

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>
<th>penalty</th>
<th>penalty</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>stall</td>
<td>predict taken</td>
<td>predict not taken</td>
<td>delayed branch</td>
<td></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
- How to make static branch predictions?
- Static branch prediction enables
  - more effective code scheduling around hazards (how?)
  - more effective use of delay slots

Misprediction rate

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>0%</th>
<th>5%</th>
<th>10%</th>
<th>15%</th>
<th>20%</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>2%</td>
<td>6%</td>
<td>9%</td>
<td>11%</td>
<td>12%</td>
</tr>
<tr>
<td>eqn</td>
<td>4%</td>
<td>5%</td>
<td>6%</td>
<td>7%</td>
<td>9%</td>
</tr>
<tr>
<td>tott</td>
<td>6%</td>
<td>5%</td>
<td>6%</td>
<td>7%</td>
<td>8%</td>
</tr>
<tr>
<td>espresso</td>
<td>9%</td>
<td>8%</td>
<td>7%</td>
<td>6%</td>
<td>5%</td>
</tr>
<tr>
<td>gcc</td>
<td>4%</td>
<td>5%</td>
<td>6%</td>
<td>7%</td>
<td>8%</td>
</tr>
<tr>
<td>li</td>
<td>4%</td>
<td>5%</td>
<td>6%</td>
<td>7%</td>
<td>8%</td>
</tr>
<tr>
<td>doduc</td>
<td>10%</td>
<td>15%</td>
<td>20%</td>
<td>25%</td>
<td>22%</td>
</tr>
<tr>
<td>ear</td>
<td>14%</td>
<td>15%</td>
<td>16%</td>
<td>17%</td>
<td>18%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>11%</td>
<td>12%</td>
<td>13%</td>
<td>14%</td>
<td>15%</td>
</tr>
<tr>
<td>mdlj</td>
<td>7%</td>
<td>5%</td>
<td>6%</td>
<td>7%</td>
<td>8%</td>
</tr>
<tr>
<td>dp</td>
<td>5%</td>
<td>6%</td>
<td>7%</td>
<td>8%</td>
<td>9%</td>
</tr>
<tr>
<td>su2cor</td>
<td>9%</td>
<td>10%</td>
<td>11%</td>
<td>12%</td>
<td>13%</td>
</tr>
</tbody>
</table>

MIPS Integer Pipeline Performance

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

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
- WB -- none

(and, of course, asynchronous can happen anytime)

Simplifying Exceptions in the ISA

1. Each instruction changes machine state only once
   1. autoincrement
   2. string operations
   3. condition codes

2. 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
- ____, some ________________ 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

- WAW hazards

ADDD IF ID A1 A2 A3 A4 MEM WB
... IF ID EX MEM WB
... IF ID EX MEM WB
LD IF ID EX MEM WB

ADDD F8, ... IF ID A1 A2 A3 A4 MEM WB
LD F8, ... IF ID EX MEM WB
Hazard Detection in the ID stage

• An instruction can only issue (proceed past the ID stage) when:
  – there are no structural hazards (divide unit is free, WB port will be free when needed)
  – no RAW data hazards (that forwarding can’t handle)
  – no WAW hazards with instructions in long pipes

Another pipeline – MIPS R4000

• Still in-order
• Still scalar
• Deeper (more pipeline stages). Why?

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 Data Load Hazard

lw R1,... IF IS RF EX DF DS TC WB
add R3, R1, R2 IF IS RF (stall) (stall) EX DF

• 2-cycle load delay

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

R4000 Performance

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