### 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?
Implements performance when?

- \( \text{lw} \ R1, 10000(\text{R7}) \)
- \( \text{add} \ R5, R6, R1 \)
- \( \text{beqz} \ R5, \text{label:} \)
- \( \text{sub} \ R8, R1, R3 \)
- \( \text{add} \ R4, R8, R9 \)
- \( \text{and} \ R2, R4, R8 \)

Label: \( \text{add} \ R2, R5, R8 \)

### 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></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?
Implements performance when?

- \( \text{lw} \ R1, 10000(\text{R7}) \)
- \( \text{add} \ R5, R6, R1 \)
- \( \text{beqz} \ R5, \text{label:} \)
- \( \text{sub} \ R8, R1, R3 \)
- \( \text{add} \ R4, R8, R9 \)
- \( \text{and} \ R2, R4, R8 \)

Label: \( \text{add} \ R2, R5, R8 \)

### 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., \( \text{beqz} \) likely
- delay slot instruction
- fall-through instruction

Squashed/nullified/canceled if branch not taken
**Branch Likely**

<table>
<thead>
<tr>
<th>Branch Likely</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I+1 (delay slot)</td>
<td>(bubble)</td>
<td>(bubble)</td>
<td>(bubble)</td>
<td>(bubble)</td>
<td>(bubble)</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>

<table>
<thead>
<tr>
<th>Branch Likely</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>I+1 (delay slot)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch Target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>T+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**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**

\[
\text{CPI} = \text{BCPI} + \text{pipeline stalls from branches per instruction} = 1.0 + \text{branch frequency} \times \text{branch penalty}
\]

Assume 20% branches, 67% taken:

<table>
<thead>
<tr>
<th>branch</th>
<th>taken</th>
<th>not taken</th>
<th>scheme</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>
<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 chart](chart)
MIPS Integer Pipeline Performance

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

[Bar chart showing percentage of instructions that stall]

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

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: FU Latency and Initiation Interval

<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>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</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>LD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></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>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
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**

- 2-cycle load delay
- 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