Control hazards

- Question: When do we find out that the PC needs to be modified?
  - Answer: In pipeline stage ID of a branch instruction
  - So, if a branch is not-taken (i.e., if the PC is not modified), need a one-cycle delay

- Question: When is a taken branch’s address known?
  - ALU used to compute, so EX stage
  - Need two (or three) cycle delay

Example

- If branch in 30% of instructions, then instead of executing 1 instruction per cycle,
  - have 70% of instructions executing in 1 cycle and 30% of instructions executing in 2 cycles
- An average of \(0.7 + 0.6 = 1.3\) cycles per instruction
  - Worse by 30%

Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles \(\Rightarrow\) new CPI = 1.9!

  - Two part solution:
    - Determine branch taken or not taken sooner, AND
    - Compute taken branch address earlier
  - MIPS branch tests if register = 0 or \(\neq 0\)
  - MIPS Solution:
    - Move Zero test to ID/RF stage
    - Adder to calculate new PC in ID/RF stage
    - 1 clock cycle penalty for branch versus 3
Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
- 53% MIPS branches taken on average
- But haven’t calculated branch target address in MIPS
- MIPS still incurs 1 cycle branch penalty
- Other machines: branch target known before outcome

#4: Delayed Branch
- Define branch to take place AFTER a following instruction

- branch instruction
  sequential successor, sequential successor, ...
  sequential successor, branch target if taken

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Scheduling Branch Delay Slots (Fig A.14)

A. From before branch
```
add R1,R2,R3  
if R2=0 then
  delay slot
```
becomes
```
if R2=0 then
  add R1,R2,R3  
```

B. From branch target
```
sub R4,R5,R6  
if R1=0 then
  delay slot
```
becomes
```
if R1=0 then
  sub R4,R5,R6  
```

C. From fall through
```
add R1,R2,R3  
if R1=0 then
  delay slot
```
becomes
```
sub R4,R5,R6  
if R1=0 then
  add R1,R2,R3  
```

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

- Delayed Branch downside:
  - As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot

- Result:
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

Assume:
- 4% unconditional branch
- 6% conditional branch-untaken
- 10% conditional branch-taken

<table>
<thead>
<tr>
<th>Scheduling Scheme</th>
<th>Branch Penalty</th>
<th>CPI</th>
<th>Speedup vs. Unpipelined</th>
<th>Speedup vs. Stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall Pipeline</td>
<td>3</td>
<td>1.60</td>
<td>3.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Predict Taken</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
<td>1.33</td>
</tr>
<tr>
<td>Predict Not Taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed Branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
<td>1.45</td>
</tr>
</tbody>
</table>
Pipelining Summary

- Pipelining can speed instruction execution (throughput)
- But need to deal with structural hazards, data hazards, and control hazards
- Next
  - How to handle exceptions?
  - How to handle long instructions, such as floating point arithmetic?

The problem

- Question: What makes pipelining hard to implement?
- Answer: Surprises
- Technical names for surprises:
  - exceptions
  - faults
  - interrupts

Some examples of exceptions

- Request for I/O
- Arithmetic troubles: overflow or underflow
- Cache miss: data not in (on-chip) cache memory
- Page fault: data not in (physical) memory
- Illegal address, giving a memory protection violation
- Hardware failure

Classifying exceptions

- Synchronous: repeatable every time
  - Example: DIV R2, R2, R0
- Asynchronous: caused by external events like hardware failure and devices external to processor and memory
- User requested: user task asks for it (example: breakpoint)
- Coerced: cannot be predicted by user
- User maskable: can be disabled by user task
  - Example: arithmetic exception
- Nonmaskable: cannot be turned off
  - Example: hardware failure

Classifying exceptions (cont.)

- Within instruction: prevents instruction from completing
- Between instructions: no instruction prevented
- Terminating: stops the task
- Resuming: task can continue

- Machines that handle exceptions, save the state, and then restart correctly are said to be restartable

Categorizing exceptions – Fig. A. 27

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synch. vs. async.</th>
<th>User request vs. coerced</th>
<th>User maskable vs. not</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Invoke OS</td>
<td>Synch</td>
<td>User req.</td>
<td>Not</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Tracing instructions</td>
<td>Synch</td>
<td>User req.</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Synch</td>
<td>User req.</td>
<td>Maskable</td>
<td>Between</td>
<td>Resume</td>
</tr>
<tr>
<td>Integer overflow</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Floating pt. overflow</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
</tbody>
</table>
### Categorizing exceptions (cont.)

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Synch. vs. async.</th>
<th>User request vs. coerce</th>
<th>User maskable vs. not</th>
<th>Within vs. between instructions</th>
<th>Resume vs. terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page fault</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Misaligned memory access</td>
<td>Synch</td>
<td>Coerced</td>
<td>Maskable</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Mem. prot. violation</td>
<td>Synch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Resume</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Hardware malfunction</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
<tr>
<td>Power failure</td>
<td>Asynch</td>
<td>Coerced</td>
<td>Not</td>
<td>Within</td>
<td>Terminate</td>
</tr>
</tbody>
</table>

### The most difficult exceptions...

- ... are those that occur within EX or MEM stages and need to be handled in a restartable way
- Why difficult? Handling one includes:
  - the next IF gets a “trap instruction”
  - until the trap is taken, turn off all “writes” for the faulting instruction and those that follow it
  - what does the trap do?
    - The trap transfers control to the exception handling routine in the operating system, which saves the PC of the faulting instruction and handles the fault
  - the task is then resumed, using the saved PC and the MIPS instruction RFE or something like it
- Note: May need to save several PCs if delayed branches are involved

### Exceptions (cont.)

- Ideally, pipeline can be interrupted so that instructions before the fault complete. Then want to restart execution just after the faulting instruction - precise exception handling
- This is the right way to do it, but sometimes architects/manufacturers take shortcuts

### When do MIPS exceptions occur?

- **IF**
  - page fault on instruction fetch
  - misaligned memory access
  - memory protection violation
- **ID**
  - undefined or illegal opcode
- **EX**
  - arithmetic exception
- **MEM**
  - page fault on data fetch/store
  - misaligned memory access
  - memory protection violation
- **WB**: None!

### Examples of exception handling

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

- Handle the MEM fault first, then restart

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

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

### How is this done?

- Answer: Don’t handle exceptions until the WB stage
  - each instruction has an associated status vector that keeps track of faults
  - any bit set in the status vector turns off register writes and memory writes
  - in WB stage, the status vector is checked and any fault is handled
  - So, since instructions reach WB in proper order, faults for earlier instructions are handled before faults for later instructions
  - Unfortunately, will need to violate this later (for instructions that don’t reach WB in proper order)
Commitment

- When an instruction is guaranteed to complete, it is committed.
- Life is easier if no instruction changes the permanent machine state before it is committed.
- In MIPS, commitment occurs at the end of the MEM stage - that's why register update occurs in the stage after that.
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started.