CMSC 411
Computer Systems Architecture
Lecture 5
Basic Pipelining (cont.)

Administrivia
• Homework problems for Unit 1 due Thu

Forwarding to Avoid LW-SW Data Hazard
Figure A.8, Page A-29

Data Hazard Even with Forwarding
Figure A.9, Page A-20

Data Hazard Even with Forwarding
(Similar to Figure A.10, Page A-21)

Software Scheduling Instead

Try producing fast code for
\[ a = b + c; \]
\[ d = e - f; \]
assuming \( a, b, c, d, e, \) and \( f \) in memory.

Slow code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Slow code</th>
<th>Instruction</th>
<th>Slow code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Rd, b</td>
<td>LW</td>
<td>Rd, b</td>
</tr>
<tr>
<td>LW</td>
<td>Rd, c</td>
<td>LW</td>
<td>Rd, c</td>
</tr>
<tr>
<td>ADD</td>
<td>Rs, Rd, Rc</td>
<td>LW</td>
<td>Rs, Rd, Rc</td>
</tr>
<tr>
<td>SW</td>
<td>a, Rs</td>
<td>ADD</td>
<td>Rs, Rd, Rc</td>
</tr>
<tr>
<td>LW</td>
<td>Re, e</td>
<td>SW</td>
<td>a, Rs</td>
</tr>
<tr>
<td>LW</td>
<td>Rd, f</td>
<td>SUB</td>
<td>Rd, Rs, Rf</td>
</tr>
<tr>
<td>SW</td>
<td>d, Rd</td>
<td>SW</td>
<td>d, Rd</td>
</tr>
</tbody>
</table>

Fast code:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fast code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Rd, b</td>
</tr>
<tr>
<td>LW</td>
<td>Rd, c</td>
</tr>
<tr>
<td>ADD</td>
<td>Rs, Rd, Rc</td>
</tr>
<tr>
<td>SW</td>
<td>a, Rs</td>
</tr>
<tr>
<td>LW</td>
<td>Re, e</td>
</tr>
<tr>
<td>LW</td>
<td>Rd, f</td>
</tr>
<tr>
<td>SUB</td>
<td>Rd, Rs, Rf</td>
</tr>
<tr>
<td>SW</td>
<td>d, Rd</td>
</tr>
</tbody>
</table>

Compiler optimizes for performance. Hardware checks for safety.
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%

Control Hazard on Branches Three Stage Stall

```
10: beq r1,r3,34
14: and r2,r3,r5
18: or r6,r1,r7
22: add r8,r1,r9
34: xor r10,r1,r11
```

What do you do with the 3 instructions in between?
How do you do it?
Where is the “commit”?

Branch Stall Impact

• If CPI = 1, 30% branch,
  Stall 3 cycles => 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 ≠0 ≠0 ≠0 ≠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

Pipelined MIPS Datapath

• Interplay of instruction set design and cycle time.
### Four Branch Hazard Alternatives

#### #4: Delayed Branch
- Define branch to take place **AFTER** a following instruction
  - branch instruction
  - sequential successor₁
  - sequential successor₂
  - sequential successorₙ
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

### Scheduling branch delay slot

- **If taken from before branch**
  - branch must not depend on rescheduled instruction
  - always improves performance
- **If taken from branch target**
  - must be OK to execute rescheduled instructions if branch not taken, and may need to duplicate instructions
  - performance improved when branch taken
- **If taken from fall through**
  - must be OK to execute instructions if branch taken
  - improves performance when branch not taken

### 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

### Evaluating Branch Alternatives

- **Pipeline speedup** = \( \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}} \)

  **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. asynch.</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>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>Maskable</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/underflow</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. asynch.</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>Synch</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