CSE 141 – Computer Architecture
Spring 2005

Lectures 14
Pipeline Control Hazards

Pramod V. Argade
May 16, 2005

Announcements

- **Reading Assignment**
  Chapter 6: Enhancing Performance with Pipelining
  - 6.6, 6.8

- **Homework 6:**
  - 6.31, 6.32
  - Chapter 7 problems to be assigned next week

- **Quiz**
  **When:** Mon., May 23rd, First 10 minutes of the class
  **Topic:** Cache Chapter 7  **Need:** Paper, pen
## Course Schedule

<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Day</th>
<th>Lecture Topic</th>
<th>Quiz Topic</th>
<th>Homework Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3/28</td>
<td>Monday</td>
<td>Introduction, Ch. 1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>3/30</td>
<td>Wednesday</td>
<td>Performance, Ch. 4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>4/4</td>
<td>Monday</td>
<td>ISA, Ch. 2</td>
<td>Performance #1</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>4/6</td>
<td>Wednesday</td>
<td>Arithmetic, Ch. 3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>4/11</td>
<td>Monday</td>
<td>Arithmetic, Ch. 3</td>
<td>ISA #2</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>4/13</td>
<td>Wednesday</td>
<td>Single cycle CPU, Ch. 5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>4/18</td>
<td>Monday</td>
<td>Single cycle CPU, Ch. 5</td>
<td>Arithmetic #3</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>4/20</td>
<td>Wednesday</td>
<td>Multi-cycle CPU, Ch. 5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>4/25</td>
<td>Monday</td>
<td>Multi-cycle CPU, Ch. 5</td>
<td>Single Cycle CPU #4</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>4/27</td>
<td>Wednesday</td>
<td>Review for the Midterm</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>5/2</td>
<td>Monday</td>
<td>Mid-term Exam Center 101</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>5/4</td>
<td>Wednesday</td>
<td>Exceptions, Ch. 5 and Pipelining, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>5/9</td>
<td>Monday</td>
<td>Pipelining, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>5/16</td>
<td>Wednesday</td>
<td>Data and control hazards, Ch. 6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>5/18</td>
<td>Wednesday</td>
<td>Memory &amp; cache design, Ch. 7</td>
<td>Pipeline Hazards #5</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>5/23</td>
<td>Monday</td>
<td>Memory &amp; cache design, Ch. 7</td>
<td>Cache #6</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>5/25</td>
<td>Wednesday</td>
<td>Virtual Memory &amp; cache design, Ch. 7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>No Class</td>
<td>5/30</td>
<td>Monday</td>
<td>Memorial Day Holiday</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>6/1</td>
<td>Wednesday</td>
<td>Course Review</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>6/10</td>
<td>Friday</td>
<td>7 - 10 PM Final Exam</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

---

## Control Hazards
Conditional Branches in a Pipeline

- In a program flow, data computed by certain instructions is used to determine next instruction to execute
  - using conditional branches
- In a pipelined processor, conditional branches result in control hazards

**Control Flow Graph**

```
sub $2, $2, $5
and $6, $2, $4
beq $6, $8, L9

fall-through path
add $a, $b, $c
or $d, $e, $f
L9: and $x, $y, $z
sub $p, $q, $r
```

**Executed Path**

```
sub $2, $2, $5
and $6, $2, $4
beq $6, $8, L9

branch path
L9: and $x, $y, $z
sub $p, $q, $r
```

Pipelined Datapath and Control

- Decision about whether to branch doesn’t occur until the MEM pipeline stage
Impact of a Branch Instruction on the Pipeline

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 beq $1, $3, 7</td>
<td>CC 1</td>
</tr>
<tr>
<td>44 and $12, $2, $5</td>
<td>CC 2</td>
</tr>
<tr>
<td>48 or $13, $6, $2</td>
<td>CC 3</td>
</tr>
<tr>
<td>52 add $14, $2, $2</td>
<td>CC 4</td>
</tr>
<tr>
<td>72 lw $4, 50($7)</td>
<td>CC 5</td>
</tr>
</tbody>
</table>

Decision about whether to branch doesn’t occur until the MEM pipeline stage
Can you explain why the PC of the target instruction is 72?

Dealing With Branch Hazards

- **Software**
  - Insert nops,
  - Insert instructions that get executed either way (delayed branch).

- **Hardware**
  - Stall until you know which direction
    - 3 cycles wasted for every branch
  - Guess which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)
  - Ignore the branch for a cycle (branch delay slot)
Branch Hazards

```
beq $2, $1, here
add ...
sub ...
lw ...
here: and $12, $2, $5
```

When we decide to branch, other instructions are in the pipeline!

Stalling for Branch Hazards:
Assume the branch is taken

```
beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...
```

Wastes cycles if branch is not taken
Assume Branch Not Taken

Same performance as stalling when you’re wrong
This is preferred approach

Pipelined Datapath and Control

There is a 3 cycle penalty if a branch is taken
How could we reduce this penalty?
Resolving Branch in ID Stage, and Flushing if Branch is Taken

Note: Forwarding paths and muxes have to be added before registers are compared in ID stage
• Forwarding path from EX/MEM to IF/ID
• Forwarding path from MEM/WB to IF/ID

Problem: What if instruction immediately preceding branch write to the required register?
• The pipeline must be stalled for one clock cycle in ID stage
Reducing the delay of branches

- Resolve the branch in ID stage
  - Move register compare in ID stage
  - Provide data forwarding
    - Ensure that most recent register values are used in ID stage
    - Add necessary forwarding muxes and paths

- Implement faster logic to compare registers
  - Current ALU approach
    - Subtract the two registers and check whether the result is zero
    - Slow!
  - Faster approach
    - Exclusive OR the two registers. OR result bits to check whether the result is zero
    - Fast, since no carry propagation

Flush Instructions in the Pipe if a Branch is Taken

- Flushing an instruction means to prevent it from changing any permanent state (registers, memory, PC).
  - Similar to a pipeline bubble...
  - The difference is that we need to be able to insert those bubbles later in the pipeline

- Flushing an instruction on a taken branch
  - Must flush the instruction being fetched in IF stage using IF.Flush signal, which changes all instruction fields to zero
    - SLL $0, $0, 0 is equivalent to NOP
  - Let the instruction fields percolate through the pipeline
Eliminating the 1 Cycle Branch Stall

- There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
- The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!
- This works great for this implementation of the architecture, but becomes a permanent part of the ISA.
- What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle?
Branch Delay Slot

Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Scheduling Branch Delay Slot

The branch delay slot is only useful if you can find something to put there. If you can’t find anything, you must put a *nop* to insure correctness.

For b and c, $t4 must be an unused temporary register
Importance of efficient processing of branches

- Our implementation assumes “branch not taken”
  - Move branch resolution to ID stage
  - Flush instruction in IF stage if the branch is taken
- 15 to 20% of all instructions are branches
- MIPS
  - branch stall of 1 cycle, 1 instruction issued per cycle
  - delayed branch
- Recent processors
  - 3-4 cycle hazard, 1-2 instructions issued per cycle
  - cost of branch mis-prediction goes up
- Pentium Pro
  - 12+ cycle misprediction penalty, 3 instructions issued per cycle
  - HUGE penalty for mispredicting a branch
  - 36+ issue slots wasted

Predicting Branch Direction

- Easiest
  - always not taken, always taken
  - forward not taken, backward always taken
    ➢ Appropriate for loops
  - compiler predicted (branch likely, branch not likely)
- Save history of branch outcome
  - If the history is available, use it in the fetch stage
    ➢ Change PC accordingly
  - In the decode stage verify that the prediction was correct
    ➢ If not, set correct PC, flush pipeline, update history
- Next easiest
  - Record 1-bit history of whether the branch was taken or not
    ➢ 1-bit predictor
1-bit Pattern History Table (PHT)

- Uses low bits of branch address to choose an entry
- The entry has 1 branch prediction bit
- Size is small, e.g. 1 bits by N (e.g. 4K)

- Why not use all bits of branch address?
- What happens when the table is too small?

Prediction is incorrect twice for loops

Resolving Branch in ID Stage, and Flushing if Branch is Taken

Note: Branch prediction logic is implemented in the IF stage
- Check PC in PHT if the instruction is a branch (need to have a “quick test”)
- Modify next PC accordingly
- Validate branch prediction in ID stage
- If prediction was incorrect
- Invalidate instructions in the pipeline
- Start execution at correct PC
2-bit Branch Prediction Scheme

Branch prediction has to be incorrect twice before it is changed

Control Hazards -- Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.

- Control hazards are detected in hardware.

- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch prediction
  - branch delay slots
Exceptions

Exception Handling in the Pipeline

- Consider arithmetic overflow exception
  - add $1, $2, $1
- Extra hardware
  - Note: add is in EX stage
  - Flush instructions that follow add
    - In IF stage, assert IF.flush
    - In ID stage, use mux added for stall (OR ID.flush)
    - In EX stage, use EX.flush signal
  - **Prevent the ADD instruction from writing to $1**
    - Enables programmer to too see the value of $1 that caused exception
  - Transfer control to PC = 0x4000 0040
  - Save PC in EPC (need to subtract 4 from it before saving)
  - Save exception cause in Cause Register
Datapath and Control for Exceptions

Exception Handling in a Pipeline

0x40 sub $11, $2, $4
0x44 and $12, $2, $5
0x48 or $13, $2, $6
0x4c add $1, $2, $1
0x50 slt $15, $6, $7
0x54 lw $16, 50($7)

0x80000180 sw $25, 1000($0)
0x80000184 sw $26, 1004($0)

Note: ALU overflow signal is input the control unit
Issues in Handling an Exception

- Five instructions are active in the pipeline
- Multiple exceptions may occur
  - Earliest instruction that generated exception is interrupted
- Exceptions are detected in different stage of the pipeline
  - Undefined instruction is discovered in ID stage
  - Overflow is detected in EX stage
  - Kernel call (i.e. OS call) is detected in EX stage
- Precise exception
  - EPC saves PC of the instruction that caused exception
  - This is required for virtual memory
- Imprecise exception
  - EPC may not save PC of the instruction that caused exception
    - For ease of implementation

Summary: Hazards & Exceptions

- Data Hazard
  - Operand forwarding
  - Stall when forwarding cannot be done
- Control Hazards
  - Stall on branch
  - Predict branch
- Exceptions
  - May be detected in different stages of the pipeline
  - Must flush the offending instructions and all following it
  - Save context
  - Jump to exception handler