CSE 141 – Computer Architecture  
Fall 2003

Lectures 13  
Pipelining and Control Hazards

Pramod V. Argade

CSE141: Introduction to Computer Architecture

Web-page:  http://www-cse.ucsd.edu/classes/fa03/cse141

Reading Assignment: Sections 6.1 through 6.5

Homework:  6.4, 6.10, 6.11, 6.12, 6.13, 6.20, 6.23, 6.26, 6.28, 6.29, 6.30

Due Date:  Tuesday, November 25th

Next Quiz:  Thursday, November 20th

Final Exam:  Tuesday, December 9, 2003, 8:00 - 11:00am.

Room:  Price Center Theatre
### Schedule

<table>
<thead>
<tr>
<th>Lecture #</th>
<th>Date</th>
<th>Day</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sep. 25</td>
<td>Thursday</td>
<td>Introduction, Ch. 1</td>
</tr>
<tr>
<td>2</td>
<td>Sep. 30</td>
<td>Tuesday</td>
<td>Performance, Ch. 2</td>
</tr>
<tr>
<td>3</td>
<td>Oct. 2</td>
<td>Thursday</td>
<td>ISA, Ch. 3</td>
</tr>
<tr>
<td>4</td>
<td>Oct. 7</td>
<td>Tuesday</td>
<td>Arithmetic, Ch. 4, Continued</td>
</tr>
<tr>
<td>5</td>
<td>Oct. 9</td>
<td>Thursday</td>
<td>Arithmetic, Ch. 4</td>
</tr>
<tr>
<td>6</td>
<td>Oct. 14</td>
<td>Tuesday</td>
<td>Single cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>7</td>
<td>Oct. 16</td>
<td>Thursday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>8</td>
<td>Oct. 21</td>
<td>Tuesday</td>
<td>Multi-cycle CPU, Ch. 5</td>
</tr>
<tr>
<td>9</td>
<td>Oct. 23</td>
<td>Thursday</td>
<td>Classes cancelled due to wildfires</td>
</tr>
<tr>
<td>10</td>
<td>Oct. 28</td>
<td>Tuesday</td>
<td>Exceptions and Review for Midterm</td>
</tr>
<tr>
<td>11</td>
<td>Oct. 30</td>
<td>Thursday</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>12</td>
<td>Nov. 4</td>
<td>Tuesday</td>
<td>Mid-term Exam</td>
</tr>
<tr>
<td>13</td>
<td>Nov. 6</td>
<td>Thursday</td>
<td>Veteran's Day Holiday</td>
</tr>
<tr>
<td></td>
<td>Nov. 11</td>
<td>Tuesday</td>
<td>Data hazards, Ch. 6</td>
</tr>
<tr>
<td>14</td>
<td>Nov. 13</td>
<td>Thursday</td>
<td>Data hazards, Ch. 6</td>
</tr>
<tr>
<td>15</td>
<td>Nov. 18</td>
<td>Tuesday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>16</td>
<td>Nov. 20</td>
<td>Thursday</td>
<td>Data and control hazards, Ch. 6</td>
</tr>
<tr>
<td>17</td>
<td>Nov. 25</td>
<td>Tuesday</td>
<td>Advanced pipelining issues, Ch. 6</td>
</tr>
<tr>
<td></td>
<td>Nov. 27</td>
<td>Thursday</td>
<td>Thanksgiving Holiday</td>
</tr>
<tr>
<td>18</td>
<td>Dec. 2</td>
<td>Tuesday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td>19</td>
<td>Dec. 4</td>
<td>Thursday</td>
<td>Memory &amp; cache design, Ch. 7</td>
</tr>
<tr>
<td></td>
<td>Dec. 9</td>
<td>Tuesday</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

### Example

```
Example

ADD R1, R2, R3
SW R1, 1000(R2)
LW R7, 2000(R2)
ADD R5, R7, R1
LW R8, 2004(R2)
SW R7, 2008(R8)
ADD R8, R8, R2
LW R9, 1012(R8)
SW R9, 1016(R8)
```

Pramod Argade
UCSD CSE 141, Fall 2003
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

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

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

Control Flow Graph

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

Executed Path
Impact of a Branch Instruction on the Pipeline

Program execution order (in instructions)

Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9

40 beq $1, $3, 7
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)

Decision about whether to branch doesn’t occur until the MEM pipeline stage

Dealing With Branch Hazards

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

- Hardware
  - stall until you know which direction
    - 4 cycles wasted for every branch
  - reduce hazard through earlier computation of branch direction
  - 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**

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

**Stalling for Branch Hazards**

Wastes cycles if branch is not taken
Assume Branch *Not Taken*

- Works pretty well when you’re right

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

Assume Branch *Not Taken*

- Same performance as stalling when you’re wrong

```
beq $4, $0, there
and $12, $2, $5
or ...
add ...
there: and $12, $2, $5
```
So far Branches are resolved in the M Stage

There is a 3 cycle penalty if a branch is taken

Reducing the delay of branches

- Resolve the branch in ID stage
  - Move register compare in ID stage
  - Add necessary forwarding muxes and paths
- Implement faster logic to compare registers
  - Current ALU approach
    - Subtract the two register and check whether the result is zero
    - Slow!
  - Faster approach
    - Exclusive OR the registers and check whether the result is zero
- Provide data forwarding
  - Ensure that most recent register values are used in ID stage
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 instructions on a taken branch
  - Must flush three instructions in IF, ID and EX stages
  - To flush an instruction,
    - Change all the control fields to zero
    - Let them percolate through the pipeline

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

Note: Forwarding paths and muxes have to be added before registers are compared
Eliminating the 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 Branch Prediction

- **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 misprediction 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)

- **Next easiest**
  - Record 1-bit history of whether the branch was taken or not
    - 1-bit predictor
    - For a loop, the predictor is incorrect twice
  - 2-bit predictor
    - Branch prediction has to be incorrect twice before it is changed
Pattern History Table

- Uses low bits of branch address to choose an entry
- The entry has 1 or 2 branch prediction bits
- Size is small, e.g. 2 bits by N (e.g. 4K)

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

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