Control Hazards
Today

- Quiz 5
- Mini project #1 solution
- Mini project #2 assigned
- Stalling recap
- Branches!
Key Points: Control Hazards

• Control hazards occur when we don’t know which instruction to execute next
• Mostly caused by branches
• Strategies for dealing with them
  • Stall
  • Guess!
    • Leads to speculation
    • Flushing the pipeline
    • Strategies for making better guesses
• Understand the difference between stall and flush
Ideal operation

Cycles
Stalling for Load

To “stall” we insert a noop in place of the instruction and freeze the earlier stages of the pipeline.
To “stall” we insert a noop in place of the instruction and freeze the earlier stages of the pipeline.
Control Hazards

- Computing the new PC

```assembly
add $s1, $s3, $s2
sub $s6, $s5, $s2
beq $s6, $s7, somewhere
and $s2, $s3, $s1
```
Computing the PC

• Non-branch instruction
  • PC = PC + 4

• When is PC ready? No Hazard.
Computing the PC

- Branch instructions
  - `bne $s1, $s2, offset`
  - `if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}`
- When is the value ready?
Solution #1: Stall on branches

- Worked for loads and ALU ops.
- But wait!
- When do we know whether the instruction is a branch?

We would have to stall on every instruction
There is a constant control hazard

- We don’t even know what kind of instruction we have until decode.
- What do we do?
Smart ISA design

- Make it very easy to tell if the instruction is a branch -- maybe a single bit or just a couple.
- Decoding these bits is nearly trivial.
- In MIPS the branches and jumps are opcodes 0-7, so if the high order bits are zero, it's a control instruction.
Dealing with Branches: Option 1 -- stall

- What does this do to our CPI?
- Speedup?

\begin{verbatim}
sll $s4, $t6, $t5
bne $t2, $s0, somewhere
add $s0, $t0, $t1
and $s4, $t0, $t1
\end{verbatim}
Performance impact of

• $ET = I \times CPI \times CT$

• Branches about about 1 in 5 instructions

• What’s the CPI for branches?


• Amdah’ls law: $\text{Speedup} = \frac{1}{(\frac{.2}{\frac{1}{3}}) + (.8)} = 0.714$

• $ET = 1 \times (.2 \times 3 + .8 \times 1) \times 1 = 1.4$
Option 2: The compiler

- Use “branch delay” slots.
- The next $N$ instructions after a branch are always executed.
- Much like load-delay slots.
- Good
  - Simple hardware
- Bad
  - $N$ cannot change.
- MIPS has one branch delay slot
  - It’s a big pain!
Delay slots.

- **Taken**
  
  ```
  bne $t2, $s0, somewhere
  ```

- **Branch Delay**
  
  ```
  add $t2, $s4, $t1
  add $s0, $t0, $t1
  ...
  somewhere:
  sub $t2, $s0, $t3
  ```
Option 3: Simple Prediction

• Can a processor tell the future?
• For non-taken branches, the new PC is ready immediately.
• Let’s just assume the branch is not taken
• Also called “branch prediction” or “control speculation”
• What if we are wrong?
Predict Not-taken

- We start the ‘add’ and the ‘and’, and then, when we discover the branch outcome, we “squash” them.
- This means we turn it into a Noop.
Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros?
- Predict not-taken
  - Pros?

Loops are commons
Not all branches are for loops.

Backward Taken/Forward not taken
Best of both worlds.
Implementing Backward taken/forward not taken
Implementing Backward taken/forward not taken

sign bit

comparison result

Compute target

Insert bubble to flush
Implementing Backward taken/forward not taken

- Changes in control
- New inputs to the control unit
  - The sign of the offset
  - The result of the branch
- New outputs from control
  - The flush signal.
  - Inserts “noop” bits in datapath and control
Performance Impact

- \( ET = I \times CPI \times CT \)
- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the pipeline increases the cycle time by 10%
- What is the speedup \( Bt/Fnt \) compared to just stalling on every branch?
Performance Impact

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
- Btfnt
  - CPI = 0.2*0.2*(1 + 2) + (1−.2*.2)*1 = 1.08
  - CT = 1.1
  - ET = 1.188
- Stall
  - CPI = .2*3 + .8*1 = 1.4
  - CT = 1
  - ET = 1.4
- Speed up = 1.4/1.188 = 1.17
The Importance of Pipeline depth

- There are two important parameters of the pipeline that determine the impact of branches on performance
  - Branch decode time -- how many cycles does it take to identify a branch (in our case, this is less than 1)
  - Branch resolution time -- cycles until the real branch outcome is known (in our case, this is 2 cycles)
Pentium 4 pipeline (Willamette)

1. Branches take 19 cycles to resolve
2. Identifying a branch takes 4 cycles.
   1. The P4 fetches 3 instructions per cycle
3. Stalling is not an option.

- Pentium 4 pipelines peaked at 31 stage!!!
- Current cpus have about 12-14 stages.
Performance Impact

- ET = I * CPI * CT

- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup Bt/Fnt compared to just stalling on every branch?
- Btfnt
  - CPI = .2*.2*(1 + 2) + .9*1
  - CT = 1.1
  - ET = 1.118
- Stall
  - CPI = .2*4 + .8*1 = 1.6
  - CT = 1
  - ET = 1.4
- Speed up = 1.4/1.118 = 1.18

What if this were 20?
Performance Impact

- \( \text{ET} = \text{I} \times \text{CPI} \times \text{CT} \)
- Back taken, forward not taken is 80% accurate
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup \( B_{bt}/F_{fnt} \) compared to just stalling on every branch?
- \( B_{bt}/F_{fnt} \)
  - \( \text{CPI} = 0.2 \times 0.2 \times (1 + 20) + 0.8 \times 1 = 1.64 \)
  - \( \text{CT} = 1.1 \)
  - \( \text{ET} = 1.804 \)
- Stall
  - \( \text{CPI} = 0.2 \times (1 + 20) + 0.8 \times 1 = 5 \)
  - \( \text{CT} = 1 \)
  - \( \text{ET} = 5 \)
- \( \text{Speed up} = \frac{5}{1.804} = 2.77 \)
Dynamic Branch Prediction

- Long pipes demand higher accuracy than static schemes can deliver.
- Instead of making the guess once, make it every time we see the branch.
- Predict future behavior based on past behavior.
Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?
Predictable control

- Use previous branch behavior to predict future branch behavior.
- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {} 9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}
    - The branch is always taken or always not taken.
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it's always the same.
    - BaseClass * t; // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function
Dynamic Predictor 1: The Simplest Thing

• Predict that this branch will go the same way as the previous one.

• Pros?
  Dead simple. Keep a bit in the fetch stage. Works ok for simple loops. The compiler might be able to arrange things to make it work better.

• Cons?
  An unpredictable branch in a loop will mess everything up. It can’t tell the difference between branches.
Dynamic Prediction 2: A table of bits

- Give each branch its own bit in a table
  - How big does the table need to be?
    - Infinite! Bigger is better, but don’t mess with the cycle time. Index into it using the low order bits of the PC
- Look up the prediction bit for the branch

- Pros: It can differentiate between branches. Bad behavior by one won’t mess up others.... mostly (why not always?)
- Cons: Accuracy is still not great.
Dynamic Prediction 2: A table of bits

```
for(i = 0; i < 10; i++) {
    for(j = 0; j < 4; j++) {
    }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>prediction</th>
<th>new prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>5</td>
<td>taken</td>
<td>not taken</td>
<td>take</td>
</tr>
<tr>
<td>6</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>7</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

• What’s the accuracy for the inner loop’s branch?

50% or 2 per loop
Dynamic prediction 3: A table of counters

• Instead of a single bit, keep two. This gives four possible states

• Taken branches move the state to the right. Not-taken branches move it to the left.

<table>
<thead>
<tr>
<th>State</th>
<th>00 -- strongly not taken</th>
<th>01 -- weakly not taken</th>
<th>10 -- weakly taken</th>
<th>11 -- strongly taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td>not taken</td>
<td>not taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

• The net effect is that we wait a bit to change our mind
Dynamic Prediction 3: A table of counters

```
for(i = 0; i < 10; i++) {
  for(j = 0; j < 4; j++) {
  }
}
```

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>state</th>
<th>prediction</th>
<th>new state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>weakly taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
<td>strongly taken</td>
</tr>
</tbody>
</table>

- What’s the accuracy for the inner loop’s branch? (start in weakly taken)
Two-bit Prediction

- The two bit prediction scheme is used very widely and in many ways.
  - Make a table of 2-bit predictors
  - Devise a way to associate a 2-bit predictor with each dynamic branch
  - Use the 2-bit predictor for each branch to make the prediction.
- In the previous example we associated the predictors with branches using the PC.
  - We’ll call this “per-PC” prediction.
Associating Predictors with Branches: Using the low-order PC bits

- When is branch behavior predictable?
  - Loops -- for(i = 0; i < 10; i++) {}  9 taken branches, 1 not-taken branch. All 10 are pretty predictable.
  - Run-time constants
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}.
    - The branch is always taken or not taken.
  - Corollated control
    - a = 10; b = <something usually larger than a >
    - if (a > 10) {}
    - if (b > 10) {}
  - Function calls
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it’s always the same.
    - BaseClass * t;  // t is usually a of sub class, SubClass
    - t->SomeVirtualFunction() // will usually call the same function
Predicting Loop Branches Revisited

for(i = 0; i < 10; i++) {
    for(j = 0; j < 4; j++) {
    }
}

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
</tr>
</tbody>
</table>

• What’s the pattern we need to identify?
Dynamic prediction 4: Global branch history

- Instead of using the PC to choose the predictor, use a bit vector made up of the previous branch outcomes.

<table>
<thead>
<tr>
<th>iteration</th>
<th>Actual</th>
<th>Branch history</th>
<th>Steady state prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
<tr>
<td>outer loop branch</td>
<td>taken</td>
<td>11110</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>11101</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
</tbody>
</table>

Nearly perfect
Dynamic prediction 4: Global branch history

• How long should the history be?

  Infinite is a bad choice. We would learn nothing.

• Imagine N bits of history and a loop that executes K iterations
• If K <= N, history will do well.
• If K > N, history will do poorly, since the history register will always be all 1’s for the last K-N iterations. We will mis-predict the last branch.
Associating Predictors with Branches: Global history

- When is branch behavior predictable?  
  - Loops -- for(i = 0; i < 10; i++) {}  9 taken branches, 1 not-taken branch. All 10 are pretty predictable.  
  - Run-time constants  
    - Foo(int v,) { for (i = 0; i < 1000; i++) {if (v) {...}}}  
    - The branch is always taken or not taken.  
  - Corollated control  
    - a = 10; b = <something usually larger than a >  
    - if (a > 10) {}  
    - if (b > 10) {}  
  - Function calls  
    - LibraryFunction() -- Converts to a jr (jump register) instruction, but it’s always the same.  
    - BaseClass * t; // t is usually a of sub class, SubClass  
    - t->SomeVirtualFunction() // will usually call the same function

Good
Not so great
Pretty good, as long as the history is not too long
Not applicable