What about branches?

- Branch outcomes are not known until EXE
- What are our options?
Control Hazards
Today

• Quiz
• Control Hazards
• Midterm review
• Return your papers
Key Points: Control Hazards

- Control occur when we don’t know what the next instruction is
- 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
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?
Computing the PC

- Non-branch instruction
  - PC = PC + 4

- When is PC ready?

Diagram:
- Fetch
- Decode
- EX
- Mem
- Write back
Computing the PC

• Branch instructions
  • `bne $s1, $s2, offset`
  • `if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}`
• When is the value ready?
Computing the PC

- Branch instructions
  - `bne $s1, $s2, offset`
  - `if ($s1 != $s2) { PC = PC + offset} else {PC = PC + 4;}`
- When is the value ready?
Computing the PC

- Wait, when we do know?

```java
if (Instruction is branch) {
    if ($s1 != $s2) {
        PC = PC + offset;
    } else {
        PC = PC + 4;
    }
} else {
    PC = PC + 4;
}
```
Computing the PC

• Wait, when we do know?

```java
if (Instruction is branch) {
    if ($s1 != $s2) {
        PC = PC + offset;
    } else {
        PC = PC + 4;
    }
} else {
    PC = PC + 4;
}
```
There is a constant control hazard

• We don’t even know what kind of instruction we have until decode.
• Let’s consider the non-branch case first.
• What do we do?
Option 1: Smart ISA design

- Make it very easy to tell if the instruction is a branch -- maybe a single bit or just a couple.
- Decode is trivial
- Pre-decode --
  - Do part of decode when the instruction comes on chip.
  - more on this later
Option 2: The compiler

- Use “branch delay” slots.
- The next $N$ instructions after a branch are always executed.
- Good
  - Simple hardware
- Bad
  - $N$ cannot change.
Delay slots.

**Branch Delay**

- Taken
  - `bne $t2, $s0, somewhere`
- `add $t2, $s4, $t1`
- `add $s0, $t0, $t1`
- `...`
- `somewhere: sub $t2, $s0, $t3`
Option 4: Stall

- What does this do to our CPI?
- Speedup?
Performance impact of stalling

• ET = I * CPI * CT

• Branches about about 1 in 5 instructions
• What’s the CPI for branches?

• Speedup =
• ET =
Performance impact of stalling

- $ET = I \times CPI \times CT$

- Branches about about 1 in 5 instructions

- What’s the CPI for branches? $1 + 2 = 3$
  
  This is really the CPI for the instruction that follows the branch.

- Speedup =

- $ET =$
Performance impact of stalling

- ET = I \times \text{CPI} \times CT

- Branches about about 1 in 5 instructions
- What’s the CPI for branches? \quad 1 + 2 = 3
  
  This is really the CPI for the instruction that follows the branch.

- Speedup = \frac{1}{0.2/(1/3) + 0.8} = 0.714
- ET =
Performance impact of stalling

- \( ET = I \times CPI \times CT \)

- Branches about about 1 in 5 instructions
- What’s the CPI for branches? \( 1 + 2 = 3 \)
  This is really the CPI for the instruction that follows the branch.

- Speedup = \( \frac{1}{0.2/(1/3) + 0.8} = 0.714 \)
- \( ET = 1 \times (0.2 \times 3 + 0.8 \times 1) \times 1 = 1.4 \)
Option 2: 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?
We start the add, and then, when we discover the branch outcome, we squash it.

- We “flush” the pipeline.
We start the add, and then, when we discover the branch outcome, we squash it.

We “flush” the pipeline.
Predict Not-taken

- We start the add, and then, when we discover the branch outcome, we squash it.
- We “flush” the pipeline.

```
Not-taken
bne $t2, $s0, somewhere

Taken
bne $t2, $s4, else

add $s0, $t0, $t1

... else:
sub $t2, $s0, $t3
```
We start the add, and then, when we discover the branch outcome, we squash it.

- We “flush” the pipeline.
Simple “static” Prediction

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

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

Loops are commons
Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
  - Pros? Loops are commons
- Predict not-taken
  - Pros? Not all branches are for loops.
Simple “static” Prediction

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

Compute target

Insert bubble
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 front end increases the cycle time by 10%
- What is the speedup \( Bt/Fnt \) compared to just stalling on every branch?
Performance Impact

- $ET = I \times CPI \times CT$
- 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?
  - $Btfnt$
    - $CPI = 0.2 \times 0.2 \times (1 + 2) + (1 - 0.2 \times 2) \times 1 = 2.8$  
    - $CT = 1.1$
    - $ET = 1.188$
  - Stall
    - $CPI = 0.2 \times 3 + 0.8 \times 1 = 1.4$
    - $CT = 1$
    - $ET = 1.4$
  - Speed up = $1.4 / 1.188 = 1.18$
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

1. Branches take 19 cycles to resolve.
2. Identifying a branch takes 4 cycles.
3. Stalling is not an option.
Performance Impact

- \( ET = I \times CPI \times CT \)
- 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**
  - \( CPI = 0.2 \times 2 = 0.4 \)
  - \( CT = 1.1 \)
  - \( ET = 1.118 \)

- **Stall**
  - \( CPI = 0.2 \times 4 = 0.8 \)
  - \( CT = 1 \)
  - \( ET = 1.4 \)
  - Speed up = \( 1.4/1.118 = 1.18 \)

What if this we 20?
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 + 20) + .8*1 = 1.64
- CT = 1.1
- ET = 1.804

Stall

- CPI = .2*(1 + 20) + .8*1 = 5
- CT = 1
- ET = 5

- Speed up = 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.
Today

• Aeronautical engineering practicum
• Quiz (sort of)
• Midterm Recap
• Dynamic branch prediction
Grade distribution

<table>
<thead>
<tr>
<th>Grade</th>
<th># of students</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>F+</td>
<td>6.2</td>
</tr>
<tr>
<td>D-</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>18</td>
</tr>
<tr>
<td>D+</td>
<td>25</td>
</tr>
<tr>
<td>D+</td>
<td>25</td>
</tr>
<tr>
<td>C</td>
<td>31</td>
</tr>
<tr>
<td>C</td>
<td>37</td>
</tr>
<tr>
<td>C+</td>
<td>43</td>
</tr>
<tr>
<td>B-</td>
<td>55</td>
</tr>
<tr>
<td>B</td>
<td>60</td>
</tr>
<tr>
<td>B+</td>
<td>63</td>
</tr>
<tr>
<td>B+</td>
<td>63</td>
</tr>
<tr>
<td>A-</td>
<td>68</td>
</tr>
<tr>
<td>A</td>
<td>74</td>
</tr>
<tr>
<td>A+</td>
<td>75</td>
</tr>
</tbody>
</table>
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 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?

- Cons?
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?
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 it’s own bit in a table
  • How big does the table need to be?

• Look up the prediction bit for the branch

• Pros:

• Cons:
Dynamic Prediction 2: A table of bits

• Give each branch it’s own bit in a table
  • How big does the table need to be?

• Look up the prediction bit for the branch

• Pros: It can differentiate between branches. Bad behavior by one won’t mess up others…. mostly.

• Cons:
Dynamic Prediction 2: A table of bits

• Give each branch it’s 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.

• Cons:
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.

• 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++) {
    }
}

• What’s the accuracy for the inner loop’s branch?
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>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</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>
</tbody>
</table>

• What’s the accuracy for the inner loop’s branch?
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>1</td>
<td>taken</td>
<td>not taken</td>
<td>take</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>
</tbody>
</table>

50% or 2 per loop

• What’s the accuracy for the inner loop’s branch?
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 out mind
Dynamic Prediction 3: A table of counters

\[
\text{for}(i = 0; i < 10; i++) \ \{ \\
\quad \text{for}(j = 0; j < 4; j++) \ \{ \\
\quad \}
\}
\]

• What’s the accuracy for the inner loop’s branch? (start in weakly taken)
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 state</th>
<th>State prediction</th>
<th>New state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>strongly taken</td>
<td>taken</td>
</tr>
<tr>
<td>1</td>
<td>taken</td>
<td>weakly taken</td>
<td>taken</td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>strongly taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

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

25% or 1 per loop
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

OK -- we miss one per loop

Good

Poor -- no help

Not applicable
Predicting Loop Branches Revisited

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

- What’s the pattern we need to identify?
• 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.
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></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td></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></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td></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></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td></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></td>
</tr>
<tr>
<td>2</td>
<td>taken</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>taken</td>
<td>10111</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>not taken</td>
<td>01111</td>
<td>not taken</td>
</tr>
</tbody>
</table>
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>
<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?

- Imagine N bits of history and a loop that executes K iterations
  - If $K \leq N$, 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.
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 \( \leq \) 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.
Performance Impact of Short History

- A loop has 5 instructions, including the branch.
- The mis-prediction penalty is 7 cycles.
- The baseline CPI is 1
- What is the speedup of the global history predictor vs the per-PC predictor if the loop executes 4 iterations and we keep 4 history bits?
- If it is executes 40 iterations and we keep 40 history bits?
Performance Impact of Short History

• A loop has 5 instructions, including the branch.
• The mis-prediction penalty is 7 cycles.
• The baseline CPI is 1
• What is the speedup of the global history predictor vs the per-PC predictor if the loop executes 4 iterations and we keep 4 history bits?
• If it is executes 40 iterations and we keep 40 history bits?

• 4 iterations
  • Per-PC mis-prediction rate is 25%
  • CPI = 1 + 0.25 * 8 = 1 + 2 = 3
  • Global history mis-prediction rate is nearly 0%
  • Global mis-prediction rate is 0%, CPI = 1
• 40 iterations
  • Per-PC mis-prediction rate is 5%
  • CPI = 1 + .025 * 8 = 1.2
Performance Impact of Short History

- A loop has 5 instructions, including the branch.
- The mis-prediction penalty is 7 cycles.
- The baseline CPI is 1
- What is the speedup of the global history predictor vs the per-PC predictor if the loop executes 4 iterations and we keep 4 history bits?
- If it is executes 40 iterations and we keep 40 history bits?

- 4 iterations
  - Per-PC mis-prediction rate is 25%
  - CPI = 1 + 0.25 * 8 = 1 + 2 = 3
  - Global history mis-prediction rate is nearly 0%
  - Global mis-prediction rate is 0%, CPI = 1

- 40 iterations
  - Per-PC mis-prediction rate is 5%
  - CPI = 1 + .025 * 8 = 1.2

With more iterations, the benefit of history decreases, so a shorter history is ok.
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

Not so great

Pretty good, as long as the history is not too long

Not applicable
Other ways of identifying branches

• Use local branch history
  • Use a table of history registers (say 128), indexed by the low-order bits of the PC.
  • Also use the PC to choose between 128 tables, each indexed by the history for that branch.
• For loops this does better than global history.
  • Foo() { for(i = 0; i < 10; i++){ } }.
  • If foo is called from many places, the global history will be polluted.
Other Ways of Identifying Branches

- All these schemes have different pros and cons and will work better or worse for different branches.
- How do we get the best of all possible worlds?
Other Ways of Identifying Branches

- All these schemes have different pros and cons and will work better or worse for different branches.
- How do we get the best of all possible worlds?

- Build them all, and have a predictor to decide which one to use on a given branch
  - For each branch, make all the different predictions, and keep track which predictor is most often correct.
  - For future branches use the prediction from that predictor.
Predicting Function Calls

- **Branch Target Buffers (BTB)**
  - The name is unfortunate, since it’s really a jump target
  - Use a table, indexed by PC, that stores the last target of the jump.
  - When you fetch a jump, start executing at the address in the BTB.
  - Update the BTB when you find out the correct destination.
Interference

- Our schemes for associating branches with predictors are imperfect.
- Different branches may map to the same predictor and pollute the predictor.
- This is called “destructive interference”
- Using larger tables will (typically) reduce this effect.