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
Stalling for Load

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

Load $s1, 0($s1)

Addi $t1, $s1, 4
To “stall” we insert a noop *in place of* the instruction and freeze the earlier stages of the pipeline.

All stages of the pipeline earlier than the stall stand still.
Stalling for Load

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

Only four stages are occupied. What’s in Mem?

All stages of the pipeline earlier than the stall stand still.

Load $s1, 0($s1)

Addi $t1, $s1, 4
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

```
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?
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?
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.
- 

Diagram:

```
Fetch → Decode → EX → Mem → Write back
```
Solution #1: Stall on branches

- Worked for loads and ALU ops.

![Instruction Pipeline Diagram]
Solution #1: Stall on branches

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

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

- Branches about about 1 in 5 instructions

- What’s the CPI for branches?

- Amdahl’s law: 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?

- Amdahl's law: \( \text{Speedup} = \frac{1}{1 + \frac{1}{2}} = 3 \)

- \( ET = \)
Performance impact of stalling

• \( ET = I \times CPI \times CT \)

• Branches about about 1 in 5 instructions

• What’s the CPI for branches?

\[
1 + 2 = 3
\]

• Amdahl’s law: Speedup =

\[
1/(.2/(1/3) + (.8)) = 0.714
\]

• \( ET = \)
Performance impact of stalling

- \[ ET = I \times CPI \times CT \]

- Branches about about 1 in 5 instructions

- What’s the CPI for branches?

\[ 1 + 2 = 3 \]

- Amdah’ls law: Speedup =

\[ 1/((.2/(1/3)) + (.8)) = 0.714 \]

- \[ ET = 1 \times (.2 \times 3 + .8 \times 1) \times 1 = 1.4 \]
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.

```assembly
bne $t2, $s0, somewhere
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?
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.
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?
Simple “static” Prediction

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

sign bit

comparison result

Instruction Memory

Fetch/Dec

Read Address

PC

Add

4

Register

Dec/Exec

File

Read Addr 1

Write Addr

Read Data 1

Read Data 2

Write Data

ALU

Data Memory

Address

Read Data

Write Data

Mem/WB

Shi<

Add

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 * CPI * 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

- \( ET = I \times CPI \times CT \)
- 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?
- \( Bt/fnt \)
  - \( CPI = 0.2 \times 0.2 \times (1 + 2) + (1 - 0.2 \times 0.2) \times 1 = 1.08 \)
  - \( CT = 1.1 \)
  - \( ET = 1.188 \)
- Stall
  - \( CPI = 0.2 \times 3 + 0.8 \times 1 = 1.4 \)
  - \( CT = 1 \)
  - \( ET = 1.4 \)
- Speed up = \( 1.4 / 1.188 = 1.17 \)
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)
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

- \( ET = I \times CPI \times CT \)
- 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
  - \( CPI = .2 \times .2 \times (1 + 2) + .9 \times 1 \)
  - \( CT = 1.1 \)
  - \( ET = 1.118 \)
- Stall
  - \( CPI = .2 \times 4 + .8 \times 1 = 1.6 \)
  - \( CT = 1 \)
  - \( ET = 1.4 \)
- Speed up = \( 1.4/1.118 = 1.18 \)

What if this were 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.
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?

- 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 (why not always?)

• 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 (why not always?)

- 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 (why not always?)

• Cons: Accuracy is still not great.
Dynamic Prediction 2: A table of bits

```c
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

```cpp
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>

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 our mind.
Dynamic Prediction 3: A table of counters

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

• What’s the accuracy for the inner loop’s branch? (start in weakly taken)
Dynamic Prediction 3: A table of counters

```c
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)
Dynamic Prediction 3: A table of counters

```c
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 map to associate a 2-bit predictor with each dynamic branch
  • Use the 2-bit predictor for each branch to make the prediction.

• The map does not need to be perfect
  • It’s ok if multiple branches share a single 2-bit predictor.
  • These “collisions” will reduce accuracy, but won’t affect correctness.

• In the previous example we associated the predictors with branches using the PC.
  • We’ll call this “per-PC” prediction.
Mapping Branches to Predictors

- Predictor selector
- Predictor index
- Predictors table(s)
- Prediction update pipeline registers
- Branch outcome
- Weather report
- Program Counter
- Branch History
- Branch prediction inputs
- Fetch
- Decode
- EX
- Mem
- Write back
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 particular sub class
    • t->SomeVirtualFunction() // will usually call the same function
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?
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.
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>
</tbody>
</table>

outer loop branch

<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>11110</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>

outer loop branch

<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>11110</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>
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 ≤ 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.
  Long is not always good: we might get confused by extraneous information

• 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.
Global History

- Correlated Control
  - $a = 11; \ b = <\text{something usually larger than } a>$
  - if ($a > 10$) {}
  - if ($b > 10$) {}

- Global history notice the correlation and predict the second branch.

- Another example:
  - if(rare case) {}
  - ...
  - if(rare case) {}
  - ...
  - if(race case) {}
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.
  - Correlated 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 (or too short)**
- **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.
Local history

• For some loops local history is better
  • Foo() { for(i = 0; i < 10; i++) { unpredictable branches } }.
  • If foo is called from many places, or the loop body has unpredictable control, the global history will be polluted.

• Local history gives the loops branch it’s own history.
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.
  • Do this on a per-branch basis.
• This has been studied to death.
  • For good reason. Without good branch prediction, performance drops by at least 90%.
Other Prediction Schemes

- The ACM Portal (an enormous index of CS research papers) lists 9,000 papers on branch prediction
- Neural networks
- Data compressions
- Tournament-based predictors
- Predictors that use analog circuits to make predictions
- Variable-length history prediction
- Confidence prediction
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.
Associating Predictors with Branches: BTB

- When is branch behavior predictable?  not applicable
  - 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  Pretty good
Predicting Returns

- Function call returns are hard to predict
  - For every call site, the return address is different
  - The BTB will do a poor job, since it’s based on PC
- Instead, maintain a “return stack predictor”
  - Keep a stack of return targets
  - `jal` pushes $ra onto the stack
  - Fetch predicts the target for `return` instruction by popping an address off the stack.
  - Doesn’t work in MIPS, because there is no return inst.
  - Not always correct because the return stack might not be large enough.
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.
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 5 history bits?
- If it is executes 40 iterations and we keep 41 history bits?
4 iterations

- Per-PC mis-prediction rate is 25%, since per-PC prediction results in one mispredict execution of the loop.
- 1/5 of the dynamic instructions are branches
- F = fraction of instructions that cause a mispredict
- F = 1/5 * 0.25 = 0.05
- CPI = 1 * (1-F) + (1+ MissPenalty)*0.05
- CPI = 1*(1-0.05) + (1+7)*(0.05) = 1.35
- Global history mis-prediction rate is 0%, CPI = 1
- SpeedUp = 1.35

40 iterations

- Per-PC mis-prediction rate is 2.5%, since per-PC prediction results in one mispredict execution of the loop.
- 1/5 of the dynamic instructions are branches
- F = fraction of instructions that cause a mispredict
- F = 1/5 * 0.025 = 0.005
- CPI = 1 * (1-F) + (1+ MissPenalty)*0.005
- CPI = 1*(1-0.005) + (1+7)*(0.005) = 1.035
- Global history mis-prediction rate is 0%, CPI = 1
- SpeedUp = 1.035
4 iterations

- Per-PC mis-prediction rate is 25%, since per-pc prediction results in one mispredict execution of the loop.
- 1/5 of the dynamic instructions are branches
- F = fraction of instructions that cause a mispredict
  - F = 1/5 * 0.25 = 0.05
  - CPI = 1 * (1-F) + (1+ MissPenalty)*0.05
  - CPI = 1*(1-0.05) + (1+7)*(0.05) = 1.35
- Global history mis-prediction rate is 0%, CPI = 1
  - SpeedUp = 1.35

40 iterations

- Per-PC mis-prediction rate is 2.5%, since per-pc prediction results in one mispredict execution of the loop.
- 1/5 of the dynamic instructions are branches
- F = fraction of instructions that cause a mispredict
  - F = 1/5 * 0.025 = 0.005
  - CPI = 1 * (1-F) + (1+ MissPenalty)*0.005
  - CPI = 1*(1-0.005) + (1+7)*(0.005) = 1.035
- Global history mis-prediction rate is 0%, CPI = 1
  - SpeedUp = 1.035

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