Midterm Question 1-5

• Questions about 1-5: Ask tomorrow in the discussion session.
• Midterms available tomorrow during discussion session or from the TAs during office hours.
6. Consider the following MIPS code:

```
ori $s0, $0, 0
ori $t0, $0, $a0
check:
bge $s0, $t0, done
nop
lw $t1, 0($s3)
nop
add $s1, $s1, $t1
addi $s0, $s0, 1
addi $s3, $s3, 4
j check
 nop
done:
```

5a. Write your C code here:

```
for(i = 0; i < N; i++)
x+=A[i];
```

```
for(i = 0; i < N; i++)
x+=A[i++];
```

b. What is the static instruction count for this code? Assuming $a0$ has value 3, what is the dynamic instruction count?

<table>
<thead>
<tr>
<th>Static inst count</th>
<th>Dynamic inst count</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>31</td>
</tr>
</tbody>
</table>

7 branches
3 loads
21 ALU
CPI=1.238
c. Rewrite the assembly code to minimize dynamic instruction count.

### 5c. Write your optimized MIPS code here:

```
ori $s0, $0, 0
check:
  bge $s0, $a0, done
  addi $s0, $s0, 1
  lw $t1, 0($s3)
  addi $s3, $s3, 4
  j check
  add $s1, $s1, $t1

done:
```

7 branches
3 loads
11 ALU

CPI = (7*1.2 + 3*3 + 11*1)/21 = 1.352

---

d. What are the new static and dynamic instruction counts?

<table>
<thead>
<tr>
<th>Static inst count</th>
<th>Dynamic inst count</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>21</td>
</tr>
</tbody>
</table>
Question 6

e. Assuming the following average CPIs for each instruction type, what is the speedup due to reduced CPI? What is the speedup due to reduced instruction count? What is the total speedup?

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>1</td>
</tr>
<tr>
<td>Memory</td>
<td>3</td>
</tr>
<tr>
<td>Branch</td>
<td>1.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduced CPI</td>
<td>0.916</td>
</tr>
<tr>
<td>Reduced IC</td>
<td>1.476</td>
</tr>
<tr>
<td>Total</td>
<td>1.352</td>
</tr>
</tbody>
</table>
Question 7

7. For the following C program:

```c
while (((x >> 1) & 0x1) != 0) {
    x = x >> 1;
    count++;
}
```

Convert it to MIPS assembly by filling in the banks below (sr is the MIPS’ shift right logical instruction):

```
addi $3, $zero, __
loop:
    srl $4, $1, __ (or $3 is fine)__
    and $5, __4__, $3
    beq $5, __zero__, done
    add __zero, $zero, $zero
    add $1, __4__, __zero
    addi$2, __2__, __1__
    beq __zero__, __zero__, __loop__

done:
```
CSE 141 Midterm Exam

2014 Spring

Professor Steven Swanson

Please write your name at the top of each page. This is a closed book, closed notes exam. No outside material may be used. You may use a calculator. Please enter your answers in the spaces provided. This allows us to grade your responses more accurately. Show your work. You will get more partial credit that way. If you need additional space attach scratch paper and clearly label which question your work refers to (e.g. 1.A). Draw a smiley face at the bottom of this page to get 1 point of extra credit. If you have any question, please raise your hand. Good luck!
# Midterm Grades

<table>
<thead>
<tr>
<th>question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7Smiley</th>
</tr>
</thead>
<tbody>
<tr>
<td>average score</td>
<td>77%</td>
<td>62%</td>
<td>69%</td>
<td>93%</td>
<td>95%</td>
<td>68%</td>
<td>87%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Score</th>
<th># of students</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>C-</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C+</td>
<td>1</td>
</tr>
<tr>
<td>B-</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>B+</td>
<td>2</td>
</tr>
<tr>
<td>A-</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A+</td>
<td>1</td>
</tr>
</tbody>
</table>
Midterm Grade Questions

• Math errors -- i.e., we added up your points wrong
  • Come to office hours.

• Other errors
  • E-mail us requesting a regrade, and explaining why you think there was an error
    • You must explain why you think there was an error
    • You must send the email.
    • You cannot just show up at office hours.
  • We will regrade your entire exam (i.e., your grade could go down)
  • You have until 1 week from tomorrow to send us the email.

• No exceptions.
• We photocopied a random sampling of the exams before turning them back to you.
Key Points: Control Hazards

- Control hazards occur when we don’t know what the next instruction is.
- Caused by branches and jumps.
- 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.
Computing the PC Normally

- Non-branch instruction
  - \( \text{PC} = \text{PC} + 4 \)
- When is PC ready?
Fixing the Ubiquitous Control Hazard

• We need to know if an instruction is a branch in the fetch stage!
• How can we accomplish this?

**Solution 1:** Partially decode the instruction in fetch. You just need to know if it’s a branch, a jump, or something else.

**Solution 2:** We’ll discuss later.
Computing the PC Normally

- Pre-decode in the fetch unit.
  - PC = PC + 4
- The PC is ready for the next fetch cycle.

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
sub $t2, $s0, $t3
sub $t2, $s0, $t3
```
Computing the PC for Branches

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

- Jump instructions
  - `jr $s1` -- jump register
  - PC = $s1
- When is the value ready?

\[
sll \ $s4, \ $t6, \ $t5
\]

\[
jr \ $s4
\]

\[
add \ $s0, \ $t0, \ $t1
\]
Dealing with Branches: Option 0 -- stall

sll $s4, $t6, $t5

bne $t2, $s0, somewhere

add $s0, $t0, $t1

and $s4, $t0, $t1

• What does this do to our CPI?
Option 1: The compiler

• Use “branch delay” slots.
• The next N instructions after a branch are always executed
• How big is N?
  • For jumps?
  • For branches?
• Good
  • Simple hardware
• Bad
  • N cannot change.
Delay slots.

**Take**

```
bne $t2, $s0, somewhere
```

**Branch Delay**

```
add $t2, $s4, $t1
add $s0, $t0, $t1
...
somewhere:
sub $t2, $s0, $t3
```
But MIPS Only Has One Delay Slot!

- The second branch delay slot is expensive!
  - Filling one slot is hard. Filling two is even more so.
- Solution!: Resolve branches in decode.
For the rest of this slide deck, we will assume that MIPS has no branch delay slot.

If you have questions about whether part of the homework/test/quiz makes this assumption ask or make it clear what you assumed.
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?
- Branch prediction vocabulary
  - Prediction -- a guess about whether a branch will be taken or not taken
  - Misprediction -- a prediction that turns out to be incorrect.
  - Misprediction rate -- fraction of predictions that are incorrect.
• We start the add, and then, when we discover the branch outcome, we *squash* it.

• Also called “flushing the pipeline”

• Just like a stall, flushing one instruction increases the branch’s CPI by 1
Flushing the Pipeline

- When we flush the pipe, we convert instructions into noops
  - Turn off the write enables for write back and mem stages
  - Disable branches (i.e., make sure the ALU does raise the branch signal).
- Instructions do not stop moving through the pipeline
- For the example on the previous slide the “inject_nop_decode_execute” signal will go high for one cycle.
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
  • The best of both worlds!
  • Most loops have have a backward branch at the bottom, those will predict taken
  • Others (non-loop) branches will be not-taken.
Implementing Backward taken/forward not taken (BTFNT)

• A new “branch predictor” module determines what guess we are going to make.

• The BTFNT branch predictor has two inputs
  • The sign of the offset -- to make the prediction
  • The branch signal from the comparator -- to check if the prediction was correct.

• And two output
  • The PC mux selector
    • Steers execution in the predicted direction
    • Re-directs execution when the branch resolves.
  • A mis-predict signal that causes control to flush the pipe.
Performance Impact (Part 1)

- BTFTN has a misprediction rate of 20%.
- Branches are 20% of instructions.
- Mispredictions increase the CPI of branches by 1.
- What is the new CPI (assume baseline CPI = 1)?

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.20</td>
</tr>
<tr>
<td>B</td>
<td>1.04</td>
</tr>
<tr>
<td>C</td>
<td>0.96</td>
</tr>
<tr>
<td>D</td>
<td>0.83</td>
</tr>
<tr>
<td>E</td>
<td>0.80</td>
</tr>
</tbody>
</table>
Performance Impact (ex 1)

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

- BTFTN is has a misprediction rate of 20%.
- Branches are 20% of instructions
- Changing the front end increases the cycle time by 10%
- What is the speedup of BTFTNT compared to just stalling on every branch?

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0.95</td>
</tr>
<tr>
<td>C</td>
<td>1.05</td>
</tr>
<tr>
<td>D</td>
<td>1.15</td>
</tr>
<tr>
<td>E</td>
<td>1.7</td>
</tr>
</tbody>
</table>
Performance Impact (ex 1)

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

- 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?

- $Bt/Fnt$
  - $CPI = 0.2 \times 0.2^1 (1 + 1) + (1 - 0.2 \times 0.2)^1 = 1.04$
  - $CT = 1.1$
  - $IC = IC$
  - $ET = 1.144$

- Stall
  - $CPI = 0.2 \times 2 + 0.8 \times 1 = 1.2$
  - $CT = 1$
  - $IC = IC$
  - $ET = 1.2$

- Speed up $= 1.2 / 1.144 = 1.05$
The Branch Delay Penalty

• The number of cycle between fetch and branch resolution is called the “branch delay penalty”
  • It is the number of instruction that get flushed on a misprediction.
  • It is the number of extra cycles the branch gets charged (i.e., the CPI for mispredicted branches goes up by the penalty for)
Performance Impact

- \( ET = I \times CPI \times CT \)
- Our current design resolves branches in decode, so the branch delay penalty is 1 cycle.
- If removing the comparator from decode (and resolving branches in execute) would reduce cycle time by 20%, would it help or hurt performance?
  - Mis predict rate = 20%
  - Branches are 20% of instructions

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Help</td>
</tr>
<tr>
<td>B</td>
<td>Hurt</td>
</tr>
<tr>
<td>C</td>
<td>No difference</td>
</tr>
<tr>
<td>D</td>
<td>Don’t answer this</td>
</tr>
<tr>
<td>E</td>
<td>Or this… Seriously…</td>
</tr>
</tbody>
</table>
Performance Impact (ex 2)

- ET = I * CPI * CT
- Our current design resolves branches in decode, so the branch delay penalty is 1 cycle.
- If removing the comparator from decode (and resolving branches in execute) would reduce cycle time by 20%, would it help or hurt performance?
  - Mis predict rate = 20%
  - Branches are 20% of instructions

 Resolve in Decode
- CPI = 0.2*0.2*(1 + 1) + (1-.2*.2)*1 = 1.04
- CT = 1
- IC = IC
- ET = 1.04

 Resolve in execute
- CPI = 0.2*0.2*(1 + 2) + (1-.2*.2)*1 = 1.08
- CT = 0.8
- IC = IC
- ET = 0.864
- Speedup = 1.2
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

• Branches take 19 cycles to resolve
• Identifying a branch takes 4 cycles.
• Stalling is not an option.
• 80% branch prediction accuracy is also not an option.
• Not quite as bad now, but BP is still very important.
• Wait, it gets worse!!!!
Performance Impact (ex 1)

• \( ET = I \times CPI \times CT \)

• 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?

\[ \text{Btfnt} \]
- \( CPI = 0.2 \times 0.2 \times (1 + 1) + (1 - 0.2 \times 0.2) \times 1 = 1.04 \)
- \( CT = 1.144 \)
- \( IC = IC \)
- \( ET = 1.144 \)

• Stall
- \( CPI = 0.2 \times 2 + 0.8 \times 1 = 1.2 \)
- \( CT = 1 \)
- \( IC = IC \)
- \( ET = 1.2 \)

• Speed up = \( 1.2 / 1.144 = 1.05 \)

What if this were 20 instead of 1?

Branches are relatively infrequent (~20% of instructions), but Amdahl’s Law tells that we can’t completely ignore this uncommon case.
Performance Impact (ex 1) revisited

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

- 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 \( \frac{Bt}{Fnt} \) compared to just stalling on every branch?
- \( Btfnt \)
  - \( CPI = 0.2 \times 0.2 \times (1 + 20) + (1 - 0.2 \times 0.2) \times 1 = 1.8 \)
  - \( CT = 1.144 \)
  - \( IC = IC \)
  - \( ET = 2.17 \)
- Stall
  - \( CPI = 0.2 \times 21 + 0.8 \times 1 = 5 \)
  - \( CT = 1 \)
  - \( IC = IC \)
  - \( ET = 1.2 \)

- Speed up = \( \frac{5}{2.17} = 2.3 \)

Branches are relatively infrequent (~20% of instructions), but Amdahl’s Law tells that we can’t completely ignore this uncommon case.
BTFNT is not nearly good enough!

14 branches @ 80% accuracy = \(0.8^{14} = 4.3\%\)
14 branches @ 90% accuracy = \(0.9^{14} = 22\%\)
14 branches @ 95% accuracy = \(0.95^{14} = 49\%\)
14 branches @ 99% accuracy = \(0.99^{14} = 86\%\)
Dynamic Branch Prediction

• Long pipes demand higher accuracy than static schemes can deliver.
• Instead of making the guess once (i.e. statically), make it every time we see the branch.
• Many ways to predict dynamically
  • We will focus on predicting 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 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 branch did.
• Pros?

Dead simple. Keep a bit in the fetch stage that is the direction of the last branch. 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.
Accuracy of 1-bit counter

• Consider the following code:
  
i = 0;
do {
    if( i % 3 != 0 ) // Branch Y, taken if i % 3 == 0
      a[i] *= 2;
    a[i] += i;
  } while ( ++i < 100 ) // Branch X

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>Last branch (x) bit</th>
<th>Actual (y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

A. 0%
B. 33%
C. 67%
D. 100%
The 1-bit Predictor

- Predict this branch will go the same way as the result of the last time this branch executed
  - 1 for taken, 0 for not taken

PC = 0x400420

<table>
<thead>
<tr>
<th>Index</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>0x20</td>
<td>1</td>
</tr>
<tr>
<td>0x24</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>1</td>
</tr>
</tbody>
</table>

How big should this table be?

What about conflicts?

Simple 1-bit Predictor
Accuracy of 1-bit counter

• Consider the following code:
  
i = 0;
  do {
      if( i % 3 != 0) // Branch Y, taken if i % 3 == 0
          a[i] *= 2;
      a[i] += i;
  } while ( ++i < 100) // Branch X

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.
Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%
2-bit counter

- A 2-bit counter for each branch
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4

![Diagram showing the 2-bit predictor]

- **Take**: 0x400420
- **Table**: 2-bit predictor
  
<table>
<thead>
<tr>
<th>Index</th>
<th>predict</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>11</td>
</tr>
<tr>
<td>0x20</td>
<td>10</td>
</tr>
<tr>
<td>0x24</td>
<td>00</td>
</tr>
<tr>
<td>...</td>
<td>01</td>
</tr>
</tbody>
</table>

**Taken!**
Performance of 2-bit counter

- 2-bit state machine for each branch

```
for(i = 0; i < 10; i++)
{
    sum += a[i];
}
```

90% prediction rate!

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
  - $1 + 20% \times (1 - 90\%) \times 2 = 1.04$
Accuracy of 2-bit counter

Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

What is the prediction accuracy of branch Y using 2-bit predictors (if all counters start with 00). Choose the closest one. Assume unlimited BTB entries.

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

A. 0%
B. 33%
C. 67%
D. 100%
Make the prediction better

• Consider the following code:
  
i = 0;
  do {
      if( i % 3 != 0)  // Branch Y,
          taken if i % 3 == 0
      a[i] *= 2;
      a[i] += i;
  } while ( ++i < 100)  // Branch X

Can we capture the pattern?

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>NT</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>NT</td>
</tr>
</tbody>
</table>
Predict using history

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Each entry in the history table has its own counter.

n-bit GHR = 101 (T, NT, T)

<table>
<thead>
<tr>
<th>Index</th>
<th>Predict</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>01</td>
</tr>
<tr>
<td>001</td>
<td>11</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>111</td>
<td>10</td>
</tr>
</tbody>
</table>

2^n entries

Taken!
Performance of global history predictor

Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0) // Branch Y,
        taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
    // Branch Y
} while ( ++i < 100) // Branch X
```

Assume that we start with a 4-bit GHR = 0, all counters are 10.

Nearly perfect after this
Accuracy of global history predictor

- Consider the following code:
  ```c
  sum = 0;
i = 0;
do {
    if(i % 2 == 0) // Branch Y, taken if i % 2 != 0
      sum+=a[i];
  } while ( ++i < 100) // Branch X
  Which of predictor performs the best?
  
  A. Predict always taken
  B. Predict alway not-taken
  C. 1-bit predictor
  D. 2-bit predictor
  E. 4-bit global history with 2-bit counters