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
  • \( PC = PC + 4 \)
• When is PC ready?

```plaintext
add $s0, $t0, $t1
sub $t2, $s0, $t3
sub $t2, $s0, $t3
sub $t2, $s0, $t3
```
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?

```
s11 $s4, $t6, $t5
bne $t2, $s0, somewhere
add $s0, $t0, $t1
and $s4, $t0, $t1
```
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.

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

```
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.
Predict Not-taken

- 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?
  - Predict not-taken
    - Pros?
    - 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.

Loops are commons
Not all branches are for loops.
Basic Pipeline Recap

- The PC is required in Fetch
- For branches, it’s not know till *decode*.

Branches only, one delay slot, simplified ISA, no control.
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 (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 BTFTNT compared to just stalling on every branch?
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 = \( 0.2 \times 0.2 \times (1 + 1) + (1 - 0.2 \times 0.2) \times 1 = 1.04 \)
  • 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)

$$\text{MIPS} \times \text{Branch Delay} = 1$$
Performance Impact (ex 2)

- $ET = I \times CPI \times CT$
- 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
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.
Performance Impact (ex 1)

- 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 + 1) + (1-.2*.2)*1 = 1.04
  - CT = 1.144
  - IC = IC
  - ET = 1.144
- Stall
  - CPI = .2*2 + .8*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 \( Bt/Fnt \) compared to just stalling on every branch?
- \( Bt/Fnt \)
  - \( CPI = 0.2 \times 0.2 \times (1 + 20) + (1 - 0.2 \times 0.2) \times 1 = 1.8 \)
  - \( CT = 1.144 \)
  - \( IC = IC \)
  - \( ET = 1.144 \)

- Stall
  - \( CPI = 0.2 \times 21 + 0.8 \times 1 = 5 \)
  - \( CT = 1 \)
  - \( IC = IC \)
  - \( ET = 1.2 \)
  - Speed up = \( 5/1.8 = 2.7 \)

Branches are relatively infrequent (~20% of instructions), but Amdahl’s Law tells that we can’t completely ignore this uncommon case.
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.
  - 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
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.
Dynamic Prediction 2: A table of bits

• Give each branch its own bit in a table
  • Look up the prediction bit for the branch
  • 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

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

• Cons:
  Accuracy is still not great.
Branch Prediction Trick #1

- Associating prediction state with a particular branch.
- We would like to keep separate prediction state for every *static* branch.
  - In practice this is not possible, since there are a potentially unbounded number of branches.
- Instead, we use a heuristic to associate prediction state with a branch.
  - The simplest heuristic is to use the low-order bits of the PC to select the prediction state.

![Diagram showing the flow from PC to prediction state]

129
### Dynamic Prediction 2: A table of bits

```c
while (1) {
    for(j = 0; j < 4; j++) {
        // branch at address 0x100A
    }
}
```

<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 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.

- The predictor waits one prediction before it changes its prediction.
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” or “local” prediction.
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><strong>taken</strong></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
Predicting Loop Branches Revisited

```c
while (1) {
    for (j = 0; j < 3; 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>111110</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

• 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

• 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.