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

```
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

- 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?

```c
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.

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

- **Branch Delay**
  - 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

- \(ET = I \times CPI \times CT\)
- 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 = \(1/(0.2/(1/3) + 0.8) = 0.714\)
- \(ET = 1 \times (0.2 \times 3 + 0.8 \times 1) \times 1 = 1.4\)
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.
Simple “static” Prediction

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

Loops are commons

Not all branches are for loops.

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

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 * 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?
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**

- \( CPI = 0.2 \times 0.2 \times (1 + 2) + (1 - 0.2 \times 0.2) \times 1 = \)
- \( CT = 1.1 \)
- \( ET = 1.188 \)

**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?
- \( Btfnt \)
  - \( CPI = 0.2 \times 2 \times 1 + 0.9 \times 1 \)
  - \( CT = 1.1 \)
  - \( ET = 1.118 \)
- Stall
  - \( CPI = 0.2 \times 4 + 0.8 \times 1 = 1.6 \)
  - \( CT = 1 \)
  - \( ET = 1.4 \)
  - Speed up = \( 1.4 / 1.118 = 1.18 \)

What if this we 20?
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**

- \[ CPI = 0.2 \times 0.2 \times (1 + 20) + 0.8 \times 1 = 1.64 \]
- \[ CT = 1.1 \]
- \[ ET = 1.804 \]

**Stall**

- \[ CPI = 0.2 \times (1 + 20) + 0.8 \times 1 = 5 \]
- \[ CT = 1 \]
- \[ ET = 5 \]

- 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.