Branch Hazards

or

“Which way did he go?”

Control Dependence

- Just as an instruction will be dependent on other instructions to provide its operands (data dependence), it will also be dependent on other instructions to determine whether it gets executed or not (branch dependence or control dependence).
- Control dependences are particularly critical with conditional branches.

```
add $5, $3, $2
sub $6, $5, $2
beq $6, $7, somewhere
and $9, $6, $1

... somewhere: or $10, $5, $2

... add $12, $11, $9
```

Branch Hazards

- Branch dependences can result in branch hazards (when they are too close to be handled correctly in the pipeline).
Dealing With Branch Hazards

- Hardware
  - until you know which direction
  - reduce hazard through of branch direction
  - which direction
    - assume not taken (easiest)
    - more educated guess based on history (requires that you know it is a branch before it is even decoded!)
- Hardware/Software
  - nops, or instructions that get executed either way (delayed branch).

Stalling for Branch Hazards

- Seems wasteful, particularly when the branch isn’t taken.
- Makes all branches cost 4 cycles.

Assume Branch Not Taken

- works pretty well when you’re right
**Assume Branch Not Taken**

- same performance as stalling when you’re wrong

```
beq $4, $0, there
and $12, $2, $5
or ...
add ...
```

there: sub $12, $4, $2

- Performance depends on percentage of time you guess right.
- Flushing an instruction means to prevent it from changing any permanent state (, , ).
  - sounds a lot like a bubble...
  - But notice that we need to be able to insert those bubbles later in the pipeline

**Reducing the Branch Delay**

- can easily get to 2-cycle stall

**Stalling for Branch Hazards**
Reducing the Branch Delay

• Harder, but possible, to get to 1-cycle stall

Stalling for Branch Hazards

beq $4, $0, there
and $12, $2, $5
or ...
add ...
sw ...

The Pipeline with flushing for taken branches

• Notice the IF/ID flush line added.

Eliminating the Branch Stall

• There’s no rule that says we have to see the effect of the branch immediately. Why not wait an extra instruction before branching?
• The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches.
• The instruction after a conditional branch is always executed in those machines, regardless of whether the branch is taken or not!
Branch Delay Slots

- The branch delay slot is only useful if you can find something to put there.
- If you can’t find anything, you must put a *nop* to insure correctness.

---

Branch delay slot instruction (next instruction after a branch) is executed even if the branch is taken.

Filling the branch delay slot

- This works great for this implementation of the architecture, but becomes a permanent part of the ISA.
- What about the MIPS R10000, which has a 5-cycle branch penalty, and executes 4 instructions per cycle? Or the Pentium 4, with a 19-cycle branch penalty and 4 instructions per cycle?
Branch Prediction

- Always assuming the branch is not taken is a crude form of *branch prediction*.
- What about loops that are *taken* 95% of the time?
  - we would like the option of assuming *not taken* for some branches, and *taken* for others, depending on ???

Two-bit predictors give better loop prediction

```
for (i=0;i<10;i++) {
    ...
    ...
    add $i, $i, #1
    beq $i, #10, loop
}
```

Two different 2-bit schemes

Strongly Taken

Weakly Taken

Weakly Not Taken

Strongly Not Taken

Decrement when not taken

Increment when taken
Branch History Table

- has limited size
- 2 bits by N (e.g. 4K)
- uses low bits of branch address to choose entry

- what happens when table too small?
- what about even/odd branch?

2-bit prediction accuracy

Can We Do Better?

- Can we get more information dynamically than just the history of this branch?
- We can look at patterns (2-level predictor) for a particular branch.
  - last eight branches 00100100, then it is a good guess that the next one is “1” (taken)

- even/odd branch?

Can We Do Better?

- Correlating Branch Predictors also look at other branches for clues
  - if (i == 0)
  - ... if (i > 7)
- Typically use two indexes
  - Global history register --> history of last m branches (e.g., 0100011)
  - branch address
Correlating Branch Predictors

- The global history register is a shift register that records the last \( n \) branches (of any address) encountered by the processor.

Two-level correlating branch predictors

- Can use both the PC address and the GHR

Performance of 2-level Correlating Branch Prediction

Are we happy yet????

- Combining branch predictors use multiple schemes and a voter to decide which one typically does better for that branch.
Pipeline performance

```
loop:    lw $15, 1000($2)
        add $16, $15, $12
        lw $18, 1004($2)
        add $19, $18, $12
        beq $19, $0, loop:
```

Control Hazards -- Key Points

- Control (or branch) hazards arise because we must fetch the next instruction before we know if we are branching or where we are branching.
- Control hazards are detected in hardware.
- We can reduce the impact of control hazards through:
  - early detection of branch address and condition
  - branch prediction
  - branch delay slots