Notes and Updates

• Office hours for Dr. Simon: **Tues 9-10**
  – She’ll be out of town for the rest of Week 8
  – A good week to attend discussion section!

• Today:
  – Finishing branch hazard options
  – Branch prediction
    ▪ More material than is in book!
  – Exceptions in pipelines

• Questions?

Dealing With Branch Hazards

• Hardware
  – stall until you know which direction
  – reduce hazard through earlier computation of branch direction
  – guess 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).
Overview: Dealing With Branch Hazards

- **Software solution**
  - insert *no-ops or independent instructions*

- **Hardware solutions**
  - *stall* until you know which direction branch goes
  - guess which direction, start executing chosen path (but be prepared to undo any mistakes!)
    - **static branch prediction**: base guess on instruction type
    - **dynamic branch prediction**: base guess on execution history
  - reduce the branch delay

- **Software/hardware solution**
  - *delayed branch*: Always execute instruction after branch.
    - Compiler puts something useful (or a no-op) there.

---

Reducing the Branch Delay

- can easily get to 2-cycle stall
Stalling for Branch Hazards

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

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 ________________ 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 Slot

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

- The branch delay slot is only useful if you can find something to put there.
  - Need earlier instruction that doesn’t affect the branch
  - OR:

- If you can’t find anything, you must put a *nop* to insure correctness.

- Worked well for early RISC machines.
  - Doesn’t help recent processors much
  - E.g. MIPS R10000, has a 5-cycle branch penalty, and executes 4 instructions per cycle.
  - The Pentium 4 branch penalty is 19 cycles!

- Meanwhile, delayed branch is a permanent part of the ISA.
Branch Prediction

• Always assuming the branch is not taken is a crude form of \textit{branch prediction} (e.g., static branch prediction)
• What about loops that are \textit{taken} 95\% of the time?
  – we would like the option of assuming \textit{not taken} for some branches, and \textit{taken} for others.

Dynamic Branch Prediction

• Makes a prediction for each execution of the branch
• Based on what?
• What’s the theoretical upper bound?
Dynamic Branch Prediction

This '1' bit means, "the last time the program counter ended with 0100 and a beq instruction was seen, the branch was taken." Hardware guesses it will be taken again.

Dean Tullsen

Branch Prediction

1^st iteration Branch Taken (predicted not taken) History -> 1

Dean Tullsen
Branch Prediction

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

2nd iteration
Branch Taken
(predicted _____)
History -> ___

3rd iteration
Branch Taken
(predicted _____)
History -> ___
Branch Prediction

10th iteration
Branch Not Taken
(predicted ______)  
History -> ___

for (i=0;i<10;i++) {
  ...
  ...
}

1st iteration again
Branch Taken
(predicted _____)
History -> ___

for (i=0;i<10;i++) {
  ...
  ...
}

  ...
  ...
  add $i, $i, #1
  bne $i, #10, loop

CSE 141  
Dean Tullsen
Two-bit predictors give better loop prediction

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

Two different 2-bit schemes
2-bit Branch History Table

- has limited size
- 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?

1st iteration
Branch Taken
(predicted ________)
History -> 10
### 2-bit Branch History Table

#### 2nd iteration
- Branch Taken
- (predicted ________)
- History -> ___

#### 3rd iteration
- Branch Taken
- (predicted ________)
- History -> ___

---

**CSE 141**

Dean Tullsen
2-bit Branch History Table

10th iteration
Branch Not Taken
(predicted ________)
History -> __

1st iteration again
Branch Taken
(predicted ________)
History -> __
2-bit prediction accuracy

Is this good enough?

Basic Problems with Dynamic Branch Prediction

- Aliasing
  - When more than one branch indexes into the same location in the BHT
    - Comes in 3 flavors:
      - Constructive
        - Where branch A and branch B map to the same BHT entry
        - A’s behavior leaves the prediction JUST RIGHT for B
        - If A and B didn’t alias, B would mispredict
      - Destructive
    - Neutral
      - Where A and B alias, but both predict the “same way”
Basic Problems with Dynamic Branch Prediction

- Some branches aren’t predictable based on previous behavior
- “Global” branch prediction schemes use recent dynamic branch traces to help make predictions
  - Global history register
  - Different lengths

Loop:

a = rand();
b = 5;
if (a>b)
c = 7;
if (c == 7)
  blah
c = 0;
beq 10 Loop

Note:
Assume a true if condition is a taken branch

2-level branch predictors

- Combine Global History Register with PC
  - Use to index into BHT

Loop:

a = rand();
b = 5;
if (a>b)
c = 7;
if (c == 7)
  blah
c = 0;
beq 10 Loop

Note:
Assume a true if condition is a taken branch
Branch Hazards -- Key Points

• Branch (or control) hazards arise because we must fetch the next instruction before we know if we are branching or not.
• Branch hazards are detected in hardware.
• We can reduce the impact of branch hazards through:
  – computing branch target and testing early
  – branch delay slots
  – branch prediction – static or dynamic
  – But these schemes have difficulties:
    ▪ Aliasing
    ▪ Difficult prediction patterns

Exceptions and Pipelines: More Complexity

• Exceptions (overflow, invalid instruction, etc) also must be caught in a pipelined world
  – Save the PC (into EPC)
  – Save the Cause
  – Transfer control to Exception Handling routine (set PC to 0)
So, where's the exception?

Dealing with Exceptions in a Pipeline

- Save the PC of the offending instruction (they use PC+4 in book, don’t worry about that detail too much)
- FLUSH the instructions following the offending instruction
- Start fetching from address “0”.

CSE 141

Dean Tullsen
Pipelines with more types of exceptions

- In pipelines that support more types of exceptions, it can be difficult to figure out which instruction to handle
  - What if multiple exceptions (from different instructions in different stages) happen at the same time?
  - Choose which to handle first.
- Some architectures don’t promise to get it perfectly correct
  - Imprecise interrupts
- Alternative: precise interrupts