Contents

- Branch delay slot
  - Performance cost
  - Re-ordering instructions to fill it
- Predicting not taken
  - Killing instructions when wrong
- Other static branch predictors
- Dynamic branch predictors
  - Loop examples
  - How to store the predictions: associative memories
- Exceptions
  - What is an exception
  - In pipelines

Material that is not in this lecture

Readings from the book

The book has excellent descriptions of this topic. Please read the book before watching this lecture. The reading assignment is on the website.

(Don’t forget: the assigned reading may include details or bits and pieces that I don’t cover in the lecture. You’re responsible for that as well on the exam.)

```
Predicting Branches and Exceptions
Introduction to Computer Architecture
David Black-Schaffer

Material that is not in this lecture

Readings from the book

The book has excellent descriptions of this topic. Please read the book before watching this lecture. The reading assignment is on the website.

(Don’t forget: the assigned reading may include details or bits and pieces that I don’t cover in the lecture. You’re responsible for that as well on the exam.)

```

Branch delay slot

```
```

Branch hazard: just stall?

```
```

Moving the branch computation earlier

```
```
Branches with the earlier branch logic

This instruction is always executed. It is called the branch delay slot.

Performance cost of the branch delay slot?
- 1 cycle used for every branch
- Branches are 17% of all instructions → 17% slower
- But, if we can re-order our code to fill the slot... we are okay.

Summary: delay slot is okay for 5 stages
- By moving logic forward we reduced the branch penalty to 1 cycle
- We can often fill a single branch delay slot
- But... modern processors have 15-20 pipeline stages
  - We can't fix this by moving the branch calculation earlier
  - We can't fill more than 1 branch delay slot reliably
  - Stalling hurts performance too much
- Remember the fundamental problem: it takes time to figure out whether a branch is taken and we have to wait
- How can we fix this? Predict the future

Predicting not taken (pipeline with 3 delay slots)
Predict not taken

- We are going to guess that the branch will not be taken and keep executing instructions (e.g., try to predict the future)
- If we guess right (correct prediction) then we have 0 branch delay
- If we guess wrong (incorrect prediction) then we have to kill the wrong instructions and suffer the delay
  - “Kill” or “Squash” so they don’t execute (they were wrong)
  - Prevent them from writing: disable RegWrite, MemWrite in the pipeline
  - Turn them into NOPs: change opcode to add R0, R0, R0 in the pipeline

How well does this work?

- About 33% of branches are taken
- Our prediction is about 67% accurate. Is that good enough?
  - On an Intel processor (Nehalem) a branch misprediction costs 17 cycles
  - 17% of instructions are branches
  - We predict correctly 67% of the time
  - \[(0.17 \text{ branches/instruction}) \times (0.33 \text{ mispredictions/branch}) \times (17 \text{ cycles/misprediction}) + (1 \text{ cycle/instruction}) \] = 1.95x slower
- Predicting not taken will run our processor 1.95x slower
- This is not good enough.

Other approaches

- Static branch predictions: they don’t change
- Predict always not taken
- Predict always taken
  - Depends on the code
- Think about loops:
  - The branch at the end almost always jumps back (the loop repeats many times)
  - How about: Backwards-Taken, Forward-Not-Taken (BTNT)
- Can the compiler help?
  - Sure, if it can identify the likely branches. (E.g., loops or error checks)
How well do predictors do?

- Always Taken/Not Taken: not so good
- BTFN: not great
- Compiler: not good enough
- How can we do a lot better?
- Dynamic predictors: learn what the branch does

Example of learning what the branch does

- Does what the loop did the last time help predict what it will do this time?

```
for (k=0; k<100,000,000; k++)
{
    // Do something
}
```

- If (a == b) {
    // Do something
} else {
    // Do something else
}

Dynamic branch prediction

- Simple idea: remember what the branch did before, and use that to predict what it will do next.

- Examples: [Taken/Not-Taken]
  - History: NTTTTTTTT  NextT? T (likely)
  - History: NTHTTHTHTT  NextT? N (likely)
  - History: NTHTTHTHTT  NextT? T (likely)
  - History: NNNNNTTTTT  NextT? T, but could be N if it is a long pattern
  - History: HTTTTTTTTT  NextT? T, but could be N if it is a long pattern

- So how do we build something that will work this way?

How do we record a branch’s history?

- History: NTTTTTTTT  \( \rightarrow \) Want to predict that the next branch will be Taken
- Need to store:
  - If the branch was last taken or not taken,
  - For every branch in the program
  - And then check it when we load the branch to predict what to do.
- Store a table of bits in the processor per branch: 1=Taken, 0=Not Taken
- Set based on what the branch did last time.

© 2012 David Black-Schaffer
Upda)ng the 1-bit branch predictor

Table before branch resolved: guess not taken

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l1: add R1, R2, R3</td>
<td>sub R3, R4, R2</td>
<td>beq R1, R3, L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l2: lui R2, 0x1234</td>
<td>bne R3, R4, L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table after branch resolved

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l1: add R1, R2, R3</td>
<td>sub R3, R4, R2</td>
<td>beq R1, R3, L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l2: lui R2, 0x1234</td>
<td>bne R3, R4, L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems with a 1-bit predictor

• Does what the loop did the last time help predict what it will do this time?

for \( k=0; k<100,000,000; ++k \) {
  // Do something
}

Q: What percentage mispredictions do we do for the k loop with a 1-bit predictor?

\[ \frac{2}{100,000,000} = 0\% \]
\[ \frac{10}{100,000,000} = 0\% \]
\[ 20\% \]

A: 20\%

We mispredict twice for every iteration of the k loop:
- the first branch in the k-loop was taken so now the loop will exit
- the last branch in the k-loop was taken so now the loop will exit

So that is 2/10 mispredictions for each of the 100M k-loops, or 20\% mispredictio\ns. (plus a first one and a last one)

2-bit branch predictor

• If 1-bit isn’t good enough, how about 2-bits?
  - More “detailed” branch behavior

2-bit branch predictor: correct prediction

• If we predicted correctly we keep our prediction

2-bit branch predictor: first misprediction

• First misprediction: don’t change prediction (hope it is a fluke)

2-bit branch predictor: back to correct

• If it was a fluke, we go back to previous state: just one misprediction
2-bit branch predictor: really wrong

- Two mispredictions in a row, so we change our prediction

![Diagram of 2-bit branch predictor example]

2-bit predictor example

1. We stay here as long as "not taken" is correct
2. Oops… first mispredict
3. Oops… 2nd mispredict; lets change our predictions for future
4. Stay here as long as "taken" is now right...

n-bit predictors: saturating counters

- Predict based on the majority of the past branches

![Diagram of n-bit predictors: saturating counters]

How well does it work?

- 4096 entry (4096 separate branches)
- 2-bit predictor

![Graph of Frequency of Mispredictions]

How do we store branch history bits?

Associative structures
Keeping branch prediction data

- We want a table that keeps track of whether each branch is taken
  - Check it on each branch to see what to do
  - Update it once the branch is resolved with what we really did
- How does this work?
  - Associative memory (or Content Addressable Memory, CAM)
  - Instead of an address, we input the PC and the memory tells us what it knows about that PC

But how do we know where to jump?

- We can now predict if the branch is taken, but how do we know where it goes?
- Add the branch target (address) to the table
- Now we have a Branch Target Buffer (BTB)

Branch target buffer (BTB)

- For each PC, the BTB stores:
  - If the PC is a branch
  - If the branch is predicted taken/not taken
  - The address of the branch if taken
- Update prediction when the branch is resolved
- BTB is not infinitely big, so it can’t hold every branch

(We’ll see a lot more about associative memories in the lecture on caches)

What are exceptions and interrupts?

- Exceptions are non-normal events that interrupt the normal flow of instructions
  - Divide by zero
  - Misaligned memory access
  - Page fault
  - Memory protection violation
- Interrupts are external events that interrupt the normal flow of instructions
  - Keyboard press
  - Disk drive ready
  - Arrival of network packet

Exceptions and interrupts
Handling exceptions and interrupts

• On an exception or interrupt, the processor transfers control to the OS to handle the event.
• When it is done, the OS restarts the program at the offending instruction.
• Some exceptions are fatal and result in the program being terminated.

Types of exceptions and interrupts

• Synchronous vs. asynchronous
  — Synchronous: occur at the same place every time a program executes.
  — Asynchronous: caused by external devices and happen at different times.
• User requested vs. coerced
  — Coerced are hardware events the user can’t control.
  — User requested are from the user.
• User maskable vs. nonmaskable
  — Can the user disable the exception/interrupt?
• Within vs. between instructions
  — Does the event prevent the current instruction from completing, or interrupt after it?
• Resume vs. terminate
  — Can the event be handled (corrected) by the OS or program, or must the program be terminated?

Types of exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Syn/Async</th>
<th>User Request?</th>
<th>User Maskable?</th>
<th>Within?</th>
<th>Resume?</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>between</td>
<td>resume</td>
</tr>
<tr>
<td>Initiating OS</td>
<td>syn</td>
<td>user req.</td>
<td>nonmaskable</td>
<td>between</td>
<td>resume</td>
</tr>
<tr>
<td>Instruction execution</td>
<td>syn</td>
<td>user req.</td>
<td>nonmaskable</td>
<td>between</td>
<td>resume</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>resume</td>
</tr>
<tr>
<td>Overflow (int)</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>resume</td>
</tr>
<tr>
<td>Overflow (fp)</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>resume</td>
</tr>
<tr>
<td>Page fault</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>resume</td>
</tr>
<tr>
<td>Misalignment check</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>resume</td>
</tr>
<tr>
<td>Instruction violation</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>terminate</td>
</tr>
<tr>
<td>Unaligned instr</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>terminate</td>
</tr>
<tr>
<td>Hardware interrupt</td>
<td>syn</td>
<td>coerced</td>
<td>nonmaskable</td>
<td>within</td>
<td>terminate</td>
</tr>
</tbody>
</table>

Exceptions in MIPS

• What exceptions can occur in which stages?
• IF — Page fault on instruction fetch — Memory protection violation.
• ID — Undefined or illegal opcode
• EX — Divide by zero — Overflow
• MEM — none

Example exception

Multiple exceptions

• lw has an exception in MEM in cycle 4 (page fault).
• add has an exception in EX in cycle 4 (overflow).
• How to handle both of them?
  — Handle the 1st one 1st (lw exception).
  — Re-execute lw (no exception since we fixed the problem in the OS).
  — Re-execute add (we’ll now get the overflow exception and can handle it).
### More complicated multiple exceptions

<table>
<thead>
<tr>
<th>Program</th>
<th>Clock Cycle 1</th>
<th>Clock Cycle 2</th>
<th>Clock Cycle 3</th>
<th>Clock Cycle 4</th>
<th>Clock Cycle 5</th>
<th>Clock Cycle 6</th>
<th>Clock Cycle 7</th>
<th>Clock Cycle 8</th>
<th>Clock Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td>lw</td>
<td></td>
<td>ME</td>
<td>ME</td>
<td>lw</td>
<td>ME</td>
<td>ME</td>
<td>lw</td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td>add</td>
<td>ME</td>
<td>ME</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- lw has an exception in MEM in cycle 4 (memory protection)
- add has an exception in IM in cycle 2 (illegal opcode)
- How to handle both of them?
  - Precise exceptions: lw is first instruction, so handle it's exception first
  - Imprecise exceptions: add exception happened first, so handle it first

### Exceptions summary

- **Extremely hard to get right**
  - Particularly with long complicated pipelines
  - Particularly when many instructions are executing at the same time

- **But very important**
  - How do you know what caused a problem if you don’t get the exceptions in the right order? (debugging)

- **Tradeoffs:**
  - Imprecise exceptions may require less hardware (easier)
  - Precise exceptions may be emulated with extra software (slower)