Processor Design – Pipelined Processor (III)

Hung-Wei Tseng
Pipeline hazards

- Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI == 1.

- Pipeline hazards:
  - Structural hazard
    - The hardware does not allow two pipeline stages to work concurrently
  - Data hazard
    - A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  - Control hazard
    - The processor is not clear about what’s the next instruction to fetch
Performance of stall

Insert a “noop” in EXE stage
Insert another “noop” in EXE stage, previous noop goes to MEM stage

add $1, $2, $3
lw $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw $1, 0($5)

15 cycles! CPI == 3
(If there is no stall, CPI should be just 1!)
Hazard detection with forwarding
Sol. of data hazard II: Forwarding

- Take the values, where ever they are!

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```

10 cycles! CPI == 2 (Not optimal, but much better!)
Control hazard

• Consider the following code and the pipeline we designed:

```
LOOP: lw  $t3, 0($s0)
    addi $t0, $t0, 1
    add  $v0, $v0, $t3
    addi $s0, $s0, 4
    bne  $t1, $t0, LOOP
    sw   $v0, 0($s1)
```

How many cycles the processor needs to stall before we figure out the next instruction after “bne”?  

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Control hazard

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards, what’s the average CPI if we execute this program on the 5-stage MIPS pipeline?

A. 1  
B. 1.2  
C. 1.4  
D. 1.6  
E. 1.8
Announcement

• Homework #3 is up
  • Due Sunday (7/17) at noon
  • Will release solution right after the deadline
  • No late submission accepted
• No office hour of Hung-Wei this Friday
  • Will have office hour this Thursday 11a-12p instead
Outline

• Control Hazard
• Reducing the overhead of control hazards
• Dynamic branch prediction
Reducing the overhead of control hazards
Solution I: Delayed branches

• An agreement between ISA and hardware
  • “Branch delay” slots: the next N instructions after a branch are always executed
  • Compiler decides the instructions in branch delay slots
    • Reordering the instruction cannot affect the correctness of the program
  • MIPS has one branch delay slot

• Good
  • Simple hardware

• Bad
  • N cannot change
  • Sometimes cannot find good candidates for the slot
Solution 1: Delayed branches

```
LOOP: lw   $t3, 0($s0)
    addi $t0, $t0, 1
    add $v0, $v0, $t3
    addi $s0, $s0, 4
    bne $t1, $t0, LOOP
```

6 cycles per loop
Solution II: always predict not-taken

- Always predict the next PC is PC+4

```plaintext
LOOP:  lw  $t3, 0($s0)
       addi $t0, $t0, 1
       add  $v0, $v0, $t3
       addi $s0, $s0, 4
       bne  $t1, $t0, LOOP
       sw   $v0, 0($s1)
       add  $t4, $t3, $t5
       lw   $t3, 0($s0)
```

If branch is not taken: no stalls!
If branch is taken: doesn’t hurt!

7 cycles per loop
Solution III: always predict taken

- Using the pipeline we have, how many cycles we have to stall if we predict branch taken?

```
LOOP: lw   $t3, 0($s0)
    addi $t0, $t0, 1
    add  $v0, $v0, $t3
    addi $s0, $s0, 4
    bne  $t1, $t0, LOOP
    lw   $v0, 0($s0)
```

A. 0  
B. 1  
C. 2  
D. 3  
E. 4  

We still don’t know the target address until EX stage completes
Solution III: always predict taken

- When is the earliest stage that we can calculate the target address?
  A. IF
  B. ID
  C. EX
  D. MEM
  E. WB
Solution III: always predict taken
Solution III: always predict taken

Still have to stall 1 cycle
Solution III: always predict taken

Branch Target Buffer

Consult BTB in fetch stage
Solution III: always predict taken

- Always predict taken with the help of BTB

```
LOOP: lw  $t3, 0($s0)  
    addi $t0, $t0, 1
    add  $v0, $v0, $t3
    addi $s0, $s0, 4
    bne $t1, $t0, LOOP
    lw  $t3, 0($s0)
    addi $t0, $t0, 1
    add  $v0, $v0, $t3
```

5 cycles per loop
(CPI == 1 !!!)

But what if the branch is not always taken?
Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch X
```

What is the prediction accuracy of branch Y if we use static always predict taken predictor? (Choose the closest one)

A. 0%
B. 33%
C. 67%
D. 100%
Static branch predictions

• How many of the following about static branch prediction method is correct?
  • Comparing with stalls, static branch prediction mechanisms are never doing worse in our current MIPS 5-stage pipeline
  • A static branch prediction mechanism never changes the prediction result during program execution
  • “Flush” occurs only after the processor detects an incorrect branch prediction
  • “Always predict taken” cannot fetch a taken instruction during the ID stage of the branch instruction without the help of BTB

A. 0
B. 1
C. 2
D. 3
E. 4
Dynamic branch prediction
1-bit counter

- Predict this branch will go the same way as the result of the last time this branch executed
- 1 for taken, 0 for not taken

PC = 0x400420

Branch Target Buffer

<table>
<thead>
<tr>
<th>PC</th>
<th>Target</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>1</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>1</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>0</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>1</td>
</tr>
</tbody>
</table>
Accuracy of 1-bit counter

- Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0 ) // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100 ) // Branch X
```

What is the prediction accuracy of branch Y using 1-bit predictors (if all counters start with 0/not taken). Choose the most close one.

Assume unlimited BTB entries.

A. 0%

B. 33%

C. 67%

D. 100%

---

<table>
<thead>
<tr>
<th>i</th>
<th>branch</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>
2-bit counter

- A 2-bit counter for each branch
- Predict taken if the counter value >= 2
- If the prediction in taken states, fetch from target PC, otherwise, use PC+4

<table>
<thead>
<tr>
<th>Branch Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC = 0x400420</td>
</tr>
<tr>
<td>0x400420</td>
</tr>
<tr>
<td>0x400464</td>
</tr>
<tr>
<td>0x400578</td>
</tr>
<tr>
<td>0x41000C</td>
</tr>
</tbody>
</table>

Taken!
Performance of 2-bit counter

- 2-bit state machine for each branch

```c
for(i = 0; i < 10; i++) {
    sum += a[i];
}
```

90% accuracy!

<table>
<thead>
<tr>
<th>i</th>
<th>state</th>
<th>predict</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4-9</td>
<td>11</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?

- $1+20%*(1-90%)*2 = 1.04$
Accuracy of 2-bit counter

- Consider the following code:

```java
i = 0;
do {
    if( i % 3 != 0)  // Branch Y, taken if i % 3 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)  // Branch X
```

What is the prediction accuracy of branch Y using 2-bit predictors (if all counters start with 00). Choose the closest one. Assume unlimited BTB entries.

A. 0%
B. 33%
C. 67%
D. 100%
Make the prediction better

• Consider the following code:

```c
i = 0;
do {
    if( i % 3 != 0 ) // Branch Y,
        if i % 3 == 0
            a[i] *= 2;
        a[i] += i;
} while ( ++i < 100 ) // Branch X
```

Can we capture the pattern?
Predict using history

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Each entry in the history table has its own counter.

\[ n \text{-bit GHR} = 101 \ (T, \ NT, \ T) \]

2\(^n\) entries

index

Taken!
Consider the following code:

```c
i = 0;
while (++i < 100) {
    if( i % 3 != 0) // Branch Y, taken
        a[i] *= 2;
    else if i % 3 == 0 // Branch X
        a[i] += i;
}
```

Assume that we start with a 4-bit GHR= 0, all counters are 10.

<table>
<thead>
<tr>
<th>i</th>
<th>?</th>
<th>GHR</th>
<th>BHT</th>
<th>prediction</th>
<th>actual</th>
<th>New BHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Y</td>
<td>0000</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0001</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>0011</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0110</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>1101</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>1010</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>0101</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>1011</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>0111</td>
<td>10</td>
<td>T</td>
<td>NT</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>1110</td>
<td>10</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>1101</td>
<td>11</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>1010</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>0101</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>1011</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>0111</td>
<td>11</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>7</td>
<td>X</td>
<td>1110</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Y</td>
<td>1101</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
<tr>
<td>8</td>
<td>X</td>
<td>1010</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>Y</td>
<td>0101</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>X</td>
<td>1011</td>
<td>11</td>
<td>T</td>
<td>T</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>Y</td>
<td>0111</td>
<td>00</td>
<td>NT</td>
<td>NT</td>
<td>00</td>
</tr>
</tbody>
</table>

Nearly perfect after this.