Processor Design (III)

Hung-Wei Tseng
Control hazard
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 does the processor need to stall before we figure out the next instruction after “bne”?

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Why do we need to stall for branch instructions

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor

   ① The target address when branch is taken is not available for instruction fetch stage of the next cycle
   ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
   ③ The branch outcome cannot be decided until the comparison result of ALU is not out
   ④ The next instruction needs the branch instruction to write back its result

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

- The processor cannot determine the next PC to fetch

```assembly
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)
```

7 cycles per loop
Branch prediction to reduce the overhead of control hazards
Why do we need to stall for branch instructions

• How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor

1. The target address when branch is taken is not available for instruction fetch stage of the next cycle
2. The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
3. The branch outcome cannot be decided until the comparison result of ALU is not out
4. The next instruction needs the branch instruction to write back its result

A. 0
B. 1
C. 2
D. 3
E. 4
Branch Target Buffer

• The processor needs a “cheat sheet” for where the branch is going without calculating it
Dynamic branch prediction

- 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
  - If we guess right — no penalty
  - If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC

PC = 0x400420

<table>
<thead>
<tr>
<th>Branch Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420 0x8048324 11</td>
</tr>
<tr>
<td>0x400464 0x8048392 10</td>
</tr>
<tr>
<td>0x400578 0x804850a 00</td>
</tr>
<tr>
<td>0x41000C 0x8049624 01</td>
</tr>
</tbody>
</table>

Taken!
Dynamic branch prediction

- 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>PC</th>
<th>Address</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400420</td>
<td>0x8048324</td>
<td>11</td>
</tr>
<tr>
<td>0x400464</td>
<td>0x8048392</td>
<td>10</td>
</tr>
<tr>
<td>0x400578</td>
<td>0x804850a</td>
<td>00</td>
</tr>
<tr>
<td>0x41000C</td>
<td>0x8049624</td>
<td>01</td>
</tr>
</tbody>
</table>
```

Branch Target Buffer
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?
  - $1 + 20\% \times (1-90\%) \times 2 = 1.04$
Branch prediction & your code
Demo revisited

- Why the sorting the array speed up the code despite the increased instruction count?

```cpp
if (option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```
Branch performance

\[
\begin{align*}
\text{if(option)} \\
\quad &\text{std::sort(data, data + arraySize);} \\
\text{for } (\text{unsigned } i = 0; i < 100000; ++i) \{ \\
\quad &\text{int threshold = std::rand();} \\
\quad &\text{for } (\text{unsigned } i = 0; i < \text{arraySize}; ++i) \{ \\
\quad &\quad \text{if (data[i] } \geq \text{ threshold)} \\
\quad &\quad \quad \text{sum }++; \\
\quad &\} \\
\} 
\end{align*}
\]

- Why the performance is better when option is not “0”
  1. The amount of dynamic instructions needs to execute is a lot smaller
  2. The amount of branch instructions to execute is smaller
  3. The amount of branch mis-predictions is smaller
  4. The amount of data accesses is smaller

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

<table>
<thead>
<tr>
<th></th>
<th>Without sorting</th>
<th>With sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>The prediction</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>accuracy of X before</td>
<td></td>
<td></td>
</tr>
<tr>
<td>threshold</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The prediction</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>accuracy of X after</td>
<td></td>
<td></td>
</tr>
<tr>
<td>threshold</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Demo: popcount

- How many 1s in binary representations
- Applications
  - Hamming weight
  - Encryption/decryption

```c
int main(int argc, char *argv[]) {
    uint64_t key = 0xdeadbeef;
    int count = 1000000000;
    uint64_t sum = 0;
    for (int i=0; i < count; i++)
        { sum += popcount (RandLFSR(key)); }
    printf("Result: %lu\n", sum);
    return sum;
}
```
Four implementations

• Which of the following implementations will perform the best on modern pipeline processors?

A

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while (x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while (x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

C

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while (x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

D

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
Why is B better than A

• How many of the following statements explains the reason why B outperforms A with compiler optimizations
  ① B has lower dynamic instruction count than A
  ② B has significantly lower branch mis-predictions than A
  ③ B has significantly fewer branch instructions than A
  ④ B can incur fewer data hazards

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

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while (x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while (x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```
Why is B better than A

A

```c
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```c
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

and $t2$, $t1$, 1
add $t3$, $t3$, $t2$
shr $t1$, $t1$, 1
bne $t1$, $zero$, LOOP

4*n instructions

and $t2$, $t1$, 1
add $t3$, $t3$, $t2$
shr $t1$, $t1$, 1
and $t2$, $t1$, 1
add $t3$, $t3$, $t2$
shr $t1$, $t1$, 1
and $t2$, $t1$, 1
add $t3$, $t3$, $t2$
shr $t1$, $t1$, 1
and $t2$, $t1$, 1
add $t3$, $t3$, $t2$
shr $t1$, $t1$, 1
and $t2$, $t1$, 1
add $t3$, $t3$, $t2$
shr $t1$, $t1$, 1
bne $t1$, $zero$, LOOP

13*(n/4) = 3.25*n instructions

Only one branch for four iterations in A