Processor Design (IV)

Hung-Wei Tseng
Recap: 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
Recap: 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 $\geq 2$
- 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
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>

90% accuracy!

- Application: 80% ALU, 20% Branch, and branch resolved in EX stage, average CPI?
  - \(1+20\%\times(1-90\%\times2 = 1.04\)
Branch prediction & your code
• 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

```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 ++;
    }
}
```

- 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 accuracy of X before threshold</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>The prediction accuracy of X after threshold</td>
<td>50%</td>
<td>100%</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;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        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;
}
```

A

B

C

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

```c
// B
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;
}
```
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
bne  $t1, $zero, LOOP
```

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

Only one branch for four iterations in A
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;
    }
    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
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
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
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
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

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

Only one branch for four iterations in A
Two versions of B

Before re-ordering

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

Lots of back-to-back data dependencies — likely to introduce data hazards

After re-ordering

and $t2, $t1, 1
shr $t4, $t1, 1
shr $t5, $t1, 2
shr $t6, $t1, 3
shr $t1, $t1, 4
and $t7, $t4, 1
and $t8, $t5, 1
and $t9, $t6, 1
add $t3, $t3, $t2
add $t3, $t3, $t7
add $t3, $t3, $t8
add $t3, $t3, $t9
bne $t1, $zero, LOOP

This re-ordering is only possible after you “unrolled” your loop — this technique is called “loop unrolling”
Why is B better than A

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

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Why is C better than B

- How many of the following statements explains the reason why B outperforms C with compiler optimizations

  1. C has lower dynamic instruction count than B
  2. C has significantly lower branch mis-prediction rates than B
  3. C has significantly fewer branch instructions than B
  4. C can incur fewer data hazards

A. 0
B. 1
C. 2
D. 3
E. 4
Why is D better than C

• How many of the following statements explains the main reason why D outperforms C with compiler optimizations
  ① D has lower dynamic instruction count than C
  ② D has significantly lower branch mis-prediction rates than C
  ③ D has significantly fewer branch instructions than C
  ④ D can incur fewer data hazards than C
A. 0
B. 1
C. 2
D. 3
E. 4
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 3, 1, 2, 3, 2, 3, 3, 4};
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    c += table[(x & 0xF)];
    x = x >> 4;
    return c;
}
Hardware acceleration

• Because popcount is important, both intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a
• In C/C++, you may use the intrinsic “_mm_popcnt_u64” to get # of “1”s in an unsigned 64-bit number
  • You need to compile the program with -m64 -msse4.2 flags to enable these new features

#include <smmintrin.h>
inline int popcount(uint64_t x) {
    int c = _mm_popcnt_u64(x);
    return c;
}
Local 2-bit predictor

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

- What’s the overall branch prediction (include both branches) accuracy for this nested for loop? (assume all states started with 00)
  A. ~25%
  B. ~33%
  C. ~50%
  D. ~67%
  E. ~75%
Local 2-bit predictor

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

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

For branch Y, almost 100%,  
For branch X, only 50%
Local 2-bit predictor

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

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>state</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>Y</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>01</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>10</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>01</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>11</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>00</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>23</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

For branch Y, almost 100%, For branch X, only 50%

Can we capture the pattern?
Branch prediction using global history
2-level global predictor

- Instead of using the PC to choose the predictor, use a bit vector (global history register, GHR) made up of the previous branch outcomes.
- Global predictor: predictor using results from all branches
- Local predictor: predictor tracking states/history for each branch
- Each entry in the history table has its own counter.

First level

3-bit GHR = 101 (T, NT, T)

2nd level

2^3 entries

Pentium Pro uses this predictor
Performance of the 2-bit global predictor

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

Nearly perfect after this
The pipeline of modern processors
The pipeline of modern processors

- Way deeper than 5 stages
- Shortening the pipeline stages helps improve the “cycle time”
- Higher marketing values since consumers usually link performance with frequencies
- Potentially higher power consumption as dynamic/active power = aCV^2f
- If the execution time is better, still consume less energy
Intel Pentium 4 Microarch.
Intel Pentium 4

- Very deep pipeline: in order to achieve high frequency! (start from 1.5GHz)
  - 20 stages in Netburst
  - 31 stages in Prescott
- 103W (3.6GHz, 65nm)
- Reference
  - The Microarchitecture of the Pentium 4 Processor
AMD Athlon 64

AMD 64 Architecture

ITLB
Level 1 Instruction Cache
4K Bytes

L2-TLB
Level 2 Cache
up to 1 MB exclusive
L2 ECC
L2 Tags
L2 Tag ECC

128 Bit + 76 Bit Predecode, Branch, Parity

Patch 2-transit

Pack

Decode

Instruction Control Unit (128-entry)

8-entry Scheduler
8-entry Scheduler
8-entry Scheduler

ALU AGU
IMUL

FMUL
SSE
SSE

Load/Store Queue (L24-entry)

Data TLB
4K entry

Level 1 Data Cache, 64 KBytes, 2-way ECC

System Request Queue (SQ)

Cross Bar (XBAR)

Memory Controller

Hyper Transport

128-bit dual channel or 64-bit single channel up to DDR2-800 SDRAM
AMD Athlon 64

- 12 stage pipeline

|---|-------------------|----------------|------------------|------|------|---------------------|---------------|------------|--------------|--------------|--------------------|------------------|

- 89W TDP (Opteron 2.2GHz 90nm)
Pentium 4 v.s. Athlon 64

- Application: 80% ALU, 20% Branch, 90% prediction accuracy, no data dependencies, perfect cache, consider the two machines:
  - Pentium 4 with 20 pipeline stages, branch resolved in stage 19, running at 3 GHz
  - Athlon 64 with 12 pipeline stages, branch resolved in stage 10, running at 2.7 GHz (11% longer cycle time)

which one is faster?

A. Athlon 64
B. Pentium 4
Pentium 4 v.s. Athlon 64

- Application: 80% ALU, 20% Branch, 90% prediction accuracy, no data dependencies, perfect cache, consider the two machines:
  - Pentium 4 with 20 pipeline stages, branch resolved in stage 19, running at 3 GHz
  - Athlon 64 with 12 pipeline stages, branch resolved in stage 10, running at 2.7 GHz (11% longer cycle time)

\[
\text{CPI}_{\text{P4}} = 0.8 \times 1 + 0.2 \times 0.9 \times 1 + 0.2 \times 0.1 \times 19 = 1.36 \\
\text{CPI}_{\text{Athlon64}} = 0.8 \times 1 + 0.2 \times 0.9 \times 1 + 0.2 \times 0.1 \times 10 = 1.18
\]

At least 15% faster clock rate to achieve the same performance

which one is faster?
A. Athlon 64
B. Pentium 4
Case Study