Pipeline Processor (4)

Hung-Wei Tseng
Recap: Three pipeline hazards

• Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
• Control hazards — the PC can be changed by an instruction in the pipeline
• Data hazards — an instruction depending on a the result that’s not yet generated or propagated when the instruction needs that
Recap: Do we still have to stall?

• How many pairs of instructions in the following MIPS instructions will result in data hazards/stalls in a basic 5-stage MIPS pipeline with “full” data forwarding?

lw $t0,0($a0)             IF ID EX MEM WB
add $t0,$t0, $t2           IF ID ID EX MEM WB
sw $t0,0($a0)              IF IF ID EX MEM WB
addi $a0,$a0, 4            IF ID EX MEM WB
bne $a0,$t1, LOOP          IF ID EX MEM WB

A. 0
B. 1
C. 2
D. 3
E. 4
Recap: The impact of control hazards

• Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. 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

\[ 1 + 20\% \times 2 = 1.4 \]
Recap: Static Not-Taken Predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```plaintext
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
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

\[
1 + 75\% \times (20\% \times 2) = 1.3
\]

For branch Y, almost 0%,
For branch X, only 50%

<table>
<thead>
<tr>
<th>i</th>
<th>branch?</th>
<th>prediction</th>
<th>actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>6</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>Y</td>
<td>NT</td>
<td>T</td>
</tr>
</tbody>
</table>
Outline

• Branch predictions (cont.)
• Performance Programming on modern pipeline processors
A basic dynamic branch predictor

Branch Target Buffer

branch PC | target PC | State
---|---|---
0x400048 | 0x400032 | 10
0x400080 | 0x400068 | 11
0x400100 | 0x401100 | 00
0x4000F8 | 0x400100 | 01
Recap: 2-bit/Bimodal local predictor

- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- 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

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
<td>10</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
<td>11</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
<td>00</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
<td>01</td>
</tr>
</tbody>
</table>

Predict Taken
Recap: 2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```
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
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

For branch Y, almost 100%, For branch X, only 50%
What’s the overall branch prediction (include both branches) accuracy for this nested for loop?

```
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
```

(assume all states started with 00)

A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75%

Can we do a better job?

For branch Y, almost 100%, For branch X, only 50%
Two-level global predictor

2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```
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
```

(assume all states started with 00)

- A. ~25%
- B. ~33%
- C. ~50%
- D. ~67%
- E. ~75%

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

This pattern repeats all the time!
Global history (GH) predictor

Branch Target Buffer

<table>
<thead>
<tr>
<th>branch PC</th>
<th>target PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400048</td>
<td>0x400032</td>
</tr>
<tr>
<td>0x400080</td>
<td>0x400068</td>
</tr>
<tr>
<td>0x401080</td>
<td>0x401100</td>
</tr>
<tr>
<td>0x4000F8</td>
<td>0x400100</td>
</tr>
</tbody>
</table>

States associated with history

\begin{align*}
00 & 01 \\
10 & 11 \\
10 & 11 \\
10 & 11 \\
00 & 00 \\
00 & 00 \\
00 & 00 \\
01 & 01 \\
00 & 00 \\
\end{align*}

Predict Taken

\[(NT, T, NT, NT)\]
Performance of GH 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

Near perfect after this
Better predictor?

Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?

A. 0
B. 1
C. 2
D. 3
E. 4
**Better predictor?**

- Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Better predictor?

- Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippets would allow (G) to outperform (L)?

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

---

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

Better predictor?

i = 0;
do {
    if( i % 10 != 0)
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);

i = 0;
do {
    a[i] += i;
} while ( ++i < 100);

i = 0;
do {
    j = 0;
    do {
        sum += A[i*2+j];
    } while( ++j < 2);
} while ( ++i < 100);

i = 0;
do {
    if( rand() %2 == 0)
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)
Putting it altogether — Performance programming on modern pipeline processors
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 ++;
    }
}
```
Demo revisited

• 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  if(option)     
    std::sort(data, data + arraySize);
B. 1

C. 2     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  if(option)
    std::sort(data, data + arraySize);
B. 1  
    for (unsigned i = 0; i < 100000; ++i) {
C. 2  
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
D. 3  
    if (data[i] >= threshold)
      sum ++;
E. 4  

}
**Demo revisited**

- 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
  ✔️ The amount of branch mis-predictions is smaller
  4. The amount of data accesses is smaller

<table>
<thead>
<tr>
<th></th>
<th>Without sorting</th>
<th>With sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. 0</td>
<td>if (option) std::sort(data, data + arraySize);</td>
<td></td>
</tr>
<tr>
<td>B. 1</td>
<td>for (unsigned i = 0; i &lt; 100000; ++i) { int threshold = std::rand(); for (unsigned i = 0; i &lt; arraySize; ++i) if (data[i] &gt;= threshold) sum ++; }</td>
<td></td>
</tr>
<tr>
<td>C. 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D. 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The population count (or popcount) of a specific value is the number of set bits (i.e., bits in 1s) in that value.

Applications
- Parity bits in error correction/detection code
- Cryptography
- Sparse matrix
- Molecular Fingerprinting
- Implementation of some succinct data structures like bit vectors and wavelet trees.
Demo: pop count

• Given a 64-bit integer number, find the number of 1s in its binary representation.

• Example 1:
  Input: 9487
  Output: 7
  Explanation: 9487’s binary representation is 0b10010100001111

```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;
}
```
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;
}
```
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;
}
```
• 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-prediction rate 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
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-prediction rate 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;
}
```
Recap: The effect of code optimization

• By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. None of the pairs can be reordered
If we can unroll the loop...

- Consider the following dynamic instructions:
  (1) lw $t0,0($a0)
  (2) add $t0,$t0, $t2
  (3) sw $t0,0($a0)
  (4) lw $t3,4($a0)
  (5) add $t3,$t3, $t2
  (6) sw $t3,4($a0)
  (7) addi $a0,$a0, 8
  (8) bne $a0,$t1, LOOP

Which of the following pair can we reorder without affecting the correctness if the loop is unrolled twice?

A. (2) and (3)
B. (3) and (4)
C. (5) and (6)
D. (6) and (7)
E. (7) and (8)
If we can unroll the loop...

Consider the following dynamic instructions:

1. \( \text{lw} \) $t0,0($a0)
2. \( \text{add} \) $t0,$t0, $t2
3. \( \text{sw} \) $t0,0($a0)
4. \( \text{lw} \) $t3,4($a0)
5. \( \text{add} \) $t3,$t3, $t2
6. \( \text{sw} \) $t3,4($a0)
7. \( \text{addi} \) $a0,$a0, 8
8. \( \text{bne} \) $a0,$t1, LOOP

Which of the following pair can we reorder without affecting the correctness if the loop is unrolled twice?

A. (2) and (3)
B. (3) and (4)
C. (5) and (6)
D. (6) and (7)
E. (7) and (8)
If we can unroll the loop...

- Consider the following dynamic instructions:

  1. \( \text{lw} \quad \text{t0}, 0(\text{a0}) \)
  2. \( \text{add} \quad \text{t0}, \text{t0}, \text{t2} \)
  3. \( \text{sw} \quad \text{t0}, 0(\text{a0}) \)
  4. \( \text{lw} \quad \text{t3}, 4(\text{a0}) \)
  5. \( \text{add} \quad \text{t3}, \text{t3}, \text{t2} \)
  6. \( \text{sw} \quad \text{t3}, 4(\text{a0}) \)
  7. \( \text{addi} \quad \text{a0}, \text{a0}, 8 \)
  8. \( \text{bne} \quad \text{a0}, \text{t1}, \text{LOOP} \)

Which of the following pair can we reorder without affecting the correctness if the loop is unrolled twice?

- A. (2) and (3)
- B. (3) and (4)
- C. (5) and (6)
- D. (6) and (7)
- E. (7) and (8)
If we can unroll the loop...

(1) lw $t0,0($a0)     (1) lw $t0,0($a0)     (1) lw $t0,0($a0)
(2) add $t0,$t0, $t2  (2) lw $t3,4($a0)  (2) lw $t3,4($a0)
(3) sw $t0,0($a0)     (3) add $t0,$t0, $t2  (3) add $t0,$t0, $t2
(4) lw $t3,4($a0)     (4) add $t3,$t3, $t2  (4) add $t3,$t3, $t2
(5) add $t3,$t3, $t2  (5) sw $t0,0($a0)  (5) sw $t0,0($a0)
(6) sw $t3,4($a0)     (6) sw $t3,4($a0)  (6) sw $t3,4($a0)
(7) addi $a0,$a0, 8   (7) addi $a0,$a0, 8 (7) addi $a0,$a0, 8
(8) bne $a0,$t1, LOOP (8) bne $a0,$t1, LOOP (8) bne $a0,$t1, LOOP
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;
}
```

4*n instructions:

```
and x2, x1, 1
add x3, x3, x2
shr x1, x1, 1
bne x1, x0, LOOP
```

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

Only one branch for four iterations in A
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-prediction rate than A
  3. B has significantly fewer branch instructions than A
  4. B can incur fewer data hazards

**Why is B better than A?**

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;
}
```
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 rate 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 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 rate 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 C better than B?

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

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

Why is C better than B?

- C only needs one load, one add, one shift, the same amount of iterations.
- The same number being predicted.
- The same amount of branches.

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

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations:
  1. D has lower dynamic instruction count than C
  2. D has significantly lower branch mis-prediction rate 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

```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;
}
```

```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 D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  1. D has lower dynamic instruction count than C
  2. D has significantly lower branch mis-prediction rate 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  

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

```c
// D
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```
Loop unrolling eliminates all branches!

```c
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1, 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 D better than C?

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

- D has lower dynamic instruction count than C
- D has significantly lower branch mis-prediction rate than C
- D has significantly fewer branch instructions than C
- D can incur fewer data hazards than C

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>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 &amp; 0xF)]; x = x &gt;&gt; 4; } return c; }</td>
<td>- Compiler can do loop unrolling — no branches</td>
</tr>
<tr>
<td>B</td>
<td>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 &lt; 16; i++) { c += table[(x &amp; 0xF)]; x = x &gt;&gt; 4; } return c; }</td>
<td>- Could be</td>
</tr>
<tr>
<td>C</td>
<td>- compiler cannot unroll because x is unknown!</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>- compiler can unroll because i is known!</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>- maybe eliminated through loop unrolling...</td>
<td></td>
</tr>
</tbody>
</table>

D — Compiler can do loop unrolling — no branches
— Could be
— maybe eliminated through loop unrolling...
— maybe eliminated through loop unrolling...
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 the number 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.

```c
#include <smmintrin.h>
inline int popcount(uint64_t x) {
    int c = _mm_popcnt_u64(x);
    return c;
}
```
Announcements

• Assignment #3 — due tomorrow evening
• Midterm
  • Thursday during the lecture — make sure that you’re available, we will use Zoom
  • The lecture on Wednesday will provide a review, highlight, sample midterm
• All lab deadlines are “soft”, except for Lab 5/6
Computer Science & Engineering

Now playing — You Need to Calm Down (Taylor Swift)