LECTURE 4
Bit Hacks and Vectorization
I–Ting Angelina Lee
September 18, 2012
Problem
Swap two integers x and y.

\[
\begin{align*}
t &= x; \\
x &= y; \\
y &= t;
\end{align*}
\]
No-Temp Swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

$$x = x \ ^\wedge \ y;$$
$$y = x \ ^\wedge \ y;$$
$$x = x \ ^\wedge \ y;$$

Example

<table>
<thead>
<tr>
<th>$x$</th>
<th>00101110</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
</tbody>
</table>

XOR

$0^\wedge 0 = 0$
$0^\wedge 1 = 1$
No-Temp Swap

**Problem**
Swap two integers $x$ and $y$ without using a temporary.

```plaintext
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

**Example**

<table>
<thead>
<tr>
<th></th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
<tr>
<td>$y$</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

**XOR**

\[
\begin{align*}
0 \oplus 0 &= 0 \\
0 \oplus 1 &= 1
\end{align*}
\]
No-Temp Swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

Example

\[
\begin{array}{c|c|c|c|c}
  x & 10111101 & 10010011 & 10010011 & 00101110 \\
  y & 00101110 & 00101110 & 10111101 & 10111101 \\
\end{array}
\]

\[
\begin{align*}
  x &= x \oplus y; \\
  y &= x \oplus y; \\
  x &= x \oplus y;
\end{align*}
\]

XOR

\[
\begin{align*}
  0 \oplus 0 &= 0 \\
  0 \oplus 1 &= 1
\end{align*}
\]
No–Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
  x &= x \land y; \\
  y &= x \land y; \\
  x &= x \land y;
\end{align*}
\]

Example

\[
\begin{array}{cccc}
 x & 10111101 & 10010011 & 10010011 & 00101110 \\
 y & 00101110 & 00101110 & 10111101 & 10111101 \\
\end{array}
\]

XOR

\[
\begin{align*}
  0 \land 0 &= 0 \\
  0 \land 1 &= 1
\end{align*}
\]
No–Temp Swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

$x = x ^ y$;
$y = x ^ y$;
$x = x ^ y$;

Example

<table>
<thead>
<tr>
<th></th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

XOR
$0^0 = 0$
$0^1 = 1$
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
x &= x \ ^\wedge \ y; \\
y &= x \ ^\wedge \ y; \\
x &= x \ ^\wedge \ y; \\
\end{align*}
\]

Example

<table>
<thead>
<tr>
<th></th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
<tr>
<td>( y )</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

Why it works
XOR is its own inverse: \( (x \ ^\wedge \ y) \ ^\wedge \ y = x \)
No-Temp Swap

Problem
Swap two integers x and y without using a temporary.

\[
\begin{align*}
x &= x \oplus y; \\
y &= x \oplus y; \\
x &= x \oplus y;
\end{align*}
\]

Example

<table>
<thead>
<tr>
<th>x</th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

Why it works
XOR is its own inverse: \((x \oplus y) \oplus y = x\)

XOR
\[
\begin{align*}
b \oplus 0 &= b \\
b \oplus 1 &= \overline{b}
\end{align*}
\]
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
x &= x \oplus y; \\
y &= x \oplus y; \\
x &= x \oplus y;
\end{align*}
\]

mask with 1s at bits that differ

Example

<table>
<thead>
<tr>
<th></th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
<tr>
<td>y</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

Why it works
XOR is its own inverse: \((x \oplus y) \oplus y = x\)

**XOR**
\[
\begin{align*}
b \oplus 0 &= b \\
b \oplus 1 &= \neg b
\end{align*}
\]
No-Temp Swap

Problem
Swap two integers x and y without using a temporary.

Example

<table>
<thead>
<tr>
<th>x</th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

Why it works
XOR is its own inverse: \((x \^ y) \^ y = x\)

XOR
\[ b^0 = b \]
\[ b^1 = \neg b \]
No-Temp Swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

\[
x = x \oplus y; \\
y = x \oplus y; \\
x = x \oplus y;
\]

Example
\[
\begin{array}{cccc}
x & 10111101 & 10010011 & 10010011 & 00101110 \\
y & 00101110 & 00101110 & 10111101 & 10111101 \\
\end{array}
\]

Why it works
XOR is its own inverse: $(x \oplus y) \oplus y = x$

\[
\begin{align*}
\text{XOR} & \quad b \oplus 0 = b \\
& \quad b \oplus 1 = \neg b
\end{align*}
\]
No-Temp Swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
x &= x \oplus y; \\
y &= x \oplus y; \\
x &= x \oplus y;
\end{align*}
\]

Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
<tr>
<td>( y )</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

Why it works
XOR is its own inverse: \((x \oplus y) \oplus y = x\)

Performance
Poor at exploiting instruction-level parallelism (ILP).
Minimum of Two Integers

Problem
Find the minimum \( r \) of two integers \( x \) and \( y \).

\[
\text{if } (x < y) \\
\quad r = x; \text{ or } \quad r = (x < y) \ ? \ x : y; \\
\text{else} \\
\quad r = y;
\]

Performance
A mispredicted branch empties the processor pipeline.

Caveat
The compiler may be smart enough to optimize away the unpredictable branch, but maybe not.
No-Branch Minimum

Problem
Find the minimum $z$ of two integers $x$ and $y$ without a branch.

$$r = y \ ^\ ( (x \ ^\ y) \ & \ -(x < y) ) ;$$

Why it works:
• The C language represents the Booleans `TRUE` and `FALSE` with the integers 1 and 0, respectively.
• If $x < y$, then $-(x < y) = -1$, which is all 1’s in two’s complement representation. Therefore, we have $y \ ^\ (x \ ^\ y) = x$.
• If $x \geq y$, then $-(x < y) = 0$. Therefore, we have $y \ ^\ 0 = y$. 
Merging Two Sorted Arrays

```c
static void merge(long *__restrict C,
                  long *__restrict A,
                  long *__restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na > 0) {
    *C++ = *A++; na--;
  }
  while (nb > 0) {
    *C++ = *B++; nb--;
  }
}
```
static void merge(long *__restrict C, 
long *__restrict A, 
long *__restrict B, 
size_t na, 
size_t nb) {

    while (na > 0 && nb > 0) {
        if (*A <= *B) {
            *C++ = *A++; na--;
        } else {
            *C++ = *B++; nb--;
        }
    }

    while (na > 0) {
        *C++ = *A++; 
        na--;
    }

    while (nb > 0) {
        *C++ = *B++; 
        nb--;
    }
}
static void merge(long * __restrict C,
    long * __restrict A,
    long * __restrict B,
    size_t na,
    size_t nb) {
    while (na > 0 && nb > 0) {
        long cmp = (*A <= *B);
        long min = *B ^ ((*B ^ *A) & (-cmp));
        *C++ = min;
        A += cmp; na -= cmp;
        B += !cmp; nb -= !cmp;
    }
    while (na > 0) {
        *C++ = *A++;
        na--;
    }
    while (nb > 0) { 
        *C++ = *B++;
        nb--;
    }
}

On the cloud machines using gcc -O3, the branchless version is 1.5 times slower than the branching version due to compiler’s use of cmov.
Problem
Compute \((x + y) \mod n\), assuming that \(0 \leq x < n\) and \(0 \leq y < n\).

\[
r = (x + y) \% n;
\]

Divide is expensive, unless by a power of 2.

\[
z = x + y;
r = (z < n) ? z : z-n;
\]

Unpredictable branch is expensive.

\[
z = x + y;
r = z - (n & -(z >= n));
\]

Same trick as minimum.
Round up to a Power of 2

**Problem**
Compute $2^{\lceil \log n \rceil}$.

```c
//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

**Example**

```
0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000
```

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Problem
Compute $2^{\lceil \log n \rceil}$.

//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000
Round up to a Power of 2

Problem
Compute $2^\lceil \log n \rceil$.

```c
//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

```
0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000
```
Round up to a Power of 2

Problem
Compute $2^{\lceil \log n \rceil}$.

//64-bit integers
--n;
n |== n >> 1;
n |== n >> 2;
n |== n >> 4;
n |== n >> 8;
n |== n >> 16;
n |== n >> 32;
++n;

Example

<table>
<thead>
<tr>
<th>0010000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001001111</td>
</tr>
<tr>
<td>0011000001101111</td>
</tr>
<tr>
<td>0011110001111111</td>
</tr>
<tr>
<td>0011111000111111</td>
</tr>
<tr>
<td>0011111111111111</td>
</tr>
<tr>
<td>0100000000000000</td>
</tr>
</tbody>
</table>
**Problem**

Compute \(2^{\lceil \log n \rceil}\).

```c
//64-bit integers
--n;

n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

**Example**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010000001010000</td>
<td>0010000001001111</td>
<td>0011000001101111</td>
<td>0011100001110111</td>
<td>0011110001111111</td>
</tr>
<tr>
<td>0011111000111111</td>
<td>0011111111111111</td>
<td>0100000000000000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Round up to a Power of 2

Problem
Compute $2^\lceil \log n \rceil$.

//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

| 0010000001010000 |
| 0010000001001111 |
| 0011000001101111 |
| 0011100001101111 |
| 0011110001111111 |
| 0011111111111111 |
| 0100000000000000 |
Problem
Compute \(2^{\lceil \log n \rceil}\).

//64-bit integers
---

```c
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00100000001010000</td>
<td>00100000001001111</td>
<td>00110000001101111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00110000001101111</td>
<td>00111110001111111</td>
<td>00111111111111111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111111111111111</td>
<td>01000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Round up to a Power of 2

Problem
Compute $2^{\lceil \log n \rceil}$.

//64-bit integers
---n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

<table>
<thead>
<tr>
<th></th>
<th>0010000001010000</th>
<th>0010000001001111</th>
<th>0011000001101111</th>
<th>0011110001111111</th>
<th>0011111111111111</th>
<th>0100000000000000</th>
</tr>
</thead>
</table>

Round up to a Power of 2

Problem
Compute \(2^{\lceil \log n \rceil}\).

// 64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;

Example

<table>
<thead>
<tr>
<th></th>
<th>0010000001010000</th>
<th>0010000001001111</th>
<th>0011000001101111</th>
<th>0011110001111111</th>
<th>0011111111111111</th>
<th>0100000000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001010000</td>
<td>0010000001001111</td>
<td>0011000001101111</td>
<td>0011110001111111</td>
<td>0011111111111111</td>
<td>0100000000000000</td>
<td></td>
</tr>
</tbody>
</table>
Round up to a Power of 2

**Problem**
Compute $2^{\lceil \log n \rceil}$.

**Example**
```
//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

- $n = 0010000001010000$
- $n = 0010000001001111$
- $n = 0011000001101111$
- $n = 0011110001111111$
- $n = 0011111111111111$
- $n = 0100000000000000$

**(log n) – 1\textsuperscript{th} bit must be set**

- Set $[\log n]^{\text{th}}$ bit
- Populate all bits to the right with 1
Round up to a Power of 2

**Problem**
Compute $2^\lceil \log n \rceil$.

```cpp
//64-bit integers
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

**Example**

```
0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000
```

**Why decrement?**
To handle the boundary case when $n$ is a power of 2.
Least-Significant 1

Problem
Compute the mask of the least-significant 1 in word x.

\[ r = x \& (-x); \]

Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0010000001010000</td>
</tr>
<tr>
<td>-x</td>
<td>11011111110110000</td>
</tr>
<tr>
<td>x &amp; (-x)</td>
<td>00000000000010000</td>
</tr>
</tbody>
</table>

Why it works
The binary representation of \(-x\) is \(~x+1\).

Question
How do you find the index of the bit, i.e., \(\log_2 r\)?
Log Base 2 of a Power of 2

Problem
Compute $\log_2 x$, where $x$ is a power of 2.

```c
const uint64_t deBruijn = 0x022fdd63cc95386d;
const unsigned int convert[64] = {
    0, 1, 2, 53, 3, 7, 54, 27,
    4, 38, 41, 8, 34, 55, 48, 28,
    62, 5, 39, 46, 44, 42, 22, 9,
    24, 35, 59, 56, 49, 18, 29, 11,
    63, 52, 6, 26, 37, 40, 33, 47,
    61, 45, 43, 21, 23, 58, 17, 10,
    51, 25, 36, 32, 60, 20, 57, 16,
    50, 31, 19, 15, 30, 14, 13, 12
};

r = convert[(x*deBruijn) >> 58];
```
Introducing
The Engineer Who Invented ESD*
★ The Technology to Read Minds ★
Why it works
A deBruijn sequence \( s \) of length \( 2^k \) is a cyclic 0–1 sequence such that each of the \( 2^k \) 0–1 strings of length \( k \) occurs exactly once as a substring of \( s \).

\[
\begin{align*}
00011101_2 \times 2^4 &= 11010000_2 \\
11010000_2 \gg 5 &= 6 \\
\text{convert}[6] &= 4
\end{align*}
\]

Performance
Limited by multiply and table look–up

Example \( k=3 \)

<table>
<thead>
<tr>
<th>0</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>010</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
</tr>
</tbody>
</table>

\[
\text{convert}[8] = \{0, 1, 6, 2, 7, 5, 4, 3\};
\]
Problem
Place $n$ queens on an $n \times n$ chessboard so that no queen attacks another, i.e., no two queens in any row, column, or diagonal.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
**Strategy**

Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
**Strategy**
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
**Strategy**
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
**Strategy**
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
The backtrack search can be implemented as a simple recursive procedure, but how should the board be represented to facilitate queen placement?

- array of $n^2$ bytes?
- array of $n^2$ bits?
- array of $n$ bytes?
- 3 bitvectors of size $n$, $2n-1$, and $2n-1$!
Bitvector Representation

Placing a queen in column \( c \) is not safe if \( \text{down} \& (1<<c) \) is nonzero.

Down
Placing a queen in row \( r \) and column \( c \) is not safe if
\[
\text{left} \& \ (1 \ll (r+c)) \text{ is nonzero.}
\]
Placing a queen in row $r$ and column $c$ is not safe if $right \& (1<<(n-r+c))$ is nonzero.
Population Count I

Problem
Count the number of 1 bits in a word $x$.

```c
for (r=0; x!=0; ++r)
    x &= x - 1;
```

Repeatedly eliminate the least-significant 1.

Example

<table>
<thead>
<tr>
<th>$x$</th>
<th>0010110111010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x-1$</td>
<td>0010110111001111</td>
</tr>
<tr>
<td>$x &amp; (x-1)$</td>
<td>0010110111000000</td>
</tr>
</tbody>
</table>

Issue
Fast if the popcount is small, but in the worst case, the running time is proportional to the number of bits in the word.
Table look-up

```c
static const int count[256] =
{0,1,1,2,1,2,2,3,1,...,8}; // #1's in index

for (r=0; x!=0; x>>=8)
  r += count[x & 0xFF];
```

Performance
Performance depends on the size of $x$; cost of the Memory operations are the main bottleneck.

- register: 1 cycle,
- L1-cache: 4 cycles,
- L2-cache: 10 cycles,
- L3-cache: 50 cycles,
- DRAM: 150 cycles.

per 64-byte cache line
Parallel divide-and-conquer

// Create masks
M5 = !((-1) << 32); // 0^{32}1^{32}
M4 = M5 ^ (M5 << 16); // (0^{16}1^{16})^2
M3 = M4 ^ (M4 << 8); // (0^{8}1^{8})^4
M2 = M3 ^ (M3 << 4); // (0^{4}1^{4})^8
M1 = M2 ^ (M2 << 2); // (0^{2}1^{2})^{16}
M0 = M1 ^ (M1 << 1); // (01)^{32}

// Compute popcount
x = (((x >> 1) & M0) + (x & M0));
x = (((x >> 2) & M1) + (x & M1));
x = (((x >> 4) + x) & M2);
x = (((x >> 8) + x) & M3);
x = (((x >> 16) + x) & M4);
x = (((x >> 32) + x) & M5);

Notation: exp^x = repeat expression exp x times
Population Count III

1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0
Population Count III

1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0

x&M0
(x>>1)&M0
Population Count III

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
+ & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]
## Population Count III

<table>
<thead>
<tr>
<th>1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1</th>
<th>0 1 0 0 0 1 1 1 1 1 0 0 0</th>
<th>( x &amp; M ) ( (x \gg 1) &amp; M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0</td>
<td>1 0 0 1 0 0 1 1 1 1 0 0 0 0 1 1 0</td>
<td>( x &amp; M ) ( (x \gg 2) &amp; M )</td>
</tr>
</tbody>
</table>
| + 1 0 0 1 0 0 1 1 1 1 0 0 0 0 1 1 0 | \hline
| 1 0 0 0 0 0 1 0 1 0 1 0 1 1 0 1 0 1 | 1 0 0 0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 |
### Population Count III

|   | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| + | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
|   | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| + | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|   | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
Population Count III

|   | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| + | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
|   | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| + | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|   | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
Population Count III

| 11000010010110111111010001111000 | \( x \& M0 \)  
| 100011011111011100 | \( (x >> 1) \& M0 \)  
| 10010011111100001110 | \( x \& M1 \)  
| 00010100001011000101 | \( (x >> 2) \& M1 \)  
| 00100010001101000110 | \( x \& M2 \)  
| 00000110000001100001010100000100 | \( (x >> 4) \& M2 \)  

© 2012 Charles E. Leiserson and I-Ting Angelina Lee
### Population Count III

| 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0 |
|-------------------------|-------------------------|-------------------------|
| +                       | +                       | +                       |
| 1 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 1 1 0 0 0 1 1 1 0 0 0         |
| +                       | +                       | +                       |
| 1 0 0 1 0 0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 0           |
| +                       | +                       | +                       |
| 0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1       |
| +                       | +                       | +                       |
| 0 0 1 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0   |
| +                       | +                       | +                       |
| 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 1 0 1 0 0 0 0 1 0 1 0 0 0 |

- $x \& M0$ ($x \gg 1) \& M0$
- $x \& M1$ ($x \gg 2) \& M1$
- $x \& M2$ ($x \gg 4) \& M2$
- $x \& M4$ ($x \gg 8) \& M4$
## Population Count III

|    | 11000001001011011111010001111000 | x&M0  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000011011101110110110001110</td>
<td>(x&gt;&gt;1)&amp;M0</td>
</tr>
</tbody>
</table>
| +  | 000101110001000100 | x&M1  
|    | 0001001100101001 | (x>>2)&M1 |
| +  | 0010001010000110 | x&M2  
|    | 00000001010001000 | (x>>4)&M2 |
| +  | 0000000100 | x&M4  
|    | 0000000101 | (x>>8)&M4 |
|    | 000000001001 |       |

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
### Population Count III

<table>
<thead>
<tr>
<th></th>
<th>0 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0</th>
<th>x&amp;M0 ((x &gt;&gt; 1) &amp; M0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1 0 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 0 0</td>
<td>x&amp;M1 ((x &gt;&gt; 2) &amp; M1)</td>
</tr>
<tr>
<td>+</td>
<td>1 0 0 1 0 0 1 1 1 1 1 1 0 0 0 1 0 0 1 0 1 0 1 0 1</td>
<td>x&amp;M2 ((x &gt;&gt; 4) &amp; M2)</td>
</tr>
<tr>
<td>+</td>
<td>0 0 0 1 0 0 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 1 0</td>
<td>x&amp;M3 ((x &gt;&gt; 8) &amp; M3)</td>
</tr>
<tr>
<td>+</td>
<td>0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1</td>
<td>x&amp;M4 ((x &gt;&gt; 16) &amp; M4)</td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>+</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
### Population Count III

<table>
<thead>
<tr>
<th>110000010010110111111010001111000</th>
<th>(x&amp;M_0) ((x\gg 1)&amp;M_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100011100111111011100001110</td>
<td>(x&amp;M_1) ((x\gg 2)&amp;M_1)</td>
</tr>
<tr>
<td>+00010110100010</td>
<td>(x&amp;M_2) ((x\gg 4)&amp;M_2)</td>
</tr>
<tr>
<td>+00100010010001</td>
<td>(x&amp;M_3) ((x\gg 8)&amp;M_3)</td>
</tr>
<tr>
<td>+11000010</td>
<td>(x&amp;M_4) ((x\gg 16)&amp;M_4)</td>
</tr>
<tr>
<td>00000000000000000100100000</td>
<td><strong>17</strong></td>
</tr>
</tbody>
</table>

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Population Count III

Parallel divide-and-conquer

// Create masks
M5 = !((-1) << 32); // 0^{32}1^{32}
M4 = M5 ^ (M5 << 16); // (0^{16}1^{16})^2
M3 = M4 ^ (M4 << 8); // (0^{8}1^{8})^4
M2 = M3 ^ (M3 << 4); // (0^{4}1^{4})^8
M1 = M2 ^ (M2 << 2); // (0^{2}1^{2})^{16}
M0 = M1 ^ (M1 << 1); // (0^{1})^{32}

// Compute popcount
x = (((x >> 1) & M0) + (x & M0));
x = (((x >> 2) & M1) + (x & M1));
x = (((x >> 4) + x) & M2);
{x = (((x >> 8) + x) & M3;  
  x = (((x >> 16) + x) & M4; 
  x = (((x >> 32) + x) & M5; 

Performance
Θ(lg n) time, where n = word length.

Need to perform & first to prevent overflow.

Enough bits to go around so no need to worry about overflow.
Vectorization

Some x86 processors incorporate vector units to process data in the **single-instruction multiple-data (SIMD)** fashion.

A little history ...

- 1996 – the MMX vector instruction set (Pentium)
  - co-opted the floating-point registers to be used as the 64-bit MMX registers (%mm0-%mm7)
- 1999 – the SSE vector instruction set (Pentium III)
  - 8 128-bit XMM registers (%xmm0-%xmm7) for single-precision floating point data
  - extends the MMX instructions to use XMM registers
- 2001 – the SSE2 vector instruction set (Pentium 4)
  - support for double-precision floating point data
  - double the XMM registers (%xmm0-%xmm15)
- 2011 – the AVX vector instruction set (Sandy Bridge)
  - 15 256-bit YMM registers (%ymm0-%ymm15)
Vectorization

```c
int sum_exceeds(uint64_t *A,
                uint64_t n,
                uint64_t threshold) {
    uint64_t sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum > threshold;
}
```

The loop is vectorized.

```c
int sum_exceeds_sc(uint64_t *A,
                    uint64_t n,
                    uint64_t threshold) {
    uint64_t sum = 0;
    for (size_t i = 0; i < n; i++) {
        sum += A[i];
        if (sum > threshold) {
            // Handling the sum exceeding the threshold
        }
    }
    return 0;
}
```

The loop is not vectorized due to jumps.

**Performance**

Depends on how quickly the sum exceeds threshold.

**Compiler flags for vectorization report:**

- **gcc:** `-ftree-vectorizer-verbose=[n]`
- **icc:** `–vec-report[n]`
Modern computers implement some of the functions discussed in this lecture directly in hardware. The Intel Software Developer’s Manual describes *compiler intrinsics* that will let you access this functionality efficiently.
Further Reading


http://chessprogramming.wikispaces.com/


Happy Bit Hacking!