Lecture 3
Bit Hacks

Charles E. Leiserson
September 12, 2013
Ordinary Swap

Problem
Swap two integers $x$ and $y$.

$t = x;$
$x = y;$
$y = t;$
No-Temp Swap

Problem
Swap x and y without using a temporary.

Example

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

\[
x = x \land y; \\
y = x \land y; \\
x = x \land y;
\]
No-Temp Swap

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

Example

<table>
<thead>
<tr>
<th>$x$</th>
<th>10111101</th>
<th>10010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>00101110</td>
<td>00101110</td>
</tr>
</tbody>
</table>
No-Temp Swap

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

$$x = x \oplus y;$$
$$y = x \oplus y;$$
$$x = x \oplus y;$$

Example

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
<th>$y$</th>
<th>$x \oplus y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10111101</td>
<td>00101110</td>
<td>10010011</td>
</tr>
<tr>
<td>1</td>
<td>10010011</td>
<td>01011101</td>
<td>10111101</td>
</tr>
</tbody>
</table>
No-Temp Swap

**Problem**
Swap \( x \) 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>

\( x \) and \( y \) are swapped using the XOR operation.
**No-Temp Swap**

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

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

**Example**

<table>
<thead>
<tr>
<th>( \text{x} )</th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{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: \((x \oplus y) \oplus y \Rightarrow x\)
No-Temp Swap (Instant Replay)

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

$x = x \oplus y$;
$y = x \oplus y$;
$x = x \oplus y$;

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \oplus y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

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

Why it works

XOR is its own inverse: $(x \oplus y) \oplus y \Rightarrow x$
No-Temp Swap (Instant Replay)

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

Mask with 1’s where bits differ.

Example

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

Why it works
XOR is its own inverse: \((x \ ^ \ y) \ ^ \ y \Rightarrow x\)
No-Temp Swap (Instant Replay)

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

Example

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

Why it works
XOR is its own inverse: \((x ^ y) ^ y \Rightarrow x\)
No-Temp Swap (Instant Replay)

Problem
Swap $x$ 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: $(x \ ^\ ^\ y) \ ^\ ^\ y \Rightarrow x$
No-Temp Swap (Instant Replay)

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

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

Example

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

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

Performance 🤕
Poor at exploiting \textit{instruction-level parallelism (ILP)}. 
Minimum of Two Integers

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

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

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 \( r \) of two integers \( x \) and \( y \) without using a branch.

\[
r = y \land ((x \land y) \land -(x < y));
\]

Why it works:
• The C language represents the Booleans \texttt{TRUE} and \texttt{FALSE} with the integers 1 and 0, respectively.
• If \( x < y \), then \(- (x < y) \Rightarrow -1\), which is all 1’s in two’s complement representation. Therefore, we have \( y \land (x \land y) \Rightarrow x \).
• If \( x \geq y \), then \(- (x < y) \Rightarrow 0\). Therefore, we have \( y \land 0 \Rightarrow y \).
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--;
    }
}
```

© 2013 Charles E. Leiserson and I–Ting Angelina Lee
Branching

```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--;
    }
}
```

<table>
<thead>
<tr>
<th>Branch</th>
<th>Predictable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
</tr>
</tbody>
</table>
This optimization works well on some machines, but on the CSAIL cloud machines using `gcc -O3`, the branchless version is actually slower than the branching version. 👎 Modern compilers can often perform this optimization better than you can!
Why Learn Bit Hacks?

Why learn bit hacks if they don’t even work?

• Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly language.
• Because sometimes the compiler doesn’t optimize, and you have to do it by hand.
• Because the bit hacks for words extend naturally to bit hacks for vectors.
Modular Addition

Problem
Compute \((x + y) \mod n\), assuming that \(0 \leq x < n\) and \(0 \leq y < n\).

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

Division is expensive, unless by a power of 2.

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

Unpredictable branch is expensive.

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

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

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

Notation
$\lg n = \log_2 n$
Round up to a Power of 2

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

```
uint64_t n;
:
--n;
```
```
\[
n \mid= n >> 1;
n \mid= n >> 2;
n \mid= n >> 4;
n \mid= n >> 8;
n \mid= n >> 16;
n \mid= n >> 32;
++n;
```

Example
```
0010000001010000
```

© 2013 Charles E. Leiserson and I-Ting Angelina Lee
Round up to a Power of 2

**Problem**
Compute \(2^{\lceil \log_2 n \rceil}\).

```c
uint64_t n;
:
- - n;
 n |= n >> 1;
 n |= n >> 2;
 n |= n >> 4;
 n |= n >> 8;
 n |= n >> 16;
 n |= n >> 32;
++n;
```

**Example**

```
0010000001010000
```
Round up to a Power of 2

**Problem**

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

```c
uint64_t n;
:
--n;

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

**Example**

```
0010000001010000
0010000001001111
```

© 2013 Charles E. Leiserson and I-Ting Angelina Lee
Problem
Compute $2^{\lceil \log_2 n \rceil}$.

Example

<table>
<thead>
<tr>
<th></th>
<th>0010000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0010000001001111</td>
</tr>
<tr>
<td>2</td>
<td>0011000001101111</td>
</tr>
</tbody>
</table>

uint64_t n;
:\n-­‐n;
n |−|= n >> 1;
n |−|= n >> 2;
n |−|= n >> 4;
n |−|= n >> 8;
n |−|= n >> 16;
n |−|= n >> 32;
++n;
Round up to a Power of 2

**Problem**

Compute $2^\lceil \lg n \rceil$.

```c
uint64_t n;
:
  --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
```
Round up to a Power of 2

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

```c
uint64_t n;
:
  --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>00100000001010000</th>
<th>0010000001001111</th>
<th>0011000001101111</th>
<th>0011110001111111</th>
<th>0011111111111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>10</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>n</td>
<td>01</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>n</td>
<td>01</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>n</td>
<td>01</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>n</td>
<td>01</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>n</td>
<td>01</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
</tbody>
</table>
```
Problem
Compute $2^{\lceil \lg n \rceil}$.

```c
uint64_t n;
:
    --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>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001010000</td>
<td>0010000001001111</td>
<td>0011000001101111</td>
<td>0011110001111111</td>
<td>0011111111111111</td>
<td></td>
</tr>
</tbody>
</table>
Round up to a Power of 2

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

```c
uint64_t n;
:
--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
```
Round up to a Power of 2

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

```c
uint64_t n;
:
--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>0010000001010000</td>
<td>0010000001001111</td>
<td>0011000001101111</td>
<td>0011110001111111</td>
<td>0011111111111111</td>
</tr>
</tbody>
</table>
```
Round up to a Power of 2

Problem
Compute $2^{\lceil \log_2 n \rceil}$.

```c
uint64_t n;
:
--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 $2^{\lceil \lg n \rceil}$.

```c
uint64_t n;
:
--n;

n |= n >> 1;

n |= n >> 2;

n |= n >> 4;

n |= n >> 8;

n |= n >> 16;

n |= n >> 32;

++n;
```

**Example**

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

- Set bit $\lceil \lg n \rceil$
- Bit $\lceil \lg n \rceil - 1$ must be set
- Populate all bits to the right with $1$
Round up to a Power of 2

Problem
Compute $2^{\lceil \log_2 n \rceil}$.

```c
uint64_t n;
:
  --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
0011111000111111
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>1101111110110000</td>
</tr>
<tr>
<td>$x &amp; (-x)$</td>
<td>0000000000010000</td>
</tr>
</tbody>
</table>

Why it works
The binary representation of $-x$ is $\sim x - 1$.

Question
How do you find the index of the bit, i.e., $\lg r$?
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];
```
Mathemagic Trick

Introducing
The Engineer Who Invented ESD*
☞ The Technology to Read Minds ☜

*Extra-Sensory Deception
Log Base 2 of a Power of 2

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 $s$.

Example: $k=3$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$s_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<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>

00011101\_2 \times 2^4 \Rightarrow 11010000\_2
11010000\_2 \gg 5 \Rightarrow 6

convert[6] \Rightarrow 4

Performance
Limited by multiply and table look-up.

start with all 0’s

```
const unsigned int convert[8] =
{0,1,6,2,7,5,4,3};
```
Queens Problem

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.
Strategic
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.

Backtrack!
Backtracking Search

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.

Backtrack!
**Strategy**
Try placing queens row by row. If you can’t place a queen in a row, backtrack.

*Backtrack!*
Strategy
Try placing queens row by row. If you can’t place a queen in a row, backtrack.
Board Representation

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 $2n-1$!
Placing a queen in column \( c \) is not safe if \( \text{down} \land (1 \ll c) \) is nonzero.
Placing a queen in row $r$ and column $c$ is not safe if

$$\text{left} \& (1 \ll (r+c))$$

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 \( x \).

\[
\text{for } (r=0; \ x!=0; \ ++r)
\quad x \ &= \ x \ - \ 1;
\]

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.
Population Count II

Table look-up

```c
static const int count[256] =
{ 0, 1, 1, 2, 1, 2, 2, 3, 1, ..., 8 };

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

Performance

Performance depends on the size of \( x \). The cost of memory operations is a major bottleneck.

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

\[ \text{per 64-byte cache line} \]
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);   // (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: \( X^k = \underbrace{XX\cdots X}_k \)
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

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
+ & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & (x >> 1) & \& 0
\end{array}
\]
Population Count III

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
& \times\& M_0 \\
& (x\gg 1)\& M_0 \\
& \times\& M_1 \\
& (x\gg 2)\& M_1 \\
\end{array}
\]
### Population Count III

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

\(x \& M_0\)  
\((x \gg 1) \& M_0\)  
\(x \& M_1\)  
\((x \gg 2) \& M_1\)
## Population Count III

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

$x \& M_0\ (x \gg 1) \& M_0$

$x \& M_1\ (x \gg 2) \& M_1$

$x \& M_2\ (x \gg 4) \& M_2$
Population Count III

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

© 2013 Charles E. Leiserson and I-Ting Angelina Lee
Population Count III

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

\[
x&M0 \quad (x>>1)&M0
\]
\[
x&M1 \quad (x>>2)&M1
\]
\[
x&M2 \quad (x>>4)&M2
\]
\[
x&M4 \quad (x>>8)&M4
\]

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

<table>
<thead>
<tr>
<th>11000010010110111111010001111</th>
<th>x&amp;M0 ((x &gt;&gt; 1) &amp; M0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000110111110111001100</td>
<td>(x &gt;&gt; 2) &amp; M1</td>
</tr>
<tr>
<td>+</td>
<td>x&amp;M1</td>
</tr>
<tr>
<td>0001010100011010001010101</td>
<td>(x &gt;&gt; 4) &amp; M2</td>
</tr>
<tr>
<td>+</td>
<td>x&amp;M2</td>
</tr>
<tr>
<td>000110001100001010001000101</td>
<td>(x &gt;&gt; 8) &amp; M4</td>
</tr>
<tr>
<td>+</td>
<td>x&amp;M4</td>
</tr>
<tr>
<td>00000010100000010001001001</td>
<td></td>
</tr>
</tbody>
</table>

© 2013 Charles E. Leiserson and I–Ting Angelina Lee
<table>
<thead>
<tr>
<th></th>
<th>1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1</th>
<th>1 0 0 0 0 0 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1 0 0</td>
<td>x&amp;M0</td>
</tr>
<tr>
<td></td>
<td>1 0 0 1 0 0 1 1 1 1 1 0 0 0 1 1 0 0</td>
<td>(x&gt;&gt;1)&amp;M0</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 0 0</td>
<td>x&amp;M1</td>
</tr>
<tr>
<td>+</td>
<td>1 0 0 0 0 1 0 1 1 0 0 1 0 1 0 1 0 1</td>
<td>(x&gt;&gt;2)&amp;M1</td>
</tr>
<tr>
<td></td>
<td>0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 1 0 1</td>
<td>x&amp;M2</td>
</tr>
<tr>
<td></td>
<td>0 0 1 0 0 0 1 0 1 0 0 1 0 1 0 1 0 1</td>
<td>(x&gt;&gt;4)&amp;M2</td>
</tr>
<tr>
<td>+</td>
<td>0 0 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0</td>
<td>x&amp;M3</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0</td>
<td>(x&gt;&gt;8)&amp;M3</td>
</tr>
<tr>
<td>+</td>
<td>0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 1 0 1</td>
<td>x&amp;M4</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0</td>
<td>(x&gt;&gt;16)&amp;M4</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Population Count III

© 2013 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 | 1 | 1 | 0 | 0 | 0 |
| + |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| + |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| + |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| + |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| + |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| + | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| + | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| + | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| + | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| + | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| + | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| + | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| + | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

The final result is 17.
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 w) time, where w = word length.

Avoid overflow

No worry about overflow.
Popcount Instructions

Most modern machines provide popcount instructions, which operate much faster than anything you can code yourself. You can access them via compiler intrinsics, e.g., in GCC:

```c
int __builtin_popcount (unsigned int x);
```

**Warning:** You may need to enable certain compiler switches to access built-in functions, and your code will be less portable.

**Exercise**
Compute the log base 2 of a power of 2 quickly using a popcount instruction.
Further Reading


http://chessprogramming.wikispaces.com/


Happy Bit–Hacking!
VECTORIZATION
Vector Hardware

The floating-point hardware of a modern microprocessor incorporates vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion.

A vector unit with vector length $k$ consists of

- $k$ vector lanes, each containing scalar floating-point hardware, and
- a set of vector registers, each comprising $k$ words distributed across the lanes.
Vector Unit

Memory Hierarchy

Vector Load/Store Unit

Lane 0

Lane 1

Lane 2

Lane 3

Instruction decode and sequencing

Vector Registers

© 2013 Charles E. Leiserson and I-Ting Angelina Lee
Vector Units

*Vector instructions* generally operate in an *elementwise* fashion:

- The $i$th element of one vector register can only take part in operations with the $i$th element of other vector registers.
- All lanes perform exactly the same operation on their respective slices of the vector.
- Cross-lane operations are generally not allowed, although some architectures support access to unaligned contiguous memory at some cost in performance.
- Others support *scatter/gather* instructions which can access memory in random fashion, also generally at reduced performance.
Vectorizable Loops

```c
void scale(double *A, size_t n, double x) {
    for (size_t i = 0; i < n; ++i) {
        A[i] *= x;
    }
}
```

- The loop’s bounds are known at runtime.
- All iterations of the loop do the same thing: no branches or function calls.
- The iterations of the loop are *independent*: any given iteration \( i \) does not need any results from a previous loop iteration \( j < i \) in order to execute.
- If you specify the compiler switch `-ftime-tree-vectorizer-vectorizer-verbose=1` to GCC, it will produce a vectorization report telling which loops it vectorized.
Cilk Plus Vector Notation

void scale(double *A, size_t n, double x) {
    for (size_t i = 0; i < n; ++i) {
        A[i] *= x;
    }
}

void scale(double *A, size_t n, double x) {
    A[0:n] *= x;
}

• The notation $A[\theta:n]$ denotes an array section of length $n$ starting at index $\theta$ of $A$.
• Operations performed on an array section are performed elementwise on every element of the array section.
Unvectorizable Loops

```c
double reduce(double *A, size_t n) {
    double sum = 0;
    for (size_t i=0; i<n; i++)
        sum += A[i];
    return sum;
}
```

- The iterations are not independent: each iteration \(i > 0\) depends on the previous iteration \(i - 1\) to update the variable \(\text{sum}\) before iteration \(i\) can.
- The compiler will not reorder the additions, because floating-point addition is not associative.
- The switch `--ffast-math` allows the GCC compiler to reorder the additions and vectorize the code using a technique called *strip mining*.
double reduce(double *A, size_t n) {
    double sum = 0;
    for (size_t i=0; i<n; i++)
        sum += A[i];
    return sum;
}

Original code

IDEA: Add down using vector ops, and then sum subtotals across:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Strip mined for 4 lanes

double reduce(double *A, size_t n) {
    // Assume that n is a multiple of 4
    double temp[4];
    for (size_t j=0; j<4; j++) {
        temp[j] = 0;
    }
    for (size_t i = 0; i < n; i+=4) {
        for (size_t j = 0; j < 4; ++j) {
            temp[j] += A[i+j];
        }
    }
    double sum = 0;
    // Sum the temporaries
    for (size_t j=0; j<4; ++j) {
        sum += temp[j];
    }
    return sum;
}