LECTURE 3
Bit Hacks

Charles E. Leiserson
September 11, 2014
**Problem**
Extract a bit field from a word \( x \).

**Idea**
Mask and shift.

\[
(x \&\text{ mask}) \gg \text{shift};
\]

**Example**
shift = 7

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>101110101101101</td>
</tr>
<tr>
<td>mask</td>
<td>000001111000000</td>
</tr>
<tr>
<td>( x &amp;\text{ mask} )</td>
<td>000001010000000</td>
</tr>
<tr>
<td>( x &amp;\text{ mask} \gg \text{shift} )</td>
<td>0000000000001010</td>
</tr>
</tbody>
</table>
Set a Bit Field

Problem
Set a bit field in a word \( x \) to a value \( y \).

Idea
Invert mask to clear, and OR the shifted value.

\[
x = (x \& \sim\text{mask}) \mid (y \ll \text{shift});
\]

Example
\( \text{shift} = 7 \)

<table>
<thead>
<tr>
<th></th>
<th>101110101101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>( y )</td>
<td>000000000000011</td>
</tr>
<tr>
<td>mask</td>
<td>000001111000000</td>
</tr>
<tr>
<td>( x &amp; \sim\text{mask} )</td>
<td>1011100001101101</td>
</tr>
<tr>
<td>( x = (x &amp; \sim\text{mask}) \mid (y \ll \text{shift}); )</td>
<td>1011100111101101</td>
</tr>
</tbody>
</table>
Set a Bit Field

Problem
Set a bit field in a word \( x \) to a value \( y \).

Idea
Invert mask to clear, and OR the shifted value.

\[
x = (x \& \neg \text{mask}) \lor (y << \text{shift});
\]

Example
shift = 7

<table>
<thead>
<tr>
<th>( x )</th>
<th>101110101101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>0000000000000011</td>
</tr>
<tr>
<td>mask</td>
<td>0000011110000000</td>
</tr>
<tr>
<td>( x &amp; \neg \text{mask} )</td>
<td>1011100001101101</td>
</tr>
<tr>
<td>( x = (x &amp; \neg \text{mask}) \lor (y &lt;&lt; \text{shift}); )</td>
<td>101110011101101</td>
</tr>
</tbody>
</table>

For safety’s sake: 
\( ((y << \text{shift}) \& \text{mask}) \)
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.

$x = x \land y$;
$y = x \land y$;
$x = x \land y$;

Example

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$y$</td>
<td>$x \land y$</td>
</tr>
<tr>
<td>10111101</td>
<td>00101110</td>
<td>00101110</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>
No-Temp Swap

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

Example
\[
\begin{array}{ccc}
\hline
x & y & x \nand y \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\hline
\end{array}
\]

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

\[
x = x \text{^} y; \\
y = x \text{^} y; \\
x = x \text{^} y;
\]
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
\[
\begin{array}{c|c|c|c}
\text{x} & 10111101 & 10010011 & 10010011 \\
\hline
\text{y} & 00101110 & 00101110 & 10111101 \\
\end{array}
\]
No-Temp Swap

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

\[
x = x \ xor \ y; \\
y = x \ xor \ y; \\
x = x \ xor \ y;
\]

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>
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>

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.

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

Example

<table>
<thead>
<tr>
<th>( x )</th>
<th>10111101</th>
</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.

Example

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

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.

Example
\[
\begin{array}{c|c|c|c}
\hline
x & y & x \oplus y \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\hline
\end{array}
\]

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.

Example

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

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

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>

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

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

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

\[
\begin{align*}
\text{if } (x < y) & \quad r = x; \\
\text{else} & \quad r = y;
\end{align*}
\]

or
\[
\begin{align*}
r & = (x < y) \, ? \, x \, : \, y;
\end{align*}
\]

**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.

\[
r = y ^ {((x ^ y) \& -(x < y))};
\]

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 \( y ^ {(x ^ y)} \Rightarrow x \).
- If \( x \geq y \), then \(- (x < y) \Rightarrow 0\). Therefore, we have \( y ^ 0 \Rightarrow y \).
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--;
    }
}
```

```c
3 12 19 46
```
```c
4 14 21 23
```
```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>
**Branchless**

```c
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--;
    }
}
```

This optimization works well on some machines, but on modern machines using gcc -O3, the branchless version is usually slower than the branching version. 😞 Modern compilers can 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 yourself by hand.
• Because many bit hacks for words extend naturally to bit and word hacks for vectors.
• Because these tricks arise in other domains, and so it pays to be educated about them.
• Because they’re fun!
Modular Addition

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

- Division is expensive, unless by a power of 2.
- Unpredictable branch is expensive.
- Same trick as minimum.

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

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

\[
z = x + y;
\]
\[
r = z - (n & -(z >= n));
\]
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 $2^{\lceil \lg n \rceil}$.

```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**

```
00100000001010000
```
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>0010000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001001111</td>
</tr>
</tbody>
</table>
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
0011000001101111
```
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>0010000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001001111</td>
</tr>
<tr>
<td>0011000001101111</td>
</tr>
<tr>
<td>0011110001111111</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>00100000001010000</th>
<th>0010000001001111</th>
<th>0011000001101111</th>
<th>0011110001111111</th>
<th>0011111111111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>$n'$</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>$n''$</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>$n'''$</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>$n''''$</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
</tbody>
</table>

© 2013-2014 Charles E. Leiserson and I-Ting Angelina Lee
Round up to a Power of 2

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

```
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 $2^{\lceil \lg n \rceil}$.

Example

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

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>000</td>
<td>000</td>
<td>001</td>
<td>000</td>
<td>101</td>
<td>000</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>000</td>
<td>000</td>
<td>001</td>
<td>000</td>
<td>100</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>110</td>
<td>000</td>
<td>001</td>
<td>110</td>
<td>111</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>111</td>
<td>100</td>
<td>001</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>1</td>
</tr>
</tbody>
</table>
```
## Round up to a Power of 2

### Problem

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

```c
uint64_t n;
⋮
-n;
```

```c
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>0011000001101111</td>
<td>0011110001111111</td>
<td>0011111111111111</td>
<td></td>
</tr>
</tbody>
</table>
```
Round up to a Power of 2

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

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

**Example**

```
00100000001010000
00100000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000
```
Problem
Compute $2^{\lceil \lg n \rceil}$.

uint64_t n;
:
--n;
if (n)
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    n |= n >> 32;
++n;

Bit $\lceil \lg n \rceil - 1$ must be set
Set bit $\lceil \lg n \rceil$
Populate all bits to the right with 1

Example

| 00100000001010000 |
| 00100000001001111 |
| 0011000001101111 |
| 0011110001111111 |
| 0011111111111111 |
| 0100000000000000 |

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

```
00100000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000
```

Why decrement?
To handle the boundary case when $n$ is a power of 2.
**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>1101111111011000</td>
</tr>
<tr>
<td>( x &amp; (-x) )</td>
<td>0000000000001000</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., \( \lg r \)?
Log Base 2 of a Power of 2

Problem
Compute \( \lg x \), where \( x \) is a power of \( 2 \).

```c
const uint64_t deBruijn = 0x022fdd63cc95386d;
const 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 \( s \).

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

\( \text{convert}[6] \Rightarrow 4 \)

Performance
Limited by multiply and table look-up.

Example: \( k=3 \)

\[
\begin{array}{c|c}
0 & 000 \\
1 & 001 \\
2 & 011 \\
3 & 111 \\
4 & 110 \\
5 & 101 \\
6 & 010 \\
7 & 100 \\
\end{array}
\]

\( \text{const int} \)
\( \text{convert}[8] = \{0,1,6,2,7,5,4,3\}; \)
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.
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.
Backtracking Search

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

![Chessboard with queens placed row by row](image)

*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.
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
\[ \text{down} \& (1 \ll c) \]
is nonzero.

\begin{verbatim}
1 1 0 1 1 1 0 1
\end{verbatim}

down
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 $\text{right} \& (1 \ll (n-r+c))$ 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></th>
<th>00101101110100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>00101101110100000</td>
</tr>
<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.

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); // \((0^{1})^{32}\)

// 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);
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

\[
\begin{array}{ccccccccccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[x \& M0\]
\[(x \gg 1) \& M0\]
# Population Count III

The image shows a binary addition process with a compare and mask operation. The table and diagram illustrate the steps of adding two binary numbers and then applying a mask operation. The steps are as follows:

1. The first binary number is:
   ```
   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
   ```
2. The second binary number is:
   ```
   1 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 1 1 0 0
   ```
3. The addition is performed column by column:
   ```
   +
   1 0 0 0 0 1 1 0 1 1 1 1 1 0 0 0 1 1 0
   ```
4. The result of the addition is:
   ```
   1 0 0 0 0 0 1 0 1 0 1 0 1 1 1 1 0 0 1 1 0 0
   ```
5. Finally, the result is compared with a zero mask (x&M0) and then shifted right by one position ((x>>1)&M0).

The masked result is:
```
1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0
```
### Population Count III

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

\[ \text{x} \& M_0 \]

\[ (\text{x} \gg 1) \& M_0 \]

\[ \text{x} \& M_1 \]

\[ (\text{x} \gg 2) \& M_1 \]
### Population Count III

| 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 0 1 0 0 0 1 1 1 1 1 0 0 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| +               | 1 0 0 0 1 1 0 1 1 1 0 1 0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 1 0|
| x&M0 (x>>1)&M0   | 0 0 0 1 0 1 1 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 1 0 0 0 1 1 0 |
| x&M1 (x>>2)&M1   | 1 0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 1 |
|                 | 0 0 1 0 0 0 1 1 0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 1 0 0 1 |
### Population Count III

|      | 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 | 1 1 0 1 1 1 0 1 0 0 0 1 1 1 1 1 0 0 0 |
|      |                                   | x&M0 (x>>1)&M0 (x>>2)&M1 (x>>4)&M2 |
|      | 1 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1 0 0 |                                   |
|      | 1 0 0 1 0 0 1 1 1 1 1 0 0 0 1 1 1 0 |                                   |
|      | +                                   |                                   |
| 0 0 1 | 0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 | x&M1 (x>>2)&M1 (x>>4)&M2 |
| 1 0 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 1 1 1 0 0 0 1 | x&M2 (x>>4)&M2 |
## Population Count III

<table>
<thead>
<tr>
<th></th>
<th>11000010010111110110001111000</th>
<th>(x &amp; M_0) ((x \gg 1) &amp; M_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>100011101111101100011100</td>
<td>(x &amp; M_1) ((x \gg 2) &amp; M_1)</td>
</tr>
<tr>
<td></td>
<td>00010101101000100</td>
<td>(x &amp; M_2) ((x \gg 4) &amp; M_2)</td>
</tr>
<tr>
<td></td>
<td>0000001100000011100000101010000100</td>
<td></td>
</tr>
</tbody>
</table>
### 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 1 1 0 1 0 0 0 1 1 1 1 0 0 0</th>
<th>x&amp;M0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 1 0 1 1 1 1 1 1 1 0 1 1 0 0</td>
<td>(x&gt;&gt;1)&amp;M0</td>
</tr>
<tr>
<td>+ 1 0 0 1 0 0 1 1 1 1 1 1 0 0 0 1 1 0</td>
<td>x&amp;M1</td>
</tr>
<tr>
<td></td>
<td>(x&gt;&gt;2)&amp;M1</td>
</tr>
<tr>
<td>0 0 0 1 0 1 1 0 1 0 0 0 1 0 0</td>
<td>x&amp;M2</td>
</tr>
<tr>
<td>+ 1 0 0 0 0 1 0 1 1 0 1 0 1 0 1 0 1 0 1</td>
<td>(x&gt;&gt;4)&amp;M2</td>
</tr>
<tr>
<td></td>
<td>x&amp;M4</td>
</tr>
<tr>
<td>0 0 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 0</td>
<td>(x&gt;&gt;8)&amp;M4</td>
</tr>
</tbody>
</table>

© 2013–2014 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 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| $x \& M_0$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \ll 1 \& M_0$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \& M_1$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \ll 2 \& M_1$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \& M_2$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \ll 4 \& M_2$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \& M_4$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| $x \ll 8 \& M_4$ |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &amp; M_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \ll 1 &amp; M_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x &amp; M_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \ll 2 &amp; M_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x &amp; M_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \ll 4 &amp; M_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x &amp; M_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \ll 8 &amp; M_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
## Population Count III

| 11000011010110111111010001111000 | x&M0  
|----------------------------------|------  
| + 10001111110110001110 | (x>>1)&M0  
| + 00 01 01 10 10 00 10 00 | x&M1  
| + 10 00 01 01 10 01 01 01 | (x>>2)&M1  
| + 0001 0011 0001 0001 | x&M2  
| + 0010 0010 0100 0011 | (x>>4)&M2  
| + 00000101 00000100 | x&M3  
| + 00000011 00000101 | (x>>8)&M3  
| + 000000001001 | x&M4  
| + 000000001001 | (x>>16)&M4  
| 00000000000000000000000000000000000000000000000001001 |  

© 2013-2014 Charles E. Leiserson and I-Ting Angelina Lee
### Population Count III

<table>
<thead>
<tr>
<th></th>
<th>11000010010111111010001111000</th>
<th>+</th>
<th>10001001111101100000111000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0001</td>
<td>+</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>+</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td>01011011</td>
<td>+</td>
<td>0000010000</td>
</tr>
<tr>
<td></td>
<td>11000010</td>
<td>+</td>
<td>0000010101</td>
</tr>
<tr>
<td></td>
<td>000000000000000000000000000000010001</td>
<td>+</td>
<td>000000000000000000000000000000010000</td>
</tr>
</tbody>
</table>

17

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee
Population Count III

Parallel divide–and–conquer

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

// 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
\(\Theta(lg w)\) 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!