Binary Representation

Let \( x = \langle x_{w-1}x_{w-2}\ldots x_0 \rangle \) be a \( w \)-bit computer word. The unsigned integer value stored in \( x \) is

\[
x = \sum_{k=0}^{w-1} x_k 2^k .
\]

For example, the 8-bit word \( \text{0b10010110} \) represents the unsigned value \( 150 = 2 + 4 + 16 + 128 \).

The prefix \( \text{0b} \) designates a Boolean constant.

The signed integer (two’s complement) value stored in \( x \) is

\[
x = \left( \sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1} .
\]

For example, the 8-bit word \( \text{0b10010110} \) represents the signed value \( -106 = 2 + 4 + 16 - 128 \).

© 2008–2018 by the MIT 6.172 Lecturers
Two’s Complement

We have $0b00...0 = 0$.

What is the value of $x = 0b11...1$?

\[
x = \left( \sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1}
\]

\[
= \left( \sum_{k=0}^{w-2} 2^k \right) - 2^{w-1}
\]

\[
= (2^{w-1} - 1) - 2^{w-1}
\]

\[
= -1.
\]
### Important identity

Since we have \( x + \sim x = -1 \), it follows that

\[
-x = \sim x + 1.
\]

### Example

\[
\begin{align*}
x & = \text{Ob011011000} \\
\sim x & = \text{Ob100100111} \\
-x & = \text{Ob100101000}
\end{align*}
\]
## Binary and Hexadecimal

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
<td>10</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
<td>11</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
<td>12</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
<td>13</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
<td>14</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
<td>15</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

The prefix _0x_ designates a hex constant.

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

**Example:** 0xDECD1DE2CODE4F00D is

```
1101111011000001110111100010110000001101111001001111000000001101
```

© 2008-2018 by the MIT 6.172 Lecturers
C Bitwise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR (exclusive OR)</td>
</tr>
<tr>
<td>~</td>
<td>NOT (one’s complement)</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
</tbody>
</table>

Examples (8-bit word)

\[
\begin{align*}
A &= \text{0b10110011} \\
B &= \text{0b01101001}
\end{align*}
\]

\[
\begin{align*}
A\&B &= \text{0b00100001} & \sim A &= \text{0b01001100} \\
A|B &= \text{0b11111011} & A \gg 3 &= \text{0b00010110} \\
A^B &= \text{0b11011010} & A \ll 2 &= \text{0b11001100}
\end{align*}
\]
Set the kth Bit

Problem
Set kth bit in a word x to 1.

Idea
Shift and OR.

Example
k = 7

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1011110101101101</td>
</tr>
<tr>
<td>1 &lt;&lt; k</td>
<td>0000000010000000</td>
</tr>
<tr>
<td>x</td>
<td>(1 &lt;&lt; k)</td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
Clear the kth Bit

Problem
Clear the kth bit in a word \( x \).

Idea
Shift, complement, and AND.

Example
\( k = 7 \)

<table>
<thead>
<tr>
<th>( x )</th>
<th>1011110111101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 &lt;&lt; k )</td>
<td>0000000010000000</td>
</tr>
<tr>
<td>( \sim(1 &lt;&lt; k) )</td>
<td>1111111101111111</td>
</tr>
<tr>
<td>( x &amp; \sim(1 &lt;&lt; k) )</td>
<td>1011110101101101</td>
</tr>
</tbody>
</table>
# Toggle the kth Bit

## Problem
Flip the $k$th bit in a word $x$.

## Idea
Shift and XOR.

$$y = x \ ^\ (1 \ll k);$$

## Example (0 → 1)

$k = 7$

<table>
<thead>
<tr>
<th>x</th>
<th>1011110101101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 \ll k$</td>
<td>0000000010000000</td>
</tr>
<tr>
<td>$x \ ^\ (1 \ll k)$</td>
<td>1011110111101101</td>
</tr>
</tbody>
</table>
Toggle the kth Bit

Problem
Flip the kth bit in a word x.

Idea
Shift and XOR.

Example (1 → 0)
k = 7

<table>
<thead>
<tr>
<th>x</th>
<th>1011110111101101</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;&lt; k</td>
<td>0000000010000000</td>
</tr>
<tr>
<td>x ^ (1 &lt;&lt; k)</td>
<td>1011110101101101</td>
</tr>
</tbody>
</table>
Extract a Bit Field

Problem
Extract a bit field from a word \( x \).

Idea
Mask and shift.

Example

<table>
<thead>
<tr>
<th>shift</th>
<th>x</th>
<th>mask</th>
<th>x &amp; mask</th>
<th>x &amp; mask &gt;&gt; shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1011110101101101</td>
<td>0000011110000000</td>
<td>0000010100000000</td>
<td>000000000000001010</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.

Example
shift = 7

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>1011110101101101</td>
</tr>
<tr>
<td>$y$</td>
<td>00000000000000011</td>
</tr>
<tr>
<td>mask</td>
<td>0000011110000000</td>
</tr>
<tr>
<td>$x &amp; \sim \text{mask}$</td>
<td>1011100001101101</td>
</tr>
<tr>
<td>$x = (x &amp; \sim \text{mask}) \mid (y &lt;&lt; \text{shift})$;</td>
<td>1011100111101101</td>
</tr>
</tbody>
</table>
Set a Bit Field

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

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

For safety’s sake:
\[
\text{((y} \ll \text{shift}) \& \text{mask})
\]

Example
shift = 7

\[
\begin{array}{|c|c|}
\hline
x & 1011110101101101 \\
\hline
y & 0000000000000011 \\
\hline
\text{mask} & 0000011110000000 \\
\hline
\text{x} \& \sim \text{mask} & 1011100001101101 \\
\hline
\text{x} = (\text{x} \& \sim \text{mask}) \mid (\text{y} \ll \text{shift}); & 1011100111101101 \\
\hline
\end{array}
\]
Ordinary Swap

Problem
Swap two integers \( x \) and \( y \).

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

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

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

**Example**

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

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

\[
x = x \, ^\wedge \, y;
\]
\[
y = x \, ^\wedge \, y;
\]
\[
x = x \, ^\wedge \, y;
\]

**Example**

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

© 2008–2018 by the MIT 6.172 Lecturers
**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**

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

Problem
Swap $x$ 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>

© 2008–2018 by the MIT 6.172 Lecturers
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

<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:
$$(x \oplus y) \oplus y \Rightarrow x$$
No-Temp Swap (Instant Replay)

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

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

Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>10111101</td>
</tr>
<tr>
<td>$y$</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.

Mask with 1’s where bits differ.

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

Example

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

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

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

Flip bits in $y$ that differ from $x$.

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

**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:
\[
(x \oplus y) \oplus y \Rightarrow x
\]
No-Temp Swap (Instant Replay)

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

Flip bits in $x$ that differ from $y$.

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>

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.

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

**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: \((x \oplus y) \oplus y \Rightarrow x\)

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

Problem
Find the minimum $r$ of two integers $x$ and $y$.

```plaintext
if (x < y)
  r = x;
else
  r = y;
```

or

```plaintext
r = (x < y) ? x : y;
```

Performance
A mispredicted branch empties the processor pipeline.

Caveat
The compiler is usually smart enough to optimize away the unpredictable branch, but maybe not.
Problem
Find the minimum \( r \) of two integers \( x \) and \( y \) without using a branch.

\[
r = y \, ^\wedge \, ((x \, ^\wedge \, y) \, \& \, -(x \, < \, y)) ;
\]

Why it works

- The C language represents the Booleans \texttt{TRUE} and \texttt{FALSE} with the integers \texttt{1} and \texttt{0}, respectively.
- If \( x < y \), then \( -(x < y) \Rightarrow -1 \), which is all 1’s in two’s complement representation. Therefore, we have \( y \, ^\wedge \, (x \, ^\wedge \, y) \Rightarrow x \).
- If \( x \geq y \), then \( -(x < y) \Rightarrow 0 \). Therefore, we have \( y \, ^\wedge \, 0 \Rightarrow y \).
Merging Two Sorted Arrays

```
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--;
    }
}
```
Branching

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 modern machines using `clang -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 code.
• 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 \(0 \leq y < n\).

\[
\begin{align*}
r &= (x + y) \% n;
\end{align*}
\]

Division is expensive, unless by a power of 2.

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

Unpredictable branch is expensive.

\[
\begin{align*}
z &= x + y;
\end{align*}
\]
\[
\begin{align*}
r &= z - (n \& -(z >= n));
\end{align*}
\]

Same trick as minimum.
Problem
Compute $2^{\lceil \lg n \rceil}$.

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

Problem
Compute $2^{[\log_2 n]}$.

Example

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

00100000001010000
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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2^{\lceil \log_2 n \rceil}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>00100000001010000</td>
</tr>
<tr>
<td>101</td>
<td>00100000001001111</td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
Problem
Compute $2^{|\lg n|}$.

```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>n</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>00100000001010000</td>
<td>00100000001001111</td>
</tr>
<tr>
<td>00100000001001111</td>
<td>00110000001101111</td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00110000001101111111</td>
<td>00010000001101111111</td>
<td>0000100000011011111111</td>
<td>00111110001111111111</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>00100000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00100000001001111</td>
</tr>
<tr>
<td></td>
<td>00110000011011111</td>
</tr>
<tr>
<td></td>
<td>00111100011111111</td>
</tr>
<tr>
<td></td>
<td>00111111111111111</td>
</tr>
</tbody>
</table>
Round up to a Power of 2

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

Example

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

<table>
<thead>
<tr>
<th>n</th>
<th>00100000001010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>00100000001001111</td>
</tr>
<tr>
<td>n</td>
<td>00110000001101111</td>
</tr>
<tr>
<td>n</td>
<td>00111100011111111</td>
</tr>
<tr>
<td>n</td>
<td>00111111111111111</td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
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

<table>
<thead>
<tr>
<th>Example Value</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001010000</td>
<td>0010000001001111</td>
</tr>
<tr>
<td>0011000001101111</td>
<td>0011110001111111</td>
</tr>
<tr>
<td>0011111000111111</td>
<td>0011111111111111</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;
```

Example

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

Problem
Compute $2^{\lfloor \log n \rfloor}$.

```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
0011111000111111
0011111111111111
0100000000000000
```
Round up to a Power of 2

Problem
Compute \(2^{\lfloor \log n \rfloor}\).

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

Bit \(\lfloor \log n \rfloor - 1\) must be set

Example

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010000001010000</td>
<td>0010000001001111</td>
</tr>
<tr>
<td>0010000001101111</td>
<td>0011111000111111</td>
</tr>
<tr>
<td>0011111111111111</td>
<td>0100000000000000</td>
</tr>
</tbody>
</table>

Set bit \(\lfloor \log n \rfloor\)

Populate all bits to the right with 1
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
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 $x$.

```c
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 $(\sim x)+1$.

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

Problem
Compute \( \log x \), 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
};
int r = convert[(x * deBruijn) >> 58];
```
5 volunteers who can follow directions

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 $s$.

$0b00011101 \times 2^4 \Rightarrow 0b11010000$
$0b11010000 \gg 5 \Rightarrow 6$
convert[6] $\Rightarrow$ 4

Performance
Limited by multiply and table look-up.

Example: $k=3$

<table>
<thead>
<tr>
<th></th>
<th>00011101</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>

start with all 0’s

const 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. Count the number of possible solutions.
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.

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!
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 $2n-1$. 
Placing a queen in column $c$ is not safe if

$$\text{down} \& (1 \ll c);$$

is nonzero.
Placing a queen in row $r$ and column $c$ is not safe if

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

is nonzero.
Placing a queen in row \( r \) and column \( c \) is not safe if
\[
\text{right} \& (1 \ll (n-1-r+c))
\]
is nonzero.
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>0010110111010000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>0010110111010000</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.
Table look-up

```
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 depends on the size of $x$. The cost of memory operations is a major bottleneck. Typical costs:

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

per 64-byte cache line
// Create masks
M5 = ~((−1) << 32);  // 0^{32}1^{32}
M4 = M5 ^ (M5 << 16);  // (0^{16}1_{16})^2
M3 = M4 ^ (M4 << 8);  // (0^81^8)^4
M2 = M3 ^ (M3 << 4);  // (0^41^4)^8
M1 = M2 ^ (M2 << 2);  // (0^21^2)^16
MO = M1 ^ (M1 << 1);  // (01)^{32}

// Compute popcount
x = (((x >> 1) & MO) + (x & MO));
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

<table>
<thead>
<tr>
<th></th>
<th>11000001010110111111101000</th>
<th>x&amp;M0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1000011011111011000</td>
<td>(x&gt;&gt;1)&amp;M0</td>
</tr>
<tr>
<td></td>
<td>10010011110000110</td>
<td>x&amp;M1</td>
</tr>
<tr>
<td>+</td>
<td>10000101010010010101</td>
<td>(x&gt;&gt;2)&amp;M1</td>
</tr>
<tr>
<td></td>
<td>00010011100010010101</td>
<td>x&amp;M2</td>
</tr>
<tr>
<td>+</td>
<td>00110010001001001001</td>
<td>(x&gt;&gt;4)&amp;M2</td>
</tr>
<tr>
<td></td>
<td>0101101110000100</td>
<td>x&amp;M3</td>
</tr>
<tr>
<td>+</td>
<td>11000010000101010101</td>
<td>(x&gt;&gt;8)&amp;M3</td>
</tr>
<tr>
<td></td>
<td>00000000000001000001</td>
<td>x&amp;M4</td>
</tr>
<tr>
<td>+</td>
<td>00000000000000100000</td>
<td>(x&gt;&gt;16)&amp;M4</td>
</tr>
<tr>
<td></td>
<td>0000000000000100010001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
Parallel divide-and-conquer

// Create masks
M5 = ~((-1) << 32);  // \(0^{32}\)
M4 = M5 ^ (M5 << 16);  // \((0^{16})^2\)
M3 = M4 ^ (M4 << 8);  // \((0^8)^4\)
M2 = M3 ^ (M3 << 4);  // \((0^4)^8\)
M1 = M2 ^ (M2 << 2);  // \((0^2)^{16}\)
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
\(\Theta(\lg w)\) time, where \(w = \text{word length}\).

Avoid overflow

No worry about overflow.
Most modern machines provide \texttt{popcount} instructions, which operate much faster than anything you can code yourself. You can access them via compiler intrinsics, e.g., in GCC:

\begin{verbatim}
int __builtin_popcount (unsigned int x);
\end{verbatim}

\textbf{Warning:} You may need to enable certain compiler switches to access built-in functions, and your code may be less portable.

\section*{Exercise}
Compute the log base 2 of a power of 2 quickly using a \texttt{popcount} instruction.
Sean Eron Anderson, “Bit twiddling hacks,”


Happy Bit–Hacking!