Bitwise Operations

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

06.01.2009
Basics

- In this lecture, we assume 32-bit wide two’s complement arithmetic for integers

- Fundamental identities of bit operations

  \[
  \begin{align*}
  0 \land x &= 0 & 0 \lor x &= x & 0 \oplus x &= x \\
  -1 \land x &= x & -1 \lor x &= -1 & -1 \oplus x &= \overline{x} \\
  x \land x &= x & x \lor x &= x & x \oplus x &= 0 \\
  \overline{x} \land x &= 0 & \overline{x} \lor x &= -1 & \overline{x} \oplus x &= -1
  \end{align*}
  \]

- Relate bit operations to arithmetic:

  \[
  x + \overline{x} = -1
  \]

- Leads to

  \[
  -x = \overline{x} + 1
  \]

- And finally

  \[
  x - y = x + \overline{y} + 1
  \]
Basics

Setting and deleting bits

- **Setting bit** \( m \)
  \[
  x \leftarrow x \mid (1 \ll m)
  \]

- **Clear bit** \( m \)
  \[
  x \leftarrow x \& \overline{1 \ll m}
  \]

- **Create mask** \( m = 0^a1^b0^c \) to set/clear multiple bits:
  \[
  ((1 \ll b) - 1) \ll c
  \]
  or
  \[
  (1 \ll (b + c)) - (1 \ll c)
  \]

- **Analogously for the inverted mask** \( m = 1^a0^b1^c \)

- **Special cases for** \( c = 0 \):

<table>
<thead>
<tr>
<th>Bitstring</th>
<th>Production</th>
<th>Example (( b = 3 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0^\infty 1^b</td>
<td>((1 \ll b) - 1)</td>
<td>7</td>
</tr>
<tr>
<td>1^\infty 0^b</td>
<td>(-(1 \ll b))</td>
<td>-8</td>
</tr>
</tbody>
</table>
Rightmost Bits

Let us consider the rightmost bits in a word

\[ x = \alpha 0 \underbrace{11 \ldots 1}_{a} 0 \ldots 0 = \alpha 01^a 10^b \quad a \geq 0, \, b \geq 0, \, \alpha \in \{0, 1\}^* \]
Rightmost Bits

- Let us consider the rightmost bits in a word

\[ x = \alpha 01 \ldots 110 \ldots 0 = \alpha 01^a 10^b \quad a \geq 0, \ b \geq 0, \ \alpha \in \{0, 1\}^* \]

- Then we have

\[
\begin{align*}
x & = \alpha 01^a 10^b \\
\overline{x} & = \overline{\alpha} 10^a 01^b \\
x - 1 & = \alpha 01^a 01^b \\
-x & = \overline{\alpha} 10^a 10^b
\end{align*}
\]
Rightmost Bits

Let us consider the rightmost bits in a word

\[ x = \alpha_0 1 \ldots 1 1 0 \ldots 0 = \alpha_0 1^a 0^b \quad a \geq 0, \ b \geq 0, \ \alpha \in \{0, 1\}^* \]

Then we have

\[ x = \alpha_0 1^a 0^b \]
\[ \overline{x} = \overline{\alpha} 0^a 1^b \]
\[ x - 1 = \alpha_0 1^a 0^b \]
\[ -x = \overline{\alpha} 0^a 1^b \]

and

\[ x \& (x - 1) = \]
Rightmost Bits

Let us consider the rightmost bits in a word

\[ x = \alpha 0 1 \ldots 1 0 \ldots 0 = \alpha 0 1^a 10^b \quad a \geq 0, \, b \geq 0, \, \alpha \in \{0, 1\}^* \]

Then we have

\[
\begin{align*}
x &= \alpha 0 1^a 10^b \\
\overline{x} &= \overline{\alpha} 10^a 01^b \\
x - 1 &= \alpha 0 1^a 01^b \\
-x &= \overline{\alpha} 10^a 10^b
\end{align*}
\]

and

\[
x \& (x - 1) = \alpha 0 1^a 00^b \quad \text{clear rightmost 1}
\]

\[
\text{test against 0 to check if } x \text{ is power-of-two}
\]

\[
x \& -x =
\]
Rightmost Bits

- Let us consider the rightmost bits in a word

\[ x = \alpha \underbrace{01 \ldots 11}_{a} \underbrace{0 \ldots 0}_{b} = \alpha 01^a 10^b \quad a \geq 0, \ b \geq 0, \ \alpha \in \{0, 1\}^* \]

- Then we have

\[
\begin{align*}
x &= \alpha 01^a 10^b \\
\bar{x} &= \overline{\alpha} 10^b 01^a \\
x - 1 &= \alpha 01^a 01^b \\
-x &= \overline{\alpha} 10^b 10^a \\
\end{align*}
\]

- and

\[
\begin{align*}
x \& (x - 1) &= \alpha 01^a 00^b \quad \text{clear rightmost 1} \\
&\text{test against 0 to check if } x \text{ is power-of-two} \\
\end{align*}
\]

\[
\begin{align*}
x \& -x &= 0^\infty 00^a 10^b \quad \text{isolate rightmost 1} \\
\bar{x} \& (x - 1) &= \\
\end{align*}
\]
Rightmost Bits

- Let us consider the rightmost bits in a word

\[ x = \alpha 01 \ldots 1 \underbrace{10 \ldots 0}_{a} = \alpha 01^a 10^b \quad a \geq 0, \ b \geq 0, \ \alpha \in \{0, 1\}^* \]

- Then we have

\[
\begin{align*}
  x & = \alpha 01^a 10^b \\
  \overline{x} & = \overline{\alpha} 10^a 01^b \\
  x - 1 & = \alpha 01^a 01^b \\
  -x & = \overline{\alpha} 10^a 10^b
\end{align*}
\]

- and

\[
\begin{align*}
  x \& (x - 1) & = \alpha 01^a 00^b \quad & \text{clear rightmost 1} \\
  x \& -x & = 0^\infty 00^a 10^b \quad & \text{test against 0 to check if } x \text{ is power-of-two} \\
  \overline{x} \& (x - 1) = x \mid -x & = 0^\infty 00^a 01^b \quad & \text{isolate rightmost 1} \\
  x \mid (x - 1) & =
\end{align*}
\]
Rightmost Bits

- Let us consider the rightmost bits in a word

\[ x = \alpha01\ldots110\ldots0 = \alpha01^a10^b \quad a \geq 0, \ b \geq 0, \ \alpha \in \{0, 1\}^* \]

- Then we have

\[
\begin{align*}
    x &= \alpha01^a10^b \\
    \bar{x} &= \bar{\alpha}0^a01^b \\
    x - 1 &= \alpha01^a01^b \\
    -x &= \bar{\alpha}01^a10^b
\end{align*}
\]

- and

\[
\begin{align*}
    x \& (x - 1) &= \alpha01^a00^b \quad \text{clear rightmost 1} \\
    x \& -x &= 0^\infty00^a10^b \quad \text{isolate rightmost 1} \\
    \bar{x} \& (x - 1) = x \| -x &= 0^\infty00^a01^b \quad \text{bitmask for lower zeroes} \\
    x \| (x - 1) &= \alpha01^a11^b \quad \text{right-propagate rightmost 1}
\end{align*}
\]
Exclusive Or

- Exclusive Or ($\oplus$) can serve as identity and not:
  \[
  x = x \oplus 0 \\
  \overline{x} = x \oplus \overline{0}
  \]

- Enables “conditional” not when condition is in sign bit
  \[
  y = c < 0 \ ? \ \sim x : x;
  \]

  equals
  \[
  y \leftarrow (c \gg 31) \oplus x
  \]
Exclusive Or

- Exclusive Or (⊕) can serve as identity and not:

  \[ x = x \oplus 0 \]
  \[ \overline{x} = x \oplus 0 \]

- Enables “conditional” not when condition is in sign bit

  \[ y = c < 0 \ ? \ \neg x : x; \]

  equals

  \[ y \leftarrow (c \gg 31) \oplus x \]

- Nice absolute value function:

  ```c
  static inline int abs(int x) {
    int t = x >> (sizeof(int) * 8 - 1);
    return (x ^ t) - t;
  }
  ```
3-Way Comparison

Compare functions often require 3-way compare:

\[
\text{cmp}(x, y) = \begin{cases} 
-1 & x < 0 \\
0 & x = 0 \\
1 & x > 0 
\end{cases}
\]

One way:

```c
int cmp(int x, int y) {
    if (x > y)
        return 1;
    if (x < y)
        return -1;
    return 0;
}
```

Without branches:

```c
int cmp(int x, int y) {
    return (x > y) - (x < y);
}
```

Look for yourself what code your compiler generates
Saturating Addition/Subtraction

- Sometimes you want addition/subtraction not to overflow but to saturate

\[
sadd(x, y) = \begin{cases} 
\text{MAX\_INT} & z(x) + z(y) \geq z(\text{MAX\_INT}) \\
\text{MIN\_INT} & z(x) + z(y) \leq z(\text{MIN\_INT}) \\
x + y & \text{otherwise}
\end{cases}
\]

(Note: \( z : 2^{32} \to \mathbb{Z} \) embeds integers into \( \mathbb{Z} \))
Saturating Addition/Subtraction

- Sometimes you want addition/subtraction not to overflow but to saturate

\[ \text{sadd}(x, y) = \begin{cases} 
\text{MAX\_INT} & z(x) + z(y) \geq z(\text{MAX\_INT}) \\
\text{MIN\_INT} & z(x) + z(y) \leq z(\text{MIN\_INT}) \\
x + y & \text{otherwise}
\end{cases} \]

(Note: \( z : 2^{32} \rightarrow \mathbb{Z} \) embeds integers into \( \mathbb{Z} \))

- So, how can we check if an addition overflowed?

static inline int sadd(int x, int y) {
    int sum = x + y;
    int overflow = (x ^ s) & (y ^ s);
    int big = (x >> 31) ^ INT_MAX;
    return overflow < 0 ? big : sum;
}
**Saturating Addition/Subtraction**

- Sometimes you want addition/subtraction not to overflow but to saturate

\[
sadd(x, y) = \begin{cases} 
\text{MAX\_INT} & z(x) + z(y) \geq z(\text{MAX\_INT}) \\
\text{MIN\_INT} & z(x) + z(y) \leq z(\text{MIN\_INT}) \\
x + y & \text{otherwise}
\end{cases}
\]

(Note: \( z : 2^{32} \rightarrow \mathbb{Z} \) embeds integers into \( \mathbb{Z} \))

- So, how can we check if an addition overflowed?
- If operands have different signs, there cannot be an overflow
Saturating Addition/Subtraction

- Sometimes you want addition/subtraction not to overflow but to saturate

\[
sadd(x, y) = \begin{cases} 
\text{MAX}_\text{INT} & z(x) + z(y) \geq z(\text{MAX}_\text{INT}) \\
\text{MIN}_\text{INT} & z(x) + z(y) \leq z(\text{MIN}_\text{INT}) \\
x + y & \text{otherwise}
\end{cases}
\]

(Note: \( z : 2^{32} \to \mathbb{Z} \) embeds integers into \( \mathbb{Z} \))

- So, how can we check if an addition overflowed?
- If operands have different signs, there cannot be an overflow
- If the signs are equal and the sum's sign is different, we had an overflow:

\[
\text{overflow} = (x \oplus s) \& (y \oplus s) \quad s = x + y
\]

- \( \text{overflow} \) has sign bit set, if \( x + y \) overflowed
Saturating Addition/Subtraction

- Sometimes you want addition/subtraction not to overflow but to saturate

\[
sadd(x, y) = \begin{cases} 
\text{MAX_INT} & z(x) + z(y) \geq z(\text{MAX_INT}) \\
\text{MIN_INT} & z(x) + z(y) \leq z(\text{MIN_INT}) \\
x + y & \text{otherwise}
\end{cases}
\]

(Note: \( z : 2^{32} \rightarrow \mathbb{Z} \) embeds integers into \( \mathbb{Z} \))

- So, how can we check if an addition overflowed?
- If operands have different signs, there cannot be an overflow
- If the signs are equal and the sum's sign is different, we had an overflow:

\[
\text{overflow} = (x \oplus s) \& (y \oplus s) \quad s = x + y
\]

- \textit{overflow} has sign bit set, if \( x + y \) overflowed

```c
static inline int sadd(int x, int y) {
    int sum = x + y;
    int overflow = (x ^ s) & (y ^ s);
    int big = (x >> 31) ^ INT_MAX;
    return overflow < 0 ? big : sum;
}
```
Rounding Up/Down to a Multiple of a Known Power of 2

- Rounding to some next power of 2 can be used for binning (remember malloc lecture)
- Rounding up (down) here means round to $+\infty$ ($-\infty$)
- Rounding down is easy:
  \[ x \& -n \]
  rounds down to next $2^k = n$
- Rounding up is almost as easy:
  \[ (x + (n - 1)) \& -n \]
- Round to nearest power of 2 toward 0:
  \[ (x + t) \& -n \quad t = (x \gg 31) \& (n - 1) \]
Rounding Up/Down to the Next Power of 2

\[ flp2(x) = \begin{cases} 
\text{undefined} & x < 0 \\
0 & x = 0 \\
2^{\lfloor \log_2 x \rfloor} & x > 0 
\end{cases} \]

\[ clp2(x) = \begin{cases} 
\text{undefined} & x < 0 \\
0 & x = 0 \\
2^{\lceil \log_2 x \rceil} & x > 0 
\end{cases} \]

- \( flp2 \) means isolating the leftmost bit
  (remember how easy this was for the rightmost!)
- We need to propagate the highest set bit down
Rounding Up/Down to the **Next** Power of 2

\[
flp2(x) = \begin{cases} 
  \text{undefined} & x < 0 \\
  0 & x = 0 \\
  2^{\lfloor \log_2 x \rfloor} & x > 0 
\end{cases}
\]

\[
clp2(x) = \begin{cases} 
  \text{undefined} & x < 0 \\
  0 & x = 0 \\
  2^{\lceil \log_2 x \rceil} & x > 0 
\end{cases}
\]

- `flp2` means isolating the leftmost bit (remember how easy this was for the rightmost!)
- We need to propagate the highest set bit down

```c
unsigned flp2(unsigned x) {
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >>16);
    return x - (x >> 1);
}
```

- The first five lines create a band of 1
- `x - (x >> 1)` isolates the most significant one
Rounding Up/Down to the Next Power of 2

\[ flp2(x) = \begin{cases} 
\text{undefined} & x < 0 \\
0 & x = 0 \\
2^{\lfloor \log_2 x \rfloor} & x > 0 
\end{cases} \]

\[ clp2(x) = \begin{cases} 
\text{undefined} & x < 0 \\
0 & x = 0 \\
2^{\lceil \log_2 x \rceil} & x > 0 
\end{cases} \]

- \textit{flp2} means isolating the leftmost bit (remember how easy this was for the rightmost!)
- We need to propagate the highest set bit down

```c
unsigned flp2(unsigned x) {
    x = x | (x >> 1);
    x = x | (x >> 2);
    x = x | (x >> 4);
    x = x | (x >> 8);
    x = x | (x >>16);
    return x - (x >> 1);
}
```

- The first five lines create a band of 1
- \( x - (x >> 1) \) isolates the most significant one
- If we have an instruction \( nlz \) that gives the number of leading zeroes:

\[ flp2(x) = 1 \ll (nlz(x) \oplus 31) \]
Number of Leading Zeroes (nlz)

- Find most significant set bit
- Basically the discrete binary logarithm
- Very useful for bit sets (remember last lecture)
- GCC has it as a compiler-known function `ffs`
- Many machines feature it as a native instruction `bsr` (bit scan reverse) on x86 (since i386)

```c
unsigned nlz(unsigned x) {
    unsigned y, n = 32;
    y = x >> 16; if (y) { n = n - 16; x = y; }
    y = x >> 8; if (y) { n = n - 8; x = y; }
    y = x >> 4; if (y) { n = n - 4; x = y; }
    y = x >> 2; if (y) { n = n - 2; x = y; }
    y = x >> 1; if (y) return n - 2;
    return n - x;
}
```
Number of Leading Zeroes (nlz)

- Find most significant set bit
- Basically the discrete binary logarithm
- Very useful for bit sets (remember last lecture)
- GCC has it as a compiler-known function `ffs`
- Many machines feature it as a native instruction `bsr` (bit scan reverse) on x86 (since i386)
- Binary-search implementation in C if not available as machine instr

```c
unsigned nlz(unsigned x) {
    unsigned y, n = 32;
    y = x >>16; if (y) { n = n -16; x = y; }
    y = x >> 8; if (y) { n = n - 8; x = y; }
    y = x >> 4; if (y) { n = n - 4; x = y; }
    y = x >> 2; if (y) { n = n - 2; x = y; }
    y = x >> 1; if (y) return n - 2;
    return n - x;
}
```

- Unfortunately has jumps
Portably using Inline Assembly

Using nlz as an Example

```c
static inline unsigned nlz(unsigned x) {
    #if defined(__GNUC__) && defined(__i386__)
    unsigned res;
    if(x == 0) return 32;
    __asm__("bsrl \%1,\%0" : "=r" (res) : "r" (x));
    return 31 - res;
    #else
    unsigned y, n = 32;
    y = x >>16; if (y != 0) { n -= 16; x = y; }
    y = x >> 8; if (y != 0) { n -= 8; x = y; }
    y = x >> 4; if (y != 0) { n -= 4; x = y; }
    y = x >> 2; if (y != 0) { n -= 2; x = y; }
    y = x >> 1; if (y != 0) return n - 2;
    return n - x;
    #endif
}
```

- Use compiler and platform define to check for the right flavor of inline assembler and CPU architecture
- **Always** provide a C version
Number of Trailing Zeroes

... and de Bruijn Numbers

- How can we find the number of trailing zeroes?

**Idea 1** Reduce problem to numbers that have only one bit set
  - We can do that easily by applying \(x \& -x\)

**Idea 2** Use hashing:
  - There are 32 numbers with 1 bit
  - Create a function \(h(x)\) that maps each one bit number to the bit’s position
  - Hash table should be small
  - Hash function easy to compute
  - Hash function should be collision-free

**Idea 3** Use de Bruijn Numbers for the hash function
### Definition (de Bruijn Sequence)

A length-$n$ ($n = 2^k$) de Bruijn sequence $s$ is a sequence of $n$ 0’s and 1’s such that every 0-1 sequence of length $k$ occurs exactly once as a contiguous substring.

### Example for $k = 3$

A length-8 de Bruijn sequence is

\[ 00011101 \]

Move a 3-bit window right (one bit at a time, wrapping around):

\[ 000, 001, 011, 111, 110, 101, 010, 100 \]

- Every 0,1-sequence of length $k$ has a unique index in 00011101
- E.g.: 000 has index 0, 010 has index 6, and so on
Number of Trailing Zeroes

... and de Bruijn Numbers

\[ h(x) = (x \times B) \gg (n - \log_2 n) \]

- \( B \) is a number whose bits are a de Bruijn sequence
- \( x \) has only one set bit
- \( x \times B \) shifts \( B \) left by \( \log_2 x \)
- Read out the upper \( \log_2 n \) bits of \( x \times B \)
- That value will be different for every \( x \)
- Index a table with \( h(x) \) and read out the number of trailing zeroes for \( x \)
Example for $n = 8$

- Use de Bruijn number $B = 00011101$
- Let $x' = 00101100$, number of trailing zeroes is 2
- $x = x' \& -x' = 00000100$
- $x \times 00011101 = 01110100$ ($00011101 \ll \log_2 x$)
- Take out the upper $\log_2 n = 3$ bits: 011
- Index the table with 011 should get 2 then
Counting Bits

- How many bits are set in a word (population count)?
- Using the things we already learned (by B. Kernighan)

```c
unsigned popcnt(unsigned x) {
    unsigned c;
    for (c = 0; x; c++)
        x &= x - 1; // clear the least significant bit set
    return c;
}
```

takes too long, has jumps, worst case 32 iterations

- We can use “divide and conquer”
Population Count
Divide and Conquer
Population Count
Simple Version

- Add bit $2k$ to bit $2k + 1$
- Then add two bits at $4k$ to the bits at $4k + 2$
- and so on

```c
unsigned popcnt(unsigned x) {
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f);
    x = (x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff);
    x = (x & 0x0000ffff) + ((x >>16) & 0x0000ffff);
    return x;
}
```
Population Count
Simple Version

- Add bit $2k$ to bit $2k + 1$
- Then add two bits at $4k$ to the bits at $4k + 2$
- and so on

```c
unsigned popcnt(unsigned x) {
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f);
    x = (x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff);
    x = (x & 0x0000ffff) + ((x >>16) & 0x0000ffff);
    return x;
}
```

- Can be tuned further
Population Count

Tuned Version

- Adding the 2-bits can be done more efficiently:
  - We need following mapping:
    - $00b \rightarrow 00b$
    - $01b \rightarrow 01b$
    - $10b \rightarrow 01b$
    - $11b \rightarrow 10b$
  - $x - (x \gg 1)$ does the trick
  - need still to mask with 0x55555555 to clear down-shifted bits

$$x \leftarrow x - ((x \gg 1) \& 0x55555555)$$
Population Count

Tuned Version

- Adding the 2-bits can be done more efficiently:
  - We need following mapping:
    
    | 00b | → | 00b |
    | 01b | → | 01b |
    | 10b | → | 01b |
    | 11b | → | 10b |
  
  - $x - (x \gg 1)$ does the trick
  - need still to mask with 0x55555555 to clear down-shifted bits

$$x \leftarrow x - ((x \gg 1) \& 0x55555555)$$

- Adding the 4-bit groups can also be optimized:
  - Each 4-bit group’s value is at most 100b (it is the number of set bits in 4 bits)
  - Hence, the largest value of the sum of two 4-bit groups is 1000b
  - That fits into 4 bits
  - Need only to mask the result: $x \leftarrow (x + (x \gg 4)) \& 0x0f0f0f0f$

```
x = 0aaa0bbb0ccc0ddd0eee0fff0ggg0hhh
x >> 4 = 00000aaa0bbb0ccc0ddd0eee0fff0ggg
sum = 0aaawwwww????xxxx????yyyy????zzzz
```
Population Count

Tuned Version: Final step

- Our value now looks like this:
  0000www0000xxx0000yyyy0000zzzz
  we need the sum www + xxx + yyyy + zzzz

unsigned popcnt(unsigned x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x + (x >> 4)) & 0x0f0f0f0f;
  return (x * 0x01010101) >> 24;
}
Population Count

Tuned Version: Final step

- Our value now looks like this:
  0000wwwww0000xxxx0000yyyy0000zzzz
  we need the sum $www + xxxx + yyyy + zzzz$

- Multiply by $0x01010101$:
  - equals $x + (x \ll 8) + (x \ll 16) + (x \ll 24)$
  - Accumulates the desired sum in the upper 8 bits ($tt$)

$0w0x0y0z \ast 01010101 =$
  $0w0x0y0z$
  $0w0x0y0z$
  $0w0x0y0z$
  $0w0x0y0z$
  $00????????:tt????0z$

Final version:

```c
unsigned popcnt(unsigned x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0f0f0f0f;
    return (x * 0x01010101) >> 24;
}
```
Population Count

Tuned Version: Final step

- Our value now looks like this:
  0000wwww0000xxxx0000yyyy0000zzzz
  we need the sum www + xxxx + yyyy + zzzz
- Multiply by 0x01010101:
  - equals \( x + (x \ll 8) + (x \ll 16) + (x \ll 24) \)
  - Accumulates the desired sum in the upper 8 bits (tt)

\[ \begin{align*}
0w0x0y0z \times 01010101 &= \quad \text{:0w0x0y0z} \\
0w:0x0y0z &\\
0w0x:0y0z &\\
0w0x0y:0z &\\
00???????:tt????0z
\end{align*} \]

- Final version:

```c
unsigned popcnt(unsigned x) {
  x = x - ((x >> 1) & 0x55555555);
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
  x = (x + (x >> 4)) & 0x0f0f0f0f;
  return (x * 0x01010101) >> 24;
}
```
References

- Henry S. Warren, Jr.
  Hacker’s Delight
  Addison Wesley, 2003

- Donald Knuth
  The Art of Computer Programming, Volume 4, Pre-Fascicle 1A
  http://www-cs-faculty.stanford.edu/~uno/fasc1a.ps.gz