Bitwise Operations

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

06.01.2009

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Basics

- In this lecture, we assume 32-bit wide two's complement arithmetic for integers
- Fundamental identities of bit operations

$$
\begin{array}{rcl}
0 \,\&\, x = 0 & 0 \mid x = x & 0 \oplus x = x \\
-1 \,\&\, x = x & -1 \mid x = -1 & -1 \oplus x = \overline{x} \\
x \,\&\, x = x & x \mid x = x & x \oplus x = 0 \\
\overline{x} \,\&\, x = 0 & \overline{x} \mid x = -1 & \overline{x} \oplus x = -1
\end{array}
$$

- 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

$$x \leftarrow x \mathbin{\&} \overline{1 \ll m}$$

- Create mask $m = 0^a 1^b 0^c$ to set/clear multiple bits:

$$((1 \ll b) - 1) \ll c$$

or

$$(1 \ll (b + c)) - (1 \ll c)$$

- Analogously for the inverted mask $m = 1^a 0^b 1^c$
- Special cases for $c = 0$:

Bitstring	Production	Example ($b = 3$)
$0^\infty 1^b$	$(1 \ll b) - 1$	7
$1^\infty 0^b$	$-(1 \ll b)$	-8

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1 \dots 1}_{a} 1 \underbrace{0 \dots 0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0, 1\}^*$$

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1 \ldots 1}_{a} 1 \underbrace{0 \ldots 0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0,1\}^*$$

- Then we have

$$\begin{aligned} x &= \alpha 0 1^a 1 0^b \\ \overline{x} &= \overline{\alpha} 1 0^a 0 1^b \\ x - 1 &= \alpha 0 1^a 0 1^b \\ -x &= \overline{\alpha} 1 0^a 1 0^b \end{aligned}$$

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1 \ldots 1}_{a} 1 \underbrace{0 \ldots 0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0,1\}^*$$

- Then we have

$$\begin{array}{rcl} x & = & \alpha 0 1^a 1 0^b \\ \overline{x} & = & \overline{\alpha} 1 0^a 0 1^b \\ x - 1 & = & \alpha 0 1^a 0 1^b \\ -x & = & \overline{\alpha} 1 0^a 1 0^b \end{array}$$

- and

$$x \,\&\, (x-1) \quad =$$

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1 \ldots 1}_{a} 1 \underbrace{0 \ldots 0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0,1\}^*$$

- Then we have

$$\begin{array}{rcl} x & = & \alpha 0 1^a 1 0^b \\ \overline{x} & = & \overline{\alpha} 1 0^a 0 1^b \\ x - 1 & = & \alpha 0 1^a 0 1^b \\ -x & = & \overline{\alpha} 1 0^a 1 0^b \end{array}$$

- and

$$\begin{array}{rcll} x \,\&\, (x-1) & = & \alpha 0 1^a 0 0^b & \text{clear rightmost 1} \\ & & \multicolumn{2}{l}{\text{test against 0 to check if } x \text{ is power-of-two}} \\ x \,\&\, -x & = & & \end{array}$$

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1 \ldots 1}_{a} 1 \underbrace{0 \ldots 0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0,1\}^*$$

- Then we have

$$\begin{array}{rcl} x & = & \alpha 0 1^a 1 0^b \\ \overline{x} & = & \overline{\alpha} 1 0^a 0 1^b \\ x - 1 & = & \alpha 0 1^a 0 1^b \\ -x & = & \overline{\alpha} 1 0^a 1 0^b \end{array}$$

- and

$$\begin{array}{rcll} x \mathbin{\&} (x-1) & = & \alpha 0 1^a 0 0^b & \text{clear rightmost 1} \\ & & & \text{test against 0 to check if } x \text{ is power-of-two} \\ x \mathbin{\&} -x & = & 0^\infty 0 0^a 1 0^b & \text{isolate rightmost 1} \end{array}$$

$\overline{x} \mathbin{\&} (x-1) =$

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1 \dots 1}_{a} 1 \underbrace{0 \dots 0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0, 1\}^*$$

- Then we have

$$\begin{array}{rcl} x & = & \alpha 0 1^a 1 0^b \\ \overline{x} & = & \overline{\alpha} 1 0^a 0 1^b \\ x - 1 & = & \alpha 0 1^a 0 1^b \\ -x & = & \overline{\alpha} 1 0^a 1 0^b \end{array}$$

- and

$$\begin{array}{rcll} x \,\&\, (x-1) & = & \alpha 0 1^a 0 0^b & \text{clear rightmost 1} \\ & & & \text{test against 0 to check if } x \text{ is power-of-two} \\ x \,\&\, -x & = & 0^\infty 0 0^a 1 0^b & \text{isolate rightmost 1} \\ \overline{x} \,\&\, (x-1) = \overline{x \mid -x} & = & 0^\infty 0 0^a 0 1^b & \text{bitmask for lower zeroes} \\ x \mid (x-1) & = & & \end{array}$$

Rightmost Bits

- Let us consider the rightmost bits in a word

$$x = \alpha 0 \underbrace{1\dots1}_{a} 1 \underbrace{0\dots0}_{b} = \alpha 0 1^a 1 0^b \qquad a \geq 0, b \geq 0, \alpha \in \{0,1\}^*$$

- Then we have

$$\begin{array}{rcl} x & = & \alpha 0 1^a 1 0^b \\ \overline{x} & = & \overline{\alpha} 1 0^a 0 1^b \\ x - 1 & = & \alpha 0 1^a 0 1^b \\ -x & = & \overline{\alpha} 1 0^a 1 0^b \end{array}$$

- and

$$\begin{array}{rcll} x \mathbin{\&} (x-1) & = & \alpha 0 1^a 0 0^b & \text{clear rightmost 1} \\ & & & \text{test against 0 to check if } x \text{ is power-of-two} \\ x \mathbin{\&} -x & = & 0^\infty 0 0^a 1 0^b & \text{isolate rightmost 1} \\ \overline{x} \mathbin{\&} (x-1) = \overline{x \mid -x} & = & 0^\infty 0 0^a 0 1^b & \text{bitmask for lower zeroes} \\ x \mid (x-1) & = & \alpha 0 1^a 1 1^b & \text{right-propagate rightmost 1} \end{array}$$

Exclusive Or

- Exclusive Or ($\oplus$) can serve as identity and not:

$$\begin{aligned} x &= x \oplus 0 \\ \overline{x} &= x \oplus \overline{0} \end{aligned}$$

- Enables "conditional" not when condition is in sign bit

```
y = c < 0 ? ~x : x;
```

equals

$$y \leftarrow (c \overset{s}{\gg} 31) \oplus x$$

Exclusive Or

- Exclusive Or ($\oplus$) can serve as identity and not:

$$\begin{aligned} x &= x \oplus 0 \\ \overline{x} &= x \oplus \overline{0} \end{aligned}$$

- Enables "conditional" not when condition is in sign bit

```
y = c < 0 ? ~x : x;
```

equals

$$y \leftarrow (c \overset{s}{\gg} 31) \oplus x$$

- Nice absolute value function:

```
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:

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

- One way:

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

- Without branches:

```
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

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

(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

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

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

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

Saturating Addition/Subtraction

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

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

(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

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

(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:

$$overflow = (x \oplus s) \;\&\; (y \oplus s) \qquad s = x + y$$

- *overflow* has sign bit set, if $x + y$ overflowed

Saturating Addition/Subtraction

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

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

(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:

$$overflow = (x \oplus s) \,\&\, (y \oplus s) \qquad s = x + y$$

- *overflow* has sign bit set, if $x + y$ 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;
}
```

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:

  $$x \mathbin{\&} -n$$

  rounds down to next $2^k = n$
- Rounding up is almost as easy:

  $$(x + (n - 1)) \mathbin{\&} -n$$

- Round to nearest power of 2 toward 0:

  $$(x + t) \mathbin{\&} -n \qquad t = (x \overset{s}{\gg} 31) \mathbin{\&} (n - 1)$$

Rounding Up/Down to the Next Power of 2

$$
flp2(x) = \begin{cases} \text{undefined} & x < 0 \\ 0 & x = 0 \\ 2^{\lfloor \log_2 x \rfloor} & x > 0 \end{cases} \qquad clp2(x) = \begin{cases} \text{undefined} & x < 0 \\ 0 & x = 0 \\ 2^{\lceil \log_2 x \rceil} & x > 0 \end{cases}
$$

- `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

$$
flp2(x) = \begin{cases} \text{undefined} & x < 0 \\ 0 & x = 0 \\ 2^{\lfloor \log_2 x \rfloor} & x > 0 \end{cases} \qquad clp2(x) = \begin{cases} \text{undefined} & x < 0 \\ 0 & x = 0 \\ 2^{\lceil \log_2 x \rceil} & x > 0 \end{cases}
$$

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

```
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

$$flp2(x) = \begin{cases} \text{undefined} & x < 0 \\ 0 & x = 0 \\ 2^{\lfloor \log_2 x \rfloor} & x > 0 \end{cases} \qquad clp2(x) = \begin{cases} \text{undefined} & x < 0 \\ 0 & x = 0 \\ 2^{\lceil \log_2 x \rceil} & x > 0 \end{cases}$$

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

```
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)

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 Exmaple

```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 $x \mathbin{\&} -x$

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

Number of Trailing Zeroes

de Bruijn Sequences

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

$$h(x) = (x * B) \overset{u}{\gg} (n - \log_2 n)$$

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

Number of Trailing Zeroes

. . . and de Bruijn Numbers

Example for $n = 8$

- Use de Bruijn number $B = 00011101$
- Let $x' = 00101100$, number of trailing zeroes is 2
- $x = x' \,\&\, -x' = 00000100$
- $x * 00011101 = 01110100$ $(00011101 \ll \log_2 x)$
- 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)

```
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

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

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

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

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

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

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

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

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

```
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`	$\longrightarrow$	`00b`
`01b`	$\longrightarrow$	`01b`
`10b`	$\longrightarrow$	`01b`
`11b`	$\longrightarrow$	`10b`

  - $x - (x \overset{u}{\gg} 1)$ does the trick
  - need still to mask with `0x55555555` to clear down-shifted bits

$$x \leftarrow x - ((x \overset{u}{\gg} 1) \mathbin{\&} \texttt{0x55555555})$$

Population Count

Tuned Version

- Adding the 2-bits can be done more efficiently:
  - We need following mapping:

00b	$\longrightarrow$	00b
01b	$\longrightarrow$	01b
10b	$\longrightarrow$	01b
11b	$\longrightarrow$	10b

  - $x - (x \overset{u}{\gg} 1)$ does the trick
  - need still to mask with 0x55555555 to clear down-shifted bits

$$x \leftarrow x - ((x \overset{u}{\gg} 1) \ \& \ \texttt{0x55555555})$$

- 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: $x \leftarrow (x + (x \overset{u}{\gg} 4)) \ \& \ \texttt{0x0f0f0f0f}$

```
x      = 0aaa0bbb0ccc0ddd0eee0fff0ggg0hhh
x >> 4 = 00000aaa0bbb0ccc0ddd0eee0fff0ggg
sum    = 0aaawwww????xxxx????yyyy????zzzz
```

Population Count

Tuned Version: Final step

- Our value now looks like this:
  `0000wwww0000xxxx0000yyyy0000zzzz`
  we need the sum $\texttt{wwww} + \texttt{xxxx} + \texttt{yyyy} + \texttt{zzzz}$

Population Count

Tuned Version: Final step

- Our value now looks like this:

  `0000wwww0000xxxx0000yyyy0000zzzz`

  we need the sum $\texttt{wwww} + \texttt{xxxx} + \texttt{yyyy} + \texttt{zzzz}$
- 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 * 01010101 =
        :0w0x0y0z
      0w:0x0y0z
    0w0x:0y0z
  0w0x0y:0z
00??????:tt????0z
```

Population Count

Tuned Version: Final step

- Our value now looks like this:

  `0000wwww0000xxxx0000yyyy0000zzzz`

  we need the sum `wwww` + `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 * 01010101 =
        :0w0x0y0z
      0w:0x0y0z
    0w0x:0y0z
  0w0x0y:0z
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;
}
```

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