



Bitwise operations cheat sheet



Nicol Leung [Follow](#)

May 17, 2018 · 5 min read



Sign in to medium.com with Google



Adrian Alonzo

adriandalonzo@gmail.com



Adrian A.

adrianlikesgames@gmail.com

4 more accounts

Recommendations and additions to this cheat sheet are welcome.

This cheat sheet is mostly suitable for most common programming languages, but the target usage is C/C++ on x86 platform.

Bitmap `i` is unsigned 32 bit integers. For 64 bit operands, the suffix `l` should be added to integer literals, e.g. `1` should be `1l`.

All calculation demonstration is done with 8 bits integers for readability.

Truth table

AND

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

OR

0	0		0
0	1		1

1	0		1
1	1		1

XOR

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

Operators

AND	&
OR	
NOT	~
XOR	^
Left shift	<<
Right shift	>>

Get a bit

$$(i \gg n) \& 1$$

Set a bit to 1

$$i | (1 \ll n)$$

Set a bit to 0

$$i \& \sim(1 \ll n)$$

Store a bit

The bit to be stored is v which is either 0 or 1 .

$$(i \& \sim(1 \ll n)) \mid (v \ll n)$$

Toggle a bit

$$i \wedge (1 \ll n)$$

Get least significant bit

$$i \& -i$$

Note: this gives you really the lowest bit but not the index of the lowest bit.

$-i$ is equivalent to $\sim i + 1$

$\sim i$	10100111
$\sim i + 1$	10101000
i	01011000
$i \& -i$	00001000

Get most significant bit

```
unsigned int get_msb(unsigned int i){
    i |= i >> 1;
    i |= i >> 2;
    i |= i >> 4;
    i |= i >> 8;
    i |= i >> 16;
    return (i + 1) >> 1;
}
```

How it works:

```
i          01000010
i|=i>>1    01100011
i|=i>>2    01111011
i|=i>>4    01111111
...
i|=i>>16   01111111
i+1        10000000
(i+1)>>1    01000000
```

Get index of most significant bit

```
inline unsigned int get_bit_index(const unsigned int i){
    unsigned int r;
    asm ( "bsr %1, %0\n"
        : "=r"(r)
        : "r" (i)
        );
    return r;
}
```

Yah, just one instruction. This instruction supports 16/32/64 bit integers, source and destination type must have the same size.

Change endianness

Convert from big-endian to little-endian or vice-versa.

Well you shouldn't need to handcraft this function but anyway FYR:

```
((i>>24) & 0xFF)      | // Move byte 3 to byte 0
((i<<8) & 0xFF0000)   | // Move byte 1 to byte 2
((i>>8) & 0xFF00)     | // Move byte 2 to byte 1
((i<<24) & 0xFF000000) // Move byte 0 to byte 3
```

Bit reversal

Lookup table is actually faster:

```
static const unsigned char BitReverseTable256[] =
{
    0x00, 0x80, 0x40, 0xC0, 0x20, 0xA0, 0x60, 0xE0, 0x10, 0x90, 0x50,
    0xD0, 0x30, 0xB0, 0x70, 0xF0,
    0x08, 0x88, 0x48, 0xC8, 0x28, 0xA8, 0x68, 0xE8, 0x18, 0x98, 0x58,
    0xD8, 0x38, 0xB8, 0x78, 0xF8,
    0x04, 0x84, 0x44, 0xC4, 0x24, 0xA4, 0x64, 0xE4, 0x14, 0x94, 0x54,
    0xD4, 0x34, 0xB4, 0x74, 0xF4,
    0x0C, 0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C, 0x9C, 0x5C,
    0xDC, 0x3C, 0xBC, 0x7C, 0xFC,
    0x02, 0x82, 0x42, 0xC2, 0x22, 0xA2, 0x62, 0xE2, 0x12, 0x92, 0x52,
    0xD2, 0x32, 0xB2, 0x72, 0xF2,
    0x0A, 0x8A, 0x4A, 0xCA, 0x2A, 0xAA, 0x6A, 0xEA, 0x1A, 0x9A, 0x5A,
    0xDA, 0x3A, 0xBA, 0x7A, 0xFA,
    0x06, 0x86, 0x46, 0xC6, 0x26, 0xA6, 0x66, 0xE6, 0x16, 0x96, 0x56,
    0xD6, 0x36, 0xB6, 0x76, 0xF6,
    0x0E, 0x8E, 0x4E, 0xCE, 0x2E, 0xAE, 0x6E, 0xEE, 0x1E, 0x9E, 0x5E,
    0xDE, 0x3E, 0xBE, 0x7E, 0xFE,
    0x01, 0x81, 0x41, 0xC1, 0x21, 0xA1, 0x61, 0xE1, 0x11, 0x91, 0x51,
    0xD1, 0x31, 0xB1, 0x71, 0xF1,
```

```

    0x09, 0x89, 0x49, 0xC9, 0x29, 0xA9, 0x69, 0xE9, 0x19, 0x99, 0x59,
    0xD9, 0x39, 0xB9, 0x79, 0xF9,
    0x05, 0x85, 0x45, 0xC5, 0x25, 0xA5, 0x65, 0xE5, 0x15, 0x95, 0x55,
    0xD5, 0x35, 0xB5, 0x75, 0xF5,
    0x0D, 0x8D, 0x4D, 0xCD, 0x2D, 0xAD, 0x6D, 0xED, 0x1D, 0x9D, 0x5D,
    0xDD, 0x3D, 0xBD, 0x7D, 0xFD,
    0x03, 0x83, 0x43, 0xC3, 0x23, 0xA3, 0x63, 0xE3, 0x13, 0x93, 0x53,
    0xD3, 0x33, 0xB3, 0x73, 0xF3,
    0x0B, 0x8B, 0x4B, 0xCB, 0x2B, 0xAB, 0x6B, 0xEB, 0x1B, 0x9B, 0x5B,
    0xDB, 0x3B, 0xBB, 0x7B, 0xFB,
    0x07, 0x87, 0x47, 0xC7, 0x27, 0xA7, 0x67, 0xE7, 0x17, 0x97, 0x57,
    0xD7, 0x37, 0xB7, 0x77, 0xF7,
    0x0F, 0x8F, 0x4F, 0xCF, 0x2F, 0xAF, 0x6F, 0xEF, 0x1F, 0x9F, 0x5F,
    0xDF, 0x3F, 0xBF, 0x7F, 0xFF
};

unsigned int bit_reversal(const unsigned int i){
    return (BitReverseTable256[i & 0xFF] << 24) |
        (BitReverseTable256[(i >> 8) & 0xFF] << 16) |
        (BitReverseTable256[(i >> 16) & 0xFF] << 8) |
        (BitReverseTable256[(i >> 24) & 0xFF]);
}

```

Count the number of bits

Hamming weight is faster than lookup table according to Google's *"Director of Engineering" Hiring Test*. The actual performance test is here.

```

const uint64_t m1 = 0x55555555; // 0101...
const uint64_t m2 = 0x33333333; // 00110011...
const uint64_t m4 = 0x0F0F0F0F; // 0000111100001111...
const uint64_t m8 = 0x00FF00FF; // 8 zeros, 8 ones...
const uint64_t m16 = 0x0000FFFF; // 16 zeros, 16 ones...
const uint64_t m32 = 0x00000000ffffffff; // 32 zeros, 32 ones
const uint64_t h01 = 0x0101010101010101; //the sum of 256 to the
power of 0,1,2,3...

//This is a naive implementation, shown for comparison,
//and to help in understanding the better functions.
//This algorithm uses 24 arithmetic operations (shift, add, and).
int popcount64a(uint64_t x)
{
    x = (x & m1) + ((x >> 1) & m1); //put count of each 2 bits
    into those 2 bits
    x = (x & m2) + ((x >> 2) & m2); //put count of each 4 bits

```

```

into those 4 bits
    x = (x & m4) + ((x >> 4) & m4); //put count of each 8 bits
into those 8 bits
    x = (x & m8) + ((x >> 8) & m8); //put count of each 16 bits
into those 16 bits
    x = (x & m16) + ((x >> 16) & m16); //put count of each 32 bits
into those 32 bits
    x = (x & m32) + ((x >> 32) & m32); //put count of each 64 bits
into those 64 bits
    return x;
}

//This uses fewer arithmetic operations than any other known
//implementation on machines with slow multiplication.
//This algorithm uses 17 arithmetic operations.
int popcount64b(uint64_t x)
{
    x -= (x >> 1) & m1;           //put count of each 2 bits into
those 2 bits
    x = (x & m2) + ((x >> 2) & m2); //put count of each 4 bits into
those 4 bits
    x = (x + (x >> 4)) & m4;       //put count of each 8 bits into
those 8 bits
    x += x >> 8; //put count of each 16 bits into their lowest 8
bits
    x += x >> 16; //put count of each 32 bits into their lowest 8
bits
    x += x >> 32; //put count of each 64 bits into their lowest 8
bits
    return x & 0x7f;
}

//This uses fewer arithmetic operations than any other known
//implementation on machines with fast multiplication.
//This algorithm uses 12 arithmetic operations, one of which is a
multiply.
int popcount64c(uint64_t x)
{
    x -= (x >> 1) & m1;           //put count of each 2 bits into
those 2 bits
    x = (x & m2) + ((x >> 2) & m2); //put count of each 4 bits into
those 4 bits
    x = (x + (x >> 4)) & m4;       //put count of each 8 bits into
those 8 bits
    return (x * h01) >> 56; //returns left 8 bits of x + (x<<8) +
(x<<16) + (x<<24) + ...
}

```

Programming Bitwise Operator



30 claps



WRITTEN BY
Nicol Leung

Follow

Write the first response

More From Medium

Related reads

How to Ace the Coding Interview,
by an Ex-Facebook Interviewer

Ish Baid in Better Programming
Oct 10, 2019 · 8 min read ★



3.5K



Related reads

How does it feel like from Inside
Google?

Priya Saxena
Mar 24, 2019 · 4 min read ★



65



Related reads

An Introduction to Sliding
Window Algorithms

Jordan Moore in Level Up Coding
Nov 4, 2019 · 10 min read ★



207



Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just \$5/month. Upgrade

