

# Bit Twiddling Hacks

## Integers

David Barina

March 28, 2014

## Counting bits set: naive

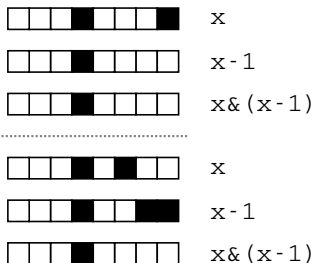
```
unsigned x;  
unsigned c;  
  
for(c = 0; x; x >>= 1)  
{  
    c += x & 1;  
}
```

a.k.a. population count, popcount

# Counting bits set: Kernighan

```
for(c = 0; x; c++)  
{  
    x &= x - 1;  
}
```

# Counting bits set: Kernighan



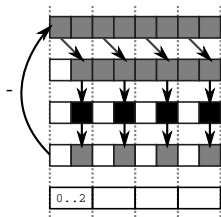
## Counting bits set: in parallel

```
x -= x >> 1 & ~0U/3;  
x = (x & ~0U/15*3) + (x >> 2 & ~0U/15*3);  
x = (x + (x >> 4)) & ~0U/255*15;  
c = (x * (~0U/255)) >> (sizeof(unsigned) - 1) * CHAR_BIT;
```

## Counting bits set

```
x -= x >> 1 & ~0U/3;  
x = (x & ~0U/15*3) + (x >> 2 & ~0U/15*3);  
x = (x + (x >> 4)) & ~0U/255*15;  
c = (x * (~0U/255)) >> (sizeof(unsigned) - 1) * CHAR_BIT;
```

# Counting bits set



x

x >> 1

&~0U/3

11 - 01 = 10 = 2

10 - 01 = 01 = 1

01 - 00 = 01 = 1

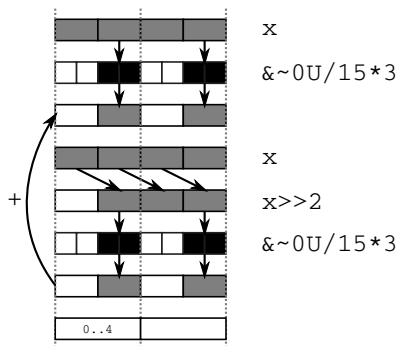
00 - 00 = 00 = 0

## Counting bits set

```
x -= x >> 1 & ~0U/3;  
x = (x & ~0U/15*3) + (x >> 2 & ~0U/15*3);  
x = (x + (x >> 4)) & ~0U/255*15;  
c = (x * (~0U/255)) >> (sizeof(unsigned) - 1) * CHAR_BIT;
```



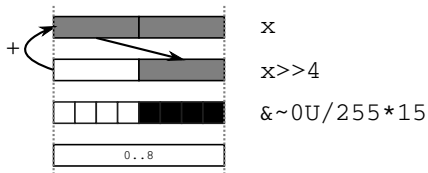
# Counting bits set



# Counting bits set

```
x -= x >> 1 & ~0U/3;  
x = (x & ~0U/15*3) + (x >> 2 & ~0U/15*3);  
x = (x + (x >> 4)) & ~0U/255*15;  
c = (x * (~0U/255)) >> (sizeof(unsigned) - 1) * CHAR_BIT;
```

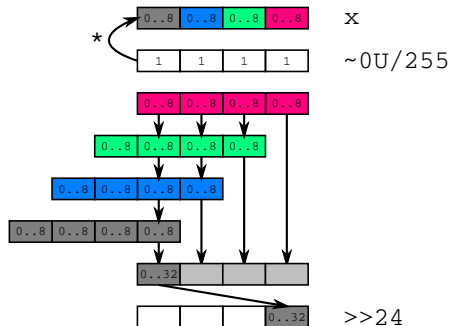
# Counting bits set



# Counting bits set

```
x -= x >> 1 & ~0U/3;  
x = (x & ~0U/15*3) + (x >> 2 & ~0U/15*3);  
x = (x + (x >> 4)) & ~0U/255*15;  
c = (x * (~0U/255)) >> (sizeof(unsigned) - 1) * CHAR_BIT;
```

# Counting bits set



# Counting bits set

algorithm	time
naive	30.943180 ns
Kernighan	17.374596 ns
parallel	4.136793 ns

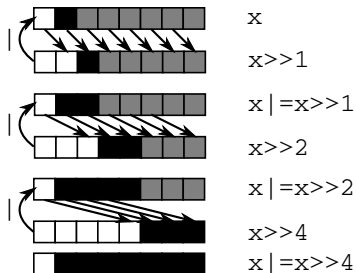
speedup  $7.5\times$

## Copy the highest set bit to all of the lower bits

```
x |= x >> 1;  
x |= x >> 2;  
x |= x >> 4;  
x |= x >> 8;  
x |= x >> 16;
```

a.k.a. copymsb

# Copy the highest set bit to all of the lower bits





## Copy the highest set bit to all of the lower bits

```
unsigned shift = 1;

while(shift < sizeof(int) * CHAR_BIT)
{
    x |= x >> shift;
    shift <=> 1;
}
```

## Copy the highest set bit to all of the lower bits

algorithm	time
unroll	3.687508 ns
loop	3.689689 ns

speedup 1.0×

## Find the log base 2 of an integer

$$\log_2(x) = -1 \quad : \quad x < 1$$

$$\lfloor \log_2(x) \rfloor = \lceil \log_2(x + 1) \rceil - 1$$

$$\lceil \log_2(x) \rceil = \lfloor \log_2(x - 1) \rfloor + 1$$

## Find the log base 2 of an integer: IEEE double

```
unsigned x;  
unsigned r;  
  
r = (unsigned)ceil(log2((double)x));  
  
r = (unsigned)floor(log2((double)x));
```

## Find the log base 2 of an integer: naive way (floor)

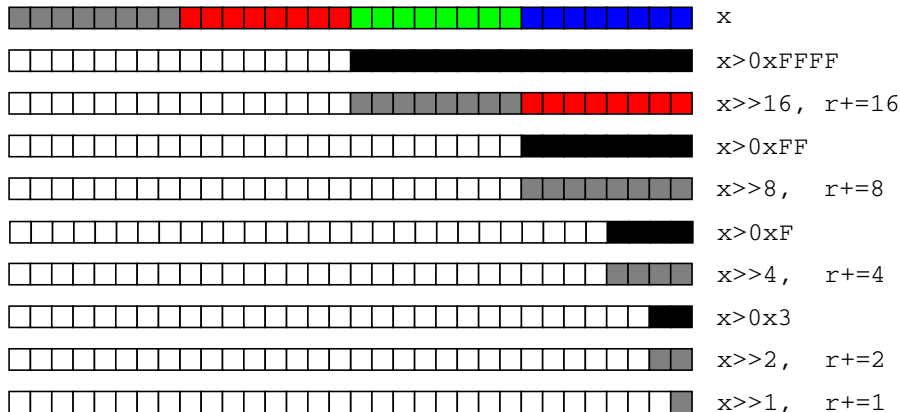
```
unsigned r = 0;
```

```
while(x >>= 1)
{
    r++;
}
```

## Find the log base 2 of an integer: fast way (floor)

```
unsigned r;  
unsigned shift;  
  
r =      (x > 0xFFFF) << 4; x >>= r;  
shift = (x > 0xFF   ) << 3; x >>= shift; r |= shift;  
shift = (x > 0xF    ) << 2; x >>= shift; r |= shift;  
shift = (x > 0x3    ) << 1; x >>= shift; r |= shift;  
r |= (x >> 1);
```

## Find the log base 2 of an integer: fast way (floor)



## Find the log base 2 of an integer: popcount (ceil)

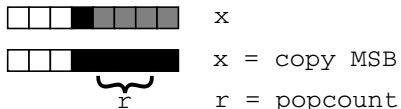
```
x--;
```

```
x = copysb(x); // next pow2 minus 1
```

```
r = popcount(x);
```



# Find the log base 2 of an integer: popcount (ceil)



## Find the log base 2 of an integer: DeBruijn (floor)

```
static const int MultiplyDeBruijnBitPosition[32] =  
{  
    0, 9, 1, 10, 13, 21, 2, 29,  
    11, 14, 16, 18, 22, 25, 3, 30,  
    8, 12, 20, 28, 15, 17, 24, 7,  
    19, 27, 23, 6, 26, 5, 4, 31  
};
```

```
x = copysb(x); // next pow2 minus 1
```

```
r = MultiplyDeBruijnBitPosition[  
    (unsigned)(x * 0x07C4ACDDU) >> 27];
```

## Find the log base 2 of an integer: fast loop (floor)

```
const unsigned int b[] = {
    0x2, 0xC, 0xF0, 0xFF00, 0xFFFF0000 };
const unsigned int S[] = { 1, 2, 4, 8, 16 };

register unsigned int r = 0;

for(int i = 4; i >= 0; i--)
{
    if(x & b[i])
    {
        x >>= S[i];
        r |= S[i];
    }
}
```

## Find the log base 2 of an integer: table (floor)

```
static const char LogTable256[256] =  
{  
#define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n  
    -1,  
    0,  
    1, 1,  
    2, 2, 2, 2,  
    3, 3, 3, 3, 3, 3, 3, 3,  
    LT(4),  
    LT(5), LT(5),  
    LT(6), LT(6), LT(6), LT(6),  
    LT(7), LT(7), LT(7), LT(7),  
    LT(7), LT(7), LT(7), LT(7)  
#undef LT  
};
```

## Find the log base 2 of an integer: table (floor)

```
if(t = x >> 24)
{
    r = 24 + LogTable256[t];
}
else if(t = x >> 16)
{
    r = 16 + LogTable256[t];
}
else if(t = x >> 8)
{
    r = 8 + LogTable256[t];
}
else
{
    r = LogTable256[x];
}
```

## Find the log base 2 of an integer

algorithm	floor	ceil
double	64.648230 ns	64.569502 ns
naive	20.249388 ns	20.179684 ns
fast	12.542725 ns	12.199326 ns
popcount	8.794856 ns	8.409071 ns
DeBruijn	4.519947 ns	4.809108 ns
fast loop	3.784315 ns	3.716202 ns
table	1.834092 ns	1.835536 ns

speedup  $> 35\times$

## Next power of 2: copymsb

```
x--;  
x = copymsb(x);  
x++;
```

## Next power of 2: table

```
x = 1 << ceil_log2(x);
```



## Next power of 2

algorithm	time
copym sb	3.976506 ns
log2 table	2.169806 ns

speedup  $> 1.8\times$

# Is power of 2

```
if( 0 == (x & (x-1)) )  
{  
    // ...  
}
```

## Select minimum, maximum: naive

$(x < y) ? x : y$

$(x > y) ? x : y$

## Select minimum, maximum: xor

```
((x-y) >> (sizeof(int)*CHAR_BIT-1)) & (x^y) ^ y
```

```
((x-y) >> (sizeof(int)*CHAR_BIT-1)) & (y^x) ^ x
```

---

```
x = y ^ (x^y)
```

```
y = y ^ 0
```

```
(-z) >> 31 = 0xffffffff
```

```
(+z) >> 31 = 0x00000000
```

## Select minimum, maximum

algorithm	min	max
naive	1.509076 ns	1.503108 ns
xor	2.035053 ns	1.847343 ns

speedup  $0.7\times$

## Absolute value (32-bit)

```
// stdlib.h  
printf("%i\n", abs(+1000000000));  
printf("%i\n", abs(-1000000000));  
printf("%i\n", abs(-2147483648));
```

## Absolute value (32-bit)

```
// stdlib.h
printf("%i\n", abs(+1000000000));
printf("%i\n", abs(-1000000000));
printf("%i\n", abs(-2147483648));
```

---

1000000000

1000000000

-2147483648

## Negate (32-bit)

```
// negate all bits and add "1"  
printf("%i\n", negate(+1000000000));  
printf("%i\n", negate(-1000000000));  
printf("%i\n", negate(-2147483648));
```



## Negate (32-bit)

```
// negate all bits and add "1"  
printf("%i\n", negate(+1000000000));  
printf("%i\n", negate(-1000000000));  
printf("%i\n", negate(-2147483648));
```

---

-1000000000

1000000000

-2147483648

## Overflow (32-bit and 64-bit)

```
printf("%i\n", (int)+2147483648); // long int => int  
printf("%li\n", +2147483648); // long int
```

## Overflow (32-bit and 64-bit)

```
printf("%i\n", (int)+2147483648); // long int => int  
printf("%li\n", +2147483648); // long int
```

---

-2147483648

2147483648

# Sources

- x86, Intel Core2 Quad @ 2.00 GHz
- gcc 4.8 -O3
- <https://www.google.com/?q=Bit+Twiddling>
- <http://graphics.stanford.edu/~seander/bithacks.html>
- <http://www.fefe.de/intof.html>