Binary Numbers

Representing Binary Numbers

Computers operate on 1's and 0's; these are called 'bits'. A byte is a group of 8 bits, like this example: 00110101. A computer word on a 32-bit computer ('int') is 4 bytes, 32 bits: 10011010110101011010001010101011. Other computers have different word sizes; over time, 64-bit integers will become more common.

As you can see, 32 ones and zeroes is a bit cumbersome to write down (or even to read). Thus, people conventionally break these large numbers of digits down into groups of 3 or 4 bits:

```
  1001.1010.1101.0101.1010.0010.1010.1011    <-- four bit groups
  10.011.010.110.101.011.010.001.010.101.011 <-- three bit groups
                            (note that the count of 3 starts on the right)
```

These grouped sets of bits are then mapped onto digits, either four bits per hexadecimal (base 16) digit or three bits per octal (base 8) digit. Obviously, hexadecimal needs some new digits (since decimal digits only go 0..9 and 6 more are needed). These days, the letters 'A'..'F' are used for the 'digits' that represent 10..15. Here's the map; the correspondence is obvious:

```
    OCTAL:                    HEXADECIMAL:
  000 -> 0  100 -> 4       0000 -> 0  0100 -> 4  1000 -> 8  1100 -> C
  001 -> 1  101 -> 5       0001 -> 1  0101 -> 5  1001 -> 9  1101 -> D
  010 -> 2  110 -> 6       0010 -> 2  0110 -> 6  1010 -> A  1110 -> E
  011 -> 3  111 -> 7       0011 -> 3  0111 -> 7  1011 -> B  1111 -> F
(both upper and lower case A-F are used across different computers and
operating systems).
```

The hex and octal representations of those integers above are easy to translate from the binary counterparts; add 0x to the front for a C-style-language hexadecimal number that the compiler will accept:

```
        1001.1010.1101.0101.1010.0010.1010.1011
   ->     9    A    D    5    A    2    A    B  --> 0x9AD5A2AB
                             (that's 0x in front of the hex number)
```

and

```
        10.011.010.110.101.011.010.001.010.101.011
         2   3   2   6   5   3   2   1   2   5   3 -> 023265321253
                             (that's a numeric '0' in front)
```

Octal is easier to write down quickly, but hexadecimal has the nice properties of breaking easily into bytes (which are pairs of hexadecimal digits).

Some aids for remembering the correspondence of hexadecimal (often called 'hex') digits and their decimal digit counterpart:

- hex 0-9 are the same as decimal digits 0-9

- A is the first one past 9 and is easy to remember as 10
- F is the last one and thus easy to remember as 15
- C is decimal 12 (the only one that you sort of have to memorize)
- All the rest are close to A, C, or F (B is just A+1, D is C+1, E is F-1)

If someone mentions the "third bit" of a number, its best to find out if they mean third bit from the left or from the right and whether they start counting from 0 or 1. A miscommunication can result in real problems later on (i.e., reversed strings of bits!).

Almost all modern computers use the left-most bit as the 'sign bit' which signifies a negative integer if its value is 1. Note that identifying the location of the sign bit requires knowledge of precisely how many bits are in a given data type. This number can change over time (i.e., when you recompile on a different computer). Generally, the sizeof() operator will tell you the number of **bytes** (which are generally 8 bits) its argument contains, e.g., sizeof(int) or sizeof (100) will yield 4 on a 32-bit machine. Sometimes one can identify a system-based include file that contains the proper constants if a program must depend on a certain word (integer) length.

Operating on Binary Numbers in Programs

Sometimes it is handy to work with the bits stored in numbers rather than just treating them as integers. Examples of such times include remembering choices (each bit slot can be a 'yes'/'no' indicator), keeping track of option flags (same idea, really, each bit slot is a 'yes'/'no' indicator for a flag's presence), or keeping track of a number of small integers (e.g., successive pairs of bit slots can remember numbers from 0..3). Of course, occasionally programming tasks actually contain 'bit strings'.

In C/C++ and others, assigning a binary number is easy if you know its octal or hexadecimal representation:

```
i = 0x9AD5A2AB;                 /* hexadecimal: 0x */
```

or

```
i = 023265321253;               /* octal: start with 0 */
```

More often, a pair of single-bit valued integers is combined to create an integer of interest. One might think the statement below would do that:

```
i = 0x10000 + 0x100;
```

and it will -- until the sign bit enters the picture or the same bit is combined twice:

```
i = 0x100 + 0x100;
```

In that case, a 'carry' occurs (0x100 + 0x100 = 0x200 which is probably not the result you want) and then i contains 0x200 instead of 0x100 as probably desired. The 'or' operation -- denoted as '|' in C/C++ and others -- does the right thing. It combines corresponding bits in its two operands using these four rules:

```
0 | 0 -> 0
0 | 1 -> 1
1 | 0 -> 1
1 | 1 -> 1
```

The '|' operation is called 'bitwise or' in C so as not to be confused with it's cousin '||'

called 'logical or' or 'orif'. The '||' operator evaluates the arithmetic value of its left side operand and, if that value is false (exactly 0), it evaluates its right side operand. The 'orif' operator is different: if either operand is nonzero, then '||' evaluates to true (exactly 1 in C).

It is "1 | 1 = 1" that distinguishes the '|' operator from '+'. Sometimes operators like this are displayed as a 'truth table':

```
                right operand
operator | | 0  1
        ---+------
   left  0 | 0  1    <-- results
operand  1 | 1  1    <-- results
```

It's easy to see that the 'bitwise or' operation is a way to set bits inside an integer. A '1' results with either or both of the input bits are '1'.

The easy way to query bits is using the 'logical and' (also known as 'bitwise and') operator which is denoted as '&' and has this truth table:

```
 & | 0  1
---+------
 0 | 0  0
 1 | 0  1
```

Do not confuse the single '&' operator with its partner named 'andif' with two ampersands ('&&'). The 'andif' operator will evaluate its left side and yield 0 if the left side is false, without evaluating its right side operand. Only if the left side is true will the right side be evaluated, and the result of the operator is the logical 'and' of their truth values (just as above) and is evaluated to either the integer 0 or the integer 1 (vs. '&' which would yield 4 when evaluating binary 100100 & 000111).

A '1' results only when *both* input bits are '1'. So, if a program wishes to know if the 0x100 bit is '1' in an integer, the if statement is simple:

```
if (a & 0x100) { printf("yes, 0x100 is on\n"); }
```

C/C++ (and others) contain additional operators, including 'exclusive or' (denoted '^') with this truth table:

```
 ^ | 0  1
---+------
 0 | 0  1
 1 | 1  0
```

The 'exclusive or' operator is sometimes called 'xor', for ease of typing. Xor yields a '1' either exactly *one* of its inputs is one: either one or the other, but not both. This operator is very handy for 'toggling' (flipping) bits, changing them from '1' to '0' or vice-versa. Consider this statement:

```
a = a ^ 0x100;   /* same as a ^= 0x100; */
```

The 0x100 bit will be changed from 0->1 or 1->0, depending on its current value.

Switching off a bit requires two operators. The new one is the unary operator that toggles every bit in a word, creating what is called the 'bitwise complement' or just 'complement' of a word. Sometimes this is called 'bit inversion' or just 'inversion' and is denoted by the tilde: '~'. Here's a quick example:

```
char a, b;      /* eight bits, not 32 or 64 */
a = 0x4A;       /* binary 0100.1010 */
b = ~a;         /* flip every bit: 1011.0101 */
printf("b is 0x%X\n", b);
```

which yields:

```
b is 0xB5
```

Thus, if we have a single bit switched on (e.g., 0x100) then ~0x100 has all but one bit switched on: 0xFFFFFEFF (note the 'E' as the third 'digit' from the right) (this example shows a 32-bit value; a 64-bit value would have a lot more F's on the front).

These two operators combine to create a scheme for switching off bits:

```
a = a & (~0x100);       /* switch off the 0x100 bit */
                        /* same as a &= ~0x100;
```

since all but one bit in ~0x100 is on, all the bits except the 0x100 bits appear in the result. Since the 0x100 bit is 'off' in ~0x100, that bit is guaranteed to be '0' in the result. This operation is universally called 'masking' as in 'mask off the 0x100 bit'.

Summary

In summary, these operators enable setting, clearing, toggling, and testing any bit or combination of bits in an integer:

```
a |= 0x20;              /* turn on bit 0x20 */
a &= ~0x20;             /* turn off bit 0x20 */
a ^= 0x20;              /* toggle bit 0x20 */
if (a & 0x20) {
        /* then the 0x20 bit is on */
}
```

Shifting

Moving bits to the left or right is called 'shift'ing. Consider a five-bit binary number like 00110. If that number is shifted one bit left, it becomes: 01100. On the other hand, if 00110 is shift one bit to the right, it becomes 000011. Mathematically inclined users will realize that shifting to the left one bit is the same as multiplying by 2 while shifting to the right one bit is usually the same as an integer divide by 2 (i.e., one discards any remainder). Why usually? Shifting -1 right by one yields an unusual result (i.e,, 0xFFFF.FFFF >> 1 == 0xFFFF.FFFF, no change at all).

Generally, one can specify a shift by more than one bit: Shifting 000001 to the left by three bits yields 001000. The shift operators are:

```
a << n        /* shift a left n bits
a >> n        /* shift a right n bits
```

One generally shifts integers (instead of floating-point numbers), although nothing prevents shifting floating point numbers in some languages. The results are generally very difficult to interpret as a floating point number, of course.

When shifting to the left, 0's are inserted in the lower end. **When shifting to the right, the high order bit is duplicated and inserted for the newly-needed bit** (thus preserving the number's sign). This means that (-1)>>1 yields -1 instead of 0!

Another type of shift, unavailable natively in most programming languages is the 'circular' shift, where bits shifted off one end are inserted at the other end instead of the default 0. Modern uses of this operation are rare but could appear occasionally. Some machines (e.g., the x86 architecture) have assembly-language instructions for this, but the prudent C or Java programming will spend a few more machine cycles (billionths of a second) to execute both a shift and then a bit-extract-shift-or combination to move the bit themselves.

A note on optimization: some zealous optimizing coders like to change a/4 to '(a>>2)' in order to save time ("since multiplies can be slow"). Modern compilers know all about this and perform such substitutions automatically, thus leaving the better programmer to write a/4 when that is what's meant.

Very Advanced Bit Manipulation

Skip this section if this is your first time dealing with bits. Read it at your leisure in the future after you've written some bit manipulation code. Really.

It turns out that the way 2's complement machines represent integers and the way they implement subtraction (the standard on virtually all modern machines) yields some very interesting possibilities for bit manipulation.

Two's complement machines represent positive integers as the binary representation of that integer with a 0 in the sign bit. A negative integer is represented as the complement of the positive integer (including turning on the sign bit) plus 1. Thus, the absolute value of the most negative representable integer is one more than the most positive representable integer. Thus:

```
   x    +x in 8-bit binary    -x in 8-bit binary
   0        0000 0000             0000 0000
   1        0000 0001             1111 1111
   2        0000 0010             1111 1110
   3        0000 0011             1111 1101
  64        0100 0000             1100 0000
  65        0100 0001             1011 1111
 126        0111 1110             1000 0010
 127        0111 1111             1000 0001
 128    [no representation]       1000 0000
```

Given this representation, addition can proceed just as on pencil and paper, discard any extra high order bits that exceed the length of the representation. Subtracting b from a (a-b) proceeds as add a and the quantity (-b).

This means that certain bit operations can exploit these definitions of negation and subtraction to yield interesting results, the proofs of which are left to the reader (see [a table of these] offsite):

```
             Binary
Value        Sample            Meaning
  x         00101100        the original x value
x & -x      00000100        extract lowest bit set
x | -x      11111100        create mask for lowest-set-bit & bits to its left
x ^ -x      11111000        create mask bits to left of lowest bit set
x & (x-1)   00101000        strip off lowest bit set
                            --> useful to process words in O(bits set)
                                instead of O(nbits in a word)
x | (x-1)   00101111        fill in all bits below lowest bit set
```

```
x ^ (x-1)   00000111        create mask for lowest-set-bit & bits to its right
~x & (x-1)  00000011        create mask for bits to right of lowest bit set
x | (x+1)   00101101        toggle lowest zero bit
x / (x&-x)  00001011        shift number right so lowest set bit is at bit 0
```

There's no reason to memorize these expressions, but rather remember what's possible to refer back to this page for saving time when you processing bits.

Back to USACO Training Gateway | Comment or Question