Bitwise Operations

CSE 220: Systems Programming

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Bitwise Operations

We have seen arithmetic and logical integer operations.

C also supports bitwise operations.

These operations correspond to circuit elements.

They are often related to, yet different from, logical operations.

The major operations are:

- Bitwise complement
- Bit shifts (left and right)
- Bitwise AND, OR, and XOR

Planning: Diagrams

After a once-through of the documentation, start drawing.

(See the PA2 Handout video!)

Draw the ideas:

- Data structures
  - Abstract the structure!
  - Arrows represent connections
  - Boxes represent data storage
  - …
- Program flow
  - Read args → open inputs → …
  - (This is more interesting if it's non-linear!)

Truth Tables

You should already be familiar with truth tables.

Every bitwise operation (except shift) is defined by a truth table.

A truth table represents one or two input bits and their output bit.

For example, bitwise OR:

x	y	Result
0	0	0
1	0	1
0	1	1
1	1	1

Bitwise Operations

OR ($\vee$):

x	y	Result
0	0	0
1	0	1
0	1	1
1	1	1

XOR ($\oplus$):

x	y	Result
0	0	0
1	0	1
0	1	1
1	1	0

AND ($\wedge$):

x	y	Result
0	0	0
1	0	0
0	1	0
1	1	1

NOT ($\neg$):

x	Result
0	1
1	0

Bit Operations on Words

Each of these bit operations can be applied to a word.

Each bit position will have the operation applied individually.

E.g., the application of XOR to an n-bit word is:

$$\forall_{i=0}^{n-1} \text{Result}_i = x_i \oplus y_i$$

Each operation applies to a single bit, so no carries are needed.

Bit Shifting

Bit shifts are slightly more complicated.

C can shift bits left or right.

- Left shift (<<): bits move toward larger bit values
- Right shift (>>): bits move toward smaller bit values

For left shift, zeroes are shifted in on the right.

Examples:

0111 left shift 1 bit $\rightarrow$ 1110

0010 left shift 2 bits $\rightarrow$ 1000

Right Shifts

Right shifts are somewhat trickier.

In particular, they may obey sign extension.

If the shifted integer is unsigned, zeroes are shifted in on the left:

0110 right shift 1 bit → 0011
1010 right shift 2 bits → 0010

If the shifted integer is signed, the sign bit may affect the shift.

- If it is zero, shifts behave as unsigned
- If it is one, it might shift in ones

If [the shifted value] is a signed type and a negative value, the resulting value is implementation-defined. — ISO C99

Operators

The C bitwise operators divide into unary and binary operators:

Unary:

- `~x`: Bitwise complement of `x` ($0 \rightarrow 1$, $1 \rightarrow 0$)

Binary:

- `x | y`: Bitwise OR of `x` and `y`
- `x & y`: Bitwise AND of `x` and `y`
- `x ^ y`: Bitwise XOR of `x` and `y`
- `x << y`: Left shift `x` by `y` bits
- `x >> y`: Right shift `x` by `y` bits

Bit versus Logical Operators

Do not confuse the bit and logical operators!

Some of them work correctly for integers; *e.g.*, `|`.

Some decidedly do not, *e.g.*, `&`:
`1 & 2` → logical false!

Not (`~`) and and (`&`) are particularly pernicious because they often work.

Masking

Many bitwise operations are used to work on a portion of a word.

This typically requires masking either:

- The bits to be modified
- The bits to be ignored

Masking uses `&` and sometimes `~`.

For example, to get the lowest 8 bits of an integer:

```
eightbits = x & 0xff;
```

(You might remember this from `dumpmem()`.)

Bit Twiddling

Setting and unsetting individual bits typically uses masking.

Assume we want to set bit zero:

```
#define LOWBIT 0x1

x = x | LOWBIT;
```

Later, we want to unset bit zero:

```
x = x & ~LOWBIT;
```

In this case, `~LOWBIT` is a mask for all bits except 0.

Twiddling with XOR

If you always want to complement a bit, you can use XOR.

This comes from the truth table; assume y is a constant 1:

x	y	Result
0	0	0
1	0	1
0	1	1
1	1	0

```
x = x ^ LOWBIT;
```

Shifting and Powers of 2

Note that bit shifting is multiplying by powers of 2!

A one-bit shift is multiplying by 2:

$0010 \rightarrow 2$
$0100 \rightarrow 4$

$0011 \rightarrow 3$
$0110 \rightarrow 6$

Successive bit shifts continue to multiply by 2.

1 (= 2^0)
1 << k (= 2^k)

Forcing Endianness

```
int htonl(int input) {
    int output;
    char *outb = (char *)&output;
    for (int b = 0; b < sizeof(int); b++) {
        int shift = (sizeof(int) - b - 1) * 8;
        outb[b] = (input >> shift) & 0xff;
    }
    return output;
}
```

htonl in Action

```
int x = 0x01020304;
int y = htonl(x);

dump_mem(&x, sizeof(x));
dump_mem(&y, sizeof(y));

04 03 02 01
01 02 03 04
```

Summary

- C can manipulate individual bits in memory.
- Bit operations can be subtle and tricky!
- Signedness matters.
- Bit manipulations can force endianness or other representations.

References I

Required Readings

[1] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 2, 2.1.6–2.1.9. Pearson, 2016.

[2] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 2, 2.9, Appendix A, A7.4.6, A7.8, A7.11–A7.13. Prentice Hall, 1988.

License

To retrieve a copy of this material, or related materials, see `https://www.cse.buffalo.edu/~eblanton/`.