Bitwise Operations

P. Danziger

1 Number Bases

1.1 Binary

Computer memory consists of a series of bits, which may on or off, 1 or 0 respectively.
A number is represented in a computer by a series of bits - giving the number in binary (base 2).
Each digit tells us whether the relevant power of 2 is present. See p. 58-60 of Epp for a discussion of binary notation.

Example 1

1. $165 = 1 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 = 10101001$ (bin)
2. $108 = 0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 = 01101100$ (bin).
3. $217 = 1 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 = 11011001$ (bin).

In a computer bits are usually grouped together. Here are some standard sizes:

Name	# of bits	# of bytes	Range in Dec.	Range in Hex.
nibble	4	$\frac{1}{2}$	0 - 15	0 - 0xF
byte	8	1	0 - 255	0 - 0xFF
word	16	2	0 - 65,535	0 - 0xFFFF
double word	32	4	0 - 4,294,967,295	0 - 0xFFFFFFFF

1.2 Hexadecimal

It is inconvenient to attempt to work in binary, so we often use the compromise of Hexadecimal.
Hexadecimal is the term used to describe base 16 numbers. We use the digits 0-9 as usual, and the letters A - F for the numbers 10 - 15.

Hex	A	B	C	D	E	F
Decimal	10	11	12	13	14	15

In C and Java we identify a hexadecimal number by prefixing it with 0x.
Other languages sometimes append h or H to indicate a hexadecimal number.

Example 2

1. 0xA5 $= 10 \times 16 + 5 = 165 =$ 10101001 (bin).
2. 0x6C $= 6 \times 16 + 12 = 108 =$ 01101100 (bin).
3. 0xD9 $= 13 \times 16 + 9 = 217 =$ 11011001 (bin).

Note that each hex digit is a nibble (4 bits).
0xD = 1101 (bin) and 0xD0 = 11010000 (bin).
Thus there is a nice correspondence between hexadecimal and binary.
You should learn to *think* in Hex!

2 Bitwise Operations

When working with bytes we may preform the *bitwise* operations corresponding to each of the basic logical operations AND, OR & XOR.
To do this we preform the logical operation on each pair of bits from the two inputs from the same position. Taking 0 as False and 1 as True.

AND

$$108 \,\&\, 217 = \text{0x6C} \,\&\, \text{0xD9} = \begin{array}{r} 01101100 \\ \&\;\; 11011001 \\ \hline 01001000 \end{array}$$

01001000 = 72 = 0x48

Note 0x6 & 0xD = 0x4 and 0xC & 0x9 = 0x8

OR

$$108 \mid 217 = \text{0x6C} \mid \text{0xD9} = \begin{array}{r} 01101100 \\ \mid\;\; 11011001 \\ \hline 11111101 \end{array}$$

11111101 = 253 = 0xFD

Note 0x6 | 0xD = 0xF and 0xC | 0x9 = 0xD

XOR

$$108 \,\hat{}\, 217 = \text{0x6C} \,\hat{}\, \text{0xD9} = \begin{array}{r} 01101100 \\ \hat{}\;\; 11011001 \\ \hline 10110101 \end{array}$$

01101100 = 181 = 0xB5

Note 0x6 ^ 0xD = 0xB and 0xC ^ 0x9 = 0x5

1's Complement

The bitwise operation corresponding to logical NOT is called 1's complement.
Unlike &, | and ^, 1's complement is a unary operator (takes only one input).

$$\sim 108 = \sim 0\text{x}6\text{C} = \begin{array}{cc} \sim & 01101100 \\ \hline & 10010011 \end{array}$$

$$10010011 = 147 = 0\text{x}93$$

$$\text{Note } \sim 0\text{x}6 = 0\text{x}9 \text{ and } \sim 0\text{xC} = 0\text{x}3$$

Most Computer Languages take 0 to be false, and any non zero value to be true.

Generally if a C compiler is required to return a true value it will default to 1. Java has a Boolean data type.

Note that there is a difference between bitwise *and, or, xor* and *1's complement* and the corresponding logical operations.

In order to highlight this difference languages like C and Java use different symbols for bitwise versus logical operations.

Operation	Bitwise	Logical
AND	&	&&
OR	|	||
1's comp./NOT	~	!

Note that according to the official C and Java specification, if the first part of a logical AND (&&) operation is false the second part is not guaranteed to be executed. Preforming '&' on a Java Boolean type guarantees that both will be executed.

Example 3

1. 0x6C & 0xD9 = 0x48, BUT
   0x6C && 0xD9 = 1
   ($T \wedge T = T$).

2. 0x6C | 0xD9 = 0xFD, BUT
   0x6C || 0xD9 = 1
   ($T \vee T = T$).

3. ~ 0x6C = 0x93, BUT
   !0x6C = 0
   (NOT T = F).

Finally note that in C and Java there is a difference between '=' and '=='.
The first '=' is the assignment operator, whereas the second '==' is a logical operator.

$a = 2$; means set the value of the variable a to 2.

$a == 2$; means test whether the variable a is 2, return true if it is, and false otherwise.

3 Masking

Bitwise operations are particularly useful for **masking**. In this case each bit in a byte represents a value which may be either *on* or *off*, i.e. true or false. In this case we wish to be able to access the bits individually, to turn each bit on or off as desired.

This is particularly common when accessing hardware devices at a low level.

We can turn a bit on by a bitwise OR (|) with 1 in the relevant position.

We can test whether a bit is set by a bitwise AND (&) with 1 in the relevant position.

We can turn a bit off by a bitwise AND (&) with NOT (∼) 1 in the relevant position.

We can toggle a bit by a bitwise XOR (^) with 1 in the relevant position.

Example 4

Suppose we have a variable *byte*, in each case initially
byte = 11001001,
and we wish to manipulate the third bit:

1. *byte* | 00000100 (bin) turns the third bit on,
   byte = 11001101 (bin).
2. *test* = *byte* & 00000100 (bin),
   test will be non zero (true) if the third bit was on in *byte*, otherwise it will be 0 (false).
3. *byte* & (∼ 00000100 (bin)) turns the third bit off, ∼ 00000100 = 11111011 (bin),
   byte & 11111011 = 11001001 (bin).
4. *byte* ^ 00000100 (bin) toggles the third bit
   byte = 11001101 (bin).

Here are the mask values in hex corresponding to the bits in a byte.

Bit	Mask Value (Hex)
7	0x80
6	0x40
5	0x20
4	0x10
3	0x8
2	0x4
1	0x2
0	0x1

Note that it is standard to count the first bit as bit 0.

If we want to mask two bits we add the corresponding values.

So to mask bits 1, 2, 5 and 7, we mask with 0xA6.

Example 5

The printer on many PCs is connected through the Parallel Port. This port has a (configurable) address of 0x378. This means that data is written and read at this address.

The next byte, 0x379, contains the parallel port status register. Reading this byte gives the current printer status. The bits of the status byte have the following meanings

Bit	Printer Status
7	Busy
6	Acknowledge
5	Out of paper
4	Selected
3	I/O error

So if an application wishes to test if the printer is out of paper it will read the status byte into the variable StatusByte and test if bit 4 is set:

```
inb(StatusByte, 0x379);    // Read Parport Status byte
if(StatusByte & 0x10) ...  // Check if bit 4 is set
```

On the other hand a check for a busy printer:

```
read(StatusByte, 0x379);   // Read Parport Status byte
if(StatusByte & 0x80) ...  // Check if bit 7 is set
```

Example 6

The BIOS (Basic Input Output Services) controls low level I/O on a computer. When a computer first starts up the system BIOS creates a data area starting at memory address 0x400 for its own use.

Address 0x417 is the Keyboard shift flags register, the bits of this byte have the following meanings:

Bit	Value	Meaning
7	0/1	Insert off/on
6	0/1	CapsLock off/on
5	0/1	NumLock off/on
4	0/1	ScrollLock off/on
3	0/1	Alt key up/down
2	0/1	Control key up/down
1	0/1	Left shift key up/down
0	0/1	Right shift key up/down

This byte can be written as well as read. Thus we may change the status of the CapsLock, NumLock and ScrollLock LEDs on the keyboard setting the relevant bit.

Here is a piece of assembler to turn Caps lock on:

```
mov   ax, [0x417]   // Get keyboard status
or    ax, 0x40      // Turn CapsLock on
                    // without changing the other bits.
mov   [0x417], ax   // Write back new result
```

Here is a piece of assembler to toggle the value of NumLock:

```
mov   ax, [0x417]   // Get keyboard status
xor   ax, 0x20      // Toggle Numlock bit
mov   [0x417], ax   // Write back new result
```

Here is a piece of assembler to turn Scrollock off (note $\sim$ 0x10 = 0xEF):

```
mov   ax, [0x417]   // Get keyboard status
and   ax, 0xEF      // Turn ScrollLock on
                    // without changing the other bits.
mov   [0x417], ax   // Write back new result
```

In order to see this byte in action open a Dos Box. At the dos prompt type

```
debug
```

At the debug prompt (-) type

```
d 0:417
```

the first location displayed will be the Keyboard shift flags register. To see it work press the relevant keys and do 'd 0:417' again.

Note that most modern operating systems block writing to the BIOS memory area. However, if you have access to a machine which runs vanilla DOS, Windows 3.1 or Windows 95 in DOS mode (not tested) you can actually use the DOS debug program to change the LEDs, at the debug prompt type:

`e 0:417` x

where x is the value you want to write.

So, for example if you wanted to turn on capslock and numlock x would be 0x60.

4 Exercises

1. Translate the following hexadecimal numbers to binary an decimal.
   0x0, 0x10, 0xF, 0x1F, 0xA4, 0xFF

2. Find the bitwise and, or and xor of the following:

   (a) 0xC6 with 0x35

   (b) 0x19 with 0x24

   (c) 0xD3 with 0xC7

   (d) 0x17 with 0xFF

3. Find the 1's complement of the following:
   0xC6, 0x35, 0xD3 and 0xC7.

4. In this question & is bitwise and, | is bitwise or, ^ is bitwise xor, and ! is 1's complement. a is any given two digit hexadecimal number. Explain why each of the following identities holds.

   (a) 0xFF & $a = a$. (0xFF is the identity for AND)

   (b) 0xFF | a = 0xFF. (0xFF is the absorbent for OR)

(c) 0xFF ^ a = !a

(d) 0 & $a = 0$. (0 is the absorbent for AND)

(e) 0 | $a = a$. (0 is the identity for OR)

(f) 0 ^ $a = a$. (0 is the identity for XOR)

(g) a ^ $a = 0$ (a is its own inverse under XOR)

(h) For any three two digit hexadecimal numbers a, b and c:
If a ^ $b = c$ then a ^ $c = b$.