Bitwise Operations

C provides functionality for manipulating the binary representation of integer values

Endian-ness

We said earlier that memory is essentially an array of bytes. There are two ways that a multi-byte data word could be stored in computer memory:

- Big-endian systems store the most significant byte (MSB) in the smallest address
- Little-endian systems store the least significant byte (LSB) in the smallest address

To determine the endianness of your platform:

```
#include <stdio.h>
#include <stdint.h>
int main( void ) {
        uint32_t u = 0x12345678;
        char *p = (char *)&u; // BAD, EVIL and WRONG - but also
FUN:-)
        printf( "%s endian\n", *p == 0x12 ? "big" : "little" );
        return 0;
}
```

Printing the Binary Representation

A function that will print out the binary representation of an unsigned integer:

```
void printBits( uint32_t x, int nbits ) {
        uint32_t mask = 1 << (nbits-1);
        for( int i=0; i<nbits; i++ ) {
                if( (x & mask) == 0 ) {
                        putchar('0');
```

```
                } else {
                        putchar('1');
                }
                mask = mask>>1;
        }
        putchar('\n');
}
```

Example Usage:

```
#include <stdint.h>
#include <stdio.h>

// our function printBits goes here

int main(void) {
  uint32_t i = 33;
  printBits(i,32);
  return 0;
}
```

Output

```
00000000000000000000000000100001
```

The AND operation

The `&` operation takes the bit-wise AND of two integer values:

```
int main( void ) {
  uint32_t i = 77;
  uint32_t j = 122;
  printf("i:   "); printBits(i,32);
  printf("j:   "); printBits(j,32);
  printf("i&j: "); printBits(i&j,32);
  return 0;
}
```

Output

```
i:   00000000000000000000000001001101
j:   00000000000000000000000001111010
i&j: 00000000000000000000000001001000
```

The OR operation

The `|` operation takes the bit-wise OR of two integer values:

```
int main( void ) {
  uint32_t i = 77;
  uint32_t j = 122;
  printf("i:   "); printBits(i,32);
  printf("j:   "); printBits(j,32);
  printf("i|j: "); printBits(i|j,32);
  return 0;
}
```

Output

```
i:   00000000000000000000000001001101
j:   00000000000000000000000001111010
i|j: 00000000000000000000000001111111
```

The right shift operation

The `>>` operation takes its first parameter, and moves each bit to the right by the number of places specified by its second parameter

Bits shifted off the right are discarded, and new zero-bits inserted on the left

```
int main( void ) {
  uint32_t i = 77;
  printf("i:      "); printBits(i,32);
  printf("i >> 1: "); printBits(i >> 1,32);
  printf("i >> 4: "); printBits(i >> 4,32);
  return 0;
}
```

Output

```
i:      00000000000000000000000001001101
i >> 1: 00000000000000000000000000100110
i >> 4: 00000000000000000000000000000100
```

The left shift operation

The `<<` operation takes its first parameter, and moves each bit to the left by the number of places specified by its second parameter

Bits shifted off the left are discarded, and new zero-bits inserted on the right

```
int main( void ) {
  uint32_t i = 77;
  printf("i:      "); printBits(i,32);
  printf("i << 1: "); printBits(i << 1,32);
  printf("i << 3: "); printBits(i << 3,32);
  return 0;
}
```

Output

```
i:      00000000000000000000000001001101
i << 1: 00000000000000000000000010011010
i << 3: 00000000000000000000001001101000
```

The NOT operation

The `~` operation inverts the bits of its operand:

```
int main( void ) {
  uint32_t i = 77;
  printf("i:  "); printBits(i,32);
  printf("~i: "); printBits(~i,32);
  return 0;
}
```

Output

```
i:  00000000000000000000000001001101
~i: 11111111111111111111111110110010
```

The XOR operation

The `^` operation takes the exclusive-OR of its operands:

```
int main( void ) {
  uint32_t i = 77;
  uint32_t j = 122;
  printf("i:   "); printBits(i,32);
  printf("j:   "); printBits(j,32);
  printf("i^j: "); printBits(i^j,32);
  return 0;
}
```

Output

```
i:   00000000000000000000000001001101
j:   00000000000000000000000001111010
i^j: 00000000000000000000000000110111
```

How our `printBits` function works

- We want to print the most significant bit (MSB) of our 32-bit integer first.
- So we create a mask that has a 1 at bit `nbits-1`, and 0 elsewhere by shifting the value 1 left by `nbits-1` places.

```
uint32_t mask = 1 << (nbits-1);
```

- For $nbits = 32$ this gives us the mask:

```
10000000000000000000000000000000
```

- We then make a loop repeat `nbits` times - once for every bit:

```
for( int i=0; i<nbits; i++ ) {
  // process one bit - discussed on next slide
}
```

- Inside the loop, we take the bitwise AND between our mask and our value x. If it is zero, we know the bit was clear, otherwise it was set:

```
if((x & mask) == 0)
  putchar('0');
else
  putchar('1');
```

- Alternatively, we could have written:

```
putchar( '0' + ((x & mask) != 0) );
```

- Lastly, we shift the mask down one bit:

```
mask = mask>>1;
```

- As with most C numeric operators, we could have simply used the in-place version:

```
mask >>= 1;
```

- So we can write a slightly more refined `printBits` as follows:

```
void printBits( uint32_t x, int nbits ) {
  uint32_t mask = 1 << (nbits-1);
  for( int i=0; i<nbits; i++ ) {
    int bit = (x & mask) != 0;
    putchar( '0' + bit );
    mask >>= 1;
  }
  putchar('\n');
}
```

An Example Usage:

- We consider the initial state:

```
x:    01000000000000000000000000000001
mask: 10000000000000000000000000000000
```

- We print a `0` since the AND of `x` and `mask` is 0.

```
x & mask: 00000000000000000000000000000000
```

- After we have performed the right shift of `mask`:

```
x:        01000000000000000000000000000001
mask    : 01000000000000000000000000000000
x & mask: 01000000000000000000000000000000
```

- Since `x & mask` is non-zero, we print `1`.

Hexadecimal and Octal Values

C doesn’t support binary literals, but it does support hexadecimal literals, which can be used to construct bit-masks, as each hex digit represents 4 bits

Hexadecimal literals in C are simply the characters “0x” followed by a hexadecimal constant. For example:

```
#include <stdio.h>
int main(void) {
        int i = 0x12ff;
        printf("decimal = %d, hex = %x\n", i, i);
        return 0;
}
```

Octal literals in C are much less used, but comprise any numeric literal starting with a “0” followed by a sequence of the digits 0-7, such as `int i = 03177;`

Suppose we wish to extract the red bits from the following 16-bit integer value:

```
0111111000010011
```

First, we shift the value 4 bits to the right using `>> 4`:

```
0000011111100001
```

Then, we create a mask that matches only the low eight bits (hex value `0xff`):

```
0000000011111111
```

We use the `&` operation to extract the required bits:

```
0000000011100001
```

```
int main( void ) {
  uint16_t i = 0x7e13;
  printf( "%4x  ", i ); printBits(i,16);
  i >>= 4;
  printf( "%4x  ", i ); printBits(i,16);
  uint16_t mask = 0xff;
  printf( "%4x  ", mask ); printBits(mask,16);
  i &= mask;
  printf( "%4x  ", i ); printBits(i,16);
  return 0;
}
```

Output

```
7e13  0111111000010011
 7e1  0000011111100001
  ff  0000000011111111
  e1  0000000011100001
```