Bitwise Operations

C includes operators that permit working with the bit-level representation of a value.

You can:

- shift the bits of a value to the left or the right
- complement the bits of a value
- combine the corresponding bits of two values using logical AND
- combine the corresponding bits of two values using logical OR
- combine the corresponding bits of two values using logical XOR

When talking about bit representations, we normally label the bits with subscripts, starting at zero, from low-order to high-order:

$$b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$$

Endian-ness

When a multi-byte value, like an `int32_t`, is stored in memory, there are options for deciding how to organize the bytes physically:

89349210_{10} 0000 0101 0101 0011 0101 1100 0101 1010

Above, we organize the bytes from left-to-right, with the byte corresponding to the highest powers of two on the left and the byte corresponding to the lowest powers of two on the right.

But in memory there's no left or right.

Instead, each byte is stored at a specific address in memory, and so the `int32_t` value will occupy four consecutive addresses.

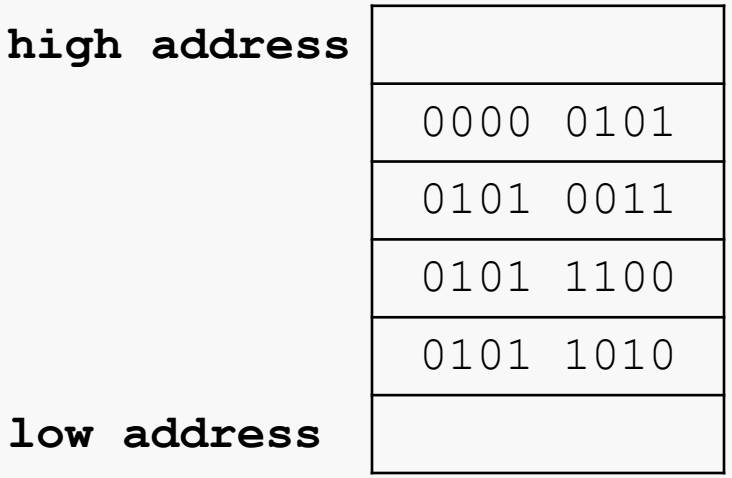
So, do we put the high-order byte at the low address? or at the high address:? or…?

Endian-ness

89349210_{10} 0000 0101 0101 0011 0101 1100 0101 1010

On *little-endian* systems, the high-order byte is stored at the high address (and the low-order byte is stored at the low address):

Note that the bits within a byte are always stored in little-endian order, high-order bit first.



On *big-endian* systems, the high-order byte is stored at the low address (and the low-order byte is stored at the high address).

x86 systems generally use little-endian byte ordering.

The JVM generally uses big-endian byte ordering.

Endian-ness

In most situations, you don't need to consider the byte-ordering used on your system.

The compiler and other tools are system-specific and will adjust for the correct ordering.

But, if you view a memory dump, like a hex dump of a binary file, or if you examine the contents of memory via pointers, you must be aware of the particular byte-ordering that's used on your system.

And, if you transfer some binary files created on a system using one byte-ordering to a system using the opposite byte-ordering, you will have to compensate for that.

Bitwise Shifts

You can shift the bits of a value to the left or the right by using the shift operators >> and <<.

Assuming the right operand is non-negative and no larger than the bit-width of the integer-valued left operand:

`EL << ER`

The bits of `EL` are shifted `ER` positions to the left;
zeros fill the vacated positions on the right;
the resulting value is returned.

`EL >> ER`

If `EL` is unsigned, or signed and non-negative, returns the value of the integer $EL / 2^{ER}$;
if `EL` is signed and negative, the result is implementation-dependent.

Bitwise Shifts with gcc

Suppose that we have the following variables:

```
int32_t X = 24061; // 00000000 00000000 01011101 11111101
int32_t Y =   -39; // 11111111 11111111 11111111 11011001
```

A little experimentation with `gcc` verifies that:

```
X <<  5  -->  00000000 00001011 10111111 10100000

X >>  5  -->  00000000 00000000 00000010 11101111

Y << 10  -->  11111111 11111111 01100100 00000000

Y >>  4  -->  11111111 11111111 11111111 11111101
```

So, `gcc` apparently maps a right shift for `int32_t` to an arithmetic right shift.

QTP: what would gcc do with a uint32_t?

Shifting and Arithmetic

Suppose again that we have the following variables:

```
int32_t X = 24061; // 00000000 00000000 01011101 11111101
int32_t Y =   -39; // 11111111 11111111 11111111 11011001
```

A little experimentation with `gcc` verifies that:

```
X <<  5  -->  769952 == 24061 * 2^5    OK

X >>  5  -->     751 == 24061 / 2^5    OK

Y << 10  --> -39936 == -39 * 2^10      OK

Y >>  4  -->      -3 == -39 / 2^4      ?
```

Bitwise Complement

Logical complement (logical negation) is defined by the following table:

```
X    ~X
------
0     1
1     0
------
```

In C, the bitwise complement (negation) operation is represented by `~`.

Again, this operator is normally applied to multi-bit operands of Standard C types.

Bitwise AND and OR

Logical AND and OR are defined by the following tables:

X	Y	X AND Y	X OR Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

In C, these bitwise operations are represented by `&` and `|`, respectively.

Normally, though, the operators are applied to multi-bit operands of Standard C types.

Bitwise XOR

Logical XOR is defined by the following table:

```
X    Y    X XOR Y
---------------
0    0       0
0    1       1
1    0       1
1    1       0
---------------
```

In C, the bitwise XOR operation is represented by `^`.

Again, this operator is normally applied to multi-bit operands of Standard C types.

Bitmasks

C programmers often create a *mask* to use with bitwise operators in order to facilitate some higher-level task:

```
bool isMultOf4(int32_t Value) {

   uint32_t Mask = 0x00000003;  // 0000 0000 . . . 0000 0011

   int32_t low2bits = Value & Mask;

   return low2Bits == 0x0;
}
```

Example: Clearing a Bit

Suppose you want to *clear* (set to 0) a single bit of a bit-sequence; say you want to clear bit b_6 of the following C `int32_t` value:

$$b_{31}b_{30}b_{29}b_{28}\cdots b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0$$

The following C code would do the trick:

```
int32_t X = 24061; // 00000000 00000000 01011101 11111101

int32_t Mask = 1 << 6;  // 0000 . . . 0000 0100 0000

Mask = ~Mask;       // 11111111 11111111 11111111 10111111

X = X & Mask;       // preserves every value in X except
                    //   for bit #6
```

QTP: **how would you set a specific bit (to 1)?**

Example: Printing the Bits of a Byte

Alas, C does not provide any format specifiers (or other feature) for displaying the bits of a value. But, we can always roll our own:

```
void printByte(FILE *fp, uint8_t Byte) {

   uint8_t Mask = 0x80;   // 1000 0000

   for (int bit = 8; bit > 0; bit--) {

      fprintf(fp, "%c", ( (Byte & Mask) == 0 ? '0' : '1') );

      if ( bit == 5 ) fprintf(fp, " ");

      Mask = Mask >> 1;   // move 1 to next bit down
   }
}
```

It would be fairly trivial to modify this to print the bits of "wider" C types.

We'll see a flexible driver for this, using pointers, on a later slide.

Example: Details

```
void printByte(FILE *fp, uint8_t Byte) {

    uint8_t Mask = 0x80;    // 1000 0000
    . . .
    Mask = Mask >> 1;    // move 1 to next bit down
    . . .
}
```

`Mask` has a 1 in position `k` and 0's elsewhere, with `k` between 7 and 0:

initially: 1000 0000
then: 0100 0000
then: 0010 0000
. . .
then: 0000 0001

Example: Details

```
void printByte(FILE *fp, uint8_t Byte) {
    . . .

       . . . (Byte & Mask) . . .;
    . . .
}
```

`Byte & Mask` == `k`-th bit of `Byte` surrounded by zeros.

Say that `Byte` was 1011 0110, then `Byte & Mask` would be:

initially: 1000 0000
then: 0000 0000
then: 0010 0000
. . .

So, `Mask` essentially plays the role of a (logical) pointer, allowing us to pick out the individual bits of `Byte` one by one.

Example: Details

```
void printByte(FILE *fp, uint8_t Byte) {
    . . .

       . . . (Byte & Mask) == 0 ? '0' : '1') . . .
    . . .
}
```

The ternary operator expression:

evaluates the Boolean expression

returns the first value after the `?` if the Boolean expression is true

returns the second value after the `?` if the Boolean expression is false

Basically, this lets us convert the 8-bit value of `Byte & Mask` to a single character.

Example: Printing the Bits with Pointers

```
void printBits(FILE *fp, const uint8_t* source,
                                        uint32_t Length) {

   // QTP:  why is pCurrByte initialized this way?

   uint8_t* pCurrByte = source + Length - 1;

   for (uint8_t byte = 0; byte < Length; byte++) {

      uint8_t currByte = *pCurrByte;

      printByte(fp, currByte);      // print bits of
                                    //     current byte

      if ( byte < Length - 1 )      // separate the bytes
         fprintf(fp, " ");

      pCurrByte--;
   }
}
```

Example: Integer Division

According to the Quotient/Remainder Theorem, given two integers x and y, where y is not zero, there are unique integers q and r such that:

$$x = q \cdot y + r$$

and

$$0 \leq r < y$$

q is called the quotient and r is called the remainder.

We all remember how to compute q and r by performing long division.

Hardware to perform integer division tends to be complex and require many machine cycles to compute a result.

For example, one source indicates that executing an integer division instruction on an Intel SandyBridge CPU may require 29 clock cycles for 32-bit operands and 92 for 64-bit operands!

Example: Integer Division

However, some special cases allow us to divide without dividing.

Suppose we want to divide an integer N by a power of 2, say 2^K.

Then, mathematically, the quotient is just N shifted K bits to the right and the remainder is just the right-most K bits of N.

So, we can obtain the quotient and remainder by applying bitwise operations:

N: 61 divisor

0000 0000 0000 0000 0000 0000 0011 1101 | 8

q: 7 *r*: 5

Example: Integer Division

But how does this really work?

```
   0000 0000 0000 0000 0000 0000 0011 1101        N
 & 1111 1111 1111 1111 1111 1111 1111 1000        mask
```

Bitwise AND applied to *N* with the right "mask" will wipe out the low bits.

Put 1's where you want to copy existing bits in *N* and 0's where you want to clear bits.

Of course, that yields this:

```
   0000 0000 0000 0000 0000 0000 0011 1000
```

We could shift this result right by 3 bits (remember we're dividing by 2^3), but it would have been just as easy (and more efficient) to just shift the original representation of N:

```
q = N >> 3;
```

Example: Integer Division

What about the remainder? Use a different mask:

```
   0000 0000 0000 0000 0000 0000 0011 1101        N
 & 0000 0000 0000 0000 0000 0000 0000 0111        mask
```

Of course, that yields this:

```
   0000 0000 0000 0000 0000 0000 0000 0101
```

So in C:

```
r = N & mask;
```

QTP: how do we form the mask if we're given the divisor, and we know it's a power of 2, but we do not know what power of 2 it is?

Hint: consider the relationship between the base-2 representations of 2^K and 2^K-1

Example: Flipping Bytes

```
uint16_t flipBytes(uint16_t N) {

   uint16_t hiByte = N & 0xFF00;    // AND with 1111 1111 0000 0000

   hiByte = hiByte >> 8;            // shift hi byte to lower half;
                                    //   high bits are now 0000 0000

   N = N << 8;                      // shift lo byte to upper half
                                    //   low bits of N are now
                                    //   0000 0000

   N = N | hiByte;                  // combine the bytes;
                                    //    high bits of N remain same
                                    //    low bits are replaced with
                                    //       the low byte of hiByte

   return N;
}
```

Example: Summing Nybbles

```
uint16_t addNybbles(uint16_t N) {

   return ( (N & 0x000F) +              // get nybble 0

            ((N & 0x00F0) >>  4) +      // get nybble 1

            ((N & 0x0F00) >>  8) +      // get nybble 2

            ((N & 0xF000) >> 12) );     // get nybble 3
}
```

```
// N = 9BC3
// N:          1001 1011 1100 0011
// low mask:   0000 0000 0000 1111   --> 000F
//             0000 0000 0000 0011
// nxt mask:   0000 0000 1111 0000   --> 00F0
//             0000 0000 1100 0000
// shift to:   0000 0000 0000 1100
// calculate:    9 + B + C + 3
```

Example: Zeroing Selected Nybbles

```
uint16_t zeroOddNybbles(uint16_t N) {

   if ( (N & 0x0001) != 0 ) {   // mask is: 0000 0000 0000 0001
      N = N & 0xFFF0;
   }

   if ( (N & 0x0010) != 0 ) {
      N = N & 0xFF0F;
   }

   if ( (N & 0x0100) != 0 ) {
      N = N & 0xF0FF;
   }

   if ( (N & 0x1000) != 0 ) {
      N = N & 0x0FFF;
   }

   return N;
}
```