Course
ELEC-A7100
Course materials
Your points
Microsoft Teams
Code Vault

This course has already ended.

« 1 Binary system Course materials 3 Bitmasks »
ELEC-A7100 / 8. Binary Operations / 2 Bitwise operators

Eighth round tasks as a `ZIP file.`

This round focuses on the bit-level manipulations, which is often necessary for programming, for example, low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization

Bitwise operators¶

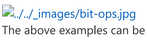
The individual bits of a byte can be manipulated through bitwise operators. The following four bitwise operators are available in C:

- bitwise **AND (&)**: `A & B` is 1 if both bit A and bit B are 1. If either of them is 0, A & B is 0 as well.
- bitwise **OR (|)**. `A | B` is 1 if either bit A or bit B is 1. If both of them are 0, A | B is 0.
- bitwise **exclusive OR (^)**. `A ^ B` is 1 if the state of bit A is different from state of bit B. If both A and B are 1 or 0, then A ^ B is 0.
- one's complement ("**NOT**"): `~A` converts bit 1 to 0, and vice versa.

A	B	(A & B)	(A | B)	(A ^ B)	~A
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

It is important to distinguish between the logical operators `(e.g., &&, || and !)` and bitwise operators. The logical operators result in integer values 0 or 1, while bitwise operators do manipulations on bit level, meaning that the resulting value can be some other integer as well, depending how the individual bits were affected by the bitwise operation. Remember that all numbers are built from combinations of bits.

The below table shows an example of these operations in two unsigned char - type values, `0x69` and `0xCA`.



The above examples can be written in C code as follows:

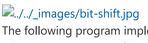
```
1  #include <stdio.h>
2
3  int main(void) {
4      unsigned char a = 0x69;  // 01101001
5      unsigned char b = 0xca;  // 11001010
6
7      printf("a & b = %02x\n", a & b);
8
9      unsigned char c = a | b;
10
11     printf("a | b = %02x\n", c);
12
13     b ^= a;  // b = b ^ a
14     printf("a ^ b = %02x\n", b);
15
16     printf("~a = %02x\n", ~a);
17     printf("~a & 0xff = %02x\n", ~a & 0xff);
18 }
```

See what the above program will print. Try program with different `a` and `b` values.

The outcome of `~a` may be surprising, and relates to integral promotion: C internally converts small integer data types (such as char or short) into the underlying architecture's "native" integer type before arithmetic operations, for optimized performance. In many cases this is not visible to the programmer because the high-end bits tend to remain zero, but here the bitwise negation turns the bits to 1, which reveals the actual data length on printout. However, we can use bitwise AND to reset the higher-order bits, as done on line 17.

In addition to the logical operators, bitwise shift operators can be applied to an expression. The `<<` operator shifts the bits left a given number of steps. For example `A << 1` shifts the bits left by one step, and A << 4 shifts the bits left by four steps. Similarly, A >> 1 shifts the bits right by one step, and `A >> 4` shifts the bits right by fours steps. When bits are shifted right by N positions, the rightmost `N` bits are lost, and the leftmost N bits will be set to 0. Correspondingly, when bits are shifted left by N positions, the N leftmost bits are lost, and `N` rightmost bits become 0. Below table illustrates a few bit shift operations.

The picture below illustrates how these operations work:



The following program implements function `printBits` that uses the bit shift operations, together with logical `AND` to output a binary presentation of the given unsigned char value, and tests its use with a couple of values shown above.

```
1  #include <stdio.h>
2
3  void printBits(unsigned char value)
4  {
5      for (int i = 7; i >= 0; i--) {
6      // (1 << i) generates value where only i'th bit is set
7      // value & (1 << i)) is non-zero only if i'th bit is set in value
8      if (value & (1 << i))
9          printf("1");
10     else
11         printf("0");
12     }
13 }
14
15 int main(void) {
16     unsigned char a = 0x69;
17     printf("0x69 = ");
18     printBits(a);
19     printf("\n0x69 << 2 = ");
20     printBits(a << 2);
21     printf("\n");
22 }
```

Task 8.2: Bit manipulation I¶

Implement **sixBits** function which returns only least significant six bits of the input value **v**. Rest of the two highest significant bits must be reset. (i.e) If the input value **v** is **11110000**, then **sixBits** function must return **110000**. One way to accomplish this is to use a bit mask.

Points **10 / 10** My submissions **1** Deadline Friday, 30 July 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Bit manipulation I

Select your files for grading

main.c

Choose File No file chosen

Submit

Task 8.3 Bit manipulation II¶

Implement **mergeBits** function which takes two 4 bit number as input values and returns a 8 bit value. This 8 bit value is formed from the two input values **a** and **b**. The value **a** forms the 4 higher order bits of the return value whereas **b** forms 4 lower order bits of the return value. For example, when a function is called as follows: `mergeBits (0x6, 0xD)`, then result will be eight-bit number, which is the hexadecimal representation `6D`.

Points **10 / 10** My submissions **1** Deadline Friday, 30 July 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Bit manipulation II

Select your files for grading

main.c

Choose File No file chosen

Submit

« 1 Binary system Course materials 3 Bitmasks »