

Programming For Engineers

C Bitwise Operations

by

Wan Azhar Wan Yusoff¹, Ahmad Fakhri Ab. Nasir²
Faculty of Manufacturing Engineering
wazhar@ump.edu.my¹, afakhri@ump.edu.my²



0.0 Chapter's Information

- Expected Outcomes
 - To organize C source, header and library files.
- Contents
 - 1.0 Introduction
 - 2.0 Complementary Bit Operator
 - 3.0 Right and Left Shift Bit Operator
 - 4.0 AND Bit Operator
 - 5.0 OR and XOR Bit Operator
 - 6.0 Logical Operator
 - 7.0 Summary



1.0 Introduction

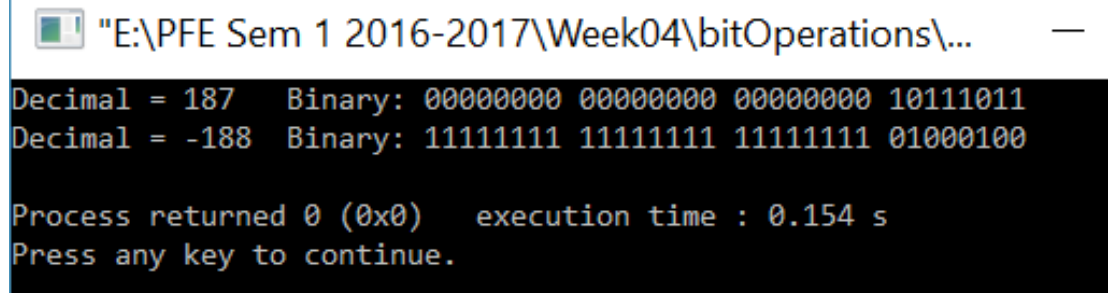
- We are going to manipulate bits of a variable. Knowing how to manipulate bits will increase our ability to program. We will learn bit operations while displaying the bits using our simple program to display a decimal number in binary format.



2.0 Bit Complimentary

- Bit complimentary means changing the bit either from 0 to 1 or from 1 to 0. In other words, complimentary operator reverses the bit. The operator for complimentary is ~ (tilde).
- In the example below, we have a number $a = 187$ and show the bits. Its complimentary is $\sim a$, and we show that $\sim a$ bits are opposite to a . This is called one complementary.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a;
    a = 187;
    bitDisplay(a);
    bitDisplay(~a);
    return 0;
}
```



```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 187   Binary: 00000000 00000000 00000000 10111011
Decimal = -188  Binary: 11111111 11111111 11111111 01000100

Process returned 0 (0x0)   execution time : 0.154 s
Press any key to continue.
```



2.0 Bit Complimentary

- In one complementary, we notice that a is 187 and $\sim a$ is -188. But if we add $a + (\sim a)$ we will not get zero as we expected. This is shown in the program below. The answer to $a + (\sim a) = -1$. We can conclude that $\sim a$ is not truly the negative of a .

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a;
    a= 187;
    bitDisplay(a);
    bitDisplay(~a);
    bitDisplay(a+(~a));
    return 0;
}
```

```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 187   Binary: 00000000 00000000 00000000 10111011
Decimal = -188  Binary: 11111111 11111111 11111111 01000100
Decimal = -1    Binary: 11111111 11111111 11111111 11111111

Process returned 0 (0x0)   execution time : 0.170 s
Press any key to continue.
```



2.0 Bit Complimentary

- To get negative of a, we have to use two complimentary. The rule for two complimentary is:
 1. Reverse all bits (complimentary one)
 2. Add 1
- Example below shows the two complimentary.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a, b;
    a = 187;
    b = ~a; //one complimentary
    b = b+1; //two complimentary
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a+b);
    return 0;
}
```

```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 187   Binary: 00000000 00000000 00000000 10111011
Decimal = -187  Binary: 11111111 11111111 11111111 01000101
Decimal = 0     Binary: 00000000 00000000 00000000 00000000

Process returned 0 (0x0)   execution time : 0.147 s
Press any key to continue.
```



2.0 Bit Complimentary

- Using `~` (tilde) bit operator, we can create a negative number. But, we must use two-complementary in order to get the true negative of a number.



3.0 Right and Left Shift Bit Operator

- We use bit shift operators to shift the bit to the left or to the right. We use << operator to shift the bits to the left and >> operator to shift the bits to the right. An example below shows the idea.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a;
    a = 128;
    bitDisplay(a);
    bitDisplay(a>>1);
    bitDisplay(a<<1);
    return 0;
}
```

```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 128   Binary: 00000000 00000000 00000000 10000000
Decimal = 64    Binary: 00000000 00000000 00000000 01000000
Decimal = 256   Binary: 00000000 00000000 00000001 00000000

Process returned 0 (0x0)   execution time : 0.132 s
Press any key to continue.
```



3.0 Right and Left Shift Bit Operator

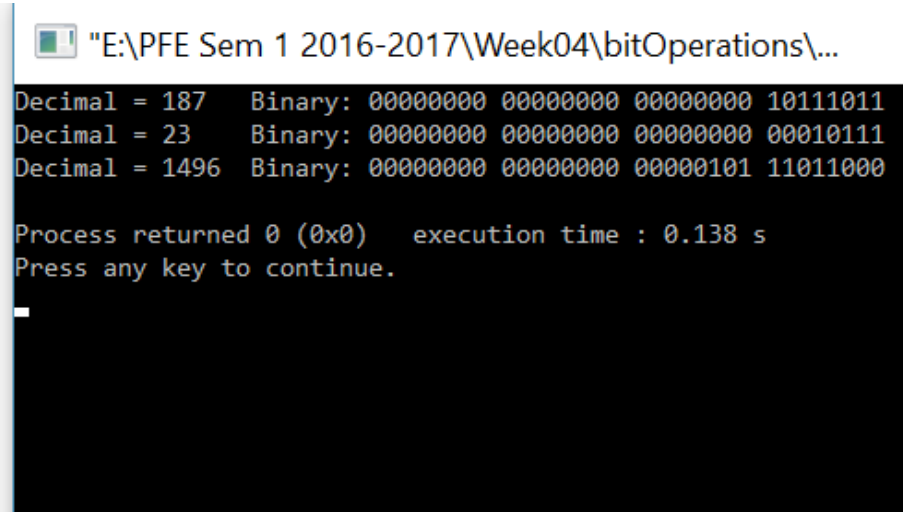
- In the previous example, we let $a = 128$. We notice that when we shift right 1 position, the bit shift to the right and the left most bit is replaced by zero. The value after the right shift is half of the original number. Notice also, when we shift to the left, we double the number. The number is shifted to the left 1 position and 0 bit is replace to the rightmost bit.
- Consider another example shown on the next page. Now, we shift 3 positions. In the first case, we shift to the right 3 positions and the 3 bits on the right disappear. Then three 0-bits on the left is added. This is called zero-padding. Notice the number is divided by 8 because each shift is divide-by-2. Three shift to the right then means divide-by-2, divide-by-2 and divide-by-2 which is divide-by-8.
- Notice also that when the computer divide the number it will retain the integer number. For example $187/8 = 23.375$. Then the number is 23.



3.0 Right and Left Shift Bit Operator

- Similarly, when we shift 3 positions to the left, we shift the bits to the left 3 times and the rightmost bits are replaced by 0-bits. Shifting to the left also is double the number. In this case we double the number 3 times i.e. $2 \times 2 \times 2 = 8$ times. The number $187 \times 8 = 1496$ is indeed the number we obtain.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a;
    a = 187;
    bitDisplay(a);
    bitDisplay(a>>3);
    bitDisplay(a<<3);
    return 0;
}
```



```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 187 Binary: 00000000 00000000 00000000 10111011
Decimal = 23 Binary: 00000000 00000000 00000000 00010111
Decimal = 1496 Binary: 00000000 00000000 00000101 11011000

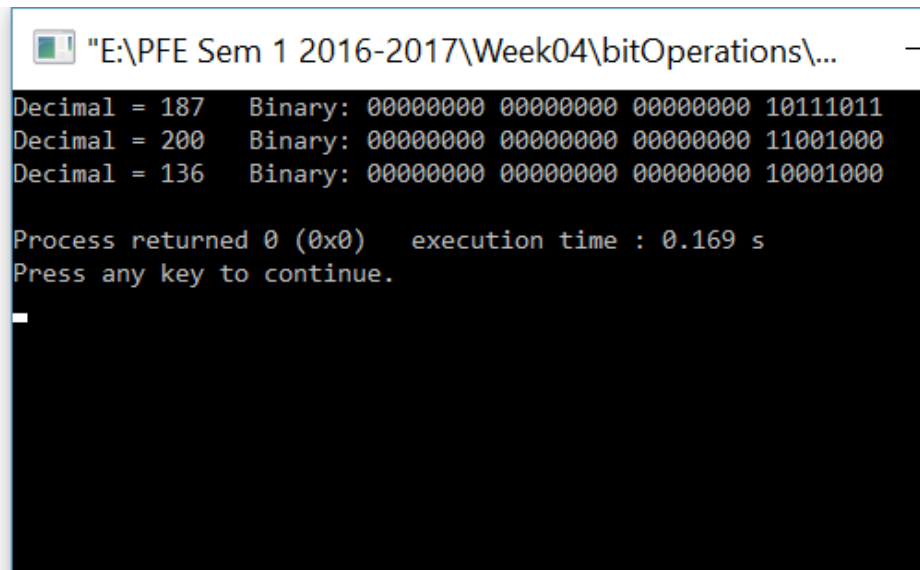
Process returned 0 (0x0) execution time : 0.138 s
Press any key to continue.
```



4.0 AND Bit Operator

- We can compare two numbers bit by bit. We AND a number by comparing each bit position and will produce 1-bit if both bits are 1-bit. Any other combination will produce 0-bit. The AND bit operator is & (ampersand). An example below will clarify the idea.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a, b;
    a = 187;
    b = 200;
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a&b);
    return 0;
}
```



```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 187   Binary: 00000000 00000000 00000000 10111011
Decimal = 200   Binary: 00000000 00000000 00000000 11001000
Decimal = 136   Binary: 00000000 00000000 00000000 10001000

Process returned 0 (0x0)   execution time : 0.169 s
Press any key to continue.
```



4.0 AND Bit Operator

- In the previous example, we have two numbers $a = 187$ and $b = 200$. We then AND these two numbers bit by bit. Notice that only combination of 1-bit will result in 1-bit. Else, the AND operator will give 0-bit. The truth table is given below.

AND Truth Table

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1



4.0 AND Bit Operator

- One application for AND operator is masking. Masking is making all bits become 0 if we AND a number with 0. For example, if we want to know a number is even or odd, we can know this by knowing the rightmost bit number. If the rightmost bit is 0, the number is even. If the rightmost number is 1, it is an odd number. We show an example below.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a, b;
    a = 187;
    b = 1;
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a&b);
    return 0;
}
```

```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 187   Binary: 00000000 00000000 00000000 10111011
Decimal = 1     Binary: 00000000 00000000 00000000 00000001
Decimal = 1     Binary: 00000000 00000000 00000000 00000001

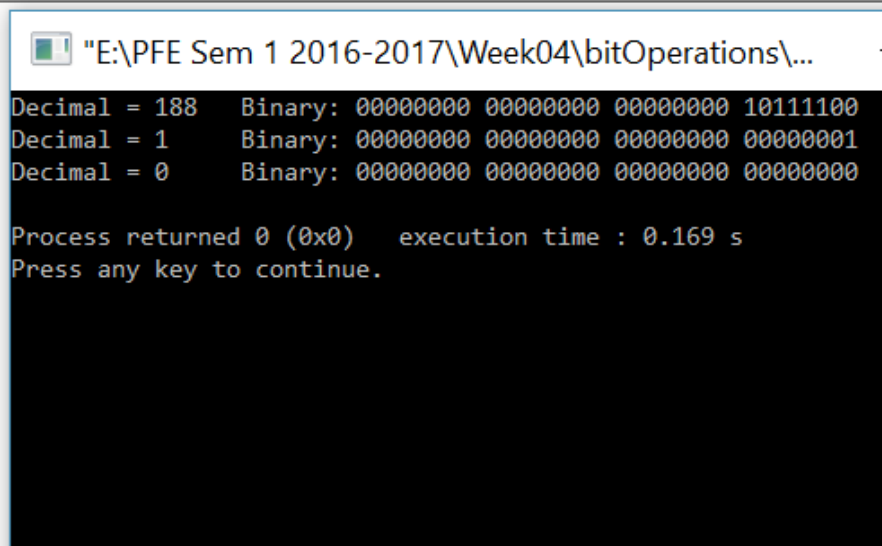
Process returned 0 (0x0)   execution time : 0.148 s
Press any key to continue.
```



4.0 AND Bit Operator

- In the previous above we AND number 187 with number 1. Since we “mask” all number except the first bit (because we AND with number 1 which all zeros except the first bit), we can check the first bit. If the first bit is 0 than the number is even. If the number is 1 then the number is odd. The even example is given below.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a, b;
    a = 188;
    b = 1;
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a&b);
    return 0;
}
```



```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 188   Binary: 00000000 00000000 00000000 10111100
Decimal = 1    Binary: 00000000 00000000 00000000 00000001
Decimal = 0    Binary: 00000000 00000000 00000000 00000000

Process returned 0 (0x0)   execution time : 0.169 s
Press any key to continue.
```



4.0 AND Bit Operator

- So, by masking we can know the number is odd or even. Similarly, we can determine whether the number is positive or negative. If the leftmost number is zero, it is a positive number. If it is negative it is a negative number. Let us check a few number using masking.
- Look at the program on the next page. We have a negative number -188. Because this is a negative number, the leftmost bit is 1-bit. We need to AND with 10000000 00000000 00000000 00000000. How to create this number? What we did is creating a number $b = 1$ and shift left 31 positions i.e. $b = b \ll 31$. You can see the result at the output line 2 which is -2147483648.



4.0 AND Bit Operator

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a, b;
    a = -188;
    b = 1;
    b = b << 31;
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a&b);
    return 0;
}
```

```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = -188 Binary: 11111111 11111111 11111111 01000100
Decimal = -2147483648 Binary: 10000000 00000000 00000000 00000000
Decimal = -2147483648 Binary: 10000000 00000000 00000000 00000000
Process returned 0 (0x0) execution time : 0.138 s
Press any key to continue.
```

- Then, after we AND number a and number b, we can see that there is bit-1 at the leftmost bit. That tells us that the number is negative.



5.0 OR and XOR Bit Operator

- XOR is the operator to know whether the two compared bits is equal or not equal. XOR is called exclusive or. If the bits are equal, the result is bit-0 and if the bits are not equal the results is bit-1. The operator for XOR is \wedge (caret character).
- OR on the other hand will produce bit-1 if there exists bit-1 in either of two bits. If there is no bit-1 i.e. both bits are bit-0, it will produce 0-bit. The operator for OR is $|$ (the vertical slash character).
- The truth table for XOR and OR are given below.



5.0 OR and XOR Bit Operator

XOR Truth Table

a	b	$a \wedge b$
0	0	0
0	1	1
1	0	1
1	1	0

OR Truth Table

a	b	$a b$
0	0	0
0	1	1
1	0	1
1	1	1

- In the example shown on the next page, we XOR and OR two numbers. The first two numbers is XOR and we notice that whenever the two bits are unequal 0-bit is produced. In the second case, we OR the number. We notice that if there exists bit-1, then bit-1 is produced.



5.0 OR and XOR Bit Operator

- Important difference is at bit position 4 from the leftmost. XOR will result in bit-0 (bit-1 XOR bit-1 – bit-0). However, the OR operation will give bit-1 (bit-1 OR bit-1 – bit-1). Please notice the difference.

```
#include <stdio.h>
#include <stdlib.h>
#include "bitOperations.h"
int main()
{
    int a, b;
    a = 188;
    b = 200;
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a^b); //XOR
    printf("\n");
    bitDisplay(a);
    bitDisplay(b);
    bitDisplay(a|b); //OR
    return 0;
}
```

```
"E:\PFE Sem 1 2016-2017\Week04\bitOperations\...
Decimal = 188   Binary: 00000000 00000000 00000000 10111100
Decimal = 200   Binary: 00000000 00000000 00000000 11001000
Decimal = 116   Binary: 00000000 00000000 00000000 01110100

Decimal = 188   Binary: 00000000 00000000 00000000 10111100
Decimal = 200   Binary: 00000000 00000000 00000000 11001000
Decimal = 252   Binary: 00000000 00000000 00000000 11111100

Process returned 0 (0x0)   execution time : 0.163 s
Press any key to continue.
```



6.0 Logical Operator

- Logical operator is different from the bitwise operator.
 1. Logical operator compare TRUE and FALSE and will give result either TRUE or FALSE. Any number value that is non-zero is TRUE. Only zero is false. So, number 188, -188, 200 are all TRUE. The operators are && for AND, || for OR and != for XOR.
 2. Bitwise operator compare the bit logic not the number logic. The results from bitwise operator is another number.



7.0 Summary

- In this note, we learn about bitwise operators:
 1. We learned about complementary operator (\sim) that reverses all bits. But, in order to represent flip number from negative to positive or vice versa, we use two-complement. In two-complement, we reverse the bits and we add 1.
 2. We learned the right and left shift operator. Right shift will shift the bits to the right and replace the leftmost bits with zero. The right shift will shift the bits to the left and replaces the rightmost bits to zero. Right shift halve the value for every shift while the left shift double the value for every shift.



7.0 Summary

3. We learned about logical AND. We know that using AND we can do masking. By masking, we can detect whether the number is even or odd and positive or negative. Masking allows us to check individual bits.
4. We learned about XOR and OR. Exclusive OR is to check the equality between bits. OR detects the presence of bit-1.
5. Finally, we learned that LOGICAL OPERATOR is not the same as BITWISE OPERATOR. In logical operator, we compare TRUE/FALSE. In bitwise operator, we compare individual bits of a number.

