Bitwise Operators

Dr. Neslisah Torosdagli

Bitwise Operators

- We commonly use the binary operators && and ||, which take the logical AND and logical OR of two Boolean expressions.
- Since Boolean logic can work with single bits, C provides operators that work on individual bits within a variable.
- As we learned earlier in the semester, if we store an int in binary with the value 47, its last eight binary bits are as follows:

    00101111

- Similarly, 72 in binary is 01001000.

- Bitwise operators would take each corresponding bit in the two input numbers and calculate the output of the same operation on each set of bits.
- For example, a bitwise AND is represented with a single ampersand sign: &.
- This operation is carried out by taking the AND of two bits. If both bits are one, the answer is one. Otherwise, the answer is zero.

```
  0 0 1 0 1 1 1 1
& 0 1 0 0 1 0 0 0
--------------------
  0 0 0 0 1 0 0 0 (which has a value of 8.)
```

Bitwise Operators

Thus, the following code segment has the output 8:

```
int x = 47, y = 72;
int z = x & y;
printf("%d", z);
```

Function	Operator	Meaning
and	&	1&1 = 1, rest = 0
or	|	0 | 0 = 0, rest = 1
xor	^	1 ^ 0 = 0 ^ 1 = 1 0 ^ 0 = 1 ^ 1 = 0
Not (unary operator)	~	~0 = 1, ~1 = 0

Now, let's calculate the other bitwise operations between 47 and 72:

```
   0 0 1 0 1 1 1 1
|  0 1 0 0 1 0 0 0
------------------------
   0 1 1 0 1 1 1 1 (which has a value of 111.)
```

```
   0 0 1 0 1 1 1 1
^  0 1 0 0 1 0 0 0
---------------------
   0 1 1 0 0 1 1 1 (which has a value of 103.)
```

Bit Masking

- **Masking** means to keep/change/remove a desired part of information.
- Lets see an image-masking operation; like- this masking operation is removing any thing that is not skin-

- A mask defines which bits you want to keep, and which bits you want to clear.

- Masking is the act of applying a mask to a value. This is accomplished by doing:
  - Bitwise ANDing in order to extract a subset of the bits in the value
  - Bitwise ORing in order to set a subset of the bits in the value
  - Bitwise XORing in order to toggle a subset of the bits in the value
- Lets say you got a 8 bit number: 00110010
  - You want to set (set means assigning bit 1) the least significant bit to 1
    - You could OR 00000001 to this number. So, all other bits are same, just the LSB (least significant bit, the right most bit) changed to 1.

Bit Masking

- Below is an example of extracting a subset of the bits in the value:
- Mask: 00001111b
- Value: 01010101b
- Applying the mask to the value means that we want to clear the first (higher) 4 bits, and keep the last (lower) 4 bits. Thus we have extracted the lower 4 bits. The result is:

- Mask: 00001111b
- Value: 01010101b
- Result: 00000101b

Bit Masking

- This function tells if an integer is odd/even. We can achieve the same result with more efficiency using bit-mask-

```
bool isOdd(int i) {
   return i&1;
}
```

- Short Explanation: If the least significant bit of a binary number is 1 then it is odd; for 0 it will be even. So, by doing AND with 1 we are removing all other bits except for the least significant bit i.e.:

```
      55  ->  0 0 1 1 0 1 1 1   [input]
(&)   1  ->  0 0 0 0 0 0 0 1    [mask]
- - - - - - - - - - - - - - - - - - - - - - - - - -
       1  <-  0 0 0 0 0 0 0 1  [output]
```

Two’s Complement

- Let us discuss how a signed integer is stored inside the computer
- During base conversion we learned mainly unsigned integers (these don’t allow negative values and are used less frequently than regular ints)
- Can you tell me how many numbers you can generate using 3 bits?

- Yes, $2^3 = 8$ different numbers. See the table in the right side.
- But all of them are positive numbers.
- How can we accommodate negative numbers?
- The left most bit is used for sign (0=+ and 1=-)
- In that case we loss one bit and we have remaining 2 bits to generate the actual numbers and left bit for +/-.
- Now you can generate only $2^2 = 4$ numbers in positive side and 4 numbers in negative sides
- -4, -3, -2, -1, 0, 1, 2, 3
  - Kind of -2^2 to $(2^2 -1)$

Decimal Number	3-bit Binary Number
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

But how to get the equivalent binary representation?

- It’s not just putting 1 (-) and 0 (+) to a number’s equivalent binary, because there is no negative 0
- Two’s complement is used in this process

Two's Complement

- Two's Complement is used to store regular ints.
- In computer, integers are stored as 32 bits
- For simplicity of discussion, let us continue our discussion with 3 bits
- See the right side table's binary numbers have different meaning now.
- If you know the original positive number (how would you know if it is positive? If Left most bit a.k.a most significant bit (MSB) is 0), Two's compliment can help you to know the equivalent negative number.
- Here is the steps:
  1) Flip the bits (0 to 1 and 1 to 0)
  2) Add One (binary addition)
- For example:
  - Consider 001 (which is 1)
  - Flipping it would result in 110
  - Adding 1 with 110 = 111

1 1 0

+ 1

----

1 1 1

  - **So, 111 is -1**
- **If you do not know binary addition: 0+0 = 0, 1+0 = 1, 0+ 1 = 1, 1 + 1 = 0 (with carry 1)**
- Can you calculate the Two's complement of 0? Consider 3 bits representation of 0

3-bit Binary Number	Unsigned Decimal Number	Signed Decimal Number
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Two’s Complement

- Two’s Complement of 000
  - After flipping: 111 then add 1

```
  1 1 1
+     1
-------------
1 0 0 0
```

- So, there is an overflow and we discard that.
- So, we get 000 as the result!
- Try to calculate Two’s complement of 2
- If you see the table:
  - All bit 1 means the largest/first negative integer (-1)
  - 1 at MSB with all the remaining bits to 0 is the smallest/last negative integer
    - 100 in this example which is -4

3-bit Binary Number	Unsigned Decimal Number	Signed Decimal Number
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

Two's Complement

- Another approach from Arup's (Another CS1 prof) notes:
- The storage scheme of Two's complement is almost identical to regular binary, except for the meaning of the most significant bit.
- An int is stored using 32 bits.
- For an unsigned number, each of these bits are place-holders with value 2^{31}, 2^{30}, 2^{29}, …, 2^{2}, 2^{1}, and 2^{0}.
- Essentially, if the bit at location i is 1, then we contribute 2^{i} to the value of the number. (If it's 0, nothing changes.) [It is exactly what we learned during base conversion]
- In two's complement, we only change the most significant bit to mean -2^{31}.
- For example, if all of the bits were 1 in a regular int:

  11111111 11111111 11111111 11111111
- Then, the value of this number would be:

  $-2^{31} + 2^{30} + 2^{29} + 2^{28} + \ldots + 2^{2} + 2^{1} + 2^{0} = -1$
- If you cannot calculate these large numbers, just consider 4 bits. 1111
  - If all the bits are 1, the MSB will result in $2^{3} = 8$ which is actually (1 more than the total of remaining bits $(2^{2} + 2^{1} + 2^{0} = 4 + 2 + 1 = 7)$.
  - So, $-2^{3} + 2^{2} + 2^{1} + 2^{0} = -8 + 7 = -1$
- Considering 32 bits, -5 will be represented as follows:
- 11111111 11111111 11111111 11111011

Two's Complement

- Taking into account two's complement, we can calculate the effect of the bitwise not operator.
- Consider calculating ~x, where x = 47, from our previous example.
- Here is all 32 bits of x:

```
00000000 00000000 00000000 00101111
```

- Flipping each bit, we get:

```
11111111 11111111 11111111 11010000
```

- Using a bit of logic, we can see that if we were to have all 1's the value would be -1. But, now that we've turned off the bits that add up to 47, our new value will be -48.
- In essence, we see that in the typical case, when x is positive, **~x is equal to –x-1.**
- Hopefully you can see that if x is negative, then **~x is ALSO –x-1. (Thus, ~~x always equals x, as you might imagine.)**

Left and Right Shift Operators

- The left-shift operator is <<.
- The right-shift operator is >>.
- When we left-shift a value, we must specify how many bits to left-shift it.
- What a left-shift does is move each bit in the number to the left a certain number of places.
- In essence, so long as there is no overflow, a left-shift of one bit multiplies a number by two (since each bit will be worth twice as much).
- For example consider 8 bits representation of 3:

  00000011 left shift one bit would result in 00000110 which is 6
- It follows that a left shift of 2 bits multiplies a number by 4 and a left shift of 3 bits multiples a number by 8. In general, a left-shift of k bits multiplies a number by 2^k.

Example Left Shift Right Shift in C

```
/* C Program to demonstrate use of left shift
operator */
#include<stdio.h>
int main()
{
        // a = 5(00000101), b = 9(00001001)
        unsigned char a = 5, b = 9;

        // The result is 00001010
        printf("a<<1 = %d\n", a<<1);

        // The result is 00010010
        printf("b<<1 = %d\n", b<<1);
        return 0;
}
```

```
Output:
a<<1 = 10
b<<1 = 18
```

```
/* C Program to demonstrate use of right
shift operator */
#include<stdio.h>
int main()
{
        // a = 5(00000101), b = 9(00001001)
        unsigned char a = 5, b = 9;

        // The result is 00000010
        printf("a>>1 = %d\n", a>>1);

        // The result is 00000100
        printf("b>>1 = %d\n", b>>1);
        return 0;
}
```

```
Output:
a>>1 = 2
b>>1 = 4
```

Important Points :

- The left shift and right shift operators should not be used for negative numbers. The result of is undefined behavior if any of the operands is a negative number. For example results of both -1 << 1 and 1 << -1 is undefined.
- If the number is shifted more than the size of integer, the behavior is undefined. For example, 1 << 33 is undefined if integers are stored using 32 bits.
- The left-shift by 1 and right-shift by 1 are equivalent to multiplication and division by 2 respectively.

Using bitwise operators to iterate through subsets

- **Imagine solving the following problem with brute force:**
- **Given an array of values, such as {9, 3, 4, 5, 12}, does there exist a subset of values in the array that adds up to a target, say 22?**
- **Our goal would be simply to try EACH possible subset of the array, add the values and see if we get the target.**
- **We have 5 elements in the array (index 0,1,2,3,4). If we think about binary and look at the binary values from 0 to 31, we have:**

00000	**01000**	**10000**	**11000**
00001	**01001**	**10001**	**11001**
00010	**01010**	**10010**	**11010**
00011	**01011**	**10011**	**11011**
00100	**01100**	**10100**	**11100**
00101	**01101**	**10101**	**11101**
00110	**01110**	**10110**	**11110**
00111	**01111**	**10111**	**11111**

- **If we assume that 0 means, “don’t put this number in the set” and 1 means, “put this number in the set, then these 32 listings represent all possible subsets of a set of 5 values.**

Using bitwise operators to iterate through subsets

- Thus, our idea is as follows:
- Loop from 0 to 31, for each value and calculate the sum of the corresponding subset.
- example, since 13 is 01101, this means that the subset we want to add up is array[3], array[2] and array[0].
- We are using the most significant bit in the number to correspond to the last array slot and the least significant bit in the number to correspond to index 0 in the array.
- In this example, when we are considering 13, the values we add are 9, 4 and 5 to obtain 18.

// remember our array is :{9, 3, 4, 5, 12},

Using bitwise operators to iterate through subsets

- **Notes:**
- **1) Remember that a left-shift of n bits multiplies by 2^n, so the value of 1 << n for this example is $2^5 = 32$, as desired.**
- **2) The j loop is going through each array element, trying to decide whether or not to add it. The value 1 << j has only one bit set to 1, it's the bit at location j.**
- **3) If we do a bitwise AND with a number of the form 000…001000…, then our answer will either be all 0s OR it will be the number itself. Basically, all of the 0s cancel out the other 31 bits. The one 1 isolates that particular bit, which is exactly what we want.**

Code:
```
   int i, j;
   int n = 5;
   int array[] = {9, 3, 4, 5, 12};
   int target = 22;

   for (i=0; i < (1 << n); i++) {

      int sum = 0;
      for (j=0; j < n; j++)
         if ( (i & (1 << j)) != 0 )
            sum += array[j];

      if (sum == target)
         printf("Can add up to the 
target!\n");
   }
```

- The full code is uploaded as bitwise.c code for more explanation.
- The printf of the uploaded code clarifies many things
- Some part of the code shown in the next slide

```
printf("process for iterating through
subset\n"); int i, j;
int n = 5;
int array[] = {9, 3, 4, 5, 12};
int target = 22;

for (i=0; i < (1 << n); i++) { //this loop is eqivalent of for(i=0; i<32;

i++) int sum = 0;
// loop through each location in the binary representation of
i for (j=0; j < n; j++) //n is 5 in our example
{
   printf("i=%d, j=%d, 1<<j=%d,(i & (1<<j)) = %d\n", i, j, 1 << j, (i & (1 <<
   j)));
/*1<<j produces 1 (00001), 2(00010), 4(00100), 8(01000), 16(10000)
and we use it as mask to extract which bit is one in
   i*/ if ( (i & (1 << j)) != 0 )
        sum += array[j];
}
printf("i=%d sum = %d\n", i, sum);
if (sum == target)
   printf("Can add up to the target!\n");
```

Some part of the output:

```
process for iterating through subset
 i =0, j = 0, 1 << j = 1,(i & (1 << j)) = 0
 i =0, j = 1, 1 << j = 2,(i & (1 << j)) = 0
 i =0, j = 2, 1 << j = 4,(i & (1 << j)) = 0
 i =0, j = 3, 1 << j = 8,(i & (1 << j)) = 0
 i =0, j = 4, 1 << j = 16,(i & (1 << j)) = 0
i=0 sum = 0
```

```
i =1, j = 0, 1 << j = 1,(i & (1 << j)) = 1
 i =1, j = 1, 1 << j = 2,(i & (1 << j)) = 0
 i =1, j = 2, 1 << j = 4,(i & (1 << j)) = 0
 i =1, j = 3, 1 << j = 8,(i & (1 << j)) = 0
 i =1, j = 4, 1 << j = 16,(i & (1 << j)) = 0
i=1 sum = 9
```

```
i =2, j = 0, 1 << j = 1,(i & (1 << j)) = 0
 i =2, j = 1, 1 << j = 2,(i & (1 << j)) = 2
 i =2, j = 2, 1 << j = 4,(i & (1 << j)) = 0
 i =2, j = 3, 1 << j = 8,(i & (1 << j)) = 0
 i =2, j = 4, 1 << j = 16,(i & (1 << j)) = 0
i=2 sum = 3
```

References

- Arup's notes on Bitwise Operators: http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/BitwiseOperators.doc

- Some interesting problems that use Bitwise Operators: http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/sampleprogs/BitwiseProbs/

Exercise

- Two useful utility functions when dealing with integers in their binary representation are

- (a) int lowestOneBit(int n) - returns the value of the lowest bit set to 1 in the binary representation of n.
  - (eg. lowestOneBit(12)returns 4. Because 12 is 1100, just keeping the lowest bit set to 1 it becomes: 0100 which is 4)
  - lowestOneBit(80) returns 16. Because 80 is 1010000, just keeping the lowest bit set to one, we get 0010000 which is 16)

- (b) int highestOneBit(int n) - returns the value of the highest bit set to 1 in the binary representation of n.
  - (eg. highestOneBit(12)returns 8. Because, 12 is 1100, just keeping the highest bit set to 1 it becomes: 1000 which is 8)
  - highestOneBit(80)returns 64. Because 80 is 1010000, just keeping the highest bit set to one, we get 1000000 which is 64)

- The pre-condition for the functions, n must be a positive integer.

Exercise Solution

- a)

```
int lowestOneBit(int n)
{
        int res = 1;
        while ((res & n) == 0)
            res = res << 1;
        return res;
}
```

Start mask from 1 and left shift it at each iteration. Keep applying mask and perform & operation with n. As soon as you see the masking result in in none zero value, it means you found the lowest bit set to one

-Let's analyze the code for lowestOneBit(12).
-n = 12 which is 1100 (ofcourse there are 28 more 0s before 1100 as n is a 32 bit int
-res = 0001
-We perform res & n which will result in 0 [0001 & 1100]

- Left shift res by 1: 0010
- Now, res & n is 0 [0010 & 1100]
- Left shift res by 1: 0100
- Now, res & n is 4 [0100 & 1100]
    - Break the loop and return res which is 4

Exercise Solution

- b)

```
int highestOneBit2(int n) {
    int res = 1<<30;
    // printf("%d & %d = %d\n",
    res, n, res&n);
    while ((res&n) == 0 )
        res = res >> 1;
-   return res;
}
```

Start mask by setting 1 to the 31th bit as 32th bit is the sign bit and we keep the mask as a positive int. Then right shift it at each iteration. Keep applying mask and perform & operation with n. As soon as you see the masking results in none zero value, it means you found the highest bit set to one.