THE UNIVERSITY *of* NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

CSCI 2467, Fall 2020
Class Activity: Two's complement, bitwise and logical operators
Wednesday, January 29, 2020

2467 Instructors: M. Toups, C. Boyce
staff@cs.uno.edu

1 Introduction

In this activity you will practice working with binary numbers, bitwise operators, and logical operators. This activity is based in part on material developed by Professor Saturnino Garcia of the University of San Diego and is used with permission.

2 Review of Negative Integers

1. In a two's complement system, what is the leftmost bit in a *non-negative* number? What about the leftmost bit in a *negative* number?

2. Complete the following table to indicate the *most positive* (i.e. largest) and *most negative* (i.e. smallest) number that can be represented with a given number of bits when using **two's complement representation**.

Bits	**Most positive**	**Most negative**
1	0	-1
2	1	-2
3		
4		

3. Use your answer from the previous question to find an expression that gives the *most positive* number that can be represented by a N-bit two's complement number. **Hint**: This will be related to a power of two in some way.

4. Use your answer from the previous question to find an expression that gives the *most negative* number that can be represented by a N-bit two's complement number. **Hint**: This will be related to a power of two in some way.

3 Bit-Level Operations

In the 1800s, George Boole proposed a logic-system based on two values: 0 and 1. We will work through how these operations are exposed in C, and by extension the underlying system. The first set of operations are performed by treating the integer as a bit-vector.

7. One of the steps to negating a signed integer was to invert all of the bits. This operation, complement or "~", can be applied to any integer. For each integer, convert it from hexadecimal to binary and then apply the operation. Show the binary version and then convert the answer back to hexidecimal.

   - *Example*: ~(0xF0F0) → ~(1111 0000 1111 0000) → 0000 1111 0000 1111 → 0x0F0F
   - ~(0x3C3C)
   - ~(0xCAFE)
   - ~(0x0000)

8. Fill in the following table. In the left column is a value (let's call it `z`). Convert it from decimal to binary, then use bitwise AND to show (`z & 0x1`):

Dec	**Bin**	**X &** `0x1`
-2		
-1		
0	0000	0000
1		
2		

9. For which numbers was the value & 0x1 not 0000? What is a common property of these integers?

10. Many times in systems programming, we want to test if a flag value is set in a given bit-pattern. The programmer will commonly use (X & FLAG) == FLAG. Give an explanation of this expression. When does it return true (1)? When does it return false (0)?

11. Systems programmers will also regularly have to combine flag values. Describe how OR (|) is used in the following example, which creates a new file for writing:

    ```
    open(filename, (O_WRONLY | O_CREAT | O_TRUNC), ...);
    ```

    (*Hint*: `O_WRONLY`, `O_CREAT`, and `O_TRUNC` are each flags, meaning only one bit is set to 1 in each, at a different bit position.)

12. We know that in signed integer datatypes (such as `int` and `short int` in C) the leftmost bit (or most-significant bit, MSB) is the sign bit. Let's say you want to write an expression in C which masks out all bits in `x` except the sign bit.

    Fill in the blank in this expression with the correct constant value to use as a bitmask:

    ```
    int signbit = x &  _____________________ ;
    ```

13. Now consider the situation, like in Data Lab, where you cannot use arbitrarily large constants. Only constants 0 through 255 (`0x00` through `0xff`) are allowed. Now, how would you write the code (like above) that masks out all bits in `x` except the sign bit? (Hint: utilize bit-shift operators)

    ```
    int signbit = x &  _____________________ ;
    ```

14. In both cases above (which should yield the same value), what is the value of `signbit` if `x` is positive?   If `x` is Negative?

15. Let's say you need to convert `signbit` from above into a value that is always either 0 or 1. Fill in the blank below to complete the C expression, in terms of `signbit`:

    (This one is tricky. Remember how arithmetic shifts work!)

    ```
    int isNegative =  ( signbit  >> ____ ) &  _________   ;
    ```

    Could you also write this expression in terms of `x`?

    ```
    int isNegative =  (  x >>  _____ ) &  _______  ;
    ```

    The blanks above should be filled with either binary or logical operators, or constants.

4 Logical Operations

This section will explore logical operations. These operations contrast with bit-level in that they treat the entire value as a single element. In other languages, the type of these values would be termed, "bool" or "boolean". C does not have any such type. Instead, the value of 0 is false and all other values are true.

The three operators are AND (&&), OR (||), and NOT (!). "!" is commonly termed "bang".

1. With 4-bit values, how many values are false and how many are true?

   (Keep in mind there are 16 total values when using 4 bits)

2. Evaluate the following expression: `(0x3 && 0xC) == (0x3 & 0xC)`

   Is it true (1) or false (0)?

   (*Hint*: Pay attention to the *logical* && and the *bitwise* &)

   Is this result what you expected? Why or why not?

3. Test whether !!X == X holds across different values of X.

X	**!X**	**!!X**
-1		
0		
1		
2		

4. Now, for each of the previous values, substitute complement (~) for logical not. Do these results differ from the previous results? How? Why?

X	**~X**	**~~X**
-1		
0		
1		
2		