16.216: ECE Application Programming

Summer 2012

Programming Assignment #3: Practicing Bit Manipulation

Due **Friday, 7/20/12**, 11:59:59 PM

1. Introduction

This assignment will give you practice with the C bitwise operators, as well as some basic output formatting. You will enter input values in hexadecimal, as well as the bit positions at which these values are to be modified, and then display the results of these operations in tabular form.

2. Deliverables

Submit your source file directly to Dr. Geiger (Michael_Geiger@uml.edu) as an e-mail attachment. Ensure your source file name is ***prog3_bits.c***. You should submit only the .c file. Failure to meet this specification will reduce your grade, as described in the program grading guidelines.

This assignment contains extra credit problems (described on page 3). Note that:

- No hints will be given or questions answered about the extra credit problems, either in person, through e-mail, or on the message board.
    - You should not post anything on the message board about these problems—please do not ask or answer any questions.
- You will not receive extra credit if you do not make a good faith effort to solve all required parts of the assignment.
- Late assignments are not eligible for extra credit.
- Extra credit solutions will only be evaluated in your initial submission. You can neither add new code nor improve an existing solution to an extra credit problem as part of a regrade request.

3. Specifications

Input: Your program should prompt the user to enter the following:

- Two unsigned integers (data type: `unsigned int`), in hexadecimal
    - Remember, hexadecimal values are read using format specifier `%x`.
    - Inputs will be interpreted as hexadecimal values regardless of whether you include a leading "`0x`"—`0x15` and `15` are the same.
    - You can specify up to 32 bits, using eight hexadecimal digits.
- A single bit position to be modified.
- The lowest and highest bit positions in a range of bits to be modified.
    - All three positions should satisfy the condition ($0 \leq \text{position} \leq 31$).

Input (cont.): A sample run of the program might produce the following first three lines (user inputs are underlined):

```
Enter two values in hexadecimal: 0xdeadbeef 0x12345678
Enter single bit position to be changed: 5
Enter lowest and highest bits to be changed: 0 15
```

Output: Your program should perform the following operations on the two unsigned integer inputs:

- Use the single bit position that was entered on the second input line to set, clear, and flip that single bit in both inputs.
  - "Set" → set the bit equal to 1
  - "Clear" → set the bit equal to 0
  - "Flip" → change the value of the bit from 1 to 0, or from 0 to 1
- Use the range of bits that were entered on the third input line to set, clear, and flip all bits in that range in both inputs.

You should then print those outputs using exactly the formatting shown below, including the number of spaces shown in the example output.

The first and last lines shown below are "rulers" that **should not be printed**—their purpose is to show you exactly how many characters are being printed. Each dot represents a single character, each slash indicates that 5 characters have been used, and each number indicates that 10 characters have been used. You can therefore see that each output line should use at most 55 characters.

```
..../....1..../....2..../....3..../....4..../....5..../
SINGLE BIT CHANGES
   Input    Bit  5 set   Bit  5 cleared  Bit  5 flipped
0xdeadbeef  0xdeadbeef     0xdeadbecf      0xdeadbecf
0x12345678  0x12345678     0x12345658      0x12345658

MULTI-BIT CHANGES
   Input     0-15 set     0-15 cleared    0-15 flipped
0xdeadbeef  0xdeadffff     0xdead0000      0xdead4110
0x12345678  0x1234ffff     0x12340000      0x1234a987
..../....1..../....2..../....3..../....4..../....5..../
```

Note: the spacing should be the same regardless of what bit positions are entered. If you don't do any formatting on your output, a one-digit bit position (for example, 5) and a two-digit bit position (for example, 15) will produce differently spaced output.

See Section 5: Test Cases for more sample program runs.

Extra credit: You can earn up to 10 extra points on this assignment by implementing the following logical operations:

- Rotate left (5 points): A rotate left operation is very similar to a left shift operation. However, rather than shifting zeroes into the least significant positions, the most significant bits are moved into those spots. Consider the following example using an 8-bit value, 1011 0100:
  - **1**011 0100 rotated left by 1 bit = 0110 100**1**
  - **101**1 0100 rotated left by 3 bits = 1010 0**101**
  - **1011 01**00 rotated left by 6 bits = 00**10 1101**
- Arithmetic right shift (5 points): Right shifts involving unsigned values always place zeroes in the most significant bits. However, in signed binary representations, the most significant bit indicates the sign and must be maintained in a right shift.

  Arithmetic right shifts preserve the sign of a number by copying the most significant bit to fill all new bit positions. Consider the following 8-bit examples:
  - 0100 0000 right shifted by 1 bit = **0**010 0000
  - 0100 0000 right shifted by 4 bits = **0000** 0100
  - 1100 0000 right shifted by 1 bit = **1**110 0000
  - 1100 0000 right shifted by 4 bits = **1111** 1100

To receive extra credit, implement each of these operations on the two unsigned integer values you originally read in. Note that:

- You will need one additional variable to hold the amount by which you rotate and shift the two variables. You may assume this amount is less than 32.
- You will receive no credit for the arithmetic right shift if you use a signed integer and just do a basic right shift, which preserves the sign. You must manipulate unsigned integer values.
- You can solve both problems using only logical and arithmetic operators (including shift operations), although you may use any code that runs correctly, including material we have not yet covered.
- Your results will display in hexadecimal—remember that each hexadecimal digit represents four bits.

The test cases in Section 5 show the results of valid solutions to these problems.

4. Hints

Output formatting: To format your output correctly, you will have to specify field widths for some outputs. We will cover field width on Friday, 2/10. If you want to finish this assignment before that date, I suggest reading the appropriate textbook section (or another resource) for more details.

Multi-bit operations: Bitwise operations often use a *bit mask*—a quantity that can be used to modify one or more bits while all other bits stay the same. The type of mask you use depends on the operation—usually, the mask bits in the position(s) to be changed will equal 1, while others will be 0.

Bit masks for single bit operations are typically generated by taking a single "1" and shifting it into the right bit position. Multi-bit masks are similar—you need to take the correct number of "1"s and shift them into the right positions. However, doing so can be somewhat tricky. One way to approach this problem might be the following:

- Write out some example bit masks for multi-bit operations. For example, to set the highest 16 bits of a number, use the mask 0xFFFF0000. For other examples, you could consider what the mask should look like to:
    - Set the lowest 16 bits of a number
    - Set the highest 8 bits of a number
    - Clear the highest 8 bits of a number
    - Flip the middle 16 bits of a number
- Now, look at those masks and determine what the lowest and highest bit positions are in which you have 1s (or 0s)—that's the range of bits being changed. In the example above (0xFFFF0000), bits 16-31 are equal to 1.
    - You should now be able to determine what the masks look like for arbitrary bit positions—say, bits 7-13, or bits 24-30.
- To generate each mask in your code, you'll need to do the following:
    - Start with a value containing multiple ones. (*Hint:* think about what hexadecimal value contains the maximum number of ones you could need.)
    - You'll use the low/high bit positions in shift operations to help generate your mask. (*Hint:* using only the positions may not work—your shift amounts may combine the positions with other values).
        - One approach: generate two masks—one based on each position—and combine them so that your final mask contains 1s (or 0s) where the two overlap
        - Another approach: shift the initial value in one direction to clear some of the lower bits, then shift it in the other direction to clear the upper bits.

5. Test Cases

Your output should match these test cases exactly for the given input values. I will use these test cases in grading of your assignment, but will also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.