1. **Introduction / Purpose / Intent**

For this assignment we were tasked to select and complete 4 of 6 available bitwise operation puzzles that were provided as function declarations by the Lab3 assignment page on canvas. These function definitions had specific goals and restrictions to challenge problem solving abilities and out of the box thinking. The intent of the overarching lab was to solidify the understanding of bitwise operations for the coding individual.

The general limitations set for accomplishing this lab were: the use of integer constants valued only 0 through 255 (0xFF), the use of only local variables, the use of unary integer operations “!” and “~”, and finally the use of binary integer operations “&”, “^”, “|”, “+”, “<<”, and “>>”. Each individual puzzle could further restrict the use of these operators. The lab expressly forbids the use of: control constructs, define or use of macros, define or use of additional functions, use of other operators such as “&&”, “||”, “-“, or “?:”, use of any form of casting, and use of any data type other than int. The lab also outlines the following assumptions of the hardware: uses 2s complement, 32-bit representation of integers, performs right shifts arithmetically, and has unpredictable behavior when shifting an integer by more than the word size.

No files were provided to outline the lab. It is explicitly stated that the main() function needs to be coded in a way that provides the necessary values to demonstrate the correct performance of these functions as well as present the results of these calculations. It is stated that in printing it is most useful to output the int variables in their hexadecimal form, and to utilize the proper formatting when using the printf() function.

1. **Process**

The first step in this lab was to select the 4 puzzles to solve. My analysis of the problems indicated that functions 1 and 2 as well as 3 and 4 were related, and would make a good pair of pairs to tackle. This was also influenced by the breakout room I was assigned to in the 15 Oct 2020 lab meeting. My breakout room began discussing the potential logic and pseudo-code of the first function. The breakout room ended shortly after and I began working on the program.

To begin I copied into a blank bits.c file the 4 function declarations I planned to solve, along with all of the associated instruction and example comment text. Underneath these declarations I wrote my main() function, where I would code the solutions of each of the functions for testing purposes prior to completion of each of the functions.

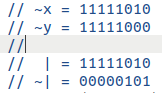
*Function1 - Compute the bit-wise AND of two integers, using only the ~ and | operators*:

Having such a limited scope of operators made the logic of this function puzzle relatively simple to imagine. I began by commenting the bits of the example variables and expected result of the function which gave me a visual with which to work.

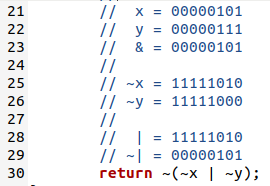




Because the OR (|) operator can only operate on bits containing a 1, I wrote out what the compliment (~) of what each of the example variables would be in bit value. This highlighted to me the pattern I needed to see, where all of the 0 values had previously been such as in the 2nd bit of the x variable. This pattern showed where bits were similar and different and was also operable with an OR statement. The OR result of the compliments of the variables had the exact bit pattern, of opposite values, I was looking for in the above screenshot, so the result of the OR operator is complimented again to reverse the 0 conversion process.



The original working code in main() utilized 3 variables to accomplish this, storing the result of the OR in a **temp** variable to be output to the console with printf(). The value returned matched expected, 4, and the code was moved to the function declaration to create a function definition of function1(). The **x** and **y** variables were removed, as they would be passed in by the function, and **temp** was initially returned after storing the value of the OR. It was indicated that returning a variable may not accomplish the intent of the lab, so I removed the temp variable and moved the operations to the return line. With this, function1() was complete and I moved on to the next puzzle.



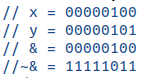
*Function2 - Compute the bit-wise XOR of two integers, using only the ~ and & operators*:

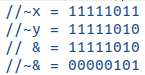
Following the same procedure I used for function1(), I began function2() by writing out the bit values of the example variables and the intended result of the function once complete.





Not knowing where to begin for this problem, I wrote out the bit values of the only potential operations I had available to me. This included an AND (&) of the variables and the AND (&) of their compliments (~). This helped to highlight the pattern I needed to see for the XOR operation, where the first bit of these results matched, but were 0s. This led to taking the compliment of both AND results in order to work with those bits.

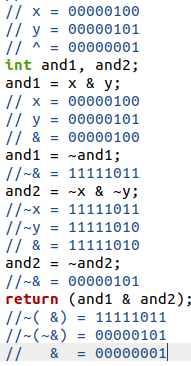




Because the only matching bits were in the first bit location, another AND of these results would result in a matching case to the expected XOR pattern. This is because the first AND result of the original variables, complimented, indicates in 0s where each of the bits that held value matched. The second AND result of the complimented variables, complimented, indicates in 1s where any of the bits may have had value existed in either variable, functioning as an OR. By utilizing these 2 values, a final AND indicates where these 2 conditions overlap, matching the expected pattern of an XOR operation on the 2 example values.



This logic was coded using 5 variables in main(), **x**, **y**, **and1**, **and2**, and **temp**. The values of the AND operations for the original and complimented variables were held in **and1** and **and2**. These variables were complimented and stored back into their variables. A final AND was conducted on the variables and placed in temp, which was finally output using printf(). The expected output matched and I moved the code to function2(). The **x**, **y**, and **temp** variables were removed from the function definition as before, leaving the **and1** and **and2** local variables to hold data.



Further analysis of the assignment page was conducted, as I was unsure that this return statement would be valid for the intent of the lab. As stated in this report in the introduction “*The use of* ***local variables*** *within the function is authorized, while the use of* ***global variables*** *is expressly forbidden*.” This indicated to me that this form of a return statement would be sufficient for accomplishment of the lab. With this, I moved onto the next pair of related functions.

*Function3 - Return 1 if any odd-numbered bit in an integer is set to 1, 0 otherwise*:

Starting a new pair of related functions I again started by writing out what the bit values would be of the expected inputs and outputs.

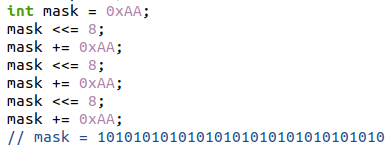




The next thought I had was to construct a mask that would preserve only the odd numbered bits. This I also recorded in binary value, assuming the length of the integer would be 32-bit as instructed.



To construct this mask, because of the limitation of integer constants 0-255 (0xFF) and the explicit forbid of use of loops, I created a **mask** variable and assigned it a value of 0xAA. This value when translated to binary is 8 bits long where every odd bit is 1 and every even bit is 0. This value of **mask** was shifted left (<<) over a byte, and another 0xAA was added to it. This process was repeated, without a loop, twice more to have a full 32-bit **mask** of odd values.



It is my understanding that this implementation does not break any limitations set for the completion of the assignment, however alternate interpretations have been proposed from other students. This implementation of an integer constant equates to decimal value of 170 and is iterated rather than simply assigning the value of **mask** to 0xAAAAAAAA. It is my understanding that assigning mask a value of 0xAAAAAAAA would break the limitations set forth by the lab document on canvas, as this value has a decimal value of 2,863,311,530 and is clearly beyond the scope of allowed values. If this were to violate the limits set, the solution would be to alter **mask** to contain only an 8-bit value of 0xAA, and left shift (<<) **mask** itself a byte and continue operation.

With **mask** constructed, I declare 2 more variables into existence, **x** and **temp**. The first of these, **x**, will be utilized for testing in main() to simulate the value passed into the final function, while **temp** will be utilized to hold any results of operations to be output with printf(). With these new variables, I assign **x** a value of 0x7 and preform an AND (&) operation with it and **mask**, placing the resulting odd bits into **temp**. The new value of **temp** is now 00000010 (0x2), and because value is present in any odd bit, the output should be a 1.

It was discussed in the lab session that the logical NOT (!) may be utilized to obtain simplified values, due to its conversion of any value to Boolean 0 or 1. Following this logic I utilized a single NOT (!) operator on with **temp** to simplify any results passed to it, which provided the opposite of the answer we seek. Utilizing an additional NOT (!) operator corrects this opposition and the function returns the correct value. I tested this logic with a printf() statement to output **temp**, which succeeded. I then changed the value of **x** to 0x5, which contains no value in the odd numbered bits, and ran the test again. The 0 from **temp** is simplified and reversed to correctly return a 0.



With the code for the definition of function3() complete, I copied what I had been working with from main into the function definition. Once copied, I removed the **x** variable declaration, as **x** was now being passed into the function. I also removed the temp function, opting instead to return the operations conducted between **x** and **mask**. Follow on testing indicated the same results as with utilization of the **temp** vartiable. With this, function3() was complete and I moved onto the final function puzzle.

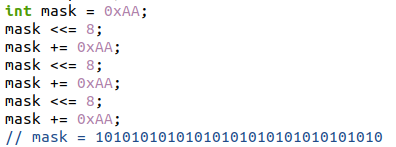


*Function4 - Return 1 if all the odd-numbered bits in an integer are set to 1, 0 otherwise*:

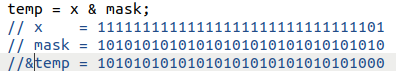
Like the prior functions I began by writing out the example values in bit value along with their expected outcomes. Being related to function3(), in that it utilizes the odd numbered bits, I also copied down the process for which the 32-bit **mask** for odd bits was created.



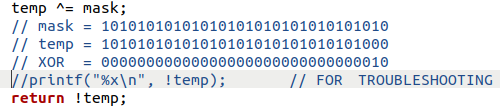




I then declared into existence the **x** and **temp** variables similarly to function3(), along with writing out the line that takes the AND (&) operation of **x** and **mask**, storing it within **temp**. I did this because I recognized that I would again only be working with the odd bits of any input, so the first thing I would do in the final function would be to eliminate the even bits. I then assigned **x** the value of 0xFFFFFFFD, which fills all of its bits with a value of 1 except for the first odd bit. I wrote out the bit values of the operation I had already declared, showing that the value 0xFFFFFFFD once masked does not fill all of the odd bits.



This code pattern highlighted that **mask** and the result of the AND (&) operation, now stored in **temp**, nearly match, and an XOR (^) could be utilized to place the only missing bits from **x** into **temp**. If **temp** contained all odd bits, an XOR (^) with **mask** would result in no value within **temp**. These results are the exact opposite of the outcome requested by the puzzle, where if all bits are present in the original variable, a 1 is to be output, and if any of the odd bits are miss, or rather any value is left after the AND and XOR, the output is to be a 0. This is achievable with the same NOT (!) operator as in function3().

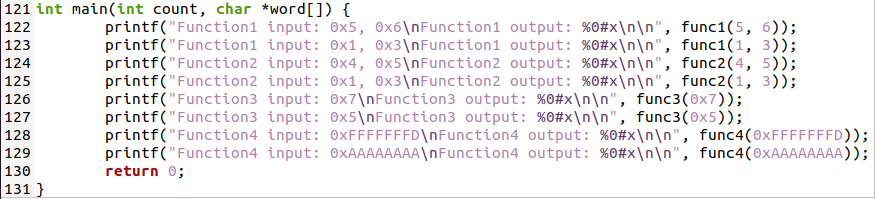


This logic was tested with a printf() of **temp** while in main(). The logic succeeded and the alternate example value was tested, 0xAAAAAAAA, which contains only values in the odd bits. Again this test passed and the code was copied into the function definition. This logic was more difficult to rewrite without using a temp variable, so the only variable to be removed was the **x** variable, as it would be passed into the function.

It is my understanding that this form of a return statement is legal within the limits set by the assignment. To quote the introduction “*The use of local variables is authorized*.” It has been noted to me by other students that the following line from the assignment page on canvas may contradict this thinking; “*Replace the "return" statement in each function with one or more lines of C code that implements the function*.” I believe my implementation of the function accomplishes what is being asked by this line, as nowhere else in the instruction are we limited to not return variables. Because these functions are declared to return values of type int, and we are expressly forbidden to utilize global variables and pointers, I believe that this is an appropriate implementation of the function.

*Main()*:

The lab page indicates, to quote the introduction; “*The main() function needs to be coded in a way that provides the necessary values to demonstrate the correct performance of these functions as well as present the results of these calculations*.” With this instruction, I proceeded to write out a printf() statement for each of the example cases for each function. Because function1() and function2() only had a single example case, I wrote an additional example for each. These utilized low bit values of 1 and 3 to demonstrate the effective AND (&) and XOR (^) operation the 2 functions accomplished respectively. With this, The program was complete.

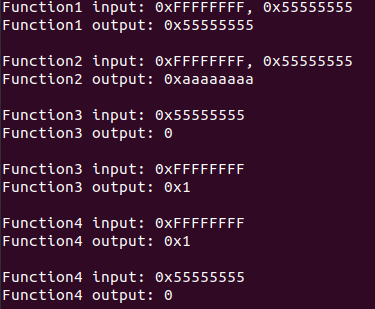


1. **Testing**

All throughout the coding process many “printf();” statements were used to output variables in various locations, mostly within the main() function while writing each of the functions code. These statements allowed me visually follow the values of my pseudo-functions and verify that my calculation and logic were correct. These statements have been removed from the final submitted bits.c file, but remain in the working copy I retain for myself.



After the code was finished, additional testing of values for each function were conducted to verify operation. The values chosen were translated to bit value, using Microsoft calculator program in programmer mode, and recorded as a comment. These values were selected to produce, for each function, expected return 0 and expected return value cases. For example in function4() a value of **x** that would return value that was tested was 0xFFFFFFFF, which fills all bits with value. A value of **x** for function4() that was tested and would not return value is 0x55555555, as it fills all of the even bits with value. These values were also used to test the other functions, as they would all produce some expected result.



1. **Results**

The results of this lab are successful in completing the intent of the assignment. I have successfully solved and tested 4 function puzzles of the available 6 fully within the constraints of the limitations provided. When limitation boundaries were in question, I utilized my other student’s interpretations to assist in my own understanding in those limits, and those instances have been recorded in the **Process** portion of this report. Additionally, I have completed the main() function of the program to accurately demonstrate the functionality of these functions and their provided inputs. These 3 tasks made up the work to be done for Lab3, and have all been accomplished as indicated by this report.

1. **Conclusions**

Based on the results and intent of this assignment I conclude that one of the keys to successful programming is out of the box thinking in terms of problem solving. This lab challenges the programmer to implement a limited set of tools and skills to obtain a specific solution. Completion of the lab indicates the ability to operate within those constraints in an effective manner, especially when these constraints force lines of thinking that are not natural to the coder. The most difficult portion of this assignment for me was the interpretation of the limits set to accomplish the lab. I operate most efficiently when I understand and know what boundaries I can operate within, and I had to refer to the original text multiple times through the coding process. Additionally, as stated, I had to ask for clarification multiple times from other students, but this process demonstrates an ability to overcome obstacles much like the coding of the lab assignment itself.

1. **References / Acknowledgements**

C Programming Language, B. W. Kernighan & D. M. Ritchie, 2nd Edition, Prentice Hall, 1988.

C Programming: A Modern Approach, K.N. King, Norton, 2008.

Windows Calculator, Programmer mode

[Bitwise Operators in C Programming](https://www.programiz.com/c-programming/bitwise-operators#xor)

[6.4.2 Logical Negation](https://www.cs.auckland.ac.nz/references/unix/digital/AQTLTBTE/DOCU_061.HTM#log_negative_sec)

Jared Larson assisted with writing the pseudo-code for function2 and general logic checking during a lab session breakout room on 15 Oct 2020

Keegan Giles assisted with writing the pseudo-code for function4 on 16 Oct 2020

Douglas Slater assisted with understanding of the assignment, which lead to me removing variables from the return lines for expressions on 16 Oct 2020