CSED211: Data Lab1 Report

20230499 / KimJaeHwan

1. Overview

This lab aims to solve 5 questions with only allowed operator such as bitwise, unary, or addition operator.

Lab has 5 questions: bitNor, isZero, addOK, absVal, logicalShift.

2. Question #1. bitNor

2-1. Explanation.

- For the first question, we have to implement bitNor functions, which perform the bitwise NOR operation.
- We are allowed to use the ~, & operators at most eight times.
- Table 1. shows the operation of the NOR operation.

	X = 0	**X = 1**
Y = 0	1	0
Y = 1	0	0

Table 1. Operation of NOR operation.

2-2. Solution.

- Based on the De Morgan's laws, we can express bitwise NOR operation using only ~ and &.
  - One of De Morgan's laws: "The negation of 'A or B' is the same as 'not A and not B.'"[1]
- Generally, De Morgan's laws are used in logical operations, but it can be used in bit operations because both operations follow Boolean algebra. Moreover, bit operations is one of logical operations with 0 and 1.

2-3. Implementation.

- We can implement the above solution with C as shown in Listing 1.
- By De Morgan's laws, ~x&~y is equal to ~(x | y), which is definition of NOR.

```
bitNor(x,y){
    return ~x&~y;

}
```

Listing 1. Code of bitNor.

3. Question #2. isZero

3-1. Explanation.

- The task is to implement a function that determines that argument is zero or not.
- We are allowed to use !, ~, &, ^, |, +, <<, >> operators at most two times.

3-2. Solution.

- In C, only 0 is considered false, but any other integers are considered true. So we can use ! operator to solve this problem.

3-3. Implementation.

- We can implement the above solution with C as shown in Listing 2.

```
ixZero(x){
    return !x;

}
```

Listing 2. Code of isZero.

4. Question #3. addOK

4-1. Explanations.

- For the third question, we have to implement the addOK function, which checks adding two integers results in overflow or not.
- We are allowed to use !, ~, &, ^, |, +, <<, >> operators at most 20 times.

4-2. Solution.

- To solve this question, we must consider several important observations and techniques:
- First is the extracting the sign bit. An integer type has 4 bytes (32 bits) and it uses the left-most bit as sign bit. So, we can calculate the sign bit with right arithmetic shift (>>) 31 times. The positive integer or 0 will be 0, and the negative integer will be -1 when they are bit-shifted 31 times. So, !!(x>>31) will be 1 when x is negative integer, and it will be 0 when x is positive integer or 0.
- Second is how to check overflow occurs. The first case is when two arguments have different sign bit. In this case, overflow never occurs. The second case is when two arguments have same sign bit. In this case, if sign bit of arguments and sign bit of sum of arguments are different, overflow occurs.

4-3. Implementation.

- We can implement the above solution with C as shown in Listing 3. The part before | operator is the first case, and the part after | operator is second case. The result of OR of two cases is our answer.

```
addOK(x,y){
    return ((!!(x>>31))^(!!(y>>31))) | !((!!((x+y)>>31)^(!!(y>>31))));

}
```

Listing 3. Code of addOK.

5. Question #4. absVal

5-1. Explanations.

- The task of question #4 is implement the absVal function that return the absolute value of an argument.
- We are allowed to use !, ~, &, ^, |, +, <<, >> operators at most 10 times.

5-2. Solution.

- If the argument is negative integer, we have to get the two's complements of argument. The calculating processes are (1) flipping each bit 1 and 0, (2) adding 1. The first process is same as doing XOR with -1 (0b11…11).
- If the argument has 0 as sign bit (in other words, it is positive integer or 0), we don't have to do anything. However, we can use a similar process. The calculating processes are (1) no flipping and (2) adding 0. The first process is same as doing XOR with 0.
- Two cases have same process, XOR and addition. So, the last one we need is adjusting operand of each process (XOR: -1 or 0, Addition: 1 or 0) using sign bit. Note the below implementation part.
- In this implementation, there is a fault: absVal(-2147483648) returns -2147483648. INT_MAX is 2147483647, but INT_MIN is -2147483648. So, we cannot store absolute value of INT_MIN in bound of integer type. We have to use more large type such as long long to prevent this fault.

5-3. Implementation.

- We can implement the above solution with C as shown in Listing 4.

```
absVal(x){
    return (x ^ (x >> 31)) + (~(x >> 31) + 1);

}
```

Listing 4. Code of absVal.

6. Question #5. logicalShift

6-1. Explanations.

- To solve the last questions, we have to implement logical shift function.
- We are allowed to use ~, &, ^, |, +, <<, >> operators at most 20 times.
- The basic right shift operator (>>) in C is arithmetic shift. This operator shifts the bits to right, and fill a left-empty bit spaces with sign bit. Contrast, logical shift fills the left-empty bit space with 0 regardless of sign bit.

6-2. Solution.

- The one of simple ways is to use & (AND) mask after performing logical shift. For example, when we execute logicalShift(x, 1), we first perform right arithmetic shift, then apply & operation with 0b011…11 (where there is a n = 1 time 0 at the left and the rest are 1s)
- In this situation, there are two cautions: UB (Undefined Behavior) and – operator
- If n, the number of shift operations, is zero, UB can occur due to shifting more times than the available 32 bits. So, we need to handle the exception where n = 0.
- We cannot use – operator because of constraint of problem. So, we have to use (~x+1) instead of (-x).

6-3. Implementation.

- We can implement the above solution with C as shown in Listing 5.

```
logicalShift(x,n){
    return (x >> n) & ((!!(n) << (32 + (~n+1))) + (~1+1));

}
```

Listing 5. Code of logicalShift.

7. Results

Figure 2. shows the output of the grading program `btest` and programming rule checking program `dlc`. I received a score of 12/12 without violating programming rules.

[Figure 2.] The result of btest and dlc.

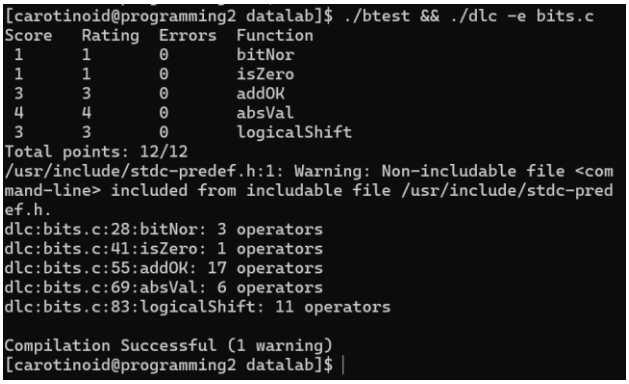
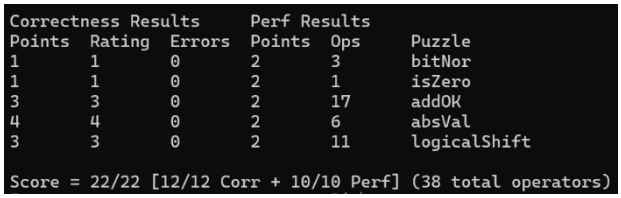


Figure 3. shows the last part of output of driver.pl, and I got 22/22 score with total 38 operators.

[Figure 3.] The result of driver.pl



8. References

Wikipedia.

[1] De Morgan's laws, Wikipedia.