CSED211 Intro. to Computer SW Systems : Lab1 Report

September 9, 2025

Student 20240505 Hyunseong Kong

1. Overview

This lab aims to solve the 5-consecutive bit puzzle to gain a better understanding of the bit-level representation of integers.

2. Implementation

2-1. bitNor(x,y)

Request: Given `x` and `y`, compute $(x \,|\, y)$ using only `~` and `&`.

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

Explanation: This follows directly from De Morgan's law. No further explanation is needed.

2-2. isZero(x,n)

Request: Given `x`, return 0 if x is non-zero, else 1.

```
int isZero(int x) {
    return !x;
}
```

Explanation: This is clear since C interprets 0 as *False* and any nonzero value as *True*. Note that the operator `!` negates the truth value.

2-3. addOk(x,y)

Request: Given `x` and `y`, determine if we can compute `x+y` without overflow.

```
int addOK(int x, int y) {
    return 0x1 & (((x ^ y) | ~(x ^ (x+y))) >> 31);
}
```

Explanation:

- `1 & (□ >> 31)` returns the most significant bit of □.
- `(x ^ y)` has its most significant bit set to 1 if x and y have different signs. Otherwise, its most significant bit is 0.
- `~(x ^ (x+y))` checks whether x and $x+y$ differ in sign. Although parentheses around `x+y` are not strictly necessary, they are included for better readability
- The operator `|` combines `(x ^ y)` and `~(x ^ (x+y))`.

The full expression indicates the following:
If x and y have different signs, the sum cannot overflow. In this case, the the operation yields 1, since the most significant bit of `x ^ y` is 1, which after shifting results in all bits being set to `0xFFFFFFFF`.
Otherwise (i.e., x and y have the same sign), we check whether $x+y$ has a different sign from x. If so, overflow has occurred. (Since in two's complement arithmetic, it is impossible for overflow to occur while $x+y$ and x still share the same sign.) In this case, the most significant bits of `x ^ y` and `~(x ^ (x+y))` are both 0, so the final result after shifting is 0x0.

2-4. absVal(x)

Request: Given `x`, return the absolute value of x.

```
int absVal(int x) {
    return ((x >> 31) & ((~x)+(~x)+2)) + x;
}
```

Explanation:

- `(x >> 31)` fills all bits with the sign bit, yielding either `0x0` or `0xFFFFFFFF`.
- If the result is `0x0`, the whole expression `((~x)+(~x)+2)` is ignored due to the `&`, leaving only x.
- For negative x, the expression becomes `((~x)+(~x)+2) + x`. Note that `(~x)+1 = -x`, so adding this twice flips the sign.
- Overflow may occur, but this is irrelevant because signed integer addition wraps around, ensuring that the associativity law still holds.

2-5. logicalShift(x,n)

Request: `x` and `n`, perform a logical right shift of x by n.

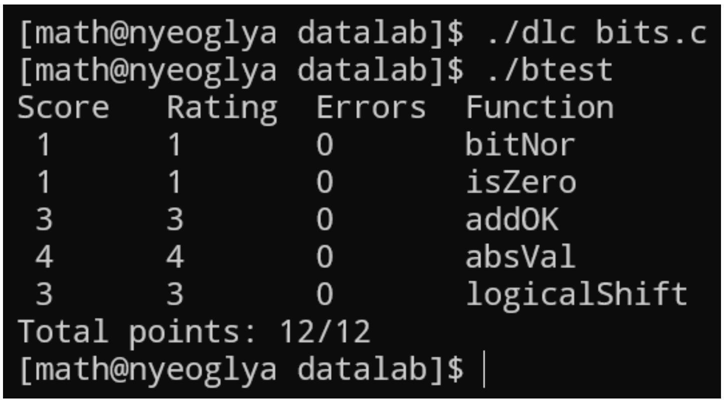
```
int logicalShift(int x, int n) {
    return (x >> n) & (~((!n) + (~0)) | (((~0) ^ (1 << 31)) >> (n + (~0))));
}
```

Explanation: The basic idea is to mask the shifted number with a value of the form `00...011...1`.

- `(x >> n)` shifts x by n. This works correctly for $x \geq 0$, but problems arise when $x < 0$.
- To fix the above issue, we can use `(0x7FFFFFFF >> (n - 1))` to mask the first n bits to 0.
- Next, if $n = 0$, the masking phase should be skipped. This is done by adding `~((!n)-1)` in front of the expression and performing a bitwise OR. If $n = 0$, then `!n = 1`, so the entire expression becomes `0xFFFFFFFF`, ignoring the following expressions. If $n = 1$, then `!n = 0`, so the entire expression becomes `0x0`, which causes the masking expression to take effect.
- Finally, special cases such as `-1` and `0x7FFFFFFF` are handled by replacing them with `~0` and `(~0) ^ (1 << 31)`, respectively.

Note that much of the parentheses exists because of the operator precedence.

3. Result



The above screenshot shows the output of the grading program `btest` and programming rule checking program `dlc` run on my local device. I obtained 12 points out of 12 without violating any programming rules.

4. References

[1] Operator Precedence: `https://cppreference.com/w/cpp/language/operator_precedence.html`