**2022-2 CSED211 Lab01 Report**

학번 : 20210479

이름 : 이주현

**명예서약 (Honor Code)**

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

1. **문제별 코드 설명**
   * **Problem 01\_bitNor()**
     + 드 모르간 법칙을 이용해 and와 not 연산으로 nor 연산을 구현했다.
     + ~(~(~x & ~y))에서 ~(~x & ~y) = x | y이므로 ~(~(~x & ~y)) = ~(x | y)이다.
   * **Problem 02\_isZero()**
     + 사용 가능한 operator 중 !는 nonzero인 수를 모두 true로 간주하고, 0을 false로 간주해 0이나 1을 리턴하는 operator이다.
     + 따라서 주어진 문제를 해결하기 위해서는 !x를 쓰는 것이 적절하다.
   * **Problem 03\_addOK()**
     + 문제 해결에 앞서, signed와 unsigned의 경우 모두 bit 표현 방법과 bitwise operation이 같으므로 overflow가 발생하는 조건은 같을 것이라고 생각하고 해결을 시작했다.
     + 우선 이 문제를 해결하기 위해 어떤 상황에서 overflow가 발생하는지 알아보아야 한다. Overflow는 두 수의 msb가 모두 0이고, carry-in이 1, carry-out이 0인 경우 또는 두 수의 msb가 모두 1이고, carry-in이 0, carry-out이 1인 두 경우에 발생한다.
     + Overflow가 발생하는 상황을 잡아내기 위해 두 경우의 공통점을 뽑았다. 우선 두 수의 msb가 같은지 확인하는 것을 조건으로 걸었다. 이는 ((x^y) >> 31) & 1이라고 구현했다. 이렇게 하면 msb가 같을 경우 0이 되고, msb가 다를 경우 1이 된다. ( nonzero가 모두 1이라고 간주되기 때문)
     + 또한 msb가 같은 상황, 즉 overflow가 발생할 수 있는 상황에서 overflow 가능성을 확인하기 위해서는 carry-in과 carry-out의 값이 다르다는 것, 즉 carry-in^carry-out의 값이 1임을 확인해야 한다. 이때 overflow가 발생할 수 있는 두 경우에서 carry-out과 msb의 값이 같으므로 carry-in^msb로 대체해 더욱 쉽게 구현할 수 있도록 하였다.
     + Msb는 x >> 31, carry-in은 (x + y) >> 31이므로 이를 좀 더 간단화해 나타내어 (x ^ (x + y)) >> 31이라고 표현하였다. 그런데 이때 overflow가 발생하는 경우 (x ^ (x + y))의 값이 무조건 1이 되게 되는데, 문제4의 조건에 의하면 overflow가 발생할 때 0을 반환해야 하므로 해당 항에 ~ operator를 붙여 overflow 발생 시 값이 0이 될 수 있도록 했다. 즉 ~(x ^ (x + y)) >> 31로 표현했다. 그리고 이것을 1과 and operation으로 비교하는 항을 더해, overflow가 발생하지 않았을 경우 무조건 1을 도출할 수 있도록 만들어 준다.
     + 이를 종합해보면 31번 left shift하는 부분과 1과의 and operation이 겹치게 된다. 따라서 이를 간단화하여 (((x ^ y) | ~(x ^ (x + y))) >> 31) & 1과 같이 나타냈다.
   * **Problem 04\_absVal()**
     + 절댓값을 구하기 위해서는 positive는 그대로 값을 출력하고, negative는 2의 보수를 취해야 한다는 점을 염두에 두고 구현을 시작했다. 2의 보수를 구하기 위해서는 not operation(~)을 취한 후, 1을 더해야 한다는 것을 이용해 구현했다.
     + Not operation을 취하기 위해 x ^ (x >> 31)이라는 항을 구성했다. 숫자가 positive인 경우 x >> 31은 0000 … 0000이 되고, negative인 경우 x >> 31은 1111 … 1111이 되기 때문에, 이것과 원래 숫자를 가지고 xor operation을 할 경우 positive number에는 아무런 영향을 주지 않고, negative number에는 not operation과 같은 작용을 하게 된다.
     + 그 후 1을 더하는 작용을 위해, ~(x >> 31) + 1이라는 항을 구성했다. 숫자가 positive인지 negative인지에 따라 x >> 31이 0000 … 0000이나 1111 … 1111이 된다는 점을 이용했다. 전자는 0, 후자는 -1에 해당하므로 이것의 2의 보수를 구해 더하면 positive number에는 0이 더해지게 되고, negative number에는 1이 더해지게 된다.
     + Operator priority에 의해 계산이 의도와 달라지는 것을 막기 위해 앞서 설명한 항들을 ()로 한번 감싼 후, +로 최종 연결하여 2의 보수 operation을 완성했다.
   * Problem 05\_logicalShift()
     + 어떤 수에서든 logical shift 작용을 하기 위해서는 original left shift operation을 적용한 후, arithmetic one에 대해서 shift된 비트 수만큼 1과 xor operation을 진행해 해당 bit들을 0으로 바꿔 logical shift를 수행하는 아이디어를 떠올렸다. 예를 들어 음수를 4 bit만큼 left shift했다면, 해당 수와 1111 0000 0000 … 0000을 가지고 xor operation을 진행해 arithmetic으로 1로 채워진 bit들을 0으로 바꿔주는 것이다.
     + 이를 위해 우선 original left shift operation을 진행할 항을 구성했다.
     + 그리고 주어진 숫자가 negative였을 경우 msb부터 n bit만큼 1로 채워진 수를 만들어 xor operation을 진행하기 위해 뒤 항을 ((((x >> 31) << 31) >> n) << 1)로 구성했다. 이것과 앞 항을 xor operator로 연결해 logical shift를 구현했다.
       - Positive number: x >> 31 자체가 0000 0000 … 0000이므로, 어떤 shift를 하든 최종 결과가 0이기 때문에 xor operation을 하여도 아무런 영향을 주지 않게 된다.
       - Negative number: x >> 31이 1111 1111 … 1111이므로 이것을 다시 31bit만큼 right shift하면 1000 0000 0000 … 0000이 된다. 이제 이것을 n bit만큼 left shift하면 (n + 1)개만큼 1로 채워진 숫자가 되고, n bit만큼 1로 채워진 숫자로 만들기 위해 마지막으로 1bit만큼 right shift해준 것이다. 그리고 이것을 최종으로 앞 항과 xor해주면 logical shift를 구현할 수 있게 된다.
2. **Lab01에서 배운 점**

* Lab01을 해결하며, 각 bitwise operator의 작용을 복습할 수 있었다. 간단한 문제들을 푸는 데 bitwise operation을 실제 적용해 보면서 해당 개념을 더욱 잘 이해할 수 있었다.
* 특히 비슷한 작용을 하는 ~와 !를 구분할 수 있게 되었다. 수업에서 배운 대로 ~은 bit-level operator로서 각 비트의 값을 뒤집는 것이고, !은 logical operator로서 변수를 하나의 숫자로 인식해 0 또는 1의 값을 내놓는다는 것을 복기할 수 있었다.
* 그리고 bitwise operation을 이용해 코딩을 하는 경우 자주 사용되는 개념을 익힐 수 있었다. 어떤 목표 숫자(Ex. 1 또는 0으로 가득찬 숫자 만들기, msb부터 원하는 개수만큼 1로 차있고 나머지는 0인 숫자 만들기 등)를 만들기 위해 shift, and, or, xor 등을 응용하는 방법을 깨닫고 잘 적용할 수 있었다.