**CSED 211 Lab 1 report**

20210643 김현준

1. bitNor

Nor 연산은 Or 연산에 Not을 붙여준 것이고, Boolean algebra에서 ~(x|y)=~x&~y 이므로, 이를 이용하여 문제를 해결하였다. ~와 &를 사용하여 ~x & ~y를 리턴하여 bitNor 연산을 구현할 수 있었다.

답: return ~x & ~y;

2. isZero

0은 논리적으로 0이고, 0 이외의 다른 정수는 1이기 때문에, 어떤 값에 논리 연산자 !를 붙이면, 0인 경우 1이 되고, 0이 아닌 경우에는 0이 된다. 따라서 입력받은 x에 !를 붙여서 !x를 리턴하면 0 여부를 판별하는 isZero 함수를 구현할 수 있었다.

답: return !x;

3. addOK

더하기 연산에서 overflow가 있는지 판별하려면 x, y, x+y의 부호 비트를 비교하여, x와 x+y의 부호 비트가 다르고 y와 x+y의 부호 비트가 다르면 overflow가 발생했다는 것을 알 수 있다. 이때, 부호 비트가 다른 것을 확인하기 위해 XOR 연산을 사용하였다. XOR은 둘 중 하나가 1이면 1이 나오는 연산임으로 부호 비트가 다른지 확인할 수 있을 것이라 판단하였다. 따라서 x와 x+y 사이, y와 x+y 사이에서 XOR 연산을 먼저 해 주었다. 그 결과값에는 부호 비트의 자리인 MSB에 부호 비트가 다르면 1, 같으면 0이 들어가 있을 것이다. 다음으로 두 조건이 모두 만족해야 overflow이므로 그 두 결과에 &연산을 해 주었고, MSB에 결과가 있기 때문에 31만큼 rightshift하여 의미 없는 bit들을 모두 지워주었고, 현재 overflow가 발생하면 1인데, 리턴해야 하는 값은 반대로 0이므로 !를 붙여주었다. 이러한 방식으로 overflow를 판별하는 addOK함수를 구현할 수 있었다.

답: return !(((x^(x+y))&(y^(x+y))) >> 31);

4. absVal

절대값을 계산하려면 0과 양수인 경우 값을 그대로 유지하고, 음수의 경우 two’s complement로 계산해줘야 한다. Two’s complement로 음수를 양수로 바꾸려면, -x+1을 해줘야 한다. 따라서 우선 x가 음수인 경우에는 모든 비트를 반전시켜야 하는데, 11…111과 XOR연산을 해 주면 모든 비트가 반전될 수 있고, x가 양수인 경우 모든 비트를 그대로 둬야 하는데 000…00과 XOR 연산을 하면 모든 비트가 유지될 수 있다. 그리고 x의 부호비트가 1이면, x>>31하면 111…111이 되고, 부호비트가 0이면 000…000이 된다. 따라서 x>>31연산을 한 값과 x사이에 XOR연산을 해 주면 그 결과가 x가 음수면 모든 비트가 반전된 결과, 양수면 유지된 결과가 나온다. 다음으로, 음수의 경우 two’s complement로 +1을 더 해줘야 하니 우선 더해주고, x가 양수일 경우 x>>31이 000…000이므로 ~(x>>31)은 111…111=-1이고 x가 음수일 경우 그 반대로 0이므로, 여기에 ~(x>>31)까지 더해주어 양수의 경우 +0, 음수의 경우 +1될 수 있도록 구현하였다. 최종적으로 아래와 같이 절대값 함수 absVal을 구현할 수 있었다.

return (x^(x>>31))+(~(x>>31))+1;

5. logicalShift

x를 n만큼 logicalshift해주는 함수를 구현하여야 한다. Logical shift는 빈칸을 모두 0으로 채워야 하므로, x의 MSB 비트를 구해 놓고, x의 MSB를 0으로 만들어 n만큼 right shift한 값을 구하여 빈칸을 0으로 채운 다음, 다시 미리 구해놓은 x의 MSB를 n만큼 shift하여 넣어 주는 식으로 구현하면 될 것이라고 생각했다. 먼저, 미리 x MSB를 구하기 위해, 1<<31과 x를 &연산해 주었다. 왜냐하면, 1<<31은 1000…000꼴이므로, MSB 비트만 1이기 때문에 &연산하면 MSB 이외의 비트는 0이 되기 때문이다. 다음으로, 이 값을 n만큼 right shift하고 ~연산으로 비트를 반전시킨 다음, 1을 더해 주었다. 이렇게 한 이유는, n만큼 right shift하면 MSB가 1인 경우 빈 곳이 1로 채워지기 때문이다. n만큼 right shift 하고 ~연산을 해주면 MSB가 1인 경우에 앞에 채워진 1과, 원래 x의 MSB비트가 n만큼 shift된 1까지 0으로 바뀌고 나머지가 1로 바뀐다. 그리고 여기에 1을 더하면 결론적으로 원래 x의 MSB 비트가 n만큼 shift된 곳만 1, 나머지는 전부 0이 된다. X가 양수일 경우에는, ((1<<31)&x)>>n이 000…000이므로, ~연산을 시키고 +1을 더하면 다시 000…0000이므로 이 때도 성립한다. 다음으로, MSB를 0으로 놓고 shift시켜 빈칸을 0으로 채운 값은 (((~(1<<31))&x)>>n)로 구하였다. 앞에서 한 것과 반대로 ~(1<<31)과 &를 해주어 x의 MSB를 0으로 놓았고, 이것을 n만큼 right shift하여 빈칸을 0으로 채웠다. 마지막으로, 구한 두 값을 |연산하여 최종 결과를 계산하였다. |연산을 한 이유는, 첫 번째로 구한 값은 n만큼 shift된 MSB 비트만 있고 나머지는 0이고, 두 번째로 구한 값은 n만큼 shift된 MSB 부터 위의 비트들이 모두 0이기 때문에, |연산을 해주면 원래 x를 n만큼 logical shift한 것과 같아지기 때문이다. 결과적으로 이러한 방식으로 아래와 같이 코드를 구성하여 logicalShift 함수를 완성할 수 있었다.

답: return (~(((1<<31)&x)>>n)+1)|(((~(1<<31))&x)>>n);