**Lab1 report**

20210774 김주은

1-1. bitNor

int bitNor(int x, int y) {

return ~x&~y;

}

x와 y의 Nor을 구하기 위해 ~와 &만을 사용하여 구현하였다.

~(x|y) = ~x&~y 이므로 이를 통해 ~와 &를 사용하여 RETURN하도록 하였다.

1-2. isZero

int isZero(int x) {

return !x;

}

x==0이면 1을 return하고 x!=0이면 0을 return하는 함수이다.

!을 사용하여 !x는 x가 0 일때 1이며, x가 0이 아닐때는 0이므로 !x로 구현하였다.

1-3. addOK

int addOK(int x, int y) {

int sign\_differ = (x^y)>>31;

int sign\_same = ~((x+y)^x)>>31;

return (sign\_differ & 1) | (sign\_same & 1);

}

x+y가 overflow가 되지 않고 계산될 수 있는 지를 판단하는 함수이다. x와 y가 부호가 다르다면 x+y가 overflow되지 않는다. overflow는 x와 y가 부호가 같은 경우에만 발생한다.

그러므로, overflow가 되지 않는 경우를 따져봤을 때

x와 y의 부호가 다른 경우는 모두 포함된다.

이외에도 overflow가 되지 않는 경우에 포함되는 경우로는 x+y와 x의 부호가 같을 때이다.

x+y라는 덧셈의 결과값과 x의 부호가 같다면 x가 음수일때도 x+y는 음수로 나온 다는 것이고 x가 양수일때도 x+y는 양수로 나온다는 것을 의미하기 때문이다.

그러므로, overflow가 되지 않는 경우는 x와 y의 부호가 다른 경우와 x+y와 x의 부호가 같은 경우 이렇게 총 2가지 경우이다.

sign\_differ 변수를 통해 먼저 x와 y의 부호가 다른지 확인한다.

(x^y)라고 했을 때 msb가 달라야 부호가 다른 것이다. 이를 shift right를 31만큼 하면 msb가 lsb로 오게 되고 부호가 다르면 1, 같으면 0으로 계산될 수 있다.

sign\_same 변수를 통해 x+y와 x의 부호가 같은 지 확인한다.

(x+y)^x라고 한다면 x+y와 x의 부호가 같으면 msb가 0이 나올 것이다. 이를 shift right를 31만큼 하여 msb가 lsb로 오게 되고 여기에 ~을 사용하여 부호가 같으면 1이 나오게 한다.

마지막에 (sign\_differ & 1) | (sign\_same & 1)을 리턴하여 둘 중 하나라도 1이면 1을 리턴하도록 한다.

1-4. absVal

int absVal(int x) {

//to be implemented

int sign\_minus = x>>31;

return (~sign\_minus & x) | (sign\_minus & (~x+1));

}

절댓값을 구하는 함수이다. x>>31을 통해 msb가 1인지 0인지 확인하여 부호를 확인한다.

(~sign\_minus & x) | (sign\_minus & (~x+1))으로 구현하여 sign\_minus가 1일 때는 x = ~x +1 이 되도록 하고, ~sign\_minus가 0일때는 x를 그대로 return하도록 한다.

sign\_minus가 0일 때는 ~sign\_minus의 모든 bit가 다 1이므로 x와 & 연산을 취했을 때 x가 그대로 나오고 sign\_minus가 모든 bit가 모두 1이므로 (~x+1)와 &연산을 취하여 ~x+1의 결과값으로 나오게 한다.

1-5. logicalShift

int logicalShift(int x, int n) {

return (x>>n) & ~(((1<<31)>>n)<<1);

말그래도 logicalShift를 하는 함수이다.

먼저 x>>n을 하여 n만큼 shift right를 한다. 이후 msb쪽의 n개의 bit들을 전부 0으로 채우기 위해 ~(((1<<31)>>n)<<1)으로 구현한다.

먼저, (1<<31)을 구현하면 msb가 1이고 나머지가 다 0인 bit pattern을 가진다. 여기서 (1<<31)>>n을 하게 되면 msb부터 n만큼 shift right가 되므로 msb쪽의 n+1개의 bit들이 모두 1이 되고 그 뒤의 나머지 bit들은 다 0이 된다. 여기서 n+1이 아닌 msb쪽에서 n개의 bit들만 모두 1로 만들어 주기 위해 <<1을 사용하여 shift left를 한번 해준다. 이렇게 구현하게 되면 결과적으로 msb쪽의 n개의 bit들이 1이 되고 그 뒤의 나머지 bit들은 다 0이 된다. 여기서 msb쪽의 n개의 bit들을 0으로 만들어주기 위해 (((1<<31)>>n)<<1)로 만들어진 것에 ~을 사용하여 msb쪽의 n개의 bit들을 1에서 0으로 바꿔주고 나머지 bit들도 0에서 1로 바꿔준다.

이렇게 나온 bit pattern을 활용하여 처음에 계산한 x>>n과 &로 계산하여 msb쪽의 n개의 bit들을 전부 0으로 만들어주고 나머지부분은 그대로 두는 방식으로 구현할 수 있었다.