Bit Manipulation Report

박수헌 2016314364

**- Detail description about your code**

/\*

\* subOK - Determine if can compute x-y without overflow

\* Example: subOK(0x80000000,0x80000000) = 1,

\* subOK(0x80000000,0x70000000) = 0,

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 20

\* Rating: 3

\*/

int subOK(int x, int y) {

int res = x + (~y+1);

int sx = x>>31;

int sy = y>>31;

우선 sx, sy에 x, y변수가 양수라면 0을 음수라면 -1을 담았다.

int sres = res>>31;

res에는 x-y의 값을 담고, sres에는 res가 양수라면 0을, 음수라면 -1을 담았다.

return !((sx&!sy&!sres) | (!sx&sy&sres));

뺄셈이 정상적으로 이루어졌다면, 양수에서 음수를 뺐다면 양수가 나와야 하고, 음수에서 양수를 뺐다면 음수가 나와야 한다. Overflow가 일어났다면, 양수에서 음수를 빼도 음수가, 음수에서 양수를 빼도 양수가 나왔을 것이기 때문이다.

뺄셈이 정상적으로 이루어진다면 0, overflow가 일어난다면 1을 return해야 하기 때문에, 정상적으로 뺄셈이 일어날 수 있는 경우인 ((sx&!sy&!sres) | (!sx&sy&sres))에 !를 붙여 return 했다.

}

/\*

\* byteSwap - swaps the nth byte and the mth byte

\* Examples: byteSwap(0x12345678, 1, 3) = 0x56341278

\* byteSwap(0xDEADBEEF, 0, 2) = 0xDEEFBEAD

\* You may assume that 0 <= n <= 3, 0 <= m <= 3

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 25

\* Rating: 2

\*/

int byteSwap(int x, int n, int m) {

int n8 = n <<3;

int m8 = m <<3;

1byte는 8bit이기 때문에, m번째, n번째라고 명시할 때 byte 단위로 본다면 m, n에 8을 곱한 값만큼 위치를 이동해야 한다고 볼 수 있다.

int n\_check = 0xFF << n8;

int m\_check = 0xFF << m8;

m, n이 0이라면 0xFF, 1이라면 0xFF00, 2라면 0xFF0000, 3이라면 0xFF000000이 m, n\_check에 저장된다.

int n\_byte = ((x & n\_check) >> n8) & 0xFF;

x & n\_check 를 실행하면, 주어진 x의 n번째 byte만 남고, 나머지 byte들은 0이 된다.

(x & n\_check) >> n8 을 실행하면, 남은 그 n번째 byte가 0번째 byte 자리로 옮겨지게 된다. 이 경우 혹시 3번째 바이트의 MSB가 1이었다면, 1, 2, 3번째 byte들이 F가 될 것이고, 0이었다면 0이 될 것이다.

((x & n\_check) >> n8) & 0xFF 는 혹시라도 1, 2, 3번째 byte들이 F가 되었을 경우에 이를 0으로 만들어주기 위함이다.

int m\_byte = ((x & m\_check) >>m8) & 0xFF;

n\_byte와 같은 과정으로 m\_byte도 만들어주었다.

int bytes\_check = n\_check | m\_check;

m, n\_check에 연산자를 취하게 되면, bytes\_check에는 m과 n번째 byte에만 F가 들어가 있고, 나머지 byte에는 0이 들어가게 된다.

int left\_over = x & ~bytes\_check;

x & ~bytes\_check 를 하게 되면, m과 n번째 byte만 0이 되고, 나머지 byte들은 그대로인 숫자가 만들어져 left\_over에 저장된다.

return left\_over | (n\_byte << m8) | (m\_byte << n8);

n\_byte에 m8을, m\_byte에 n8을 left shift하게 되면, n번째 byte가 m번째 byte 자리에, m번째 byte가 n번째 byte 자리에 들어가고, 나머진 0인 수가 만들어질 것이다. 그때, 이 두 값과 m, n번째 byte만 0인 left\_over값 세 개에 or 연산을 취하게 되면, 결론적으로 m번째, n번째 byte값이 뒤바뀐 결과값이 return될 것이다.

}

/\*

\* logicalShift - shift x to the right by n, using a logical shift

\* Can assume that 0 <= n <= 31

\* Examples: logicalShift(0x87654321,4) = 0x08765432

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 20

\* Rating: 3

\*/

int logicalShift(int x, int n) {

int ArithShift = x >> n;

x의 MSB가 1이라면 앞에 1이 n만큼 복사되고, 0이라면 앞에 0이 n만큼 복사돼, n만큼 오른쪽으로 이동된 숫자가 ArithShift에 저장될 것이다.

int MakeZero = ((0x1 << 31) >> n) << 1 ;

1110000…와 같이 앞에 n개만 1을 갖고 나머지는 0을 갖는 수 MakeZero가 만들어진다.

return ~MakeZero & ArithShift;

~MakeZero는 0001111…과 같이 앞에 n개만 0을 갖는 수이다.

ArithShift의 앞에 n개의 1을 0으로 만들어주는 작업이 필요하기 때문에, 두 수를 & 연산하게 되면 원하는 결과값이 return된다.

}

/\*

\* bitCount - returns count of number of 1's in word

\* Examples: bitCount(5) = 2, bitCount(7) = 3

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 40

\* Rating: 4

\*/

int bitCount(int x) {

int cnt;

int check;

int check2;

cnt = 0;

check = 0x11 | (0x11 << 8);

0x00001111를 만들어 check에 넣어준다.

check = check | (check << 16);

0x11111111을 check에 넣어준다.

check2 = 0xF | (0xF << 8);

0x00000F0F를 check2에 넣어준다.

cnt = x & check;

x의 각 16진수 1 byte마다 첫번째 bit이 1이면 1이 더해지고, 0이면 그대로 유지돼 cnt에 저장된다.

cnt = cnt + ((x >> 1) & check);

cnt = cnt + ((x >> 2) & check);

cnt = cnt + ((x >> 3) & check);

동일한 방법으로 네번째 bit까지 진행한다.

cnt = cnt + (cnt >> 16);

x의 0~15번째 bit에서 1인 bit의 수와, 16~31번째 bit에서 1인 bit의 수의 합이 cnt의 0~3번째 16진수 숫자에 저장된다.

cnt = (cnt & check2) + ((cnt >> 4) & check2);

cnt를 4만큼 right shift하게 되면, 16진수 한 byte가 오른쪽으로 옮겨오게 된다. 따라서, cnt와 cnt를 4만큼 right shift한 값을 모두 0과 2번째 16진수만 1111인 check2와 &연산을 해 더해준다.

return (cnt & 0xFF) + (cnt >> 8);

cnt와 1111 1111을 &연산하면, 0과 1번째 byte에 있는 배열만 가져올 수 있고, 여기에 cnt를 16진법 수 단위로 두 칸 오른쪽으로 가져오면, 2번째, 3번째에 있는 byte들을 가져올 수 있다. 이 두개를 더해 return하면 1이 몇 개 인지에 대한 개수가 나오게 된다.

}

/\*

\* isGreater - if x > y then return 1, else return 0

\* Example: isGreater(4,5) = 0, isGreater(5,4) = 1

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 24

\* Rating: 3

\*/

int isGreater(int x, int y) {

int sx = x >> 31;

int sy = y >> 31;

x, y가 양수라면 sx, sy에 0을, x, y가 음수라면 sx, sy에 -1을 담는다.

int df = x +(~y);

int sdf = df >> 31;

x가 더 크다면 sdf에는 0이, x가 더 크지 않다면 sdf에는 -1이 담길 것이다.

int equal = !(sx ^ sy) & sdf;

x와 y의 부호가 같다면, !(sx^sy)는 -1이고, x, y의 부호가 다르면 !(sx^sy)는 0일 것이다.

x가 더 크다면 sdf는 0일 것이고, y가 더 크다면 sdf는 -1일 것이다.

즉, x와 y의 부호가 다르면 equal에는 0이 저장될 것이다.

X, y의 부호가 같을 때, x가 y보다 크면 equal에는 0이, y가 x보다 크다면 equal에 1이 담길 것이다.

불가능한 경우에 1이 담긴다는 뜻이다.

int notEqual = sx & !sy;

만약 x와 y의 부호가 같지 않다면, 역으로 만드는 줄이다.

X가 양 y가 음이라면, notEqual에는 0이, x가 음 y가 양이라면, notEqual에는 1이 담길 것이다.

불가능한 경우에 1이 담긴다는 뜻이다.

return !( equal | notEqual);

x가 더 클 때 0의 값을 가지기 때문에 !를 앞에 취해준다.

}

/\*

\* fitsBits - return 1 if x can be represented as an

\* n-bit, two's complement integer.

\* 1 <= n <= 32

\* Examples: fitsBits(5,3) = 0, fitsBits(-4,3) = 1

\* Legal ops: ! ~ & ^ | + << >>

\* Max ops: 15

\* Rating: 2

\*/

int fitsBits(int x, int n) {

int check = 32 + (~n + 1);

check에 32 – n의 값을 담는다.

int result = (x << check ) >> check;

x를 (32-n)만큼 left shift한 후 right shift로 복원한다.

return !(result ^ x);

result의 값과 원래 주어졌던 x의 값이 같다면 n비트로 표현이 가능하다는 뜻이므로 1을 return하기 위해 !를 취한다.

}

/\*

\* rotateLeft - Rotate x to the left by n

\* Can assume that 0 <= n <= 31

\* Examples: rotateLeft(0x87654321,4) = 0x76543218

\* Legal ops: ~ & ^ | + << >> !

\* Max ops: 25

\* Rating: 3

\*/

int rotateLeft(int x, int n) {

int left;

int right;

int check;

int result;

left = x << n;

x를 n만큼 left shift 한다.

right = x >> 32 + (~n+1);

x를 32-n만큼 right shift 한다.

check = ((0xFF <<24) >> 24) << n ;

0xff를 24만큼 left shift한 후, right shift하게 된다면, 0xFFFFFFFF이 만들어진다.

0xFFFFFFFF을 n만큼 left shift하게 된다면, 1111111000…의 배열이 만들어지고, 뒤의 0의 개수는 n개일 것이다.

right = right & ~check;

~check는 00000000111…의 배열이며 뒤의 1의 개수는 n개일 것이다.

이 때, ~check와 right을 &연산 시키게 된다면, 앞의 32-n비트들은 모두 0이 될 것이며, 뒤의 n개는 x MSB쪽 n개의 비트들이 LSB쪽으로 이동해 저장될 것이다.

Return left | right;

left는 logical shift였기 때문에, 그대로 이동한 후 LSB쪽 n개의 비트들만 0이었을 것이고, right은 지금 x의 MSB쪽 n개의 비트들을 LSB쪽에 옮겨진 채고 담고 있기 때문에, 이 둘을 or 연산한다면, 결과적으로 숫자가 왼쪽으로 회전된 결과를 return하게 된다.

}

/\*

\* float\_abs - Return bit-level equivalent of absolute value of f for

\* floating point argument f.

\* Both the argument and result are passed as unsigned int's, but

\* they are to be interpreted as the bit-level representations of

\* single-precision floating point values.

\* When argument is NaN, return argument..

\* Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

\* Max ops: 10

\* Rating: 2

\*/

unsigned float\_abs(unsigned uf) {

int check;

unsigned minNaN;

int result;

check = 0x7FFFFFFF;

check의 비트 배열은 0 11111111 1111…. 과 같다.

minNaN = 0x7F800001;

최소의 불능값(0 1111 1111 00000000000000000000001 을 mimNaN에 저장한다.

result = uf & check;

주어진 uf의 sighn bit(s)을 0으로 설정해 result에 저장한다.

if ( result >= minNaN)

return uf;

결과값이 최소의 불능값보다 크다는 것은, 불능이라는 뜻이기 때문에 argument인 uf 그대로 return한다.

if ( result < minNaN)

return result;

만약 결과값이 최소의 불능값보다 작다면, 불능이 아니기 때문에 sign bit(s)가 0이 된 uf의 값을, 즉 절대값을 return한다.

}

/\*

\* float\_i2f - Return bit-level equivalent of expression (float) x

\* Result is returned as unsigned int, but

\* it is to be interpreted as the bit-level representation of a

\* single-precision floating point values.

\* Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

\* Max ops: 30

\* Rating: 4

\*/

unsigned float\_i2f(int x) {

int s;

int bit;

int exp;

int frac;

int frac\_check;

int rounding;

s = x & 0x80000000;

주어진 int형 x와 100000000…를 x연산하게 되면 sign bit(s)이 1인지 0인지 구해 s 변수에 저장할 수 있다.

frac\_check = 0x7FFFFF;

frac 부분을 확인하기 위한 변수로 0111111111…로 설정했다.

if (x == 0)

return 0;

주어진 x값이 0이라면 값도 0이기 때문에 0을 return한다.

if (x == 0x80000000){

x가 불능인 경우이다.

만약 x가 0x80000000이라면 int의 최솟값이다.

exp = 158;

이 때 exp의 값은 127(single precision) +0001 1111(2) 으로 지정해준다.

return 0xCF000000;

158은 2진수로 나타냈을 때 1001 1110(2)으로 나타내짐으로

결과값은 1 10011110 0000… 으로 반환한다.

}

else{

if(s)

x = -x;

x가 음수라면 우선 양수로 바꿔준다.

bit = 1;

그리고 비트에 1값을 지정해준다.

while ((x>>bit) != 0)

x를 양수로 바꾸는 과정 후 오른쪽으로 bit값만큼 shift했다는 것은 만큼 곱한다는 뜻이다.

x의 가장 왼쪽부분의 숫자들을 찾기 위해 while문을 삽입했다.

bit++;

x에 를 곱한 값이 0이될 때 while 문을 나온다.

bit--;

exp = bit + 127;

single-precision이기 때문에 127을 더한다.

x = x << (31 - bit);

frac = frac\_check & (x >> 8);

if (bit > 23){

rounding = x & 0xFF;

floating point에서 rounding을 계산한다.

if ((rounding > 128) || ((rounding == 128) && (frac & 1)))

frac++;

if (frac >> 23){

exp++;

frac = 0;

}

}

}

return (s | (exp << 23) | frac);

}

/\*

\* float\_twice - Return bit-level equivalent of expression 2\*f for

\* floating point argument f.

\* Both the argument and result are passed as unsigned int's, but

\* they are to be interpreted as the bit-level representation of

\* single-precision floating point values.

\* When argument is NaN, return argument

\* Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while

\* Max ops: 30

\* Rating: 4

\*/

unsigned float\_twice(unsigned uf) {

int exp;

int s;

exp = 0x7F800000 & uf;

주어진 uf의 exp 부분만 남기고 나머진 0으로 만든 배열을 exp에 저장한다.

s = 0x80000000 & uf;

주어진 uf의 s부분만 남기고 나머진 0으로 만든 배열을 s에 저장한다.

if (exp == 0x7F800000)

return uf;

exp가 0 1111 1111 0000…. 라면, 2배를 한 값이 불능이기 때문에 uf 그대로 return한다.

if (exp == 0)

return s | (uf << 1);

exp가 0이라면, uf를 1만큼 left shift해 2를 곱한 후, s와 or연산을 하게 되면, sign bit인 s는 그대로 가져갈 수 있다.

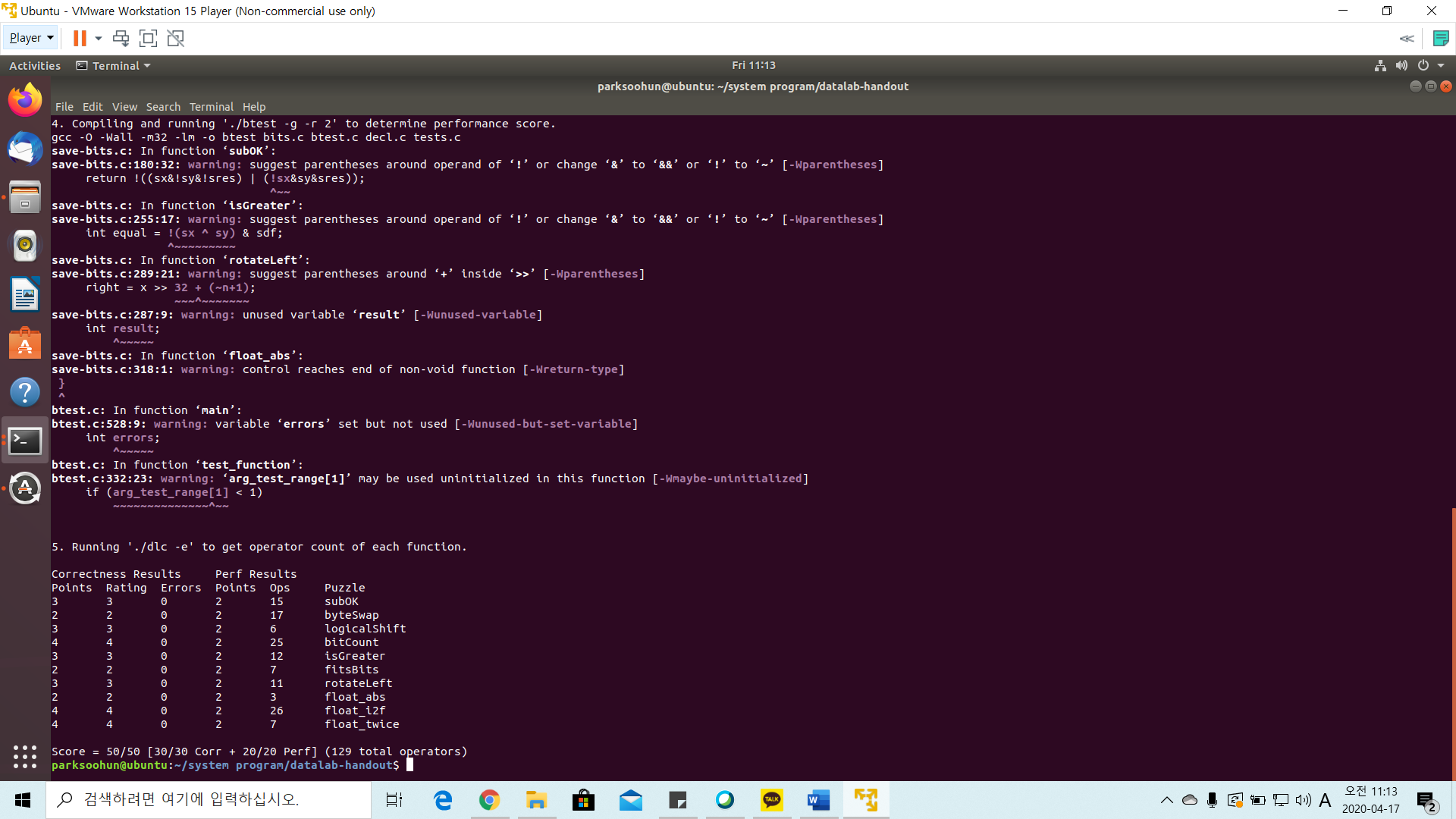
else

return uf + 0x00800000;

나머지의 경우, uf에 s와 mantissa가 0인 숫자 2(0 00000001 0000000…)를 uf와 더해 2를 곱한 값을 return하게 된다.

}

**- Screenshot which includes results of your program**



**- Progress and unique experience of your work**

3일 정도의 긴 시간이 걸린 과제였다. 처음 문제들을 읽었을 때는, 저번에 나왔던 Integer Puzzles와 비슷하다고 생각됐지만, 점점 코딩을 실제로 돌린다는 점에서 어려움들이 많이 발견 되었다. 그래서 생각보다 많은 노력이 필요했다. 교안도 정말 많이 돌려보고 따로 유튜브나, 구글 검색을 통해 공부가 되어 어떤 면으로는 아주 의미 있고 뿌듯한 결과물이라고 생각한다.

Integer 관련 문제들에 있어서는, 긴 고민이 필요하긴 했지만, 결국 답을 찾아낼 수 있었다. 하지만 floating point 관련 문제들은 정말 많은 고민과 추가적인 공부가 필요했다. 이로 인해 integer와 floating point의 연산 과정이라든지, 코딩을 통한 시스템에 대한 정확하고 확실한 이해가 되었다. 또한 같은 과정이라도 정말 많은 방법의 연산 과정들이 있다는 점에서 재미있게 느껴졌다. 그런 과정들을 고민하다 보니, 시간이 정말 많이 든 이유도 있다.

이런 과제를 하기 전, 코딩을 꽤 해보았지만, 이런 2진법 숫자의 연산 같은 건 크게 고민해본 적이 없었다. 그저 정수면 int, 소수면 float을 쓰며 했다. 하지만 이렇게 공부를 해보니, 다른 C 코딩을 할 때도, 계속 어떤 숫자의 배열일까에 대한 고민을 하며 코딩을 하고 있는 내 자신을 발견했다. 그리고 그런 지속적인 사고가 코딩의 논리적 흐름에 도움이 되고 있음을 느끼고 있다.