# CSED211: Data Lab2 Report

20200952 / Chaeyeon Jang

## 1. Overview

Data Lab2 에서는 integer, floating point 와 관련된 함수들을 구현해야 한다. 이번 Lab 은 총 6 개의 문제가 있으며, 각각 negate, isLess, float_abs, float_twice, float_i2f, float_f2i 함수를 구현하는 문제이다.

## 2. Question #1. negate

2-1. Explanation.

- Question 1 에서는 negate 함수를 구현해야 하는데, input x 에 대해 -x 를 return 하는 함수를 구현하면 된다.
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 5, Rating: 2
- 예를 들어, negate(1) = -1 이 된다.

2-2. Solution.

- Lecture 2 pdf 의 p.33 에서, 아래 공식이 2's complement 에서 성립함을 확인할 수 있다.
  - ~x + 1 = -x
- 위 공식을 활용하여 input x 에 대한 negate 를 계산할 수 있다.

2-3. Implementation.

- We can implement the above equation with C as shown in Listing 1.

```c
int negate(int x) {
  return ~x + 1;
}
```

*Listing 1. Implementation of negate function.*

## 3. Question #2. isLess

3-1. Explanation.

- Question 2 에서는 isLess 함수를 구현해야 하는데, input x 와 y 에 대해 x < y 라면 1 을, 그 외의 경우라면 0 을 반환하도록 구현해야 한다.
- Legal ops: ! ~ & ^ | + << >>
- Max ops: 24, Rating: 3
- 예를 들어, isLess(4, 5) = 1 이 된다.

3-2. Solution.

- 우선, x 와 y 의 부호가 다른 경우를 생각해볼 수 있다. x 와 y 의 부호가 다르고, x 가 음수, 즉 x 의 msb 가 1 이라면, x 가 y 보다 작고 isLess 함수의 반환값도 1 이 된다. 반대로 x 가 양수, 즉 x 의 msb 가 0 이라면, x 가 y 보다 크고 isLess 함수의 반환값이 0 이 된다. 이를 통해 x 와 y 의 부호가 다르다면 함수의 반환값은 x 의 msb 부호와 같다는 것을 알 수 있다.
- 다음으로 x 와 y 의 부호가 같은 경우를 생각해보면, x – y 결과의 부호를 통해 isLess 함수의 반환값을 결정할 수 있다. x – y 가 음수여서 msb 가 1 이라면, x 가 y 보다 작다는 것이고, isLess 함수의 반환값도

1 이 된다. 반대로 x – y 가 양수여서 msb 가 0 이라면, x 가 y 보다 크거나 같다는 것이고 함수의 반환값도 0 이 된다. 이를 통해 x 와 y 의 부호가 같다면 함수의 반환값은 x – y 의 msb 부호와 같다는 것을 알 수 있다.

3-3. Implementation.

```
int isLess(int x, int y) {
   int signDiff = (((x >> 31) & 1) ^ ((y >> 31) & 1)) & ((x >> 31) & 1);//If
the sign of x and y are different, save the sign of x in the variable signDiff
   int yNeg = ~y + 1; //compute the negative of y
   int signSame = (!(((x >> 31) & 1) ^ ((y >> 31) & 1))) & (((x + yNeg) >> 31)
& 1); //If the sign of x and y are same, save the sign of x - y in the variable
signSame
   return signDiff | signSame;
}
```

*Listing 2. Implementation of isLess function.*

- x 를 31 만큼 right shift 하고 1 과 & operation 을 하여 얻어진 x 의 msb 와, 같은 방식으로 얻어진 y 의 msb 를 ^ operation 을 통해 서로 같은지 다른지 판단한다. 부호가 달라 ^ operation 결과가 1 이라면, x 부호를 signDiff 변수에 저장한다.
- ~y + 1 연산을 통해 input y 에 대해 -y 값을 계산하여 yNeg 변수에 저장한다.
- x 와 y 의 부호가 같아 ^ operation 결과가 0 이라면, x + yNeg 의 msb 값을 signSame 에 저장한다.
- 최종적으로 signDiff | signSame operation 을 통해 x 와 y 부호가 다르면 signDiff 에 저장된 값을, x 와 y 부호가 같으면 signSame 에 저장된 값을 반환해준다.

## 4. Question #3. float_abs

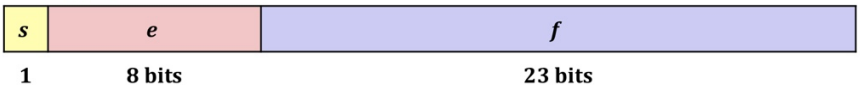
4-1. Explanation.

- Float_abs 함수에서는 input uf 에 대한 절댓값을 구해 반환해야 한다. 이때, question 3 부터는 floating point 에 대한 연산을 구현해야 하고, 함수의 argument 과 result 는 unsigned int 형태로 전달되지만 이는 single-precision floating point 의 bit-level 형태로 해석된다.
- 만약 argument 가 NaN 이라면, 함수 argument 를 그대로 반환해야 한다.
- Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
- Max ops: 10, Rating: 2

4-2. Solution.

- Single precision 은 [Figure 1.]과 같이 s 1 bit, e 8 bits, f 23 bits, 총 32bit 로 floating point 를 나타내는 방식이다.
    - **[Figure 1.]**

- Single precision: 32 bits




- Lecture 3 pdf 의 p.13 에서, 함수의 argument 가 NaN 일 조건은 e = 111…1 이고, f 가 0 이 아닐 경우이다. 그러므로, input uf 의 e 부분, f 부분이 위의 NaN 조건을 만족하는지 확인하고, uf 가 NaN 일 경우 그대로 argutment 를 반환해준다. 만약 NaN 조건을 만족하지 않는 경우 uf 의 sign bit 를 0 으로 바꿔서 return 하면 절댓값을 반환할 수 있다.

4-3. Implementation.

```
unsigned float_abs(unsigned uf) {
   //if e(8 bits) are all 1 and f(23 bits) != 000...0, input uf is NaN.
   //The code below checks whether the input uf is NaN by determining if the
bits of e and f meet the conditions for NaN.
  if ((((uf >> 23) & 0x000000ff) == 0x000000ff) && (((uf << 9) | 0) != 0))
return uf; //if argument is Nan, return argument
  return 0x7fffffff & uf; //else change MSB(sign bit) to 0 and return
}
```

*Listing 3. Implementation of float_abs function.*

- 이때, uf 가 NaN 이 아닌 경우, uf 의 MSB 를 0 으로 바꿔주면 uf 의 절댓값을 얻을 수 있다. 이를 위해 msb 가 0 이고 나머지 bit 는 1 인 0x7fffffff 와 uf 를 & operation 해주어 절댓값을 도출하였다.

## 5. Question #4. Float_twice

5-1. Explanation.

- Question4 에서는 float_twice 함수를 구현해야 하는데, 이는 input uf 에 대해 2 * uf 값을 계산하여 반환하는 함수이다.
- 함수의 argument 과 result 는 unsigned int 형태로 전달되지만 이는 single-precision floating point 의 bit-level 형태로 해석된다.
- 만약 argument 가 NaN 이라면, 함수 argument 를 그대로 반환해야 한다.
- Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
- Max ops: 30, Rating: 4

5-2. Solution.

- Input uf 에 대해 Single precision 에서 e=111…1 이라면, uf 는 infinite value 이거나 NaN 이 된다. 이 경우, 그대로 argument 를 반환해준다.
- 만약 e=000…0 이라면, uf 를 1 만큼 left shift 해주면 된다.
  - e=000…0 인 denormalized value 의 경우, 지수부가 0 이고, 가수부 M 는 0.xxx…x 과 같이 나타낼 수 있으며, uf 의 값은 0 이거나 0 에 매우 가깝다. 또한 아래와 같은 식으로 나타낼 수 있다.
    - $uf = (-1)^s \times M \times 2^{-126}$

        - $E$ 값은 1 – Bias = 1 – 127 = -126 으로 고정이므로, 이 경우 가수부 M 을 2 배로 만들어주면 전체 floating point 인 uf 도 2 배가 된다.

    - 이때, $x \ll a$는 $x \times 2^a$ 와 같으므로, M 을 2 배로 만들어주려면 f 부분을 1 만큼 left shift 해주면 된다. 이를 위해서 uf 를 1 만큼 Left shift 해주면 f 부분도 1 만큼 left shift 가 되고, 여기에 원래 uf 의 부호만 bitwise or operation 으로 붙여주면 된다. 또한 uf 를 left shift 해주게 되면 가수부가 23bit 를 넘어 왼쪽으로 이동할 때 자동으로 normailized case 로 바뀌줄 수 있어 계산 과정이 편해진다. 이 과정을 거치면 uf 의 2 배 값을 계산할 수 있다.
- 그 외의 경우, e part 에 1 을 더해주면 2 * uf 를 구현할 수 있다.
    - 아래 공식은 uf floating point 값을 구하는 공식이다.
        - Uf = $(-1)^s \times M \times 2^E$
    - 아래 공식은 위의 E 값을 구하는 공식이다.
        - E = e – Bias
    - 위 두 공식을 통해, e 에 1 만큼 더해주면 E 값 역시 1 만큼 증가한다는 것을 알 수 있고, E 가 1 만큼 증가하면 $(-1)^s \times M \times 2^{E+1} = (-1)^s \times M \times 2^E \times 2$ 에서 볼 수 있듯이 2 를 곱한 것과 같아진다는 것을 알 수 있다.
    - 따라서 2*uf 를 계산하기 위해서는 e 에 1 만큼 더해주면 된다.

5-3. Implementation.

```
unsigned float_twice(unsigned uf) {
   //if argument is Nan, return argument
   int e = uf & 0x7f800000;
   if (e == 0x7f800000) return uf;
   //if e = 000...0(denomarlized values), do left-shift by 1 bit to uf to
implement 2*uf
   else if(e == 0) return (uf & 0x80000000) | (uf << 1);
   //else, add 1 to e for implementing 2*uf
   else return uf + 0x00800000;
}
```

*Listing 4. Implementation of float_twice function.*

- e 의 bit 가 모두 1 이고, 나머지 bit 는 모두 0 인 0x7f80000rhk uf 를 & operation 해주어 input uf 에서 e 부분만 추출해주고 e 변수에 저장해준다.
- 만약 uf 의 e 부분이 111…1 이어서 e 변수가 0x7f800000 과 같다면, NaN 이거나 infinite 인 경우이므로 그대로 uf 를 반환한다.
- 만약 e 부분이 000…0 이라면, uf << 1 을 통해 uf 를 1 만큼 left shift 해주고, uf & 0x80000000 를 통해 uf 의 부호 bit 만 추출한 후 이들을 bitwise or operation 해주어 반환한다.
- 그 외의 경우, e 부분에 1 만큼 더하기 위해 e 8 bit 중 e 의 lsb 가 1 이고 그 외의 모든 bit 는 0 인 0x00800000 을 uf 에 더해주고 반환해준다.

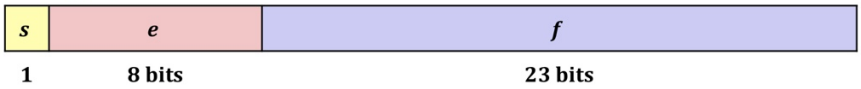
## 6. Question #5. float_i2f

6-1. Explanation.

- Question5 에서는 float_i2f 함수를 구현해야 하는데, 이는 integer type 의 input x 에 대해 float(x)를 반환하는 함수, 즉 x 를 float type 으로 바꾸어 반환하는 함수이다.
- Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
- Max ops: 30, Rating: 4

6-2. Solution.

- Integer type 의 input x 를 single-precision floating point 로 바꾸려면, 아래 식에서 M, E, S 를 구해야 한다.
  - $x = (-1)^s \times M \times 2^E$
- Significand M 의 소수점 아래 부분은 아래 [Figure 2.]의 f 부분에 채워지고(f 의 msb 부터 채워짐), Exponent E 는 e = E + 127 의 계산에 활용되어 계산 결과 도출된 e 는 아래 [Figure 2.]에서 e 부분에 채워진다.
  - **[Figure 2.]**




- 우선, E 를 구하기 위해 x 의 절댓값(unsigned int) abs 를 1 만큼 right shift 하면서, shift 한 결과 모든 bit 가 0 이 될 때까지, 즉 마지막 1 인 bit 가 right shift 될 때까지 반복한다. 그러면 반복한 횟수에서 1 을 뺀 수가 E 의 값이 되는 것을 알 수 있다.
  - 예를 들어, 1110 = $1.110 \times 2^3$의 경우에서 볼 수 있듯이 1 이 나오지 않을 때까지 1110 의 right shift 를 반복한 횟수(4)에서 1 을 빼면 E 값인 3 을 구할 수 있다.
- 그 후, E 값이 f 의 bit 수인 23bit 보다 작을 경우, M 의 소수점 아래 자리 수가 f 의 전체 bit 를 넘지 않는다는 의미이므로 x 의 unsigned int 값인 abs 를 23 – E 만큼 right shift 해주어 그대로 f 자리에 넣어주면 된다.
- 만약 E 값이 23 bit 와 같을 경우, abs 를 그대로 f 자리에 넣어주면 된다.
- 만약 E 값이 23 bit 보다 클 경우, M 의 소수점 아래 자리 수를 msb 부터 23bit 만큼만 가져와 f 자리에 넣어주어야 하는데, 이때 round 를 해주어야 한다.
  - 이때, Lecture 3 pdf 의 p.23 에서, round 를 하기 위해서는 Round bit r, Stick bit s 를 구하고 r 과 s 의 값에 따라 G 를 1 만큼 높여줄지 말지 결정할 수 있음을 확인할 수 있다.
  - R = 1, s = 1 이면 G 를 증가시키고, R = 1, S = 0 이면 round to even 에 따라 G 를 증가시킬지 말지 결정해주어야 하는데, 원래 수의 마지막 bit 가 1 일 경우 odd number 이므로 G 를 1 만큼 높여주고, 원래 수의 마지막 bit 가 0 일 경우 이미 even number 이므로 G 를 1 만큼 높여주지 않는다.
- 그 후 overflow 가 발생했다면 round 결과값을 1 만큼 Right shift 하고, E 값을 1 만큼 증가시킨다.
- e 자리에 들어갈 값을 구하기 위해 e = E + 127 공식을 통해 e 를 구하고, 이를 23 만큼 left shift 하여 e 자리에 넣어준다.
- 마지막으로 input x 의 부호, e 값, f 값을 모두 합쳐 최종적인 floating point 를 구할 수 있다.

6-3. Implementation.

```
unsigned float_i2f(int x) {
   int sign = x & 0x80000000;
   int exp = 0;
   int f;
   int e;
   unsigned int abs = sign == 0x80000000 ? -x : x;
   unsigned int forShift = abs;
   if(x == 0) return x;
   //calculating exp by repeat of right shift operation
   while(forShift){
      forShift = forShift >> 1;
      exp ++;
   }
   exp --;

   if(exp < 23){
      f = abs << (23 - exp);
   }else if(exp == 23){
      f = abs;
   }else{
      int shiftF = abs >> (exp - 24);
      int r = shiftF & 1;
      int s = (exp != 24) ? abs << (56 - exp) : 0;

      f = shiftF >> 1;
      if(r && (s || (f&1))) f += 1; //same as (r && s) || (r && (f & 1)).
Change due to limit # of max operation
   }

   //post-normalization
   if(f >= 0x01000000){
      f =  f >> 1;
      exp++;
   }

   //calculate e based on formula e = E + Bias
   e = exp + 127;

   return sign | (e << 23) | (f & 0x007fffff);
}
```

*Listing 5. Implementation of float_i2f function.*

- 이때, input x 의 절댓값을 구하기 위해 sign 이 0x80000000 인 경우, 즉 음수인 경우 -x 를, 양수인 경우 x 를 abs 변수에 저장해주었다.

- forShift 가 0 이 아닐 때까지 while 문을 반복하여 forShift 를 1 만큼 right shift 해주고, exp 값은 1 만큼 increment 해주었다. 그 후 while 문이 종료되고 exp 에서 1 을 빼주었다.
- 위 과정을 통해 구해진 exp 가 23 보다 큰 경우, rounding 을 위해 round bit, sticky bit 을 구해주었다. Round bit r 을 구하기 위해 abs 변수를 exp – 24 만큼 right shift 하고 1 과 bitwise and operation 을 해주었다. 같은 방식으로 sticky bit s 를 구하기 위해 abs << (56 - exp)을 통해 구하였고, 만약 exp 가 24 와 같다면 sticky bit 은 0 이 되기 때문에 s = (exp != 24) ? abs << (56 - exp) : 0 를 통해 s 를 설정해주었다.
- 그 후 만약 f 가 0x01000000 보다 클 경우, 즉 overflow 가 발생했을 경우 f>>1 을 통해 f 를 1 만큼 right shift 해주고 exp++를 통해 exp 값을 1 만큼 증가시켰다.
- 마지막으로, sign | (e << 23) | (f & 007fffff)를 통해 s, e, f 값을 모두 합쳐주어 최종적인 floating point 값을 반환해주었다.

## 7. Question #6. float_f2i

6-1. Explanation.

- Question6 에서는 float_f2i 함수를 구현해야 하는데, 이는 floating point type 의 input x 에 대해 int(x)를 반환하는 함수, 즉 x 를 integer type 으로 바꾸어 반환하는 함수이다.
- Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
- Max ops: 30, Rating: 4

6-2. Solution.

- Floating point type 의 input x 를 integer type 으로 바꾸려면, 주어진 single-precision 방식의 bit-level floating point 에서 아래 식의 E 값을 구하고 M 값을 기반으로 원래 정수 값을 구해야 한다.
  - $x = (-1)^s \times M \times 2^E$
- 이때 E 값은 e – 127 을 통해서 구할 수 있으며, 원래 정수 값은 f 앞의 bit 에 1 을 붙인 후 23 – E(혹은 E 가 23 보다 크거나 같다면 E – 23)만큼 right shift(혹은 E 가 23 보다 크거나 같다면 left shift)하여 구해질 수 있다.
  - 예를 들어, 110(10 진수로 6)의 floating point 값은 0100000010000…0 인데, 여기서 e 값은 129 이고, E = 129 – 127 = 2 가 된다. 그리고 f 값은 1000…00 인데, f 앞에 1 을 붙이고 (23 – 2)만큼 right shift 하면 110, 즉 6 이 얻어짐을 알 수 있다.
  - f 를 위와 같이 Right/left shift 한 후, 원래 x 의 부호에 따라 x 가 음수면 -를 붙여서 반환해주고, 양수면 그대로 반환해주면 된다.
- 이때, E 값이 0 보다 작다면, 이는 0 이거나 0 에 가까운 수임을 의미하므로 0 을 반환해주면 된다.
- 만약 E 값이 31 보다 같거나 크다면, 이는 out of range 인 수가 된다. Floating point 는 32bit 인데 이를 넘어가기 때문이다. Out of range 인 수일 경우 0x8000000(unsigned)를 반환한다.
- 이때, E 값이 31 이고 f=000…0 이며, msb 가 1(x 가 음수)인 경우, 음의 무한대이기 때문에 0x80000000(signed)을 반환해줘야 한다.

6-3. Implementation.

```
int float_f2i(unsigned uf) {
  int sign = uf & 0x80000000;
```

```
  int e = uf & 0x7f800000;
  unsigned int f = uf & 0x007fffff;
  int exp = (e >> 23) - 127;
  unsigned int result;

  if(exp < 0) return 0;
  else if(exp == 31 && !f && sign) return 0x80000000;
  else if(exp >= 31) return 0x80000000u;
  else if(exp < 23){
   f = f | 0x00800000;
   result = sign? -(f >> (23 - exp)) : f >> (23 - exp);
  }
  else if(exp >= 23){
   f = f | 0x00800000;
   result = sign? -(f << (exp - 23)) : f << (exp - 23);
  }

  return result;
}
```

*Listing 6. Implementation of float_f2i function.*

- 이때, input x 의 sign 을 저장하기 위해 uf & 0x80000000 의 결과를 sign 변수에 저장해주었다.
- 같은 원리로 e, f 값을 저장하기 위해 각각 uf & 0x7f800000, uf & 0x007fffff 값을 e, f 변수에 저장해주었다.
- E = e – Bias = e – 127 공식에 따라 E 값을 구해주기 위해 (e >> 23) – 127 값을 exp 변수에 저장하였다.
- 이때, 만약 exp < 23 인 경우, f = f | 0x00800000 을 통해 f 앞에 1 을 붙여주고, result = sign? -(f >> (23 - exp)) : f >> (23 - exp) 3 항 연산자를 통해 sign 이 0 이 아니면, 즉 음수이면 앞에 -를 붙여 f 를 23-exp 만큼 Right shift 한 값을 반환해주었다.
- 같은 방식으로 exp > 23 인 경우, f = f | 0x00800000 을 통해 f 앞에 1 을 붙여주고, result = sign? -(f << (exp - 23)) : f << (exp - 23) 3 항 연산자를 통해 sign 이 0 이 아니면, 즉 음수이면 앞에 -를 붙여 f 를 exp-23 만큼 left shift 한 값을 반환해주었다.

## 8. Results

Figure 3. shows the output of the programming rule checking program `dlc`. And Figure4. shows the output of the grading program `btest`. I get 19 points without violating programming rules.

**[Figure 3.]**

```
[jcy2749@programming2 datalab-floating-point]$ ./dlc bits.c
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line>
included from includable file /usr/include/stdc-predef.h.

Compilation Successful (1 warning)
```

**[Figure 4.]**

```
[jcy2749@programming2 datalab-floating-point]$ ./btest
Score   Rating  Errors  Function
 2      2       0       negate
 3      3       0       isLess
 2      2       0       float_abs
 4      4       0       float_twice
 4      4       0       float_i2f
 4      4       0       float_f2i
Total points: 19/19
```