CSED211: Data Lab2 Report

20240832 / 채승현

1. Overview

이번 랩에서는 컴퓨터에서 소수가 비트로 어떻게 표현되는지에 대해 알아보았다. 특히 이번 랩에서 주어진 문제는 주어진 비트 연산, 논리 연산, if문, 반복문을 한정된 수 이내로 이용하여 int / float과 관련된 함수를 구현하는 것으로, 총 6 문제가 있었다. 이후 내용에서 이 문제들에 대해 상세히 서술할 것이다.

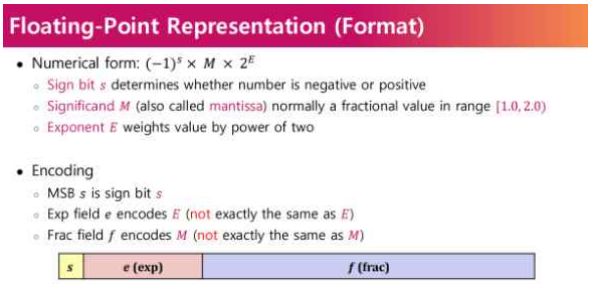

Figure 1. Floating-Point Format. [1]

2. Question #1. negate

2-1. Explanation.

- 주어진 int형 변수 x에 대해 -x를 반환하는 함수 negate를 구현해야 한다. 이때 !, ~, &, ^, |, +, <<, >> 연산을 최대 5번 사용할 수 있다.
- x의 2의 보수를 구하면, 그것이 -x가 된다. 이때 2의 보수를 구하는 것은 ~, + 연산만 사용하여 간단하게 구할 수 있다.

2-2. Solution.

- x의 2의 보수는 ~x + 1이며, 이는 -x와 같다. 이는 ~, + 연산을 2번만 사용한 것이다.

2-3. Implementation.

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

Listing 1. Code of negate.

- 앞선 2-2에서 설명한 바와 같이 -x 는 ~x + 1로 표현할 수 있으며, 이를 그대로 반환해주었다.

3. Question #2. isLess

3-1. Explanation.

- 주어진 int형 변수 x, y에 대해 x < y이라면 1, 그렇지 않으면 0을 반환하는 함수 isLess를 구현해야 한다. 이때 ! ,~, &, ^, |, +, <<, >> 연산을 최대 24번 사용할 수 있다.
- ^(XOR)을 사용하면 두 비트가 다른 경우 1, 같은 경우 0이 되도록 할 수 있다.
- int형 변수에서는 >> (right shift)를 할 경우 빈 비트가 부호 비트(1: 음수, 0: 음수가 아님)로 채워진다.

- y의 2의 보수를 사용하면 -y = ~y + 1이므로, x - y= x + ~y + 1으로 표현할 수 있다.
- 부호가 같은 경우 x + ~y를 계산하고, 1을 더해주면 y가 -2^{31}인 경우에도 overflow가 나지 않는다.

3-2. Solution.

- 3-1에서 보았던 ^와, 2의 보수를 활용하여 isLess 함수를 구현할 수 있다.
- x < y인 경우는 아래 설명할 두 가지 경우 밖에 없으므로, 이를 나누어 구현한다. 그냥 x - y를 계산하면 overflow가 발생하는 경우가 있기 때문에 (x와 y의 부호가 다른 경우), 이와 같이 나누어 구현하였다.
    - x, y의 부호가 다른 경우, x의 부호가 -일 때 x < y이다. 먼저 x ^ y를 하게 된다면, x와 y의 부호가 다를 경우 부호 비트가 1이 된다. 따라서 (x ^ y) >> 31을 계산하면, 두 수의 부호가 다른 경우 모든 비트가 1이, 같은 경우 모든 비트가 0이 된다. 또한 x >> 31을 계산하면 x가 음수일 때 모든 비트가 1로, 그렇지 않다면 모든 비트가 0이 된다. 이때 이 두 값을 &로 묶으면 두 경우를 모두 만족할 때 (즉, x < 0 <= y일 때) 모든 비트가 1, 그렇지 않으면 모든 비트가 0가 된다. 여기서 1과 0을 반환하기 위해서 ! 연산을 2번 진행하였다.
        - !!(((x ^ y) >> 31) & (x >> 31))
    - x, y의 부호가 같은 경우, x - y의 부호가 -일 때 x < y이다. 먼저 ~((x ^ y) >> 31)을 계산하면 앞의 경우와 반대되게 x와 y의 부호가 같은 경우 모든 비트가 1이, 다른 경우 모든 비트가 0이 된다. 또한 x - y = x + ~y + 1 = (x + ~y) + 1이므로 ((x + ~y) + 1) >> 31을 계산하면 x - y < 0인 경우 모든 비트가 1, 그렇지 않은 경우 모든 비트가 0이 된다. 이때 이 두 값을 &로 묶으면 두 경우를 모두 만족할 때 (즉, x < y일 때) 모든 비트가 1, 그렇지 않으면 모든 비트가 0이 된다. 여기서 1과 0을 반환하기 위해서 ! 연산을 2번 진행하였다.
        - !!((~((x ^ y) >> 31)) & (((x + ~y) + 1) >> 31))
    - 두 경우를 | 으로 묶으면 x < y인 경우 1, 그렇지 않은 경우 0이 계산되므로, 이를 그대로 반환하면 isLess 함수가 구현된다. 이는 !, ^, >>, &, ~, +를 17번 사용한 것이다.
        - (!!(((x ^ y) >> 31) & (x >> 31))) | (!!((~((x ^ y) >> 31)) & (((x + ~y) + 1) >> 31)))

3-3. Implementation.

```
int isLess(int x,int y){
    return (!!(((x ^ y) >> 31) & (x >> 31))) | (!!((~((x ^ y)
>> 31)) & (((x + ~y) + 1) >> 31)));
}
```

Listing 2. Code of isLess.

- 앞선 3-2에서 설명한 바와 같이 isLess 함수를 구현할 수 있다.

4. Question #3. float_abs

4-1. Explanation.

- 주어진 unsigned형 변수 uf의 비트들이 single-precision floating-point value로 해석되었을 때, 그것의 절댓값을 single-precision floating-point form으로 반환하는 함수 float_abs를 구현해야 한다. 이때 비트 연산, 논리 연산, if문, 반복문을 최대 10번 사용할 수 있다.
- single-precision floating-point는 32개의 비트로 구성되어 있으며, 맨 왼쪽부터 차례대로 1개의 비트는 부호(1이 음수), 그 다음 8개의 비트는 지수 (exponent), 나머지 23개의 비트는 가수 (mantissa, fraction)이다.
- 지수가 255(11111111)이고, 가수가 0이 아니라면 NaN이 된다.
- unsigned 형 변수에서는 >> (right shift)를 할 경우 빈 비트가 0으로 채워진다.
- 부호의 경우 맨 왼쪽 비트로, uf >> 31을 하면 uf의 부호값을 가지는 정수가 된다.
- 지수의 경우 왼쪽부터 2~9번째 비트이므로 uf << 1을 하면 1~8번째 비트가 된다. 여기서 >> 24를 하면 uf의 지수 값을 가지는 정수가 된다.
- 가수의 경우 왼쪽부터 10~32번째 비트이므로 uf << 9을 하면 1~23번째 비트가 된다. 여기서 >> 9를 하면 uf의 가수 값을 가지는 정수가 된다.

4-2. Solution.

- 4-1에서 보았던 NaN을 예외처리 해주고, unsigned에서의 >>와 <<를 이용하면 float_abs함수를 구현할 수 있다.
- 먼저 지수와 가수를 구해 unsigned형 변수 exp, f에 각각 저장해준다. 이는 4-1에서 설명한 방법을 사용한다.
    - exp = (uf << 1) >> 24
    - f = (uf << 9) >> 9
- 이후 NaN인 경우(if(exp == 255 && f != 0)) uf를 반환하도록 해준다.
- 그렇지 않은 경우엔 uf의 부호 비트를 제거하여 반환해준다. 이는 (uf << 1) >> 1로 구현할 수 있다.

4-3. Implementation.

```
unsigned float_abs(unsigned uf) {

    unsigned exp = (uf << 1) >> 24;
    unsigned f = (uf << 9) >> 9;

    if (exp == 255 && f != 0) return uf;
    else return (uf << 1) >> 1;
}
```

Listing 3. Code of float_abs.

- 앞선 4-2에서 설명한 바와 같이 float_abs를 구현할 수 있다. 이는 제한된 연산을 9번 사용한 것이다.

5. Question #4. float_twice

5-1. Explanation.

- 주어진 unsigned형 변수 uf의 비트들이 single-precision floating-point value로 해석되었을 때, 그것의 2배를 single-precision floating-point form으로 반환하는 함수 float_twice를 구현해야 한다. 이때 비트 연산, 논리 연산, if문, 반복문을 최대 30번 사용할 수 있다.
- 지수가 255인 경우엔 가수가 0이라면 infinite, 0이 아니라면 NaN이기 때문에 곱하기 2를 계산할 수 없다.

5-2. Solution.

- 먼저 부호, 지수, 가수를 구해 unsigned형 변수 s, exp, f에 각각 저장해준다. 이는 4-1에서 설명한 방법을 사용한다.
    - s = uf >> 31
    - exp = (uf << 1) >> 24
    - f = (uf << 9) >> 9
- 이후 exp(지수)이 255인 경우(if(exp == 255)) uf를 반환하도록 해준다.
- 또는 exp이 0인 경우(if(exp == 0)) 부호 비트 자리에 s를 넣고, 가수 자리에 f << 1을 넣어 반환한다. 이때 f << 1이 지수 자리를 침범하게 되어도, 원래 지수 자리가 0에서 1로 바뀌어야 하기 때문에 그대로 넣어도 상관 없다.
    - (s << 31) | (f << 1)
- 나머지 경우엔 부호 비트 자리에 s, 지수 자리에 exp + 1, 가수 자리에 f를 넣어 반환한다. 이때 exp이 254인 경우 2배를 하면 infinite가 되어야 하므로, 이 경우에만 가수 자리에 0을 넣어 반환한다.
    - (s << 31) | ((exp + 1) << 23) | (exp == 254 ? 0 : f)

5-3. Implementation.

```
unsigned float_twice(unsigned uf) {

    unsigned s = uf >> 31;
    unsigned exp = (uf << 1) >> 24;
    unsigned f = (uf << 9) >> 9;

    if (exp == 255) return uf;
    else if (exp == 0) return (s << 31) | (f << 1);
    else return (s << 31) | ((exp + 1) << 23) | (exp == 254 ? 0 : f);
}
```

Listing 4. Code of float_twice.

- 앞선 5-2에서 설명한 바와 같이 float_twice를 구현할 수 있다. 이는 제한된 연산을 16번 사용한 것이다.

6. Question #5. float_i2f

6-1. Explanation.

- 주어진 int형 변수 x를 single-precision floating-point form으로 변형하여 반환하는 함수 float_i2f를 구현해야 한다. 이때 비트 연산, 논리 연산, if문, 반복문을 최대 30번 사용할 수 있다.
- 가수를 계산할 때, 다음 규칙을 만족한다면 반올림을 한다.

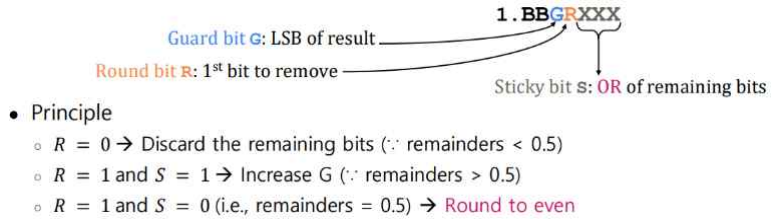

Figure . Rounding [2]

6-2. Solution.

- 먼저 x의 부호를 unsigned형 변수 s에 저장한다. 즉, x < 0일때 1, 그렇지 않을 때 0을 s에 저장한다.
    - s = x < 0
- 이후 unsigned형 변수 f (가수 저장), exp (지수 저장), ux (x를 unsigned 형으로 저장), round (반올림 여부 저장), check (맨 왼쪽 비트만 1인 수를 저장, x의 길이를 구할 때 사용)를 선언하고 check는 0x80000000로 초기화한다. int형 변수 length (x의 길이를 저장)를 선언하고 32로 초기화한다.
- x가 0인 경우엔(if(x == 0)) 그대로 0을 반환한다.
- 만약 s가 1이라면(if(s)), 즉 x가 음수라면, x를 -x로 바꾼다. 이때 x가 -2^{31}인 경우 그대로 -2^{31}가 되지만, unsigned에서는 2^{31}가 되므로 상관이 없다.
- 이후 ux에 x를 넣고, for문을 사용해 ux의 맨 왼쪽 비트가 1이 아닐때까지 length를 1씩 빼며 ux를 1씩 left shift한다. 이를 통해 x를 나타내는 비트의 길이를 구하여 length에 저장할 수 있다.
    - for(; length && !(ux & check); length--, ux <<= 1))
- 일단 f에 ux의 왼쪽에서부터 25개의 비트를 저장하기 위해 ux >> 7을 계산하고, 이를 저장한다. 이때 맨 왼쪽의 7개 비트를 제외했을 때 첫 번째 비트가 1이고(원래 가수 부분은 1.XX...를 나타내며, 반올림을 했을 때 10.00..이 될 수 있기 때문에 일단 없애지 않았다), 그 다음부터 23개의 비트가 실제 f값이 되게 될 것이다. 또한 맨 오른쪽에 있는 25번째 비트가 1일때 반올림이 진행될 수 있다.
    - f = ux >> 7
- 이후 round의 값을 구해준다. round가 1인 경우는 ux의 왼쪽 기준 26번째 비트부터 1이 존재하는 경우((ux << 25) != 0) 또는 f(가수)의 오른쪽 기준 2번째 비트가 1인 경우 ((f >> 1) & 1)이다.
    - round = ((ux << 25) != 0) | ((f >> 1) & 1)
- 이렇게 round에 할당된 값에 추가적으로 f(가수)의 오른쪽 기준 1번째 비트가 1이여야 하기 때문에 (f & 1)와 & 연산하여 저장한다.
    - round &= (f & 1)
- 그리고 f에 round를 더하면 반올림이 되는 경우 1이 더해지고, 그렇지 않은 경우 0이 더해진다.
    - f += round

- f의 맨 왼쪽의 7개 비트를 제외한 맨 왼쪽 비트와, 맨 오른쪽 비트를 없애주기 위해 (f << 8) >> 9 를 계산하여 f에 저장한다.
    - f = (f << 8) >> 9
- 이후 exp에 127 + (length - 1), 즉 126 + length를 저장한다.[3] 이때 127은 BIAS이며, length - 1을 더하는 이유는 주어진 x를 1.XX... * $2^{length - 1}$로 표현하여 저장하기 때문이다.
    - exp = 126 + length
- 마지막으로 s를 부호 비트 자리, f를 가수 부분에 넣으며, 지수 부분에는 exp를 넣되 f가 1...1일 때 반올림이 진행된 경우(즉, round가 1이고 현재 f가 0인 경우, (round & !f))에는 exp + 1 넣어 반환하면 float_i2f가 구현된다.
    - (s << 31) | ((exp + (round & !f)) << 23) | f

6-3. Implementation.

```
unsigned float_i2f(int x) {

    unsigned s = x < 0;
    unsigned f, exp, ux, round;
    unsigned check = 0x80000000u;
    int length = 32;

    if (x == 0) return 0;

    if (s) x = -x;

    ux = x;
    for (; length && !(ux & check); length--, ux <<= 1);

    f = ux >> 7;
    round = ((ux << 25) != 0) | ((f >> 1) & 1);
    round &= (f & 1);
    f += round;
    f = (f << 8) >> 9;

    exp = 126 + length;

    return (s << 31) | ((exp + (round & !f)) << 23) | f;
}
```

Listing 5. Code of float_i2f.

- 앞선 6-2에서 설명한 바와 같이 float_i2f를 구현할 수 있다. 이는 제한된 연산을 27번 사용한 것이다.

7. Question #6. float_f2i

7-1. Explanation.

- 주어진 unsigned형 변수 uf의 비트들이 single-precision floating-point value로 해석되었을 때, 그 값을 int형을 구해 반환하는 함수 float_f2i를 구현해야 한다. 이때 비트 연산, 논리 연산, if문, 반복문을 최대 30번 사용할 수 있다.

- 실제로 float 값을 계산할때 2의 지수로 사용하는 수를 E라고 하자. 이때 E = 지수 - BIAS이다. float의 경우에 지수가 8비트이므로 BIAS는 127이다. [3]
- float to int는 소수점 아래 수를 버림한다.

7-2. Solution.

- 먼저 부호, 지수, 가수를 구해 unsigned형 변수 s, exp, f에 각각 저장해준다. 이는 4-1에서 설명한 방법을 사용한다.
    - s = uf >> 31
    - exp = (uf << 1) >> 24
    - f = (uf << 9) >> 9
- 이후 exp(지수)이 157보다 큰 경우(if(exp > 157)) int형의 범위를 벗어나므로 0x80000000u를 반환한다. 157 = 127 + 30으로, 이는 ±1.XX...에 2^{31}이상을 곱하면 int형의 범위를 벗어나기 때문이다. 이때 -2^{31}은 원래의 반환값과 같으므로 이를 포함해도 된다.
- float to int는 소수점 아래 수를 버림하므로 exp(지수)이 127보다 작은 경우(if(exp < 127)) 0.XX... 이므로 0을 반환한다.
- 나머지 경우엔 left, right shift를 적절히 사용하여 반환할 값을 계산해준다.
    - f가 나타내는 값은 1.XX.. 이므로 맨 오른쪽부터 24번째 비트를 1로 만들어준다.
        - f |= (1 << 23)
    - 이후 반환할 값을 저장할 변수인 i에는 f와 exp, 127(BIAS)를 이용하여 계산한 결과를 넣는다.
    - 이때 exp의 크기에 따라 f를 left shift / right shift 해준다. exp가 150(127 + 23)보다 큰 경우 (if(exp > 150)) f << (exp - 150)를 i에 저장한다. 나머지 경우 f >> (150 - exp)을 i에 저장한다. 이는 f에 2^{-23}을 곱한 것이 가수가 나타내는 원래 수이기 때문이다. 이렇게 경우를 나누어 계산하지 않는다면 shift하면서 비트가 소실될 수 있다.
- 이후 s가 1인 경우(if(s == 1)) i에 -i를 저장한다.
- 마지막으로 이렇게 계산된 i를 반환하면 float_f2i가 구현된다.

7-3. Implementation.

```
int float_f2i(unsigned uf) {

    unsigned s = uf >> 31;
    unsigned exp = (uf << 1) >> 24;
    unsigned f = (uf << 9) >> 9;
    int i;

    if (exp > 157) return 0x80000000u;
    if (exp < 127) return 0;

    f |= (1 << 23);
    if (exp > 150) i = f << (exp - 150);
    else i = f >> (150 - exp);

    if (s == 1) i = -i;

    return i;
}
```

Listing 6. Code of float_f2i.

- 앞선 7-2에서 설명한 바와 같이 float_f2i를 구현할 수 있다. 이는 제한된 연산을 16번 사용한 것이다.

8. Results

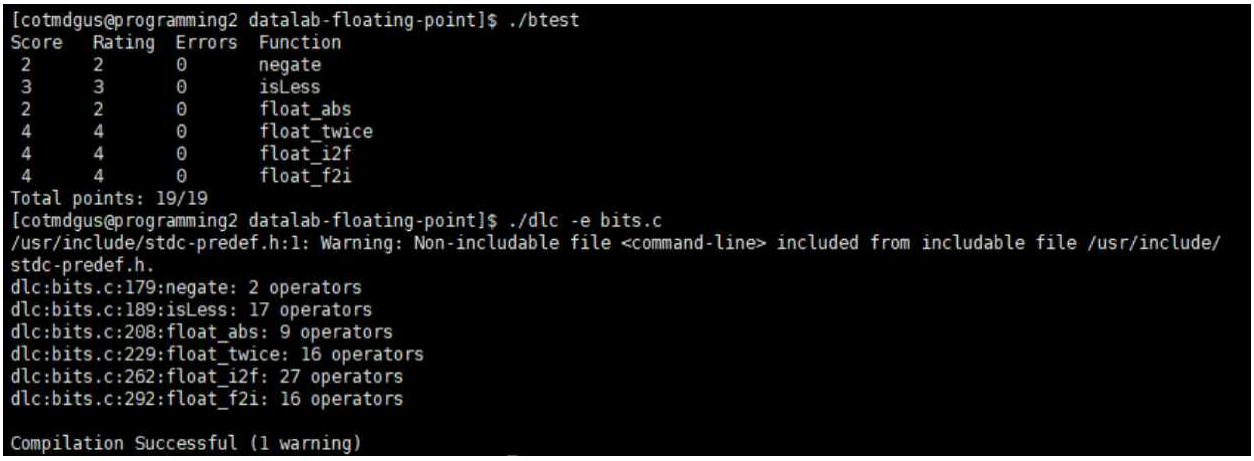

Figure 3. Output of the grading program btest and programming rule checking program dlc.

Figure 3에서 확인할 수 있듯이 프로그래밍 규칙을 어기지 않으며, btest의 결과로 만점인 19점을 얻었다. dlc의 결과로 나오는 경고는 내가 건드리지 않았던 부분에서 발생하였기 때문에 무시하였다.

9. References

[1] Lecture Note of CSED211, “[CSED211 2025 Fall] Lecture 3 Floating Point”, 7p

[2] Lecture Note of CSED211, “[CSED211 2025 Fall] Lecture 3 Floating Point”, 31p

[3] Lecture Note of CSED211, “[CSED211 2025 Fall] Lecture 3 Floating Point”, 9p