Lab 2

20200437 김채현

Problem 1

```
unsigned float_neg(unsigned uf) {
    // frac part is 111111...11
     int frac_mask = (1 << 24) - 1;

    //exp part is 11111111
    int exp_mask = 0xFF << 23;

    //if it's NaN, return uf
    if (((uf & exp_mask) == exp_mask) && (uf & frac_mask))
         return uf;

    //if it's not NaN, change sign bit
    return uf ^ (1 << 31);
}
```

본 문제는 floating point 변수 부호를 바꾸는 함수를 구현하는 문제이다. 이는 쉽게 sign bit만 바꿔주면 된다. 하지만 문제는 NaN 일 경우 argument uf를 다시 return 해주어야 한다. NaN은 exp = 111...1, frac != 000...0인 경우이다. frac 부분만 다 1인 frac_mask와 exp 부분만 다 1인 exp_mask를 만들어주었다. uf가 exp 부분이 모두 1이고, frac 부분이 0이 아닌 경우 NaN이 1이 되게 하였고, NaN이 1이면 uf를 return한다. NaN이 아닌 경우 sign bit를 바꿔주면 되므로 1000...000과 XOR 연산을 해준 결과값을 return한다.

Problem 2

```
unsigned float_i2f(int x) {

    //temporary variance for s, exp, frac
    int s_temp = 0;
    int exp_temp = 158;
    int frac_temp = x;

    // x is 000...00
    if (x == 0)
        return 0;

    // x is 1000...00
    int MSB = 1 << 31;
    if (x == MSB)
        return (MSB | (exp_temp << 23));
```

본 문제는 Two's complement type integer를 bit-level에서 floating point로 바꾸는 함수를 구현하는 문제이다. 먼저, x가 0일 때는 0을, x가 MSB 1을 제외한 모든 수가 0일 때 (1000...00)일 때는 1 11111111 000...00을 return 하도록 예외처리를 해주었다.

```
    // even if x is negative, change to positive
    if (x < 0) {
        s_temp = 1;
        frac_temp = -x;
    }

    // calculate for exp and frac
    while (!(frac_temp & MSB)) {
        frac_temp <<= 1;
        exp_temp -= 1;
    }

    //find s, exp, frac
    int s = 0;
    if (s_temp == 1)
        s = MSB;
    int exp = exp_temp << 23;
    int frac = (frac_temp & ~(1 << 31)) >> 8;

    //rounding
    int rounding = frac_temp & ((1 << 8) - 1);
    if ((rounding > 128) || ((rounding == 128) && (frac & 1)))
        frac += 1;

    //result
    return s + exp + frac;
}
```

x가 negative인 경우 sign bit만 1로 표시해주고 positive와 계산이 똑같으므로 s_temp에 1을 저장해주고, 계산을 위한 frac_temp에는 x를 양수로 부호를 고친 -x를 저장해준다.

반복문을 이용하여 normalize 과정을 진행한다. 가장 왼쪽에 있는 1 bit가 MSB에 올 때까지 shift해주고, 한 번 shift를 할 때마다 exp를 최댓값 158에서 1씩 줄여나갔다. (exp의 최댓값은 Exp = E + Bias = 31 + 127 = 158이다.) 이후 s, exp, frac값을 구하는데, s_temp가 0이면 s에 0을, s_temp가 1이면 s에 100...00을 저장한다. exp는 exp_temp 값에서 23만큼 left shift해서 exp 부분에 오도록 해서 저장한다. frac은 frac_temp의 2~24번째 bit까지를 저장해준다.

frac 계산에서는 rounding 처리를 해주어야 하는데 이를 위해 frac_temp의 25~32번째 bit를 저장하는 변수 rounding을 선언해주었다. rounding은 0~255 사이의 값을 가지기 때문에 medium이 128이다. rounding이 128보다 크거나, 128인데 frac 부분의 마지막 bit가 1인 경우 rounding up을 해준다. 이 외에는 다 rounding off를 하면 된다.

마지막 rounding 계산을 해준 후, s와 exp와 frac를 다 합친 결과 값을 return 해주면 된다.

Problem 3

```
unsigned float_twice(unsigned uf) {
    //for NaN
    int exp_mask = 0xFF << 23;
    int frac_mask = (1 << 24) - 1;
    if (((uf & exp_mask) == exp_mask) && (uf & frac_mask))
        return uf;

    //for infinity, -infinity, 0, -0
    if ((uf & exp_mask) == exp_mask || (uf == 0) || (uf == (1 << 31)))
        return uf;

    //for denormalized numbers
    if ((uf & exp_mask) == 0)
        return uf + (uf & frac_mask);

    //for normalized numbers
    return uf + (1 << 23);
}
```

본 문제는 bit level에서 주어진 float value를 두 배 하여 반환하는 함수를 구현하는 문제이다. 먼저 NaN, 즉 exp = 111...1, frac != 000...0인 경우에 대해 uf를 반환하는 예외 처리를 해준다. infinity, -infinity, 0, -0, 즉 exp = 111...1인 경우, 0000...0, 1000...0인 경우는 결과값이 uf와 동일하게 나오므로 uf를 반환해준다. denormalized number 같은 경우 frac 부분을 2배 해주면 된다. 하지만 frac에서 overflow가 발생하는 경우나 sign bit가 1인 경우 계산이 복잡해지기 때문에 uf에다 uf의 frac 부분을(uf&frac_mask를 해주면 된다.) 한 번 더 더해주는 방법을 택하였다.

예외 외의 그냥 normalized numbers에 대해서는 2를 곱해주면 E가 1이 커지므로 exp 부분에 1을 더해주는 방법을 사용하면 된다. 따라서 uf에 0 00000001 000....0을 더해주면 된다.

Problem 4

```c
unsigned float_abs(unsigned uf) {
    //for NaN
    int exp_mask = 0xFF << 23;
    int frac_mask = (1 << 24) - 1;
    if (((uf & exp_mask) == exp_mask) && (uf & frac_mask))
        return uf;

    //when sign bit is 0, f is positive
    if ((uf >> 31) == 0)
        return uf;

    //when sign bit is 1, f is negative. so, return -f
    if ((uf >> 31) & 1)
        return uf ^ (1 << 31);
}
```

본 문제는 bit level에서 주어진 float value의 절대값을 반환하는 함수를 구현하는 문제이다. 먼저 NaN에 대해서는 problem 3과 동일한 예외처리를 해준다. sign bit가 0인 경우 positive이므로 그대로 uf를 return하면 된다. sign bit가 1인 경우 sign bit를 뒤집어서 return 하면 된다. 사실상 problem 1과 비슷한 문제이다.

Problem 5

```c
unsigned float_half(unsigned uf) {
    //for NaN
    int exp_mask = 0xFF << 23;
    int frac_mask = (1 << 24) - 1;
    if (((uf & exp_mask) == exp_mask) && (uf & frac_mask))
        return uf;

    //for infinity, -infinity, 0, -0
    if ((uf & exp_mask) == exp_mask || (uf == 0) || (uf == (1 << 31)))
        return uf;

    //for denormalized numbers
    if ((uf & exp_mask) == 0)
        return uf >> 1;

    //for normalized numbers
    return uf - (1 << 23);
}
```

본 문제는 bit level에서 주어진 float value를 0.5배 하여 반환하는 함수를 구현하는 문제이다. 먼저 NaN, 즉 exp = 111...1, frac != 000...0인 경우에 대해 uf를 반환하는 예외 처리를 해준다. infinity, -infinity, 0, -0, 즉 exp = 111...1인 경우, 0000...0, 1000...0인 경우는 결과값이 uf와 동일하게 나오므로 uf를 반환해준다. denormalized number 같은 경우 frac 부분을 0.5배하면 된다. right logical shift 해주면 되는데, c언어 상에서 right shift는 arithmetic shift를 의미하므로 sign bit가 1인 경우

계산이 복잡해진다. 따라서 uf에다 uf의 frac 부분만을 살린(uf&frac_mask를 해주면 된다.) 수를 1회 right shift 해준 후 빼는 방법을 택하였다.

예외 외의 그냥 normalized numbers에 대해서는 2를 나눠주면 E가 1이 작아지므로 exp 부분에 1을 빼 주는 방법을 사용하면 된다. 따라서 uf에 0 00000001 000....0을 빼 주면 된다.