2020 시스템 프로그래밍
- Lab 03. DataLab -

제출일자	2020. 10. 06
분 반	02
이 름	전병민
학 번	201702068

1	bitOr(int x, int y)

```
int bitOr(int x, int y) {
    return ~(~x & ~y); //    ~( ~( x | y ) )  =  ~( ~x & ~y )
}
```

드 모르간의 법칙을 이용하면 ~ (x | y) = (~x & ~y) 이 된다. 여기서 다시 (x | y)을 만들어주기 위해서 양변에 ~(not)을 해주면 ~ (~ (x | y)) = ~(~x & ~y) 이 된다.

2	tmin(void)

```
int tmin(void) {
    int a = 1;
    // 00000000 00000000 00000000 00000001  =  1
    // 10000000 00000000 00000000 00000000  =  -2,147,483,648
    return a << 31;
}
```

32비트 Signed 정수에서 가장 작은 값은 -2,147,483,648이며 이는 **이진수에서 부호 비트인 MSB가 1이고 나머지는 모두 0인 수**이다. 따라서 1을 왼쪽으로 31만큼 쉬프트 해주었다.

3	evenBits(void)

```
int evenBits(void) {
    // 01010101 01010101 01010101 01010101
    //    5   5    5   5    5   5    5   5
    // -->   0x55 55 55 55

    int a = 0x55; // 0번 바이트 0x 00 00 00 55
    int b = 0x55; // 1번 바이트
    int c = 0x55; // 2번 바이트
    int d = 0x55; // 3번 바이트

    // 8비트씩 더해가며 left shift
    b = b << 8;  // 0x 00 00 55 00
    c = c << 16; // 0x 00 55 00 00
    d = d << 24; // 0x 55 00 00 00

    return a|b|c|d;
}
```

01010101 01010101 01010101 01010101 인 비트를 리턴하는 것이다. **이것을 16진수로 표현하면 0x55555555이다.** 문제 특성상 0x55씩 끊어서 합쳐야 0x55555555을 만들 수 있다. 4개의 0x55를 각각 왼쪽으로 0, 1, 2, 3바이트 쉬프트하고 OR를 이용하여 0x55555555을 만들어 리턴하였다.

4	getByte(int x, int n)

```
int getByte(int x, int n) {

    int a = n << 3; // a = n * 2^3
    int temp = 0x000000ff; // 맨 끝자리 빼고 다 0으로 만들어주기 위해서

    x = x >> a; // n바이트만큼 오른쪽 쉬프트

    return temp & x; // 실질적으로 값을 만들어줌
}
```

n << 3 을 이용하여 n * 2^3, 즉n바이트를 만들어주었다. 하나의 바이트만 리턴해주기 위해서 0x000000ff 를 가지는 temp를 만들어주었고 x를 n바이트만큼 오른쪽으로 쉬프트하고 최종적으로 temp와 &하여 n번째 바이트에 있는 값을 리턴하였다.

5	float_abs(unsigned uf)

```
unsigned float_abs(unsigned uf) {
    int answer = uf;
    int s, e, f, temp, temp2;

    // 부호비트가 1이라면 0으로 바꿔준다.
    s = 0x80000000; // s = 1 00000000 00000000000000000000000
    if((s & uf) == s){
        answer = uf - s;
    }

    // When argument is NaN, return argument
    // NaN의 조건 : exp = 11...11   frac != 0

    e = 0x7f800000; // exp = 111...1 --> 0 11111111 00000000000000000000000
    f = 0x007fffff; // frac != 0     --> 0 00000000 11111111111111111111111

    temp = e & uf; // uf의 exp비트가 11111111이라면 temp = 0x7f800000 이다
    temp2 = f & uf; // uf의 frac비트가 0이 아니라면 temp2에 0이 아닌 값이 들어갈 것

    if((temp == e) && temp2){ // when argument is NaN
        //return argument
        return uf;
    }

    // NaN의 조건을 충족하지 않는다면 정상적으로 answer을 리턴함
    return answer;
}
```

부호 비트만 0으로 만들어 리턴하면 된다. 하지만 NaN일 때에는 argument를 리턴해야한다. NaN일 때의 조건은 exp비트가 전부 1이어야 하고, frac비트가 0이 아니어야 한다. 이를 구하는 과정은 주석으로 작성하였다.

6	addOK(int x, int y)

```
int addOK(int x, int y) {

    int a = x;
    int b = y;
    int c = x + y;

    a = a >> 31; // 최상위비트 1
    b = b >> 31; // 최상위비트 2
    c = c >> 31; // 두 수의 합의 최상위비트

    // 0x00000001과 &하여 하나의 비트만 남도록 바꿔줌.
    a = a & 1;
    b = b & 1;
    c = c & 1;

    return (a ^ b) | !((a & b) ^ c);
```

최상위 비트가 다를 땐 항상 overflow가 발생하지 않으며, **두 개의 최상위 비트가 같을 때 두 수를 더한 값의 최상위 비트와 비교하여 다르면 overflow가 발생한다.**

a, b는 두 수의 최상위 비트이며, c는 두 수를 더한 값의 최상위 비트이다. (a ^ b)을 이용하여 최상위 비트가 다를 때 무조건 1을 리턴하도록 하였다. 하지만 최상위 비트가 같을 때에는 !((a & b) ^ c)을 이용하여 두 수를 더한 최상위비트와 비교하여 다르면 0을 리턴하고 같으면 1을 리턴하도록 하였다. 즉, 다음과 같다. (최상위비트가 같을 때와 다를 때 모두 판단해주기 위해서 OR를 사용하였다.)

```
  // !((a & b) ^ c) :
  // a != b 일 때에는 무조건 1을 리턴하기 때문에 이 연산에서 신경쓰지 않아도 됨
  // a = 1, b = 1, c = 1일 때 --> 1을 리턴
  // a = 1, b = 1, c = 0일 때 --> 0을 리턴
  // a = 0, b = 0, c = 1일 때 --> 0을 리턴
  // a = 0, b = 0, c = 0일 때 --> 1을 리턴
```

7	replaceByte(int x, int n, int c)

```
int replaceByte(int x, int n, int c) {

    int a = n << 3; // 왼쪽으로 쉬프트 --> a = n * 2^3 (n바이트)
    int b = 0x000000ff;
    int temp, temp2;

    temp = x >> a; // n바이트 만큼 오른쪽 쉬프트
    temp = b & temp; // 0x000000XX 형태로 만듬
    temp = temp << a; // 0x000000XX를 n바이트 즉, 같은 숫자가 있는 위치까지 쉬프트

    c = c << a; // 바꿔줄 바이트를 똑같이 n바이트만큼 왼쪽으로 쉬프트
    temp2 = x^temp; // 같으면 0, 다르면 1이기 때문에 ^로 해당 위치를 0으로 만들어줌

    return temp2+c; // 그것을 바꿔줄 바이트와 더하면 바이트를 replace하게 됨.
}
```

getByte와 유사하다. 하지만 이 함수에서는 교체해주어야 할 바이트를 0으로 만들고 교체할 바이트를 더하여 리턴해주었다.

getByte에서와 같이 a에 n * 2^3 을 저장해주었다.(바이트 단위로 x를 쉬프트하기 위해서 2^3을 곱해주는 것) temp에 x를 오른쪽으로 a만큼 쉬프트해준 값을 넣고, 0x000000ff와 &하여 하나의 바이트만 남겨주었고, 다시 왼쪽으로 a만큼 쉬프트해주어 교체할 대상을 찾았다. **x ^ temp를 이용하여 그 자리를 0으로 바꿔주었다.**

c를 0으로 바뀐 자리에 넣어주기 위해서 왼쪽으로 a만큼 쉬프트해주었다. +연산으로 c를 넣어주고 리턴하였다.

8	isNonZero(int x)

```
int isNonZero(int x) {

    // 0은 음수도 양수도 아닌 수
    // x의 2의 보수는 ~x+1
    // 0이라면 ~x+1이 다시 0값을 갖게됨
    // 0 | 0 = 0
    // x | ~x+1 = 1.....

    int x2 = ~x+1; // x가 0이라면 x2는 0값을 가질 것.
    int answer = x | x2; // answer는 0 아니면 음수인 어떤 값을 가짐.

    // 즉, x가 0이면 MSB가 0
    //     x가 0이 아니라면 MSB가 1이 된다.

    answer = answer >> 31 & 1; // 오른쪽으로 31비트만큼 논리쉬프트

    return answer;
}
```

정수에서 0은 음수도 양수도 아닌 수이다. 즉, 0의 2의 보수도 0이다. 이 특성을 이용하여 함수를 작성하였다.

x가 양수나 음수라면, 2의 보수를 취한 값과 OR연산을 해주면 무조건 최상위비트가 1이다. 이것을 쉬프트 연산으로 최하위 비트로 옮겨주면 1을 리턴할 수 있게 된다.

x가 0이라면, 0의 2의 보수도 0이기 때문에 결론적으로 0 | 0 와 같은 연산을 하게 되어 결국 0이 되어 쉬프트 연산 후 0을 리턴할 수 있게 된다.