2018 시스템 프로그래밍
- Lab 03 -

제출일자	2018.10.14
분 반	01
이 름	이규봉
학 번	201502085

실습 1 [bits.c 구현] -> bitAnd(int x, int y)

```
int bitAnd(int x, int y) {
  return ~(~x|~y);
}
```

위 문장은 bitAnd의 구현 문장입니다. 드모르간의 법칙에 의해 x&y를 ~(~x|~y)로 바꾸어도 성립합니다.

실습 2 [bits.c 구현] -> getByte(int x, int n)

처음엔 ~(~1<<8*n) 식으로 0000 0000 0000 0000 0000 0000 1111 1111를 구현하려 했으나, 연산자 없이 상수로 0xFF라고 쓰면 충분하다는 것을 알게 되었습니다. 이 값과 x를 8*n번 Shift한 값을 &하면 됩니다. 다만, * 연산자를 사용할 수 없으므로, << 연산자를 쓰면 됩니다. 즉, 아래처럼 쓸 수 있습니다.

```
int getByte(int x, int n) {
        return (x>>(n<<3)) & (0xFF);
}
```

실습 3 [bits.c 구현] -> logicalShift(int x, int n)

```
int logicalShift(int x, int n) {
        int arithmetic = (x>>n);
        // int mask = (1 << 33+~n) +(~0); ①
        // int mask = 0xFFFFFFFF >> n; ②
        // int mask = (~0) >> n; ③
        // int mask = ((1 << 31+~n) << 2) +(~0); ④
        int mask = ~(((1 << 31) >> n) << 1);
        return (arithmetic & mask);
}
```

예제에서 주어진 값을 산술쉬프트 연산에 넣으면, 2^31 값을 넘는 범위이기 때문에, 오버플로우 되어 0xf8765432가 출력됩니다. unsingned로 캐스팅하면, 논리쉬프트 연산과 같은 결과를 낼 수 있습니다. (0x08765432를 출력) 이 실습에선 데이터 타입 변환 없이 논리 쉬프트를 구현하기 위해, 위처럼 작성했습니다. (32가 아닌 33인 이유는 2의 보수 형태인 32+~n+1에서 연산자를 줄이기 위해 +1을 합쳐 썼기 때문입니다.)

mask를 ①번 주석처럼 구현했다가 n=0에서 작동하지 않아 (mask가 0이 되어, 0이 리턴됨) ②번 코드로 바꾸었더니 예상과 다르게 코드가 작동했습니다. 그런데 btest에서 0xFFFFFFFF이 허용되지 않는 상수라고 나와서, ③처럼 바꾸어 보았습니다. 그런데, (0xFFFFFFFF)==(~0)가 1값을 출력하는데도, 리턴값은 ②와 다르게 0xf8765432 였습니다.

C언어는 0xFFFFFFFF에 대해선, 논리시프트를 적용한다는 것을 알게 되었습니다.

그 후엔 mask에 대한 코드는, 위처럼 ①번 코드에서 2를 따로 빼줘서 1이 사라지지 않게 작성 했습니다. (④)

그러나, 이번엔 n 값이 31일 때, 31+~n은 –1이기 때문에 1<<-1는 0x2로 계산되어, 기대했던 값과 다른 값을 얻게 되었습니다. 따라서, 31+~n라는 표현을 ((1<<31) >>n) <<1로 바꿈으로써, 최종적으로 코드를 작성했습니다.
또한, -1을 더해 011..11을 만들던 표현이, 기대대로 작동하지 않아 ~로 바꿨습니다.

실습 4 [bits.c 구현] -> bitCount(int x)

BitCount를 구현하는 방법으로 가장 먼저 생각한 것은, x의 각 자리 비트들을 모두 1의 자리로 Shift 시켜 1과 & 한 값들(LSB) 을 더하는 것입니다. 코드로 쓰면 아래와 같습니다.

```
int bitCount(int x) {
        int sum = 0;
        sum = sum + ((x) & 1);
        sum = sum + ((x>>1) & 1);
        sum = sum + ((x>>2) & 1);
        ...
        sum = sum + ((x>>31) & 1);
        return sum;
}
```

그러나 위 코드는 연산자의 개수가 Max ops를 한참 초과하므로, 연산자의 갯수를 줄일 다른 방법을 생각했습니다.

&1 이라는 코드가 들어가게 되면 반드시 >> 연산자가 32번 들어가야 합니다. 따라서, &1 부분을 다른 값으로

고쳐야 연산자의 개수를 줄일 수 있습니다. 만약, 1을 0000 0001 0000 0001 0000 0001 0000 0001 라고

고쳐 생각해보면, 한 번에 여러 sum을 실행할 수 있을 것이고 연산자의 개수가 줄어들 것입니다.

```
int bitCount(int x) {

        int sum = 0;
        int t = 0x01010101;
        sum = sum + ((x) & t);
        sum = sum + ((x>>1) & t);
        sum = sum + ((x>>2) & t);
        sum = sum + ((x>>3) & t);
        sum = sum + ((x>>4) & t);
        sum = sum + ((x>>5) & t);
        sum = sum + ((x>>6) & t);
        sum = sum + ((x>>7) & t);
        return sum;
}
```

위 함수로 0b11110010010100101011011101110를 대입해보면, 0b111,10010010,10010110,11101110 이기 때문에 0x3030406이 나옵니다. 이 리턴값에서 0이 아닌 수들, 즉 3+3+4+6 = 16이 구하는 1의 개수가 됩니다.

즉, 구하는 값은 16진수 sum의 각 자리수들을 합한 값입니다. 그런데 t는 0000 0001이 반복되므로, 짝수 번째 자릿수의 값만 합쳐 주면 됩니다.

따라서 아래처럼 쓸 수 있습니다.

```
int bitCount(int x) {

        int sum = 0;
        int result = 0;
        // int t = 0x01010101;
        int t = 0x01;
        t = t | (t<<8) | (t<<16) | (t<<24);

        sum = sum + ((x) & t);
        sum = sum + ((x>>1) & t);
        sum = sum + ((x>>2) & t);
        sum = sum + ((x>>3) & t);
        sum = sum + ((x>>4) & t);
        sum = sum + ((x>>5) & t);
        sum = sum + ((x>>6) & t);
        sum = sum + ((x>>7) & t);

        result = sum & 0xFF;
        result = result + ((sum>>8) & 0xFF);
        result = result + ((sum>>16) & 0xFF);
        result = result + ((sum>>24) & 0xFF);

        return result;
}
```

그런데, btest를 실행해보니, t가 허용되지 않는 상수로 나와 위처럼 << 연산자를 이용해 코드 (t)를 수정했습니다. 대입연산자를 제외하고 총 39개의 제한된 연산자를 사용했습니다.

실습 5 [bits.c 구현] -> isZero(int x)

```
int isZero(int x) {
  return !(0^x);
}
```

실습 6에서 y를 0으로 바꾸면 간단하게 얻어집니다.

실습 6 [bits.c 구현] -> isEqual(int x, int y)

```
int isEqual(int x, int y) {
  // return !((x|y)^x) & !((x|y)^y);
  return !(x^y);
}
```

x와 y가 다르다면 반드시 한 비트 이상에서 다른 값을 가져야 합니다. x|y와 x나 y의 ^를 했을 때, 그 값들이 모두 0이라면, x와 y는 완전히 같은 자리에 1을 가지게 되기 때문에, !((x|y)^x) & !((x|y)^y)로 생각했습니다. 다만, 연산자의 개수가 Max ops를 초과하기 때문에, 계속 생각해보니 (x^y)가 x와 y가 다른 비트를 하나 이상 포함하면 항상 1이상의 값을 출력한다는 것에서, !(x^y)로 축약해 썼습니다. (결론적으로, 실습5와 비슷한 형태가 되었습니다.)

실습 7 [bits.c 구현] -> fitsBits(int x, int n)

```
int fitsBits(int x, int n) {
    int plus = 1 << n+(~0);
    int minus = ~0 << n+(~0);

    return ((x+ (~plus+1) >> 31) &1) ^ ((x+ (~minus+1) >> 31) &1);
}
```

x값이 (–2^n-1)과 (2^n-1)-1의 범위에 있을 때 1을 리턴하는 함수를 작성하면 됩니다. 1을 빼는 연산은 ~0을 더하는 연산으로 구현할 수 있으며, 변수 plus와 minus는 2의 n-1 제곱 값이 됩니다. 작거나 같다는 연산은 실습8을 참고해보았습니다.

그러나 x가 –2147483648일 때, x+( ) 부분에서 오버플로우 에러가 나기 때문에, 잘못된 값을 리턴하게 되었습니다. 따라서, 코드를 아래와 같이 고쳐 썼습니다. 고쳐쓰는 과정에서 원래 참고했던 실습8이 길어지고, 연산자가 Max ops를 초과하여 다른 코드를 짜야겠다고 생각하게 되었습니다.

크기 비교가 아닌 비트들의 배열을 보면, n = 4 일 때, 마지막 네 비트는 0111, 0110, 0101, 0100, 0010 ... 이 1을 리턴하는 양의 x값입니다. 따라서, n번 >> 시프트 연산을 해, 값이 0인지 확인하는 것으로 구현할 수 있을거라 생각했지만, 0001 0111 같은 경우가 더 많기 때문에 다른 방법을 찾았습니다.

입력된 x값이 1을 리턴할 x값이라면, << 연산을 32-n번 시행한 후, 다시 >>를 32-n번 시행했을 때, 원래의 값과 동일해야 합니다. 이 때 32-n을 33+(~n)으로 쓸 수 있으며, 선조건에서 n>0이므로, 오버플로우는 고려하지 않아도 됩니다. (일어나지 않습니다.)

따라서, 실습6을 참고하여 아래처럼 쓸 수 있습니다.

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

    return !(x^( (x << 33+(~n)) >> 33+(~n)));

}
```

실습 8 [bits.c 구현] -> isLessOrEqual(int x, int y)

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

int는 8바이트, 즉 32비트이고, MSB를 Sign으로 사용하므로, -(2^31)부터 (2^31)-1 까지의 표현 범위를 갖습니다. x-y의 Sign 비트를 확인해 1 (음수)이라면 y가 큰 경우이고, 0(양수) 라면, x가 큰 경우일 것입니다. 그런데 – 연산자는 사용할 수 없으므로, y에 ~를 사용하고 2의 보수로 만들기 위해 +1을 해주면 – 연산자가 하는 기능과 동일하게 구현할 수 있습니다. 이 때, MSB (Sign bit)를 확인하기 위해선, x-y를 31번 >> 쉬프트 해 주면 됩니다.

그런데, btest를 실행해보니 x와 y가 각각 (2^31)-1, -2^31가 될 때, 오버플로우로 인한 Error가 발생했습니다. x는 –2147483648값을, y는 ~y+1에서, -2147483647를 가지고, 이 둘을 더해, 1로 계산하므로 MSB는 0입니다. 즉, 1을 리턴해야 할 상황에서 0을 리턴하게 됩니다.

그래서, x와 y가 서로 같은 MSB 값을 가질 땐, 위의 논리대로 리턴값을 반환하고, 서로 다른 MSB 값을 가질 땐, x가 음수이고 y가 양수일 때만 1을 반환하도록 코드를 수정했습니다. (반대의 경우엔 항상 0을 리턴하면 되므로)

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

        int MSBofx = (x >> 31) & 1;
        int MSBofy = (y >> 31) & 1;


        return (MSBofx&!(MSBofy))| (!(MSBofx ^ MSBofy) & (((x + (~y + 1))>>31) & 1))| !(x^y);

}
```

실습 9 [bits.c 구현] -> rotateLeft(int x, int n)

저는 rotateLeft에 대해 0000 0000 0000 0000 0100 0011 0010 0001을 n=4로 rotateLeft 했을 때, (0x4321이 입력일 때)

0000 0000 0000 0000 0001 0100 0011 0010가 리턴되는 경우와 (0x1432를 리턴)

0000 0000 0000 0100 0011 0010 0001 0000가 리턴되는 경우가 있을 수 있다고 생각했습니다. (0x4320를 리턴)

그런데, 전자의 경우 주어진 bitwise 연산만으로 구현하는 것은 불가능하다고 생각했고, 따라서 후자로 생각해봤습니다.

후자의 코드의 경우, 일단 x를 n번 Left시키고 (x<<n), x를 32-n번 RightShift 시킨 것 (비어 있는 n만큼의 자리를 앞의 비트로 채우면 되므로) 을 OR연산 시켜 구현할 수 있습니다. 여기에 x에 unsigned를 붙이면 코드를 아래처럼 쓸 수 있습니다.

```
int rotateLeft(int x, int n) {
        return (((unsigned)x << n) | ((unsigned)x >> (32 - n)));
}
```

그런데 btest를 돌려보니 unsigned 연산자를 사용할 수 없다는 것을 알게 되었습니다. unsigned가 없으면 예제처럼 x를 >> 시프트할 때, 산술 시프트가 실행되기 때문에 원하는 값을 얻을 수 없었습니다.

그래서 아래처럼 코드를 고쳐썼습니다.

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

        int temp = (x >> 33+(~(n))) & ((0x1 << n) + (~0));

        return (x << n) | temp;

}
```

./driver.pl 실행 결과

```
Correctness Results     Perf Results
Points  Rating  Errors  Points  Ops     Puzzle
1       1       0       2       4       bitAnd
2       2       0       2       3       getByte
3       3       0       2       6       logicalShift
4       4       0       2       39      bitCount
1       1       0       2       2       isZero
2       2       0       2       2       isEqual
2       2       0       2       8       fitsBits
3       3       0       2       18      isLessOrEqual
3       3       0       2       9       rotateLeft

Score = 39/39 [21/21 Corr + 18/18 Perf] (91 total operators)
b201502085@2018-sp:~/Pracice03/Datalab$
```