시스템 프로그래밍 과제 #1 풀이

0. 문서 설명

시스템 프로그래밍 과제#1 Datalab 의 풀이과정입니다.
전반적인 로직, 그리고 해당 로직을 도출해 낸 경위를 서술하였으며, 필요에 따라 표 혹은 코드를 활용하였습니다.
작성자 : 컴퓨터공학과 201611306 최병규

1. `isZero()`

`!` 연산자는 0에 대해서는 1, 0이 아닌 모든 숫자에 대해서는 0을 리턴해 줍니다.
따라서, `!x` 를 해주면 그만입니다.

2. `bitOr()`

`a | b` 를 드모르간의 법칙에 의하여, `~` 연산자와 `&` 연산자만을 이용해서 바꿀 수 있습니다.
`a | b` = `~(~a & ~b)` 입니다.

3. `bitAnd()`

2번 문제랑 똑같은 방법으로 접근합니다. `a & b = ~(~a | ~b)` 입니다.

4. `minusOne()`

-1의 signed 비트패턴은, 숫자 모두가 1입니다. 따라서, `~0` 해줘서 모든 비트를 1로 만들어주면 됩니다.

5. `negate()`

주어진 숫자의 2의 보수를 구하면 됩니다. 2의 보수를 구하는 방법은, 비트를 반전시킨 후 1을 더하면 되므로,
`~x + 1` 을 리턴해주면 됩니다.

6. `bitXor()`

2, 3번 문제랑 같은 방식으로, 드모르간을 이용하여 식을 만들어주면 됩니다.
`a ^ b = (a & ~b) | (~a | b)` 인 점에 주목합시다.
하지만 우리는 bitwise or 연산자를 쓸 수 없죠? 드모르간 법칙을 사용합시다.
따라서, `a ^ b = ~(~(a & ~b) & ~(~a & b))` 가 됩니다.

7. `isPositive()`

이 문제를 보기 전에, 우리는 정수의 성질에 주목할 필요가 있습니다. 정수를 크게 나누면 3가지로 나눌 수 있습니다.

1. 양수
2. 0
3. 음수

이 문제가 까다로워지는 부분은 바로 0을 처리하는 부분입니다. 0의 sign bit 도 0이지만 우리는 0을 리턴해야 하기 때문이죠.
우선, 음수를 구분하기 위해서, sign bit 만 뽑아와줍시다. 부호비트는 MSB죠? 그렇다면,
`(x >> 63) & 0x1` 을 하면 부호 비트만 뽑아올 수 있습니다! (괄호는 되게 중요합니다 `&` 연산자가 `>>` 연산자보다 우위입니다.)
그 다음, 둘을 반전시키면 되죠? `!` 씁시다.
이렇게 하면, 모든 음수에 대하여 0인 결과를 이끌어낼 수 있습니다.
다음은, 0을 걸러내는 부분에 주목합시다. 생각해 보니깐 우리는 1번 문제에서 이것을 한 적이 있습니다!
`!!x` 를 해주면 0이 아닌 숫자들은 1, 0인 숫자는 0이 리턴됩니다.
이것을 한눈에 표로 나타내보면,

`x`	`!((x >> 63) & 0x1)`	`!!x`
+	1	1
0	1	0
-	0	1

이런 식으로 나와 주는 것이죠! 따라서, 최종 오퍼레이션은, `!((x >> 63) & 0x1) & !!x` 가 됩니다.

8. `getByte()`

바이트 추출 문제군요. 예시를 하나 들어볼께요.
`1111 0101` 에서 `0101` 만 추출해 낼려면, `0000 1111` 과 & 연산을 해주면 됩니다. 이렇게 마스킹을 해서 바이트를 추출해보도록 하겠습니다.
우리의 마스크는 `11111111` , 10진수로 따지면 255 입니다. 왜냐고요? 1바이트 = 8비트니깐요. 우리는 추출할 비트를 맨 오른쪽으로 먼저 시프트 한 후, 마스크를 이용해서 추출할 겁니다.
n번째 바이트를 추출할려면, 8비트 시프트를 n 번 해야 합니다. 총 8n 번을 시프트 해야 하죠. 그렇다면, `n << 3` 을 해주면 8n 의 값이 얻어지므로,

`n >> (n << 3)` 을 해주면, 우리는 하나의 바이트를 맨 오른쪽 바이트로 가져올 수 있습니다!

원하는 위치에 넣었으면 이제 마스크를 해줘야죠? 그래서 최종 코드는,

```
long getByte(long n)
{
  long mask = 255;
  return mask & (x >> (n << 3));
}
```

이렇게 나옵니다!

9. `isNotEqual()`

A 와 B 의 비트 패턴이 같은지, 다른지 판별하는 것은 너무 간단합니다. 그냥 xor 시켜주면 되잖아요? xor 한 결괏값이 0이면 같은거고, 0이 아니면 다른 겁니다.
우리는 이제 "0이 아님" 을 1로 만들어 주는 과정에 주목합시다. `isZero()` 를 반대로 해주면 되겠죠?
따라서 최종 오퍼레이션은 `!!(x ^ y)` 입니다.

10. `evenBits()`

짝수 비트를 켜 주기 위해서 첫 번째 비트를 켜면, 0x01 이라는 숫자가 나오는데, 이 숫자를 2씩 시프트하여 전부 `|` 로 묶어주면 우리가 원하는 결과를 얻을수 있으나...
우리가 쓸 수 있는 최대 연산자 수는 8 개 입니다. 시프트 연산자와 or 연산자를 8번만 써서 64비트를 채우려면, 애초에 기준 숫자를 크게 잡아줘야 될 것입니다.

시프트 연산 4번, bitwise or 연산 4번을 써서, 자릿수 높은 비트에다 1을 넣어주면 될 것 같다는 생각이 들어서, 64비트의 숫자를 균등하게 16비트로 쪼개고, 처음 16비트, 그다음 16비트, 그 다음 16비트, 마지막 16비트 - 이런 식으로 1을 집어넣어 주려는 의도로,

```
long mask = 21845; // 0b0101010101010101
return mask | (mask << 16) | (mask << 32) | (mask << 48);
```

이런 코드를 짰으나...
우리는 상수로 0b11111111(255) 까지 쓸 수 있습니다. 따라서, 우리는 0b01010101(85) 를 마스크로 써야 합니다.
하지만 우리는 8비트가 아니라 16비트의 숫자가 필요합니다. 8비트 패턴을 16비트로 만들어주려면 8비트 시프트 해주고 더해주면 되죠?

```
long mask = 85; // 0b01010101
mask |= (mask << 8);
return mask | (mask << 16) | (mask << 32) | (mask << 48);
```

이런 식으로 시프트 4번, bitwise or 4번. 총 8번의 연산으로 64비트의 짝수 자리 비트에 1을 집어넣었습니다.

11. `reverseBytes()`

8번의 `getByte` 문제의 심화 버전이라는 생각이 들었습니다. 제가 생각한 로직은,

1. 각각의 바이트를 추출한다
2. 역순으로 담아준다

였습니다.

1 바이트 = 8비트 이므로 바이트 추출을 위한 마스크는 255가 됩니다. `getByte` 에서 해준 방식으로, 8개의 바이트들을 추출해서 각각 변수에 담아주었습니다.

```
long mask = 255;
long first_byte = mask & x;
long second_byte = mask & (x >> 8);
// ...
long eighth_byte = mask & (x >> 56);
```

그 다음, 역순으로 담아주는 과정에서는 8번째 바이트부터 차례대로 담아주면 됩니다. 따라서,

```
return eighth_byte | (seventh_byte << 8) | ... | (first_byte <<
56);
```

처음 바이트 추출 때랑 순서 및 시프트 방향 반대로 넣어서 `|` 로 묶어주면 됩니다.

12. `conditional()`

삼항 연산자를 쉽게 풀어 써보면 다음과 같습니다.

```
if(x)
  return y
else return z
```

x 에 따라서 y나 z를 리턴해야 하는 구조이므로, 주 오퍼레이션은,

```
(mask & y) | (~mask & z)
```

가 되야 함을 알 수가 있습니다. 여기서 `mask` 는 0 또는 -1 이 되면 되겠죠. (비트가 전부 0이거나 전부 1)

또한, C언어에서의 boolean 체계에서는 숫자 0이 `false` , 1이 `true` 를 의미합니다. 따라서, x를 0 또는 1인 숫자로 바꿔주어야 합니다. 따라서,
`x = !!x` 를 해 주는 것이 첫 번째 단계입니다.

이제 마스크를 만들면 될 것 같군요! 첫 번째 단계에서 x가 0 또는 1의 값을 갖도록 조작해 놓았습니다. x가 1이면 mask는 -1, 0이면 mask는 0이 되야 하므로, x의 2의 보수를 구해주면 됩니다! 따라서,

```
long conditional(long x, long y, long z)
{
  x = ~(!!x) + 1;
  return (x & y) | (~x & z);
}
```

이런 식으로 단순 `if-else`, 삼항 연산자를 구현하였습니다.

13. `isGreater()`

수학적으로 `x - y > 0 ==> x > y` 인 것을 알고 있으므로, 일단 코드에 적용을 시켜보면,

```
long subtraction_result = x + (~y + 1); // 2의보수!
long sign_subtraction = (subtraction_result >> 63) & 1;
return !sign_subtraction & !!subtraction_result;
```

이런 식으로 틀이 잡힙니다. `!!subtraction_result` 를 해줌으로써 뺄셈의 결과가 0인 경우에도 정상적으로 0을 리턴할 수 있게 해 줍니다.

하지만 이 코드는 완벽하지 않습니다. 왜냐하면, 컴퓨터의 경우 뺄셈이 오버플로우가 날 수 있으므로 x - y > 0 이어도 x < y 인 경우가 존재하기 때문입니다.
그렇다면, 오버플로우가 나는 경우를 찾아서 그 경우만 따로 처리해주어야 합니다. 뺄셈이 오버플로우가 날 수 있는 경우는,
`양수 - 음수` 이거나, `음수 - 양수` 인데, 애초에 두 수의 부호가 다른 경우입니다!

따라서,

1. 두 수의 부호가 같으면 뺄셈으로 비교하면 된다.
2. 두 수의 부호가 다르면 그냥 부호만 보고 결정하면 된다.

라는 결론에 도달할 수 있습니다.

두 수의 부호가 같은지 다른지는, sign bit를 추출해서 xor 시키면 쉽게 구할수 있습니다. 따라서,

```
long isGreater(long x, long y)
{
  long sign_x = (x >> 63) & 1;
  long sign_y = (y >> 63) & 1;
  long is_same_sign = !(x ^ y); // 부호가 같으면 1, 아니면 0을 리턴
  long subtraction_result = x + (~y + 1); // 2의보수!
  long sign_subtraction = (subtraction_result >> 63) & 1;

  return ((is_same_sign & !sign_subraction) &
!!subtraction_result) |(!is_same_sign & !sign_x);
}
```

`return` 문이 살짝 긴데, "부호가 같으면 뺄셈으로 비교하고, 다르면 x의 부호를 보고 판단해라" 라는 문장입니다. 기존 수학적인 아이디어에서 고려해줘야 하는 부분이 많은 문제였던 것 같습니다.

14. `bang()`

마지막 문제답게 난이도도 엄청 높았던 것 같습니다.

우리의 주 목적은 바로 "비트 패턴에 1이 존재하는가" 를 찾아내는 것입니다. 하지만, 어디에 1이 있을지는 모르기 때문에, 우리가 해줘야 하는 액션은 바로 **우리가 알 수 있는 자리에 1을 강제로 집어넣어 주는 것** 입니다!
우리가 명시적으로 비트를 뽑아올 수 있는 자리는 대표적으로 MSB 와 LSB, 두 자리가 있는데, 우선 LSB 에 강제로 1을 주입시키는 오퍼레이션을 시도해보았습니다.

```
x |= x >> 32;
x |= x >> 16;
x |= x >> 8;
x |= x >> 4;
x |= x >> 2;
x |= x >> 1;
```

위와 같이 하면 어떤 위치에 1이 존재하든 LSB 자리로 1을 집어넣을 수 있으나, 이렇게 하면 12번의 연산을 전부 다 쓰게 되어서 실패합니다.
결론은 다른 방법을 찾아야 한다는 건데요, MSB 자리에 1을 주입시키는 방법을 생각해 보았습니다.

MSB 자리에 1을 집어넣기 위해서, "MSB만 0이고 나머지는 다 1인(`LONG_MAX`)" 마스크를 사용해서 x랑 더해주면, 무조건 캐리(carry) 가 발생해서 sign bit 가 1이 되어버립니다! 이것을 코드로 나타내면,

```
long mask = ~(1L << 63);
long masked = x + mask;
```

하지만, 이 방법도 커버하지 못하는 영역의 수가 있습니다. 바로 **음수** 들이죠. 음수들은 애초에 부호 비트가 1이기 때문에 만약에 캐리가 발생하면 부호 비트가 0이 되어버립니다. 생각한 방향과는 전혀 딴판으로 흘러가게 되버리는 것이죠.

그래서 생각한 방법은, `x` 과 `-x` (x의 2의보수) 둘 다 마스크랑 덧셈을 해서, 결과에 or 연산을 해주는 방법입니다. 이 방법이 가능한 이유를 표로 보여드리자면,

수의 종류	MSB	나머지 비트	LONG_MAX 랑 더하기 연산 시
양수	0	어딘가엔 1이 있음	캐리 발생하여 부호 비트 —> 1
음수	1	어딘가엔 1이 있음	캐리 발생하여 부호 비트 —> 0
LONG_MIN	1	0	-1로 변함(모든 비트 1)
0	0	0	LONG_MAX 로 변함(부호 비트 0, 나머지 1)

일반적인 양수 및 음수는, **MSB 혹은 나머지 비트 자리 어딘가엔 1이 반드시 있기 때문에**, `LONG_MAX` 과 덧셈을 해주게 되면 해당 수와, 해당 수의 2의 보수 둘 중 하나는 무조건 부호 비트가 1로 변합니다!

LONG_MIN(최솟값, 비트 패턴은 부호 비트만 1이고 나머지는 0인 수) 의 경우, 2의 보수도 자기 자신(오버플로우 때문) 이고, LONG_MAX 이랑 연산 시 -1이 되어버리므로, 부호 비트가 변하지 않아 상관없습니다.

하지만, 0 같은 경우는 0의 2의 보수를 구해봤자 0입니다!

이렇게 얻은 덧셈 결과를 63번 오른쪽 시프트를 하게 되면, 0의 경우 0이, 나머지의 경우는 -1(모든 자릿수가 전부 1) 이 얻어지는데, 이 결과에다 1을 한번 더 더해주면 우리가 원하는 결과가 나오게 됩니다.

코드로 표현하면,

```
long bang(long x)
{
  long mask = ~(1L << 63);
  long masked = (x + mask) | ((~x + 1) + mask);
  return (masked >> 63) + 1;
}
```

위와 같이 됩니다.
숫자 어딘가에 숨어있는 1을 최고자리 비트로 꺼내옴으로써 논리연산 NOT을 구현하였습니다.