시스템 소프트웨어(059)
HW1- datalab

학번 :
소속 :
이름 : 김태훈

1. bitAnd
드모르간의 법칙을 이용하여 x&y = ~(~x|~y)임을 사용하여 해결하였다.

2. getByte

먼저 추출하고자 하는 자리를 least significant로 만들기 위해 n이 0,1,2,3 일 때 각각 0,8,16,24비트씩 오른쪽으로 옮겨야 된다(16진수 1자리 수는 4비트이기 때문이다.).
따라서 n이 주어지면 (n<<3)비트만큼 x를 오른쪽으로 shift해야 되고 least significant를 추출하기 위해 255(=0b11111111)와 bit and 하여 최종적으로 getByte를 한다,

3. logicalShift
int m = ((1<<31)>>n)<<1은 11111100000....을 만들어내는데, 여기서 1의 개수는 n이다.
x=x>>n으로 arithmetic shift를 한 후, ~m(=00000011111... 0의 개수는 n)과 x를 bit and 하면 arithmetic shift에 의해 생성된 n개의 1이 0과 and되면서 0이되어, logical shift가 된다.

4.

5. bang
int nz = x | (~x + 1); //0이 아니면 -x와의 연산의 sign bit가 1이 된다.
(nz>>31)은 0이 아닌 경우에는 0xFFFFFFFF가 되고 0인 경우 0이 되고 0xFFFFFFFF +1 = 0, 0+1은 1이므로 답은 (nz>>31) +1이다.

6. tmin
2의 보수에서 가장 작은 int 정수는 -2^31 = 1<<31이다.

7. fitBits
수 x를 32-n번 left shift 했다가 32-n번 right shift해서 앞 32-n비트를 0으로 만들어 원래 숫자와 비교하면 된다.
-를 쓸 수 없으므로 32-n = 32 + (~n + 1) = 33+ ~n = shift
x를 shift번 왼쪽으로 옮겼다가 오른쪽으로 옮긴 후 원래 숫자와 bit xor를 하면 같으면 0, 다르면 1이 나온다. 따라서 not을 취하면 각각 1, 0이 나오므로 원하는 결과가 나오게 된다.

8.
나눗셈일 때 x가 음수이면 x+(2^k - 1)을 한 후 나누게 된다.
이것을 뺄셈과 조건문 없이 구현하면 된다,
negative = x>>31을 하면 음수이면 -1(0xFFFFFFFF)이 양수이면 0(0x00000000)이 저장된다.
tmp = x + (negaitve&((1<<n) (~1 + 1))는 negative가 -1이면 (1<<n) + -1가 선택되고, 0이면 0(=negative)가 선택된다.
선택된 tmp를 2^n으로 나누면 된다.

9. negate
2의 보수에서 음수는 양수를 bit not한 후 1을 더하면 된다.

10. isPositive
!(x>>31)에서 x가 음수일 경우 !(0xFFFFFFFF) = 0, 아닐 경우 !(0x00000000) = 1이 저장된다.
따라서 !x^!(x>>31)는 양수일 경우 0^1 = 1, 음수일 경우 0^0 = 0, 0일 경우 1^1이 출력되어 양수 일때만 1이 출력되게 된다.

11. isLessOrEqual
int xs = (x>>31)&1;int ys = (y>>31)&1;
에서 x와 y가 양수이거나 0이면 0이 저장되고, 음수이면 1이 저장된다.
x와 y의부호가 다른지 검사하는 것이 중요한데, 두 수의 부호가 다르다면 x-y가 overflow가 나서 sing bit를 검사할 때 오류가 발생할 수 있기 때문이다.
또한 -x == (~x + 1)이므로 y-x == y +(~x + 1)이고, yMinusx>>31 & 1을 통해 y-x의 sing bit를 추출한다.
x와 y의 부호가 다를 때(x^y == 1) x가 음수이고, y가 양수라면 1을 리턴해야한다. 따라서 condition1 = (xs^ys) & (xs&!ys);
그리고 x와 y의 부호가 같다면(!(sx^sy)) y-x의 sign bit의 not을 리턴하면 된다(condtion2 =!(xs^ys) & !signBitOfyMinusx;)
12. ilog2

13. float_neg
int ret = (0x01<<31) ^ uf;
을 통해 맨 앞 sing bit의 not 값을 만든다.
이제 NaN인지 검사해야한다. NaN은 exp == 0b11111111, frac !=0일 때이므로,
(0xFF<<23) 과 uf을 bit and 하여 uf에서 exp를 추출하고
0b0000 0000 0111 1111 1111 1111 ...과 uf을 bit and 하여 frac을 추출한다,
추출한 exp와 frac으로 NaN인지 검사하고, NaN이면 uf를 아니면 ret을 리턴한다.

14. float_i2f

15. float_twice
13번과 마찬가지로 sign, exp, frac을 추출하고 NaN이면 uf를 리턴한다.
그리고 uf가 0이거나 -0이면 uf 그대로 리턴한다.
float이 denormalized인 경우 너무 작아(0.xxxx... * 2^-126) exp가 1증가되지 않고, frac이 왼쪽 shift되는 경우가 있다. normalized인 경우 frac에 1을 더하며 된다.
즉 denormalized인 경우 (0.xxx *2) *2^-126을 연산하고, normalized인 경우 (1.xxxx) *2^a * 2 =
(1.xxx) * 2^(a+1)을 연산하면 된다.