ICS Homework 1

Cao Shengcao

2.61

A.

Note that the result of `!` is either `0` or `1`. Only when every bit of `~x` is `0`, this expression returns `1`, which means every bit of `x` is `1`.

```
!~x
```

B.

The same as A.

```
!x
```

C.

The binary expression of `0xFF` is `1111 1111`. Here `0xFF` is used as a "mask". For example, an expression `x & 0xFF` can extract the lowest 8 bits (1 byte) of `x`.

```
!(~x & 0xFF)
```

D.

The same idea. Please don't always assume `int` is 32-bit long.

```
!(x & (0xFF << ((sizeof(int) - 1) << 3)))
```

2.62

The difference between arithmetic right shift and logical right shift is about the sign bit. We can test this fact by right shifting `-1`, whose binary expression is `1111...1111`. If the shift is arithmetic, the result would be `1111...1111`. Otherwise it would be `0111...1111`.

```
int int_shifts_are_arithmetic()
{
    return (-1 >> 1) == -1;
}
```

2.65

We can check the oddness of the number of `1`s by "folding" the integer with the xor operator. We can regard an `int a` as an array of 32 bits, with the least significant bit named `a[0]`. Consider `y = x ^ (x >> 16)`, then `y[0]` shows the oddness of `x[16]` and `x[0]`, which means, if exactly one of `x[16]` and `x[0]` is `1`, then `y[0]` would be `1`. `y[1]` `y[2]` ... `y[15]` are similar. Following this idea, we can finally fold the integer into one bit, and this bit shows the oddness of the number of `1`s in the original integer.

```c
int odd_ones(unsigned x)
{
    x ^= x >> 16;
    x ^= x >> 8;
    x ^= x >> 4;
    x ^= x >> 2;
    x ^= x >> 1;
    return x & 1;
}
```