Nicholas Livingstone

CS 341L

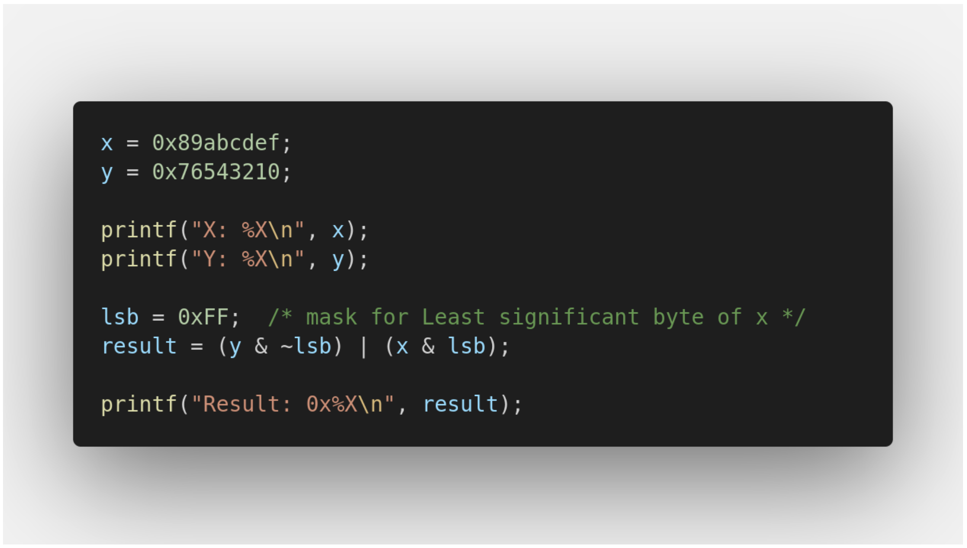
8/28/2020

Homework 1

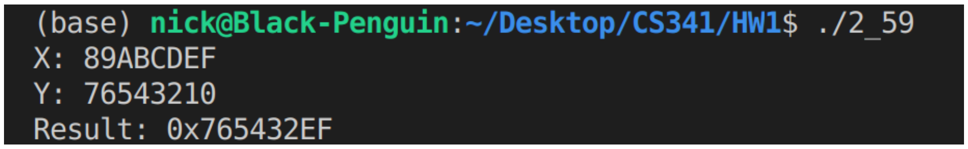
Computer Systems A Programmer's Perspective

2.59

Code and example output



```
x = 0x89abcdef;
y = 0x76543210;

printf("X: %X\n", x);
printf("Y: %X\n", y);

lsb = 0xFF;  /* mask for Least significant byte of x */
result = (y & ~lsb) | (x & lsb);

printf("Result: 0x%X\n", result);
```

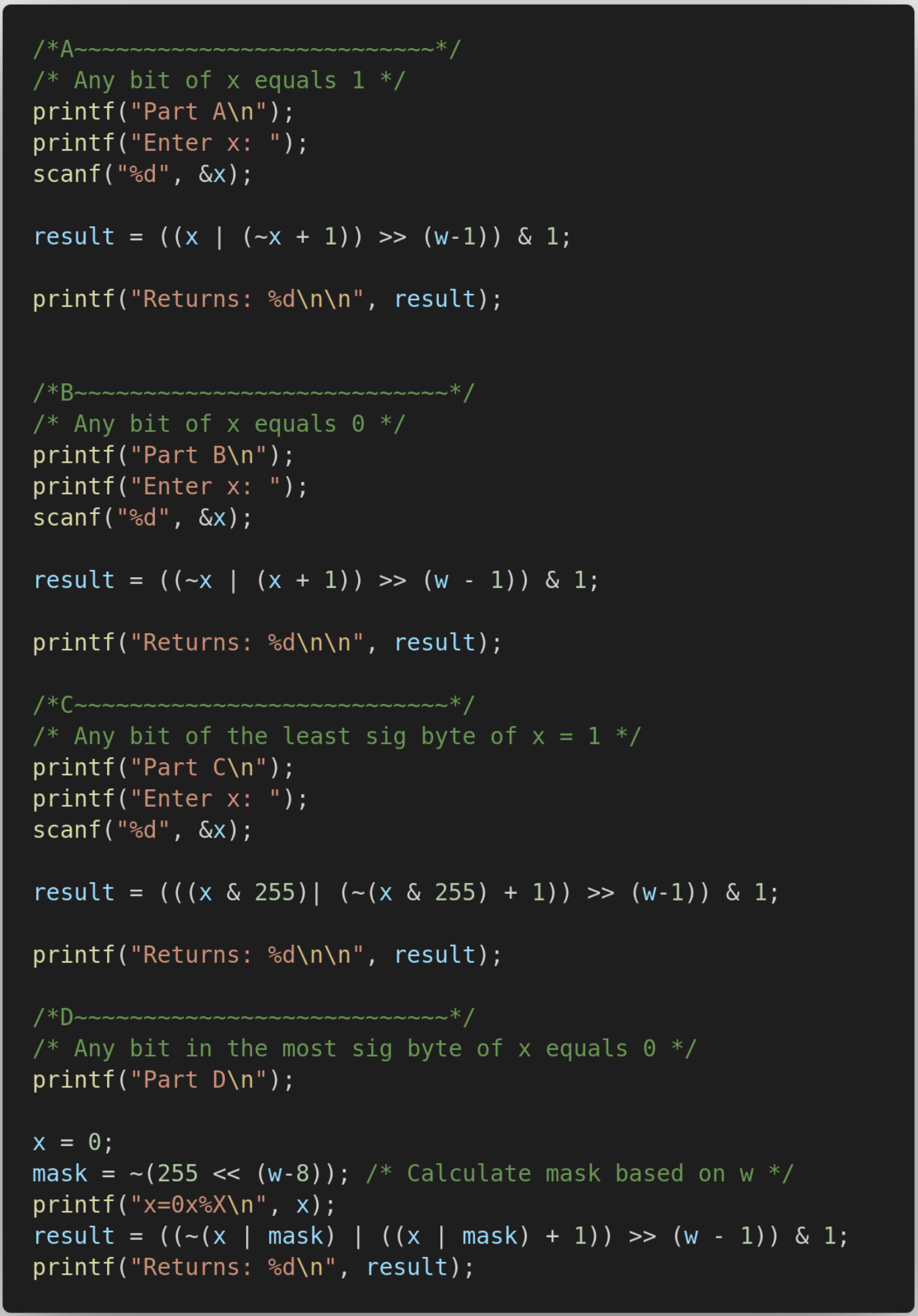


```
(base) nick@Black-Penguin:~/Desktop/CS341/HW1$ ./2_59
X: 89ABCDEF
Y: 76543210
Result: 0x765432EF
```

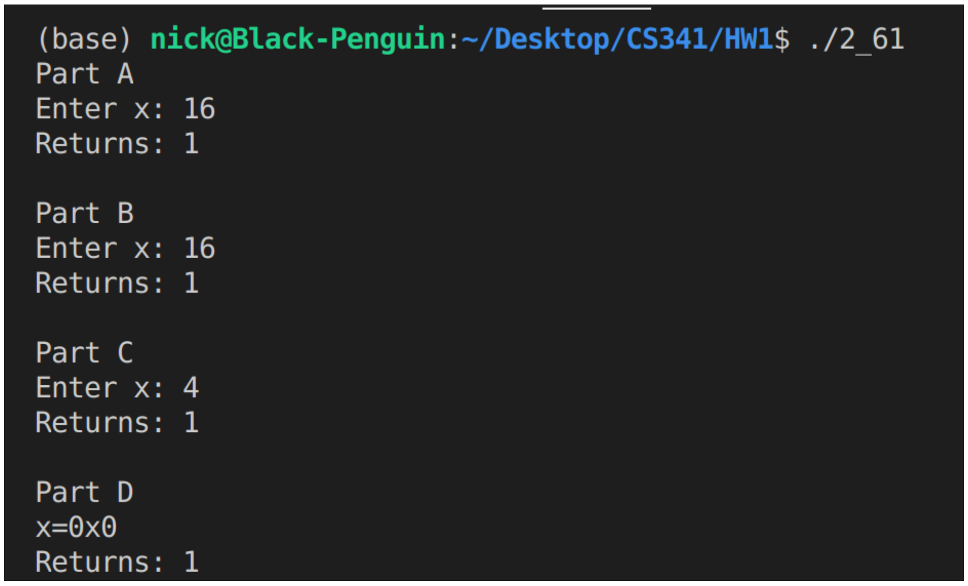
The key to this problem is forming a mask of the least significant byte i.e. 0xFF. By completing the *AND* operation on x using the mask and an *AND* operation on y using the compliment of y, the target bits of y and x can be separated. The bits are then recombined into the expected result using the *OR* operator.

2.61

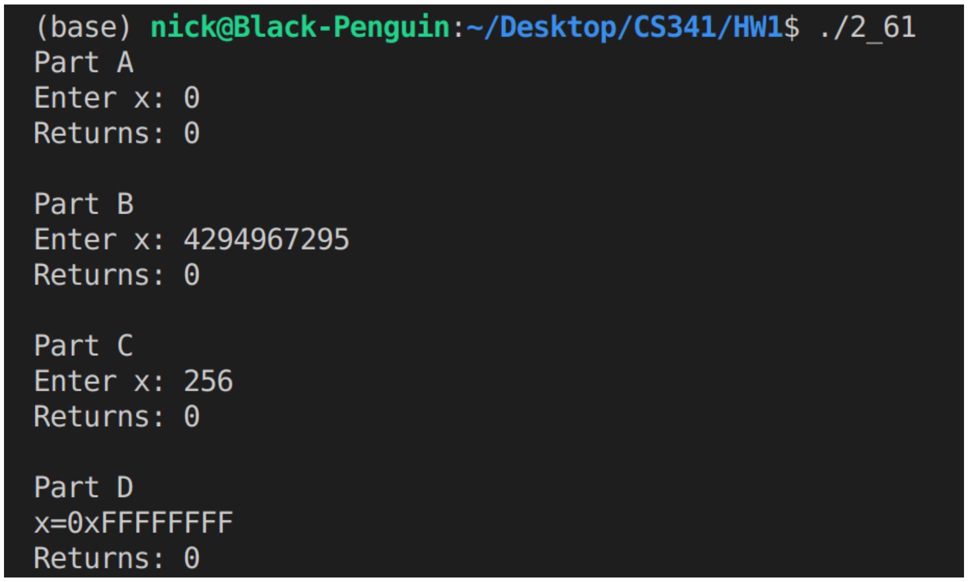
Code and Example Output



```c
/*A~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* Any bit of x equals 1 */
printf("Part A\n");
printf("Enter x: ");
scanf("%d", &x);

result = ((x | (~x + 1)) >> (w-1)) & 1;

printf("Returns: %d\n\n", result);


/*B~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* Any bit of x equals 0 */
printf("Part B\n");
printf("Enter x: ");
scanf("%d", &x);

result = ((~x | (x + 1)) >> (w - 1)) & 1;

printf("Returns: %d\n\n", result);

/*C~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* Any bit of the least sig byte of x = 1 */
printf("Part C\n");
printf("Enter x: ");
scanf("%d", &x);

result = (((x & 255)| (~(x & 255) + 1)) >> (w-1)) & 1;

printf("Returns: %d\n\n", result);

/*D~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* Any bit in the most sig byte of x equals 0 */
printf("Part D\n");

x = 0;
mask = ~(255 << (w-8)); /* Calculate mask based on w */
printf("x=0x%X\n", x);
result = ((~(x | mask) | ((x | mask) + 1)) >> (w - 1)) & 1;
printf("Returns: %d\n", result);
```



```
(base) nick@Black-Penguin:~/Desktop/CS341/HW1$ ./2_61
Part A
Enter x: 16
Returns: 1

Part B
Enter x: 16
Returns: 1

Part C
Enter x: 4
Returns: 1

Part D
x=0x0
Returns: 1
```

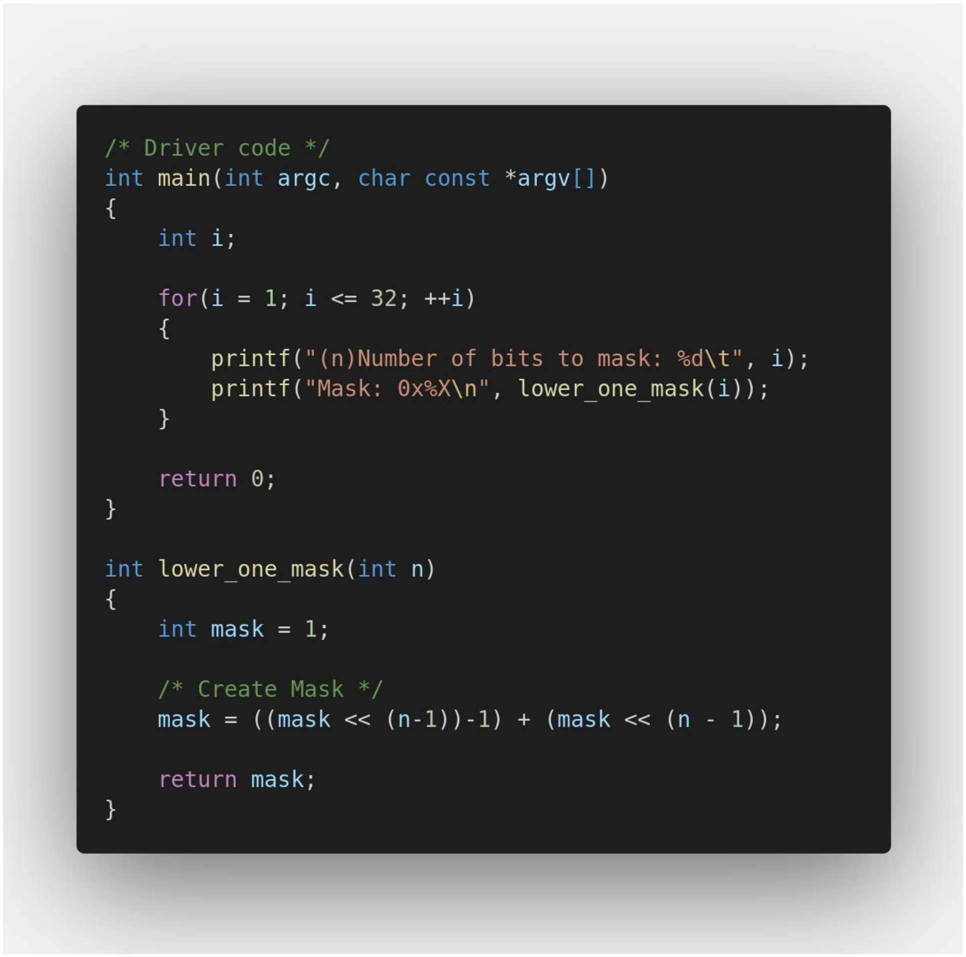


```
(base) nick@Black-Penguin:~/Desktop/CS341/HW1$ ./2_61
Part A
Enter x: 0
Returns: 0

Part B
Enter x: 4294967295
Returns: 0

Part C
Enter x: 256
Returns: 0

Part D
x=0xFFFFFFFF
Returns: 0
```

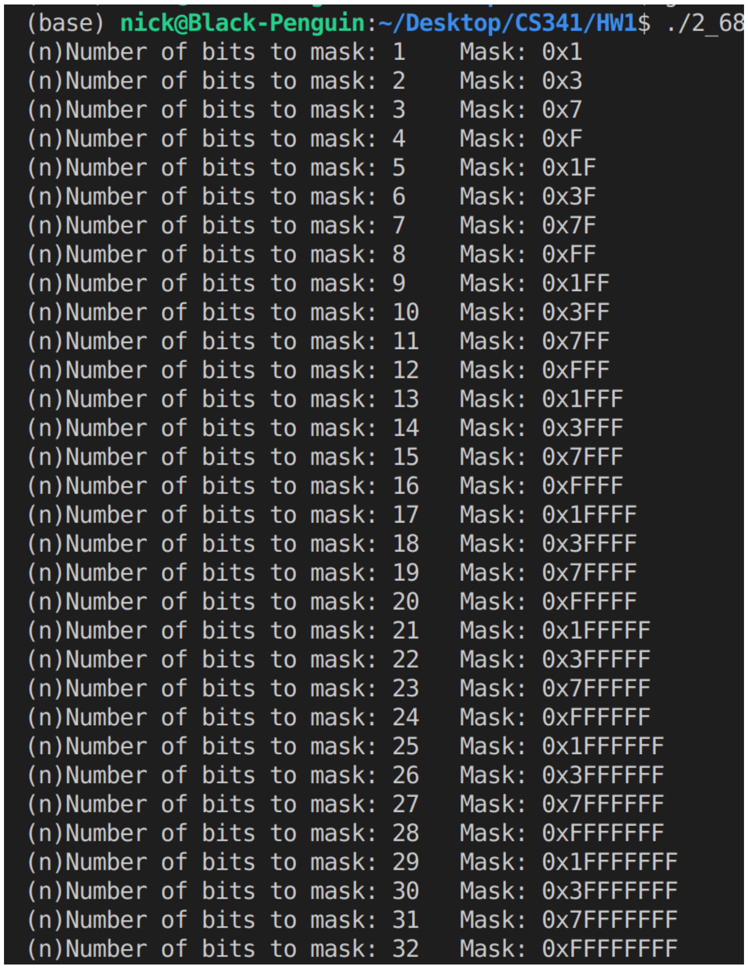
For a number to have any bit set, it must be non-zero. An efficient way of checking this is based on the most significant bit i.e. the sign bit in the case of two's compliment. For negative numbers, this bit will be set, for non-negative numbers we can negate the number using (~) and adding one based on the two's compliment formula. This is how the formula for part is developed, this allows us to make an and comparison with the right shifted sign bit with 1 to produce the desired output. For part B, the equation is modified to check the compliment of the number using the bitwise not (~). For part C, the same formula from A can be used but X must be masked with the first byte by completing the and operation with 255. Part D can use modified version of the equation from B. A mask of the most significant byte is first formed, and the number is and-ed with the mask to set all of the bits below the most significant byte to 1. This ensures that the expression only considers the top byte when evaluating if there is a 0.

2.68

Code and example output



```c
/* Driver code */
int main(int argc, char const *argv[])
{
    int i;

    for(i = 1; i <= 32; ++i)
    {
        printf("(n)Number of bits to mask: %d\t", i);
        printf("Mask: 0x%X\n", lower_one_mask(i));
    }

    return 0;
}

int lower_one_mask(int n)
{
    int mask = 1;

    /* Create Mask */
    mask = ((mask << (n-1))-1) + (mask << (n - 1));

    return mask;
}
```

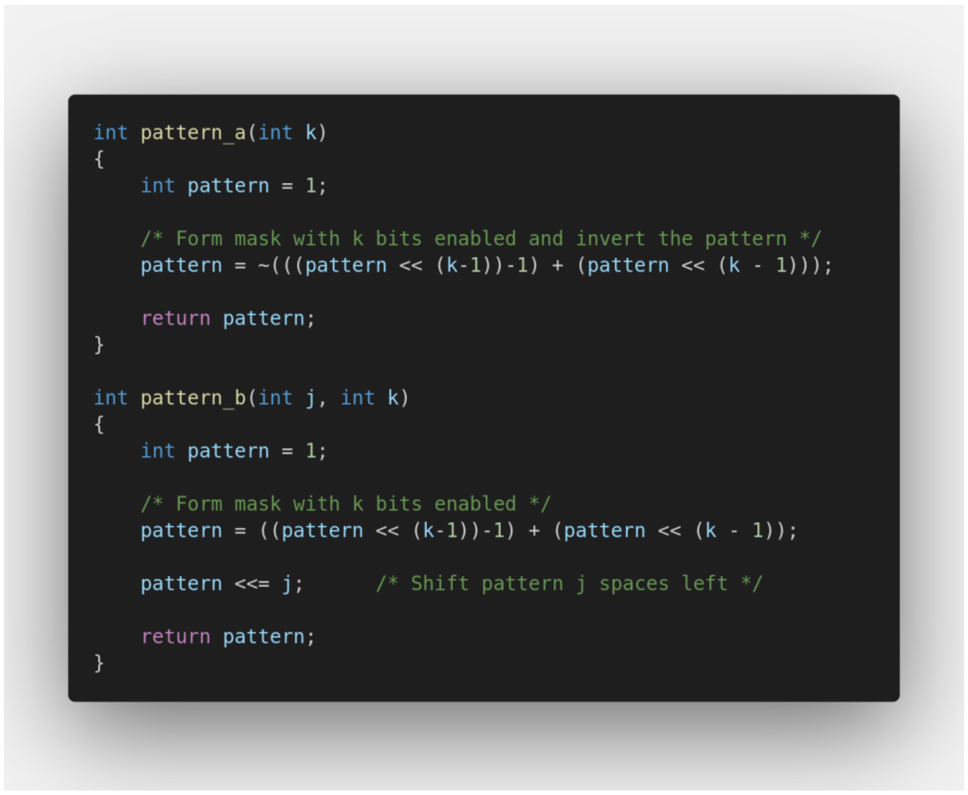


```
(base) nick@Black-Penguin:~/Desktop/CS341/HW1$ ./2_68
(n)Number of bits to mask: 1    Mask: 0x1
(n)Number of bits to mask: 2    Mask: 0x3
(n)Number of bits to mask: 3    Mask: 0x7
(n)Number of bits to mask: 4    Mask: 0xF
(n)Number of bits to mask: 5    Mask: 0x1F
(n)Number of bits to mask: 6    Mask: 0x3F
(n)Number of bits to mask: 7    Mask: 0x7F
(n)Number of bits to mask: 8    Mask: 0xFF
(n)Number of bits to mask: 9    Mask: 0x1FF
(n)Number of bits to mask: 10   Mask: 0x3FF
(n)Number of bits to mask: 11   Mask: 0x7FF
(n)Number of bits to mask: 12   Mask: 0xFFF
(n)Number of bits to mask: 13   Mask: 0x1FFF
(n)Number of bits to mask: 14   Mask: 0x3FFF
(n)Number of bits to mask: 15   Mask: 0x7FFF
(n)Number of bits to mask: 16   Mask: 0xFFFF
(n)Number of bits to mask: 17   Mask: 0x1FFFF
(n)Number of bits to mask: 18   Mask: 0x3FFFF
(n)Number of bits to mask: 19   Mask: 0x7FFFF
(n)Number of bits to mask: 20   Mask: 0xFFFFF
(n)Number of bits to mask: 21   Mask: 0x1FFFFF
(n)Number of bits to mask: 22   Mask: 0x3FFFFF
(n)Number of bits to mask: 23   Mask: 0x7FFFFF
(n)Number of bits to mask: 24   Mask: 0xFFFFFF
(n)Number of bits to mask: 25   Mask: 0x1FFFFFF
(n)Number of bits to mask: 26   Mask: 0x3FFFFFF
(n)Number of bits to mask: 27   Mask: 0x7FFFFFF
(n)Number of bits to mask: 28   Mask: 0xFFFFFFF
(n)Number of bits to mask: 29   Mask: 0x1FFFFFFF
(n)Number of bits to mask: 30   Mask: 0x3FFFFFFF
(n)Number of bits to mask: 31   Mask: 0x7FFFFFFF
(n)Number of bits to mask: 32   Mask: 0xFFFFFFFF
```

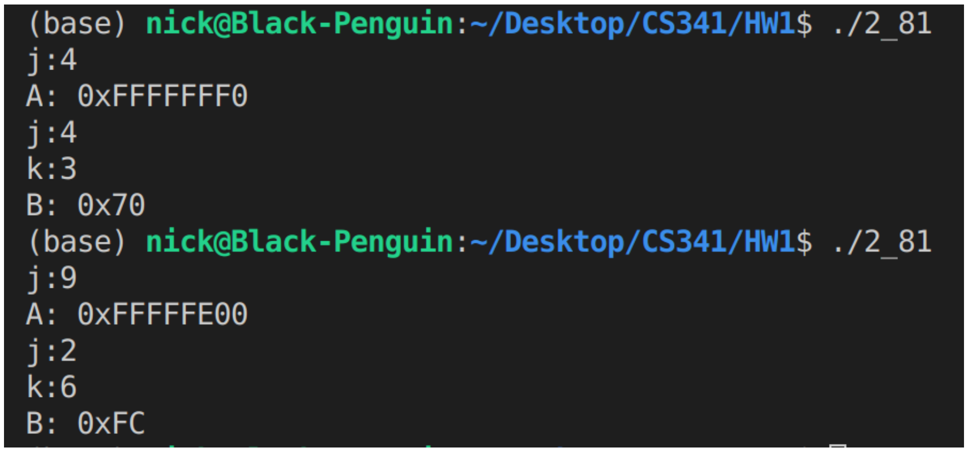
The mask can be formed using bit shifts and addition. The addition and the two terms of the expression are necessary to ensure that a mask of *w* can be produced as expected. First all bits before the mask length requested are set by bit shifting left n-1 spaces and subtracting one. The final bit is added on. If this approach was not used and the mask was shifted n spaces and then 1 was subtracted, when a 32-bit mask was called for it would have resulted in a mask of 0x0.

2.81

Code and example output



```c
int pattern_a(int k)
{
    int pattern = 1;

    /* Form mask with k bits enabled and invert the pattern */
    pattern = ~(((pattern << (k-1))-1) + (pattern << (k - 1)));

    return pattern;
}

int pattern_b(int j, int k)
{
    int pattern = 1;

    /* Form mask with k bits enabled */
    pattern = ((pattern << (k-1))-1) + (pattern << (k - 1));

    pattern <<= j;      /* Shift pattern j spaces left */

    return pattern;
}
```



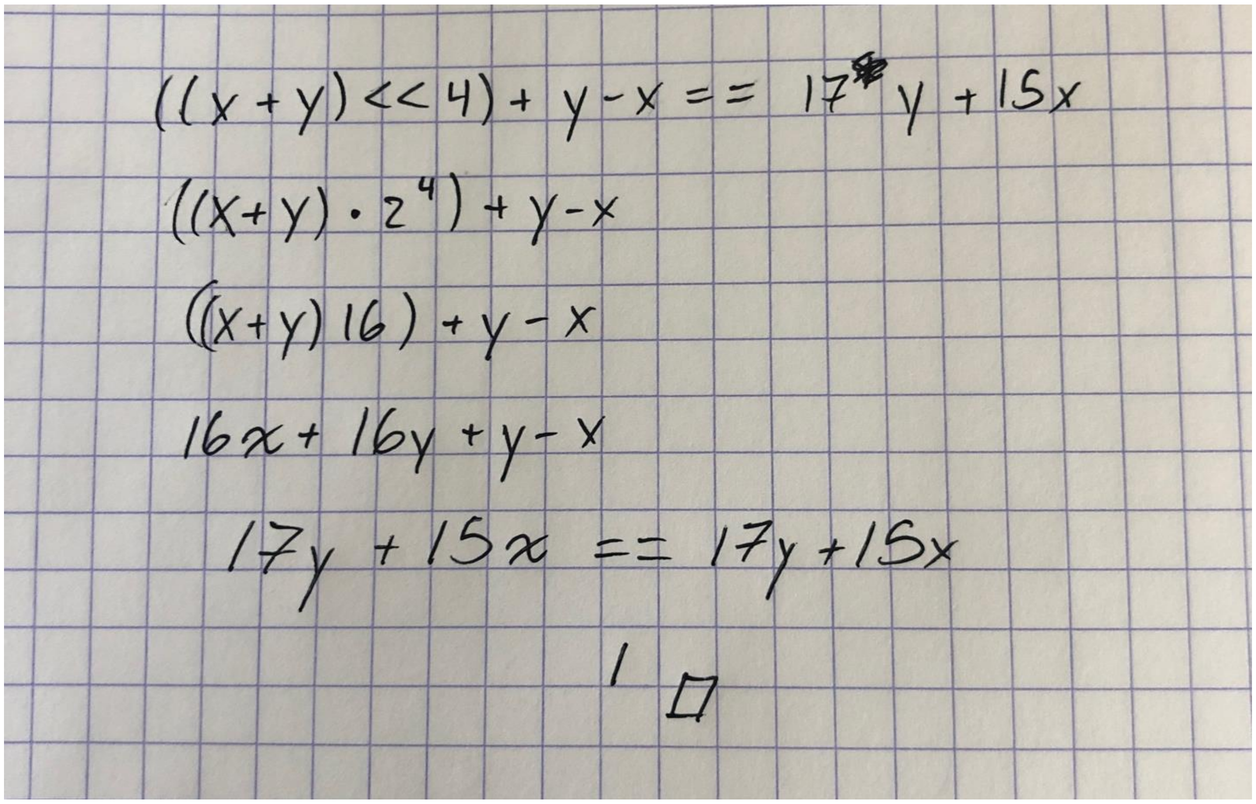
```
(base) nick@Black-Penguin:~/Desktop/CS341/HW1$ ./2_81
j:4
A: 0xFFFFFFF0
j:4
k:3
B: 0x70
(base) nick@Black-Penguin:~/Desktop/CS341/HW1$ ./2_81
j:9
A: 0xFFFFFE00
j:2
k:6
B: 0xFC
```

Implementing the mask expression from problem 2.68 allows this problem to be solved. For pattern A, a set mask of width k is formed and then the compliment of the mask is produced using the NOT operator. These steps form pattern A. Pattern B is formed in a similar manner, but after the mask is formed it is shifted j spaces left.

2.82

A. Expression A *almost* always yields 1. When comparing x and y, if x < y, then the greater than comparison of their relative additive inverses will always equal true as well e.g. 5 < 10 == -5 > -10. This holds true even when the numbers x and y are equal as both expressions will always equate to false. However, by letting x = *int_min* and y = 0, the expression could result in undefined behavior. As depending on the machine, *int_min* might not be representable when negating it.

B. Expression B always yields one. This can be proven by looking at the mathematical implications of the left shift that occurs on the left side of the expression. Any single left shift of a number *n* is equivalent to a multiplication of 2 to *n*, x shifts are the equivalent of 2^x times *n*. Completing a substitution in the expression results in the two sides being algebraically equal as shown below.



$$((x+y)<<4)+y-x == 17*y+15x$$

$$((x+y)\cdot 2^4)+y-x$$

$$((x+y)16)+y-x$$

$$16x+16y+y-x$$

$$17y+15x == 17y+15x$$

$$1 \quad \square$$

C. Expression c always yields 1. Assuming Two's compliment representation, the following substitutions will result in algebraic equivalency of the two sides of the expression.

[illegible] $\sim x + \sim y + 1 == \sim(x+y)$

$(-x-1) + (-y-1) + 1 == -(x+y) - 1$

$-x - y - 1 \cancel{-1+1}$

$-x - y - 1 == -x + y - 1$

$1 == 1 \quad \square$

D. Expression D always yields 1. Because all variables are casted to unsigned, the sides are equivalent. By ignoring the casts, we get the following:

$(Ux - Uy) == -(uns)(y-x)$

$(x - y) == -(y - x)$

$x - y == x - y \quad \square$

E. Expression E always yields 1. This is because the left and right shift operations ensure the first two least significant bits of x are set to zero. If the number had the first two bits set, the number is subtracted a value <= 3, resulting in the left side being less than the right i.e. x-3 < x. Otherwise if the bits were not set, the number x is not changed and results in the sides being equal i.e. x = x.